681.142.2
К-98

Ч. КЭПС, Р. СТАФФОРД

VAX: ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА И АРХИТЕКТУРА

«Радио и связь»

BYTE, WORD, and LONG INSTRUCTIONS

TWO and THREE OPERAND GROUP

Descriptions (x equals B, W, or L)

ADDx2	add,sum	sum ← sum + add	ADD
ADDx3	addl,add2,sum	sum ← addl + add2	ADD
SUBx2	sub,dif	dif ← dif - sub	SUBtract
SUBx3	sub,min,dif	dif ← min - sub	SUBtract
MULx2	mulr,prd	prd ← prd * mulr	MULtiply
MULx3	mulr,muld,prd	prd ← muld * mulr	MULtiply
DIVx2	divd,quo	quo ← quo / divd	DIVide
DIVx3	divr,divd,quo	quo ← divd / divr	DIVide
BICx2	mask,dst	dst ← -dst AND mask	Bit Clear
BICx3	mask,src,dst	dst ← -src AND mask	Bit Clear
BISx2	mask,dst	dst ← dst OR mask	Bit Set (OR)
BISx3	mask,src,dst	dst ← src OR mask	Bit Set (OR)
XORx2	mask,dst	dst ← dst EXOR mask	eXclusive OR
XORx3	mask,src,dst	dst ← src EXOR mask	eXclusive OR

Operation Codes

	Byte-2	Byte-3	Word-2	Word-3	Long-2	Long-3	N	Z	V	C
ADD	ADDB2	ADDB3	ADDW2	ADDW3	ADDL2	ADDL3	*	*	*	*
SUB	SUBB2	SUBB3	SUBW2	SUBW3	SUBL2	SUBL3	*	*	*	*
MUL	MULB2	MULB3	MULW2	MULW3	MULL2	MULL3	*	*	*	0
DIV	DIVB2	DIVB3	DIVW2	DIVW3	DIVL2	DIVL3	*	*	0	-
BIC	BICB2	BICB3	BICW2	BICW3	BICL2	BISL3	*	*	0	-
BIS	BISB2	BISB3	BISW2	BISW3	BISL2	BISL3	*	*	0	-
XOR	XORB2	XORB3	XORW2	XORW3	XORL2	XORL3	*	*	0	-

TWO OPERAND GROUP

Descriptions (x equals B, W, or L)

MOVx	src,dst	dst ← src	MOVe
MNEGx	src,dst	dst ← -src	Move NEGative
MCOMx	src,dst	dst ← -src	Move COMplement
MOVAx	src,dst	dst ← ADR(src)	MOVe Address
BITx	mask,src	tmp ← -src AND mask	Bit Test (AND)
CMPx	src1,src2	tmp ← src1 - src2	CoMPare
MOVZBx	src,dst	dst ← ZEROS ! src	Zero extend Byte
MOVZWx	src,dst	dst ← ZEROS ! src	Zero extend Word

Operation Codes

	Byte	Word	Long	N	Z	V	C	
MOVx	MOVB	MOVW	MOVL	*	*	0	-	
MNEGx	MNEGB	MNEGW	MNEGL	*	*	*	*	* reflects result
MCOMx	MCOMB	MCOMW	MCOML	*	*	0	-	
MOVAx	MOVAB	MOVAW	MOVAL	*	*	0	-	- unchanged
BITx	BITB	BITW	BITL	*	*	0	-	
CMPx	CMPB	CMPW	CMPL	*	*	0	*	0 cleared to zero
MOVZBx		MOVZBW	MOVZBL	0	*	0	-	
MOVZWx			MOVZWL	0	*	0	-	1 set to one

SINGLE OPERAND GROUP

Descriptions (x equals B, W, or L)

INCx	dst	dst ← dst + 1	INCrement
DECx	dst	dst ← dst - 1	DECrement
CLRx	dst	dst ← 0	CLeaR
TSTx	src	tmp ← src	TeST (set N, Z, V, C)
PUSHAx	src	MOVAx src,-(SP)	PUSH Address on stack

Operation Codes

	Byte	Word	Long	N	Z	V	C
INCx	INCB	INCW	INCL	*	*	*	*
DECx	DECB	DECW	DECL	*	*	*	*
CLRx	CLRB	CLRW	CLRL	0	1	0	-
TSTx	TSTB	TSTW	TSTL	*	*	0	0
PUSHAx	PUSHAB	PUSHAW	PUSHAL	*	*	0	-

ADD, COMPARE, and BRANCH INSTRUCTIONS (N=Z=V=*,C=-)

ACBB		index ← index + add, branch
ACBW	limit,add,index,disp	if add GEQ 0 and index LEQ limit
ACBL		or add LSS 0 and index GEQ limit

FLOATING POINT INSTRUCTIONS

FLOATING POINT FORMATS

	F_flt	D_flt	G_flt	H_flt
exponent bits	8	8	11	15
fraction bits	23	55	52	112
total bits	32	64	64	128

TWO and THREE OPERAND GROUP

Descriptions (x equals F, D, G, or H)

ADDx2	add,sum	sum ← sum + add	ADD
ADDx3	addl,add2,sum	sum ← addl + add2	ADD
SUBx2	sub,dif	dif ← dif - sub	SUBtract
SUBx3	sub,min,dif	dif ← min - sub	SUBtract
MULx2	mulr,prd	prd ← prd * mulr	MULtiply
MULx3	mulr,muld,prd	prd ← muld * mulr	MULtiply
DIVx2	divd,quo	quo ← quo / divd	DIVide
DIVx3	divr,divd,quo	quo ← divd / divr	DIVide

Operation Codes

	F_flt	D_flt	G_flt	H_flt	N	Z	V	C
ADDx2	ADDF2	ADDD2	ADDG2	ADDH2	*	*	*	0
ADDx3	ADDF3	ADDD3	ADDG3	ADDH3	*	*	*	0
SUBx2	SUBF2	SUBD2	SUBG2	SUBH2	*	*	*	0
SUBx3	SUBF3	SUBD3	SUBG3	SUBH3	*	*	*	0
MULx2	MULF2	MULD2	MULG2	MULH2	*	*	*	0
MULx3	MULF3	MULD3	MULG3	MULH3	*	*	*	0
DIVx2	DIVF2	DIVD2	DIVG2	DIVH2	*	*	*	0
DIVx3	DIVF3	DIVD3	DIVG3	DIVH3	*	*	*	0

TWO OPERAND INSTRUCTIONS

Descriptions (x equals F, D, G, or H)

MOVx	src,dst	dst ← src	MOVe
MOVAx	src,dst	dst ← adr(src)	MOVe Address
MNEGx	src,dst	dst ← -src	Move NEGative
CMPx	src1,src2	tmp ← src2 - src1	CoMPare

Operation Codes

	F_flt	D_flt	G_flt	H_flt	N	Z	V	C
MOVx	MOVF	MOVD	MOVG	MOVH	*	*	0	-
MOVAx	MOVAF	MOVAD	MOVAG	MOVAH	*	*	0	-
MNEGx	MNEGF	MNEGD	MNEGG	MNEGH	*	*	0	0
CMPx	CMPF	CMPD	CMPG	CMPH	*	*	0	0

SINGLE OPERAND INSTRUCTIONS (x equals F, D, G, or H)

Descriptions

CLRx	dst	dst ← 0	CLeaR
TSTx	src	tmp ← src	TeST (set N, Z, V, C)
PUSHAx	src	MOVAx src,-(SP)	PUSH Address on stack

Operation Codes

	F_flt	D_flt	G_flt	H_flt	N	Z	V	C
CLRx	CLRF	CLRD	CLRG	CLRH	0	1	0	-
TSTx	TSTF	TSTD	TSTG	TSTH	*	*	0	0
PUSHAx	PUSHAF	PUSHAD	PUSHAG	PUSHAH	*	*	0	-

ADD, COMPARE, and BRANCH INSTRUCTIONS (N=Z=V=*, C is unchanged)

ACBF		index ← index + add, branch
ACBD	limit,add,index,disp	if add GEQ 0 and index LEQ limit
ACBG		or add LSS 0 and index GEQ limit
ACBH		

OTHER INSTRUCTIONS (x equals F, D, G, or H)

EMODx	mulr,add,index,disp	Extend MODulus (multiply) (see Chapter 12.6)
POLYx	arg,degree,tbladr	evaluate POLYnomial (see Chapter 12.6)

BRANCH, JUMP, CALL, and CASE INSTRUCTIONS

(Unless noted otherwise, operand is one byte displacement)

SIGNED and UNSIGNED BRANCH INSTRUCTIONS

Signed Branch Instructions	Unsigned Branch Instructions	
	BEQLU	Branch if EQUal
BEQL	BNEQU	Branch if Not EQual
BNEQ	BLSSU	Branch if LeSS
BLSS	BGTRU	Branch if Greater
BGTR	BLEQU	Br if Less or EQual
BLEQ	BGEQU	Branch if Greater or EQual
BGEQ		

CONDITION CODE BRANCH INSTRUCTIONS

BNEQ	Branch if Z clear (result Not Equal to zero)
BEQL	Branch if Z Set (result EQuaL to zero)
BCC	Branch if C Clear (no unsigned overflow)
BCS	Branch if C Set (signed overflow)
BVC	Branch if V Clear (no signed overflow)
BVS	Branch if V Set (signed overflow)
BGEQ	Branch if N clear (result is not negative)
BLSS	Branch if N set (result is negative)

UNCONDITIONAL BRANCHES and JUMPS

BRB	disp	BRanch with Byte displacement
BRW	disp	BRanch with Word displacement
JMP	adr	JuMP (general operand specifier)

SUBROUTINE CALLS and RETURN

BSBB	disp	Branch to SuBroutine with Byte displacement
BSBW	disp	Branch to SuBroutine with Word displacement
JSB	adr	Jump to SuBroutine (general operand specifier)
RSB		Return from SuBroutine (no operand)

PROCEDURE CALLS and RETURN (See Chapter 8)

CALLG	arglist,dst	CALL procedure with General arg list
CALLS	numarg,dst	CALL procedure with arguments on Stack
RET		RETurn from procedure

ADD, COMPARE, and BRANCH INSTRUCTIONS

(N=Z=V=*, C=-)
(disp is a word displacement)

ACBx	limit,add,index,disp	index ← index + add, branch if add GEQ 0 and index LEQ limit or add LSS 0 and index GEQ limit

Byte	Word	Long	F_flt	D_flt	G_flt	H_flt
ACBB	ACBW	ACBL	ACBF	ACBD	ACDG	ACDH

ADD or SUBTRACT ONE and BRANCH INSTRUCTIONS

(disp is a byte displacement, lim and index are longwords)

AOBLSS	lim,index,disp	index = index + 1, BR if index LSS lim
AOBLEQ	lim,index,disp	index = index + 1, BR if index LEQ lim
SOBGTR	index,disp	index = index - 1, BR if index GTR 0
SOBGEQ	index,disp	index = index - 1, BR if index GEQ 0

BRANCH ON LOW BIT (scr is a longword)

BLBC	scr,disp	branch if low bit (bit 0) clear
BLBS	scr,disp	branch if low bit (bit 0) set

CASE INSTRUCTIONS (for multiway branching)

CASEB	str,index,lim,disp[0],	tmp ← index - str	
CASEW	disp[1],	if tmp LEQU lim, BR disp[tmp]	
CASEL	disp[lim]	else no branch	

BIT STRING BRANCH INSTRUCTIONS - see Bit String Instructions

LONGWORD (but not byte or word) INSTRUCTIONS

ADWC	add,sum	sum ← sum + add + C	ADd With Carry
SBWC	sub,dif	dif ← dif - sub - C	SuBtract With Carry
ASHL	cnt,src,dst	dst ← src * 2**cnt	Arith. SHift Long
ROTL	cnt,scr,dst	dst ← ROTATE(src)	ROTate Long cnt bits
PUSHL	src	-(SP) ← src	PUSH Long
INDEX	sub,low,high,size,indexin,indexout		see COMPUTE INDEX

QUADWORD AND OCTAWORD INSTRUCTIONS

ASHQ	cnt,src,dst	dst ← src * 2**cnt	Arithmetic SHift
CLRQ	dst	dst ← 0	CLeaR
CLRO	dst	dst ← 0	CLeaR
MOVQ	src,dst	dst ← src	MOVe
MOVO	src,dst	dst ← src	MOVe
MOVAQ	src,dst	dst ← ADR(src)	MOVe Address
MOVAO	src,dst	dst ← ADR(src)	MOVe Address
PUSHAQ	src	-(SP) ← ADR(src)	PUSH Address
PUSHAO	src	-(SP) ← ADR(src)	PUSH Address
EMUL	mulr,muld,add,prod		Extended MULtiply (see Ch 6.3)
EDIV	divr,divd,quo,rem		Extended DIVide (see Ch 6.3)

COMPUTE INDEX

INDEX sub,low,high,size,indexin,indexout (see Chapter 7.5)

indexout ← (indexin + sub*size)
if indexout LSS low or indexout GTR high, trap

SIGNED CONVERT INSTRUCTIONS excluding Packed (N=Z=V=*, C=0)

Description

CVTxy src,dst dst ← src ConVerT scr (type x)

Instructions

from \ to	Byte	Word	Long	F_flt	D_flt	G_flt	H_flt
Byte		CVTBW	CVTBL	CVTBF	CVTBD	CVTBG	CVTBH
Word	CVTWB		CVTWL	CVTWF	CVTWD	CVTWG	CVTWH
Long	CVTLB	CVTLW		CVTLF	CVTLD	CVTLG	CVTLH
F-flt	CVTFB	CVTFW	CVTFL		CVTFD	CVTFG	CVTFH
D_flt	CVTDB	CVTDW	CVTDL	CVTDF			CVTDH
G_flt	CVTGB	CVTGW	CVTGL	CVTGF			CVTGH
H_flt	CVTHB	CVTHW	CVTHL	CVTHF	CVTHD	CVTHG	

PACKED CONVERT INSTRUCTIONS (N=Z=V=*, C=0)

from \ to	Packed	Long	Leading	Trailing
Packed		CVTPL	CVTPS	CVTPT
Long	CVTLP			
Leading	CVTSP			
Trailing	CVTTP			

OTHER CONVERSIONS

UNSIGNED CONVERSIONS
(MOVe Zero extended)

from \ to	Word	Long
Byte	MOVZBW	MOVZBL
Word		MOVZWL

FLOATING to LONGWORD
ROUNDED CONVERSIONS

from \ to	Long
F_flt	CVTRFL
D_flt	CVTRDL
G_flt	CVTRGL
H_flt	CVTRHL

****

****

VAX

assembly language and architecture

Charles Kapps
Temple University

Robert L. Stafford
Temple University

PRINDLE, WEBER & SCHMIDT
Boston

v 17 2011

Ч·КЭПС, Р·СТАФФОРД

VAX: ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА И АРХИТЕКТУРА

Перевод с английского
И. В. Сперанской, В. М. Северьянова

Под редакцией
А. С. Богданова

списано

Москва
«Радио и связь»
1991

ББК 32.973
К98
УДК 681.3.068

Редакция переводной литературы

Ч. Кэпс, Р. Стаффорд

К98 VAX: Программирование на языке ассемблера и архитектура: Пер. с англ. М.: Радио и связь, 1991. – 416 с.; ил.

ISBN 5-256-00706-8.

В книге американских авторов изучение особенностей архитектуры ЭВМ семейства VAX тесно переплетаются с рассмотрением программирования на языке ассемблера для ЭВМ этого семейства. Большое внимание уделяется разработке программ в среде операционной системы VAX/MS, системному уровню организации ввода-вывода, взаимодействию с языками высокого уровня и функциям операционной системы. Изложение материала сопровождается примерами конкретных программ, контрольными вопросами и задачами.

Для программистов.

К $\frac{2404000000\text{-}116}{046\,(01)\text{-}91}$ 139-90

ББК 32.973

ISBN 5-256-00706-8 (рус.)
IS2N 0-87150-837-0 (англ.)

© 1985 by PWS Publishers

© Перевод на русский язык И. В. Сперанской, В. М. Северьянова, примечания А. С. Богданова, 1991

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Изменение технологии обработки информации в немалой степени определяется уровнем развития вычислительных машин универсального назначения – массовых по выпуску, успешно используемых для различных приложений и доступных для многих пользователей. На каждом этапе развития вычислительной техники можно выделить весьма популярные в свое время и достаточно широко распространенные семейства ЭВМ универсального назначения такие, как наверняка известные читателю ЭВМ серии IBM 360/370 или мини-ЭВМ серии PDP-11. Архитектура (организация ЭВМ с точки зрения программиста) этих двух семейств ЭВМ, являющихся представителями ЭВМ разных классов – "большие" и мини-ЭВМ, стала поистине классическим примером, на их основе изучают организацию вычислительных машин и систем, учат программировать на языках ассемблера. В один ряд с названными семействами ЭВМ несомненно можно поставить и семейство ЭВМ VAX, разработанное как и семейство PDP-11 американской фирмой Digital Equipment Corporation (DEC).

ЭВМ семейства VAX относятся к классу супермини-ЭВМ. Это 32-разрядные ЭВМ, обеспечивающие поддержку виртуальной памяти со страничной организацией. Именно расширенное виртуальное адресное пространство – 4 Гбайта, которое потенциально может использоваться программами, является наиболее привлекательной особенностью ЭВМ семейства VAX, что и подчеркнуто в самом названии VAX – Virtual Address eXtension. Как предмет изучения, архитектура VAX интересна тем, что является логическим расширением архитектуры PDP-11, образцом блестящего последовательного подхода при создании новой вычислительной техники. Сохраняя преемственность с архитектурой PDP-11 и опираясь на ее основные решения, архитектура VAX не просто новая проекция старой архитектуры, ее чисто шаблонное повторение. Включение в архитектуру новых типов данных и режимов адресации, используемых с данными любого типа, расширение набора инструкций и другие решения, все это позволяет достичь в архитектуре VAX новых качественных изменений – "расширение" старой архитектуры становится многомерным, образно говоря, происходит как бы переход из плоскости в объем. Поскольку управление виртуальной памятью поддерживается единым комплексом технических и программных средств, рассматривая архитектуру VAX, также нельзя не заметить сказавшегося на ней влияния разработки программных средств, а именно операционной системы для ЭВМ с архитектурой VAX, получившей название VAX/VMS (VMS – система виртуальной памяти), создание которой велось одновременно с разработкой аппаратуры.

Семейство ЭВМ VAX открывает серия ЭВМ VAX-11, первая модель этой серии VAX-11/780, анонсированная в 1977 г., сейчас рассматривается как эталон вычислительной машины с быстродействием 1 млн оп./с. Несколько позже по времени появилась параллельная ветвь супермикроЭВМ – серия MicroVAX. В настоящее время серию VAX-11 полностью заменила серия VAX 8000 и новая ветвь – серия VAX 6000. Семейство ЭВМ VAX на сегодня насчитывает большое число разных моделей ЭВМ, существенно различающихся по производительности и стоимости, сюда же можно отнести и мультимашинные комплексы типа кластер. Но тем не менее, все эти ЭВМ объединяет общая для них архитектура VAX, что позволяет достичь полной программной совместимости между всеми моделями семейства.

В последние годы появились отечественные разработки вычислительных машин с архитектурой VAX – это ЭВМ "Электроника 82" и СМ 1700; также подобные разработки велись и в странах СЭВ – ЭВМ ИЗОТ 1055 С (БНР) и ТРА 580 (ВНР). Хотя все эти ЭВМ представляют самостоятельное и независимое исполнение архитектуры VAX, по своим функциональным решениям они в основном повторяют модели серии VAX-11. Прототипом операционной системы VAX/VMS для этих ЭВМ является операционная система МОСВП.

Предлагаемая вниманию читателей книга представляет собой введение в язык ассемблера и архитектуру ЭВМ VAX. Материал излагается на примере ассемблера VAX MACRO, входящего в состав операционной системы VAX/VMS. Книга построена как учебный курс, позволяющий сформировать довольно полное общее представление об особенностях ЭВМ семейства VAX. Кроме ассемблера и архитектуры ЭВМ VAX, в книге рассматриваются элементы операционной системы VAX/VMS. Книга включает множество практических примеров и заданий, способствующих усвоению пройденного материала. Ознакомившись буквально с первыми главами книги, читатель может начать осваивать полученные знания на практике – за терминалом вычислительной системы. В приложениях приводятся основные команды операционной системы VAX/VMS и редактора текста EDT. Тем не менее, несмотря на довольно большой объем книги, она приоткрывает только верхушку айсберга, который называется архитектура VAX и операционная система VAX/VMS.

Несколько слов о терминологии, поскольку в мнемонику инструкций языка ассемблера VAX MACRO вложена большая смысловая нагрузка, при работе над книгой мы стремились максимально сохранить свежесть оригинальной терминологии. Например, для обозначения форматов представления целых чисел использовались такие термины, как байт (byte), слово (word), длинное слово (longword), квадраслово (quadword), октаслово (octaword).

Мы надеемся, что книга вызовет интерес у широкого круга читателей, начинающим поможет глубже освоить основы организации ЭВМ и научиться программировать на языке ассемблера, имеющим опыт работы на ЭВМ с другой архитектурой облегчит переход на ЭВМ с архитектурой VAX, а программистам, уже работающим на ЭВМ с архитектурой VAX, послужит справочным пособием по программированию на языке ассемблера VAX MACRO.

А. С. Богданов

ПРЕДИСЛОВИЕ

Эта книга задумана как учебное пособие для односеместрового курса по программированию на языке ассемблера для ЭВМ семейства VAX. Предполагается, что читатель немного знаком с программированием на языках высокого уровня, таких как Паскаль или Фортран. Тем не менее в книге достаточно подробно излагаются основы организации ЭВМ. Наша цель – тесно связать вместе теорию и практику. Уделяя много внимания непосредственной работе на ЭВМ, одновременно мы попытаемся дать понять, что такое архитектура VAX. Эта книга будет полезной также для читателей, самостоятельно изучающих устройство и языки программирования ЭВМ семейства VAX.

Тема этой книги выбрана не только вследствие популярности семейства ЭВМ VAX, но и из-за того, что ЭВМ представляется нам идеальной для изучения. Логичность структуры делает ЭВМ семейства VAX легкими для программирования на языке ассемблера, а богатство архитектуры VAX позволяет использовать этот язык для решения сложных задач. Последняя особенность также делает ЭВМ семейства VAX идеальной основой для изучения архитектуры других ЭВМ. На примере одного семейства ЭВМ мы рассмотрим такие сложные вопросы, как операции над числами с плавающей точкой и упакованными десятичными числами, системный уровень ввода-вывода, согласование связи с языками высокого уровня, функции операционной системы. Изложение этих вопросов расширяет возможности применения книги в различных областях вычислительной техники.

Мы хотели сделать книгу доступной и содержательной как для изучающих программирование, так и для обучающих программированию. В первых главах книги показаны приемы, которые дают читателю возможность выполнять на ЭВМ простые программы. Приводятся программы для выполнения операций ввода-вывода, написанные на языке ассемблера, а также на языках высокого уровня. В последних главах рассматривается выполнение операций ввода-вывода с использованием сервисных средств операционной системы VAX/VMS. Кроме обсуждения вопросов, связанных с вводом-выводом, в приложениях даются способы выполнения и отладки программ, объясняется использование команд операционной системы VAX/VMS, а также приводится пример создания программы с помощью редактора текста EDT.

Материал книги изложен следующим образом:

Главы 1, 2 содержат начальные сведения, необходимые для читателей, имеющих ограниченный опыт работы с ЭВМ.

В гл. 3–5 вводятся основные понятия архитектуры VAX и языка ассемблера. Эти главы содержат достаточно сведений об использовании регистров процессора и организации простых подпрограмм, для того чтобы самостоятельно приступить к созданию довольно сложных программ.

Главы 6–8 содержат вспомогательный материал, который в основном посвящен обработке данных. Сюда включены сложные операции с числами, работа с символьной информацией и массивами.

В гл. 9–13 более детально рассматриваются такие темы, как подпрограммы и глобальные имена, макроинструкции и условное ассемблирование, системный уровень ввода-вывода, операции над числами с плавающей точкой, символьными строками и упакованными десятичными числами. Эти главы можно изучать в любом порядке или вообще пропустить.

В гл. 14, 15 излагаются новые темы, которые можно рассматривать как введение в архитектуру вычислительных систем, операционную систему и системное программирование.

Главы 1–7 рекомендуется изучать последовательно. Следующие главы надо читать в соответствии со схемой.

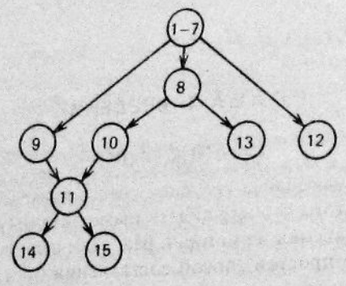


Рис. П.1

Мы выражаем искреннюю признательность всем, кто оказал помощь при подготовке этой книги. С. Лифшиц обеспечил системную поддержку при подготовке рукописи. Д. Йегер из Филадельфийского государственного колледжа естественных наук и текстильной промышленности проверил ранние варианты рукописи.

Своими замечаниями очень помогли, просмотрев часть или всю рукопись, Г. Барта (г. Женева), Д. А. Каньяс (Техасский университет, г. Остин), Дж. У. Горслайн и Д. Роуч (Политехнический институт и Университет шт. Вирджиния), П. С. Лафоллет, мл. (Темпльский университет), М. Г. Мерфи (Хьюстонский университет, Даунтаун колледж), Д. Д. Райли (Висконсинский университет в Ла-Кроссе), П. У. Росс (Миллерсвилский государственный колледж), Р. Спиллман (Пасификский лютеранский университет).

Мы благодарны П. Деспирито за самоотверженность и терпение при перепечатке рукописи.

Мы благодарим Д. Гоулдштейна за работу над упражнениями, а также фирму Integrated Circuit Systems и администрацию Темпльского университета за доступ к машинным ресурсам и документации.

Мы очень признательны нашему редактору К. Эллисон и техническим редакторам Б. О'Брайант и Х. Мандич-Протас из издательства PWS.

Наконец, выражаем особую благодарность Л. Дж. Кенаху из фирмы Digital Equipment Corporation, который просмотрел нашу рукопись и дал ценные советы и детальные комментарии по архитектуре VAX и операционной системе VAX/VMS.

Посвящаем эту книгу Кристиан, Джути, Марсии и Саре.

Чарльз Кэпс, Роберт Стаффорд

ГЛАВА 1. ВВЕДЕНИЕ

1.1. АННОТАЦИЯ

В этой главе содержатся начальные сведения, необходимые для понимания последующего материала книги. Для того чтобы увидеть вычислительную систему VAX в перспективе, приводятся краткие сведения из истории развития вычислительной техники. В конце главы представлен довольно простой способ составления программ на языке ассемблера. Этот материал предполагает некоторое знакомство читателя с языками высокого уровня, такими как Фортран или Паскаль.

1.2. ИСТОРИЧЕСКИЙ ОБЗОР

ПЕРВЫЕ ШАГИ

История вычислительных машин начинается гораздо раньше, чем многие предполагают. В 30-, 40-х годах прошлого столетия английский математик Ч. Бэббидж сделал попытку создать универсальное механическое вычислительное устройство с вводом информации с перфокарт. К сожалению, ему не удалось завершить работу над своей "аналитической машиной". Несколько позднее в том же веке американец Г. Холлерит разработал счетно-перфорационную систему для обработки исходных данных, вводимых с перфокарт, которая была использована при переписи населения в США в 1890 г.

Счетно-перфорационное оборудование, сделанное по проектам Холлерита, получило широкое распространение в начале XX в. Это оборудование, сокращенно называемое ЭСМ (электронные счетные машины)[1], состояло из электрических и механических узлов (моторы, переключатели, соленоиды, реле, приводы, муфты, храповики и т.д.). Хотя современная аппаратура существенно отличается от аппаратуры первых ЭСМ, тем не менее для представления данных на перфокартах до сих пор используется стандарт Холлерита (более детальное описание кодов Холлерита см. в гл. 8).

Одним большим недостатком оборудования электронных счетных машин было то, что его состав представлял собой конгломерат специальных устройств – перфораторов-репродукторов, табуляторов, сортировальных и раскладочных машин. Для выполнения определенной задачи эти устройства необходимо было соединить между собой особым образом. Любое изменение в системе достигалось изменением схемы межсоединений на коммутационных панелях, которые позволяли пользователю направлять данные и управляющую информацию почти таким же способом, как телефонистка осуществляет соединение абонентов.

Следующий большой шаг в развитии вычислительной техники был сделан в 40-х годах нашего столетия, когда Г. Айкен из Гарвардского университета разработал более универсальный и удобный метод управления процессом вычислений. Вычислительная машина Марк I была по существу гибридом гигантского арифмометра и механического пианино. Управление работой машины полностью "программировалось" с помощью пробивки со-

[1] Эти машины были скорее электромеханическими, чем электронными. – *Прим. ред.*

ответствующих комбинаций отверстий на некотором материале – носителе информации, закрепляемом на нескольких механических валиках, подобно механизму управления механического пианино.

ЭЛЕКТРОННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ

Как и ее предшественники, вычислительная машина Марк I была электромеханической. Это означает, что электричество использовалось исключительно для приведения в действие механических узлов. Механические узлы, в свою очередь, приводили в действие переключатели, которые управляли прохождением электрического тока. В лучшем случае на выполнение подобных механических операций требуется 0,001 с, а зачастую – гораздо больше. Увеличить это относительно низкое быстродействие можно заменой механических переключателей на электронные. *Электронным переключателем* называется переключатель, в котором отсутствуют движущиеся части. Переключение происходит при воздействии электростатических или магнитных полей, изменяющем электропроводность в вакууме или твердых материалах. В 40-х годах в качестве электронных переключателей широко использовались электронные лампы.

Вскоре после того, как начала работать машина Марк I, П. Эккерт и Д. Мочли из Пенсильванского университета создали первую большую электронную машину, названную ЭНИАК. Переключатели на электронных лампах способны срабатывать за 0,000001 с, поэтому ЭНИАК потенциально была в 1000 раз быстрее, чем Марк I. Рост быстродействия вычислительных машин породил раздумья о том, как лучше использовать эту скорость. Один из первых исследователей в области вычислительной техники выдвинул тезис о том, что шесть машин ЭНИАК способны навеки занять всех математиков страны поиском для них задач.

Для выполнения заданного процесса вычисления на ЭНИАКе было необходимо соединить большое количество проводов по определенной схеме, что занимало много времени. Следующее нововведение было связано с идеей возможности определения процесса вычисления *машинной программой*, которая, так же как и данные, хранится в памяти машины. Хранимая в памяти программа помимо того, что облегчает работу с вычислительной машиной, предоставляет возможность модифицировать саму себя во время своего выполнения. Следует отметить, что, хотя это было важно для первых моделей вычислительных машин, современные ЭВМ спроектированы с такими наборами инструкций, которые не требуют модификации программы во время выполнения. На самом деле модификация программ считается нежелательной по многим причинам, одной из которых является сложность их отладки.

В прошлом разработку идеи хранения программы во внутренней памяти машины многие приписывали Д. фон Нейману из Принстонского университета. Однако последние данные показывают, что Эккерт и Мочли заслуживают такого же большого уважения, как Д. фон Нейман.

ЭРА ТРАНЗИСТОРОВ

В конце 50-х – начале 60-х гг. транзисторы начали заменять электровакуумные ламповые переключатели в ЭВМ. Транзисторы имели пять очевидных преимуществ: меньший размер, потребляли много меньше энергии, были быстрее, дешевле и надежнее.

Хотя между вычислительными машинами на транзисторах и на электронных лампах нет принципиальной разницы, пять перечисленных преимуществ транзисторов имели огромное экономическое значение. Возникли два различных направления развития ЭВМ. Во-первых, стало возможным создание очень больших и мощных суперЭВМ, первыми из которых были IBM 7094, CDC 6600 и DEC PDP-6*. Во-вторых, впервые было начато производство небольших и недорогих мини-ЭВМ. Эти ЭВМ по цене были доступны небольшим лабораториям и отдельным пользователям. (Большие ЭВМ были настолько дороги, что их использование требовало планирования и разделения ресурсов). Первыми примерами мини-ЭВМ были IBM 1620, Royal McBee RPG 4000 и DEC PDP-5.

Распространение ЭВМ обоих типов положило начало широкому применению и более глубокому изучению вычислительной техники, ее теоретических основ; последнее соответственно отразилось на архитектуре и организации более поздних моделей. Тем не менее принципы организации ЭВМ в основном остались теми же, что и принципы организации первых универсальных вычислительных машин.

ИНТЕГРАЛЬНЫЕ СХЕМЫ

Транзисторы получают внедрением небольшого количества примесей в полупроводниковый кристалл, например кремний. Первые транзисторы были собраны в отдельном маленьком металлическом или пластмассовом корпусе с выведенными наружу контактами. При этом пропадало много места, так как сам транзистор был намного меньше корпуса. Интегральные схемы изготавливаются формированием большого количества транзисторов на поверхности кремниевой подложки. Затем непосредственно на поверхности подложки с помощью фотолитографии получают межсоединения. Это позволяет в очень малом пространстве разместить очень сложные схемы. (В настоящее время возможно размещение сотен тысяч транзисторов на кристалле площадью менее 1 $см^2$.)

Успехи технологии интегральных схем полностью изменили экономические характеристики вычислительных машин. Большие ЭВМ стали менее дорогими, а мини-ЭВМ – более сложными, и теперь зачастую их трудно отличить друг от друга. Существуют также так называемые микроЭВМ; такая ЭВМ может целиком размещаться на одном кремниевом кристалле, а ее стоимость составляет всего несколько долларов. Первоначально микроЭВМ не были совершенством, но последние достижения в технологии стерли отличия между микро- и мини-ЭВМ.

В настоящее время технология интегральных микросхем продолжает развиваться, и можно только догадываться, что нас ждет в будущем.

ДОСТИЖЕНИЯ В РАЗВИТИИ ТЕХНИЧЕСКИХ СРЕДСТВ

Совокупность физических устройств, входящих в вычислительную систему, принято называть *техническими средствами* (hardware). В предыдущих подразделах в основном говорилось о достижениях в проектировании и технологии изготовления процессоров. Одновременно, но не так быстро происходило развитие других технических средств, таких как память и периферийные устройства.

Аппаратура запоминающих устройств ЭВМ имела аналогичную историю развития – от электромеханических конструкций к интегральным схемам. С другой стороны, многие периферийные устройства, такие как печатающие устройства, терминалы и накопители на магнитной ленте, не сильно изменились. Периферийные устройства представляют наиболее дорогостоящую часть большинства вычислительных систем.

1.3. РАЗВИТИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

МАШИННЫЙ ЯЗЫК И ЯЗЫК АССЕМБЛЕРА

При разработке вычислительной машины проектировщик в первую очередь должен выбрать определенный набор команд или инструкций машины, после чего ЭВМ будет выполнять программы, составленные из этих инструкций. Такие инструкции называются *инструкциями на машинном языке*, а программы, составленные из этих инструкций, называются *программами на машинном языке*. Отметим, что обычно инструкции машинного языка одной вычислительной машины полностью отличаются от инструкций другой.

*Некоторые могут оспаривать правомерность употребления термина "суперЭВМ" для перечисленных примеров. Но для первой половины 60-х годов они были именно "супер".

в программе могут привести к существенным изменениям времени выполнения или объема занимаемой памяти. Знание языка ассемблера полезно для понимания и возможности прогнозирования подобных результатов. В частности, оно полезно при переносе программы, написанной на языке высокого уровня, с одной ЭВМ на другую. В конце концов, специалист по вычислительной технике обязан знать языки ассемблера для того, чтобы действительно понимать, как работает ЭВМ.

УПРАЖНЕНИЯ 1.1

Вопросы упражнений, отмеченные *, требуют дополнительных источников информации.

1. Чем знамениты Холлерит, Эккерт, Мочли фон Нейман, Бэкус, Атанасофф, Ада Байрон (графиня Лавлейс), Вирт?

*2. Используя дополнительные справочные материалы, составьте краткую биографию кого-либо из лиц, названных выше.

3. Ламповые вычислительные машины были полностью вытеснены полупроводниковыми. В настоящее время практически не осталось (если осталось вообще) работающих ламповых вычислительных машин. Для объяснения причин этого перечислите недостатки ламповых вычислительных машин.

*4. Имя Г. Хоппер – в числе имен основоположников вычислительной техники. С чем связана ее известность?

1.4. СЕМЕЙСТВО ЭВМ VAX

ФИРМА DIGITAL EQUIPMENT CORPORATION

Фирма DEC (Digital Equipment Corporation), как известно, была основана в 1957 г. В 60-е годы фирма выпустила целый ряд вычислительных машин. Первая из них, разработанная фирмой, была названа PDP-1. Последующим моделям присваивались имена по порядку до PDP-15 (сокращение от Programmed Data Processor). Однако по различным причинам PDP-2, PDP-3 и PDP-13 не были выпущены на рынок.

В 60-х годах фирма DEC первой начала разработку компьютеров стоимостью менее 100 тыс. долл. Так как даже малые вычислительные машины в то время стоили больше 100 тыс. долл., эти ЭВМ получили название мини-ЭВМ. Благодаря выпуску таких ЭВМ фирма DEC остается до сих пор ведущей в области разработки мини-ЭВМ.

Кроме мини-ЭВМ фирма DEC разработала также более мощные ЭВМ, предназначенные для одновременной работы 100 и более пользователей в режиме разделения времени. Машинами этого класса являются PDP-10 (сейчас она называется DECsystem-10) и DECSYSTEM-20. В настоящее время фирма DEC является вторым крупнейшим производителем ЭВМ после фирмы IBM.

СЕМЕЙСТВО ЭВМ PDP-11

В 1970 г. фирма DEC выпустила мини-ЭВМ PDP-11, которые должны были заменить семейство мини-ЭВМ PDP-5 и PDP-8, устаревшие к тому времени. (Тем не менее PDP-8 в микросхемном исполнении является основой персональных ЭВМ серии DECmate фирмы DEC).

ЭВМ PDP-11 представляют собой 16-*разрядную* мини-ЭВМ. Термин "16-разрядный" означает, что единица информации, над которой производится большинство операций, состоит из 16 двоичных разрядов (см. гл. 2 и 3). Эту единицу еще называют *длиной слова процессора.* В некоторой степени формат машинного слова определяет быстродействие процессора, а также влияет на его стоимость. Так как 32-разрядная ЭВМ может быть более быстродействующей, чем 16-разрядная, то и стоить она может дороже.

В 70-е гг. мини-ЭВМ PDP-11 положила начало разработке нового семейства ЭВМ. Все ЭВМ этого семейства имеют одинаковый машинный язык, но различаются по стоимости

и производительности на несколько порядков. Некоторые представители семейства чересчур малы, чтобы их можно было назвать мини-ЭВМ, они относятся к классу микроЭВМ. Так, например, в семейство ЭВМ PDP-11 входят персональные ЭВМ семейства DEC Professional 300. Другие представители семейства, такие как ЭВМ PDP-11/70, имеют высокую производительность и способны одновременно обслуживать до 50 и более пользователей в режиме разделения времени. Вычислительные машины, подобные PDP-11/70, иногда называют *супермини-ЭВМ.*

СЕМЕЙСТВО ЭВМ VAX

Семейство ЭВМ VAX сохраняет преемственность со старшими моделями ЭВМ семейства PDP-11. Название VAX – сокращение от Virtual Address eXtension, оно отражает тот факт, что в ЭВМ этого семейства применяется сложный способ управления памятью, который позволяет существенно расширить виртуальное адресное пространство программы. ЭВМ семейства VAX являются 32-разрядными. И хотя они могут выполнять инструкции машинного языка 16-разрядных ЭВМ PDP-11, в обычном режиме работы используется другой набор инструкций, который называется *основным* (native code).

Семейство VAX включает такие ЭВМ, как VAX-11/730, VAX-11/750, VAX-11/780 и VAX-11/782*. Все эти модели ЭВМ имеют одинаковый общий машинный язык. Старшие модели ЭВМ семейства VAX обозначаются большими номерами (например, VAX-11/785). Эти модели намного мощнее и дороже, чем модели с меньшими номерами в обозначении (например, VAX-11/730). Некоторые модели ЭВМ семейства VAX показаны на рис. 1.1. Стоимость вычислительных систем VAX колеблется от 100 тыс. до 1 млн. долл. в зависимости от конфигурации системы. Младшими моделями ЭВМ семейства VAX являются ЭВМ серии MicroVAX. Стоимость вычислительной системы на основе ЭВМ серии MicroVAX в среднем составляет 20 тыс. долл., но может быть и ниже, что также зависит от конфигурации (рис. 1.2)

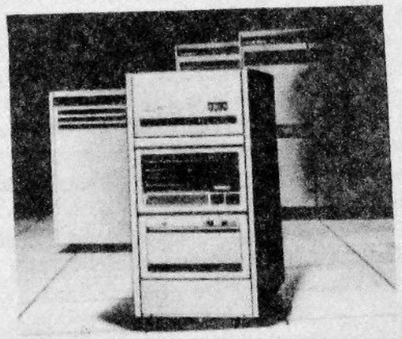


Рис. 1.1. Вычислительная система VAX-11/730 и другие ЭВМ семейства VAX-11: VAX-11/750, VAX-11/780 и VAX-11/782 (с разрешения фирмы Digital Equipment Corporation)

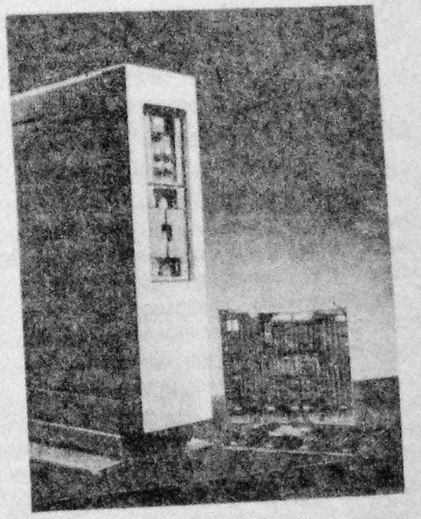


Рис 1.2. ЭВМ MicroVAX I

*Здесь перечислены только основные модели ЭВМ серии VAX-11. В настоящее время фирма DEC выпускает новую серию ЭВМ VAX 8000, которая полностью заменила серию VAX-11, сохранив с последней полную программную совместимость. – *Прим. ред.*

Вычислительные системы VAX находят применение во многих областях. Например, они могут быть использованы для работы в режиме разделения времени, обслуживая одновременно 100 и более пользователей, а также для решения научно-технических задач в такой специфической области, как автоматизированное проектирование и автоматизированное производство (CAD/CAM). Эти приложения, требующие выполнения большого объема вычислений, связаны с задачами, решение которых занимает много машинного времени даже на ЭВМ с очень высоким быстродействием. В результате для таких приложений одна ЭВМ типа VAX может обеспечить одновременную работу только нескольких пользователей.

Но в основном ЭВМ семейства VAX предназначены для широкого круга приложений, связанных с обработкой данных, где существенным является быстродействие при умеренных затратах. Операционная система для ЭВМ семейства VAX называется VMS (сокращение от Virtual Memory System) и поддерживает работу пользователей в интерактивном и пакетном режимах обработки информации. Она позволяет организовать распределенную обработку данных в вычислительной сети, объединяющей ЭВМ семейства VAX. В эту сеть могут быть включены также ЭВМ других типов. Для ЭВМ VAX существует целый ряд языков программирования высокого уровня, таких как Фортран, Кобол, Бейсик, Паскаль, ПЛ/1, APL и т.д., а также и язык ассемблера, которому посвящена данная книга.

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

На вычислительной машине можно работать и без операционной системы, но тогда реализация различных операций, таких как ввод, вывод, управление файлами и трансляция с языков программирования, может потребовать значительных усилий при программировании. В итоге фактически всем пользователям приходится приобретать для своей ЭВМ комплекс программ, обеспечивающий работу системы в целом. Этот комплекс программ называется *системным программным обеспечением* (systems software); они разделены на несколько групп:

1. Мониторы. Эти программы управляют и координируют выполнение всех остальных программ.

2. Обслуживающие программы. Вызываются по командам пользователя для создания или модификации файлов, назначения устройств ввода-вывода, получения информации о состоянии вычислительной системы и т.д.

3. Системные подпрограммы. Вызываются из программ пользователя для выполнения системных функций, аналогичных функциям программ, описанных во второй группе.

4. Трансляторы. Позволяют пользователю создавать программы на различных языках: ассемблере, Фортране, Паскале и т.д.

5. Библиотеки программ. Программы, из которых состоят библиотеки, позволяют использовать специальные математические и статистические функции, обеспечивают управление графическими устройствами и т.д.

Как правило, операционная система содержит программы первых трех групп, а трансляторы и библиотеки программ выбираются по усмотрению пользователя. Поскольку разработка системного программного обеспечения требует значительных усилий, вместе с ЭВМ приходится приобретать лицензию на право пользования операционной системой. Стоимость такой лицензии может составить несколько тысяч долларов.

Операционная система, поставляемая фирмой DEC, называется VAX/VMS. Другой операционной системой, которая также может использоваться для ЭВМ семейства VAX, является операционная система UNIX[1], существенно отличающаяся от VAX/VMS. Настоящая книга ориентирована на применение операционной системы VAX/VMS.

[1] Название UNIX является фирменным знаком AT&T Bell Laboratories. Фирма DEC поставляет переработанный вариант операционной системы UNIX, который называется ULTRIX-32. – *Прим. ред.*

1.5. НАЧАЛЬНЫЕ СВЕДЕНИЯ

ПРОГРАММА НА ЯЗЫКЕ АССЕМБЛЕРА

Зная один из языков высокого уровня, довольно легко написать и опробовать простые программы на языке ассемблера. В качестве примера на рис. 1.3 представлена простая программа на трех языках: Паскале, Фортране и MACRO, языке ассемблера для ЭВМ семейства VAX. Строки программ имеют последовательную нумерацию. Приведенные номера строк не являются частью какой-либо из этих программ.

Оператор определенного типа в одном языке программирования может не иметь аналогичного оператора в другом языке. Тогда в программах, представленных на рис. 1.3, в некоторых строках вводится комментарий НЕ НУЖНО, что позволяет обеспечивать соответствие друг другу остальных строк. Как видно из рисунка, комментарии в Паскале выделяются с помощью символов (* и *), а в Фортране комментарий начинается с буквы C в первой позиции строки. В языке ассемблера комментарием считается любая последовательность символов, начинающаяся с точки с запятой (;).

В языках высокого уровня, таких как Паскаль и Фортран, такие символы языка, как J, K и DIF, называются *именами переменных*. В языках ассемблера они называются *символическими адресами*. В рассматриваемых трех программах действие операторов в строках 3, 4 и 5 одинаково. Они указывают, что переменные (символические адреса) J, K и DIF являются именами ячеек памяти длиной 32 бита (4 байта), в которых будут находиться целые числа. (Директива J: .BLKL1 (сокращение от BLocK of Longword) объявляет J именем блока, состоящего из одного длинного слова. В языке ассемблера для семейства ЭВМ VAX *длинное слово* есть по сути 32-разрядное двоичное целое число.)

Во всех трех программах одинаковые действия выполняют операторы в строках 7–10. Во всех программах после выполнения 10-й строки значение переменной DIF будет равно 8. Необходимо отметить, что в операторах присваивания, таких как J := 20; или J = 20, пересылка информации осуществляется справа налево, а в языке ассемблера для семейства ЭВМ VAX пересылка информации производится слева направо. Оператор MOVL #20, J следует читать как "Переслать число 20 в длинное слово J". Аналогично оператор SUBL2K, DIF следует читать как "Вычесть содержимое длинного слова K из содержимого длинного слова DIF (результат поместить в длинное слово DIF)".

На рис. 1.4 показана структура простой программы на языке ассемблера. Пользователь должен вставить текст своей программы и имена переменных (символические адреса) на места, помеченные в этой заготовке соответствующими надписями. Следует отметить, что различные поля инструкций выровнены и образуют вертикальные колонки. Этому правилу легко следовать, используя табуляцию; такая операция реализована почти на

Паскаль

```
1.  PROGRAM SAMPLE;
2.  VAR
3.     J : INTEGER;
4.     K : INTEGER;
5.     DIF : INTEGER;
6.  BEGIN
7.     J := 20;
8.     K := 12;
9.     DIF := J;
10.    DIF := DIF-K;
11. (*НЕ НУЖНО В ПАСКАЛЕ*)
12  END.
```

Фортран

```
1.  C НЕ НУЖНО В ФОРТРАНЕ
2.  C НЕ НУЖНО В ФОРТРАНЕ
3.        INTEGER J
4.        INTEGER K
5.        INTEGER DIF
6.  C НЕ НУЖНО В ФОРТРАНЕ
7.        J=20
8.        K=12
9.        DIF=J
10.       DIF=DIF-K
11.       STOP
12.       END
```

Ассемблер

```
1.  ;НЕ НУЖНО В АССЕМБЛЕРЕ
2.  ;НЕ НУЖНО В АССЕМБЛЕРЕ
3.  J:      .BLKL   1
4.  K:      .BLKL   1
5.  DIF:    .BLKL   1
6.          .ENTRY  START,0
7.          MOVL    #20,J
8.          MOVL    #12,K
9.          MOVL    J,DIF
10.         SUBL2   K,DIF
11.         $EXIT_S
12.         .END    START
```

Рис. 1.3. Три варианта программы на разных языках программирования

```
ИМЯ_1:      .BLKL    1          ;РЕЗЕРВИРОВАНИЕ ПАМЯТИ ДЛЯ ИМЯ_1
ИМЯ_2:      .BLKL    1          ;РЕЗЕРВИРОВАНИЕ ПАМЯТИ ДЛЯ ИМЯ_2
...
ИМЯ_N:      .BLKL    1          ;РЕЗЕРВИРОВАНИЕ ПАМЯТИ ДЛЯ ИМЯ_N
            .ENTRY   START,1    ;ТОЧКА ВХОДА В ПРОГРАММУ
            ИНСТРУКЦИЯ 1        ;ТЕЛО ПРОГРАММЫ
            ИНСТРУКЦИЯ 2
              ...
            ИНСТРУКЦИЯ К
            $EXIT_S             ;КОНЕЦ ПРОГРАММЫ
            .END     START      ;START - АДРЕС ТОЧКИ ВХОДА
```

Рис. 1.4. Структура программы на языке ассемблера

всех терминалах фирмы DEC. Выравнивание не является обязательным для языка ассемблера, но оно необходимо с практической точки зрения, для удобства чтения текста программы. Надо обратить внимание на то, что в предпоследнем операторе программы 6-й символ в имени $EXIT_S является знаком подчеркивания (_), а не знаком минус (–).

ВЗАИМОДЕЙСТВИЕ С ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМОЙ

Обычно взаимодействие с вычислительной системой VAX осуществляется с помощью интерактивного терминала. Имеется два основных типа терминалов – телетайпы и видеотерминалы. Телетайпы похожи на пишущую машинку, вся информация, вводимая с телетайпа или поступающая на него от вычислительной системы, регистрируется на бумаге. Видеотерминалы похожи на телевизор с присоединенной к нему клавиатурой. В приложении А описывается последовательность действий при работе с вычислительной системой, которая позволяет подготовить программу на языке ассемблера, а затем ее выполнить.

ИНСТРУКЦИИ ЯЗЫКА АССЕМБЛЕРА

Инструкции языка ассемблера, такие как

```
MOVL    J, DIF
```

состоят из двух частей или полей. Первое поле называется *кодом операции* или *полем кода операции*. В нем записываются коды операций, например MOVL или SUBL2. Второе поле называется *полем операндов*. Это поле может содержать символические адреса (имена переменных), например J или DIF, а также константы, такие как #20 и #12. Операнды разделяются запятой. Число операндов определяется кодом операции.

Язык ассемблера VAX имеет несколько сотен кодов операций. Например, по инструкции

```
ADDL2   BONUS, WAGE
```

производится сложение содержимого длинного слова BONUS с содержимым длинного слова WAGE, результат сложения помещается в длинное слово WAGE. Эта инструкция аналогична операторам WAGE:=WAGE+BONUS на Паскале или WAGE=WAGE+BONUS на Фортране. (Предполагается, что переменные WAGE и BONUS объявлены как целые числа.) Аналогично по инструкции

```
ADDL3   BILL, TAX, TOTAL
```

производится сложение содержимых длинных слов BILL и TAX, а результат сложения помещается в длинное слово TOTAL. Эта инструкция имеет следующие аналоги: TOTAL:=:=BILL+TAX; на Паскале и TOTAL=BILL+TAX на Фортране. Содержимое длинных

слов BILL и TAX при этом не меняются, а предыдущее содержимое длинного слова TOTAL теряется.

Ниже приведены некоторые инструкции языка ассемблера и аналогичные им (по действию) операторы Паскаля и Фортрана. Предполагается, что в программах на Паскале и Фортране переменные A, B, C и D объявлены как целые. Для инструкций языка ассемблера символические адреса A, B, C и D предполагаются объявленными заранее директивой .BLKL 1.

Ассемблер		Паскаль	Фортран
MOVL	#100, A	A:=100;	A=100
MOVL	A, B	B:=A;	B=A
ADDL2	#10, B	B:=B+10;	B=B+10
ADDL3	#40, A, C	C:=40+A;	C=40+A
SUBL2	B, C	C:=C-B;	C=C-B
SUBL3	#30, A, D	D:=A-30;	D=A-30
SUBL3	B, #200, D	D:=200-B;	D=200-B

Заметим, что последний из указанных операторов языка ассемблера SUBL3 B, #200, D, следует читать как "Вычесть содержимое длинного слова B из числа 200 и поместить результат в D".

Ниже следуют операторы на языке ассемблера, содержащие различные ошибки. Более детально последствия этих ошибок описаны в гл. 4 и приложении А.

Оператор ассемблера		Допущенная ошибка
MOVL	100, A	ПРОПУЩЕН ЗНАК НОМЕРА (#)
MOVL	A, #100	ПОСЛЕДНИЙ ОПЕРАНД НЕ МОЖЕТ БЫТЬ КОНСТАНТОЙ
ADDL2	A B	МЕЖДУ ОПЕРАНДАМИ ТРЕБУЕТСЯ ЗАПЯТАЯ
ADDL3	A, B	В ИНСТРУКЦИИ ADDL3 ТРЕБУЕТСЯ ТРИ ОПЕРАНДА
MOVE	A, B	ДОЛЖНО БЫТЬ MOVL, А НЕ MOVE
SUB2L	A, B	ДОЛЖНО БЫТЬ SUBL2, А НЕ SUB2L
ADDL4	A, B, C, D	НЕ СУЩЕСТВУЕТ КОД ОПЕРАЦИИ ADDL4

Языки программирования высокого уровня обычно настолько сложны, что один оператор языка транслируется в несколько инструкций машинного языка, тогда как в языке ассемблера большинство операторов транслируется "один к одному". В этом состоит основное различие между языками ассемблера и языками высокого уровня. Предыдущие примеры, написанные на языках высокого уровня, были выбраны достаточно простыми, чтобы их реализация на языке ассемблера была очевидной. В большинстве случаев трансляция осуществляется не так просто. Например, оператор Паскаля A:=B+C–D–10 может быть реализован на языке ассемблера следующим образом:

```
ADDL3   B, C, A
SUBL2   D, A
SUBL2   #10, A
```

УПРАЖНЕНИЯ 1.2

1. Прочитайте приложение А, а затем ассемблируйте и выполните на ЭВМ программу, представленную на рис. 1.3.

2. Изменив немного программу из п. 1, как показано ниже попробуйте ассемблировать и выполнить ее на ЭВМ (изменения необходимо вносить по очереди). Каждое изменение внесет в программу ошибку. Опишите последствия каждой ошибки, т.е. к чему она приводит: выдается ли сообщение об ошибке ассемблером или компоновщиком? Выдается ли оно на этапе выполнения или ошибка просто ведет к неверным результатам?

   а) исключите третью строку (J:.BLKL1), так чтобы символический адрес J не был объявлен в программе;

б) уберите знак : в 4-й строке, после чего в ней останется K .BLKL 1;
в) уберите из строки 7 знак #, после чего останется MOVL 20, J;
г) замените 7-ю строку строкой MOVL 700, J;
д) замените 9-ю строку строкой MOVL #J, DIF.

3. Напишите на языке ассемблера и выполните программу, эквивалентную следующим программам на Паскале и Фортране:

Фортран

```
INTEGER J, K, L
J=15
K=22
L=J-K+9
STOP
END
```

Паскаль

```
PROGRAM THREE;
VAR
 J, K, L: INTEGER;
BEGIN
 J := 15;
 K := 22;
 L := J-K+9;
END.
```

4. Напишите на языке ассемблера и выполните программу, эквивалентную приведенным ниже программам на Паскале и Фортране. Заметьте, что результат умножения может быть получен с помощью последовательных операций сложения. Зарезервируйте в вашей программе промежуточное длинное слово для хранения суммы J и K во время выполнения программы. После выполнения программы переменные J и K должны сохранить свои прежние значения 15 и 9 соответственно.

Фортран

```
INTEGER J, K, L
J=15
K=9
L=3*(J+K)
STOP
END
```

Паскаль

```
PROGRAM FOUR;
VAR
 J, K, L: INTEGER;
BEGIN
 J := 15;
 K := 9;
 L := 3*(J+K);
END.
```

5. Напишите на языке ассемблера и выполните программу, эквивалентную следующим программам на Паскале и Фортране (*подсказка*: K можно вычислить меньше чем за 10 сложений):

Фортран

```
INTEGER J, K
J=15
K=32*J
STOP
END
```

Паскаль

```
PROGRAM FIVE;
VAR
 J, K: INTEGER;
BEGIN
 J := 50;
 K := 32*J;
END.
```

6. Выполните п. 5 упр. 1.2, заменив выражение для вычисления K на K := 23 * J (*подсказка:* так как 23 = 16 + 4 + 2 + 1, вычислите выражения 1 * J, 2 * J, 4 * J и 16 * J и сложите результаты).

7. Ниже представлены две инструкции MULL2 и MULL3, которые служат для умножения двух чисел. Они имеют следующие форматы:

Ассемблер		Паскаль	Фортран
MULL2	#10, B	B: =10*B;	B=10*B
MULL3	#40, A, C	C: =40*A;	C=40*A
MULL2	A, B	B: =A*B;	B=A*B
MULL3	A, B, C	C: =A*B;	C=A*B

Выполните п. 5 упр. 1.2, используя инструкции MULL2 и MULL3.

ГЛАВА 2. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ, АРИФМЕТИЧЕСКИЕ И ЛОГИЧЕСКИЕ ОПЕРАЦИИ В ВЫЧИСЛИТЕЛЬНЫХ МАШИНАХ

2.1. ВВЕДЕНИЕ

Вычислительные машины выполняют обработку информации под управлением программы. Существенным является способ представления информации в ЭВМ. В данной главе будет рассмотрено двоичное представление целых чисел и логических значений в памяти ЭВМ VAX. В последующих главах будет показано, как представляются символьная информация,числа с плавающей точкой и простые структуры данных, такие как массивы.

2.2. СИСТЕМЫ СЧИСЛЕНИЯ

ИСТОРИЧЕСКИЕ СВЕДЕНИЯ

На протяжении всей истории человечества было изобретено много различных способов счисления или счета. И сегодня еще можно встретить людей, которые используют примитивные способы счета, например складывают камешки в мешок или делают зарубки на палочке. До сих пор существует и более сложная римская система счисления, римские

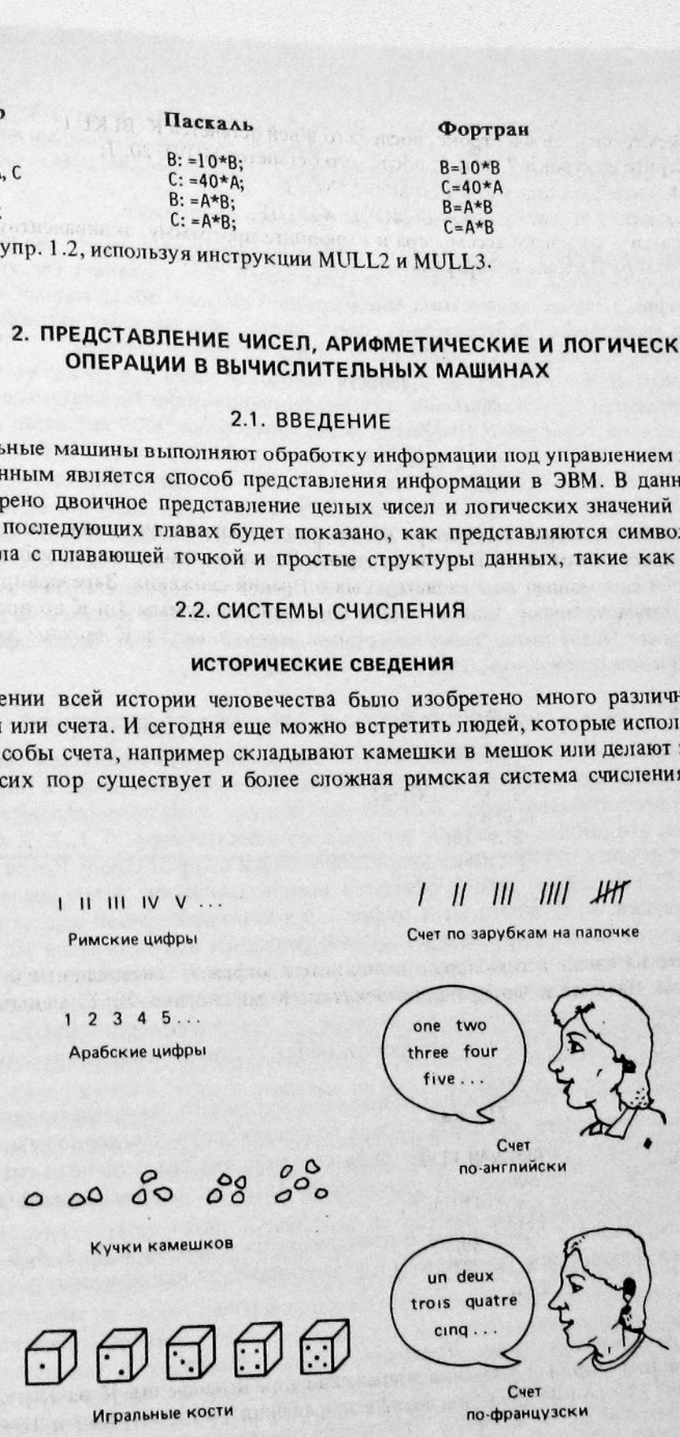


Рис. 2.1. Некоторые системы представления чисел

цифры сейчас используются главным образом в демонстрационных целях[1]. Однако наиболее привычной для всех нас системой счисления является десятичная или арабская[2]. На рис. 2.1 показаны примеры представления чисел в различных системах.

ДЕСЯТИЧНАЯ СИСТЕМА СЧИСЛЕНИЯ

Одним из свойств традиционных систем представления чисел является стремление сгруппировать числа по "пятеркам" и "десяткам". Это заставляет считать числа 5, 10 и кратные им, а также степени этих чисел крайне важными, обладающими чуть ли не магическими свойствами. Действительно, очень легко умножить или разделить число на 10. Выполнить ту же операцию с числами 8, 9, 11 или 12 гораздо сложнее.

На самом деле единственной причиной наличия у чисел 5 и 10 каких-либо особенных свойств является то, что основание системы счисления равно 10. Системы счисления могут иметь и другие основания. Например, в Вавилоне более 4000 лет назад использовалась система счисления с основанием 60, называемая шестидесятеричной. В какой-то степени она сохранилась и до наших дней, например: в 1 ч – 60 мин, а в 1 мин – 60 с. Употребление слов "дюжина" для обозначения числа 12 и "гросс"[3] для числа 144 является примером использования системы счисления с основанием 12. В действительности особых преимуществ в десятичной системе счисления нет. Наиболее вероятно, что начало развитию десятичной системы положил счет на пальцах рук. Поскольку вычислительные машины не имеют рук с пятью пальцами, использование для них десятичной системы не дает заметного преимущества. Более того, вычислительные машины работают более эффективно при применении систем счисления, отличных от десятичной.

СЧЕТ В СИСТЕМАХ СЧИСЛЕНИЯ, ОТЛИЧНЫХ ОТ ДЕСЯТИЧНОЙ

По мере рассмотрения различных систем счисления будут излагаться основы десятичной системы, т.е. как в ней осуществляются счет, сложение, вычитание, а также как интерпретируется представление чисел. В *десятичной системе* числа представляются в виде строки символов, выбранных из набора, состоящего из десяти цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Счет ведется начиная с 0 при последовательной записи цифр от 0 до 9. Когда будет достигнуто число 9, все цифры набора окажутся использованными. Затем слева добавляется еще одна позиция, куда помещается цифра 1, а в начальной правой позиции повторяется последовательная запись цифр начиная с 0, что позволяет получать числа 10, 11, 12 и т.д. Когда получено число 19, при продолжении счета цифра 9 заменяется на 0, а в соседней левой позиции прибавляется 1, в результате получается число 20. Следовательно, когда справа образуется последовательность девяток, то при увеличении счета на 1 все они заменяются на 0, при этом перенос распространяется справа налево. Таким образом, после 3999 следует 4000.

Как было отмечено в предыдущем подразделе, нет ничего сверхестественного ни в числе 10, ни в использовании для счета набора из десяти цифр. Предположим, что имелось бы всего шесть цифр. Такой могла бы быть сегодняшняя традиционная система счисления, будь у людей по три пальца на руке вместо пяти. Эта система называлась бы системой счисления с *основанием* 6.

В системе счисления с основанием 6 имеется шесть цифр: 0, 1, 2, 3, 4, 5. Счет в этой системе ведется фактически так же, как и в десятичной, за исключением того, что отсутствуют цифры 6, 7, 8 и 9, а возврат к 0 и перенос осуществляются по достижении цифры 5. Так, за числом 5 следует число 10, за 15 – 20, за 255 – 300. В табл. 2.1 показано, как осуществляется счет от 0 до 36 в системе счисления с основанием 6.

---

[1] Например, на циферблатах часов. – *Прим. ред.*

[2] Арабские цифры проникли в европейскую культуру в XII в. при переводе на латынь книги арабского математика Мухаммеда Ибн-Муса аль-Хорезми (780 – 850 гг.). Трансформированное имя аль-Хорезми лежит в основе слова "*алгоритм*, означающего однозначно определенный пошаговый процесс решения задачи".

[3] "Дюжина дюжин". – *Прим. ред.*

Таблица 2.1
Счет в системе счисления с основанием 6

Основание 10	Основание 6	Основание 10	Основание 6
0	0	19	31
1	1	20	32
2	2	21	33
3	3	22	34
4	4	23	35
5	5	24	40
6	10	25	41
7	11	26	42
8	12	27	43
9	13	28	44
10	14	29	45
11	15	30	50
12	20	31	51
13	21	32	52
14	22	33	53
15	23	34	54
16	24	35	55
17	25	36	100
18	30		

2.3. ДВОИЧНЫЕ ЧИСЛА

ЗАЧЕМ НУЖНЫ ДВОИЧНЫЕ ЧИСЛА

В предыдущем разделе была представлена система счисления с основанием 6 для иллюстрации того, как может осуществляться счет в системах, отличных от десятичной. Однако намного более важной, хотя и странной на вид, является система счисления с *основанием* 2, или *двоичная* система счисления.

Напомним, что используемая нами десятичная система счисления основана на примитивной практике счета на пальцах. Другими словами, пальцы были для человека доступным техническим средством счета. Именно потому, что пальцев – десять, десятичная система кажется человеку естественной. Использование систем счисления с основаниями 5 и 20 имеет такое же историческое происхождение.

Возникает следующий вопрос: "Какая система естественна для ЭВМ?" Ясно, что вычислительные машины не имеют пальцев, следовательно, отсутствуют предпосылки к использованию десятичной системы. То, что естественно для ЭВМ, зависит от природы процессов, которые происходят в различных узлах вычислительной машины. Анализ работы ЭВМ показывает, что каждый процесс складывается из одного или более событий, которые либо происходят, либо – нет. Если взглянуть на некоторый участок перфокарты, то можно увидеть, что на каждой позиции перфокарты либо есть пробитое отверстие, либо его нет. Существует два, и только два возможных исхода. Отверстие не может быть пробито наполовину. Событие, результатом которого может быть только один из двух возможных

исходов (отверстие пробито или не пробито), называется *двоичным*[1]. Ниже приведено несколько примеров различных двоичных событий.

Клавиша на клавиатуре	*Нажата* или *не нажата*
Отверстие в перфокарте	*Пробито* или *не пробито*
Двухпозиционный тумблер	*Включен* или *выключен*
Электрическая лампа	*Зажжена* или *погашена*
Цифровой сигнал на шине	*Высокий* или *низкий* уровень напряжения

СЧЕТ В ДВОИЧНОЙ СИСТЕМЕ СЧИСЛЕНИЯ

Счет в двоичной системе выполняется практически так же, как в десятичной или в шестеричной системах счисления, с той разницей, что в двоичной системе имеется всего две цифры: 0 и 1. Счет, как обычно, начинается с 0. Следующим числом является 1, но дальнейший счет без добавления позиции слева невозможен, поскольку больше нет цифр. Поэтому происходит возврат к 0 и перенос 1 на следующую позицию, в результате будет получено двоичное число 10, которое соответствует числу 2 в десятичной системе счисления. В табл. 2.2 приведено представление чисел от 0 до 32 в двоичной системе счисления.

Таблица 2.2

Счет в двоичной системе счисления

Основание 10	Основание 2	Основание 10	Основание 2
0	0	17	10001
1	1	18	10010
2	10	19	10011
3	11	20	10100
4	100	21	10101
5	101	22	10110
6	110	23	10111
7	111	24	11000
8	1000	25	11001
9	1001	26	11010
10	1010	27	11011
11	1011	28	11100
12	1100	29	11101
13	1101	30	11110
14	1110	31	11111
15	1111	32	100000
16	10000		

ДВОИЧНАЯ АРИФМЕТИКА

Двоичное сложение и вычитание выполняются по той же схеме, которая обычно используется в десятичной арифметике. Сначала необходимо сформулировать правило сложения двух двоичных цифр. Хотя приемы сложения аналогичны приемам, используемым в десятичной арифметике, легче просто запомнить следующую таблицу:

[1] Читатели могут отметить, что для таких событий, как пробивка отверстий, число возможных состояний может быть более двух. Например, может быть пробито три, четыре и более отверстий различной формы. Однако в вычислительной технике, как правило, это не практикуется, поскольку устройство, способное надежно распознавать несколько отверстий различной формы, было бы значительно дороже устройства, умеющего распознавать только наличие или отсутствие пробивки.

0 + 0 = 00 (0 без переноса);
0 + 1 = 01 (1 без переноса);
1 + 0 = 01 (1 без переноса);
1 + 1 = 10 (0 с переносом).

Аналогичный набор правил может быть выведен и для вычитания. На рис. 2.2 показаны некоторые простые действия с двоичными числами.

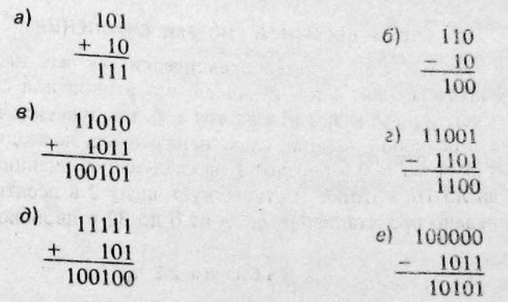


Рис. 2.2. Двоичные вычисления

Представление чисел в двоичной системе счисления может интерпретироваться таким же образом, как и представление чисел в десятичной системе. Но в отличие от десятичной в двоичной системе имеется только две цифры: 0 и 1. Значение каждой цифры числа зависит от того места, которое эта цифра занимает, и определяется умножением на соответствующую степень числа 2. Так, $11010 = (1*16) + (1*8) + (0*4) + (1*2) + (0*1) = 16 + 8 + 2 = 26$.

Аналогично

$$1011 = (1*8) + (0*4) + (1*2) + (1*1) = 8 + 2 + 1 = 11.$$

И наконец,

$$100101 = (1*32) + (0*16) + (0*8) + (1*4) + (0*2) + (1*1),$$

что равняется $32 + 4 + 1 = 37$. Отметим также, что $37 = 11 + 26$, что и получено на рис. 2.2, *в*. Таблица степеней числа 2 представлена в конце книги.

2.4. ПРЕОБРАЗОВАНИЕ ЧИСЕЛ

ОСНОВНЫЕ АЛГОРИТМЫ

Существуют два распространенных способа преобразования чисел из одной системы счисления в другую. В одном из них используется умножение, в другом – деление. Преобразование умножением применяется для перевода чисел из какой-либо системы в систему счисления, в которой в данный момент выполняются арифметические действия. Для выполнения арифметических действий человек обычно использует десятичную систему счисления, тогда как большинство ЭВМ используют двоичную систему. Преобразование делением применяется для преобразования чисел системы, в которой в данный момент выполняются арифметические действия, в какую-либо другую систему.

ПРЕОБРАЗОВАНИЕ УМНОЖЕНИЕМ

Способ преобразования умножением основан на представлении чисел в виде суммы степеней. Например, в десятичной системе значение цифры определяется в зависимости от того, в каком разряде числа она находится, умножением соответственно на 1, 10, 100, 1000 и т.д. В двоичной системе такое умножение выполняется на степени числа 2, т.е. на 1, 2, 4, 8, 16, 32 и т.д. В общем случае в системе счисления с основанием b значения цифр, составляющих число, определяются умножением на степени числа b. Поэтому заданное в системе с основанием b число

$$d_4 d_3 d_2 d_1 d_0$$

можно представить в виде

$$d_4 * b^4 + d_3 * b^3 + d_2 * b^2 + d_1 * b^1 + d_0.$$

Перепишем эту формулу так, чтобы не требовалось каждый раз выполнять операцию возведения в степень. Новая формула имеет вид

$$((((0 * b + d_4) * b + d_3) * b + d_2) * b + d_1) * b + d_0.$$

Читатель может убедиться, что эта формула эквивалентна предыдущей. Отметим, что член $0 * b$ равен 0 и его можно было бы исключить, однако его присутствие упрощает описание алгоритма. Алгоритм выполнения вычислений по этой формуле может быть описан следующим образом:

Шаг 1. Присвоить переменной ANSWER значение 0.

Шаг 2. Начать преобразование с самой левой цифры числа.

Шаг 3. Умножить значение переменной ANSWER на b.

Шаг 4. Получить значение следующего разряда числа (очередную цифру) и сложить его с хранящимся в переменной ANSWER значением.

Шаг 5. Если в числе есть еще цифры, вернуться на шаг 3. Иначе переменная ANSWER содержит преобразованное число.

В качестве примера рассмотрим преобразование двоичного числа 10110 в десятичное. Ниже для каждого шага алгоритма приведены оставшиеся цифры двоичного числа и значения переменной ANSWER.

Шаг	ANSWER	Оставшиеся разряды двоичного числа
1	0	—
2	0	10110
3	0	10110
4	1	0110
5	1	0110
3	2	0110
4	2	110
5	2	110
3	4	110
4	5	10
5	5	10
3	10	10
4	11	0
5	11	0
3	22	0
4	22	—
5	22	Двоичное число 10110 соответствует десятичному числу 22

ПРЕОБРАЗОВАНИЕ ДЕЛЕНИЕМ

Способ перевода чисел из одной системы в другую с помощью деления по своему действию является обратным преобразованию, рассмотренному выше. Как известно, при делении десятичного числа на 10 остаток от деления будет равен значению цифры в самом правом разряде числа. Оставшиеся слева цифры числа образуют полученное при делении частное. Таким образом, при делении числа 3754 на 10 остаток равен 4, а частное равно 375. Этот же принцип соблюдается при делении числа в системе счисления с основанием b на число b. Остаток будет равен значению самой правой цифры числа, а частное будет представлено цифрами, оставшимися слева.

Так как частное содержит оставшиеся цифры, этот прием может повторяться, чтобы преобразовать все число. Ниже приведен алгоритм для перевода числа N в систему счисления с основанием b.

Шаг 1. Присвоить n значение N.

Шаг 2. Вычислить q (частное) и r (остаток) от деления n на b.

Шаг 3. Вывести содержимое r как очередную цифру числа в системе с основанием b (цифры располагаются справа налево).

Шаг 4. Присвоить n значение q. Если n не равно нулю, перейти на шаг 3. Иначе преобразование закончено. При использовании этого алгоритма для преобразования десятичного числа 25 в двоичное число получается следующая последовательность шагов:

Шаг	n	q	r	Вывод	Преобразованное число	
1	25					
2	25	12	1			
3	25	12	1	1	1	
4	12	12	1		1	
2	12	6	0		1	
3	12	6	0	0	01	
4	6	6	0		01	
2	6	3	0		01	
3	6	3	0	0	001	
4	3	3	0		001	
2	3	1	1		001	
3	3	1	1	1	1001	
4	1	1	1		1001	
2	1	0	1		1001	
3	1	0	1	1	11001	Искомое
4	0	0	1		11001	число равно 11001

Отметим, что алгоритм преобразования умножением использовался при преобразовании двоичного числа в десятичное, а алгоритм преобразования делением – для преобразования десятичного числа в двоичное. Причина такого выбора алгоритма заключается в том, что для нас естественной является десятичная арифметика. Положив это в основу общего правила, можно сказать, что при преобразовании чисел из одной системы счисления в другую выбор алгоритма зависит от того, в какой из этих двух систем выполняются арифметические действия. Если арифметические действия выполняются в той системе, в которую осуществляется преобразование, то следует выбрать алгоритм преобразования умножением, и наоборот, если арифметические действия выполняются в системе, из которой делается преобразование, то выбирают алгоритм преобразования делением. Человек, осуществляя преобразование вручную, пользуется десятичной арифметикой. Однако для ЭВМ, в частности для ЭВМ семейства VAX, естественной является двоичная арифметика. Поэтому в машинной программе следует ожидать применения алгоритма преобразования умножением для преобразования чисел из десятичной в двоичную систему и алгоритма преобразования делением для преобразования чисел из двоичной системы в десятичную, что совершенно противоположно тому, что делалось в рассмотренных выше примерах.

2.5. ШЕСТНАДЦАТЕРИЧНЫЕ ЧИСЛА

ЕЩЕ НЕМНОГО О ПРЕДСТАВЛЕНИИ ЧИСЕЛ

Можно заметить, что чем меньше основание системы счисления, тем меньше возможных значений у каждого разряда числа: 10 значений в десятичной системе, 6 — в системе с основанием 6,и 2 — в двоичной системе. Чем меньше значений может иметь каждый разряд, тем меньше информации несут разряды числа. Как следствие этого, для представления чисел в системе с основанием 6 требуется больше разрядов, чем для представления тех же чисел в десятичной системе. Например, число 10000 в системе с основанием 6 эквивалентно числу 1296 в десятичной системе. Ситауция еще больше усложняется для двоичных чисел. Например, число 71230 в десятичной системе представляется как 10001011000111110 в двоичной системе, т.е. число разрядов двоичного представления числа более чем в 3 раза превышает число разрядов его десятичного представления. Обычно соотношение между разрядами двоичного и десятичного представления числа равно $3^1/_3$.

Один двоичный разряд вмещает наименьшее возможное количество дискретной информации и носит название *бит* (сокращение от binary digit – двоичный разряд). Запись двоичных чисел может быть такой длинной, что человеку неудобно их использовать. Например, семизначный телефонный номер при переводе в двоичную форму будет занимать примерно 23 разряда или бита. Кто из нас способен запомнить без ошибки свой собственный телефонный номер, состоящий из цепочки двадцати трех нулей и единиц? Даже профессиональные программисты с многолетней практикой не всегда в ладах с большими двоичными числами. Как в таком случае возможно общение с машиной?

ШЕСТНАДЦАТЕРИЧНОЕ ПРЕДСТАВЛЕНИЕ ЧИСЕЛ

Для кратной записи двоичной информации может использоваться *шестнадцатеричное кодирование.* При шестнадцатеричном кодировании двоичное число разбивается на группы по четыре двоичных разряда, начиная справа. Для двоичного числа 10001011000111110 такое разбиение проводится следующим образом:

0001 0001 0110 0011 1110

Отметим, что исходное число содержит 17 битов, а число 17 не кратно 4. Поэтому, чтобы в крайней левой группе было 4 бита, необходимо слева к числу приписать три нуля, при этом значение числа не изменится.

Рассмотрим теперь каждую группу из четырех битов как четырехбитовое двоичное число. В четырех битах может быть образовано 2^4, или 16, двоичных комбинаций. Таким образом, для обозначения 16 возможных комбинаций нужен набор из 16 простых символов. Как правило, для этого используются 10 цифр десятичной системы счисления и буквы латинского алфавита от A до F. Все эти символы являются *шестнадцатеричными цифрами.*

Заключительная операция состоит в замене каждой группы, состоящей из четырех двоичных цифр, эквивалентной шестнадцатеричной цифрой (рис. 2.3). Выполнение замены для приведенной двоичной последовательности в результате дает следующее:

0001	0001	0110	0011	1110	Двоичное число
1	1	6	3	E	Шестнадцатеричное представление

Таким образом, 1163E является шестнадцатеричным представлением данной двоичной последовательности.

Было бы неверно думать, что шестнадцатеричное представление это только код, который применяется для более простого представления двоичных чисел. Шестнадцатеричное представление чисел по праву образует систему счисления, а именно систему с основанием 16. Основная трудность, связанная с представлением шестнадцатеричных чисел, да и вообще чисел в любой системе счисления с основанием, превышающим 10, заключа-

Двоичная комбинация	Шестнадцатеричное представление
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Рис. 2.3. Шестнадцатеричное представление

ется в необходимости использования более 10 числовых символов для обозначения цифр. В результате получаются числа, которые не похожи на числа, например 1163E. Однако вспомним основное правило счета: счет осуществляется последовательным перебором всех цифр до тех пор, пока не будет достигнуто значение старшей цифры, после чего выполняется перенос единицы в следующий разряд. В шестнадцатеричной системе значением

Таблица 2.3

Десятичные, двоичные и шестнадцатеричные представления

Десятичные числа	Двоичные числа	Шестнадцатеричные числа	Десятичные числа	Двоичные числа	Шестнадцатеричные числа	Десятичные числа	Двоичные числа	Шестнадцатеричные числа	Десятичные числа	Двоичные числа	Шестнадцатеричные числа
1	1	1	26	11010	1A	51	110011	33	76	1001100	4C
2	10	2	27	11011	1B	52	110100	34	77	1001101	4D
3	11	3	28	11100	1C	53	110101	35	78	1001110	4E
4	100	4	29	11101	1D	54	110110	36	79	1001111	4F
5	101	5	30	11110	1E	55	110111	37	80	1010000	50
6	110	6	31	11111	1F	56	111000	38	81	1010001	51
7	111	7	32	100000	20	57	111001	39	82	1010010	52
8	1000	8	33	100001	21	58	111010	3A	83	1010011	53
9	1001	9	34	100010	22	59	111011	3B	84	1010100	54
10	1010	A	35	100011	23	60	111100	3C	85	1010101	55
11	1011	B	36	100100	24	61	111101	3D	86	1010110	56
12	1100	C	37	100101	25	62	111110	3E	87	1010111	57
13	1101	D	38	100110	26	63	111111	3F	88	1011000	58
14	1110	E	39	100111	27	64	1000000	40	89	1011001	59
15	1111	F	40	101000	28	65	1000001	41	90	1011010	5A
16	10000	10	41	101001	29	66	1000010	42	91	1011011	5B
17	10001	11	42	101010	2A	67	1000011	43	92	1011100	5C
18	10010	12	43	101011	2B	68	1000100	44	93	1011101	5D
19	10011	13	44	101100	2C	69	1000101	45	94	1011110	5E
20	10100	14	45	101101	2D	70	1000110	46	95	1011111	5F
21	10101	15	46	101110	2E	71	1000111	47	96	1100000	60
22	10110	16	47	101111	2F	72	1001000	48	97	1100001	61
23	10111	17	48	110000	30	73	1001001	49	98	1100010	62
24	11000	18	49	110001	31	74	1001010	4A	99	1100011	63
25	11011	19	50	110010	32	75	1001011	4B	100	1100100	64

старшей цифры является десятичное число 15; оно обозначается буквой F. Поэтому в шестнадцатеричной системе за числом F следует число 10, за числом FF – 100, за числом B7CFFF – B7D000. В табл. 2.3 приведены двоичные и шестнадцатеричные числа, эквивалентные десятичным числам от 1 до 100. Обратите внимание на соответствие между двоичными и шестнадцатеричными числами.

В языке ассемблера, в дампах памяти и файлов для упрощения записи чисел, имеющих в ЭВМ семейства VAX двоичное представление, широко используется шестнадцатеричное представление. Очевидно, что число 6EA7 не может быть десятичным, поэтому можно с уверенностью полагать, что для ЭВМ VAX это число является шестнадцатеричным. Но, например, число 1734 может интерпретироваться и как десятичное, и как шестнадцатеричное. Чтобы избежать недоразумений, шестнадцатеричные числа в программах на языке ассемблера ЭВМ VAX сопровождаются префиксом ^X. Числа без префикса ^X обычно рассматриваются как десятичные. В соответствии с этим далее там, где необходимо избежать разночтения, для обозначения шестнадцатеричных чисел будет применяться префикс ^X. Таким образом, если два предыдущих числа надо было обозначить как шестнадцатеричные, то их следовало бы записать как ^X6AE7 и ^X1734.

СЛОЖЕНИЕ И ВЫЧИТАНИЕ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

Во многих случаях бывает полезно сложить или вычесть шестнадцатеричные числа вручную. Одним из способов сложения является следующий: шестнадцатеричные цифры в каждом разряде переводятся в десятичные числа, выполняется сложение десятичных чисел и сумма десятичных чисел снова приводится к шестнадцатеричному виду. Рассмотрим следующий пример:

```
 C 8 A
+A B 3
-----
```

Самый правый разряд (разряд единиц) содержит ^XA и ^X3. Преобразуя сумму ^XA + + ^X3 в десятичную систему, получаем 10+3, или 13. Преобразуя число 13 обратно в шестнадцатеричную систему, находим ^XD. Это значит, что сумма ^XA и ^X3 равна ^XD без переноса в следующий разряд. На этом этапе сложения имеем:

```
  0      (перенос)
 C 8 A
+A B 3
-----
    D
```

В среднем разряде или в разряде "десятков" (десятичное число 16), сумма ^X8 + + ^XB, равна сумме 8+11 в десятичной системе, или числу 19. Преобразуя число 19 в шестнадцатеричную систему, получаем 16+3, или ^X13. Таким образом, сумма ^X8 и ^XB равна ^X3 с переносом в следующий старший разряд. (Перенос осуществляется всегда, когда десятичная сумма больше или равна 16.) На этом этапе сложения имеем:

```
 1 0      (перенос)
 C 8 A
+A B 3
-----
   3 D
```

Наконец, в самом левом разряде или в разряде "сотен" (десятичное число 256) сложим ^X1 + ^XC + ^XA, что равняется сумме 1+12+10 в десятичной системе, или 23. Так как число 23 равно 16+7, оно равно ^X17 или ^X7 с переносом в следующий старший разряд. Окончательный результат сложения следующий:


```
1 1 0      (перенос)
  C 8 A
+ A B 3
-------
1 7 3 D
```

Аналогично выполняется и вычитание. Вычитание чисел

```
  C 8 A
-A B 3
-----
```

начинается с самого правого разряда или разряда единиц. Разность ^XA – ^X3 равна разности 10 – 3, или 7, в десятичной системе, что соответствует ^X7. (Поскольку 10 больше 3, заем из старшего разряда не требуется.) Выполнив это вычитание, получим:

```
   0      (заем)
 C 8 A
- A B 3
-----
     7
```

В среднем разряде или в разряде "десятков" (десятичное число 16), разность ^X8 – – ^XB равна разности чисел 8–11 в десятичной системе. Поскольку 8 меньше 11, необходимо занять единицу (16 десятичное) из следующего старшего разряда. Таким образом, разность ^X8 – ^XB равна результату выражения 16 + 8 – 11, или числу 13 с заемом. В шестнадцатеричной системе это соответствует ^XD с заемом. На этом этапе вычитания имеем

```
-1 0      (заем)
 C 8 A
-A B 3
-----
   D 7
```

И наконец, в самом левом разряде или в разряде "сотен" (десятичное число 256) вычисляется выражение ^XC – ^XA – ^X1, где ^X1 – значение заема. Окончательный результат вычитания

```
-1 0      (заем)
 C 8 A
-A B 3
-----
 1 D 7
```

Правильность вычитания можно проверить, убедившись, что сумма ^XAB3 и ^X1D7 равна ^XC8A.

УПРАЖНЕНИЯ 2.1

1. Воспользуйтесь информацией, приведенной в энциклопедиях и словарях, и опишите три известные системы счисления, отличные от римской и арабской. Сравните эти системы:
   а) по легкости освоения;
   б) по применимости для вычислений;
   в) по удобству представления больших чисел.
2. Продолжите счет шестнадцатеричных и двоичных чисел, представленных в табл. 2.3, до числа 200 (десятичного).
3. Сложите приведенные ниже двоичные числа:

```
а)     101|      б)     110|      в)     101|
        +11|             +101|             +101|
г)   10111|      д)   11011|      е)   11101|
      +1010|             +1001|             +101|
ж) 1011011|      з) 110101101|      и) 110101100011|
    +101101|         + 10110010|         +101100011010|
```

4. Выполните вычитание двоичных чисел, используя пары чисел из упр. 3.
5. Преобразуйте следующие двоичные числа в десятичные:

а) 101; б) 11010;
в) 111010; г) 101110;
д) 110011; е) 1011101;
ж) 1100011; з) 1101111;
и) 11100101; к) 1110101011;
л) 101110100001; м) 11001010101110.

6. Двоичные числа из упр. 5 преобразуйте в шестнадцатеричные.
7. Сложите приведенные ниже шестнадцатеричные числа:

а) 7F б) E3 в) E3
+32 +1A +AA
г) A7B д) 5E4 е) E3B
+82 +39D +C8D
ж) A5B9 з) 48FD и) 2B89
+276 +3CFF +FFF

8. Выполните вычитание шестнадцатеричных чисел, используя пары чисел из п. 7.
9. Преобразуйте следующие шестнадцатеричные числа в десятичные:

а) ^XA; б) ^X10; в) ^X100;
г) ^X1F; д) ^X2A; е) ^X45;
ж) ^XA4; з) ^XC8; и) ^XDA;
к) ^X1374; л) ^X1A3D; м) ^XF3DE.

10. Преобразуйте следующие десятичные числа в двоичные:

а) 10; б) 15; в) 24;
г) 37; д) 42; е) 51;
ж) 85; з) 97; и) 128;
к) 1000; л) 3555; м) 5999.

11. Преобразуйте десятичные числа из упр. 10 в шестнадцатеричные. Используйте в ответах обозначение ^X.

*12. Покажите, как выполняется счет от 1 до числа, эквивалентного десятичному числу 200 в системе счисления с основанием 7. Как в этой системе можно отличить четные числа от нечетных? Действует ли здесь такое же простое правило, как в десятичной системе?

2.6. ДВОИЧНАЯ АРИФМЕТИКА В ДОПОЛНИТЕЛЬНЫХ КОДАХ

ОСОБЕННОСТИ ПРЕДСТАВЛЕНИЯ ЧИСЕЛ В ВЫЧИСЛИТЕЛЬНЫХ МАШИНАХ

До сих пор при изложении материала предполагалось, что на длину чисел не накладывается никаких ограничений. Однако в вычислительных машинах арифметические операции в основном выполняются с помощью устройств, называемых *регистрами*. *Регистр* – это устройство, в котором содержится представление числа. Наглядным примером регистра может служить автомобильный одометр – счетчик пройденного расстояния. Шкала счетчика составлена из колесиков с нанесенными по окружности цифрами. При вращении колесиков на шкале может быть отображено любое число в диапазоне 0 – 99999, в данном случае расстояние, выраженное в милях. Важно отметить наличие фиксированной верхней границы диапазона представляемых чисел. В вычислительных машинах большинство регистров состоят из строго определенного числа запоминающих элементов, и поэтому имеется фиксированная верхняя граница диапазона, зависящая от длины числа, которое может быть представлено в регистре. Например, в ЭВМ семейства VAX большинство операций ограничивается действиями, выполняемыми над числами длиной 8, 16 или 32 двоичных разрядов (битов). Из-за таких ограничений могут быть получены странные результаты, например когда одометр старого автомобиля, прошедшего более 100 тыс. миль, показывает небольшое расстояние.

00000	0	01000	8	10000	16	11000	24
00001	1	01001	9	10001	17	11001	25
00010	2	01010	10	10010	18	11010	26
00011	3	01011	11	10011	19	11011	27
00100	4	01100	12	10100	20	11100	28
00101	5	01101	13	10101	21	11101	29
00110	6	01110	14	10110	22	11110	30
00111	7	01111	15	10111	23	11111	31

Рис. 2.4. Пятибитовые числа без знака

11101	=	29	11101	=	29	11101	=	29
+ 00111	=	7	+ 01000	=	8	+ 01001	=	9
00100	=	4	00101	=	5	00110	=	6

Рис. 2.5. Сложение в пятиразрядном регистре

В качестве примера рассмотрим небольшую вычислительную машину с пятиразрядными двоичными регистрами. В пяти разрядах можно составить 2^5, или 32, двоичные комбинации. Поэтому такой регистр можно использовать для счета от 0 до 31 в десятичной системе или от 00000 до 11111 – в двоичной. Так как в счете участвуют только положительные целые числа, то такая интерпретация содержимого регистра называется *беззнаковой*, а такие числа называются *числами без знака*. Любая попытка использовать в счете числа больше 31 приведет к циклическому обращению содержимого регистра в 0 и снова к отсчету с нуля, что даст неверный результат, как в случае с одометром автомобиля, прошедшего более 99 999 миль. Эта ошибка называется *переполнением числа без знака*. На рис. 2.4 показаны все 32 5-битовых числа и их беззнаковая десятичная интерпретация.

Однако на самом деле оказывается, что свойство переполнения может быть использовано для представления отрицательных целых чисел. Рассмотрим опять ЭВМ с пятиразрядными двоичными регистрами. Посмотрим, что произойдет при сложении двоичного числа 11101 (десятичное число 29) с различными числами, такими как 7, 8 и 9. Сложение этих чисел в двоичной системе показано на рис. 2.5. Обратите внимание на то, что, так как мы ограничены использованием пяти разрядных двоичных регистров, при каждом сложении в ответе теряется разряд переноса.

ОТРИЦАТЕЛЬНЫЕ ЧИСЛА В ДОПОЛНИТЕЛЬНОМ КОДЕ

Исследуя результаты сложения, представленные на рис. 2.5, можно видеть, что при сложении двоичного числа 11101 с двоичным представлением любого из чисел 7, 8 или 9 полученная сумма меньше каждого второго слагаемого на 3, как если бы выполнялось вычитание числа 3. То же самое наблюдается в примерах с другими числами. В итоге при работе с 5-битовыми числами двоичное число 11101 можно рассматривать как десятичное число –3. Аналогично двоичное число 11111 при сложении подобно десятичному числу –1, двоичное 11110 – десятичному числу –2, и т.д.

Такой способ представления *чисел со знаком* называется системой представления чисел в двоичном *дополнительном коде* или поразрядным дополнением до двух. Название "дополнение до двух" дано по той причине, что отрицательное число получается при вычитании заданного числа из числа, которое является степенью основания системы счисления (2) и не помещается в разрядной сетке регистра. Например,

```
  100000
- 00011  =  3
-------
  11101  = -3
```

Другой способ вычисления дополнительного кода числа состоит в замене всех нулей в двоичном представлении числа на единицы, а единиц на нули, после чего к числу прибавляется 1. Например,

00011 равно 3
11100 замена 0 на 1 и наоборот
11101 после прибавления 1 получается –3

На рис. 2.6 приведены все 32 возможных 5-битовых числа и их десятичная интерпретация как чисел со знаком. Самый левый разряд используется для обозначения знака числа: 1 обозначает отрицательные, а 0 – положительные числа. Это означает, что числа, подобные 11101, которые можно интерпретировать как большие числа без знака, на самом деле являются отрицательными числами в дополнительном коде. Из рис. 2.6 видно, что в пяти битах возможно представить в дополнительном коде отрицательные числа от –1 до –16 и положительные числа от 0 до 15. Заметьте, что имеется двоичное представление числа –16, но нет представления числа +16. Отсутствие симметрии обусловлено тем, что представление числа 0 относится к положительным числам. Любая попытка породить число меньше –16 или больше +15 в таком представлении приводит к ошибке, называемой *переполнением числа со знаком.*

10000	–16	11000	–8	00000	0	01000	8
10001	–15	11001	–7	00001	1	01001	9
10010	–14	11010	–6	00010	2	01010	10
10011	–13	11011	–5	00011	3	01011	11
10100	–12	11100	–4	00100	4	01100	12
10101	–11	11101	–3	00101	5	01101	13
10110	–10	11110	–2	00110	6	01110	14
10111	– 9	11111	–1	00111	7	01111	15

Рис. 2.6. Пятибитовые числа в дополнительном коде

Важно понять, что при выполнении арифметических операций над числами и со знаком и без знака действуют одни и те же правила сложения. Различие заключается в интерпретации числа. Например, рассмотрим следующее сложение:

```
 11000
+00101
-------
 11101
```

При интерпретации этих чисел как чисел без знака сложение эквивалентно сложению десятичных чисел 24 + 5 = 29. При сложении их как чисел со знаком эквивалентным является –8 + 5 = –3.

2.7. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ В ЭВМ СЕМЕЙСТВА VAX

БАЙТЫ, СЛОВА И ДЛИННЫЕ СЛОВА

Как отмечалось выше, большинство операций в ЭВМ семейства VAX выполняются над группами из 8, 16 или 32 двоичных разрядов (или битов). Группа из 8 битов называется *байтом* (byte), группа из 16 битов – словом (word), а группа из 32 битов – длинным словом (longword).

Информация, содержащаяся в байте, может быть представлена непосредственно в виде 8 двоичных цифр, например

00111010

Однако обычно информация в байте представляется в виде двух шестнадцатеричных цифр. Содержимое рассмотренного выше байта в шестнадцатеричном виде представляется так:

3A

Обратите внимание, что двоичное и шестнадцатеричное представления информации, содержащейся в байте, являются всего лишь различными способами ее представления. На самом деле в ЭВМ байт представляется в двоичном виде. Для человека удобным является шестнадцатеричное представление.

В действительности слово "байт", обозначающее небольшое количество (кусочек) битов, возникло в конце 50-х годов как каламбур. В слове bite (кусок) вторая буква была заменена на у, поскольку легко спутать слова bit и bite в печатном виде. Продолжая каламбур, группу из четырех битов (полубайт) иногда называют словом nibble (огрызок) или даже nybble.

В восьми битах можно составить только 2^8, или 256, двоичных комбинаций. Если 8-битовый байт используется для представления числа без знака, то возможно представить целые числа только от 0 до 255 (десятичные числа). Диапазон представления чисел со знаком ограничен от -128 до +127.

Чтобы избежать подобных ограничений, в ЭВМ семейства VAX для хранения бо́льшего количества информации используется 2, 4, 8 или даже 16 байтов. В ЭВМ семейства VAX информация, содержащаяся в двух байтах, называется *словом.* Так как каждый байт состоит из 8 битов (две шестнадцатеричные цифры), то слово состоит из 16 битов (четыре шестнадцатеричные цифры) информации. Число ^X1A2B в двоичном виде можно представить так:

```
0001  1010  0010  1011
  1     A     2     B
```

Следующей, более крупной единицей информации является *длинное слово,* состоящее из двух слов, или четырех байтов. Так как содержимое каждого байта (8 битов) может быть представлено двумя шестнадцатеричными цифрами, то содержимое длинного слова (32 бита) представляется как 8 шестнадцатеричных цифр. Длинное слово ^X1A2B3C4D можно представить в двоичном виде так:

```
0001  1010  0010  1011  0011  1100  0100  1101
  1     A     2     B     3     C     4     D
```

Еще более крупными единицами информации являются *квадраслово* и *октаслово.* В соответствии со своим названием квадраслово состоит из четырех слов, или 8 байтов. Квадраслово представляется 16 шестнадцатеричными цифрами (64 бита). Октаслово состоит из восьми слов, или 16 байтов (128 битов), а его содержимое может быть представлено 32 шестнадцатеричными цифрами. В данной книге чаще всего будут использоваться такие единицы информации, как байт и длинное слово. В табл. 2.4 приведены различные единицы информации, используемые в ЭВМ семейства VAX.

Почти на всех современных вычислительных машинах понятие "байт" служит для обозначения 8 битов информации. Однако количество битов в слове зависит от производителя и конкретной модели ЭВМ. Например, в ЭВМ DECsystem10 и DECSYSTEM 20 слово составляют 36 битов информации, тогда как в больших ЭВМ фирмы IBM, как правило, слово – это 32 бита информации. Для совместимости ЭВМ семейства VAX с появившимся ранее семейством ЭВМ PDP-11, описанным в гл. 1, длина слова была определена как 16 битов.

Как показывают предыдущие примеры, шестнадцатеричное представление содержимого байтов, слов и длинных слов намного компактнее, чем двоичное представление. Поэтому при описании операций ЭВМ семейства VAX, как правило, будет использоваться шест-

Таблица 2.4

Единицы информации на ЭВМ семейства VAX

Формат данных	Число			
	битов	шестнадцатеричных цифр	байтов	слов
Байт (byte)	8	2	1	–
Слово (word)	16	4	2	1
Длинное слово (longword)	32	8	4	2
Квадраслово (quadword)	64	16	8	4
Октаслово (octaword)	128	32	16	8

надцатеричное представление. Это возможно потому, что во многих случаях представление информации в шестнадцатеричном и двоичном виде приводит к эквивалентным результатам. Ниже дан пример сложения двух байтов.

Двоичный код	Шестнадцатеричный код
0010 1110	2E
+ 0100 1111	+ 4F
0111 1101	7D

Суммой содержимого двух байтов, представленных в шестнадцатеричном виде ^X2E и ^X4F, является ^X7D, что в принципе идентично шестнадцатеричному представлению результата, полученному при сложении двоичных чисел.

В дальнейшем шестнадцатеричное представление информации будет использоваться почти всегда, так как есть стремление представить ЭВМ семейства VAX как ЭВМ, работающие в шестнадцатеричной системе счисления, а не в двоичной. Однако это не всегда нужно, и, как мы увидим далее, имеются некоторые операции, например логические (см. разд. 2.8), действие которых очень легко понять, рассматривая их фактическое двоичное представление. Но все-таки для подавляющего большинства операций ЭВМ семейства VAX вполне приемлемо их представление в намного более компактном шестнадцатеричном виде.

ПРЕДСТАВЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ В ФОРМАТЕ БАЙТА

Для представления целых чисел со знаком и без знака в байте могут использоваться 8 битов. В табл. 2.5 показана интерпретация байтов, представленных в шестнадцатеричном виде от ^X00 до ^XFF, как чисел без знака и со знаком. Как видно из таблицы, числа без знака могут быть представлены в диапазоне от 0 – 255, где код ^X00 является представлением числа 0, а код ^XFF – представлением числа 255. Для представления чисел со знаком применяется дополнительный код. Числа со знаком представляются в диапазоне от – 128 до +127. Заметим, что код ^X80 представляет число – 128, код ^XFF – – число – 1, код ^X00 – число 0, а код ^X7F – число +127.

По отношению к 5-битовым числам было отмечено, что для чисел со знаком и без знака применимы одни и те же правила сложения. Это же справедливо и по отношению к 8-битовым числам со знаком и без знака. Различие состоит только в способе интерпретации этих чисел. Например, при сложении ^XFC с ^X02 результатом является ^XFE. В случае беззнаковой интерпретации это соответствует сложению чисел 252 и 2, что в сумме дает 254. При интерпретации этих кодов как представлений чисел со знаком приведенный пример соответствует сложению чисел –4 и 2, а полученный результат будет равен –2.

Таблица 2.5

Представление чисел со знаком и без знака в формате байта

Шестнадцатеричное представление	Интерпретация как число без знака	Интерпретация как число со знаком
00	0	0
01	1	1
02	2	2
03	3	3
04	4	4
05	5	5
.	.	.
.	.	.
.	.	.
7C	124	124
7D	125	125
7E	126	126
7F	127	127
80	128	– 128
81	129	– 127
82	130	– 126
83	131	– 125
.	.	.
.	.	.
.	.	.
FC	252	– 4
FD	253	– 3
FE	254	– 2
FF	255	– 1

Ошибки переполнения возможны при любой интерпретации. Например, суммой ^XFF и ^X03 является ^X02. (В обычном случае, при сложении чисел без знака, правильным результатом была бы сумма ^X102, но это число не поместится в разрядной сетке байта.) Для чисел со знаком этот результат корректен, так как сумма чисел – 1 и 3 равна 2. Однако для чисел без знака он ошибочен, поскольку сумма чисел 255 и 3 не равна 2. В данном случае произошло *переполнение числа без знака.* Аналогично суммой ^X7E и ^X04 является ^X82. При интерпретации этих кодов как чисел без знака полученный результат будет правильным: 126 + 4 = 130, а при интерпретации их как чисел со знаком результат неверен: 126 + 4 = – 126. В этой ситуации произошло *переполнение числа со знаком.* Очевидно, что возникновение ситуации переполнения любого рода служит признаком того, что при выполнении арифметической операции мог быть получен неверный результат. В гл. 6 объясняется, как осуществляется проверка переполнения.

ПРЕДСТАВЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ В ФОРМАТЕ СЛОВА И ДЛИННОГО СЛОВА

Кроме целых чисел в формате байта ЭВМ семейства VAX могут работать с целыми числами в формате 16-битового слова, 32-битового длинного слова и отчасти – с целыми числами в формате 64-битового квадраслова. Это позволяет выполнять арифметические операции над числами с более широким диапазоном значений. Например, при представлении целых чисел в форматах байта, слова и длинного слова значения целых чисел без знака находятся в следующих диапазонах:

	Байт	Слово	Длинное слово
Нуль	^X00	^X0000	^X00000000
Максимальное значение	^XFF, или 255	^XFFFF, или 65 535	^XFFFFFFFF, или 4 294 967 295

Ниже приводятся граничные значения представления целых чисел со знаком в формате байта, слова и длинного слова, а также показано представление чисел – 2, – 1, 0, +1 и +2 в этих форматах.

	Байт	Слово	Длинное слово
Минимальное значение	^X80, или –128	^X8000, или –32 768	^X80000000, или –2 147 483 648
Минимальное значение плюс единица	^X81, или –127	^X8001, или –32 767	^X80000001, или –2 147 483 647
–2	^XFE	^XFFFE	^XFFFFFFFE
–1	^XFF	^XFFFF	^XFFFFFFFF
0	^X00	^X0000	^X00000000
+1	^X01	^X0001	^X00000001
+2	^X02	^X0002	^X00000002
Максимальное значение минус единица	^X7E, или +126	^X7FFE, или +32 766	^X7FFFFFFE, или +2 147 483 646
Максимальное значение	^X7F, или +127	^X7FFF, или +32 767	^X7FFFFFFF, или +2 147 483 647

Отсюда видно, что представление целых чисел в формате слова и длинного слова подобно представлению в формате байта, за исключением того, что расширение разрядной сетки позволяет представить числа в более широком диапазоне значений. И точно так же, как в случае чисел в формате байта, при выполнении арифметических операций над целыми числами в формате слова и длинного слова может произойти переполнение числа со знаком или переполнение числа без знака.

2.8. БУЛЕВА ЛОГИКА

ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ И ОПЕРАЦИИ

В XIX в. английский математик Джордж Буль разработал алгебраические методы выполнения операций над логическими значениями "истина" (true) и "ложь" (false). При работе с вычислительными машинами вполне пригодно иногда интерпретировать двоичные значения 1 и 0 как *логические значения* "истина" и "ложь".

Для обработки логических значений необходимо иметь логические операции. Основными логическими операциями являются операции **И** (AND), **ИЛИ** (OR) и **НЕ** (NOT). Операции **И** и **ИЛИ** объединяют истинностные значения двух высказываний, почти так же, как в естественном языке. Например, высказывание "А **И** В" истинно тогда, и только, когда истинны оба высказывания "А" и "В". Аналогично высказывание "А **ИЛИ** В" истинно тогда, когда истинно одно из высказываний "А" или "В", или истинны оба высказывания "А" и "В". Операция **НЕ** изменяет истинностное значение высказывания на противоположное. Таким образом, высказывание "**НЕ** А" истинно тогда, когда высказывание "А" ложно, а высказывание "**НЕ** А" ложно тогда, когда высказывание "А" истинно. На рис. 2.7 показаны все возможные комбинации истинностных значений для трех основных логических операций.

Таблицы логических операций, подобные приведенным на рис. 2.7, называются таблицами истинности и могут применяться для определения любой логической операции помимо основных трех операций, показанных выше. Например, еще одной обычно используе-

0	И	0 = 0	0ИЛИ0 = 0	НЕ 0 = 1
0	И	1 = 0	0ИЛИ1 = 1	НЕ 1 = 0
1	И	0 = 0	1ИЛИ0 = 1	
1	И	1 = 1	1ИЛИ1 = 1	

Рис. 2.7. Логические операции

мой логической операцией является операция **ИСКЛЮЧАЮЩЕЕ ИЛИ**. Она определяется так же, как и операция **ИЛИ**, но в случае истинности обоих операндов значение операции ложно. Ниже дана таблица истинности для операции **ИСКЛЮЧАЮЩЕЕ ИЛИ**:

0 **ИСКЛЮЧАЮЩЕЕ ИЛИ** 0 = 0
0 **ИСКЛЮЧАЮЩЕЕ ИЛИ** 1 = 1
1 **ИСКЛЮЧАЮЩЕЕ ИЛИ** 0 = 1
1 **ИСКЛЮЧАЮЩЕЕ ИЛИ** 1 = 0

Оказывается, что любая логическая операция может быть составлена из трех основных операций: **И**, **ИЛИ** и **НЕ**. Например,

А **ИСКЛЮЧАЮЩЕЕ ИЛИ** В = (А **ИЛИ** В) **И НЕ** (А **И** В)

ПОБИТОВАЯ ОБРАБОТКА

ЭВМ семейства VAX включают инструкции для выполнения логических операций над содержимым байтов, слов и длинных слов. Логические операции в ЭВМ семейства VAX реализованы таким образом, что выполняются поразрядно над соответствующими битами байта, слова и длинного слова. Например, с содержимым двух длинных слов может быть выполнена следующая операция:

```
      1011 0111 1100 1010 0000 0000 1111 1111
ИЛИ   0010 0101 0101 0011 0000 1111 0000 1111
      ---------------------------------------
      1011 0111 1101 1011 0000 1111 1111 1111
```

Отметим, что каждый бит результата является результатом логической операции **ИЛИ** над соответствующими битами двух операндов. Кроме чисто логических имеются операции, которые могут использовать логические операции для побитовой обработки. Следующий пример иллюстрирует применение логической операции **И** для маскирования (сброса в 0) самых левых 16 бит в длинном слове:

```
      1011 0111 0000 1111 1111 0000 1010 1111   длинное слово
И     0000 0000 0000 0000 1111 1111 1111 1111   маска
      ---------------------------------------
      0000 0000 0000 0000 1111 0000 1010 1111   результат
```

В заключение отметим, что при выполнении логической операции **НЕ** над содержимым длинного слова каждый бит длинного слова будет инвертирован, т.е. произойдет замена каждой 1 на 0, а каждого 0 – на 1.

2.9. ДРУГИЕ СПОСОБЫ КОДИРОВАНИЯ

ВОСЬМЕРИЧНОЕ ПРЕДСТАВЛЕНИЕ

Большинство вычислительных машин имеют количество информации, запоминаемое в двоичном виде в регистрах и кратное 8 битам, в том числе 8, 16, 32 и 64 бита. Так как такие числа делятся без остатка только на 1, 2, 4 и 8, то единственными возможными системами представления чисел, разделяющими содержимое регистров на равное число бит, могут быть двоичная система, система с основанием 4, шестнадцатеричная система и сис-

тема с основанием 256. Как уже обсуждалось, запись чисел в двоичной системе слишком длинная. Система с основанием 4 в этом смысле не намного лучше, а система с основанием 256 невероятно трудна для изучения. Остается шестнадцатеричная система; именно она и является общераспространенной для большинства ЭВМ нынешнего поколения.

Однако в вычислительных машинах предыдущего поколения широко применялись регистры с числом разрядов, кратным 6 битам. В итоге предпочтение обычно отдавалось системе счисления с основанием 8, иначе – *восьмеричной* системе счисления, так как в ней двоичные числа можно разделить на группы из трех битов. Среди ЭВМ, для которых использовалось восьмеричное представление двоичных чисел, были все основные ЭВМ фирмы DEC, предшествующие ЭВМ семейства VAX, ЭВМ фирмы CDC, а также большинство ЭВМ серии IBM 7000.

Чтобы преобразовать двоичное число в восьмеричное, необходимо разделить двоичное число на группы из трех битов, начиная справа. Слева к двоичному числу могут быть добавлены нули для гарантии того, что количество разрядов двоичного числа будет точно кратно 3. Затем каждая группа из трех двоичных цифр заменяется их числовым эквивалентом. Соответствие между двоичным и восьмеричным представлением чисел можно найти в верхней половине рис. 2.3, где рассматривалось преобразование шестнадцатеричных чисел. Например, двоичное число 1101110011100010 может быть преобразовано в восьмеричное следующим образом:

```
001  101  110  011  100  010
 1    5    6    3    4    2
```

Одно преимущество, которое имеет восьмеричная система перед шестнадцатеричной, состоит в том, что она проще для освоения и в ней легче выполнять арифметические действия вручную. Но из-за того, что используется разбиение на группы из трех битов, восьмеричное представление информации плохо сочетается со структурой байта, слова и длинного слова, принятых в ЭВМ семейства VAX. Тем не менее восьмеричная система представляет определенный интерес для пользователей ЭВМ семейства VAX вследствие их совместимости с предшествующим семейством ЭВМ PDP-11, описанным в гл. 1. Дело в том, что в документации для ЭВМ семейства PDP-11 в основном применялось восьмеричное представление информации, чтобы сохранить совместимость с другими разработками фирмы DEC, уже существовавшими к моменту выпуска PDP-11.

ПРЕДСТАВЛЕНИЕ ЧИСЕЛ В ОБРАТНОМ КОДЕ

Как описывалось выше, отрицательные числа в дополнительном коде могут быть получены с помощью инвертирования всех битов положительного числа с последующим прибавлением 1. Например, 8-битовое представление числа –3 получается следующим образом:

```
 00000011  = 3
 11111100  = НЕ 00000011
+        1
----------
 11111101  = –3 в дополнительном коде
```

В противоположность этому в некоторых ЭВМ отрицательные числа образуются простым инвертированием 0 и 1. В них для образования отрицательного числа выполняется логическая операция **НЕ**. Например, 8-битовым представлением числа –3 является **НЕ** 00000011 или 11111100. Такой способ представления чисел называется системой представления чисел *в обратном коде* или поразрядным дополнением до 1, так как отрицательное число может быть также получено при вычитании положительного числа из двоичного числа, во всех разрядах которого содержатся единицы. Например,

```
 11111111
–00000011  = 3
----------
 11111100  = –3 в обратном коде
```


Особенность системы представления чисел в обратном коде состоит в том, что в ней имеется два представления числа 0. В случае 8 битовых чисел этими представлениями являются 00000000 и его дополнение до единицы 11111111. Наличие двух представлений числа 0 требует от программистов особого внимания при проверке результатов на равенство нулю. При использовании системы представления чисел в обратном коде соответственно должны быть несколько модифицированы арифметические операции сложения и вычитания, умножения и деления. Так как арифметика в обратных кодах применяется лишь в немногих вычислительных машинах, то в данной книге эти вопросы рассматриваться не будут[1].

ДРУГИЕ СПОСОБЫ ПРЕДСТАВЛЕНИЯ ИНФОРМАЦИИ

Числа могут быть представлены и другими способами. Например, четыре бита могут использоваться для представления десятичной цифры, хотя в четырех битах может быть составлено и 16 различных комбинаций. Если шесть из этих возможных комбинаций рассматривать как "запрещенные", то оставшиеся десять "разрешенных" комбинаций в результате образуют 10-позиционный переключатель, который можно использовать для представления одной десятичной цифры. В этой системе последовательность двоичных цифр

```
0001  1000  1001  0010  0000  0100
```

представляет десятичное число 189204 (где код 0000 является представлением десятичной цифры 0, код 0001 представляет десятичную цифру 1, и т.д.). Такое представление чисел называется *двоично-десятичным* или *упакованным десятичным*. Еще одно представление чисел основывается на экспоненциальной форме записи чисел и называется представлением *действительных чисел* или *чисел с плавающей точкой*. Оно подобно экспоненциальной форме записи, применяемой для представления больших чисел в языках высокого уровня, таких как Фортран и Паскаль, а также в калькуляторах для научных расчетов (например, 3.84536E+08).

Так как особое внимание уделялось представлению чисел, в конечном счете у читателя может сложиться впечатление, что вычислительные машины в основном используются для выполнения арифметических вычислений. Но это неверно. Последовательности двоичных цифр могут применяться для представления любого фиксируемого физического события. Например, многие терминалы способны напечатать 95 разных символов (включая пробел). В семи битах информации может быть составлено 2^7, или 128, различных комбинаций. Таким образом, достаточно иметь семь битов для представления какого-либо одного из 95 печатных символов, при этом 128–95, или 33 комбинации, могут использоваться для представления служебных символов, т.е. как код специальных функциональных клавиш клавиатуры терминала, таких как RETURN (Возврат каретки) или TAB (Табуляция). Действительно, в гл. 8 такой 7-битовый код будет использоваться при обработке символьных строк. Способы представления символьной информации играют очень важную роль. Например, они применяются в процессе трансляции программы с языка высокого уровня или с языка ассемблера в машинный язык.

Еще пример: клавиатура стандартного пианино имеет 88 клавиш. Поэтому семь битов информации можно использовать для обозначения нажатия на конкретную клавишу. Конечно, могут потребоваться дополнительные биты для указания таких атрибутов, как время нажатия клавиши, скорость удара по клавише и продолжительность нажатия на клавишу. Применяя такие способы кодирования, вполне возможно писать программы для анализа музыки и даже музыкальные сочинения.

И наконец, последний пример. Возможно представлять изображения в виде битовых строк. Для этого на фотографию накладывается сетка-шаблон, которая может иметь 1000

[1] Замечательным примером вычислительных машин, в которых для представления чисел используется обратный код, являются ЭВМ CDC Cyber.

строк и 1000 столбцов. Для каждой из миллиона ячеек сетки измеряется интенсивность отраженного светового потока, а результат преобразуется в двоичное число. (Если различается 64 градации яркости, то интенсивность светового потока, отраженного от каждой ячейки, может быть преобразована в 6-битовое число, при этом двоичный код 000000 служит для представления уровня белого, код 111111 – уровня черного тона, а все другие комбинации служат для представления различных полутонов.) Теперь изображение преобразовано в такую форму, при которой оно может быть обработано ЭВМ. Подобные методы кодирования изображений обычно применяются при обработке на ЭВМ фотографий, полученных со спутников, некоторых видов рентгеновских и других медицинских снимков.

УПРАЖНЕНИЯ 2.2

1. Интерпретируя 64 возможные двоичные комбинации, которые могут быть образованы в шести битах, как представление чисел в дополнительном коде, приведите эквивалентные им десятичные числа со знаком.

2. Сложите следующие 8-битовые пары чисел в дополнительном коде с указанием всех переносов, в том числе и тех, что теряются из-за ограничения разрядной сетки в 8-разрядном регистре. В следующих примерах приведите для двоичных слагаемых и результата эквивалентные им десятичные числа со знаком:

а) 00000011
00000101

б) 00001011
11111100

в) 11111101
11111010

г) 11101100
11111111

д) 00100111
11011011

е) 10011101
00100101

3. Для каждой пары чисел из п. 2 выполните вычитание второго числа из первого. Укажите заемы, в том числе и те, которые теряются.

4. Преобразуйте в шестнадцатеричную систему представления числа из п. 2 и выполните сложение байтов в шестнадцатеричном виде. Укажите переносы, в том числе и те, которые теряются в случае переноса из старшего разряда числа.

5. Повторите п. 4, выполнив вместо сложения вычитание шестнадцатеричных чисел. Укажите заемы, включая потерянные.

6. Каждое из следующих длинных слов представляет число со знаком. Приведите соответствующие им десятичные числа.

Для некоторых из этих чисел при преобразовании основания системы счисления должен быть использован алгоритм преобразования умножением, описанный в гл. 2:

а) 0000000F;
б) FFFFFFF0;
в) 00000100;
г) FFFFFF00;
д) 00001000;
е) FFFFF000;
ж) 00ABCDEF;
з) FFABCDEF.

7. Преобразуйте содержимое каждого из приведенных ниже байтов, слов и длинных слов в шестнадцатеричную форму представления:

а) 0011 0111;
б) 1010 0001;
в) 0111 1100;
г) 1110 1111;
д) 0011 1001 1110 0101;
е) 1010 1011 1111 0010;
ж) 0011 1011 1110 1111 0101 1000 1101 0001;
з) 1101 1110 0110 1001 1000 1110 0101 1110.

8. Каждый из следующих байтов представляет собой число со знаком от –128 до +127.

Приведите соответствующие им десятичные числа (*подсказка*: для отрицательных чисел вычислите их дополнительный код):

а) 01;
б) FF;
в) 0F;
г) 1A;
д) FA;
е) A0;
ж) 7A;
з) 83.

9. Предполагая, что содержимое каждого байта из п. 8 является представлением числа без знака от 0 до 255, приведите эквивалентные им десятичные числа.

10. Для каждой пары двоичных чисел из п. 2 выполните логические операции **И**, **ИЛИ** и **ИСКЛЮЧАЮЩЕЕ ИЛИ**.

11. Покажите, как выполняется счет в восьмеричной системе от 0 до числа, эквивалентного десятичному числу 200.

12. Для двоичных чисел из п. 7 приведите эквивалентные им восьмеричные числа.

ГЛАВА 3. ПРОЦЕССОР И СТРУКТУРА ПАМЯТИ

3.1. ВВЕДЕНИЕ

Работа большинства ЭВМ сопряжена с передачей данных и управляющей информации между *процессором* и *памятью*. Обычно программируемая управляющая информация называется *машинным языком*. Для того чтобы выполнить некоторую программу, данная программа должна быть представлена на машинном языке используемой ЭВМ. С развитием вычислительной техники машинные языки стали весьма сложными и трудными для освоения программистом. Вследствие этого были созданы языки ассемблера и языки высокого уровня, которые затем транслируются на машинный язык. Среди различных языков, поддерживаемых вычислительными системами, наиболее близок к машинному языку язык ассемблера. Поэтому для должного понимания языка ассемблера необходимо понимание машинного языка, на который транслируется программа, написанная на языке ассемблера. В данной главе описываются основы организации ЭВМ семейства VAX, процессор, память, а также то, как осуществляется управление некоторыми основными операциями с помощью машинного языка.

3.2. ОРГАНИЗАЦИЯ ЭВМ

На рис. 3.1 показаны важнейшие части простой цифровой ЭВМ: *устройство ввода, устройство вывода, память* и *центральное устройство управления*, или *процессор*. *Устройство ввода* позволяет вводить информацию в ЭВМ. К простым устройствам ввода относятся кнопки или клавиши электронного калькулятора либо клавиатура, подобная клавиатуре пишущей машинки. *Устройство вывода* позволяет получить обратно результаты работы ЭВМ. В число простых устройств вывода входят цифровые индикаторы электронного калькулятора, подобные пишущей машинке печатающие устройства, а также экран видеотерминала. *Память* используется для запоминания информации. В общем случае память может быть представлена как множество *ячеек*, каждая из которых содержит некоторое число. (Кроме основной памяти большинство ЭВМ для хранения дополнительной информации имеет внешние запоминающие устройства, такие как магнитные диски.) Устройство ввода, устройство вывода и память соединяются электрическими проводниками с центральным устройством управления, называемым процессором. Процессор, посылая по этим проводникам электрические сигналы, может:

1) запросить устройство ввода, чтобы получить или *считать* информацию и сделать ее доступной для процессора или памяти;

2) запросить устройство вывода, чтобы *напечатать* определенную порцию информации;

3) запросить память, чтобы сохранить или *запомнить* определенную порцию информации в определенной ячейке памяти;



Рис. 3.1. Структура ЭВМ

4) запросить память, чтобы получить обратно или *выбрать* порцию информации, которая до этого была запомнена в определенной ячейке памяти.

В этой простой ЭВМ устройство ввода, устройство вывода и память являются пассивными устройствами. Они не выполняют никаких действий до тех пор, пока не получат указание процессора выполнить какую-либо операцию. Процессор представляет собой активное устройство, управляющее обменом информацией между ним самим и другими устройствами. Последовательность операций, выполняемых процессором, определяется множеством *инструкций*, которые образуют *машинную программу*. Работа программиста заключается в том, чтобы составить соответствующую последовательность инструкций, указывающих процессору, какие операции необходимо выполнить для решения конкретной задачи.

3.3. ОРГАНИЗАЦИЯ ПАМЯТИ ЭВМ СЕМЕЙСТВА VAX

ОСНОВНЫЕ ПОНЯТИЯ

Память ЭВМ семейства VAX можно рассматривать как набор *ячеек памяти*, каждая из которых содержит 8-битовый байт. Поэтому назовем содержимое такой ячейки *байтом памяти* или просто *байтом*. Байт в ячейке памяти обычно можно представить двумя шестнадцатеричными цифрами (от ^X00 до ^XFF).

Каждая ячейка памяти идентифицируется числом, называемым *адресом* ячейки памяти. Он позволяет однозначно идентифицировать ячейку. По аналогии адрес ячейки памяти можно представить себе как адрес дома на улице. И так же, как по адресу дома на улице можно найти или идентифицировать конкретное здание, адрес памяти позволяет найти или идентифицировать определенную ячейку памяти.

В ЭВМ семейства VAX адресом является 32-битовое двоичное целое число. Обычно 32-битовые адреса представляются восемью шестнадцатеричными цифрами. Например, предположим, что по адресу ^X00ABCDEF содержится байт ^X35. В шестнадцатеричном представлении это выглядит следующим образом:

Шестнадцатеричное содержимое	**Шестнадцатеричный адрес**
35	00ABCDEF

Здесь адрес показан справа, а содержимое показано слева потому, что таким образом располагает информацию ассемблер ЭВМ VAX. Причины этого будут выявлены позже. Обратите внимание, что с каждым байтом памяти связываются два числа – адрес (32-битовое число) и содержимое (8-битовое число).

Поскольку в 32 двоичных разрядах могут быть образованы 2^{32}, или 4 292 967 296, двоичные комбинации, то в программе для ЭВМ семейства VAX можно было бы адресовать до 4 294 967 296 различных байтов памяти (приблизительно 5 млрд. байтов, или 4 Гбайта). Первый байт памяти имеет адрес ^X00000000, а последний – адрес ^XFFFFFFFF. На рис. 3.2 показано возможное содержимое первых 11 байтов памяти.

Шестнадца-теричное содержимое	Шестнадца-теричный адрес
90	00000000
9F	00000001
34	00000002
12	00000003
00	00000004
00	00000005
9F	00000006
EF	00000007
CD	00000008
0B	00000009
00	0000000A

Рис. 3.2. Пример шестнадцатеричного представления содержимого памяти

Как было показано в гл. 2, ЭВМ семейства VAX кроме байтов работают со *словами*, *длинными словами* и *квадрасловами*. В памяти ЭВМ 16-битовое слово запоминается в двух смежных байтах. В следующем примере показано, как слово ^X1234 можно запомнить в байтах памяти с адресами ^X00000002 и ^X00000003.

Шестнадцатеричное содержимое	**Шестнадцатеричный адрес**
34	00000002
12	00000003

Отметим, что здесь представлено слово ^X1234, а не слово ^X3412. В ЭВМ семейства VAX младшие разряды слова, в данном случае имеющие шестнадцатеричное представление ^X34, запоминаются в байте с младшим адресом ^X00000002, а старшие разряды слова ^X12 – в байте со старшим адресом ^X00000003. При таком представлении слов байты слова нужно читать сверху вниз. Чтобы избежать этого, распечатки данных для ЭВМ семейства VAX обычно читаются справа налево. Поэтому слова в памяти часто представляются следующим образом:

Шестнадцатеричное содержимое	**Шестнадцатеричный адрес**
1234	00000002

Здесь указывается только один адрес, однако при этом понимается, что в байте памяти с адресом ^X00000002 содержится ^X34, а в байте с адресом ^X00000003 содержится ^X12. Для напоминания программистам о возможном расположении информации в нескольких байтах такие программы, как ассемблер, располагают адрес справа от содержимого.

В памяти ЭВМ 32-битовые длинные слова запоминаются в четырех смежных байтах. В следующем примере показано, как может быть запомнено в памяти длинное слово ^X000BCDEF, начиная с адреса ^X00000007.

Шестнадцатеричное содержимое	**Шестнадцатеричный адрес**
EF	00000007
CD	00000008
0B	00000009
00	0000000A

Отметим, что здесь снова необходимо читать байты длинного слова снизу вверх. Содержимым длинного слова, начинающегося с адреса ^X00000007, является ^X000BCDEF, а не

Шестнадцатеричное содержимое	Шестнадцатеричный адрес
90	00000000
9F	00000001
1234	00000002
0000	00000004
9F	00000006
000BCDEF	00000007

Рис. 3.3. Компактное представление содержимого памяти

^XEFCD0B00. Чтобы устранить это неудобство, длинные слова в памяти часто представляются в следующем виде:

Шестнадцатеричное содержимое	Шестнадцатеричный адрес
000BCDEF	00000007

Программист должен помнить, что в данном случае длинное слово в действительности занимает байты с адресами от ^X00000007 до ^X0000000A, а младшие разряды длинного слова ^XEF содержатся в байте с адресом ^X00000007.

Квадраслово (64 бита) запоминается в памяти в восьми смежных байтах. А еще бо́льшая единица информации – 128-битовое октаслово запоминается в памяти в шестнадцати смежных байтах.

Предположим, что в 11 байтах, показанных на рис. 3.2, представлены три байта, два слова и одно длинное слово в следующем порядке:

1. Байты располагаются по адресам ^X00000000, ^X00000001 и ^X00000006.
2. Слова начинаются с адресов ^X00000002 и ^X00000004.
3. Длинное слово начинается с адреса ^X00000007.

Содержимое памяти, показанное на рис. 3.2, может быть представлено и более компактно, как показано на рис. 3.3. Далее в основном будет использоваться такое представление. Однако важно понимать, что на рис. 3.2 и 3.3 изображена одна и та же информация, которая является просто различным представлением одного и того же содержимого памяти. Рассматривая рис. 3.3, можно предположить, что содержимое ^X9F байта памяти с адресом ^X00000006 будет обрабатываться как байт. Тем не менее вполне возможно, что в один момент времени программа будет обрабатывать содержимое данного байта памяти как байт, в другой момент времени – как часть слова, а в последующее время – как часть длинного слова. Не существует способа, который позволил бы, посмотрев на байт в памяти, определить, как он будет использоваться.

АДРЕС И СОДЕРЖИМОЕ ПАМЯТИ

Очень важно не путать адрес байта, слова и длинного слова с содержимым байта, слова и длинного слова. В частности, это может происходить при работе с длинными словами, так как длина адреса памяти и длина длинного слова совпадают и равны 32 бита. Например, ниже показаны длинные слова, расположенные в памяти, начиная с адреса ^X00001000.

Шестнадцатеричное содержимое	Шестнадцатеричный адрес
00001234	00001000
A3B4F303	00001004
00001000	00001008
00000000	0000100C
.....	00001010

Во второй строке начальным адресом длинного слова является ^X00001004, а его содержимым является ^XA3B4F303. Возможно изменить содержимое ячейки памяти, но нельзя изменить ее адрес. Иначе говоря, если содержимое ячейки памяти представляет собой переменную величину, то адрес ячейки памяти фиксирован.

Различие между адресом ячейки памяти и ее содержимым усложняет то обстоятельство, что содержимым одной ячейки памяти часто является адрес другой ячейки памяти. Например, содержимое длинного слова ^X00001000, расположенного по адресу ^X00001008, само может быть адресом ^X00001000. В этом случае говорят, что содержимое длинного слова с адресом ^X00001008 *указывает на* длинное слово ^X00001234 с адресом ^X00001000.

Использование таких указателей весьма распространено и крайне важно. Если программа должна выполнять обработку объектов, расположенных в памяти, то ей должны быть доступны адреса этих объектов и эти адреса вместе с машинной программой должны присутствовать в памяти. Более того, указатели и адреса составляют основу для построения многих структур данных. Путаница между адресом ячейки памяти и содержимым ячейки памяти представляет собой одну из наиболее общих ошибок при написании программ на языке ассемблера.

3.4. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОРА С ПАМЯТЬЮ

РАБОТА С ПАМЯТЬЮ. ОПЕРАЦИИ ВЫБОРКИ И ЗАПИСИ

Память ЭВМ работает под управлением процессора. Процессор может либо запросить память произвести выборку байта из определенной ячейки памяти, либо осуществить запись отдельного байта в определенную ячейку памяти. Операция выборки не изменяет, а операция записи изменяет содержимое указываемой ячейки памяти. Для того чтобы произвести выборку, процессор посылает устройству памяти адрес требуемой ячейки памяти, а устройство памяти отвечает на запрос процессора, посылая ему содержимое адресуемой ячейки. Например, воспользуемся данными, приведенными на рис. 3.3. Если процессор выдал запрос устройству памяти на выборку байта из ячейки памяти с адресом ^X00000006, то устройство памяти отвечает на запрос, посылая процессору число ^X9F. Содержимым ячейки памяти с адресом ^X00000006 останется число ^X9F.

Для выполнения операции записи процессор посылает устройству памяти адрес требуемой ячейки памяти вместе с числом, которое должно быть помещено в указываемую ячейку. Снова воспользуемся данными, приведенными на рис. 3.3. Предположим, что процессор запросил устройство памяти выполнить запись байта ^X1B в ячейку памяти с адресом ^X00000006. Старое содержимое ячейки памяти с адресом ^X00000006 будет потеряно или уничтожено, а ее новым содержимым будет ^X1B. При этом не останется никакой записи о том, что в ячейке памяти с адресом ^X00000006 когда-либо содержалось ^X9F. Если бы впоследствии процессор выдал устройству памяти запрос на выборку байта из ячейки памяти с адресом ^X00000006, то устройством памяти в ответ на запрос было бы передано новое содержимое этой ячейки ^X1B.

Операции выборки и записи, производимые над более крупными порциями информации, такими как слова и длинные слова, выполняются аналогично. Воспользуясь данными, приведенными на рис. 3.3, предположим, что процессор запрашивает устройство памяти, чтобы осуществить выборку содержимого *слова,* начинающегося с адреса ^X00000002. Устройство памяти отвечает на запрос, посылая процессору содержимое байта ^X34, расположенного по адресу ^X00000002, а также содержимое байта ^X12 с адресом ^X00000003. Процессор объединяет эти два байта, чтобы образовать слово ^X1234. Аналогично если процессор запрашивает устройство памяти для выполнения выборки длинного слова, начинающегося с адреса ^X00000007, то устройство памяти отвечает, посылая процессору содержимое байтов с адресами ^X00000007, ^X00000008, ^X00000009 и ^X0000000A. Процессор из этих четырех байтов формирует длинное слово ^X000BCDEF. Если бы процессор выдал запрос устройству памяти на выборку длинного слова, начинаю-

щего с адреса ^X00000004, то устройство памяти в ответ на запрос послало бы процессору четыре байта, расположенные в памяти по адресам с ^X00000004 до ^X00000007, которые после этого были бы собраны процессором в длинное слово ^XEF9F0000*.

Выборка и запись являются единственными операциями, которые может выполнять устройство памяти. Например, устройство памяти не может выполнить сложение. Если желательно сложить два числа, находящиеся в памяти, то оба эти числа должны быть выбраны из памяти и переданы процессору, а после сложения полученный результат снова должен быть записан в память.

ИНСТРУКЦИЯ СЛОЖЕНИЯ БАЙТОВ

Как уже упоминалось, процессор является активным устройством, управляющим передачей данных между ним самим и другими устройствами, входящими в состав ЭВМ, такими, как, например, память. В свою очередь, программист управляет процессором с помощью написанной им последовательности инструкций, называемой программой, которая выполняется процессором. Например, приведенная ниже инструкция заставит процессор выполнить сложение байта, расположенного в ячейке памяти с адресом ^X00001001, с байтом, содержащимся в ячейке памяти с адресом ^X00001003, а 8-битовый результат поместить в ячейку памяти с адресом ^X00001003.

```
ADDB2    ^X00001001, ^X00001003
```

Отметим, что эта инструкция не заставляет процессор выполнять сложение чисел ^X00001001 и ^X00001003, что в результате дает число ^X00002004. Вместо этого инструкция дает указание процессору сложить содержимое байтов памяти с адресами ^X00001001 и ^X00001003, а их сумму поместить в байт памяти с адресом ^X00001003. Эта инструкция выполняется процессором за четыре шага.

Шаг 1. Выбрать байт из ячейки памяти с адресом ^X00001001.
Шаг 2. Выбрать байт из ячейки памяти с адресом ^X00001003.
Шаг 3. Сложить байты, выбранные на шаге 1 и шаге 2.
Шаг 4. Запомнить полученную сумму в ячейке памяти с адресом ^X00001003.

Байты с адресами ^X00001001 и ^X00001003 называются *операндами*, так как они являются теми объектами, над которыми выполняется операция сложения по инструкции ADDB2.

ИНСТРУКЦИИ ВЫЧИТАНИЯ БАЙТОВ, ПЕРЕСЫЛКИ БАЙТОВ И ВОЗВРАТА УПРАВЛЕНИЯ

Инструкции SU BB2 (SUBtract Byte – вычесть байт, два операнда) и MOVB (MOVe Byte – переслать байт) подобны инструкции ADDB2 (ADD Byte – сложить байты, два операнда). Чтобы выполнить инструкцию.

```
SUBB2    ^X00001FFF, ^X00000500
```

процессор совершает такую последовательность действий:

Шаг 1. Выбирает байт из ячейки памяти с адресом ^X00001FFF.
Шаг 2. Выбирает байт из ячейки памяти с адресом X00000500.
Шаг 3. Вычитает байт, выбранный на шаге 1, из байта, выбранного на шаге 2.
Шаг 4. Запоминает полученную разность в ячейке памяти с адресом ^X00000500.

Для выполнения инструкции

```
MOVB     ^X00001000, ^X00001003
```

процессор совершает два следующих действия:

Шаг 1. Выбирает байт из ячейки памяти с адресом ^X00001000.
Шаг 2. Запоминает выбранный на шаге 1 байт в ячейке памяти с адресом ^X00001003.

Программа представляет собой просто последовательность инструкций. Например, в результате выполнения программы, приведенной на рис. 3.4, содержимое байта памяти с адресом ^X00001003 станет равным сумме байтов с адресами ^X00001000, ^X00001001 и ^X00001002.

```
ИНСТРУКЦИЯ 1     MOVB     ^X00001000, ^X00001003
ИНСТРУКЦИЯ 2     ADDB2    ^X00001001, ^X00001003
ИНСТРУКЦИЯ 3     ADDB2    ^X00001002, ^X00001003
ИНСТРУКЦИЯ 4     RET
```

Рис. 3.4. Пример простой программы

Приведенные на рис. 3.4 инструкции процессор выполняет последовательно одну за другой. По инструкции RET (сокращение от RETurn – возврат) управление процессором из текущей программы возвращается программе, инициировавшей выполнение текущей программы. Если текущая программа является основной программой, запущенной на выполнение из операционной системы VAX/VMS, то по инструкции RET управление процессором будет возвращено ОС VAX/VMS.

Например, предположим, что в памяти первоначально содержится следующая информация:

Содержимое	Адрес
1A	00001000
26	00001001
05	00001002
??	00001003

Два вопросительных знака означают, что не надо заботиться (или знать) о первоначальном содержимом байта с адресом ^X00001003. По первой инструкции программы, приведенной на рис. 3.4, произойдет пересылка значения ^X1A в байт с адресом ^X00001003; по второй инструкции выполнится сложение значения ^X26 с содержимым байта с адресом ^X00001003, после чего его содержимое изменится и станет равно ^X1A+^X26, или ^X40; по третьей инструкции сложение со значением ^X05 изменит содержимое байта с адресом ^X00001003, которое станет равно ^X40+^X05, или ^X45. Когда этой программой будет возвращено управление, содержимое памяти будет следующим:

Содержимое	Адрес
1A	00001000
26	00001001
05	00001002
45	00001003

3.5. ПРОГРАММЫ В МАШИННОМ КОДЕ

КОДЫ МАШИННОГО ЯЗЫКА

Теперь вполне резонно возникает вопрос: "Где в ЭВМ хранится программа?" В приведенном ранее описании было названо одно место, в котором может содержаться информация, – память. Поэтому память может быть использована для хранения программ. Однако так как в ячейках памяти информация может храниться только в двоичном виде, то конкретные инструкции процессора должны быть закодированы в двоичном

* В действительности для повышения быстродействия в ЭВМ семейства VAX между процессором и памятью осуществляется одновременная передача двух или четырех байтов в зависимости от модели ЭВМ. Однако наличие 16- и 32-битовых трактов передачи данных прозрачно для программиста, т.е. скрыто от него.

коде. В следующей таблице приведены коды операций инструкций MOVB, ADDB2, SU BB2 и RET; каждое двухзначное шестнадцатеричное число обозначает 8-битовый двоичный код операции:

Мнемоника	Код операции
MOVB	90
ADDB2	80
SUBB2	82
RET	04

Кроме кода операции в инструкциях ЭВМ VAX присутствуют *спецификаторы операндов*, которые указывают, как могут быть найдены операнды (в данном случае – пересылаемые, складываемые или вычитаемые байты). Простейший способ обозначить операнд – это задать его 32-битовый адрес (он уже применялся в данной главе). Такой способ нахождения операнда называется *абсолютной адресацией*. Для обозначения абсолютной адресации используется код или спецификатор операнда 9F.

В следующем примере в шестнадцатеричном представлении показано, как кодируется инструкция MOVB:

90 9F 00001000 9F 00001003

Код операции 90 определяет инструкцию MOVB. Спецификатор первого операнда 9F указывает, что будет задан 32-битовый адрес операнда, а ^X00001000 является этим адресом. Спецификатор второго операнда 9F указывает, что и второй операнд задается 32-битовым адресом, затем следует адрес ^X00001003. Если данная инструкция размещена в памяти с адреса ^X00001004, то это выглядит следующим образом:

Примечание	Содержимое	Адрес
Код операции MOVB	90	00001004
Спецификатор операнда	9F	00001005
Адрес, по которому выбирается байт	00001000	00001006
Спецификатор операнда	9F	0000100A
Адрес, по которому заносится байт	00001003	0000100B

Инструкции в таком полностью числовом формате называются *инструкциями на машинном языке или в машинном коде.* Обратите внимание, что, когда инструкция MOVB кодируется в машинном коде, она занимает 11 байтов памяти. Код операции занимает 1 байт, а также требуется по 5 байтов для определения месторасположения каждого из двух операндов – 1 байт для спецификатора операнда и 4 байта для 32-битового адреса.

Более компактно приведенная выше инструкция MOVB может быть записана следующим образом:

Содержимое памяти	Адрес
00001003 9F 00001000 9F 90	00001004

Инструкцию в таком формате следует читать справа налево. Адрес начала инструкции ^X00001004 помещается в конце строки. Слева от адреса расположена информация, обозначающая содержимое памяти в диапазоне адресов от ^X00001004 до ^X0000100E. После адреса начала инструкции в строке в последовательности справа налево находятся: код операции ^X90, спецификатор первого операнда байт ^X9F, адрес ^X00001000, по которому выбирается пересылаемый байт, спецификатор второго операнда байт ^X9F и адрес ^X00001003, по которому заносится в память пересылаемый байт. Далее в книге обычно будет использоваться именно такое представление для инструкций в машинном коде. Отметьте, что два приведенных представления инструкции MOVB содержат одинаковую информацию о содержимом памяти.

Инструкции ADDB2 и SUBB2 имеют одинаковый формат с инструкцией MOVB и различаются только кодами операций. В противоположность этому инструкция RET занимает

Примечания	Содержимое		Адрес
БАЙТ ДАННЫХ		1A	00001000
БАЙТ ДАННЫХ		26	00001001
БАЙТ ДАННЫХ		05	00001002
РЕЗУЛЬТАТ (СУММА)		??	00001003
MOVB ИНСТРУКЦИЯ	00001003 9F 00001000 9F	90	00001004
ADDB2 ИНСТРУКЦИЯ	00001003 9F 00001001 9F	80	0000100F
ADDB2 ИНСТРУКЦИЯ	00001003 9F 00001002 9F	80	0000101A
RET ИНСТРУКЦИЯ		04	00001025

Рис. 3.5. Стандартный формат программы в машинном коде

только 1 байт памяти, так как она не имеет никаких операндов. На рис. 3.5 показано, как будет выглядеть в машинном коде программа, приведенная на рис. 3.4.

При выполнении инструкций с адреса ^X00001004 процессор:

1) выполняет инструкцию MOVB, начинающуюся с адреса ^X00001004;
2) выполняет инструкцию ADDB2, начинающуюся с адреса ^X0000100F;
3) выполняет инструкцию ADDB2, начинающуюся с адреса ^X0000101A;
4) выполняет инструкцию RET, расположенную по адресу ^X00001025.

В результате этого в байт памяти с адресом ^X00001003 было бы помещено значение ^X45.

3.6. ПРОГРАММНЫЙ СЧЕТЧИК

В предыдущем разделе было описано, как числа, включая данные и инструкции на машинном языке, хранятся в памяти. В дополнение к основной памяти непосредственно сам процессор ЭВМ VAX содержит небольшое количество ячеек памяти с очень быстрым доступом. Чтобы отличить эти довольно ограниченные внутренние запоминающие ячейки процессора от существенно большей области основной памяти, внутренние запоминающие ячейки процессора называются *регистрами процессора* или просто *регистрами.*

Хотя в процессоре ЭВМ VAX существуют разнообразные внутренние запоминающие ячейки, большинство программистов в основном имеют дело с 16 регистрами, называемыми *регистрами общего назначения* (сокращенно РОН). Поскольку все 16 регистров общего назначения являются 32-разрядными, то в каждый регистр помещается длинное слово (четыре байта). Чтобы различать между собой регистры общего назначения, им присвоены номера от 0 до 15 (десятичные числа) или от ^X0 до ^XF (шестнадцатеричные числа). Обратите внимание, что эти регистры сильно отличаются от ячеек памяти. Каждая минимальная адресуемая ячейка памяти может содержать только один байт (восемь битов информации). Если в память помещается 32-битовое длинное слово, то для него требуется четыре таких ячейки памяти. В противоположность этому в каждый регистр общего назначения помещается длинное слово, так что в 16 регистрах общего назначения всего размещаются 16 длинных слов, или 64 байта.

Вопреки своему названию регистр общего назначения 15 используется для весьма определенной цели. (Как мы увидим далее, регистры общего назначения 12, 13 и 14 также применяются для определенных целей.) При выполнении процессором каждой инструкции на машинном языке он использует регистр 15 для адресации следующей инструкции. Вследствие этого регистр процессора 15 называется *программным счетчиком* (Program Counter) или просто PC. Чтобы понять, как работает программный счетчик, ниже показаны первые 15 байтов программы, приведенной на рис. 3.5.

Память		Процессор
Содержимое	Адрес	00001004 PC
1A	00001000	
26	00001001	
05	00001002	
??	00001003	
90	00001004	
9F	00001005	
00	00001006	
10	00001007	
00	00001008	
00	00001009	
9F	0000100A	
03	0000100B	
10	0000100C	
00	0000100D	
00	0000100E	
...	0000100F	

Для того чтобы выполнить программу, начинающуюся с адреса ^X00001004, необходимо поместить адрес ^X00001004 в программный счетчик PC, как показано выше. Такая загрузка адреса производится ОС VAX/VMS при передаче ею управления программе на машинном языке. После этого процессор выполняет инструкцию MOVB, начинающуюся с адреса ^X00001004, для чего он должен совершить следующие действия.

Шаг 1. Выбрать 8-битовый код операции. Так как программный счетчик содержит значение ^X00001004, процессор произведет выборку байта ^X90, расположенного по адресу ^X00001004, считая, что данный байт является кодом операции. Кроме того, процессор увеличит содержимое программного счетчика на 1, так что теперь программный счетчик будет содержать значение ^X00001005 (в этом случае говорят, что программный счетчик указывает на адрес ^X00001005).

Шаг 2. Выбрать байт спецификатора операнда. Так как выбранный на шаге 1 код операции является кодом операции инструкции MOVB, то вслед за этим процессор производит выборку байта ^X9F, на который в текущий момент времени указывает регистр PC. Процессор считает, что данный байт является спецификатором операнда. Содержимое программного счетчика увеличивается на 1, после чего программный счетчик указывает на адрес ^X00001006.

Шаг 3. Выбрать адрес первого операнда. Выбранный на шаге 2 байт спецификатора операнда ^X9 показывает, что программный счетчик указывает на адрес операнда. Иначе говоря, следующие после байта спецификатора операнда четыре байта представляют собой адрес операнда. Процессор производит выборку четырех байтов с адресами от ^X00001006 до ^X00001009 и собирает эти байты, чтобы сформировать адрес ^X00001000. Одновременно с этим содержимое программного счетчика увеличивается на 4, так что он указывает теперь на адрес ^X0000100A.

Шаг 4. Выбрать первый операнд. Так как на шаге 3 был выбран адрес нахождения операнда ^X00001000, то процессор производит выборку байта ^X1A, расположенного по этому адресу.

Шаг 5. Выбрать байт спецификатора второго операнда. Поскольку в инструкции MOVB требуется два операнда, то процессор производит выборку байта ^X9F, на который указывает регистр PC, содержащий адрес ^X0000100A, и интерпретирует этот байт как байт спецификатора операнда. Процессор также увеличивает на 1 содержимое регистра PC, которое становится равным ^X0000100B.

Шаг 6. Выбрать адрес второго операнда. Так как на шаге 5 был выбран код ^X9F, то процессор производит выборку содержимого байтов с адресами от ^X0000100B до ^X0000100E и, объединяя их вместе, формирует адрес ^X00001003. Кроме того, содержимое регистра PC увеличивается на 4. Регистр PC содержит теперь адрес ^X0000100F.

Шаг 7. Выполнить пересылку. Процессор помещает байт ^X1A, выбранный на шаге 4, по адресу ^X00001003, выбранному на шаге 6. Этим завершается выполнение инструкции MOVB. Содержимым регистра PC остается ^X0000100F, так как на текущем шаге содержимое регистра не модифицировалось.

После завершения выполнения инструкции процессор производит выборку байта, указываемого программным счетчиком, и увеличивает содержимое регистра PC, считая, что данный байт является новым кодом операции. (После завершения шага 7 приведенной выше последовательности действий процессор будет производить выборку содержимого памяти с адресом ^X0000100F, считая, что это новый код операции.) Последовательность действий, которую затем выполняет процессор, зависит от нового кода операции. Если код операции соответствует инструкции MOVB, то процессор выполнит предыдущие семь шагов; если новый код операции задает инструкцию ADDB2 или SUBB2, процессор выполнит последовательность действий, состоящую из девяти шагов. Первые шесть шагов идентичны первым шести шагам выполнения инструкции MOVB, показанным выше. Остальные три шага приводятся ниже.

Шаг 7. Выбрать второй операнд. Процессор производит выборку байта, используя адрес, полученный на шаге 6.

Шаг 8. Выполнить сложение или вычитание. Процессор складывает (или вычитает) байт, выбранный на шаге 4, с байтом (из байта), выбранным на шаге 7.

Шаг 9. Запомнить результат. Процессор помещает вычисленное на шаге 8 значение в байт памяти, расположенный по адресу, который был выбран на шаге 6.

Выполняя программу, процессор слепо выбирает из памяти и выполняет инструкции на машинном языке до тех пор, пока: 1) не будет встречена инструкция RET, 2) оператор сам (вручную) не остановит ЭВМ или 3) некоторое состояние ошибки не заставит процессор завершить выполнение программы.

3.7. ОШИБКИ ВЫПОЛНЕНИЯ

Как мы только что видели, содержимое байта памяти можно интерпретировать или использовать различными способами. Например, на рис. 3.5 некоторые байты трактовались как коды операций, другие – как спецификаторы операндов, третьи – как байты, составляющие 4-байтовый адрес, а прочие рассматривались как операнды. Невозможно посмотреть содержимое байта памяти и определить, как следует интерпретировать это содержимое. Если в каком-либо конкретном байте памяти содержится код ^X9F, то это содержимое может быть:

1) кодом операции инструкции PUSHAB (PUSH Address of Byte – включить в стек адрес байта), она будет рассматриваться в гл. 9,
2) спецификатором операнда ^9F;
3) одним из четырех байтов длинного слова ^X9F9F9F9F, являющегося адресом;
4) байтовым операндом, интерпретируемым как целое число без знака, эквивалентное десятичному числу 159;
5) байтовым операндом, интерпретируемым как целое число со знаком, эквивалентное десятичному числу –97.

Важно отличать друг от друга коды операций, спецификаторы операндов, адреса и операнды, так как каждому из них отводится различная роль. Код операции дает указание процессору совершить некоторую операцию, такую как MOVB, ADDB2, SUBB2 или RET. Поскольку каждый код операции является 8-битовым, то потенциально имеется 2^8, или

256, различных кодов операции[1]. В программах пользователей может использоваться большинство допустимых кодов операций. Однако некоторые коды операций, включая инструкцию HALT (остановить) с кодом операции ^X00, относятся к *привилегированным инструкциям* и могут выполняться только теми компонентами операционной системы, которые имеют особые привилегии. Если в программе пользователя предпринимается попытка выполнить привилегированную инструкцию, то возникнет ошибка, называемая *прерыванием по резервной инструкции* (reserverd instruction trap), и управление будет возвращено ОС VAX/VMS.

Байт спецификатора операнда указывает способ определения местонахождения операнда. Поскольку спецификатор операнда занимает восемь битов (один байт), то потенциально существует 2^8, или 256, различных способов определения местонахождения операнда. До сих пор рассматривался единственный спецификатор операнда – код ^X9F. Это значение спецификатора операнда указывает, что после байта спецификатора операнда непосредственно следует длинное слово адреса операнда. Как было отмечено ранее, такой способ определения местонахождения операнда называется *абсолютной адресацией*. Некоторые из других 255 возможных значений байта – спецификатора операнда являются недопустимыми. Попытка использования недопустимого значения вызовет *прерывание по резервному режиму адресации* (reserved addressing mode fault), по которому управление будет возвращено ОС VAX/VMS.

Адреса представляют собой 32-битовые длинные слова, значение адреса может изменяться от ^X00000000 до XFFFFFFFF. Однако операционная система VAX/VMS обычно ограничивает диапазон адресов, к которым может иметь доступ программа пользователя. Попытка нарушить эти ограничения приводит к ошибке, называемой *прерыванием по нарушению контроля доступа* (access control violation fault).

Для байтового операнда допустимо любое шестнадцатеричное значение от ^X00 до ^XFF. Как было отмечено в гл. 2, в формате байта могут быть представлены числа со знаком и без знака. Важно понять, что имеется только одна инструкция сложения байтов – ADDB2. Например, при сложении байтов ^XFC и ^X02 результат будет равен ^XFE. Если эти значения интерпретировать как числа без знака, то их сложение будет соответствовать сложению десятичных чисел 252 и 2, что даст в результате 254. При интерпретации этих значений как чисел со знаком их сложение соответствует сложению чисел –4 и 2, что в сумме даст –2.

Ошибки переполнения возможны при любой интерпретации чисел. Например, сумма ^XFF и ^X03 равна ^X02. (В обычных условиях правильной суммой является ^X102, но это число не помещается в формате байта.) Для чисел со знаком такой результат корректен, так как сумма чисел –1 и 3 равна 2. Однако в случае чисел без знака результат неверен, так как сумма чисел 255 и 3, конечно, не равна 2. Тогда говорят, что произошло *переполнение числа без знака*. На ЭВМ семейства VAX переполнение числа без знака не рассматривается как ошибка. В гл. 6 объясняется, как проводится проверка на переполнение беззнаковых чисел.

Аналогично сумма ^X7E и ^X04 равна ^X82. В этом случае для чисел без знака результат будет правильным (126 плюс 4 равно 130), но для чисел со знаком он неверен (126 плюс 4 не равно –126). Тогда говорят, что произошло *переполнение числа со знаком*. Если в специальный разряд процессора, называемый битом IV, занесена 1, то при переполнении числа со знаком будет порождаться ошибка, называемая *прерыванием по арифметической операции* (arithmetic trap). В противном случае ошибка не возникает. В гл. 6 показано также, как программист может проверить наличие ошибки переполнения числа со знаком.

[1] В действительности некоторые коды операций действуют как управляющие коды, которые служат признаком того, что код определенной операции может быть найден в следующем байте. Это позволяет определять много дополнительных кодов операций. В настоящее время в ЭВМ семейства VAX реализовано всего около 300 кодов операций.

Как отмечалось ранее, помимо байтовых операндов ЭВМ семейства VAX могут работать с операндами в формате 16-битового слова, 32-битового длинного слова и в некоторых случаях – с операндами в формате 64-битового квадраслова. При выполнении арифметических операций над операндами в формате слова или длинного слова может происходить переполнение чисел со знаком и без знака, точно так же как и при выполнении операций над байтами.

УПРАЖНЕНИЯ 3.1

1. На рис. 3.3 укажите шестнадцатеричные значения следующих элементов:
   а) слова, начинающегося с адреса ^X00000000;
   б) слова, начинающегося с адреса ^X00000001;
   в) слова, начинающегося с адреса ^X00000008;
   г) длинного слова, начинающегося с адреса ^X00000000;
   д) длинного слова, начинающегося с адреса ^X00000001;
   е) длинного слова, начинающегося с адреса ^X00000005.
2. Дайте определение каждому из следующих терминов:
   а) код операции;
   б) спецификатор операнда;
   в) операнд;
   г) программный счетчик;
   д) регистр процессора;
   е) регистр общего назначения;
   ж) ошибка выполнения;
   з) инструкция на машинном языке;
   и) программа в машинном коде.
3. Пусть байты памяти с адресами от ^X0200 до ^X0203 содержат следующее:

Содержимое	Адрес
1F	00000200
05	00000201
FF	00000202
3F	00000203

Опишите, что произойдет при выполнении нижеследующих инструкций на машинном языке. Для каждой инструкции выпишите адрес и новое содержимое каждого байта, если оно изменилось. (Считайте, что инструкции выполняются независимо друг от друга. Перед выполнением каждой инструкции содержимое байтов памяти с адресами от ^X0200 до ^X0203 имеет начальные значения, приведенные выше):

а) 00000203 9F 00000200 9F 90;
б) 00000201 9F 00000200 9F 80;
в) 00000202 9F 00000200 9F 80;
г) 00000200 9F 00000201 9F 82;
д) 00000203 9F 00000203 9F 82.

3.8. НЕКОТОРЫЕ ДОПОЛНИТЕЛЬНЫЕ ИНСТРУКЦИИ

ОПИСАНИЕ ИНСТРУКЦИЙ

Для описания инструкций ЭВМ семейства VAX будут применяться условные обозначения, подобные обозначениям, используемым фирмой Digital Equipment Corporation в документации для вычислительных систем VAX. Например, описание четырех инструкций, с которыми читатель уже знаком, выглядит следующим образом:

Мнемоника	Код операции	Операнды	Операция (описание)
ADDB2	80	add, sum	sum ← sum + add
SUBB2	82	sub, dif	dif ← dif – sub
MOVB	90	src, dst	dst ← src
RET	04		возврат управления вызывающей программе

В первой строке дано описание инструкции ADDB2. Мнемонический код операции ADDB2 обозначает инструкцию сложения байтов (Add Byte) с двумя операндами. В графе "Код операции" указывается код операции ^X80, соответствующий инструкции ADDB2. Обозначения add (слагаемое) и sum (сумма) в поле операндов могут рассматриваться как имена переменных в языках высокого уровня, таких как Фортран и Паскаль. Данные обозначения используются также в графе описания действия операции "Операция". В этой графе описание sum ← sum + add показывает, что первый операнд, обозначенный add, складывается со вторым операндом sum, а результат сложения помещается в sum – во второй операнд.

Другие инструкции описываются аналогично. Для инструкции SUBB2 (SUBtract Byte – вычесть байт, два операнда) описанием операции является dif ← dif – sub. Это означает, что по месту нахождения второго операнда dif (разность) помещается разность между первоначальным значением второго операнда dif и значением первого операнда sub (вычитаемое). В описании инструкции MOVB обозначение src (от SouRCe – источник) относится к тому операнду, который будет выбираться или считываться из памяти, а обозначение dst (от DeSTination – место назначения или получатель) относится к операнду, по месту нахождения которого будет занесен или записан в память результат.

Следует сделать два замечания относительно такого описания инструкций. Во-первых, описания следует читать в обычном порядке – слева направо, в отличие от кодов машинного языка, которые читаются справа налево. Во-вторых, такой формат описания использовался при описании инструкций ЭВМ семейства VAX, приведенных на развороте обложки и в приложении Е.

ДОПОЛНИТЕЛЬНЫЕ ИНСТРУКЦИИ

На примере инструкции сложения будет проиллюстрировано, как из отдельных инструкций образуется семейство инструкций. В гл. 1 рассматривалась инструкция ADDL2 (ADD Longword – сложить длинное слово, два операнда). Семейство инструкций сложения ADD включает инструкции сложения операндов в формате байта и слова. Описание инструкций сложения дано ниже.

Мнемоника	Код операции	Операнды	Операция
ADDB2	80	add, sum	sum ← sum + add
ADDW2	A0		
ADDL2	C0		

Эти инструкции будут называться семейством инструкций ADD2.

В гл. 1 рассматривалась также инструкция ADDL3 (ADD Longword – сложить длинное слово, три операнда). Семейство инструкций сложения ADD включает также инструкции с тремя операндами, в том числе инструкции сложения операндов в формате байта и слова. Описание семейства инструкций ADD3 приведено ниже.

Мнемоника	Код операции	Операнды	Операция
ADDB3	81	add1, add2, sum	sum ← add1 + add2
ADDW3	A1		
ADDL3	C1		

Например, инструкция на машинном языке

```
00000800  9F  00001000  9F  00001200  9F  C1
```


вызовет сложение длинного слова, начинающегося с адреса ^ X00001200, с длинным словом, начинающимся с адреса ^X00001000, а сумма будет помещена в длинное слово, расположенное по адресу ^X00000800. Если бы адресам ^X0000800, ^X00001000 и ^X00001200 соответствовали символические имена A, B и C, то эта инструкция могла быть написана на языке ассемблера следующим образом:

```
ADDL3   C, B, A
```

Семейство инструкций вычитания SUB подобно семейству инструкций сложения ADD. Ниже приведем шесть из них:

Мнемоника	Код операции	Операнды	Операция
SUBB2	82	sub, dif	dif ← dif – sub
SUBW2	A2		
SUBL2	C2		
SUBB3	83	sub, min, dif	dif ← min – sub
SUBW3	A3		
SUBL3	C3		

Семейство инструкций пересылки MOVB включает следующие инструкции:

Мнемоника	Код операции	Операнды	Операция
MOVB	90	src, dst	dst ← src
MOVW	B0		
MOVL	D0		
MOVQ	7D		

Инструкция MOVQ (MOVe Quadword) служит для пересылки 64-битового квадраслова.

3.9. ДОПОЛНИТЕЛЬНЫЕ СПЕЦИФИКАТОРЫ ОПЕРАНДОВ

НЕПОСРЕДСТВЕННАЯ АДРЕСАЦИЯ

Байт спецификатора операнда указывает способ определения местонахождения операнда. Рассмотренный ранее спецификатор операнда ^X9F указывает, что для нахождения операнда используется режим *абсолютной адресации*. Данный режим адресации указывает, что адрес операнда содержится в длинном слове, расположенном непосредственно после байта спецификатора оперенда.

Режим *непосредственной адресации* подобен режиму абсолютной адресации, за тем исключением, что после байта спецификатора операнда непосредственно следует не адрес операнда, а сам операнд. Для указания режима непосредственной адресации используется спецификатор операнда ^X8F. Рассмотрим, например, следующую инструкцию на машинном языке:

Второй операнд	Первый операнд	Код операции
00000800 9F	50 8F	80

Код операции ^X80 соответствует инструкции ADDB2 – сложение байтов. Спецификатор первого операнда ^X8F указывает, что используется непосредственная адресация. В следующем байте содержится значение ^X50, которое является значением первого операнда. В остальных пяти байтах инструкции содержится указание, что для выборки второго операнда применяется абсолютная адресация, а сам операнд расположен по адресу ^X00000800. В результате по данной инструкции будет складываться значение ^X50 и содержимое байта памяти с адресом ^X00000800. Если перед выполнением данной инструкции значение этого содержимого равно ^X60, то после выполнения инструкции содержимое этого байта памяти будет равно значению ^XB0.

Подобным образом по следующей инструкции будет выполнено сложение числа ^X01234567 с содержимым длинного слова, начинающегося с адреса ^X00000A00:

Второй операнд	Первый операнд	Код операции
00000A00 9F	01234567 8F	C0

Поскольку коду операции ^XC0 или инструкции ADDL2 требуются операнды в формате длинного слова, то непосредственный операнд (в данном случае 01234567) размещается в четырех байтах, следующих за байтом спецификатора операнда. Обратите внимание, что ячейка памяти с адресом ^X01234567 никак не участвует в выполнении этой инструкции.

Операнды, содержащиеся в машинном коде самой инструкции, такие как 50 и 01234567 в предыдущих примерах, называются *непосредственными операндами*. Поскольку операнды находятся непосредственно в машинном коде самой инструкции, то они становятся доступными сразу же после выборки инструкции.

В языке ассемблера для указания непосредственной адресации используется знак номера #. Например, в следующих инструкциях на языке ассемблера:

```
ADDB2   #80, ALPHA
ADDL2   #1000, BETA
```

знак номера #, стоящий перед первым операндом, укажет ассемблеру при трансляции этих инструкций на машинный язык, что используется непосредственная адресация.

ОТНОСИТЕЛЬНАЯ АДРЕСАЦИЯ

Режим абсолютной адресации, определяемый спецификатором ^X9F, позволяет получать доступ к любой адресуемой области памяти. Однако такой режим является неэкономным с точки зрения распределения памяти. Например, программа в машинном коде, приведенная на рис. 3.5, повторно показана на рис. 3.6. Обратите внимание, что большинство байтов этой программы отводится для хранения адресов. Программа занимает 38 байтов памяти, из них 24 байта отведено для указания адресов.

Более эффективен другой способ адресации, называемый *относительной адресацией*. При относительной адресации местонахождение операнда задается разностью между действительным адресом нахождения операнда и текущим значением программного счетчика. Эта разность называется смещением. В сущности, смещение определяет местонахождение операнда по отношению к текущему адресу, т.е. предшествует ли адрес операнда текущему адресу или находится после него, а также насколько велико удаление адреса операнда относительно текущего. Вспомним, что программный счетчик PC содержит адрес текущей инструкции или части этой инструкции.

Чтобы понять, как работает относительная адресация, рассмотрим рис. 3.7. Программа, приведенная на рис. 3.7, идентична программе, показанной на рис. 3.6, за тем исключением, что вместо абсолютной адресации используется относительная. На рис. 3.7 вместо спецификатора операнда ^X9F применяется спецификатор ^XAF. Спецификатор ^XAF означает, что для указания адреса используется единственный байт. Если содержимое байта (рассматриваемое как число, представленное в дополнительном коде) является отрицательным, это означает, что операнд расположен в памяти перед текущим адресом и смещен относительно него на указанное число байтов. Если содержимое байта является

Примечания	Содержимое		Адрес
БАЙТ ДАННЫХ		1A	00001000
БАЙТ ДАННЫХ		26	00001001
БАЙТ ДАННЫХ		05	00001002
РЕЗУЛЬТАТ (СУММА)		??	00001003
MOVB ИНСТРУКЦИЯ	00001003 9F 00001000 9F	90	00001004
ADDB2 ИНСТРУКЦИЯ	00001003 9F 00001001 9F	80	0000100F
ADDB2 ИНСТРУКЦИЯ	00001003 9F 00001002 9F	80	0000101A
RET ИНСТРУКЦИЯ		04	00001025

Рис. 3.6. Стандартный формат программы в машинном коде

положительным, это означает, что операнд размещается в памяти после текущего адреса через соответствующее число байтов.

Например, рассмотрим на рис. 3.7 инструкцию MOVB. Код операции ^X90 расположен по адресу ^X00001004. Содержимым байта с адресом ^X00001006 является код ^XAF, который указывает, что местонахождение первого операнда будет определяться с помощью относительной адресации. Следующий байт с адресом ^X00001006 содержит смещение ^XF9. Число ^XF9, интерпретируемое как число со знаком, эквивалентно десятичному числу – 7. После выборки из памяти смещения содержимое программного счетчика будет увеличено, так что он будет содержать адрес ^X00001007. Поэтому адресом первого операнда инструкции MOVB является

```
00001007
      –7
--------
00001000
```

Обратите внимание, что адрес операнда ^X00001000 совпадает с адресом первого операндов предлагается читателю в качестве упражнения (См. п. 3 упр. 3.1).

Применение относительной адресации вместо абсолютной уменьшает объем памяти, занимаемый программой, с 38 до 20 байтов. Программа не только получается меньше, но и выполняется быстрее, так как при выполнении производится меньшее число выборок из памяти. Однако относительная адресация со смещением, заданным в формате байта, пригодна только для небольших программ, поскольку, задавая смещение в формате байта, максимально можно адресовать 127 ячеек после или 128 ячеек перед, относительно текущего значения программного счетчика. Чтобы избежать подобных ограничений, в ЭВМ семейства VAX для относительной адресации позволяется использовать слова и длинные слова. Эти режимы адресации распознаются по спецификаторам операндов ^XCF и ^XEF соответственно. Использование смещений, задаваемых в формате слова или длинного слова, не только увеличит объем программы, но и позволит также адресовать любое место в программе независимо от ее объема.

Режимы относительной адресации со смещением, заданным в формате байта, слова и длинного слова, можно найти в программах на языке ассемблера, приведенных в следующей главе, а более подробное описание этих режимов можно найти в гл. 7. Как мы увидим, кроме экономии времени выполнения и объема занимаемой памяти относительная адресация имеет и другие преимущества. В результате когда ассемблер выполняет трансляцию операторов, таких как

```
MOVB    A, B
ADDL3   C, D, E
```

на машинный язык, то вместо абсолютной он использует относительную адресацию. Отметим, что применение последней не требует какой-либо дополнительной работы от программиста, так как смещение вычисляет сам ассемблер.

Примечания	Содержимое		Адрес
БАЙТ ДАННЫХ		1A	00001000
БАЙТ ДАННЫХ		26	00001001
БАЙТ ДАННЫХ		05	00001002
РЕЗУЛЬТАТ (СУММА)		??	00001003
MOVB ИНСТРУКЦИЯ	FA AF F9 AF	90	00001004
ADDB2 ИНСТРУКЦИЯ	F5 AF F5 AF	80	0000100F
ADDB2 ИНСТРУКЦИЯ	F0 AF F1 AF	80	0000100F
RET ИНСТРУКЦИЯ		04	00001025

Рис. 3.7. Применение относительной адресации в программе на машинном языке

УПРАЖНЕНИЯ 3.2

1. Пусть в памяти содержатся следующие длинные слова:

Содержимое	Адрес
0000000F	00000200
FFFFFFFF	00000204
00000100	00000208
00000001	0000020C

Опишите, что произойдет при выполнении каждой из нижеследующих инструкций на машинном языке. Выпишите для каждой инструкции адрес и новое содержимое каждого длинного слова, если оно изменилось. (Считайте, что инструкции выполняются независимо друг от друга. Перед выполнением каждой инструкции байты памяти с адресами от ^X0200 до ^X020F принимают исходные значения, приведенные выше.)

а) 00000208 9F 00000200 9F D0
б) 00000208 9F 00000200 8F D0
в) 00000204 9F 0000020C 9F C0
г) 00000204 9F 0000020C 8F C0
д) 0000020C 9F 00000204 9F C2
е) 00000204 9F 00000010 8F 00000208 9F C1
ж) 00000204 9F 00000208 9F 00000020 8F C3
з) 00000208 9F 00000200 9F 7D

2. Какие шестнадцатеричные числа будут содержаться в длинных словах с адресами ^X0200, ^X0204, ^X0208 и ^X020C после того, как следующая программа вернет управление операционной системе? Выполнение начинается с адреса ^X0210:

Инструкция	Адрес
00000200 9F 00000020 8F D0	00000210
00000204 9F 00000040 8F D0	0000021B
00000208 9F 00000200 9F 00000204 9F C1	00000226
0000020C 9F 00000208 9F 0000000F 8F C3	00000236
04	00000246

3. Для программы рис. 3.7 определите адреса оставшихся пяти операндов. Сравните эти адреса с адресами операндов программы рис. 3.6.

4. Напишите, начиная с адреса ^X0200, программы в машинном коде, эквивалентные следующим программам на языках Фортран и Паскаль:

Фортран

```
INTEGER J, K, L
J=15
K=22
L=J-K+9
STOP
END
```

Паскаль

```
PROGRAM FOUR;
VAR
 J, K, L:INTEGER;
BEGIN
 J := 15;
 K := 22;
 L := J-K+9;
END.
```


5. Выполните ассемблирование следующей программы на языке ассемблера и проанализируйте листинг ассемблера с программой в машинном коде. Определите коды операций, спецификаторы операндов, адреса и числа в листинге ассемблера. (Директива .ENABLE в первой строке программы предписывает ассемблеру использование абсолютной адресации, которая была описана в данной главе.)

```
        .ENABLE   ABSOLUTE
J:      .BLKL     1
K:      .BLKL     1
DIF:    .BLKL     1
        .ENTRY    ADDRESS, 0
FIRST:  MOVL      #512, J
        MOVL      #64, K
        SUBL3     K, J, DIF
LAST:   $EXIT_S
        .END      ADDRESS
```

6. Повторите п. 5, исключив из программы первую строку (.ENABLE ABSOLUTE). Это позволит ассемблеру использовать относительную адресацию. Объясните формирование каждого относительного адреса и покажите, что при этом обеспечивается правильный исполнительный адрес в процессе выполнения программы.

ГЛАВА 4. АССЕМБЛЕР И ПРОЦЕСС АССЕМБЛИРОВАНИЯ

4.1. ВВЕДЕНИЕ

Программирование на машинном языке затруднительно для программиста. Например, для сложения содержимого двух длинных слов с именами TAX и TOTAL программисту следовало бы написать в машинном коде такую инструкцию, как

Операнд	Операнд	Код операции
E4 AF	F2 AF	C0

При этом программист должен помнить, что ^XC0 – это код операции инструкции ADDL2, а ^XAF – спецификатор операнда, определяющий режим относительной адресации с заданным в формате байта смещением. Кроме того, чтобы получить доступ к длинным словам TAX и TOTAL, программист должен вычислить их смещение. В данном примере смещение для длинного слова TAX равно ^XF2, а для длинного слова TOTAL – ^XE4. Чтобы оценить ситуацию, с которой сталкивается программист при кодировании, стоит отметить, что ЭВМ семейства VAX имеют несколько сотен различных кодов операций; к тому же обычно в программе используются несколько тысяч ячеек памяти.

Языки ассемблера **намного** снижают умственную нагрузку программиста, позволяя использовать вместо чисел *символические имена.* Например, предыдущую инструкцию на машинном языке можно написать на языке ассемблера следующим образом:

```
ADDL2   TAX, TOTAL
```

Программа, называемая *ассемблером,* выполняет трансляцию программы на языке ассемблера в программу на машинном языке, заменяя символические имена и специальные символы соответствующими числами. Для приведенного выше оператора языка ассемблера. Ассемблер произведет замену мнемоники ADDL2 на код операции ^XC0, сим-

волического имени TAX на коды ^XAF и ^XF2, имени TOTAL на коды ^XAF и ^XE4. Обратите внимание, что операторы языка ассемблера читаются обычным образом – слева направо. Рассматриваемый оператор языка ассемблера следует читать как "Сложить содержимое длинного слова, начинающегося с адреса TAX, с содержимым длинного слова, начинающегося с адреса TOTAL, а результат поместить в длинное слово, начинающееся с адреса TOTAL".

Помимо того, что ассемблер позволяет программисту использовать символические имена, он может также выполнять некоторые вспомогательные вычисления, такие как вычисление смещений и преобразование чисел из одной системы счисления в другую.

Обычно для каждого типа (или модели) ЭВМ имеется свой собственный язык ассемблера. Но бывает и так, что иногда для ЭВМ одного типа (или модели) существуют различные языки ассемблера. Фактически для ЭВМ семейства VAX имеются по крайней мере два различных ассемблера. Одним из них является программа-ассемблер, выполняемая под управлением операционной системы UNIX. Вторым является ассемблер, называемый VAX-11 MACRO или просто MACRO – макроассемблер, поставляемый в составе ОС VMS производителем ЭВМ семейства VAX – фирмой Digital Equipment Corporation. В данной книге будет рассматриваться только ассемблер VAX-11 MACRO.

4.2. ПРИМЕР ИСПОЛЬЗОВАНИЯ ЯЗЫКА АССЕМБЛЕРА

ПРОГРАММА НА ЯЗЫКЕ АССЕМБЛЕРА

На рис. 4.1 показана программа на языке ассемблера, в которой выполняется сложение чисел 26, 38 и 5, представленных в формате байта, а полученная сумма (число 69) помещается в ячейку памяти SUM. Как показано на рисунке, операторы языка ассемблера состоят из четырех полей. Первое поле называется полем *метки* и используется для определения символических адресов, таких как A, B, C, SUM и START. Вторым полем является поле кода операции, в нем может находиться мнемоническое обозначение кода операции, например MOVB, ADDB2 и RET. Кроме того, в этом поле могут помещаться *директивы ассемблера*, например .BYTE, .BLKB, .WORD и .END. Директивы дают указание ассемблеру выполнить некоторые специальные действия. Третье поле, называемое полем *операндов*, может использоваться для размещения в нем операндов инструкции. Например, в инструкции

```
ADDB2      B, SUM          ; SUM: =SUM+B
```

операнды B и SUM указывают, что байт, расположенный по адресу B, должен быть сложен с байтом, расположенным по адресу SUM. Последнее поле – поле комментариев, используется для документирования программы.

```
                Код
Метка       операции   Операнды        Примечания

A:          .BYTE      26             ; БАЙТ ДАННЫХ
B:          .BYTE      38             ; БАЙТ ДАННЫХ
C:          .BYTE      5              ; БАЙТ ДАННЫХ
SUM:        .BLKB      1              ; РЕЗЕРВИРОВАНИЕ БАЙТА
START:      .WORD      0              ; В СЛОВЕ СОДЕРЖИТСЯ 0
            MOVB       A, SUM         ; SUM: =A
            ADDB2      B, SUM         ; SUM: =SUM+B
            ADDB2      C, SUM         ; SUM: =SUM+C
            RET                       ; ВОЗВРАТ
            .END       START
```

Рис. 4.1. Программа на языке ассемблера


```
Машинный код                                  Язык ассемблера
--------------------                  ------------------------------------------------------
                                                  Код
Содержимое          Адрес Стр.  Метка   операции   Операнды   Примечания

                 1A 0000   1    A:       .BYTE      26         ; БАЙТ ДАННЫХ
                 26 0001   2    B:       .BYTE      38         ; БАЙТ ДАННЫХ
                 05 0002   3    C:       .BYTE      5          ; БАЙТ ДАННЫХ
           00000004 0003   4    SUM:     .BLKB      1          ; РЕЗЕРВИРОВАНИЕ БАЙТА
               0000 0004   5    START:   .WORD      0          ; В СЛОВЕ СОДЕРЖИТСЯ 0
F8 AF   F7 AF   90 0006   6             MOVB       A, SUM     ; SUM: =A
F3 AF   F3 AF   80 000B   7             ADDB2      B, SUM     ; SUM: =SUM+B
EE AF   EF AF   80 0010   8             ADDB2      C, SUM     ; SUM: =SUM+C
                04 0015   9             RET                   ; ВОЗВРАТ
                   0016  10             .END       START
```

Рис. 4.2. Листинг программы на языке ассемблера

При ассемблировании ассемблер транслирует программу на языке ассемблера в программу на машинном языке. На рис. 4.2 показан фрагмент листинга программы на языке ассемблера.

Как видно из рисунка, исходный текст программы на языке ассемблера располагается на правой стороне листинга. Результат трансляции программы на машинный язык располагается слева. В центральной части рисунка показано, что ассемблер присваивает каждой строке программы номер, и это помогает при разборе сообщений об ошибках.

Ниже приведена строка с номером 6 листинга программы

```
Машинный код                                  Язык ассемблера
--------------------                  ------------------------------------------------------
                                                  Код
Содержимое          Адрес Стр.  Метка   операции   Операнды   Примечания

F8 AF   F7 AF   90 0006   6             MOVB       A, SUM     ; SUM: =A
```

Ассемблер **транслировал** мнемоническое обозначение кода операции MOVB в код операции ^X90 и поместил его по адресу ^X00000006, на листинге этот адрес приводится в краткой записи 0006. Адреса, приводимые в тексте книги, могут сокращаться подобным образом тогда, когда ясно, что имеется в виду. Для адресации операнда A ассемблером был выбран режим относительной адресации с заданным в формате байта смещением (спецификатор операнда ^XAF) и произведено вычисление значения этого смещения (^XF7). Для адресации операнда SUM ассемблером был использован также режим относительной адресации, смещение данного операнда равно ^XF8. Строки программы с номерами 7 – 9 обрабатываются аналогично, начиная со следующего доступного адреса 000B.

На листинге ассемблера строка с номером 1 имеет следующий вид:

```
Машинный код                                  Язык ассемблера
--------------------                  ------------------------------------------------------
                                                  Код
Содержимое          Адрес Стр.  Метка   операции   Операнды   Примечания

                 1A 0000   1    A:       BYTE       26         ; БАЙТ ДАННЫХ
```

Директива ассемблера .BYTE применяется для резервирования памяти и задания начального значения данных, представляемых в формате байта. Если не указано противное, то ассемблер назначает адреса, начиная с адреса ^X0000. В результате метка A в рассматриваемом операторе является символическим указанием адреса ^X0000 и поэтому называется *символическим адресом*. По умолчанию ассемблер считает, что следующее за директивой

.BYTE значение представляет собой десятичное число. Следовательно, число 26, рассматриваемое как десятичное, преобразуется ассемблером в шестнадцатеричное число ^X1A и помещается по адресу ^X0000. Строки программы с номерами 2 и 3 обрабатываются аналогично.

В четвертой строке листинга директива .BLKB (BLock Byte – блок байтов) применяется для резервирования байта памяти без задания начального значения. Значение, следующее за директивой .BLKB, определяет число резервируемых байтов памяти. (Резервирование пространства памяти для массива из 20 байтов могло быть осуществлено с помощью директивы ассемблера .BLKB 20). В данном случае число байтов равно 1, таким образом, SUM – это просто символическое имя байта памяти, содержимое которого не определено перед началом выполнения программы. Число 00000004, расположенное в графе "Содержимое" раздела листинга "Машинный код", показывает, что следующим адресом, который может использовать ассемблер, является адрес ^X00000004. В действительности эта информация не включается в программу на машинном языке, а используется другими системными программами в процессе подготовки данной программы для выполнения. Программисту следует учитывать, что перед началом выполнения программы содержимое области памяти с адресом SUM не определено.

Директива .WORD в пятой строке программы подобна директиве .BYTE, отличаясь только тем, что по директиве .WORD резервируется память и задается начальное значение для данных в формате 16-битового слова. Символический адрес START является *адресом точки входа* в программу[1]. Операционная система VAX/VMS обращается с программами пользователей как с подпрограммами или процедурами, передавая им управление с помощью инструкций CALLS или CALLG, описанных в гл. 9. Эти инструкции требуют задания 16-битовой маски, которая обычно для основных программ равна 0. Ее назначение поясняется в гл. 9. Первая выполняемая инструкция программы следует непосредственно за маской.

Директива .END в строке с номером 10 помечает физический конец программы. Символическое имя START в директиве .END определяет имя START как адрес точки входа в программу.

4.3. ПРОЦЕСС АССЕМБЛИРОВАНИЯ

ТАБЛИЦЫ ИМЕН

Процесс трансляции программы, написанной на языке ассемблера, в программу на машинном языке называется *ассемблированием* и в основном заключается в замене имен числами. В процессе замены интенсивно используются две таблицы: *таблица постоянных имен* и *таблица имен, определяемых пользователем*. Таблица постоянных имен содержит значения для неизменяемых символических имен, таких как мнемонический код операций и директивы ассемблера. Приведем небольшую часть таблицы постоянных имен:

Постоянное имя	Шестнадцатеричное значение
ADDB2	80
ADDB3	81
ADDL2	C0
ADDL3	C1
ADDW2	A0
ADDW3	A1

[1] Адрес, с которого начинается выполнение программы. – *Прим. ред.*

Таблица имен, определяемых пользователем, содержит значения для заданных пользователем символических имен, таких как метки. В качестве примера приведем таблицу имен, определяемых пользователем, для программы рис. 4.2:

Имя, определенное пользователем	Шестнадцатеричное значение
A	00000000
B	00000001
C	00000002
START	00000004
SUM	00000003

В этой таблице каждому символическому имени ставится в соответствие числовое значение. В частности, A – символическое имя для адреса ^X0000, B – символическое имя для адреса ^X0001 и т. д. В листинге ассемблера таблица имен дается в алфавитном порядке, как и в нашем примере.

ПРОСТАЯ ТРАНСЛЯЦИЯ

Если таблица постоянных имен и таблица имен пользователя заданы, то процесс ассемблирования происходит следующим образом. Ассемблер просматривает (сканирует) программу строка за строкой. Когда будет обнаружено символическое имя в поле кода операции, производится поиск этого имени в таблице постоянных имен и устанавливается, является это имя кодом операции или директивой ассемблера. Директивы ассемблера вызывают выполнение соответствующих подпрограмм в программе-ассемблере.

Коды операций обрабатываются следующим образом. В таблице постоянных имен каждому символическому коду ставится в соответствие числовой код, а также информация о количестве и типе операндов. Ассемблер помещает числовой код операции в программу в машинном коде, а затем обрабатывает операнды. Он размещает соответствующую информацию в программе, в машинном коде, используя несколько простых правил. При этом, в частности, формируются спецификаторы операндов и определяется значение смещения. В таблице имен, определяемых пользователем, производится поиск в тех случаях, когда необходимо определить значения каких-либо символических имен, заданных в поле операндов[1].

СОЗДАНИЕ ТАБЛИЦЫ ИМЕН

В предыдущем разделе предполагалось, что таблицы постоянных имен и имен, определяемых пользователем, уже имеются. Таблица постоянных имен не меняется от программы к программе, поэтому она "встроена" в ассемблер.

И наоборот, таблица имен, определяемых пользователем, специфична для каждой исходной программы, и поэтому ассемблер создает ее для каждой программы заново. Ассемблер MACRO создает таблицу исходя из предположения, что *числа должны размещаться в смежных байтах памяти*. Используя довольно простые правила, ассемблер вычисляет число байтов, порождаемых при ассемблировании каждой строки программы. При условии, что первый сгенерированный байт должен размещаться по адресу ^X00000000, ас-

[1] Символические имена, выбранные как из поля кода операции, так и из поля операндов, ассемблер VAX-11 MACRO пытается прежде найти по таблице имен, определяемых пользователем. Если в ней отсутствует данное символическое имя, то ассемблер просматривает таблицу постоянных имен. Такой порядок обращения к таблицам позволяет пользователям переопределять символические имена, описанные в таблице постоянных имен.

семблер может определить точный адрес каждой инструкции и любых данных в программе. Поскольку некоторые строки программы содержат метки, меткам ставятся в соответствие определенные адреса и таким образом формируется таблица имен, определяемых пользователем.

Более строго: ассемблер создает таблицу имен, определяемых пользователем, отслеживая единственное значение, а именно – адрес следующего доступного байта памяти. Ассемблер запоминает текущее значение этого адреса в длинном слове, которое называется *счетчиком адресов.* (Опытные программисты пишут программы по частям. Для обеспечения такого стиля работы ассемблер VAX позволяет использовать несколько счетчиков адресов, по одному для каждой программной секции.)

В счетчике адресов обычно устанавливается начальное значение ^X00000000. Ассемблер просматривает исходную программу на языке ассемблера с начала до конца, соблюдая следующие правила.

Правило 1. Если ассемблер встречает символическое имя, после которого следует двоеточие (например, START:, A: или SUM:), он считает его символическим адресом. Символическое имя вместе с текущим значением счетчика адресов заносится в таблицу имен, определяемых пользователем. Содержимое счетчика адресов не меняется.

Правило 2. Когда ассемблер обнаруживает символическое имя в поле кода операции, то прибавляет некоторое соответствующее значение к содержимому счетчика адресов. Это значение представляет собой просто количество байтов машинного кода, которое будет сгенерировано при обработке этой инструкции, включая байты, занимаемые операндами. Например,

Код операции	Операнды	Соответствующее число байт
.BYTE	57	1
.WORD	0	2
.BLKB	1	1
.END	START	0
MOVB	A, SUM	ПЕРЕМЕННОЕ ДЛЯ ИНСТРУКЦИИ
ADDB2	B, SUM	ПЕРЕМЕННОЕ ДЛЯ ИНСТРУКЦИИ
RET		1

Объем памяти, занимаемый инструкциями MOVB и ADDB2, зависит от режима адресации и правил, используемых ассемблером для подбора спецификатора операнда. На рис. 4.2 для каждой из инструкций MOVB и ADDB2 требуется по 5 байтов. Применяя эти правила к программе на рис. 4.2, получаем приведенную выше таблицу имен.

Нетрудно заметить, что счетчик адресов при ассемблировании играет ту же роль, что и программный счетчик во время выполнения программы. Однако есть существенные различия между этими двумя счетчиками. Счетчик адресов – это ячейка памяти (длинное слово), помещенная внутри программы, называемой ассемблером. Он определяет место в памяти, куда будет занесен следующий байт программы, сгенерированный в процессе ассемблирования. Программный счетчик – это особый регистр (т.е. специальная часть аппаратуры) процессора ЭВМ VAX. Он определяет адрес начала следующей подлежащей выполнению инструкции в машинном коде.

ДВУХПРОХОДНОЕ АССЕМБЛИРОВАНИЕ

На рис. 4.3 дан модифицированный вариант программы, приведенной на рис. 4.1. Он отличается от предыдущего только тем, что директивы резервирования памяти для данных A, B, C и SUM перенесены в конец программы.

Посмотрим, что произойдет при ассемблировании этой программы. Вторая строка программы содержит оператор MOVB A, SUM. Пытаясь сгенерировать для этой строки машинный код, ассемблер может подставить числовой код операции ^X90 вместо мне-


```
Метка    Код операции   Операнды    Примечания
START:   .WORD          0           ;В СЛОВЕ СОДЕРЖИТСЯ 0
         MOVB           A, SUM      ; SUM: =A
         ADDB2          B, SUM      ; SUM: =SUM+B
         ADDB2          C, SUM      ; SUM: =SUM+C
         RET                        ;ВОЗВРАТ
A:       .BYTE          26          ;БАЙТ ДАННЫХ
B:       .BYTE          38          ;БАЙТ ДАННЫХ
C:       .BYTE          5           ;БАЙТ ДАННЫХ
SUM:     .BLKB          1           ;РЕЗЕРВИРОВАНИЕ БАЙТА
         .END           START
```

Рис. 4.3. Модифицированная программа на языке ассемблера

монического кода MOVB. Однако он не может вычислить значения спецификаторов операндов и смещения для операндов A и SUM, они являются символическими адресами и будут внесены в таблицу имен только при обработке шестой и седьмой строк этой программы.

Чтобы решить проблему, ассемблер делает два прохода по программе, т.е. просматривает программу на языке ассемблера дважды. Сначала не генерируется никакого машинного кода, поскольку адреса еще не определены. После первого просмотра программы адреса уже можно определить и ассемблер может сформировать таблицу имен, определяемых пользователем. Затем, при втором проходе по программе, генерируется программа в машинном коде.

Процесс формирования таблицы имен, определяемых пользователем, при сканировании программы на языке ассемблера называется *первым проходом* процесса ассемблирования. Программа в машинном коде создается ао время *второго* прохода. При этом ассемблер просматривает программу на языке ассемблера второй раз и, пользуясь уже сформированными при первом проходе таблицами постоянных имен и имен, определяемых пользователя, подставляет вместо символических имен числа для формирования программы в машинном коде.

Но и при двух проходах остается проблема вычисления смещения. Например, инструкция MOVB A, SUM во второй строке данной программы может занимать от 5 до 11 байтов в зависимости от того, используется ли для адресации операндов A и SUM относительная адресация со смещением, заданным в формате байта, слова или длинного слова.

При первом проходе ассемблер должен выбрать длину инструкции, чтобы поставить в соответствие символическим адресам правильные числовые значения. Если именам A и SUM выше уже присвоены значения, то ассемблер может легко выбрать один из форматов смещения – байт, слово, длинное слово для относительной адресации A и SUM. В данной программе, однако, имена A и SUM определяются ниже и при первом проходе ассемблер не может определить, насколько далеко от инструкции MOVB будут размещаться байты с именами A и SUM.

Чтобы разрешить эту проблему, ассемблер выбирает относительную адресацию со смещением, заданным в формате длинного слова, во всех случаях, когда на первом проходе обнаруживается неопределенное символическое имя[1]. Как показано на рис. 4.4, это су-

[1] Ассемблер разрешает программисту явно переопределять некоторые правила, принимаемые в языке ассемблера по умолчанию. В данном случае программист может задать смещение в формате байта, слова или длинного слова для относительной адресации с помощью префиксов B^, W^ и L^, помещаемых перед операндом. В частности, в инструкции ADDB3 B^X, W^Y, L^Z ассемблеру дается указание, что для относительной адресации X, Y и Z надо использовать смещение в формате байта, слова и длинного слова соответственно.

```
                    Машинный код                    Язык ассемблера
                    ------------                    ---------------
                                                            Код
                Содержимое          Адрес  Стр.  Метка  операции  Операнды   Примечания

                                    0000 0000   1  START:  .WORD     0         ;В СЛОВЕ СОДЕРЖИТСЯ 0
00000027'EF  00000024'EF  90 0002   2          MOVB      A, SUM    ; SUM: =A
00000027'EF  00000024'EF  80 000D   3          ADDB2     B, SUM    ; SUM: =SUM+B
00000027'EF  00000024'EF  80 0018   4          ADDB2     C, SUM    ; SUM: =SUM+C
                          04 0023   5          RET                 ;ВОЗВРАТ
                          1A 0024   6  A:       .BYTE     26        ;БАЙТ ДАННЫХ
                          26 0025   7  B:       .BYTE     38        ;БАЙТ ДАННЫХ
                          05 0026   8  C:       .BYTE     5         ;БАЙТ ДАННЫХ
                    00000028 0027   9  SUM:    .BLKB     1         ;РЕЗЕРВИРОВАНИЕ БАЙТА
                             0016  10          .END      START
```

Рис. 4.4. Листинг модифицированной программы на языке ассемблера

щественно увеличивает длину программы. (Программа рис. 4.2 занимает ^X16, или 22 байта, а функционально эквивалентная программа на рис. 4.4 занимает ^X28, или 40 байтов.)

Чтобы избежать таких потерь, программисты, пишущие на языке ассемблера VAX, обычно размещают область данных в начале программы (см. рис. 4.1), если намереваются использовать относительную адресацию. Не следует создавать программы со структурой, аналогичной структуре программы рис. 4.3.

4.4. ЗАВЕРШЕННЫЙ ВАРИАНТ ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

Если программу на языке ассемблера рис. 4.1 ассемблировать, скомпоновать и выполнить, она вычислит сумму 26,38 и 5 и в результате будет получено 69. Однако в этой программе есть недостатки, которые считаются следствием плохого стиля программирования. Улучшенный вариант этой программы показан на рис. 4.5. Первое отличие от программы рис. 4.1 заключается в использовании директивы .TITLE в начале программы. Как отмечалось в гл. 1, эта директива не требуется для выполнения, но настоятельно рекомендуется для документирования программы. Символическое имя, следующее за директивой .TITLE, в данном случае IMPROVED, рассматривается как имя программы. Остальной текст в строке (УСОВЕРШЕНСТВОВАННАЯ ПРОГРАММА НА АССЕМБЛЕРЕ) комментарий. Строки, начинающиеся с точки с запятой обрабатываются как комментарии, что показано на примере строк 2–6. Пустые строчки комментариев, например строчка 5, полезно вводить в программу для деления ее на разделы, что облегчает чтение программы.

Указание директивы .ENTRY в строчке 13 программы – это предпочтительный способ определения точки входа в программу. Оператор.ENTRY START,0 на рис. 4.5 заменяет оператор START:.WORD 0 на рис. 4.1. Первое символическое имя (START), следующее за директивой .ENTRY, определяет адрес точки входа в программу. По этой директиве адрес START не только заносится в таблицу имен, но и помечается как некоторый специальный адрес (в данном случае – адрес точки входа, который операционная система VAX/VMS использует для передачи управления программе). Кроме того, по директиве .ENTRY генерируется 16-битовая маска, которая используется, как уже описывалось инструкциями CALLS и CALLG. Число 0, следующее за адресом START, определяет, что значение этой маски равно 0, т.е. равно значению, задаваемому ранее директивой .WORD.

Еще одно отличие между программами рис. 4.1 и рис. 4.5 заключается в том, что инструкция RET заменена символическим именем $EXIT_S. Этот способ возврата управления операционной системе VAX/VMS наиболее предпочтителен. Имя $EXIT_S – не мнемонический код инструкции (как MOVB) и не директива (как .ENTRY), а имя макроинструкции. Макроинструкцией называется несколько строчек текста программы, которым дано некоторое имя и которые сохраняются в некотором, известном ассемблеру месте. Если в программе используется имя макроинструкции (в данном случае $EXIT_S), ассемблер


```
                        Код
Метка   Код операции  Операнды        Примечания

        .TITLE   IMPROVED - УСОВЕРШЕНСТВОВАННАЯ ПРОГРАММА НА АССЕМБЛЕРЕ
;ОПИСАНИЕ        ПРИМЕР ВЫЧИСЛЕНИЯ СУММЫ 26+38+5
;ПРОГРАММИСТ     КЭПС И СТАФФОРД
;ДАТА            ИЮЛЬ 1, 1985
;
;ОБЛАСТЬ ДАННЫХ
A:      .BYTE    26              ;БАЙТ ДАННЫХ
B:      .BYTE    38              ;БАЙТ ДАННЫХ
C:      .BYTE    5               ;БАЙТ ДАННЫХ
SUM:    .BLKB    1               ;РЕЗЕРВИРОВАНИЕ БАЙТА
;
;ОБЛАСТЬ ПРОГРАММЫ
        .ENTRY   START           ;АДРЕС ТОЧКИ ВХОДА
        MOVB     A, SUM          ; SUM: =A
        ADDB2    B, SUM          ; SUM: =SUM+B
        ADDB2    C, SUM          ; SUM: =SUM+C
        $EXIT_S                  ;ВОЗВРАТ
        .END     START,0
```

Рис. 4.5. Улучшенный вариант программы на языке ассемблера

автоматически заменяет его соответствующими строками текста программы. В случае макроинструкции $EXIT_S заменяющие ее строчки текста служат для возврата управления ОС VAX/VMS. Чтобы подчеркнуть то, что ассемблер для ЭВМ VAX включает возможности макрообработки, фирма DEC дала ему название VAX-11 MACRO. Подробно макроинструкции будут рассмотрены в гл. 10.

4.5. СИНТАКСИС ЯЗЫКА АССЕМБЛЕРА VAX-11 MACRO

СИМВОЛИЧЕСКИЕ ИМЕНА

В ассемблере требуется, чтобы операторы языка ассемблера удовлетворяли определенным *синтаксическим правилам.* Часть правил касается написания символических имен, определяемых программистом, таких как A, B, C, SUM и START в программе на рис. 4.5. Символические имена могут иметь длину от 1 до 31 символа и включают буквы латинского алфавита (A – Z), цифры (0 – 9) и три специальных символа: знак доллара ($), точку (·) и знак подчеркивания (_). Однако символические имена не должны начинаться с цифры (0 – 9). Приведем примеры допустимых символических имен:

```
X
TAX
R2D2
THISISALONGSYMBOLICNAME
A_MORE_READABLE_LONG_NAME
JULY_4_1976
$14.96
```

А следующие имена недопустимы по перечисленным ниже причинам:

Недопустимое имя	Причина
4_JULY_1776	ИМЯ НАЧИНАЕТСЯ С ЦИФРЫ
WAGE RATE	ИМЯ СОДЕРЖИТ НЕДОПУСТИМЫЙ СИМВОЛ - ПРОБЕЛ
GROSS-PAY	ИМЯ СОДЕРЖИТ НЕДОПУСТИМЫЙ СИМВОЛ - ЗНАК МИНУС
$1,234,56	ИМЯ СОДЕРЖИТ НЕДОПУСТИМЫЙ СИМВОЛ - ЗАПЯТУЮ
THIS_NAME_IS_LONGER_THAN_31_CHARACTERS	ДЛИНА ИМЕНИ БОЛЬШЕ ЧЕМ 31 СИМВОЛ

Выбирать имена надо с осторожностью. Употребление осмысленных имен, таких как MAGE (заработная плата), HOURS_WORKED (отработанные часы) и MONTH (месяц), сделает программу на языке ассемблера более простой для отладки и сопровождения. Следует помнить, что знак доллара не следует использовать в символических именах. В операционной системе VAX/VMS знак доллара присутствует в именах, которые применяются для наименования специальных системных функций, имена, содержащие знак доллара, резервируются для названий системных программ. Если программист случайно использует одно из этих имен, это может привести к ошибкам или непредсказуемым последствиям. Например, по макроинструкции $EXIT_S генерируется вызов системной программы SYS $EXIT. Если пользователь определит в своей программе символический адрес SYS $EXIT, то макроинструкция $EXIT_S не сможет работать нормально.

ПОЛЕ МЕТКИ

Поле метки содержит метки или имена символических адресов. Каждая метка – это символическое имя (адрес) некоторой области памяти. (Обычно остальные поля в каждой строке описывают содержимое одного или нескольких байтов, начинающихся по этому адресу.) Метка должна быть допустимым символическим именем, завершающимся двоеточием (:)[1]. По соглашению метки обычно размещаются в начале строки, но в принципе могут находиться в любом месте строки, если только перед меткой нет никаких символов, кроме пробелов. В общем случае метка в некоторой строке необходима только тогда, когда на эту строку есть ссылки из других строк программы.

ПОЛЕ КОДА ОПЕРАЦИИ

Поле кода операции может начинаться с любой позиции строки после метки, если данный оператор имеет метку. В противном случае началом этого поля считается первый отличный от пробела символ. Однако по соглашению поле кода операции начинается с первой позиции табуляции, т.е. обычно с девятой позиции строки. В итоге коды операций начинаются с одной колонки, облегчая чтение программы. (Обычно позиции табуляции размещаются через 8 символов, т.е. в позициях 9, 17, 25, 33, . . .)

Если оператор начинается с метки, в имени которой содержится более 7 символов, эту метку можно поместить на отдельной строчке, чтобы все коды операций размещались в одной колонке. Например, для записи инструкции с меткой A_LONG_NAME можно использовать две строки:

```
                Код
Метка           операции        Операнды

A_LONG_NAME:
                ADDL2           READ_VALUE, X_TOTAL
```

Некоторые программисты применяют такой формат для всех инструкций с метками независимо от длины имени метки.

Поле кода операции может содержать три типа символических имен:

1. Мнемонические коды операций, такие как MOVL и SUBL2.
2. Директивы ассемблера, такие как .TITLE, .BLKB и .END.
3. Имена макроинструкций, такие как $EXIT_S.

Следует отметить, что все три типа имен удовлетворяют правилам, приведенным выше для символических имен, определяемых пользователем.

---

[1] Допустимы пробелы и символы табуляции между именем и двоеточием. Это позволяет программисту при желании выравнивать двоеточия в разных строках.

ПОЛЕ ОПЕРАНДОВ

Поле операндов может начинаться с любой позиции после поля кода операции и должно отделяться от последнего по крайней мере одним пробелом или символом табуляции. Однако по соглашению поле операндов начинается с 17-й позиции, т.е. 2-й позиции табуляции. Поле операндов состоит из некоторого числа операндов, разделенных запятыми. Число операндов зависит от типа инструкции, т.е. от содержимого поля кода операции. В директиве .BLKB требуется один операнд, в инструкции ADDB3 требуется три операнда, а в инструкции RET операнды вообще отсутствуют. Если пользователь укажет неверное число операндов, ассемблер выдаст сообщение об ошибке.

Ассемблер будет выдавать также сообщения об ошибках и при нелепых комбинациях кода операции и операндов. Например, в случае оператора

```
Метка           Код
                операции        Операнды

J:              .BLKL           #1
```

будет порождено сообщение об ошибке, поскольку знак номера (#) представляет собой спецификатор операнда, который допустим только тогда, когда в поле кода операции содержится код операции.

Аналогично оператор

```
Метка           Код
                операции        Операнды

FIRST:          MOVL            #5, #3
```

приведет к ошибке, поскольку бессмысленно заменять непосредственный операнд – число 3 непосредственным операндом – числом 5. (Значение константы не может меняться.)

КОММЕНТАРИИ

Комментарии должны начинаться со знака ; . Все, что расположено после точки с запятой, игнорируется, т.е. не рассматривается как часть программы. Комментарии могут начинаться с любой позиции строки после поля операндов (или после поля кода операции, если операнды не нужны). Когда для комментариев нужно использовать всю строку, точка с запятой ставится в начале строки.

4.6. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ АССЕМБЛЕРА

ПРЕОБРАЗОВАНИЕ ЧИСЕЛ

Ассемблер предоставляет программисту целый ряд дополнительных возможностей. Одной из них является преобразование чисел с различными основаниями. Обычно числа в программе на ассемблере интерпретируются как десятичные. Однако если перед числом стоят символы ^X, то это число будет рассматриваться как шестнадцатеричное. Например, действие следующих двух инструкций идентично:

```
MOVL    #64, K
MOVL    #^X40, K
```

Каждая из этих инструкций пересылает десятичное число 64 (шестнадцатеричное – 40) в длинное слово K. Именно по этой причине обозначение ^X использовалось, чтобы отличить в тексте шестнадцатеричные числа.

Аналогично если число начинается с символов ^B (binary), то оно интерпретируется как двоичное. В частности, все следующие инструкции эквивалентны:

```
MOVL    #24,K
MOVL    #^X18,K
MOVL    #^B11000,K
```

Каждая из этих трех инструкций пересылает десятичную константу 24 в длинное слово K. Программист может выбрать любое представление, которое более всего соответствует решаемой задаче.

Неправильная запись чисел может привести к ошибкам. Ниже приведены некорректно заданные числа и указаны допущенные в них ошибки.

Число	Причина ошибки
^B1012	ДВОИЧНЫЕ ЧИСЛА МОГУТ ВКЛЮЧАТЬ ТОЛЬКО ЦИФРЫ 0 И 1
1O	ИСПОЛЬЗОВАНА БУКВА O ВМЕСТО ЦИФРЫ 0
X1234ABCD	ПЕРЕД ЧИСЛОМ ПРОПУЩЕН СИМВОЛ (^)
123,456	ЧИСЛО СОДЕРЖИТ ЗАПЯТУЮ (,) - ЭТО НЕДОПУСТИМО

ДИРЕКТИВЫ .BYTE, .WORD И .LONG

Как уже говорилось в разд. 4.2, директива .BYTE используется для резервирования памяти и задания начального значения для данных в формате байта. Например, директивы

```
BOND:  .BYTE    007
MAX_BYTE:
       .BYTE    ^XFF
UPDOWN:
       .BYTE    ^B10101010
```

установят для байтов с адресами BOND, MAX_BYTE и UPDOWN начальные значения 7, ^XFF (десятичное число 255) и ^B10101010 (десятичное число 170) соответственно. В процессе ассемблирования каждой строки содержимое счетчика адресов будет увеличиваться на 1, а к программе в машинном коде будет добавляться один байт информации. Наибольшее допустимое значение операнда в директиве .BYTE равно 255.

Директива .WORD очень похожа на директиву .BYTE, за исключением того, что она резервирует и задает начальное значение для 16-битового слова. Например, операторы

```
YEAR:  .WORD    1984
MAX_WORD:
       .WORD    ^XFFFF
GROSS: .WORD    144
```

установят начальные значения слов с адресами YEAR, MAX_WORD и GROSS равными 1984, ^XFFFF (65535 – десятичное) и 144 соответственно. При первом проходе ассемблера каждый из этих операторов вызовет увеличение содержимого счетчика адресов на 2. Во время второго прохода операнды будут преобразованы, если это необходимо, в шестнадцатеричную форму и помещены в последовательно расположенных парах байтов программы. Заметим, что ^XFFFF, или 65535, – наибольшее допустимое значение операнда в директиве .WORD.

Аналогично для инициализации 32-битовых длинных слов используется директива .LONG. В отличие от других ЭВМ в ЭВМ семейства VAX директивы .BYTE, .WORD и .LONG могут располагаться в любом порядке. Например, операторы


```
HOMERS:
       .BYTE    61
MAX_LONG:
       .LONG    ^XFFFFFFFF
PERMUTES:
       .WORD    5040
```

установят для байта с адресом HOMERS начальное значение 61, для длинного слова MAX_LONG – значение ^XFFFFFFFF, которое приблизительно равно 4 млрд, и для слова PERMUTES – значение 5040. При обработке этих операторов значение счетчика адресов увеличится на 7, при этом будет сгенерировано 7 байтов машинного кода.

Существуют два различных способа присваивания байту, слову или длинному слову некоторого значения. Например, предположим, что данное слово COUNT должно содержать число 10. Конечно, это значение можно было бы задать с помощью директивы

```
COUNT:    .LONG   10
```

Этот же результат можно получить, используя два оператора:

```
COUNT:    .BLKL   1
          ...
          MOVL    #10,COUNT
```

При первом способе инициализация COUNT осуществляется только один раз во время ассемблирования. При применении второго способа инструкцию MOVL можно поместить в цикл, тогда при каждом проходе цикла значение COUNT будет установлено равным 10. Второй способ аналогичен использованию операторов COUNT=10 или COUNT:=10; языков Фортран или Паскаль. Присваивание значений первым способом аналогично действию оператора Фортрана DATA COUNT/10/. В стандартном языке Паскаль нет аналога первому способу.

При выборе одного из этих двух способов задания значения надо руководствоваться следующими соображениями.

1. Если COUNT содержит константу, которая не меняется программой в процессе выполнения, то лучше использовать директиву .LONG, а не инструкцию MOVL; так экономится и время выполнения, и память.

2. Если COUNT содержит переменное значение, изменяемое программой в процессе выполнения, то лучше использовать инструкцию MOVL, поскольку этот фрагмент текста программы может применяться как часть большой программы и вызываться несколько раз. Директива .LONG инициализирует соответствующим образом COUNT только перед выполнением.

ДИРЕКТИВЫ .BLKB, BLKW И .BLKL

Как уже упоминалось, директивы .BLKB и .BLKL используются при резервировании памяти для данных в формате байтов и длинных слов. Аналогично директива .BLKW используется для резервирования слов. Рассмотрим следующие операторы:

```
ALPHA:  .BLKB   1
BETA:   .BLKL   1
GAMMA:  .BLKW   1
THETA:  .BLKB   1
```

Эти четыре директивы будут резервировать память для двух байтов (ALPHA и THETA), одного слова (GAMMA) и одного длинного слова (BETA). Все вместе они вызовут увеличение значения счетчика адресов 1+1+2+4=8.

Как описано в гл. 7, различные директивы .BLKx могут использоваться для резервирования памяти под массивы. Например, оператор

```
VECTOR: .BLKL   1
```

означает, что VECTOR – это начальный адрес массива, состоящего из 20 длинных слов. Этот оператор увеличит значение счетчика адресов на 4*20=80.

ПРОСТЫЕ ВЫРАЖЕНИЯ

Ранее были описаны символические имена разного типа метки (символические адреса), мнемонические коды операций, директивы ассемблера и макроинструкции. Было бы желательно также применять символические имена для обозначения чисел в программе.

Замена чисел символическими именами осуществляется следующим образом: если в программе есть оператор

```
                Код
Метка    операции    Операнды

FACTORIAL_7=5040
```

то при каждом появлении в программе символического имени FACTORIAL_7 оно будет заменяться числом 5040. Например, для инструкции

```
                Код
Метка    операции    Операнды

FACTORIAL_7=5040
          . . .
         MOVL      #FACTORIAL_7, ANS
```

будет порожден тот же машинный код, что и для инструкции

```
MOVL    #5040, ANS
```

В процессе ассемблирования знак равенства (=) рассматривается аналогично двоеточию (:). При первом проходе ассемблера двоеточие указывает на то, что предшествующее ему имя надо занести в таблицу имен, определяемых пользователем вместе с текущим значением счетчика адресов. При обнаружении знака равенства также происходит занесение предшествующего ему имени в таблицу имен, определяемых пользователем. Однако значение, заносимое в эту таблицу, представляет собой число, стоящее справа от знака равенства. Символические имена, обозначающие числа (FACTORIAL_7), следует определять (FACTORIAL_7 = 5040) до того, как они будут использованы (MOVL # FACTORIAL_7, ANS). По соглашению все имена определяются обычно в начале программы сразу после директивы .TITLE. Определения имен (FACTORIAL_7 = 5040) могут помещаться в любом месте строки. Однако по соглашению они начинаются с начала строчки. Использование имен для обозначения чисел имеет два важных преимущества: программа становится наглядной и облегчается ее сопровождение. Для того чтобы убедиться, что использование поименованных констант облегчает чтение текста программы, рассмотрим следующий оператор:


```
MOVL    #12, COUNT
```

Он показывает, что осуществляется подсчет чего-то (COUNT – счетчик), но ничего не сообщает о том, что именно считается. При этом #12 может означать число единиц в некоторой дюжине, число дюймов в футе, число месяцев в году, число участников Тайной Вечери, число лет, начиная с которых человека можно считать подростком, и число нот в октаве. Если же пользователь определит символические имена, например, следующим образом:

```
DOZEN=12
INCHES_IN_FOOT=12
MONTHS_IN_YEAR=12
DISCIPLES_AT_SUPPER=12
PRE_TEEN_YEARS=12
NOTES_IN_OCTAVE=12
```

то он сможет применять в своей программе уже гораздо более осмысленные операторы, такие как

```
MOVL    #NOTES_IN_OCTAVE, COUNT
```

Наконец, использование символических имен для обозначения чисел позволяет облегчить сопровождение программы. Например, программа для бухгалтерского учета и управления ресурсами в кондитерском магазине, вероятно, неоднократно использует число 12 для преобразования в дюжины единиц. Если программа должна эксплуатироваться в другом магазине, где дюжина соответствует "чертовой дюжине", то необходимо просмотреть всю программу, чтобы заменить определенные константы 12 на 13. В программе, написанной надлежащим образом, следует изменить только один оператор DOZEN=12 на DOZEN==13. Если этот пример покажется слишком надуманным, то вспомните, как много программ требуется модифицировать для перевода из английской системы мер (фунты, футы, кварты) в метрическую систему (грамм, метр, литр). Использование имен для представления чисел намного упростит такое преобразование.

Оператор DOZEN=12 в языке ассемблера аналогичен следующим операторам языка Фортран:

```
Фортран                          Паскаль

PARAMETR (DOZEN=12)              VAR
                                  CONST DOZEN=12;
```

Определение имен в языке ассемблера с помощью оператора, подобного A=7, кажется очень похожим на операторы присваивания в языках высокого уровня (A=7 в Фортране или A:=7; в Паскале). Однако сходство обманчиво. В языке ассемблера *все выражения вычисляются во время ассемблирования.* При обработке оператора A=7 имя и значение 7 заносится в таблицу имен. Когда ассемблер находит в дальнейшем тексте имя A, он сразу заменяет его на значение 7.

В большинстве языков высокого уровня *выражения вычисляются во время выполнения программы.* В процессе компиляции оператор присваивания, такой как A=7 или A:=7, в действительности преобразуется в инструкцию, такую как MOVL#7, A. Во время выполнения эта инструкция осуществит пересылку числа 7 в ячейку памяти A.

ЧИСЛА И АДРЕСА

Как было упомянуто, таблица имен, определяемых пользователем, может содержать два различных типа объектов: имена, обозначающие адреса, и имена, представляющие числа. Адреса – это числа специального вида, на которые наложен целый ряд ограничений.

Например, адреса – это 32-битовые числа без знака. Кроме того, администраторы вычислительных систем VAX, используя особенности аппаратуры и программного обеспечения системы, обычно ограничивают диапазон адресов, доступных программисту. Нарушение этих ограничений при выполнении программы приводит к ошибке адресации. Числа, разумеется, таким ограничениям не подвергаются.

Хотя числа и адреса довольно сильно отличаются, имена, применяемые для их обозначения, очень похожи. Основное отличие между ними – то, что при определении символического адреса обычно используется двоеточие, а при определении имени для обозначения числа – знак равенства. В результате легко использовать число как адрес и наоборот. Например, в следующих операторах:

```
MILE=5280
        . . .
        MOVL    MILE, COUNT
```

программист случайно пропустил символ # в операнде MILE. В результате имя MILE, определенное как число (MILE=5280), используется в инструкции MOVL в качестве адреса. В процессе выполнения инструкции MOVL будет произведена попытка выборки длинного слова, расположенного по адресу 5280 (или ^X14A0). Последствия этого предсказать очень трудно. В зависимости от конкретной ситуации такая инструкция может вызвать сообщение об ошибке во время ассемблирования, сообщение об ошибке во время выполнения или просто будет получен неправильный результат.

Ошибка возникает и тогда, когда там, где не надо, указывается знак #. Например, в операторах

```
ALT:    .BLKL   1
        . . .
        MOVL    #ALT, COUNT
```

знак # ошибочно поставлен перед операндом ALT. При выполнении инструкции MOVL операнд COUNT получает в качестве значения адрес ALT, а не его содержимое. На самом деле встречаются ситуации, в которых адреса обрабатываются как данные и ЭВМ VAX имеют специальные инструкции выполнения операций над адресами (см. гл. 7). Поэтому обычно считается, что использование знака # вместе с символическим адресом свидетельствует о плохом стиле программирования.

УПРАЖНЕНИЯ 4.1

1. Выполните ассемблирование программ, приведенных на рис. 4.1 и 4.3, и сравните полученные листинги с листингами на рис. 4.2 и 4.4 соответственно. Выполните ассемблирование программы рис. 4.5 и сравните листинг с листингом, полученным при ассемблировании программы рис. 4.1.

2. Выполните три программы из п. 1 убедитесь, что в байте SUM содержится число ^X63 после завершения каждой из программ.

3. В следующей программе на языке ассемблера определите в шестнадцатеричном виде значения, которые будут содержаться в ячейках с символическими адресами от A до H до выполнения программы.


```
A:      .BYTE   20
B:      .BYTE   128
C:      .BYTE   -128
D:      .WORD   128
E:      .WORD   -128
F:      .LONG   ^X128
G:      .BYTE   ^B10000000
H:      .LONG   4096
        .ENTRY  INST,0
        $EXIT_S
        .END    INST
```

4. Используя возможности отладчика DEBUG, с помощью команд MACRO, LINK и RUN загрузите программы п. 3 упражнения в память. Вместо команды отладчика GO (которая запускает программу на выполнение) используйте команды E/BYTE A и E/LONG F для проверки своих ответов на п. 3 упражнения (см. разд. 1.5 и приложение А).

5. Ассемблируйте, скомпонуйте и выполните следующую программу, используя возможности отладчика DEBUG. Для того чтобы посмотреть программу в машинном коде, используйте команду отладчика E/BYTE FIRST:LAST, которая распечатывает байты памяти, расположенные между адресами FIRST и LAST. (Не забудьте, что программа загружена в память, начиная с адреса ^X0200.) Выделите коды операций, спецификаторы операндов, адреса и числа в полученной распечатке. Убедитесь в том, что относительная адресация со смещением, заданным в формате байта, обеспечивает указание соответствующих адресов данных, хотя программа начинается с адреса ^X0200.

```
J:          .BLKL   1
K:          .BLKL   1
DIF:        .BLKL   1
            .ENTRY  ADDRESS,0
FIRST:      MOVL    #512,J
            MOVL    #64,K
            SUBL3   K,J,DIF
LAST:       $EXIT_S
            .END    ADDRESS
```

*6. В программе рис. 4.3 ассемблер для адресации операндов A, B, C и SUM использовал относительную адресацию со смещением, заданным в формате длинного слова, поскольку эти символические адреса были определены в конце программы. Возможно ли разработать трехпроходный ассемблер, позволяющий использовать там, где допустимо, смещение в формате байта или слова – даже в случае, когда адреса встречаются до того, как они определены?

4.7. КОМПОНОВЩИК

Отлаживать одну большую программу обычно гораздо труднее, чем несколько небольших. Поэтому хорошие программисты обычно разбивают большую программу на несколько небольших *модулей*. При таком разбиении один из этих модулей указывается в качестве *основной* программы, которая первой во время выполнения получает управление. Остальные модули по очереди вызываются основной программой и называются *подпрограммами*. На ЭВМ VAX имеется несколько инструкций для вызова подпрограмм и возврата из них. Некоторые из этих инструкций будут описаны в следующей главе.

В принципе программные модули можно написать на разных языках программирования. Например, основная программа А может представлять собой программу в машинном коде, полученную в результате трансляции с языка Фортран, а подпрограмма В может быть результатом трансляции с языка ассемблера. Разбивая сложную задачу на небольшие подзадачи, программист может выбрать для решения каждой подзадачи наиболее подходящий язык.

Поскольку два модуля в одной и той же программе не могут занимать одно и то же место в памяти, необходимо, чтобы каждый модуль занимал свое собственное множество адресов. Такую работу не могут выполнить ни ассемблер, ни компиляторы языков высокого уровня, так как обычно они не располагают информацией о всех модулях, составляющих программу. Для разрешения этой проблемы применяется программа, называемая *компоновщиком* (linker).

Компоновщик выполняет две функции: *перемещает* различные модули, составляющие программу, так, чтобы они занимали непересекающиеся области памяти; в случае необходимости делает адреса, определенные в одном модуле, доступными для других модулей. (Такие адреса требуются для передачи управления из одного модуля в другой). Эта программа называется компоновщиком потому, что она компонует взаимозависимые модули машинного кода в одну программу, готовую к выполнению.

Обычно предполагается, что программа на ассемблере должна быть обработана компоновщиком, даже если она состоит только из одного модуля. Программа сразу после ассемблирования не может быть выполнена. Программа, полученная в результате работы ассемблера, называется *объектным модулем* (object module) и хранится в объектном *файле,* имеющем *тип* OBJ. Результат работы компиляторов языков высокого уровня, таких как Фортран или Паскаль, имеет такой же формат, и помещается в файлы типа OBJ.

Компоновщик считывает файлы типа OBJ, перемещает и связывает объектные модули, чтобы сформировать единую, готовую к выполнению программу. Программа в этом виде называется *выполняемым образом* (executable image) и хранится в *файле выполняемого* образа, имеющем *тип* EXE).

В конце гл. 3 было отмечено, что относительная адресация по сравнению с абсолютной дает кроме экономии памяти и времени, другие существенные преимущества: использование относительной адресации внутри модуля означает то, что относительные адреса не должны модифицироваться при перемещении этого модуля. Внутри модуля расстояние (смещение) между инструкциями и данными, к которым они обращаются, не должно изменяться при перемещении модуля. Напротив, если в модуле используется абсолютная адресация, то при перемещении этого модуля все абсолютные адреса будут изменены. Поэтому абсолютные адреса в объектном модуле должны быть помечены ассемблером, для того чтобы компоновщик смог модифицировать их при создании выполняемого образа. (Модули, в которых не требуется модификация адресов при компоновке, называются *позиционно независимыми.* Эта важная тема подробно рассматривается в гл. 7 и 15.)

4.8. ВЫПОЛНЕНИЕ ПРОГРАММЫ

ПРИМЕР ПРОГРАММЫ

В качестве иллюстрации рассмотрим процесс ассемблирования, компоновки и выполнения программы, написанной на языке ассемблера. Предполагается, что читатель должен использовать терминал некоторого типа. В рассматриваемом примере программист зарегистрирован в системе под именем JOHNDOE. Предполагается, что он с помощью одного из редакторов текста, имеющихся на вычислительных системах VAX, создал исходный файл с именем SAMPLE.MAR. Если теперь программист введет с терминала команду DIRECTORY (каталог) или просто DIR, то на экране дисплея появится

```
$ DIR

Directory DISK$USER:[JOHNDOE]

SAMPLE.MAR;1

Total of 1 file.
$
```


Символы, которые вводит с клавиатуры пользователь, выделяются подчеркиванием. Знак доллара $ в последней строке служит признаком того, что операционная система VAX/VMS готова принять очередную команду.

Пользователь может посмотреть содержимое файла SAMPLE.MAR, выведя его на свой терминал с помощью команды TYPE:

```
$ TYPE SAMPLE.MAR
        .TITLE  SQUARES  -ТАБЛИЦА КВАДРАТНЫХ КОРНЕЙ-
        .DISABLE GLOBAL                         ;УКАЗАТЬ НЕОПРЕДЕЛЕННЫЕ ИМЕНА
;ВЫЧИСЛИТЬ ТАБЛИЦУ КВАДРАТНЫХ КОРНЕЙ С ПОМОЩЬЮ МЕТОДА КОНЕЧНЫХ РАЗНОСТЕЙ
;АВТОР - ДЖОН ДОУ
;ДАТА СОЗДАНИЯ - ИЮЛЬ 31, 1985
SECOND_DIF=2                                    ;КОНСТАНТА - ВТОРАЯ РАЗНОСТЬ
FIRST_DIF_INIT=1                                ;НАЧАЛЬНАЯ ПЕРВАЯ РАЗНОСТЬ
SQUARE_INIT=1                                   ;КОРЕНЬ ИЗ ЕДИНИЦЫ
FIRST_DIF:                                      ;ПЕРЕМЕННАЯ - ПЕРВАЯ РАЗНОСТЬ
        .BLKL   1
SQUARE_1:                                       ;СОДЕРЖИТ ЗНАЧЕНИЕ ПЕРВОГО КОРНЯ
        .BLKL   1
SQUARE_2:                                       ;СОДЕРЖИТ ЗНАЧЕНИЕ ВТОРОГО КОРНЯ
        .BLKL   1
SQUARE_3:                                       ;СОДЕРЖИТ ЗНАЧЕНИЕ ТРЕТЬЕГО КОРНЯ
        .BLKL   1
        .ENTRY  BEGIN,0                         ;ТОЧКА ВХОДА
        MOVL    #FIRST_DIF_INIT,FIRST_DIF       ;ЗАДАТЬ ПЕРВУЮ РАЗНОСТЬ
        MOVL    #SQUARE_INIT,SQUARE_1           ;ВЫЧИСЛИТЬ ПЕРВЫЙ КОРЕНЬ
        ADDL2   #SECOND_DIF,FIRST_DIF           ;СКОРРЕКТИРОВАТЬ ПЕРВУЮ РАЗНОСТЬ
        ADDL3   FIRST_DIF,SQUARE_1,SQUARE_2     ;ВЫЧИСЛИТЬ ВТОРОЙ КОРЕНЬ
        ADDL2   #SECOND_DIF,FIRST_DIF           ;СКОРРЕКТИРОВАТЬ ВТОРУЮ РАЗНОСТЬ
        ADDL3   FIRST_DIF,SQUARE_2,SQUARE_3     ;ВЫЧИСЛИТЬ ТРЕТИЙ КОРЕНЬ
        $EXIT_S                                 ;ЗАВЕРШИТЬ ПРОГРАММУ
        .END    BEGIN
$
```

Эта программа вычисляет квадраты чисел от единицы до трех с помощью метода разностей. В основу этого метода положен тот факт, что разности между квадратами соседних чисел натурального ряда образуют ряд 1, 3, 5, 7, . . . Заметим, что разность между смежными членами последнего ряда постоянна и равна 2. Иллюстрация этого метода приведена ниже.

Квадраты	0	1	4	9	16	. . .
Первая разность		1	3	5	7	. . .
Вторая разность			2	2	2	. . .

Для вычисления каждого следующего члена ряда квадратов надо прибавить 2 к последней вычисленной разности (например, 7 + 2 = 9). Полученный результат надо прибавить к последнему вычисленному квадрату (9 + 16 = 25).

АССЕМБЛИРОВАНИЕ ПРОГРАММЫ

Ассемблирование программы, написанной на языке ассемблера, производится по команде $MACRO/DEBUG/LIST SAMPLE. Параметры /DEBUG и /LIST, следующие за командой MACRO, устанавливают дополнительные режимы работы ассемблера. Параметр /DEBUG информирует ассемблер о том, что пользователю потребуются средства отладки во время выполнения программы. Параметр /LIST указывает ассемблеру, что требуется выдать листинг программы как в машинном коде, так и на языке ассемблера.

Результат выполнения команды MACRO можно обнаружить, просмотрев каталог:

```
$ MACRO/DEBUG/LIST SAMPLE
$ DIR

Directory DISK$USER:[JOHNDOE]

SAMPLE.LIS;1          SAMPLE.MAR;1          SAMPLE.OBJ;1

Total of 3 files.
$
```

Новый файл SAMPLE.OBJ содержит вариант программы в машинном коде. Не надо пытаться вывести этот файл на терминал, он имеет не символьный, а двоичный формат. Еще один файл, SAMPLE.LIS, содержит листинг программы на языке ассемблера. По умолчанию ассемблер присваивает объектному файлу тип .OBJ, файлу листинга – тип .LIS. Так как файл SAMPLE.LIS – символьный, его содержимое можно вывести на терминал пользователя:

```
$ TYPE SAMPLE.LIS

SQUARES                    -ТАБЛИЦА КВАДРАТНЫХ КОРНЕЙ-          24-JUN-1985 20:00:43  VAX-11 Macro V03-00        Page  1
                                                                 24-JUN-1985 19:54:02  DISK$USER:[JOHNDOE]SAMPLE.MAR;1 (1)

                                0000     1        .TITLE  SQUARES  -ТАБЛИЦА КВАДРАТНЫХ КОРНЕЙ-
                                0000     2        .DISABLE GLOBAL                          ;УКАЗАТЬ НЕОПРЕДЕЛЕННЫЕ ИМЕНА
                                0000     3 ;ВЫЧИСЛИТЬ ТАБЛИЦУ КВАДРАТНЫХ КОРНЕЙ С ПОМОЩЬЮ МЕТОДА КОНЕЧНЫХ РАЗНОСТЕЙ
                                0000     4 ;АВТОР - ДЖОН ДОУ
                                0000     5 ;ДАТА СОЗДАНИЯ - ИЮЛЬ 31, 1985
                        00000002 0000     6 SECOND_DIF=2                                   ;КОНСТАНТА - ВТОРАЯ РАЗНОСТЬ
                        00000001 0000     7 FIRST_DIF_INIT=1                               ;НАЧАЛЬНАЯ ПЕРВАЯ РАЗНОСТЬ
                        00000001 0000     8 SQUARE_INIT=1                                  ;КОРЕНЬ ИЗ ЕДИНИЦЫ
                                0000     9 FIRST_DIF:                                     ;ПЕРЕМЕННАЯ - ПЕРВАЯ РАЗНОСТЬ
                        00000004 0000    10        .BLKL   1
                                0004    11 SQUARE_1:                                      ;СОДЕРЖИТ ЗНАЧЕНИЕ ПЕРВОГО КОРНЯ
                        00000008 0004    12        .BLKL   1
                                0008    13 SQUARE_2:                                      ;СОДЕРЖИТ ЗНАЧЕНИЕ ВТОРОГО КОРНЯ
                        0000000C 0008    14        .BLKL   1
                                000C    15 SQUARE_3:                                      ;СОДЕРЖИТ ЗНАЧЕНИЕ ТРЕТЬЕГО КОРНЯ
                        00000010 000C    16        .BLKL   1
                            0000 0010    17        .ENTRY  BEGIN,0                         ;ТОЧКА ВХОДА
                EA AF   01   D0 0012    18        MOVL    #FIRST_DIF_INIT,FIRST_DIF       ;ЗАДАТЬ ПЕРВУЮ РАЗНОСТЬ
                EA AF   01   D0 0016    19        MOVL    #SQUARE_INIT,SQUARE_1           ;ВЫЧИСЛИТЬ ПЕРВЫЙ КОРЕНЬ
                E2 AF   02   C0 001A    20        ADDL2   #SECOND_DIF,FIRST_DIF           ;СКОРРЕКТИРОВАТЬ ПЕРВУЮ РАЗНОСТЬ
        E3 AF   E1 AF   DF AF   C1 001E    21        ADDL3   FIRST_DIF,SQUARE_1,SQUARE_2     ;ВЫЧИСЛИТЬ ВТОРОЙ КОРЕНЬ
                D7 AF   02   C0 0025    22        ADDL2   #SECOND_DIF,FIRST_DIF           ;СКОРРЕКТИРОВАТЬ ВТОРУЮ РАЗНОСТЬ
        DC AF   DA AF   D4 AF   C1 0029    23        ADDL3   FIRST_DIF,SQUARE_2,SQUARE_3     ;ВЫЧИСЛИТЬ ТРЕТИЙ КОРЕНЬ
                                0030    24        $EXIT_S                                 ;ЗАВЕРШИТЬ ПРОГРАММУ
                                0039    25        .END    BEGIN

SQUARES                    -ТАБЛИЦА КВАДРАТНЫХ КОРНЕЙ-          24-JUN-1985 20:00:43  VAX-11 Macro V03-00        Page  2
Symbol table                                                     24-JUN-1985 19:54:02  DISK$USER:[JOHNDOE]SAMPLE.MAR;1 (1)

BEGIN           00000010 RG D  01
FIRST_DIF       00000000 R  D  01
FIRST_DIF_INIT = 00000001    D
SECOND_DIF     = 00000002    D
SQUARE_1        00000004 R  D  01
SQUARE_2        00000008 R  D  01
SQUARE_3        0000000C R  D  01
SQUARE_INIT    = 00000001    D
SYS$EXIT        ********  G    01

                                        +----------------+
                                        ! Psect synopsis !
                                        +----------------+
                                (данная секция листинга пропущена)
                                    +-----------------------+
                                    ! Performance indicators !
                                    +-----------------------+
                                (данная секция листинга пропущена)
                                    +-------------------------+
                                    ! Macro library statistics !
                                    +-------------------------+
                                (данная секция листинга пропущена)

There were no errors, warnings or information messages.
/DEBUG/LIST SAMPLE
$
```


Листинг ассемблерной программы и программы в машинном коде достаточно понятен, за исключением строк 18, 19, 20 и 22. В соответствии с изложенным выше можно было ожидать, что при обработке оператора ассемблера

```
MOVL    #FIRST_DIF_INIT,FIRST_DIF
```

будет сгенерирован код

Операнд	Операнд	Код операции	Адрес
EE AF	00000001	8F D0	0012

где ^X0012 – начальный адрес, ^X8F – спецификатор операнда (байт) непосредственной адресации, ^X00000001 – число, соответствующее имени FIRST_DIF_INIT. Однако в строке 18 сгенерирован следующий код:

Операнд	Операнд	Код операции	Адрес
EA AF	01	D0	0012

Вообще говоря, существует два способа задания непосредственных операндов, требующих меньше и больше памяти. Первый – наиболее экономный способ применим только для небольших чисел от 0 до 63. Любые другие числа задаются вторым способом со спецификатором операнда ^X8F. Первый способ задания называется *литеральным режимом* адресации и позволяет хранить число непосредственно в байте спецификатора операнда. Для задания таких чисел используются спецификаторы операндов со значениями от ^X00 до ^X3F. Более подробно литеральный режим адресации описан в гл. 7.

После распечатки программы ассемблер печатает таблицу имен, определенных пользователем. Строка

```
FIRST_DIF        00000000   R   D   01
```

в этой таблице сообщает, что значениям символического имени FIRST_DIF является ^X00000000. Стоящая далее буква R показывает, что FIRST_DIF – *переместимый адрес*, т.е. адрес будет настраиваться при перемещении программы. Буква D означает, что это символическое имя доступно отладчику. Число 01 – ссылка на *программную секцию* (см. гл. 9).

Переместимый адрес BEGIN отличается от остальных адресов в программе, поскольку он определен директивой .ENTRY. Это означает, что хотя адрес BEGIN определяется в данной программе, но на него можно ссылаться в других программах (например, из операционной системы VAX/VMS). Символические имена, которые определены в данной программе, но доступны другим программам, называются *глобальными именами* (global symbol). В данном случае буквы RG означают, что BEGIN – это переместимое глобальное имя. В последней строке таблицы имен содержится запись

```
SYS$EXIT         ********   G       01
```

Это имя генерируется макроинструкцией $EXIT_S, которая используется для возврата управления операционной системе VAX/VMS. Макроинструкция $EXIT_S в действительности генерирует инструкции, передающие управление по адресу SYS$EXIT, значение которого определяется операционной системой VAX/VMS. Поскольку адрес SYS$EXIT используется в этой программе, а определен в другой, он обозначается как глобальный (G). Символы ******** означают, что имя SYS$EXIT в настоящий момент не определено. Более подробно глобальные имена описаны в гл. 9.

Как следует из последней строки таблицы имен, ассемблер допускает наличие неопределенных имен, таких как SYS$EXIT. По умолчанию ассемблер считает неопределенные имена глобальными и предполагает, что компоновщик где-нибудь найдет значение для этих

имен. Это свойство, хотя и является полезным, приводит к тому, что некоторые ошибки пользователя, такие как неправильно заданные имена, ассемблер не обнаруживает. Данную возможность ассемблера можно исключить с помощью помещаемого в начало программы оператора

```
.DISABLE   GLOBAL  ;УКАЗАТЬ НЕОПРЕДЕЛЕННЫЕ ИМЕНА
```

При наличии такого оператора ассемблер генерирует сообщение об ошибке каждый раз, когда обнаруживает неопределенное имя в программе пользователя. (Имя SYS$EXIT не вызовет сообщения об ошибке, поскольку в макроинструкции $EXIT_S оно определено как внешнее.)

Теперь можно описать действия, выполняемые при указании параметра /DEBUG команды MACRO. Когда этот параметр отсутствует, то в создаваемый объектный файл ассемблер включает информацию только о глобальных именах. Оставшаяся часть таблицы имен теряется с окончанием ассемблирования. Параметр /DEBUG указывает ассемблеру на необходимость сохранить в объектном файле таблицу имен целиком. Это позволяет программисту использовать при отладке символические имена.

КОМПОНОВКА ПРОГРАММЫ

Программа компонуется по команде операционной системы VAX/VMS LINK/DEBUG SAMPLE. Результат выполнения команды LINK обнаруживается в каталоге пользователя:

```
$ LINK/DEBUG SAMPLE
$ DIR

Directory DISK$USER:[JOHNDOE]

SAMPLE.EXE;1          SAMPLE.LIS;1          SAMPLE.MAR;1          SAMPLE.OBJ;1

Total of 4 files.
$
```

Новый файл SAMPLE.EXE – это выполняемый вариант пользовательской программы в машинном коде. В процессе преобразования объектной программы SAMPLE.OBJ в выполняемую программу SAMPLE.EXE компоновщик настраивает программу так, чтобы она начиналась с адреса ^X0200, а не с адреса ^X0000. Процесс настройки очень прост, поскольку в программе используется только относительная адресация. Причины, по которым производится эта настройка, описаны в разделе, посвященном ошибкам программирования и отладке. Параметр /DEBUG в команде LINK указывает на то, что программисту потребуются средства отладки при выполнении программы SAMPLE.EXE.

ВЫПОЛНЕНИЕ ПРОГРАММЫ

Программу пользователя можно запустить на выполнение, введя команду RUN SAMPLE. Так как при компановке к программе были присоединены средства отладки (был указан параметр /DEBUG в команде LINK), то теперь управление передается не пользовательской программе, а программе-отладчику DEBUG. (Отладчик можно также активизировать командой RUN/DEBUG SAMPLE.) Отладчик выводит на терминал две строки информации, а затем сообщение DBG>:

```
$ RUN SAMPLE

        VAX-11 DEBUG Version V3.4-2

%DEBUG-I-INITIAL, language is MACRO, module set to 'SQUARES'
DBG>
```


Точно так же, как операционная система VAX/VMS использует символ $ как приглашение к вводу следующей команды операционной системы, так и отладчик использует сообщение DBG>: для указания того, что пользователю следует вводить команду отладчика. Обычно в этот момент вводят команду GO, по которой начинается выполнение программы пользователя.

```
DBG>GO
routine start at SQUARES\BEGIN
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

На этом этапе обычно просматривают содержимое памяти по команде E (Examine – посмотреть), например

```
DBG>E SQUARE_1
SQUARES\SQUARE_1:       00000001
DBG>E SQUARE_2
SQUARES\SQUARE_2:       00000004
DBG>E/DEC SQUARE_3
SQUARES\SQUARE_3:       9
DBG>
```

По команде E SQUARE_1 выводится на терминал содержимое длинного слова SQUARE_1 в программе SQUARES (SQUARES\SQUARE_1), оно равно ^X00000001; по второй команде E SQUARE_2 выводится содержимое длинного слова SQUARE_2, равное ^X00000004. Наконец, по команде E/DEC SQUARE_3 выводится содержимое адреса SQUARE_3 в десятичном виде; число 9 соответствует ожидаемому результату. Отметьте, что в то время, как по умолчанию в ассемблере используется десятичная система счисления, в отладчике по умолчанию числа выводятся в шестнадцатеричном виде.

С помощью отладчика можно посмотреть область памяти, где расположены инструкции. Например, по команде

```
DBG>E BEGIN+2
SQUARES\BEGIN+2:        0EAAF01D0
DBG>
```

выдается машинный код первой инструкции программы. (Так как BEGIN – точка входа программы, этот адрес является адресом 16-битовой маски (два байта). Первая инструкция расположена непосредственно за маской, поэтому ее адрес BEGIN+2. Инструкция располагается в длинном слове справа налево: ^XD0 – код операции, ^X01 – литеральный операнд ^X01, ^XAF – спецификатор второго операнда и ^XOE – смещение, заданное в формате байта.

Отладчик не может определить, что содержится по данному адресу: байт, слово, длинное слово или инструкция. Поэтому, если не указано что-либо другое, по команде EXAMINE отображается содержимое длинного слова. Содержимое байтов и слов можно посмотреть с помощью команд E/BYTE и E/WORD соответственно. Командой E/INST можно посмотреть инструкцию в символьном формате. Поскольку в команде MACRO присутствовал параметр /DEBUG, отладчику доступна таблица имен, а это означает, что он может частично восстановить по машинному коду заданный оператор исходной ассемблерной программы, например

```
DBG>E/INST BEGIN+2
SQUARES\BEGIN+2:        MOVL    #01,B^SQUARES\FIRST_DIF
DBG>
```

Обратите внимание, что второй операнд инструкции начинается с символов B^. Эти символы означают, что для имени FIRST_DIF в программе SQUARES используется относительная адресация с заданным в формате байта смещением.

Команда Examine может использоваться для просмотра содержимого памяти по абсолютным адресам. Следующие шесть команд позволяют посмотреть содержимое различных ячеек памяти с адресами от ^X0000 до ^X020A:

```
DBG>E 0
%DEBUG-E-NOACCESSR, no read access to virtual address 00000000
DBG>E 1FF
%DEBUG-E-NOACCESSR, no read access to virtual address 000001FF
DBG>E 200
SQUARES\FIRST_DIF:       00000005
DBG>E 204
SQUARES\SQUARE_1:        00000001
DBG>E/WORD 210
SQUARES\BEGIN:   0000
DBG>E/INST 212
SQUARES\BEGIN+2:         MOVL     #01,B^SQUARES\FIRST_DIF
DBG>EXIT
$
```

Как показывают первые две команды, невозможно посмотреть содержимое памяти с адреса ^X0000 до адреса ^X01FF. Программисту эти адреса не нужны, поскольку компоновщик размещает программу с адреса ^X0200. Третья команда E 200 выводит на экран содержимое длинного слова с символическим адресом FIRST_DIF. В результате перемещения программы абсолютный адрес FIRST_DIF равен ^X0200. При указании соответствующих адресов выводится содержимое ячеек с именами SQUARE_1, BEGIN и BEGIN+2. Наконец, команда отладчика EXIT возвращает управление операционной системе VAX/VMS.

ОШИБКИ ПРОГРАММИРОВАНИЯ И ОТЛАДКА

Ошибки программирования в ассемблерной программе могут вызвать сообщения об ошибках разных типов. Чтобы продемонстрировать, как проявляются ошибки, в строку 19 предыдущей программы будет внесено несколько ошибок. Строка без ошибок выглядит следующим образом:

```
MOVL     #SQUARE_INIT,SQUARE_1     ;ВЫЧИСЛИТЬ ПЕРВЫЙ КОРЕНЬ
```

В приведенной ниже строке вместо кода операции MOVL ошибочно указан код MOV:

```
MOV      #SQUARE_INIT,SQUARE_1     ;ВЫЧИСЛИТЬ ПЕРВЫЙ КОРЕНЬ
```

В результате ассемблер не может найти значение для имени MOV и выдает следующее сообщение об ошибке (неопределенный оператор):


```
$ MACRO SAMPLE

korenx                                  0016   19        MOV     #SQUARE_INIT,SQUARE_1
%MACRO-E-UNRECSTMT, Unrecognized statement

There were 1 error, 0 warnings and 0 information messages, on lines:
  19 (1)
```

Сходное сообщение будет выдано, если после метки будет пропущено двоеточие.

Весьма различные ошибки могут возникать при использовании числа в качестве адреса. Например, в следующей строке программист забыл поставить знак # перед именем SQUARE_INIT:

```
MOVL     SQUARE_INIT,SQUARE_1     ;ВЫЧИСЛИТЬ ПЕРВЫЙ КОРЕНЬ
```

Ни ассемблер, ни компоновщик не сообщает об ошибке. Во время выполнения программа попытается произвести выборку длинного слова, расположенного по абсолютному адресу ^X00000001. Так как пользователю адреса, меньшие ^X0200, недоступны, то выдается следующее сообщение об ошибке (нарушение доступа):

```
$ MACRO SAMPLE
$ LINK SAMPLE
$ RUN SAMPLE
%SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual address=00000001, PC=0000021
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name                    line       rel PC    abs PC

SQUARES         . BLANK .                                  0000001A  0000021A
```

По этому сообщению программист должен найти оператор, который, по всей вероятности, сгенерировал запрещенный виртуальный адрес ^X00000001. После символов abs PC распечатывается адрес ^X0000021A, который содержался в программном счетчике в момент обнаружения ошибки. Просмотр содержимого памяти вблизи адреса ^X021A с помощью отладчика может помочь локализовать ошибку. Адрес ^X0000001A, выведенный после символов rel PC, – это тот же адрес, но относительно начала модуля. Просмотр адресов вблизи адреса ^X001A по распечатке ассемблерной программы также может помочь обнаружить ошибку. Адрес abs PC больше адреса rel PC на ^X200, так как компоновщик настроил модуль на адрес ^X0200.

К генерации младших запрещенных адресов может привести множество ошибок программирования, в том числе и пропуск символа #. Если программа пытается обратить по адресу, меньшему ^X0200, то операционная система VAX/VMS прекращает выполнение программы. Фактически поэтому в операционной системе VAX/VMS перемещается начало основной программы в адрес ^X0200, а младшие адреса делаются недоступными для пользователя.

Использование адресов в качестве чисел может привести к тому, что программы будут выполняться без ошибок, но при этом будут выдавать неправильные результаты. Например, рассмотрим, что произойдет, если в строке 6 заменить SQUARE_INIT = 1 на SQUARE_INIT=^X210. Если ассемблировать, скомпоновать и запустить на выполнение такую программу, то она благополучно завершится без выдачи сообщений об ошибке. Однако значения переменных SQUARE_1, SQUARE_2 и SQUARE_3 будут совершенно неверными. Пусть программист, используя число как адрес, случайно сгенерировал допустимый адрес. Программа пытается произвести выборку длинного слова, расположенного по абсолютному адресу ^X0210. Так как программа начинается с адреса ^X0200, то искомое длин-

ное слово размещается на ^X10 байт дальше от начала программы и содержит ^X01D00000. Первые четыре шестнадцатеричные цифры ^X01D0 – это первые байты машинного кода инструкции (см. строку 18 распечатки), а последние четыре цифры ^X0000 – это 16-битовая маска, сгенерированная в строке 17.

Отладка ассемблерных программ более сложна, чем отладка программ на языках высокого уровня. Иногда программист затрудняется найти ошибку, и поэтому приходится проверять ассемблерную программу символ за символом. Особенно осторожно следует обходиться с символами, вызывающими генерацию спецификаторов операндов, например со знаком #.

УПРАЖНЕНИЯ 4.2

1. Определите с помощью команд отладчика E 1000, E 2000, E 4000, E 8000, E 10000 и т. д. самый старший адрес, к которому вам разрешен доступ на вашей вычислительной системе VAX. Можете ли вы использовать отладчик DEBUG для преобразования полученных шестнадцатеричных значений в десятичные числа?

2. Ассемблируйте, скомпонуйте и выполните программу, которая вычисляет квадраты чисел 1, 2, 3, и с помощью отладчика убедитесь в ее правильной работе. Затем повторите этот процесс, внося в программу одну из нижеперечисленных ошибок. Опишите, к чему приводит каждая из ошибок. Порождает ли эта ошибка сообщения при ассемблировании, компоновке, выполнении или просто приводит к неправильным результатам?

а) удалите двоеточие после метки FIRST_DIF в строке 9;

б) замените метку в строке 9 с FIRST_DIF на FIRST_DIFF;

в) удалите директиву .DISABLE GLOBAL в строке 2 и повторите пункт б);

г) удалите оператор .BLKL 1 в строке 14 (одной строкой ниже метки SQUARE_2);

д) удалите оператор $EXIT_S в строке 24.

3. Измените программу, вычисляющую квадраты чисел 1, 2, 3 так, чтобы она вычисляла квадраты чисел от 1 до 6.

4. Напишите программу по образцу программы SQUARES, которая вычисляет кубы целых чисел от 1 до 6. (*Подсказка*: для описания алгоритма выпишите ряд кубов 1, 8, 27, 64, 125, . . . , а затем на бумаге вычислите разности первого, второго и третьего порядков.)

ГЛАВА 5. УПРАВЛЕНИЕ ПОСЛЕДОВАТЕЛЬНОСТЬЮ ВЫПОЛНЕНИЯ ПРОГРАММЫ

5.1. ВВЕДЕНИЕ

В языках высокого уровня предусмотрены средства управления последовательностью выполнения программы, которые позволяют программисту задавать выполнение тех или иных операторов в зависимости от определенных условий, а именно от значений, которые принимают соответствующие переменные в ходе выполнения программы. В языках Паскаль и Фортран такое управление обеспечивается управляющими конструкциями типа IF-THEN-ELSE. Операторы, образующие эти конструкции, транслируются в инструкции машинного языка, называемые условными переходами, которые входят в набор инструкций ЭВМ VAX[1].

В языках высокого уровня предусмотрены средства для повторения выполнения отдельных частей программы. Повторяемые части программ называются *циклами*. В Паскале циклы создаются с помощью операторов FOR или WHILE. В Фортране для образования циклов используется оператор DO. Эти операторы могут транслироваться в те же инструкции условных переходов, которые используются и для реализации конструкций типа IF-THEN-ELSE. Однако в архитектуре ЭВМ VAX предусмотрены дополнительные инструкции, предназначенные специально для выполнения циклов. В этой главе показано, как записываются на языке ассемблера операторы условных переходов и операторы цикла.

Кроме того, в этой главе рассматривается использование шестнадцати регистров общего назначения процессора. Здесь же затрагивается вопрос организации подпрограмм для ЭВМ VAX, который детально рассматривается в гл. 9, а в данной главе будет разобран ряд простых примеров использования подпрограмм, в частности подпрограмм ввода и вывода чисел.

[1] Некоторые компиляторы с языка Паскаль транслируют программы на Паскале в программы на промежуточном языке, называемом p-кодом, которые затем интерпретируются. Но в системе VAX/VMS разработан компилятор с языка Паскаль, который транслирует программы на Паскале непосредственно в программы на машинном языке ЭВМ VAX.

5.2. ОПЕРАТОРЫ ПЕРЕХОДОВ

ПРИМЕР КОНСТРУКЦИИ IF-THEN

На рис. 5.1 представлены фрагменты программ на Фортране и Паскале, в которых из двух переменных A и B выбирается переменная с меньшим значением, после чего это значение присваивается переменной MINAB. Обратите внимание на то, что в начале обеих программ все указанные переменные объявлены как целые (INTEGER). Встретив в программе описание INTEGER, компилятор с языка Паскаль или Фортран резервирует для переменных MINAB, A, B, которые представляют собой символические адреса, 32-битовые длинные слова. В дальнейшем содержимое длинных слов MINAB, A, B будет рассматриваться как 32-битовые целые со знаком. Поскольку компиляторы языков Паскаль и Фортран в операционной системе VAX/VMS по умолчанию используют для хранения целых чисел длинные слова, в наших примерах почти для всех целых чисел также будут применяться длинные слова. Кроме того, все переменные, используемые во фрагментах программ на Паскале и Фортране, будут по умолчанию рассматриваться как 32-битовые целые со знаком.

Вышеуказанные фрагменты программ на Фортране и Паскале могли бы транслироваться во фрагмент программы на языке ассемблера, который приведен на рис. 5.2. Обратите внимание, что это – фрагмент программы, а не программа целиком. Для краткости в ней опущены некоторые директивы и инструкции языка ассемблера, а также макроинструкция $EXIT_S.

После комментария "Начало условия" следуют инструкции: CMPL A, B и BLEQNEXT. По инструкции CMPL производится сравнение содержимого длинных слов, а по инструкции BLEQ производится переход по условию "меньше или равно". Эти инструкции после трансляции преобразуются в последовательность инструкций машинного языка, согласно которым процессор должен сравнить содержимое длинных слов A и B, а затем перейти к инструкции, начинающейся с символического адреса NEXT в случае, если целое

Паскаль	Фортран
. . .	. . .
VAR MINAB, A, B : INTEGER;	INTEGER MINAB, A, B
. . .	. . .
MINAB := A;	MINAB = A

IF A > B THEN
  MINAB := B;
	IF (A.GT.B) THEN
	  MINAB = B
	ENDIF

. . .	. . .

Рис. 5.1. Пример условного перехода

```
          . . .
MINAB:   .BLKL   1              ;32-БИТОВОЕ ЦЕЛОЕ СО ЗНАКОМ
A:       .BLKL   1              ;32-БИТОВОЕ ЦЕЛОЕ СО ЗНАКОМ
B:       .BLKL   1              ;32-БИТОВОЕ ЦЕЛОЕ СО ЗНАКОМ
          . . .

         MOVL    A,MINAB       ;MINAB:=A
;НАЧАЛО БЛОКА УСЛОВИЯ
         CMPL    A,B           ;СРАВНИТЬ  A  И  B
         BLEQ    NEXT          ;И ПЕРЕЙТИ, ЕСЛИ B>A
         MOVL    B,MINAB       ;MINAB:=B
;КОНЕЦ УСЛОВИЯ
NEXT:    . . .
```

Рис. 5.2. Пример условного перехода реализованного на ассемблере VAX-11 MACRO

со знаком, содержащееся в длинном слове с адресом A, меньше или равно целому со знаком, содержашемуся в длинном слове B. Если содержимое длинного слова с адресом A больше содержимого длинного слова с адресом B, процессор выполнит инструкцию, непосредственно следующую за инструкцией BLEQ (в данном случае инструкцию MOVL B, MINAB).

Процесс выполнения условного перехода, организованного так, как показано на рис. 5.2, складывается из двух отдельных действий, для которых требуются две соответствующие инструкции. Сначала должно быть проведено сравнение значений, после чего на основе результата сравнения может произойти условный переход. В данном случае по инструкции CMPL (сравнить длинные слова) сравнивается содержимое длинных слов A и B. Затем по инструкции BLEQ (перейти, если меньше или равно) производится переход к ячейке с адресом NEXT, если условие выполняется (в данном случае содержимое слова A меньше или равно содержимому слова B). Заметьте, что в самой инструкции BLEQ не содержится информация о том, что с чем сравнивается. Предполагается, что этой инструкции всегда предшествует инструкция типа CMPL.

Обратите внимание на то, что проверяемое условие перехода в программе на ассемблере обратно условию перехода на Паскале и Фортране. Во всех трех программах программист хочет присвоить переменной MINAB значение B тогда, и только тогда, когда A больше B. В программах на Паскале и Фортране программист проверяет то условие ("A больше B"), при выполнении которого должно быть произведено какое-либо действие ("переменной MINAB присвоить значение Б"). В программах на языке ассемблера программист должен проверять обратное условие ("A меньше или равно B"), после чего делать условный переход, если это обратное условие выполняется.

В заключение отметим, что последняя строка фрагмента программы на языке ассемблера начинается с символического адреса NEXT. В языках высокого уровня, таких как Паскаль и Фортран, как правило, различают *имена переменных*, обозначающих ячейки памяти, содержащие данные, и *метки операторов*, которые указывают на ячейки памяти, содержащие инструкции. В языках ассемблера этого различия не делается. В процессе ассемблирования двоеточие, следующее за меткой NEXT в последней строке фрагмента на рис. 5.2, воспринимается как признак того, что метку NEXT следует поместить в таблицу имен как символический адрес. Таким образом, NEXT является начальным адресом инструкции, которая следует за инструкцией MOVL B, MINAB.

УСЛОВНЫЕ ПЕРЕХОДЫ ПРИ ИНТЕРПРЕТАЦИИ ОПЕРАНДОВ КАК ЧИСЕЛ СО ЗНАКОМ

Инструкция BLEQ относится к группе из шести инструкций условных переходов, представленных ниже, для которых операнды интерпретируются как числа со знаком.

Мнемоника	Код операции	Описание
BNEQ	12	Перейти, если первый операнд не равен (Not EQual) второму операнду
BEQL	13	Перейти, если первый операнд равен (EQuaL) второму операнду
BGTR	14	Перейти, если первый операнд больше (GReaTer) второго операнда
BLEQ	15	Перейти, если первый операнд меньше (Less) или равен (EQual) второму операнду
BGEQ	18	Перейти, если первый операнд больше (Greater) или равен (EQual) второму операнду
BLSS	19	Перейти, если первый операнд меньше (LeSS) второго операнда

Приведенные шесть инструкций в паре с предшествующей им инструкцией сравнения реализуют следующие шесть операций отношения языков Паскаль и Фортран:

Операция отношения в Паскале	=	<>	<	>=	<=	>
Операция отношения в Фортране	.EQ.	.NE.	.LT.	.GE.	.LE.	.GT.
Инструкция условного перехода ассемблера VAX	BEQL	BNEQ	BLSS	BGEQ	BLEQ	BGTR

Для каждой операции (и соответствующей ей инструкции перехода) существует обратная операция. Так, BLSS является обратной к BGEQ инструкцией, а BLEQ – обратной к BGTR инструкцией. Обратите внимание на то, что большое значение имеет порядок записи операндов в инструкции сравнения. Так, например, действие инструкций

```
CMPL    A,B
BGTR    ALPHA
```

существенно отличается от действия инструкций

```
CMPL    A,B
BGTR    ALPHA
```

По первой паре инструкций осуществляется переход, когда содержимое длинного слова, расположенного по адресу A, больше, чем содержимое длинного слова, расположенного по адресу B. А по второй паре инструкций осуществляется переход, когда содержимое длинного слова с адресом B больше, чем содержимое длинного слова с адресом A.

Фрагмент программы на языке ассемблера, показанный на рис. 5.2, может быть заменен следующим фрагментом:

```
         MOVL    A,MINAB       ;MINAB:=A
;НАЧАЛО БЛОКА УСЛОВИЯ
         CMPL    B,A           ;СРАВНИТЬ  B  И  A
         BGTR    NEXT          ;И ПЕРЕЙТИ, ЕСЛИ B > A
         MOVL    B,MINAB       ;MINAB:=B
;КОНЕЦ УСЛОВИЯ
NEXT:    . . .
```

Обратите внимание на то, что в этом примере в инструкции CMPL изменен порядок записи операндов (B, A вместо A, B) и при этом условие перехода также заменено на обрат-

ное (BGTR вместо BLEQ). Таким образом, приведенные на рис. 5.1 операторы Фортрана и Паскаля могут быть реализованы любым из двух представленных фрагментов программы на языке ассемблера.

ПРИМЕР КОНСТРУКЦИИ IF-THEN-ELSE

Фрагменты программ на Паскале и Фортране, показанные на рис. 5.1, могут быть переписаны с использованием конструкции типа IF-THEN-ELSE, как показано на рис. 5.3. Программа, имитирующая на языке ассемблера конструкцию IF-THEN-ELSE, показана на рис. 5.4.

В этой программе инструкции CMPL и BGTR осуществляют передачу управления по адресу AGTRB (часть ELSE конструкции управления), если содержимое длинного слова с адресом A больше содержимого длинного слова с адресом B. В противном случае процессор ЭВМ VAX выполнит инструкцию, непосредственно следующую за инструкцией BGTR, в этом примере – инструкцию MOVL A, MINAB. После инструкции MOVL процессор выполнит следующую инструкцию – BRB. Этот оператор осуществляет безусловную передачу управления по символическому адресу NEXT. Если бы инструкции BRB не было, то в программе была бы ошибка: после присвоения переменной MINAB значения A далее процессор выполнил бы оператор MOVL B, MINAB, в результате чего переменной MINAB было бы присвоено значение B. Инструкция безусловного перехода BRB (сокращение от BRunch with Byte displacement – перейти по адресу, заданному смещением в формате байта) аналогична безусловным операторам перехода GO TO в Паскале и Фортране. Объяснение названию этой инструкции будет дано при описании форматов инструкций переходов.

Отметим, что фрагмент программы на языке ассемблера, представленный на рис. 5.2, проще, чем фрагмент на рис. 5.4, поскольку в нем используется только одна инструкция перехода вместо двух. Кроме того, фрагмент на рис. 5.2 занимает меньший объем памяти и требует меньше времени для выполнения. В результате оптимизирующий компилятор выполнит трансляцию конструкции IF-THEN-ELSE (рис. 5.3) в машинный код так, как показано на рис. 5.2.

Паскаль

```
. . .

IF A <= B THEN
  MINAB := A;
ELSE
  MINAB := B;
. . .
```

Фортран

```
. . .

IF (A.LE.B) THEN
   MINAB = A
ELSE
   MINAB = B
ENDIF
. . .
```

Рис. 5.3. Измененный пример условного перехода

```
;НАЧАЛО БЛОКА УСЛОВИЯ  IF A > B
        CMPL    A,B             ;СРАВНИТЬ  A  И  B
        BGTR    AGTRB           ;ПЕРЕЙТИ,  ЕСЛИ A > B
;THEN
        MOVL    A,MINAB         ;MINAB:=A
        BRB     NEXT
;ELSE
AGTRB:  MOVL    B,MINAB         ;MINAB:=B
;КОНЕЦ УСЛОВИЯ
NEXT:   . . .
```

Рис. 5.4. Измененный вариант программы на ассемблере VAX-11 MACRO

ЛОКАЛЬНЫЕ МЕТКИ

При программировании представляется полезным выбирать для символических адресов имена, несущие смысловую нагрузку. Например, если длинное слово содержит значение общего дохода (GROSS WAGE) некоторого лица, то такие имена, как GWAGE, GROSS_WAGE или GWAGEL[1], более удачны, чем, например, имя K. Аналогично первой инструкции блока, вычисляющего удержания с зарплаты служащих (DEDUCTIONS), должно быть присвоено имя (метка) DEDUCT.

Однако в очень больших программах программисту трудно придумать имеющие смысл оригинальные символические имена. Большие ассемблерные программы могут содержать сотни символических имен, и программист может случайно использовать одинаковые символические имена для двух различных адресов, что приведет к сообщению об ошибке ассемблера. Имеется множество решений этой проблемы. Одним из них является разбиение большой программы на меньшие, которые могут ассемблироваться отдельно, а затем компоноваться в единую программу в машинном коде. Этот подход рассматривается в гл. 9. Другая возможность состоит в использовании длинных символических имен, таких как WAGE_LOOP или TAX_LOOP. Третья возможность заключается в использовании специальных символических имен, которые называются локальными метками.

Локальная метка является символическим адресом, состоящим из последовательности цифр, которая заканчивается знаком доллара, например 1$, 57$ или 999$. Ниже показан фрагмент программы, в котором используются локальные метки 10$, 20$ и 30$[2]. В данном фрагменте переменной MINONE присваивается значение, равное содержимому наименьшего из трех (A, B, C) длинного слова:

```
MINABC: CMPL    A,B
        BLSS    10$
        MOVL    B,MINONE
        BRB     20$
10$:    MOVL    A,MINONE
20$:    CMPL    C,MINONE
        BGTR    30$
        MOVL    C,MINONE
30$:
MINXYZ: . . .
```

Символы 10$, 20$ и 30$ называются локальными метками, потому что обращение к ним возможно только в области программы между адресами MINABC и MINXYZ. Другими словами, задание обычных символических имен MINABC и MINXYZ ограничивает область видимости локальных меток 10$, 20$ и 30$. Попытка перехода по адресу 10$ из инструкции, предшествующей метке MINABC или следующей за меткой MINXYZ строки, вызовет ошибку ассемблера, обусловленную появлением неопределенного имени.

Отметим, что метке 30$ соответствует пустая строка, после которой следует строка с меткой MINXYZ. Таким образом, обе метки указывают на одну ячейку памяти, а именно на начальный адрес следующей инструкции программы. Вместо инструкции BGTR 30$ можно было бы написать инструкцию BGTR MINXYZ – после ассемблирования результат был бы одинаковым. Однако использование имени 30$ более предпочтительно, так

---

[1] Некоторые программисты предпочитают в имена включать буквы L, W или B для указания того, что определяет данное имя: длинное слово, слово или байт.

[2] Принято выбирать для таких символических имен числа, кратные 10. Это позволяет делать вставки с новыми символическими именами, сохраняя упорядоченность чисел. Фактические значения чисел не играют никакой роли, за исключением того, что проще отлаживать программу с упорядоченной последовательностью меток.

как MINXYZ – метка, которая не имеет отношения к фрагменту программы MINABC и является началом некоторой самостоятельной части программы.

Так как локальные метки определены только в области программы между двумя метками, задаваемыми обычными именами (такими как MINABC и MINXYZ), они могут использоваться в программе многократно. Например, ниже показан фрагмент программы, в котором переменной MINONE присваивается минимальное из значений A, B и C, после чего переменной MINTWO присваивается минимальное из значений операндов X, Y и Z:

```
MINABC:  CMPL    A, B              ; MINONE=MIN( A, B, C)
         BLSS    10$
         MOVL    B, MINONE
         BRB     20$
10$:     MOVL    A, MINONE
20$:     CMPL    C, MINONE
         BGTR    30$
         MOVL    C, MINONE
30$:
MINXYZ:  CMPL    X, Y              ; MINTWO=MIN( X, Y, Z)
         BLSS    10$
         MOVL    Y, MINTWO
         BRB     20$
10$:     MOVL    X, MINTWO
20$:     CMPL    Z, MINTWO
         BGTR    30$
         MOVL    Z, MINTWO
30$:
NEXTMIN: . . .
```

Поскольку локальные метки 10$, 20$ и 30$, используемые в первой половине фрагмента программы, становятся неопределенными, как только встречается метка MINXYZ, их применение во второй половине данного фрагмента не приведет к ошибкам, вызванным повторным определением.

В дальнейшем локальные метки будут применяться во многих фрагментах программ. Однако, если в программном блоке реализована функция, к которой происходит обращение из других частей программы, блоку должно быть присвоено обычное символическое имя (например, MINAB или MINXYZ в предыдущем примере). Хороший стиль программирования исключает чрезмерное увлечение локальными метками, так как большое количество меток в листинге на несколько страниц делает программу трудночитаемой. В основном они предназначены только для задания тривиальных имен (меток) для перехода вперед или назад на небольшое количество программных строк. Локальные метки также широко используются в макроинструкциях (см. в гл. 10).

5.3. КОДЫ УСЛОВИЙ

СЕМЕЙСТВО ИНСТРУКЦИЙ CMP

Семейство инструкций CMP можно описать следующим образом:

Мнемоника	Код операции	Операнды	Операция
CMPB	91	src1, src2	src1 - src2
CMPW	B1		
CMPL	D1		

В него входят инструкции сравнения значений байтов, слов и длинных слов. Например, пара инструкций

```
CMPB    BYTE_1, BYTE_2
BLSS    10$
```

выполняет переход к метке 10$, если число со знаком, содержащееся в байте BYTE_1, меньше, чем число со знаком в байте BYTE_2.

Формат этих инструкций аналогичен формату инструкций групп MOV и SUB. (Отметим, что коды операций для инструкций MOVB, MOVW и MOVL – это ^X90, ^XB0 и ^XD0 соответственно.) За кодом операции следуют два операнда. Каждому операнду предшествует байт спецификатора операнда, который определяет способ адресации операнда. Инструкции семейства CMP могут использовать любой из режимов адресации, включая литеральную, непосредственную, относительную и абсолютную адресацию.

Литеральные или непосредственные операнды могут применяться в качестве и первого, и второго операнда. Например, инструкции

```
CMPW    #21, ALPHA
BLSS    20$
```

осуществят переход к метке 20$, если число 21 меньше содержимого слова с адресом ALPHA. По инструкции

```
CMPL    WAGE, #1000
BLSS    NOTAX
```

выполнится переход к метке NOTAX, если содержимое длинного слова с адресом WAGE меньше 1000.

Как указано в описании семейства инструкций CMP, инструкция сравнения на самом деле осуществляет вычитание. В отличие от семейства инструкций SUB, вычитающих первый операнд из второго, инструкции семейства CMP вычитают второй операнд из первого. (Обратите внимание, что в описании указана операция src 1 – src 2.) Кроме того, опять в отличие от семейства инструкций SUB, результат вычитания при выполнении инструкции CMP не сохраняется. Однако процессор сохраняет четыре бита информации или четыре признака, характеризующие полученный при вычитании результат:

1. Результат вычитания равен 0? Процессор заносит эту информацию в бит, названный *битом* Z.
2. Результат вычитания отрицательный? Процессор заносит эту информацию в бит, названный *битом* N.
3. Возникло ли в результате вычитания переполнение числа со знаком? Процессор заносит эту информацию в бит, названный *битом* V (от oVerflow – переполнение).
4. Возникло ли в результате вычитания переполнение числа без знака? Процессор заносит эту информацию в бит, названный *битом* C (от Carry – перенос).

Вместе эти четыре бита называются *кодами условий* и записываются в специальный регистр процессора, который называется *длинным словом состояния процессора*. Вместе биты кодов условий содержат всю информацию, необходимую для того, чтобы определить, должна ли инструкция условного перехода передать управление по требуемому адресу, т.е. осуществить переход.

Ниже показана инструкция CMPL, за которой следует инструкция условного перехода.

```
CMPL    SCORE, #90
BGEQ    A_GRADE
```

По инструкции CMPL процессор должен вычесть число 90 из содержимого длинного слова с адресом SCORE. По результату вычитания процессор заполняет четыре бита кодов условий (уничтожая их предыдущие значения), а затем уничтожает сам результат вычитания. Далее по инструкции BGEQ анализируется значение битов кодов условий для определения того, нужно ли передавать управление по адресу A_GRADE. Значения кодов условий, при которых каждая инструкция условного перехода передает управление по требуемому адресу, описаны в гл. 6.

УСТАНОВКА КОДОВ УСЛОВИЙ

При выполнении условных переходов необходимо, чтобы коды условий были установлены до того, как будет выполнен сам переход. В предыдущих примерах установка кодов условия выполнялась инструкцией из семейства CMP. Но имеются и другие способы установки битов кодов условия.

Один из этих способов состоит в выполнении арифметической операции. При выполнении каждой арифметической инструкции процессор автоматически устанавливает коды условий в зависимости от результата арифметической операции. Например, по инструкции ADDL2 A, B автоматически производится проверка значения результата по адресу B. Результат почти такой же, как при выполнении пары инструкций

```
ADDL2   A, B
CMPL    B, #0
```

Это справедливо для всех арифметических инструкций, т.е. еще для остальных пяти инструкций сложения, для шести инструкций вычитания и для четырех инструкций пересылки (хотя никаких вычислений при этом не производится).

Таким образом, инструкции условного перехода могут использоваться непосредственно за инструкциями сложения, вычитания и пересылки. При выполнении инструкций пересылки значения кодов условий устанавливаются в зависимости от результатов сравнения пересылаемого значения с нулем. Следующие инструкции осуществят переход к оператору с меткой 30$, если значение числа, содержащееся по адресу B, больше или равно 0:

```
MOVL    A, B
BGEQ    30$
```

Приведенная пара операторов пересылает содержимое длинного слова с адресом A в длинное слово с адресом B и осуществляет переход по адресу 30$, если *содержимое длинного слова с адресом* B *больше или равно* 0. Если же содержимое длинного слова с адресом B требуется сравнить с числом, отличным от нуля, то необходимо вставить между приведенными двумя инструкциями инструкцию CMPL.

Инструкции условного перехода анализируют значения битов кодов условий, не изменяя при этом значений этих битов. Это дает возможность использовать несколько инструкций условного перехода после инструкции пересылки, сложения, вычитания или сравнения. Рассмотрим, например, инструкцию SUBL3:

```
.       SUBL3   A, B, C
        BGTR    30$
        BEQL    20$
10$:    . . .

20$:    . . .

30$:    . . .
```


По инструкции SUBL3 произойдет вычитание содержимого длинного слова с адресом A из содержимого длинного слова с адресом B; результат вычитания при этом будет помещен в длинное слово по адресу C. Кроме того, инструкция SUBL3 установит коды условия в зависимости от содержимого длинного слова с адресом C. В результате инструкция BGTR осуществит передачу управления по адресу 30$, если содержимое длинного слова с адресом C окажется больше 0. Если оно равно 0, то процессор оставит невыполненной инструкцию BGTR и следующая инструкция – BEQL осуществит передачу управления по адресу 20$, если меньше 0, то процессор оставит невыполнимыми обе инструкции перехода и будет выполнять инструкцию, начинающуюся с адреса 10$.

Аналогично в следующем фрагменте программы осуществляется передача управления оператором с метками 40$, 50$ или 60$ в зависимости от того, меньше, равно или больше числа 50 содержимое длинного слова с адресом D:

```
        CMPL    D, #50
        BGTR    60$
        BEQL    50$
40$:    . . .

50$:    . . .

60$:    . . .
```

5.4. ФОРМАТ ИНСТРУКЦИЙ ПЕРЕХОДОВ

ОТНОСИТЕЛЬНАЯ АДРЕСАЦИЯ

Формат семейства инструкций перехода отличается от формата остальных инструкций ЭВМ VAX в машинном коде. Во-первых, у них отсутствует байт спецификатора операнда. Все инструкции условного перехода, за редким исключением, имеют длину 16 битов или 2 байта. Первый байт содержит 8-битовый код операции, а второй – 8-битовое *смещение*; указывающее, куда должен осуществляться переход. Так как смещение имеет длину 8 битов, существует только 2^8, или 256, различных значений смещения. Как отмечалось в гл. 3, с помощью 8-битового относительного смещения можно адресовать только 128 байтов с адресами, предшествующими адресу в программном счетчике, и 127 байтов с адресами, следующими за адресом в программном счетчике. Однако это не является серьезным ограничением, поскольку существуют инструкции, позволяющие делать переход по более "удаленным" адресам.

Как уже отмечалось, идея относительной адресации довольно проста. Предположим, например, что вы находитесь на скоростной автостраде, которая имеет 100 съездов, пронумерованных от 00 до 99. Вы остановили свою машину у 32-го съезда, чтобы помочь советом заблудившемуся водителю. Давая совет, вы можете сказать: "Вам нужен 35-й съезд" или "Вам нужен 25-й съезд". При этом вы используете абсолютную адресацию. Однако вы могли бы сказать и так: "Поезжайте вперед, ваш съезд – 3-й отсюда" (32 + 3 = 35) или "Поезжайте назад, ваш съезд – 7-й отсюда (32 – 7 = 25). При этом вы используете относительную адресацию, т.е. советуете водителю проехать вперед или назад определенное количество съездов от места, где он находится в данный момент.

Инструкция перехода действует аналогично. Инструкция перехода указывает процессору, что необходимо передать управление по адресу, смещенному на определенное число байтов вперед или назад относительно адреса, содержащегося в программном счетчике, 8-битовое смещение интерпретируется как 8-битовое число со знаком. Таким образом, с его помощью можно представить десятичные числа от –128 до +127 (см. табл. 2.6). Это означает, что инструкции перехода могут задать переход назад в следующем диапазоне: назад на 128 байтов и вперед на 127 байтов.

Инструкция перехода выполняется просто. Предположим, что следующая инструк-

ция BRB (безусловный переход с относительной адресацией, заданной смещением в формате байта) записана в памяти, начиная с адреса 00000400.

Инструкция	Адрес
50 11	00000400

Процессор считывает код операции инструкции BRB – ^X11 из ячейки с адресом ^X0400, после чего считывает смещение ^X50 из ячейки с адресом ^X0401. К этому моменту содержимое программного счетчика достигло ^X0402. Процессор прибавляет ^X50 к ^X0402 и результат суммирования ^X0452 помещает в программный счетчик. Выборку следующей инструкции процессор будет осуществлять по адресу ^X0452.

Поскольку смещение интерпретируется как число со знаком, значения смещения от ^XFF до ^X80 задают переход назад. Например, по инструкции

Инструкция	Адрес
B0 11	0000400

процессор делает переход назад по адресу ^X000003B2.

Ассемблер автоматически вычисляет значения смещения, поэтому программист должен помнить следующее. Инструкция перехода задает переход на определенное число байтов вперед или назад. Поскольку смещение является 8-битовым числом, значения смещения ограничены диапазоном –128...+127 (десятичные числа). Поскольку содержимое программного счетчика увеличивается на два перед тем, как к нему прибавляется значение смещения, переход назад возможен только на 126, а вперед – на 129 байтов от начального адреса инструкции перехода. В программе на языке ассемблера программисту достаточно написать инструкцию перехода, например BRB, и если адрес перехода находится в пределах 126 байтов до и 129 байтов после инструкции перехода, то ассемблер вычислит смещение правильно. В противном случае ассемблер выдает сообщение об ошибке, указывающее на то, что адрес перехода находится дальше, чем это допускает инструкция перехода.

ИНСТРУКЦИИ BRW И JMP

В ЭВМ VAX предусмотрена инструкция BRW (безусловный переход по адресу, заданному смещением в формате слова) специально для случаев, когда адрес перехода расположен слишком далеко от инструкции перехода. Инструкция BRW аналогична инструкции BRB, за исключением того, что за кодом операции следует 16-битовое смещение вместо 8-битового. Инструкция BRW может осуществлять переход на 32765 байтов назад и на 32770 байтов вперед от адреса инструкции BRW. Например, по инструкции

Инструкция	Адрес
5000 31	00000400

процессор сделает переход по адресу ^X00005403. Кодом операции инструкции BRW является ^X31. К тому моменту, когда процессор считает значение 16-битового смещения – число со знаком ^X5000, содержимое программного счетчика достигнет ^X00000403. Процессор прибавит положительное значение смещения ^X5000 к адресу ^X00000403 и поместит сумму ^X00005403 в регистр PC. В результате следующий код операции процессор будет считывать из ячейки с адресом ^X00005403.

Чтобы понять, как используется инструкция BRW, рассмотрим следующий пример:

```
        CMPL    A,B
        BLSS    ALPHA
```

Если адрес перехода (ALPHA) находится от инструкции BLSS слишком далеко и его невозможно задать с помощью байтового смещения, ассемблер выдаст сообщение об ошибке. Программист может переписать предыдущие операторы так:


```
        CMPL    A,B
        BGEQ    10$
        BRW     ALPHA
10$:    . . .
```

Отметим, что в новом варианте необходимо заменить инструкцию BLSS на обратную инструкцию BGEQ.

Инструкция JMP не имеет таких ограничений, как рассмотренные инструкции. В отличие от других инструкций перехода она имеет формат, согласующийся с форматами остальных инструкций ЭВМ VAX. Инструкция JMP является инструкцией с одним операндом, в котором за кодом операции следует байт спецификатора операнда. В качестве примера рассмотрим следующую инструкцию:

Инструкция	Адрес
00010000 EF 17	00000400

Кодом операции инструкции JMP является ^X17. Спецификатор операнда ^XEF определяет относительную адресацию со смещением, заданным в формате длинного слова. К тому моменту, когда из памяти производится выборка смещения ^X00010000, заданного в формате длинного слова, содержимое программного счетчика увеличивается до ^X00004006. Таким образом, по этой инструкции будет осуществляться переход по адресу ^X00004006 плюс ^X00010000, т.е. по адресу ^X00014006.

При ассемблировании инструкции JMP с относительной адресацией ассемблер по умолчанию будет использовать относительную адресацию со смещением, заданным в формате длинного слова, для переходов вперед, а для переходов назад – относительную адресацию со смещением, заданным в формате байта, слова или длинного слова (по необходимости)[1]. Использование в инструкции JMP относительной адресации со смещением в формате байта или слова не имеет большого смысла, поскольку с помощью инструкций BRB (перейти – смещение байт) или BRW (перейти – смещение слово) может быть достигнут тот же результат, но сами инструкции занимают на один байт меньше. Инструкцию JMP с относительной адресацией и со смещением, заданным в формате длинного слова, целесообразно использовать для передачи управления по адресу, который не может быть достигнут с помощью инструкции BRW (т.е. по адресу, удаленному более чем на 32000 байтов от инструкции перехода). Например, предыдущий фрагмент программы перехода по адресу ALPHA может быть переписан следующим образом:

```
        CMPL    A,B
        BGEQ    10$
        JMP     ALPHA
10$:    . . .
```

Однако в случае соответствующего разбиения программы на модули необходимость перехода к такому удаленному адресу возникает крайне редко.

На самом деле преимущество инструкции JMP заключается в том, что она может использоваться с большим числом режимов адресации, что позволяет легко организовать сложные управляющие конструкции. Более детально режимы адресации описаны в гл. 7. Приведем описание инструкций BRB, BRW и JMP:

[1] Как отмечено в гл. 4, программист может отменять правила, принимаемые по умолчанию. Для этого перед символическим адресом операнда следует поместить префикс B^, W^ или L^, задающий относительную адресацию со смещением, заданным в формате байта, слова и длинного слова соответственно.

Мнемоника	Код операции	Описание
BRB	11	Безусловная передача управления, используя смещение заданное в формате байта
BRW	31	Безусловная передача управления, используя смещение заданное в формате слова
JMP	17	Безусловная передача управления, с использованием спецификатора операнда для задания режима адресации

5.5. ДОПОЛНИТЕЛЬНЫЕ ИНСТРУКЦИИ

СЕМЕЙСТВО ИНСТРУКЦИЙ CLR

При создании программ часто бывает необходимо установить содержимое некоторой ячейки памяти равным 0. Это может быть выполнено с помощью инструкций семейства MOV. Например, инструкция MOVL

```
MOVL    #0, ALPHA
```

установит содержимое длинного слова, начинающегося с адреса ALPHA, равным 0. Тот же результат может быть получен с помощью инструкции SUBL2:

```
SUBL2   ALPHA, ALPHA
```

Для экономии времени, памяти и для того, чтобы программы на языке ассемблера были более читаемыми, в набор инструкций ЭВМ VAX включена группа инструкций CLR (от CLeaR – очистить). Например, инструкция

```
CLRL    ALPHA
```

установит содержимое длинного слова, начинающегося с адреса ALPHA, равным 0. Эта инструкция не только проще в написании, чем инструкция MOVL# 0, ALPHA, но и более эффективна, поскольку требует описания только одного операнда. Семейство инструкций CLR состоит из следующих инструкций:

Мнемоника	Код операции	Операнды	Операция
CLRB	94	dst	dst <- 0
CLRW	B4		
CLRL	D4		
CLRQ	7C		

Эти инструкции могут использоваться для очистки байтов, слов, длинных слов и 64-битовых квадраслов.

Семейство инструкций CLR представляет пример инструкции с одним операндом, так же как MOVL – пример инструкций с двумя, а ADDL3 – с тремя операндами. Трансляция этих инструкций в машинный код имеет фиксированный формат. За кодом операции следует спецификатор единственного операнда, который начинается с байта спецификатора операнда, указывающего, как найти этот операнд. Остальные инструкции, приведенные в этом разделе, также являются инструкциями с одним операндом и имеют тот же формат.

СЕМЕЙСТВО ИНСТРУКЦИЙ INC И DEC

Еще одной распространенной операцией является прибавление 1 к содержимому ячейки памяти. Инструкция


```
ADDL2   #1, ALPHA
```

может быть заменена инструкцией INCL (от INCrement Longword – увеличение содержимого длинного слова – инкремент)

```
INCL    ALPHA
```

Аналогично инструкция DECL (от DECrement Longword – уменьшение содержимого длинного слова – декремент) используется для вычитания единицы из содержимого длинного слова. Например, инструкция

```
DECL    ALPHA
```

эквивалентна инструкции SUBL2 # 1, ALPHA. Однако инструкция DECL короче, быстрее выполняется и делает программы на языке ассемблера более наглядными.

Приведем описание инструкций групп INC и DEC:

Мнемоника	Код операции	Операнды	Операция
INCB	96	sum	sum <- sum + 1
INCW	B6		
INCL	D6		
DECB	97	dif	dif <- dif - 1
DECW	B7		
DECL	D7		

Как следует из описания, эти инструкции применяются для увеличения или уменьшения на 1 содержимого байтов, слов и длинных слов.

СЕМЕЙСТВО ИНСТРУКЦИЙ TST

Предположим, что программист хочет установить содержимое длинного слова с адресом B равным содержимому длинного слова с адресом A плюс единица, а затем сделать переход к оператору с адресом 10$ в случае, если содержимое длинного слова с адресом A равно 0.

Операторы

```
ADDL3   A, #1, B
BEQL    10$
```

не дадут желаемого результата, поскольку биты кодов условия отражают состояние памяти по адресу B, а не A, и, кроме того, по инструкции BEQL произойдет переход, если содержимое длинного слова с адресом B будет равно 0.

Требуемый результат можно получить с помощью следующих операторов:

```
ADDL3   A, #1, B
CMPL    A, #0
BEQL    10$
```

Отметим, что единственным назначением инструкции CMPL является установка битов кодов условий в соответствии с содержимым длинного слова с адресом A. В архитектуру ЭВМ VAX входит семейство инструкций TST специально для этой цели. С использованием инструкции этого семейства предыдущий фрагмент программы может быть переписан следующим образом:

```
        ADDL3   A, #1, B
        TSTL    A
        BEQL    10$
```

Инструкция TSTL (от TeST Longword – проверить содержимое длинного слова) просто выполняет выборку содержимого длинного слова с адресом A и в соответствии с его значением устанавливает биты кодов условий, после чего уничтожает полученное значение. Содержимое длинного слова с адресом A при этом не изменяется.

Ниже приведено описание семейства инструкций TST.

Мнемоника	Код операции	Операнды	Операция
TSTB	95	src	src - 0
TSTW	B5		
TSTL	D5		

Графа "Операнды" указывает на то, что считывается исходный операнд (src), в качестве которого может быть содержимое байта, слова или длинного слова. Запись src – 0 в графе "Операция" означает, что процессор вычитает 0 из исходного операнда и в зависимости от полученного результата устанавливает коды условий. Это вычитание не изменяет его значения, оно позволяет установить коды условий. Кроме того, приведенное описание показывает, как могут быть аппаратно реализованы в процессоре инструкции TST.

5.6. ОРГАНИЗАЦИЯ ЦИКЛОВ

Структуры циклов можно легко организовать с помощью инструкций условного перехода. Например, в следующем фрагменте программы вычисляется сумма целых чисел от 1 до 10:

```
SUM_TEN_TO_ONE:
        CLRL    SUM
        MOVL    #10, J
10$:    ADDL2   J, SUM
        DECL    J
        BNEQ    10$
20$:    . . .
```

Здесь в длинном слове SUM содержится сумма целых чисел, а длинное слово с адресом J используется как счетчик цикла. Показанный цикл организован с уменьшением значения в счетчике цикла. В качестве начального значения в длинное слово J записывается число 10 с помощью инструкции MOL #10, J, расположенной вне цикла. Внутри цикла по инструкции DEC J вычитается единица из содержимого длинного слова J на каждом шаге цикла. На первых десяти шагах цикла содержимое длинного слова J больше 0 и инструкция BNEQ #10$ передает управление на метку 10$. На десятом шаге цикла содержимое длинного слова становится равным 0, процессор пропускает выполнение инструкции BNEQ и переходит к инструкции с меткой 20$. В момент перехода к этому оператору в длинном слове SUM будет содержаться 10+9+8+7+6+5+4+3+2+1, или 55.

Этот фрагмент программы можно упростить, применив инструкцию SOBGTR (вычесть 1 и перейти, если результат больше 0). Рассмотрим действие этой инструкции. Для нее требуется два операнда. Первый задает длинное слово, а второй является адресом перехода. При выполнении инструкции содержимое длинного слова автоматически уменьшается на 1. Если содержимое длинного слова все еще больше 0, происходит переход по адресу, заданному во втором операнде. Как и в других инструкциях перехода, адрес перехода задается как 8-битовое смещение.


```
SUM_TEN_TO_ONE:
        CLRL    SUM
        MOVL    #10, J
10$:    ADDL2   J, SUM
        SOBGTR  J, 10$
        . . .
```

Инструкция SOBGEQ (вычесть 1 и перейти, если результат больше или равен 0) во всем идентична инструкции SOBGTR, за исключением того, что по ней делается переход и тогда, когда первый операнд равен 0.

При программировании на языке ассемблера часто используются циклы с уменьшением значений счетчиков цикла. Выход из таких циклов происходит, когда содержимое счетчика цикла становится равным 0. В языках высокого уровня принято в основном использовать циклы с увеличением значений счетчиков цикла. Например, следующие структуры циклов в программах на Фортране и Паскале задают изменение значения счетчика цикла от 5 до 25; в обеих программах предполагается, что J объявлено как целое:

```
Фортран                 Паскаль

DO 100 J=5,25           FOR J := 5 TO 25 DO
                         BEGIN
  ТЕЛО ЦИКЛА              ТЕЛО ЦИКЛА
100 CONTINUE            END;
```

Аналогичные циклы могут быть организованы на языке ассемблера с помощью инструкции AOBLEQ (прибавить 1 и перейти, если результат меньше или равен чему-либо). Для этой инструкции требуются три операнда. Первым операндом является длинное слово, в котором задается предельное значение параметра цикла (т.е. 25). Второй операнд – это длинное слово, содержащее текущее значение индекса цикла (т.е. J). И наконец, третий операнд является смещением, заданным в формате байта. Ниже показано, как приведенные программы на Фортране и Паскале могут быть реализованы на языке ассемблера.

```
        MOVL    #5, J
10$:
          ТЕЛО ЦИКЛА

        AOBLEQ  #25, J, 10$
```

Инструкция AOBLEQ увеличивает содержимое длинного слова J на 1, после чего сравнивает его с содержимым первого длинного слова – непосредственным операндом #25. Если J меньше или равно 25, то происходит переход к метке 10$. Инструкция AOBLSS во всем идентична инструкции AOBLEQ, за исключением того, что переход осуществляется только тогда, когда параметр цикла меньше заданного предельного значения.

5.7. РЕГИСТРЫ ПРОЦЕССОРА

СИСТЕМА ОБОЗНАЧЕНИЙ В ЯЗЫКЕ АССЕМБЛЕРА

Как отмечалось в гл. 3, в процессоре ЭВМ VAX имеется 16 регистров общего назначения; каждый имеет 32 разряда. Эти регистры обозначаются цифрами от 0 до 15 в десятичной или от ^X0 до ^XF в шестнадцатеричной системах. Несмотря на свое название, регистры 12–15 (^XC–^XF) имеют специальное назначение. Например, регистр 15 является программным счетчиком, с помощью которого процессор отслеживает текущий адрес при выполнении каждой инструкции. Однако регистры 0–11 являются регистра-

ми общего назначения и могут использоваться для работы с данными различного формата.

В гл. 7 будет показано, как регистры процессора используются в различных режимах адресации. В этом разделе будет представлен режим адресации, называемый *регистровым режимом адресации*. При регистровом режиме адресации операнд находится в регистре, а не в памяти. Такое использование регистров делает программы и короче, и эффективнее.

Ниже для удобства использования регистров процессора представлена таблица постоянных имен ассемблера:

Имя	Значение	
R0	Регистровая адресация с использованием регистра	0
R1	То же	1
...	–	
R9	–	9
R10	–	10(^XA)
R11	–	11(^XB)
AP или R12	–	12(^XC)
FP	–	13(^XD)
SP	–	14(^XE)
PC	–	15(^XF)

Чтобы легче было запомнить то, что регистры 12–15 используются для специальных целей, им присвоены следующие имена: AP (указатель аргументов), FP (указатель блока вызова), SP (указатель стека), PC (программный счетчик). В языке ассемблера разрешается обращаться к регистру 12 как по имени (AP), так и по номеру (R12).

После того как указанные символы определены, использование регистров процессора не представляет трудностей. Например, следующая инструкция скопирует длинное слово из регистра 3 в регистр 8.

```
MOVL    R3, R8
```

Обозначение R3 указывает на то, что исходный операнд находится в регистре общего назначения 3, а обозначение R8 – что длинное слово должно быть переслано в регистр 8. Аналогично следующая инструкция прибавит содержимое длинного слова, начинающегося по адресу A, к длинному слову из регистра 2, а полученную сумму поместит в регистр 0.

```
ADDL3   A, R2, R0
```

Хотя каждый из регистров общего назначения содержит 32-битовое длинное слово, можно использовать эти регистры также и для работы с операндами в формате байта или слова.

Для иллюстрации этого предположим, что регистры 3 и 8 содержат длинные слова 11223344 и 55667788 соответственно. Инструкция MOVL R3, R8 перешлет длинное слово из регистра 3 в регистр 8, в результате чего содержимое регистра 8 станет равным ^X11223344. Однако, если вместо инструкции MOVL использовать инструкцию MOVB, процессор перешлет только содержимое младшего байта из регистра 3 в регистр 8. При этом 24 старших разряда регистра 8 своего значения не изменят. Таким образом инструкция MOVB R3, R8 сделает содержимое регистра 8 равным ^X55667744. Аналогично инструкции, работающие со словами, имеют дело только с двумя младшими байтами (16 разрядов) в регистре. В этом случае два старших байта вообще не участвуют в выполнении инструкции. Так, инструкция MOVW R3, R8 поместит в регистр 8 значение ^X55663344.

НЕКОТОРЫЕ ПРИМЕРЫ

Допускается использовать в одной инструкции различные спецификаторы операндов. Например, ниже приведены инструкции, каждая из которых является допустимой.

```
MOVB    A, R1
ADDW3   R1, #50, R10
SUBL2   R7, C
CMPL    R4, #20
SUBB3   #50, A, R11
INCW    R4
```

Однако, например, инструкция BLEQ R5 является запрещенной. Поскольку в инструкциях перехода отсутствует спецификатор операнда, в них не может использоваться регистровая адресация. Инструкция JMP R5 также является запрещенной, хотя и имеет байт спецификатора операнда. Дело в том, что вне памяти возможно только выполнение инструкций и, следовательно, переходы к регистру запрещены.

Для демонстрации того, как использование регистров процессора может ускорить выполнение программы, рассмотрим следующий фрагмент программы, который устанавливает содержимое длинного слова A равным B + C – D – 5:

```
ADDL3   B, C, A
SUBL2   D, A
SUBL2   #5, A
```

Подсчитаем, сколько раз происходит выборка или занесение длинного слова в ячейку памяти с адресом A. Инструкция ADDL3 помещает результат в ячейку с адресом A. Обе инструкции SUBL2 осуществляют выборку и запись в ячейку с адресом A. Далее в последовательности инструкций используется регистр R0, что позволяет исключить все три рассмотренных обращения к ячейке с адресом A, кроме одной записи в эту ячейку:

```
ADDL3   B, C, R0
SUBL2   D, R0
SUBL3   #5, R0, A
```

Новый фрагмент программы на языке ассемблера кажется более длинным, поскольку в третьей инструкции добавился третий операнд. Однако, как будет показано далее, для регистровой адресации требуется меньше памяти, чем для относительной адресации со смещением, заданным в формате байта. Поэтому новый фрагмент программы на самом деле занимает меньше памяти, чем исходный.

Использование регистров существенно при организации циклов. Значения, к которым происходит частое обращение, должны храниться в регистрах. Ниже воспроизведен один из фрагментов программы на языке ассемблера, осуществляющий суммирование целых чисел от одного до десяти.

```
SUM_TEN_TO_ONE:
        CLRL    SUM
        MOVL    #10, J
10$:    ADDL2   J, SUM
        SOBGTR  J, 10$
        . . .
```

При использовании R0 вместо J и R1 вместо SUM этот фрагмент принимает следующий вид:

```
SUM_TEN_TO_ONE:
        CLRL    R1
        MOVL    #10,R0
10$:    ADDL2   R0,R1
        SOBGTR  R0,10$
        MOVL    R1,SUM
        . . .
```

Несмотря на то, что пришлось добавить инструкцию MOVL R1, SUM в конце цикла, оказалось, что новый вариант программы короче и использует меньший объем памяти, чем исходный.

РЕГИСТРОВАЯ АДРЕСАЦИЯ

В предыдущих примерах показано, что регистровая адресация может использоваться в языке ассемблера. Когда регистровые операнды транслируются в машинный код, номер регистра (от ^X0 до ^XF) становится частью байта спецификатора операнда. Например, ассемблер для инструкции MOVL R8, R3 сгенерирует следующий машинный код (предполагается, что инструкция размещается в памяти по адресу ^X0400):

Операнд	Операнд	Код операции	Адрес
58	53	D0	04000

Отметим, что код этой инструкции на машинном языке занимает только 3 байта: ^XD0 – код операции инструкции MOVL; спецификатор первого операнда ^X53 указывает на то, что исходным операндом является 32-битовое длинное слово, находящееся в регистре 3; спецификатор второго операнда ^X58 указывает на то, что содержимое длинного слова будет помещено в регистр 8, при этом предыдущее содержимое регистра 8 уничтожается.

На примере регистровой адресации можно показать, каким образом формируются коды спецификаторов операндов для других режимов адресации. Спецификатор операнда описывает способ нахождения операнда. Содержимое спецификатора операнда обозначается двумя шестнадцатеричными цифрами. Правая шестнадцатеричная цифра (от 0 до F) определяет один из шестнадцати регистров общего назначения процессора; левая шестнадцатеричная цифра обозначает *режим* адресации и указывает, каким образом используется регистр общего назначения при нахождении операнда:

Режим	Регистр	
mmmm	rrrr	8 битов байта спецификатора операнда

Единственным исключением из этого правила является режим литеральной адресации, при котором в байте спецификатора операнда хранится 6-битовая константа.

Регистровая адресация задается указанием кода ^B0101 или ^X5 в битах режима. Любой спецификатор операнда от ^X50 до ^X5E задает регистровую адресацию. В битах регистра указывается номер регистра, в котором хранится операнд. Например, операнды языка ассемблера R0 и R11 в машинном коде будут соответствовать спецификаторам операндов ^X50 и ^X5B.

Поскольку код режима состоит из четырех битов, в архитектуре ЭВМ VAX всего могло быть 16 режимов адресации (от ^X0 до ^XF). Однако, как описано в гл. 7, ЭВМ VAX режимы ^X0 – ^X3 используют для литеральной адресации. Остальные 12 режимов (^X4–^XF) наряду с режимами литеральной адресации называются *режимами адресации с использованием регистров* общего назначения. Ниже описаны два таких режима адресации.

Спецификатор операнда	Название режима	Значение
00 – 3F	Литеральный	Байт спецификатора операнда содержит 6-битовую константу. Ассемблер выбирает этот режим адресации, если обозначение операнда начинается со знака номера (#), за которым следует небольшая по значению константа или имя, ее обозначающее (#1, #20 и др.)
50 – 5E	Регистровый	Операнд находится в регистре общего назначения процессора, номеру которого соответствует вторая шестнадцатеричная цифра в значении спецификатора операнда (от ^X0 до ^XE). Ассемблер выбирает этот режим адресации всегда, когда в поле операнда указано имя регистра (R3, R10, SP и т. д.)

Отметим, что для регистровой адресации допустимы значения спецификаторов операндов в диапазоне ^X50 – ^X5E, а не ^X50 – ^X5F. Дело в том, что при регистровой адресации никогда не должен использоваться программный счетчик (^X5F) в качестве регистра общего назначения, так как его содержимое увеличивается в процессе выполнения инструкции. Программный счетчик может применяться только в режимах ^X8 – ^XF. Поскольку программный счетчик PC является специальным регистром, режимы ^X8 – ^XF, в которых применяется регистр PC, получили название *режимов адресации с использованием программного счетчика.* Пять режимов адресации с использованием программного счетчика описаны ниже.

Спецификатор операнда	Название режима	Значение
8F	Непосредственный	Операнд содержится в байтах, следующих за байтом спецификатора операнда. Ассемблер выбирает этот режим, когда обозначение операнда начинается со знака номера (#), за которым следует число (или имя, обозначающее число), слишком большое, чтобы его можно было задать как литерал (#100, #123456 и т. д.).
9F	Абсолютный	Адрес операнда содержится в четырех байтах, следующих за байтом спецификатора операнда. (Ассемблер выбирает абсолютную адресацию, когда обозначение операнда начинается символами @#, например @#EXTNAME.)
AF	Относительный со смещением, заданным в формате байта	За байтом спецификатора операнда следует смещение, заданное в формате байта
CF	Относительный со смещением, заданным в формате слова	За байтом спецификатора операнда следует смещение, заданное в формате слова

EF	Относительный со смещением, заданным в формате длинного слова	За байтом спецификатора операнда следует смещение, заданное в формате длинного слова

Ассемблер выбирает один из режимов относительной адресации тогда, когда для обозначения операнда используется символическое имя или число (WAGE, 20$ и т.д.). В каждом режиме адресации с использованием программного счетчика, информация, необходимая для нахождения операнда, следует непосредственно за байтом спецификатора операнда. Программный счетчик используется в этих режимах потому, что после того, как произведена выборка байта спецификатора операнда, программный счетчик указывает на байт, который следует непосредственно за байтом спецификатора операнда. Эти режимы адресации описаны в гл. 7.

5.8. ПОДПРОГРАММЫ

ВЫЗОВ И ВОЗВРАТ

Вопрос использования подпрограмм подробно рассмотрен в гл. 9. Но есть две причины, по которым начать знакомство с подпрограммами следует именно теперь. Первая: подпрограммы играют большую роль в организации правильной структуры программ, и материал данного раздела позволит читателю написать некоторые простые подпрограммы. Вторая: в приложении В представлены некоторые подпрограммы ввода-вывода, которые могут быть включены в программы. Использование этих подпрограмм позволит читателю приступить к созданию более сложных программ.

Существует три проблемы, связанные с подпрограммами. Во-первых, необходимо иметь средства передачи управления подпрограмме. Во-вторых, должны быть средства обратной передачи управления вызывающей программе. Для вызова подпрограммы с именем SUB в ЭВМ VAX используется инструкция JSB SUB (Jump to SuBroutine – перейти к подпрограмме). Для возврата может быть использована инструкция RSB (Return from SuBroutine – возврат из подпрограммы). И третья проблема – это передача информации между основной программой и подпрограммой; подробно она рассматривается в гл. 9. Сейчас мы ограничимся рассмотрением простого способа передачи информации подпрограммам и обратно с помощью регистров общего назначения процессора. Например, основная программа может переслать данные в регистры R0, R1, . . . перед тем, как вызвать подпрограмму, а подпрограмма может также поместить результаты своей работы в регистры общего назначения, после чего вернуть управление вызывающей программе. Этот способ подходит для простых задач, но, как будет показано в гл. 9, является слишком примитивным для более сложных задач.

ВВОД И ВЫВОД

Приложение В содержит некоторые подпрограммы ввода-вывода, которые называются IOINIT, RNUM и PNUM. Подпрограмма IOINIT выполняет инициализацию, которая необходима для осуществления ввода или вывода. Действие подпрограммы IOINIT аналогично действию оператора OPEN в некоторых языках высокого уровня.

Для того чтобы осуществить инициализацию, требуется инструкция:

```
JSB     IOINIT
```

Отметим, что эта инструкция должна выполняться только один раз в процессе работы программы.

Подпрограмма RNUM осуществляет ввод шестнадцатеричного числа с терминала пользователя и помещает его 32-битовое значение в регистр R0. Подпрограмма PNUM выводит на терминал (на печать) содержимое регистра R0 в шестнадцатеричном и десятичном виде. (При использовании режима пакетной обработки подпрограмма RNUM считывает длинное слово из входного потока данных, а подпрограмма PNUM помещает содержимое регистра R0 в файл регистрации пакетного задания.) Таким образом, чтобы ввести число и записать его значение в ячейку с адресом X, необходимо выполнить следующие инструкции:

```
JSB     RNUM
MOVL    R0, X
```

Для того чтобы распечатать содержимое ячейки с адресом X, необходимо выполнить инструкции

```
MOVL    X, R0
JSB     PNUM
```

Используйте подпрограммы IOINIT, RNUM и PNUM из приложения В как составную часть ваших программ.

Вызов подпрограммы можно сравнить с временным дорожным объездом. Например, основная программа на рис. 5.5 должна ввести два числа и поместить их сумму в память, в длинное слово с адресом SUM. (Не обращайте внимание на инструкции типа PUSHR # #M‹R0, R1› в подпрограммах. Многие из них будут описаны в последующих главах.) При выполнении первой инструкции JSB RNUM процессор временно останавливает выполнение основной программы и передает управление подпрограмме RNUM (см. стрелку, помеченную буквой A). Когда процессор встречает инструкцию RSB в конце подпрограммы RNUM, управление автоматически передается инструкции, следующей за инструкцией JSB, в данном случае – MOVL R0, A (см. стрелку B). При выполнении процессором второй инструкции, JSB RNUM, управление снова передается подпрограмме RNUM (см. стрелку C). В конце концов инструкция RSB в конце подпрограммы RNUM передает управление назад – инструкции MOVL R0, B, т.е. основной программе (см. стрелку D).

Инструкция JSB, так же как инструкция JMP, осуществляет передачу управления. (Формат инструкции JSB такой же, как у инструкции JMP, за исключением кода операции – ^X16 вместо ^X17). Еще одно отличие состоит в том, что инструкция JSB также обеспечивает механизм возврата через какое-то время к оператору, непосредственно следующему за инструкцией JSB. Для возврата управления инструкция RSB использует информацию, которая была сохранена при выполнении инструкции JSB (RSB – однобайтовая инструкция с кодом операции ^X05).

Использование подпрограмм является мощным средством для разбиения больших программ на более удобные для работы части. Однако их неправильное применение может привести к труднообнаруживаемым ошибкам. Предположим, например, что для передачи управления подпрограмме RNUM программист использовал оператор JMP RNUM вместо JSB RNUM. Подпрограмма будет выполняться правильно до тех пор, пока не дойдет очередь до инструкции RSB в конце подпрограммы. Использование инструкции RSB предполагает, что предшествующая инструкция JSB сохранила адрес возврата. А поскольку адрес возврата не был сохранен, инструкция RSB осуществит возврат управления неизвестно куда, что приведет к непредсказуемым результатам. Такой же результат может возникнуть, если программист забудет написать инструкцию $EXIT_S в конце основной программы (тогда управление "провалится" из основной программы в подпрограмму).

```
        .TITLE  SHOW_SUBS
;В ПРОГРАММЕ ВЫПОЛНЯЕТСЯ ВВОД ЧИСЕЛ A И B
;И ПЕЧАТАЕТСЯ ИХ СУММА  SUM := A + B
;БЛОК ДАННЫХ
A:      .BLKL   1
B:      .BLKL   1
C:      .BLKL   1
;ОСНОВНАЯ ПРОГРАММА
        .ENTRY  START, 0        ;START ЯВЛЯЕТСЯ ТОЧКОЙ ВХОДА
        JSB     IOINIT          ;ВЫЗОВ IOINIT ДЛЯ ПОДГОТОВКИ ВВОДА-ВЫВОДА
        JSB     RNUM            ;СЧИТАТЬ ЧИСЛО В R0
        MOVL    R0, A           ;И ЗАПОМНИТЬ В  A
        JSB     RNUM            ;СЧИТАТЬ  ВТОРОЕ ЧИСЛО В R0
        MOVL    R0, B           ;И ЗАПОМНИТЬ В  B
        ADDL3   A, B, SUM       ;SUM := A + B
        MOVL    SUM, R0         ;ПЕРЕСЛАТЬ SUM В R0
        JSB     PNUM            ;И НАПЕЧАТАТЬ
        $EXIT_S                 ;ЗАВЕРШИТЬ ПРОГРАММУ
;
; КОНСТАНТЫ ДЛЯ ПРОГРАММ RNUM, PNUM И IOINIT
        . . .
;
; ОБЛАСТЬ ПЕРЕМЕННЫХ ДЛЯ ПРОГРАММ RNUM, PNUM И IOINIT
;
; ПРОГРАММА ИНИЦИАЛИЗАЦИИ
IOINIT: PUSHR   #^M<R0, R1>
        . . .
        RSB
;
; ПРОГРАММА ПЕЧАТИ ЧИСЛА
PNUM:   PUSHR   #^M<R0, R1>
        . . .
        RSB
;
; ПРОГРАММА ВВОДА ЧИСЛА
RNUM:   PUSHR   #^M<R1>
        . . .
        RSB
;КОНЕЦ ПРОГРАММЫ,  START - АДРЕС ТОЧКИ ВХОДА
        .END    START
```

Рис. 5.5. Основная программа с подпрограммами

ДОПОЛНИТЕЛЬНЫЕ ИНСТРУКЦИИ ВЫЗОВА ПОДПРОГРАММ

Использование подпрограмм – очень важный вопрос. Набор инструкций ЭВМ VAX содержит пять инструкций машинного языка для передачи управления подпрограмме: JSB, BSBB, BSBW, CALLG и CALLS. Инструкции BSBB (перейти к подпрограмме по адресу, заданному смещением в формате байта) и BSBW (перейти к подпрограмме по адресу, заданному смещением в формате слова) подобны инструкции JSB, за исключением того, что для них используется формат семейства инструкций переходов. Инструкция BSBB состоит из однобайтового кода операции, за которым следует байт, содержащий 8-битовый относительный адрес. Инструкция BSBW состоит из однобайтового кода операции, за которым следуют два байта, содержащие 16-битовый относительный адрес. Если управление подпрограмме передается с помощью инструкций JSB, BSBB или BSBW, подпрограмма должна заканчиваться инструкцией RSB. Когда подпрограмма вызывается инструкцией CALLG или CALLS, она должна заканчиваться инструкцией RET.

Подпрограммы могут быть *вложенными*. Например, основная программа может вызывать подпрограмму с именем SUBA, которая, в свою очередь, может вызывать подпрограмму с именем SUBB. Подпрограмма SUBB осуществит возврат управления в подпрограмму SUBA, которая, в свою очередь, передаст его основной программе.

Для упрощения вызова вложенных подпрограмм пять инструкций, осуществляющих передачу управления подпрограммам, используют регистр общего назначения с номером 14. Поскольку регистр 14 служит для указания области памяти, называемой стеком, он называется *указателем стека* (SP – Stack Pointer). Название "стек" объясняется тем, что области памяти подобна стопке (stack) тарелок или книг. Еще требуется добавить одну тарелку в стопку, ее обычно кладут сверху стопки. При необходимости взять одну тарелку из стопки обычно берется верхняя. В результате, как правило, из стопки берется тарелка, положенная последней. Инструкции вызова подпрограмм используют указатель стека (SP) для помещения адреса возврата на вершину стека. Инструкции возврата извлекают адрес возврата с вершины стека. Таким образом, стек является способом организации памяти, обеспечивающим вызов вложенных подпрограмм. Если программа MAIN вызывает подпрограмму SUBA, а SUBA вызывает подпрограмму SUBB, то на вершине стека будет находиться адрес возврата к SUBA. В результате инструкция возврата в конце подпрограммы SUBB правильно осуществит передачу управления в подпрограмму SUBA, а инструкция возврата в конце SUBA осуществит возврат управления в MAIN. В гл. 9 будут показаны другие важные области использования стека.

Инструкции CALLS, CALLG и RET обеспечивают стандартную процедуру вызова подпрограмм, написанных на различных языках. Помимо сохранения адресов возврата они поддерживают работу более сложной системы сохранения информации. Кроме указателя стека SP эти инструкции используют регистры 12 и 13. Общее описание работы этих инструкций приведено в гл. 9. Ниже даны имена регистров специального назначения.

Регистр	Имя	Название	Назначение
12	AP	Указатель аргументов	Передача аргументов
13	FP	Указатель блока вызова	Сохранение содержимого указателя стека
14	SP	Указатель стека	Сохранение адреса возврата и других данных
15	PC	Программный счетчик	Указывает следующую выполняемую инструкцию

ПРИМЕР ПОДПРОГРАММЫ

Чтобы понять на примере, как использовать подпрограммы IOINIT, RNUM и PNUM и как самому писать подпрограмму, рассмотрим следующую задачу. Она состоит в том, чтобы ввести три числа и распечатать наибольшее из них. Хотя наш способ решения этой задачи может показаться надуманным и слишком сложным, он реализует общий подход, который существенно упрощает решение более сложных задач.

Как известно, у нас уже есть подпрограммы для инициализации процесса ввода-вывода, чтения и печати числа. Кроме того, мы должны будем написать подпрограмму нахождения максимума из двух чисел, которая будет называться MAX. При вызове подпрограмма MAX должна будет сравнить содержимое регистров R0 и R1 и поместить наибольшее из них в регистр R0 (рис. 5.6).

Теперь основная программа может вызвать подпрограммы IOINIT RNUM, MAX и PNUM для ввода трех чисел и для печати наибольшего из них. На рис. 5.7 показано, как может выглядеть основная программа.

И наконец, на рис. 5.8 показано, как объединить основную программу с подпрограммами так, чтобы получилась единая программа. Отметим, что области описания констант и переменных, используемых в подпрограммах IQINIT, RNUM и PNUM, расположены перед соответствующими подпрограммами. Это позволяет ассемблеру использовать отно-

```
;
;ПОДПРОГРАММА MAX ПОМЕЩАЕТ В РЕГИСТР R0 НАИБОЛЬШЕЕ ИЗ ЧИСЕЛ,
;СОДЕРЖАЩИХСЯ В РЕГИСТРАХ R0 И R1
;
MAX:    CMPL    R0,R1           ;ЕСЛИ ЧИСЛО В R0 БОЛЬШЕ ЧИСЛА В R1
        BGEQ    10$             ;ДА, ЧИСЛО В R0 ЯВЛЯЕТСЯ НАИБОЛЬШИМ
        MOVL    R1,R0           ;НЕТ, НАИБОЛЬШИМ ЯВЛЯЕТСЯ ЧИСЛО В R1
10$:    RSB                     ;ВОЗВРАТ К ОСНОВНОЙ ПРОГРАММЕ
```

Рис. 5.6. Подпрограмма нахождения большего из двух чисел

```
;
;В ПРОГРАММЕ ВЫПОЛНЯЕТСЯ ВВОД ТРЕХ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ
;В ФОРМАТЕ ДЛИННОГО СЛОВА И ПЕЧАТЬ НАИБОЛЬШЕГО ИЗ ЭТИХ ЧИСЕЛ
;
        .ENTRY  START,0
        JSB     IOINIT          ;ИНИЦИАЛИЗАЦИЯ RNUM И PNUM
        JSB     RNUM            ;ВВЕСТИ ЧИСЛО  A
        MOVL    R0,R2           ;ПОМЕСТИТЬ ЧИСЛО A  В R2
        JSB     RNUM            ;ВВЕСТИ ЧИСЛО  B
        MOVL    R0,R1           ;ПОМЕСТИТЬ ЧИСЛО B  В R1
        JSB     RNUM            ;ВВЕСТИ ЧИСЛО  C
        JSB     MAX             ;R0=MAX(B,C)
        MOVL    R2,R1           ;ПОМЕСТИТЬ ЧИСЛО A  В R1
        JSB     MAX             ;R0=MAX(A,R0)
        JSB     PNUM            ;НАПЕЧАТАТЬ ЧИСЛО ИЗ R1
        $EXIT_S                 ;ЗАВЕРШИТЬ ВЫПОЛНЕНИЕ
```

Рис. 5.7. Основная программа нахождения большего из трех чисел

```
        .TITLE  НАИБОЛЬШЕЕ ИЗ ТРЕХ ЧИСЕЛ
;
;В ПРОГРАММЕ ВЫПОЛНЯЕТСЯ ВВОД ТРЕХ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ
;В ФОРМАТЕ ДЛИННОГО СЛОВА И ПЕЧАТЬ НАИБОЛЬШЕГО ИЗ ЭТИХ ЧИСЕЛ
;
        .ENTRY  START,0
        JSB     IOINIT          ;ИНИЦИАЛИЗАЦИЯ RNUM И PNUM
        JSB     RNUM            ;ВВЕСТИ ЧИСЛО  A
        MOVL    R0,R2           ;ПОМЕСТИТЬ ЧИСЛО A  В R2
        JSB     RNUM            ;ВВЕСТИ ЧИСЛО  B
        MOVL    R0,R1           ;ПОМЕСТИТЬ ЧИСЛО B  В R1
        JSB     RNUM            ;ВВЕСТИ ЧИСЛО  C
        JSB     MAX             ;R0=MAX(B,C)
        MOVL    R2,R1           ;ПОМЕСТИТЬ ЧИСЛО A  В R1
        JSB     MAX             ;R0=MAX(A,R0)
        JSB     PNUM            ;НАПЕЧАТАТЬ ЧИСЛО ИЗ R1
        $EXIT_S                 ;ЗАВЕРШИТЬ ВЫПОЛНЕНИЕ
;
;ПОДПРОГРАММА MAX ПОМЕЩАЕТ В РЕГИСТР R0 НАИБОЛЬШЕЕ ИЗ ЧИСЕЛ,
;СОДЕРЖАЩИХСЯ В РЕГИСТРАХ R0 И R1
;
MAX:    CMPL    R0,R1           ;ЕСЛИ ЧИСЛО В R0 БОЛЬШЕ ЧИСЛА В R1
        BGEQ    10$             ;ДА, ЧИСЛО В R0 ЯВЛЯЕТСЯ НАИБОЛЬШИМ
        MOVL    R1,R0           ;НЕТ, НАИБОЛЬШИМ ЯВЛЯЕТСЯ ЧИСЛО В R1
        RSB                     ;ВОЗВРАТ К ОСНОВНОЙ ПРОГРАММЕ
;
; КОНСТАНТЫ ДЛЯ ПРОГРАММ RNUM, PNUM И IOINIT
        . . .
```



```
;
; ОБЛАСТЬ ПЕРЕМЕННЫХ ДЛЯ ПРОГРАММ RNUM, PNUM И IOINIT
;
; ПРОГРАММА ИНИЦИАЛИЗАЦИИ
IOINIT: PUSHR   #^M<R0,R1>
        . . .
        RSB
;
; ПРОГРАММА ПЕЧАТИ ЧИСЛА
PNUM:   PUSHR   #^M<R0,R1>
        . . .
        RSB
;
; ПРОГРАММА ВВОДА ЧИСЛА
RNUM:   PUSHR   #^M<R1>
        . . .
        RSB
;КОНЕЦ ПРОГРАММЫ,  START - АДРЕС ТОЧКИ ВХОДА
        .END    START
```

Рис. 5.8. Завершенная программа нахождения большего из трех чисел

сительную адресацию со смещением, заданным не длинным словом, а байтом или словом. Отметим, что в программе имеется только одна директива ассемблера .END, которая помещена после последней подпрограммы. Итак, мы объединили основную программу и подпрограммы в единую программу на языке ассемблера. В гл. 9 будет показано, как могут объединяться программы на языке ассемблера, ассемблированные раздельно, и даже то, как они могут включаться в раздельно скомпилированные программы, написанные на языках высокого уровня, таких как Фортран и Паскаль.

УПРАЖНЕНИЯ 5.1

1. Напишите программу на языке ассемблера, которая осуществляет ввод 20 чисел и выводит на печать их сумму. При выполнении этого упражнения используйте подпрограммы ввода и вывода из приложения В.

2. а) напишите программу на языке ассемблера, которая вводит три числа, а затем выводит на печать 0, если все три числа разные; 1, если совпадают два из трех чисел; 2, если все числа одинаковы;

б) перепишите программу так, чтобы она повторилась в цикле 20 раз, т.е. были бы напечатаны результаты вычислений для 20 наборов из трех чисел;

в) перепишите программу так, чтобы сначала вводилось число – количество наборов из трех чисел, а затем выполнилось в цикле 1 раз.

3. Напишите программу на языке ассемблера, которая вводит три числа, а затем распечатывает их в порядке убывания.

4. Напишите программу для выполнения п. 3, используя подпрограмму, которая помещает числа в регистры R0 и R1 и при необходимости переставляет их местами так, чтобы перед выходом из нее содержимое регистра R0 было бы меньше или равно содержимому регистра R1.

5. Напишите программу на языке ассемблера, в которой осуществляется ввод 20 чисел и распечатывается наибольшее из них. Используйте показанную на рис. 5.6 подпрограмму MAX. Перепишите программу таким образом, чтобы сначала вводилось число N – количество вводимых чисел, а затем было найдено наибольшее из этих чисел.

6. Используя метод разностей, описанный в гл. 4, напишите программу, которая в цикле распечатывает квадраты целых чисел от 1 до 50.

ГЛАВА 6. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НА ЭВМ VAX

6.1. ВВЕДЕНИЕ

В предыдущих главах были рассмотрены сложение и вычитание чисел со знаком и без знака, а также проверка и сравнение чисел со знаком. В этой главе более детально будут рассмотрены свойства чисел со знаком и без знака, а также вопросы, связанные с переполнением, умножением, делением и многократно увеличенной точностью. Будет описано много новых инструкций и показано, как с их помощью можно упростить программирование. Кроме того, будет продолжено изучение инструкций проверки, сравнения, перехода и их особенностей.

6.2. ЧИСЛА СО ЗНАКОМ И БЕЗ ЗНАКА

РАБОТА С ЧИСЛАМИ И ИХ ИНТЕРПРЕТАЦИЯ

Одно из преимуществ представления чисел в дополнительном коде заключается в том, что для чисел со знаком и без знака используются одинаковые алгоритмы сложения и вычитания. (Другие представления чисел со знаком, применяемые в различных ЭВМ, не обладают таким свойством.) Хотя этот факт кажется удивительным – объяснить его довольно просто. Для этой цели удобнее рассматривать 4-битовые числа, а не 8-, 16- или 32-битовые, поскольку существенно уменьшается число возможных комбинаций.

С помощью четырех битов можно составить 2^4, или 16, комбинаций. Как показано на рис. 6.1, можно составить схему, на которой 16 комбинаций располагаются по окружности. Это напоминает циферблат часов, на котором вместо 12 позиций – 16, а в позиции, соответствующей 12 час., помещено двоичное число 0000. Для указания одного из 16 двоичных чисел используется указатель. Прибавление к числу 1 рассматривается как перемещение указателя на одну позицию вперед. Вычитание 1 рассматривается как перемещение указателя на одну позицию назад.

Если прибавлять единицу 16 раз подряд, указатель совершит полный оборот и вернется в начальное положение. Математики это называют системой счисления *по модулю 16**. Но можно было бы назвать это ошибочным, так как очевидно, что X плюс 16 не равно X. Для того чтобы сделать эту систему счисления непротиворечивой, необходимо условиться, что где-то на циферблате должна быть точка возникновения ошибки.

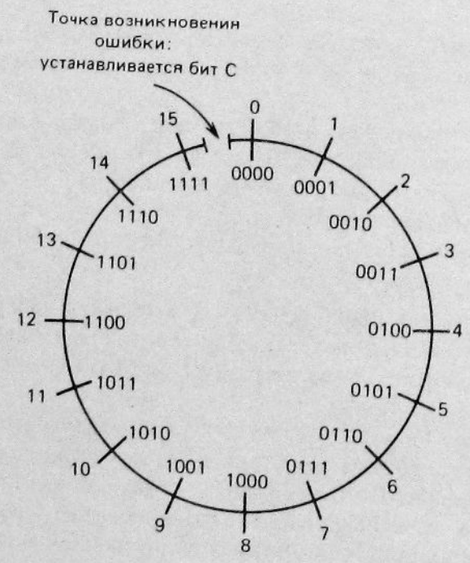


Рис. 6.1. Арифметика чисел без знака

Например, эту точку можно расположить между числами 1111 и 0000. Каждый раз, когда к числу 1111 прибавляется или из числа 0000 вычитается 1, возникает ошибка, которая называется *переполнением числа без знака.* После того, как точка возникновения ошибки определена, каждой двоичной комбинации может быть поставлена в соответствие ее десятичная интерпретация. Если двоичная комбинация 0000 является представлением десятичного нуля, то тогда комбинация 1111 должна быть представлением десятичного числа 15. В итоге будет получена система счисления по модулю 16 для чисел без знака.

Однако точку возникновения ошибки можно расположить на циферблате в каком-нибудь другом месте. В частности, ее можно поместить между числами 0111 и 1000, как показано на рис. 6.2. Когда к числу 0111 прибавляется или из числа 1000 вычитается 1, возникает ошибка, называемая *переполнением* числа со знаком. Если двоичным комбинациям дать теперь десятичную интерпретацию, то окажется, что они представляют десятичные числа от –8 до +7. (Отметим, что если двоичная комбинация 0000 представляет десятичный нуль, то в соответствии с определением вычитания мы вынуждены интерпретировать комбинацию 1111 как десятичное число –1). В результате получим уже знакомое нам представление чисел в дополнительном коде.

Именно так интерпретируются двоичные числа в ЭВМ семейства VAX, только они включают не 4 бита, а 8, 16 или 32. Фактическое число битов определяется типом используемой инструкции, т.е. чем работает она с байтами, словами или длинными словами. Точка возникновения ошибки переполнения для чисел без знака, представленных в формате длинного слова, расположена между значениями ^XFFFFFFFF и ^X00000000, а между значениями ^X7FFFFFFF и ^X80000000 расположена точка возникновения ошибки переполнения для чисел со знаком, представленных в формате длинного слова.

Причины, по которым программист выбирает ту или иную систему представления чисел, зависит от требований конкретной задачи. Как будет показано в гл. 7, адреса иногда

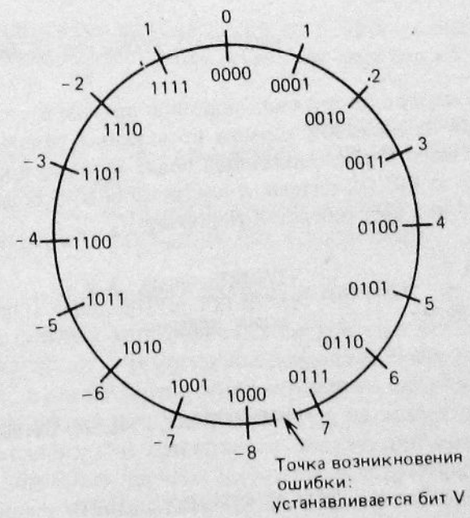


Рис. 6.2. Арифметика чисел со знаком

*Аналогия с часами в данном случае относится только к сложению и вычитанию положительных чисел.

используются как данные для вычисления местоположения элементов массива. Поскольку считается, что адрес ^X80000000 больше адреса ^X7FFFFFFF, эти числа необходимо рассматривать как числа без знака. С другой стороны, если в результате вычислений могут получиться отрицательные числа, необходимо использовать представление чисел со знаком.

ОБРАБОТКА ПЕРЕПОЛНЕНИЯ

Для того чтобы программист мог обнаружить два указанных типа переполнения, в процессоре имеется два флага условий. Они называются бит С и бит V. Бит С устанавливается равным 1, когда в процессе выполнения арифметической операции происходит перенос из старшего разряда результата. Это аналогично переполнению числа без знака, показанному на рис. 6.1. В случае, если переноса не происходит, бит-С сбрасывается в нуль. Бит-V устанавливается в 1 в случае переполнения числа со знаком, как показано на рис. 6.2, и сбрасывается в нуль, если такого переполнения не происходит.

Для того чтобы биты С и V можно было использовать, необходимы средства для проверки их состояния. Ниже приведены инструкции, необходимые для этого:

Мнемоника	Значение
BCS	Перейти, если бит С установлен (С = 1)
BCC	Перейти, если бит С сброшен (С = 0)
BVC	Перейти, если бит V установлен (V = 1)
BVC	Перейти, если бит V сброшен (V = 0)

В качестве примера использования этих битов приведем фрагмент программы, в котором производится сложение двух чисел со знаком, представленных в формате длинного слова, после чего происходит переход к метке ERROR, если возникает переполнение результата, а именно если результат либо меньше – 2147483648, либо больше +2147483647:

```
        ADDL2   A, B            ;СЛОЖИТЬ ЧИСЛА  A И B
        BVS     ERROR           ;ПРИ ПЕРЕПОЛНЕНИИ ПЕРЕЙТИ НА ERROR
        .
        .
ERROR:  .
                                ;НАПЕЧАТАТЬ СООБЩЕНИЕ ОБ ОШИБКЕ
```

Отметим, что эти четыре инструкции подобны другим инструкциям перехода и имеют такие же ограничения на диапазон адресов возможных переходов: 128 байтов, расположенных перед, и 127 байтов, расположенных после адреса в программном счетчике. В случае, если адрес перехода ERROR удален от инструкции BVS более чем на 128 байтов, могут потребоваться инструкции JMP или BRW. Например,

```
        ADDL2   A, B            ;СЛОЖИТЬ ЧИСЛА  A И B
        BVC     10$             ;ЕСЛИ НЕТ ПЕРЕПОЛНЕНИЯ - ПРОДОЛЖИТЬ,
        JMP     ERROR           ;ИНАЧЕ ПЕРЕЙТИ НА ERROR
10$:    .
        .
        .
ERROR:  .
                                ;НАПЕЧАТАТЬ СООБЩЕНИЕ ОБ ОШИБКЕ
```

ДРУГИЕ ФЛАГИ УСЛОВИЙ

Уже было рассмотрено два флага условий: бит С и бит V. В ЭВМ семейства VAX имеется еще два флага условий: бит N и бит Z. Все эти 4 флага называются *кодами условий*. Биты N и Z предназначены для упрощения проверки условий, возникающих в ходе выполнения любых арифметических операций. В гл. 5 уже рассматривались инструкции проверки и сравнения; покажем теперь, что в действительности происходит при выполнении условных переходов.

Бит N устанавливается в 1 всегда, когда результат операции отрицателен. Это происходит даже при выполнении такой простой инструкции, как MOVL. После ее выполнения бит N будет установлен, если пересылаемое число отрицательно. Бит N сбрасывается, если число является положительным. На самом деле значение бита N обычно равно значению знакового или старшего бита результата. Однако переполнение может привести к тому, что знаковый бит результата будет содержать неверное значение. Фактически бит N отражает значение знакового бита *правильного* результата, даже если при переполнении результат не может целиком поместиться в ячейке результата операции.

Бит Z в определенном смысле аналогичен биту N. Бит Z устанавливается в 1 в случае, если результатом операции является число 0. Бит Z будет сброшен, если результат *не равен* 0. В том же смысле, что и бит N, бит Z отражает состояние правильного результата даже в случае переполнения. Значения битов N и Z, так же как и других битов кодов условий (битов С и V), могут проверяться с помощью инструкций перехода. Но в отличие от инструкций перехода по значению битов С и V, в мнемоническом обозначении которых фиксируется имя конкретного бита кодов условий (С или V), инструкции перехода по значению битов N и Z имеют мнемонику, обозначающую только тип условного перехода. Ниже приведены четыре инструкции, которые составляют список инструкций условных переходов по значению одного бита кодов условий:

Мнемоника	Значение
BNEQ	Перейти, если предыдущий результат *не равен 0*, т.е. если бит Z сброшен
BEQL	Перейти, если предыдущий результат *равен* нулю, т.е. если бит Z установлен в 1.
BGEQ	Перейти, если предыдущий результат *больше или равен* 0, т.е. если бит N сброшен
BLSS	Перейти, если предыдущий результат меньше нуля, т.е. если бит N установлен в 1.

Заметьте, что эти же инструкции описывались в гл. 5. Для дополнения этого списка рассмотрим еще две инструкции, которые осуществляют переход по комбинации значений битов кодов условий:

Мнемоника	Значение
BGTR	Перейти, если предыдущий результат *больше* 0, т.е. оба бита N и Z сброшены
BLEQ	Перейти, если предыдущий результат *меньше или равен* 0, т.е. если либо бит N, либо бит Z установлен в 1.

ИНСТРУКЦИИ ОБРАБОТКИ ДАННЫХ

Инструкции обработки данных устанавливают биты кодов условий в нулевое или единичное состояние в соответствии с результатом операции обработки данных. Такие инструкции, как MOVL, CLRL, а также логические операции не могут вызывать переполнения. При выполнении этих инструкций бит V сбрасывается, но значение бита С не изменяется. Почти то же самое происходит при выполнении инструкций *проверки*, рассмотренных в гл. 5. Инструкциями обработки данных являются инструкции, работающие непосредственно с данными, например MOVL, ADDL, SUBL и т.д. В отличие от них инструкции управления, такие как JMP, BRB, BNEQ и т.д., могут осуществлять проверку значений битов кодов условий, но не могут работать непосредственно с данными и, следовательно, устанавливать в необходимое состояние биты кодов условий.

Среди инструкций обработки данных имеется две дополнительные группы инструкций: TSTL, TSTW, TSTB и CMPL, CMPW, CMPB. Частично эти инструкции были описаны

в гл. 5, но понять, как она работают, можно, только рассмотрев их совместно с состоянием кодов условий. По инструкции TSTL выполняется выборка элемента данных, проводится его проверка и устанавливаются в соответствующие состояния коды условий. На этом действие инструкции завершается. При выполнении инструкции данные не изменяются и не используются для других целей. Инструкция TSTL всегда сбрасывает биты V и C, поскольку проверка числа не может вызвать ни одного из видов переполнения. Биты N и Z будут устанавливаться в соответствующее состояние в зависимости от того, меньше или равно 0 просматриваемое число.

Инструкция CMPL по действию аналогична инструкции TSTL, но в отличие от нее оперирует двумя данными. По инструкции CMPL A, B происходит вычитание содержимого длинного слова с адресом B из содержимого длинного слова с адресом A, при этом результат вычитания проверяется и в соответствии с его значением устанавливаются биты кодов условий, после чего результат уничтожается. Таким образом, результат не сохраняется и содержимое длинных слов A и B не изменяется, что и позволяет использовать инструкцию CMPL так, как было показано в предыдущих главах, например:

```
CMPL    A, B
BEQL    10$
```

Необходимо отметить, что инструкции условного перехода BGEQ, BGTR, BLSS и BLEQ не следует использовать при обработке чисел без знака. Например, если рассматривать содержимое длинных слов как число без знака, то число ^XFFFFFFFF больше числа ^X00000000. Однако операция сравнения, предшествующая инструкции BGTR, даст противоположный результат, поскольку число ^XFFFFFFFF будет рассматриваться как отрицательное, а именно как –1. Для решения этой проблемы предусмотрены четыре инструкции условного перехода, применяемые при обработке чисел без знака:

Мнемоника	Значение
BGTRU	Перейти, если *больше* (для чисел без знака)
BGEQU	Перейти, если *больше или равно* (для чисел без знака)
BLEQU	Перейти, если *меньше или равно* (для чисел без знака)
BLSSU	Перейти, если *меньше* (для чисел без знака)

Эти инструкции имеют смысл только тогда, когда они используются совместно с инструкциями сравнения. Так, с помощью следующих инструкций можно перейти к метке 10$, если содержимое длинного слова с адресом A больше, чем содержимое длинного слова с адресом B (в беззнаковом представлении):

```
CMPL    A, B
BGTRU   10$
```

Любопытно отметить, что инструкция BLSSU идентична инструкции BCS. На машинном языке эти инструкции имеют одинаковый код операции ^X1F (см. п. 6 в упр. 6.1).

Сравнение или проверка на равенство осуществляется одинаково для чисел со знаком и без знака. Однако для того, чтобы избавить программиста от необходимости помнить, когда надо использовать различные переходы для чисел со знаком и без знака, в языке ассемблера предусмотрены мнемонические коды BEQLU и BNEQU, которые в действительности при ассемблировании переводятся в те же машинные коды операции, что и мнемоника инструкций BEQL и BNEQ. Это позволяет программисту быть последовательным и использовать символ U каждый раз, когда он имеет дело с числами без знака.

ИНСТРУКЦИИ ПРЕОБРАЗОВАНИЯ ДАННЫХ И ЗНАКА

Часто в одной задаче возникает необходимость в совместной обработке различной информации, представленной в различных форматах: байта, слова и длинного слова. Но при

вычитании и сложении данных различного формата возникают трудности. Так, нельзя непосредственно сложить содержимое байта с содержимым слова. Для этого необходимо преобразовать формат байта в формат 16-битового слова. Затем содержимое полученного слова можно сложить с содержимым другого слова. С помощью следующих инструкций можно к содержимому байта с адресом B прибавить содержимое слова с адресом W:

```
CLRW    R0          ;ОЧИСТИТЬ МЛАДШИЕ БИТЫ РЕГИСТРА R0
MOVB    B, R0       ;ПЕРЕСЛАТЬ В МЛАДШИЕ 8 БИТ ЧИСЛО B
ADDW2   R0, W       ;И СЛОЖИТЬ ВОСЬМИБИТОВОЕ ЧИСЛО С W
```

Для выполнения этой операции требуется последовательность из трех инструкций, а также временная память для хранения содержимого нового слова. В данном случае для этого используется регистр R0, но можно было бы использовать и ячейку памяти. Инструкция CLRW обнуляет 16 битов регистра R0. Затем содержимое байта с адресом B пересылается в восемь младших битов слова в регистре R0. После этого новое содержимое слова в регистре R0 складывается с содержимым слова, расположенного по адресу W.

При применении этого способа необходимо помнить, что содержимое байта должно быть числом без знака. Поскольку инструкция CLRW обнуляет старшие 8 битов слова в регистре R0, этот способ нельзя использовать, если содержимое байта должно интерпретироваться как число со знаком. Это означает, что содержимое регистра R0 рассматривается всегда как положительное число от 0 до 255. Для сложения чисел со знаком, представленных в различном формате, необходимо проверить значение знакового бита числа, содержащегося в байте. В случае отрицательного числа старшие биты слова в регистре R0 необходимо установить равными 1, а не 0, что может быть реализовано с помощью следующих инструкций:

```
        CLRW    R0          ;ОЧИСТИТЬ МЛАДШИЕ БИТЫ РЕГИСТРА R0
        TSTB    B           ;ЕСЛИ ЧИСЛО В  ОТРИЦАТЕЛЬНОЕ,
        BGEQ    10$
        MOVW    #-1, R0     ;ТО ВО ВСЕХ БИТАХ ДОЛЖНЫ БЫТЬ ЕДИНИЦЫ
10$:    MOVB    B, R0       ;ПЕРЕСЛАТЬ В МЛАДШИЕ 8 БИТ ЧИСЛО В
        ADDW2   R0, W       ;И СЛОЖИТЬ ЧИСЛО СО ЗНАКОМ  С W
```

Как видите, это довольно сложно. Поэтому ЭВМ семейства VAX включают группу инструкций преобразования формата, которые копируют числа со знаком, представленные в формате байта, слова или длинного слова, в области памяти требуемого формата. Приведенный ниже список включает инструкции преобразования формата, работающие с целочисленными операндами. Кроме того, существуют инструкции преобразования формата, работающие с операндами с плавающей точкой; они рассматриваются в гл. 12.

Мнемоника	Значение
CVTBW	Преобразовать байт в слово
CVTBL	Преобразовать байт в длинное слово
CVTWB	Преобразовать слово в байт
CVTWL	Преобразовать слово в длинное слово
CVTLB	Преобразовать длинное слово в байт
CVTLW	Преобразовать длинное слово в слово

На самом деле эти инструкции действуют так же, как инструкции пересылки семейства MOV, за исключением того, что форматы операнда-источника и операнда-получателя различны. При выполнении этих инструкций значения бита N и бита Z устанавливаются в зависимости от значения результата так, как это было описано выше. Бит C всегда сбрасывается. Бит V устанавливается равным 1 в случае, если результат (число со знаком) слишком велик и выходит за пределы требуемого формата. Очевидно, что это может произойти только при выполнении инструкций, которые преобразуют бо́льшие форматы

в ме́ньшие, например CVTWB. Операнд-источник, представляющий собой слово, вполне может иметь значение больше 127 или меньше −128, но оно может не поместиться в формате байта (формат результата).

Поясним, как работают эти инструкции на примере предыдущей программы, в которой происходит сложение двух чисел со знаком, представленных в форматах байта и слова. Перепишем эту программу следующим образом:

```
CVTBW   B,R0
ADDW2   R0,W
```

Эти инструкции выполняют то же самое, что и предыдущая программа. Инструкции преобразования ме́ньших форматов в бо́льшие, такие как CVTBW, заполняют старшие биты результата значением знакового бита исходного операнда. Например, содержимое байта ^X03 будет расширено для представления в формате слова как ^X0003, но при преобразовании ^X83 получается ^XFF83.

Для инверсии знака числа используется другая группа инструкций. Приведенные ниже инструкции по действию аналогичны следующим операторам Фортрана или Паскаля:

```
Фортран         Паскаль

X = -Y          X := -Y;
```

Мнемоника	Значение
MNEGB	Переслать байт с изменением знака
MNEGW	Переслать слово с изменением знака
MNEGL	Переслать длинное слово с изменением знака

Эти инструкции имеют два операнда и осуществляют пересылку дополнительного кода исходного числа, изменяющего знак числа на противоположный, в область памяти операнда-получателя. Например, инструкция

```
MNEGL   X,R0
```

осуществит пересылку 32-битового дополнительного кода операнда-источника с адресом X в регистр R0, изменив таким образом знак исходного числа на противоположный.

Данные инструкции обрабатывают биты N и Z обычным образом, устанавливая знак результата и признак равенства результата равным 0. Бит V устанавливается равным 1, если результат переполняет разрядную сетку. Напомним, что при представлении чисел в дополнительном коде положительных чисел будет на одно меньше, чем отрицательных. Так, самое большое по абсолютному значению отрицательное число, которое можно представить в формате слова, равно −32768, или ^X8000, а самое большое положительное число равно +32767, или ^X7FFF. Поэтому не существует противоположного по знаку значения для отрицательного числа ^X8000, представленного в формате слова. При этом произойдет установка бита V.

Использование бита C несколько необычно. Он устанавливается равным 1, если результат не равен 0. Это может показаться бессмысленным, но является полезным при изменении знака чисел с многократно увеличенной точностью (см. раздел этой главы, посвященный многократно увеличенной точности). Фактически логика состоит в том, что получение противоположного по знаку значения числа эквивалентно вычитанию его из 0. Например, инструкция

```
MNEGL   X,R0
```

установит бит C в то же состояние, что и пара инструкций


```
CLRL    R0
SUBL2   X,R0
```

(На самом деле каждый из четырех битов кодов условий N, Z, V и C будет установлен в обоих случаях в одно и то же состояние.) Если содержимое длинного слова с адресом X не равно 0, то в результате выполнения инструкции SUBL2 бит C установится равным 1, потому что вычитание ненулевого значения из 0 всегда вызывает переполнение числа без знака.

Инструкции преобразования формата и инструкции пересылки с изменением знака предназначены для работы с числами со знаком. Также бывает необходимо переслать число без знака в область памяти большего формата, например, чтобы сложить два числа без знака, т.е. содержимое байта и длинного слова. Инструкции преобразования формата, такие как CVTBL, не могут использоваться как надежное средство для этой цели, поскольку содержимое байта будет в данном случае рассматриваться как число со знаком.

Одним из решений этой проблемы является использование пары инструкций для преобразования числа без знака в формате байта в формат длинного слова. Например,

```
CLRL    R0
MOVB    B,R0
```

Однако имеется группа инструкций, которые выполняют те же действия быстрее и занимают меньше памяти. Они работают так же, как инструкции преобразования формата, за исключением того, что их операнды интерпретируются как целые числа без знака. Кроме того, они обеспечивают только пересылку с преобразованием меньшего формата операнда в больший. Этот набор представлен ниже.

Мнемоника	Значение
MOVZBW	Переслать содержимое байта, дополненное незначащими нулями, в слово
MOVZBL	Переслать содержимое байта, дополненное незначащими нулями, в длинное слово
MOVZWL	Переслать содержимое слова, дополненное незначащими нулями, в длинное слово

Эти инструкции обнуляют старшие биты операнда-получателя и пересылают значение операнда-источника в младшие биты операнда-получателя. Бит V сбрасывается, так как переполнения, очевидно, быть не может. Бит N сбрасывается, поскольку в знаковый бит операнда-получателя засылается нуль. Значение бита C не изменяется.

Может показаться странным, что нет инструкций преобразования большего формата в меньший для чисел без знака, аналогичных, например, инструкции CVTLB, которая преобразует форматы чисел со знаком. Дело в том, что преобразовать больший формат в меньший для числа без знака можно с помощью обычной инструкции пересылки. Например, инструкция

```
MOVB    L,B
```

осуществит пересылку младших восьми битов длинного слова с адресом L в байт с адресом B. Однако инструкция CVTLB имеет преимущество, так как позволяет обнаруживать ошибки. Признаком ошибки является установка бита V, которая происходит, если результат выходит за рамки требуемого формата. К сожалению, такая проверка для чисел без знака невозможна, поскольку бит V устанавливается только при выполнении инструкций, работающих с числами со знаком, а бит C обычно используется как признак переноса и не может использоваться как бит признака общего переполнения.

УПРАЖНЕНИЯ 6.1

1. Приведите десятичные эквиваленты для следующих слов, представленных в шестнадцатеричном виде, интерпретируя их как числа без знака, а затем как 16-битовые числа со знаком в дополнительном коде:

а) ^X00FD; б) ^XFFFD; в) ^X7FFF;
г) ^XF716; д) ^X8000; е) ^X8001;
ж) ^X80FD; з) ^X7655; и) ^X6EEF.

2. Для каждого числа из п. 1 упражнения 6.1 покажите его представление в 16-битовом дополнительном коде при изменении знака числа на противоположный, как это было бы сделано инструкцией MNEGW. Кроме того, для каждого случая укажите значения битов N, Z, V и C, которые могут изменяться в результате выполнения этой инструкции.

3. Сложите пары слов так, как это было бы сделано инструкцией ADDW3. Кроме того, для каждой пары чисел укажите значения битов N, Z, V и C, которые могут изменяться в результате выполнения этой инструкции:

а) ^X011A и ^X00FD; б) ^X6A5A и ^X7FEC; в) ^X65ED и ^XFFEB;
г) ^XFFFF и ^XFFFF; д) ^X7FFF и ^X8001; е) ^XD7B7 и ^X6DF8.

4. Выполните п. 3 упр. 6.1, заменив сложение вычитанием *первого* числа из *второго*, так, как это было бы сделано инструкцией SUBW3.

5. Покажите, что в случае представления чисел в дополнительном коде при возникновении переполнения числа со знаком знаковый бит результата имеет значение, противоположное тому, которое следовало бы ожидать.

6. Объясните, почему инструкция BLSSU аналогична инструкции BCS. Какая инструкция условного перехода для чисел без знака аналогична инструкции BCS? Объясните, почему.

7. Ниже приведен список значений, которые являются исходным содержимым регистра R0. Для каждого из этих значений укажите, каким станет новое содержимое регистра R0 и какие значения примут биты N, C, V и Z в результате выполнения инструкций INCL R0, DECL R0, INCW R0, DECW R0:

а) ^X7FFFFFFF; б) ^X80000000; в) ^XFFFFFFFF;
г) ^X00000000; д) ^X00000001; е) ^X80000001.

8*. Напишите подпрограмму, которая печатает четыре числа, каждое из которых может иметь значение ^X00000000 или ^X00000001, отражающее текущие значения битов N, Z, V и C. (*Примечание*: выполнение инструкции JSB не влияет на коды условий, так же как не влияет на коды условий выполнение инструкций переходов различного типа. Однако коды условий изменяются при выполнении таких инструкций, как MOVL. Набор инструкций ЭВМ VAX включает специальные инструкции (нами еще не рассмотренные), которые могли бы упростить выполнение этого задания, но мы рекомендуем обойтись пока без них. Напишите программу в виде дерева переходов). Кроме того, напишите основную программу, которая должна тестировать подпрограмму следующим образом: она должна выполнять различные вычисления, после которых вызывать подпрограмму, которая будет выводить на печать результаты (т.е. значения битов N, Z, V и C), полученные после каждого вычисления.

6.3. УМНОЖЕНИЕ И ДЕЛЕНИЕ

ОСНОВНЫЕ ИНСТРУКЦИИ

Для умножения целых чисел в ЭВМ VAX имеются две группы инструкций: основные инструкции и инструкции расширенной арифметики. Основные инструкции предназначены для простого умножения и деления содержимого байтов, слов и длинных слов, как таковых. Инструкции расширенной арифметики, которые будут рассматриваться позже, позво-

ляют неограниченно расширять точность представления результата и содержат такие дополнительные операции, как вычисление остатка от деления.

В группу основных инструкций умножения и деления входят следующие:

```
MULB2    DIVB2
MULB3    DIVB3
MULW2    DIVW2
MULW3    DIVW3
MULL2    DIVL2
MULL3    DIVL3
```

Во многом они действуют аналогично инструкциям сложения и вычитания с двумя и тремя операндами. Например, инструкция

```
MULL3    A, B, C
```

перемножит содержимое длинных слов с адресами A и B и поместит результат в длинное слово с адресом C. Аналогично инструкция

```
MULB2    X, Y
```

перемножит содержимое байтов с адресами X и Y и поместит результат в байт с адресом Y. Необходимо отметить, что в результате умножения могут получиться большие числа. Поэтому для предотвращения переполнения необходимы специальные меры, особенно при умножении содержимого байтов, поскольку инструкции умножения, работающие с байтами, имеют очень ограниченный диапазон возможных значений произведения.

Инструкции деления действуют аналогично, за исключением того, что в них, как и при вычитании, важен порядок следования операндов. Эти инструкции всегда осуществляют деление *второго* операнда на первый. Так, инструкция

```
DIVL3    A, B, C
```

разделит содержимое длинного слова с адресом B на содержимое длинного слова с адресом A и поместит частное в длинное слово с адресом C. Аналогично инструкция

```
DIVB2    X, Y
```

разделит содержимое байта с адресом Y на содержимое байта с адресом X и поместит частное в байт с адресом Y.

Между инструкциями умножения и деления и инструкциями сложения и вычитания имеется существенное различие. Как было показано ранее, числа со знаком и без знака обрабатываются основными инструкциями сложения и вычитания одинаково; единственная разница в их обработке состоит в том, что по-разному обнаруживается и интерпретируется переполнение. Тогда как умножение и деление чисел со знаком существенно отличается от умножения и деления чисел без знака.

Все инструкции умножения и деления, описанные в этом разделе, предназначены для работы с числами *со знаком*. Обработка знаков чисел подчиняется обычным правилам алгебры. Умножение и деление чисел с одинаковыми знаками дают положительный результат (произведение или частное). Умножение и деление чисел с разными знаками дают отрицательный результат.

При делении целых чисел основная трудность заключается в том, что результат деления может не быть целым числом. Описанные в этом разделе инструкции целочисленного деления ЭВМ VAX используют те же правила обработки результата, что и большинство языков высокого уровня, таких как Фортран или Паскаль. Дробная часть частного всег-

да усекается, и результат, таким образом, "округляется" в сторону 0. Это справедливо как для положительных, так и для отрицательных результатов. Так, результат деления 7 на 2 с усечением равен 3. Аналогично результат деления –7 на 2 с усечением равен –3.

Наконец, рассмотрим, как эти инструкции воздействуют на коды условий. Как и следовало ожидать, биты N и Z устанавливаются или сбрасываются в зависимости от того, каким является результат (произведение или частное): отрицательным или равным нулю. При этом имеется в виду реальный результат, записанный в памяти, без учета потерянных из-за переполнения или округления битов.

Бит V устанавливается при переполнении. Так как в результате умножения могут получаться большие числа, то легко понять, где именно возникает переполнение. Бит V устанавливается, если произведение слишком велико и выражается числом, которое не помещается в заданном формате результата. Другими словами, при умножении чисел в формате байта их произведение должно находиться в диапазоне значений ^X80 – ^X7F. При умножении чисел в формате слова значение произведения должно находиться в диапазоне значений ^X8000 – ^X7FFF, а в случае длинных слов значение произведения должно быть в диапазоне значений ^X00000000-^X7FFFFFFF.

Может быть непонятно, каким образом возникает переполнение при делении, поскольку частное обычно меньше делимого, которое уже находится в заданном формате. Очевидной исключительной ситуацией является деление на 0. Понятно, что при попытке деления на 0 будет установлен бит N. Но имеется еще одна исключительная ситуация, которая является результатом асимметрии, свойственной системе представления чисел в дополнительном коде. Вспомним, что наибольшее по абсолютному значению отрицательное число в таком представлении не имеет противоположного себе по знаку. Так, в формате байта можно представить числа от –128 до +127, в формате слова – от –32768 до 32767, в формате длинного слова – от –2147483648 до +2147483647. Поэтому при выполнении арифметических операций над числами в формате слова для числа –32768 не существует числа, противоположного ему по знаку. Это означает, что инструкции DIVW2 и DIVW3 при попытке деления числа –32768 на –1 вызовут переполнение. В аналогичной ситуации возникает переполнение при выполнении инструкций деления, работающих с байтами и длинными словами. И во всех случаях произойдет установка бита V в 1.

В заключение рассмотрим, что происходит с битом C. Вспомним, что бит C используется главным образом для операций над числами без знака, а рассматриваемые инструкции деления и умножения используются только для арифметических операций над числами со знаком. Таким образом, эти инструкции *всегда* сбрасывают бит C вне зависимости от значения полученного результата.

ИНСТРУКЦИИ РАСШИРЕННОГО УМНОЖЕНИЯ И ДЕЛЕНИЯ

При умножении двух n-битовых чисел результат может в общем случае занимать 2*n битов. Принимая во внимание эту возможность наряду с возможностью двухкратного уменьшения разрядности числа при делении, в ЭВМ семейства VAX предусмотрены две инструкции расширенного умножения и деления. Инструкция расширенного умножения осуществляет умножение двух 32-битовых чисел со знаком, результатом является 64-битовое число со знаком. Инструкция расширенного деления осуществляет деление 64-битового числа (делимого) на 32-битовое число (делитель), в результате получается 32-битовое частное и остаток.

Ниже показано действие инструкции расширенного умножения:

```
EMUL    A, B, C, D
```

где A, B, C и D являются обычными операндами. Операнды A и B – 32-битовые множитель и множимое. Операнд D- 64-битовое произведение, имеющее формат *квадраслова*. Операнд C – 32-битовое число со знаком, прибавляемое к 64-битовому произведению. Эта инструкция часто используется для выполнения таких операций, как перевод из одной системы счисления в другую, где реализуется функция умножения с последующим суммированием. Именно для этой цели в инструкцию EMUL включен операнд C. Если к произведению ничего прибавлять не надо, эту инструкцию всегда можно записать в виде

```
EMUL    A, B, #0, D
```

Отметим, что в инструкции мы впервые сталкиваемся с практическим использованием квадраслов. В гл. 3 упоминалось, что информация может пересылаться блоками размером 8, 16, 32 и 64 бита с помощью инструкций MOVB, MOVW, MOVL и MOVQ соответственно. Инструкция MOVQ осуществляет пересылку квадраслова, т. е. 64 бита или 8 байтов. Обращение к ячейкам памяти при выполнении инструкции MOVQ не представляет сложностей, поскольку эта инструкция является просто расширением других инструкций многобайтовой пересылки MOVW и MOVL. Однако если инструкция MOVQ адресуется не к ячейкам памяти, а к регистрам, то могут возникнуть трудности, связанные с тем, что разрядность регистра 32 бита. Поэтому, если операнд имеет формат квадраслова и применяется режим регистровой адресации, это означает, что операнд расположен в двух регистрах – регистре, указанном в инструкции, и регистре со следующим старшим номером. Таким образом, инструкция

```
MOUQ    X, R0
```

поместит 32 младших бита операнда X, имеющего формат квадраслова, в регистр R0, а старшие 32 бита этого операнда – в регистр R1. Аналогично инструкция

```
EMUL    A, B, #0, R3
```

поместит 64-битовое произведение операндов A и B в регистры R3 и R4.

Инструкция расширенного деления в некотором смысле действует обратно инструкции расширенного умножения. Делимое в формате квадраслова делится на делитель, имеющий формат длинного слова. Получаемые в результате деления частное и остаток имеют формат длинных слов. Приведем формат инструкции расширенного деления:

```
EDIV    D, E, F, G
```

где операнд D –32-битовый делитель, операнд E – 64-битовое делимое, операнды F и G – 32-битовые частное и остаток. Если остаток не равен 0, его знак соответствует знаку делимого. Поскольку в этой инструкции все операнды являются числами со знаком, определение знака частного следует обычным правилам алгебры. Если делимое и делитель имеют одинаковый знак, предполагается, что частное будет положительным. Если делимое и делитель имеют разные знаки, то предположительно частное будет отрицательным. В действительности знак частного может отличаться от предполагаемого в случае, если частное равно нулю или произошло переполнение.

Переполнение может возникать по двум причинам. Бит V установится при делении на 0. Но при делении 64-битового числа на 32-битовое число *не гарантируется* получение 32-битового результата. Например, деление числа ^X000123456789ABCD на число ^X0000100 дает частное ^X00000123456789AB и остаток ^X000000CD. Отметим, что остаток всегда меньше делителя и, следовательно, должен поместиться в 32 битах. Частное в этом случае не умещается в формате длинного слова, и, следовательно, произойдет переполнение.

Последнее замечание касается того, что инструкции расширенного умножения и деления в некотором роде симметричны. В частности, операнд-слагаемое в инструкции EMUL является отчасти дополнением остатка в инструкции EDIV. Например, рассмотрим расширенное деление

```
EDIV    D, E, F, G
```

Если при выполнении этой инструкции не возникло переполнения, то расширенное умножение

```
EMUL    D, F, G, X
```

даст результат в формате квадраслова (операнд X), идентичный по значению делимому (операнд E) в инструкции EDIV, имеющему такой же формат.

При выполнении этих инструкций биты N и Z устанавливаются в зависимости от значения результата: в случае инструкции EMUL – произведения, в случае инструкции EDIV – частного. Бит V установится, если при выполнении инструкции EDIV возникнет переполнение. При выполнении инструкции EMUL переполнение невозможно, так как 32-битовые операнды при умножении всегда дают результат, занимающий не более 64 бита. Бит C, так же как и в случае основных инструкций умножения и деления, всегда сбрасывается.

6.4. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ С ПОВЫШЕННОЙ ТОЧНОСТЬЮ

ПРЕДСТАВЛЕНИЕ С ДВОЙНОЙ ТОЧНОСТЬЮ

Как было показано выше, существуют инструкции сложения и вычитания для чисел в формате байтов (8 битов), слов (16 битов) и длинных слов (32 бита). В форматах байтов и слов можно представить числа весьма ограниченного диапазона, но даже в формате длинного слова не всегда удается представить требуемое число. Числа без знака, представляемые в формате длинного слова, имеют значения от 0 до 4294967295. Представленные в этом же формате числа со знаком могут иметь значения от −2147483648 до +2147483647. Казалось бы, такой диапазон достаточно велик для большинства задач, однако существуют некоторые задачи, оперирующие бóльшими числами. Для того чтобы вычислительная система была универсальной, в ней должны быть предусмотрены возможности работы с любыми числами.

Существует только один способ представления числа, не помещающегося в одной ячейке памяти, – использовать для его хранения несколько ячеек. Очевидный способ – размещение числа в двух ячейках, что называется представлением числа с *двойной точностью.*

Число с двойной точностью можно воспринимать просто как 64-битовое число или квадраслово. Оно может размещаться в двух длинных словах: 32 старших бита в одном длинном слове и 32 младших бита – в другом. Часто бывает удобно рассматривать эту пару длинных слов как одно 64-битовое квадраслово. При этом становится очевидной возможность представления 64-битовых чисел как со знаком, так и без знака. Для этого требуется только обобщить идею арифметики в дополнительных кодах и расширить ее область применения с 32-битовых на область 64-битовых чисел. Как и раньше, крайний левый бит всего числа целиком является знаковым. Им мог бы быть знаковый бит старшего из двух длинных слов, используемых для представления числа. (Отметим, что знаковый бит младшего длинного слова является просто одним из битов в середине числа и не имеет отношения к знаку всего числа.)

СЛОЖЕНИЕ И ВЫЧИТАНИЕ ЧИСЕЛ С ДВОЙНОЙ ТОЧНОСТЬЮ

Способность представления чисел с двойной точностью сама по себе малоприменима, если отсутствует возможность выполнения над этими большими числами арифметических операций. На самом деле, как будет видно из следующих примеров, основные арифметические операции можно реализовать довольно легко. Для простоты изложения представим, что шестизначные *десятичные* числа обрабатываются в ЭВМ, которая имеет трехзначные десятичные слова.

Рассмотрим сложение


```
  1 2 3     4 5 6
+ 1 1 2   + 2 3 3
-------   -------
  2 3 5     6 8 9
```

Как видно, правая половина суммы равна сумме правых половин складываемых чисел. Аналогично левая половина суммы равна сумме левых половин складываемых чисел.

Но следует отметить, что этот пример является частным случаем, поскольку при сложении правых половин чисел может возникнуть переполнение. Например,

```
      1          Перенос
  1 2 3        7 8 9
+ 1 1 2      + 5 6 6
-------      -------
  2 3 6        3 5 5
```

В данном случае необходимо прибавить значение бита переноса, устанавливаемого при переполнении, к сумме левых половин.

Аналогичная процедура используется при вычитании:

```
      1          Заем
  2 3 6        3 5 5
- 1 1 2      - 5 6 6
-------      -------
  1 2 3        7 8 9
```

В данном случае возникает переполнение числа без знака, так как делается попытка вычесть 566 из 355, а число без знака не может быть отрицательным. Это приводит к необходимости заема единицы из разности старших половин чисел.

ИНСТРУКЦИИ СЛОЖЕНИЯ И ВЫЧИТАНИЯ С ПЕРЕНОСОМ

Описанные выше методы применимы также для операций двоичной арифметики, выполняемых с 32-битовыми регистрами. В ЭВМ семейства VAX бит C показывает наличие переноса при сложении (или вычитании) младших частей больших чисел. Следовательно, для учета переноса необходимо прибавить (вычесть) значение бита C к (из) старшей части результата. Конечно, можно было бы проверить значение бита C, а затем увеличить (или уменьшить) результат, находящийся в регистре, на 1. Специально для этой цели разработчики ЭВМ VAX ввели следующие инструкции. Для сложения с переносом используется инструкция

Мнемоника	Значение
ADWC	Сложить с переносом

По этой инструкции выполняется сложение двух операндов, имеющих формат длинных слов и значения бита признака переноса C, итоговая сумма помещается по адресу второго операнда. Для вычитания с заемом (переносом) используется следующая инструкция:

Мнемоника	Значение
SBWC	Вычесть с заемом (переносом)

По данной инструкции выполняется вычитание первого операнда, имеющего формат длинного слова и значения бита C, из второго операнда, имеющего формат длинного слова, разность помещается по адресу второго операнда.

Эти две инструкции могут использоваться совместно с другими арифметическими инструкциями для сложения 64-битовых чисел A и B. Пусть числа A и B содержатся в длинных словах с адресами AL, AR, BL, BR, где буквы L и R обозначают левую и правую половину числа соответственно. Сложение с двойной точностью осуществляется с помощью инструкций

```
ADDL2    AR, BR
ADWC     AL, BL
```

Аналогично вычитание A из B производится с помощью инструкций

```
SUBL2    AR, BR
SBWC     AL, BL
```

Каждая из этих программ может быть расширена для работы с тройной точностью, т.е. для работы с 96-битовыми числами и даже с еще большей точностью. Требуется только последовательное использование инструкций ADWC или SBWC для всех старших длинных слов, занимаемых числом.

6.5. АЛГОРИТМЫ УМНОЖЕНИЯ И ДЕЛЕНИЯ

ИНСТРУКЦИИ СДВИГА

В десятичной системе счисления легко умножать и делить на 10 или на любую степень числа 10, например 100, 1000, 10000 и т.д. Для этого нужно только приписать к числу нули справа при умножении или отбросить цифры справа при целочисленном делении. В регистре это выполняется с помощью сдвига числа: влево – при умножении, вправо – при делении. Представим себе ЭВМ с 10-разрядными регистрами для хранения десятичных чисел. Для такой ЭВМ при умножении 0000057342 на 1000 получим 0057342000; а при делении на 100 – 0000000573.

Такой способ умножения и деления на степень основания системы счисления применим для систем счисления с любым основанием. Следовательно, в ЭВМ с двоичной системой счисления умножение и деление чисел на 2^n может осуществляться посредством сдвига числа влево или вправо на n двоичных разрядов. Например, умножение 16-битового числа 0000010011011110 на 8_{10} или 1000_2 равно 0010011011110000; деление числа 0000010011011110 на 4_{10} или на 100_2 равно 0000000100110111. Эти примеры показывают, что происходит с положительными числами. Однако непонятно, что произойдет при сдвиге отрицательных чисел. При сдвигах влево проблем нет, поскольку 1111111111111000 равно -8_{10}, а 1111111111110000 равно -16_{10}. Единственная проблема, которая могла бы возникнуть, связана с тем, что при сдвиге влево на большое число разрядов крайний левый (знаковый) бит может быть обнулен. Но это есть не что иное, как случай переполнения, которое аналогично переполнению при сдвиге положительного числа влево, когда происходит потеря левых битов или когда в знаковом разряде оказывается 1. Переполнение такого типа возникает всегда, когда делается попытка в некоторую область памяти поместить число, значение которого выходит за пределы диапазона представимых чисел.

Итак, умножение и деление чисел со знаком на степень числа 2 может осуществляться посредством сдвига числа влево или вправо. При сдвиге влево освобождаемые разряды заполняются нулями. При сдвиге вправо такие разряды заполняются значением знакового бита. Смысл этого иллюстрируется следующими двумя примерами. Числа

```
0000000000010000 = 16                    1111111111110000 = –16
                       разделить на 4
0000000000000000 = 4                     1111111111111100 = –4
(пустые разряды заполняются нулями)      (пустые разряды заполняются единицами)
```


В наборе инструкций ЭВМ семейства VAX имеется две инструкции для выполнения операций такого типа[1]:

```
ASHL    N, A, B
```

и

```
ASHQ    N, X, Y
```

В обоих случаях N – 8-битовое слово со знаком, определяющее количество разрядов, на которое следует произвести сдвиг. Если N положительно, то происходит сдвиг влево. Если N отрицательно, то происходит сдвиг вправо. Фактически это аналогично умножению на 2^n или на 2^{-n}. При выполнении инструкции ASHL операнд A – содержимое длинного слова с адресом A – сдвигается на N разрядов и полученный результат помещается в длинное слово с адресом B. Инструкция ASHQ выполняется аналогично, за исключением того, что операнды X и Y имеют формат 64-битовых квадраслов.

Указанные инструкции устанавливают биты N и Z в зависимости от значения результата. Бит V будет установлен, если при сдвиге влево получен результат, значение которого выходит за пределы диапазона представимых чисел. Поскольку эти инструкции предназначены для работы только с числами со знаком, бит C всегда сбрасывается.

Остановимся на проблеме потери битов при сдвиге вправо. Понятно, что при сдвиге положительного числа вправо потерянные биты есть не что иное, как дробный остаток. Отбрасывание этих битов приводит к усечению числа. Оно аналогично усечению чисел при выполнении над целыми числами операции деления (/) в Фортране или при выполнении операции целочисленного деления DIV в Паскале. К сожалению, при сдвиге вправо отрицательных чисел этого не происходит, что видно из следующего примера:

```
111...1111011101 = –35
сдвинем вправо на три разряда, получим
111...1111111011 = –5.
```

В целочисленной арифметике Фортрана и Паскаля деление –35 на 8 дает –4, а не –5, как в данном примере. Это объясняется тем, что при сдвиге вправо чисел в дополнительном коде округление происходит в меньшую сторону (в алгебраическом смысле). Поэтому –4.375 округляется в меньшую сторону до следующего *более отрицательного* целого числа. Для того чтобы данный способ деления соответствовал целочисленному делению в Фортране и Паскале, необходимы дополнительные инструкции (см. п. 6 в упр. 6.2).

Рассматривая инструкции сдвига, нельзя не упомянуть еще об одной инструкции сдвига, которая используется в первую очередь для упаковки и распаковки данных и работает с длиными словами. Ранее было показано, как длинное слово может быть разбито на байты. Однако иногда бывает полезным разбивать длинные слова на части с нечетным числом битов. После этого возникает необходимость в сдвиге содержимого длинного слова с целью выравнивания указанных частей по определенной границе. В гл. 8 подробно обсуждается, зачем нужен такой тип операций. Эта инструкция работает почти так же, как инструкции арифметических сдвигов, за исключением того, что освобождаемые разряды заполняются не нулями и не значением знакового бита, а значениями тех разрядов, которые теряются при сдвиге. Эта инструкция обрабатывает операнд – длинное слово – так, как если бы были замкнуты его концы[2].

Инструкция имеет формат

```
ROTL     N, A, B
```

---

[1] Эти инструкции называются инструкциями арифметического сдвига. – *Прим. ред.*

[2] Операнд по сути рассматривается как замкнутое кольцо битов. Подобные инструкции называются инструкциями циклического сдвига. – *Прим. ред.*

Так же как в инструкции ASHL, N является 8-битовым числом, задающим количество разрядов, на которое нужно сделать сдвиг. В результате выполнения этой инструкции содержимое длинного слова с адресом A циклически сдвигается на N разрядов и помещается в длинное слово с адресом B. В случае, если N положительно, циклический сдвиг осуществляется влево, если отрицательно – вправо. Например, при циклическом сдвиге числа ^X12345678 на 12 разрядов влево получится число ^X45678123. Напомним, что каждая шестнадцатеричная цифра соответствует четырем битам, следовательно, сдвиг на 12 битов равнозначен сдвигу на три шестнадцатеричных разряда. Сдвиги на количество битов, не кратное четырем, нельзя проиллюстрировать так же просто с помощью шестнадцатеричных чисел: эти числа нужно переводить в двоичный вид. При выполнении инструкции ROTL биты N и Z устанавливаются, как обычно, в зависимости от значений результата, при этом не важно, какие значения имеют биты V и C. После выполнения инструкции бит V сбрасывается, а значение бита C сохраняется.

ОСНОВНЫЕ АЛГОРИТМЫ УМНОЖЕНИЯ И ДЕЛЕНИЯ (ДОПОЛНЕНИЕ)

Пользователям ЭВМ VAX не приходится иметь дело с алгоритмами умножения и деления, поскольку они реализуются аппаратурой процессора ЭВМ VAX. Однако некоторые ЭВМ не имеют инструкций умножения и деления. К ним относятся некоторые модели семейства PDP-11, а также большинство вычислительных систем на базе 8-разрядных микропроцессоров. В этих ЭВМ умножение и деление должно быть реализовано программно. Настоящий раздел посвящен написанию таких программ.

Описанные выше операции сдвига положены в основу большинства эффективных алгоритмов умножения и деления. Причина этого станет понятной при рассмотрении того, как происходит умножение и деление в двоичной системе счисления.

По сути двоичное умножение выполняется так же, как десятичное, но оно проще, поскольку умножение производится только на нуль и единицу. Для простоты в качестве примера рассмотрим умножение содержимого 4-битовых "слов". Пусть необходимо умножить 0110 на 0101:

```
   0110
   0101
  ------
   0110
  0000
 0110
0000
--------
00011110    (8-битовое произведение)
```

Восьмой бит (0 слева) вводится в связи с возможным переносом.

Отметим, что указанная операция состоит из последовательности сдвигов множимого, за которыми могут следовать операции сложения сдвинутых значений множимого. Сдвинутое значение множимого складывается с произведением, если значение соответствующего бита множителя равно 1. Если этот бит содержит 0, сложения не происходит. Классический способ реализации этого алгоритма состоит в использовании трех регистров. Два из них содержат множимое и множитель, а третий используется как накопитель, в котором последовательным суммированием вычисляется произведение. Он называется сумматором. Сумматор и регистр множителя обычно объединяются в один сдвоенный регистр для проведения общего сдвига. После завершения операции умножения в этих двух регистрах будет находиться произведение двойной длины.

Сумматор представляет собой старшую часть сдвоенного регистра. Все содержимое регистра сдвигается вправо таким образом, что разряды сумматора в конце концов оказываются сдвинутыми в младшую часть сдвоенного регистра. Использование сдвига вправо для проведения умножения может показаться необычным, но на примере можно убедиться, что это правильно. Алгоритм умножения можно описать следующим образом.

Шаг 1. Поместить множитель и множимое в соответствующие регистры и очистить регистр-сумматор.

Шаг 2. Если младший бит регистра множителя равен 1, прибавить множимое к значению в сумматоре.

Шаг 3. Сдвинуть содержимое сдвоенного регистра (сумматора и множителя) на один разряд вправо.

Шаг 4. Повторить шаги 2 и 3 столько раз, сколько битов содержит регистр множителя.

Шаг 5. В сдвоенном регистре (сумматора и множителя) теперь находится произведение двойной длины.

В качестве примера рассмотрим, что происходит на каждом шаге алгоритма при умножении чисел 0110 и 0101.

Шаг алгоритма	Регистр-сумматор	Регистр-множитель	Регистр-множимое
1	0000	0101	0110
2	0110	0101	
3	0011	0010	
2	0011	0010	
3	0001	1001	
2	0111	1001	
3	0011	1100	
2	0011	1100	
3	0001	1110	
5	Произведение равно 00011110		

Для данного алгоритма стоит отметить следующее.

Во-первых, как уже упоминалось, при сложении множимого с содержимым сумматора может возникнуть переполнение. Этого можно избежать, если увеличить длину сумматора на один разряд. Читатель может убедиться – этого достаточно, чтобы переполнение никогда не возникало.

Во-вторых, такой алгоритм работает с числами без знака. Но если его немного модифицировать, то можно использовать и для чисел со знаком. В случае отрицательного множимого модификация будет заключаться в том, что при сдвиге вправо пустые разряды слева должны заполняться значением знакового бита, так же как при выполнении инструкций семейства ASH. В случае отрицательного множителя ситуация несколько усложняется. Напомним, что n-битовое отрицательное число x в дополнительном коде должно интерпретироваться как $x - 2^n$. С учетом этого можно модифицировать алгоритм следующим образом: если множитель отрицательный, то необходимо на шаге 5 алгоритма вычесть множимое из содержимого сумматора. Так как непосредственно перед шагом 5 текущее произведение было сдвинуто вправо после прибавления коэффициента 2^{n-1}, то вычитание будет соответствовать умножению множимого на -2^n.

На рис. 6.3 представлена программа, реализующая этот алгоритм. Программа выполняет умножение чисел со знаком во многом аналогично тому, как работает инструкция EMUL: выполняется умножение 32-битовых чисел, содержащихся в регистрах R0 и R1; их произведение, имеющее формат квадраслова, помещается в эти же регистры. Отметим, что две инструкции MOVL с меткой 30$ можно было бы заменить одной инструкцией MOVQ MP, R0. Именно так и нужно сделать, так как две инструкции MOVL были использованы исключительно для наглядности. Обратите внимание на то, что в этой программе применяется новая инструкция BLBC. Она относится к следующей паре инструкций:

Мнемоника		Значение
BLBC	A,LOC	Перейти, если младший бит сброшен
BLBS	A,LOC	Перейти, если младший бит установлен

```
MP:     .BLKL   1               ;МНОЖИТЕЛЬ
AC:     .BLKL   1               ;СУММАТОР (ДОЛЖЕН БЫТЬ ПОСЛЕ MP)
MC:     .BLKL   1               ;МНОЖИМОЕ
MULT:   CLRL    AC              ;ОЧИСТИТЬ AC
        MOVL    R1,MP           ;ЗАДАТЬ МНОЖИТЕЛЬ
        MOVL    R0,MC           ;ЗАДАТЬ МНОЖИМОЕ
        MOVB    #32,R0          ;УСТАНОВИТЬ СЧЕТЧИК БИТОВ
10$:    BLBC    MP,20$          ;ПРОВЕРИТЬ МЛАДШИЙ БИТ MP
        ADDL2   MC,AC           ;ЕСЛИ 1, СЛОЖИТЬ МНОЖИМОЕ
20$:    ASHQ    #-1,AC,AC       ;ВСЕ СДВИНУТЬ ВПРАВО
        DECB    R0              ;И ПОВТОРИТЬ В ЦИКЛЕ 32 РАЗА
        BNEQ    10$
        TSTL    R1              ;ПРОВЕРИТЬ ИСХОДНОЕ ЗНАЧЕНИЕ MP
        BGEQ    30$             ;ЕСЛИ ЧИСЛО ПОЛОЖИТЕЛЬНОЕ,
        SUBL2   MC,AC           ;ТО ВЫЧЕСТЬ MC, ЧТОБЫ ПОЛУЧИТЬ  -2**N;
30$     MOVL    MP,R0           ;ПОМЕСТИТЬ ПРОИЗВЕДЕНИЕ
        MOVL    AC,R1           ;В R0 И R1
        RSB
```

Рис. 6.3. Программа умножения

По инструкции BLBC будет осуществлен переход по адресу LOC, если младший бит содержимого ячейки с адресом A сброшен. Поскольку этот бит – самый младший, не имеет значения, является ли A адресом байта, слова, длинного слова или квадраслова. По инструкции BLBS будет осуществлен переход, если младший бит установлен. Эти инструкции обычно используются для проверки установки признаков ошибок в программах. Оказались полезными они и для реализации шага 2 данного алгоритма.

Необходимо также отметить, что не предпринималось никаких мер для увеличения длины сумматора на один разряд с целью предотвращения переполнения. Это означает, что программа не будет работать, если значение множимого будет слишком большим. Эту проблему можно разрешить, добавив в программу дополнительные инструкции (см. п. 9 упр. 6.2).

Деление может выполняться по алгоритму, который является обратным алгоритму умножения. Но для выявления ситуаций переполнения и деления на нуль в него должны быть включены некоторые дополнительные проверки. Здесь, как и прежде, используется сдвоенный регистр сдвига, состоящий из двух частей: регистра-сумматора и регистра частного. Чтобы исключить возможность переполнения, длину сумматора необходимо увеличить на один разряд, так же как в алгоритме умножения. Первоначально делимое двойной длины помещается в сдвоенный регистр сдвига. Алгоритм описывается следующим образом.

Шаг 1. Поместить делимое двойной длины в сумматор (старшую часть) и в регистр частного (младшую часть). Поместить делитель в регистр делителя.

Шаг 2. В случае, если делитель меньше или равен той части делимого, которая содержится в сумматоре, закончить вычисления с признаком переполнения или деления на нуль.

Шаг 3. Сдвинуть содержимое сдвоенного регистра сумматора частного на один разряд *влево*.

Шаг 4. В случае, если делимое меньше или равно содержимому сумматора, вычесть делимое из содержимого сумматора и установить младший бит регистра частного равным 1.

Шаг 5. Повторить шаги 3 и 4 столько раз, сколько битов содержит регистр частного.

Шаг 6. Теперь в регистре частного находится частное, а в регистре сумматора – остаток.

Рассмотренный алгоритм работает с числами без знака. Для работы с числами со знаком алгоритм необходимо модифицировать (см. п. 11 упр. 6.2).

УПРАЖНЕНИЯ 6.2

1. Перемножьте следующие пары шестнадцатеричных чисел, представляющих содержимое слов, которые интерпретируются как числа со знаком, и укажите, какие операции

приведут к переполнению при выполнении инструкций MULW2:

а) ^X0014; ^X002E;
б) ^X00FC; ^X0088;
в) ^XFFFF; ^X0001;
г) ^XFFFF; ^X0002;
д) ^XFFFF; ^XFFFF;
е) ^XFF01; ^XFEFF.

2. Выполните деление для следующих пар чисел со знаком, представленных в формате слова, и укажите, какие из этих пар дадут при делении результат с ошибкой:

а) ^X0DCB/^X002C;
б) ^X0163/^X7716;
в) ^X0B8C/^X0000;
г) ^X0000/^X0B8C;
д) ^XFFF8/^XFFFB;
е) ^X8000/^XFFFF.

3. Площадь треугольника равна A = b*h/2, где b – длина основания и h – высота. Напишите программу на языке ассемблера для ЭВМ VAX, которая вводит значения b и h, вычисляет и печатает значение A. Программа должна выполняться в цикле до тех пор, пока в качестве входного значения b не будет введен 0. Для представления чисел используйте формат длинного слова.

4. Напишите программу на языке ассемблера, которая вводит список чисел и выводит на печать их среднее арифметическое. Длина списка может быть различной, но ввод списка заканчивается, когда в нем встречается число 0 (число 0 не учитывается при вычислении среднего арифметического).

5. *Простым* числом называется целое число, большее или равное 2, которое не имеет других множителей, кроме 1 и самого себя. Напишите программу на языке ассемблера, которая вычисляет и распечатывает первые 50 простых чисел.

6. Какие инструкции следует использовать совместно с инструкцией ASHL, чтобы при сдвигах вправо округление дробных остатков и положительных, и отрицательных чисел производилось бы без ошибок?

7. Напишите программу на языке ассемблера, которая вводит 10 пар чисел в формате длинного слова и интерпретирует их как числа двойной точности со знаком (64 бита). Ваша программа должна вывести на печать две пары чисел, интерпретируемые как наибольшее и наименьшее числа двойной точности.

8. Проанализируйте программу, показанную на рис. 6.3, Как можно модифицировать эту программу, чтобы включить в нее дополнительный операнд, аналогичный третьему операнду инструкции EMUL? Внесите необходимые изменения и напишите основную программу, которая вводит некоторое количество троек чисел и производит над каждой тройкой операции умножения с суммированием, причем сначала с помощью инструкции EMUL. Программа должна выводить на печать оба множества результатов для сравнения.

9*. В дополнение к модификациям, сделанным в п. 8. упр. 6.2, модифицируйте программу так, чтобы устранить возможность переполнения из-за того, что длина сумматора на один разряд короче, чем необходимо. Выполните программу с критическими значениями исходных данных и убедитесь, что результаты совпадают с результатми выполнения инструкции EMUL.

10. Напишите и протестируйте программу на языке ассемблера, реализующую алгоритм деления чисел без знака, приведенный в последнем разделе данной главы.

11*. Как можно модифицировать алгоритм деления чисел без знака, приведенный в последнем разделе данной главы, чтобы он мог работать для чисел со знаком? Реализуйте алгоритм в виде подпрограммы на языке ассемблера и убедитесь в том, что при выполнении подпрограммы получаются те же результаты, что и при выполнении инструкции EDIV. Как быть в тех случаях, когда возникает переполнение из-за того, что делитель оказывается слишком большим и сумматору не хватает одного бита для представления правильного результата? Как можно учесть эти ситуации? Протестируйте вашу программу на критических значениях исходных данных.

ГЛАВА 7. МАССИВЫ

7.1. ВВЕДЕНИЕ

Большинство читателей знакомы с понятием массива, которое используется в языках программирования высокого уровня, таких как Паскаль и Фортран. Программист имеет некоторую совокупность данных, которую требуется определенным образом обработать. Более того, программист хочет иметь одновременно всю совокупность данных в памяти. В этой главе будет показано, как создаются и обрабатываются массивы в программах на языке ассемблера для ЭВМ семейства VAX. Кроме того, будет описан полный набор режимов адресации, который имеют ЭВМ семейства VAX.

Одним из примеров задач с использованием массивов является сортировка чисел в возрастающем порядке. Поскольку действительный порядок вводимых чисел предположительно неизвестен, нельзя распечатать ни одно число, пока не будут введены все числа. Кроме того, последнее введенное число может оказаться наименьшим и, следовательно, должно быть напечатано первым. На рис. 7.1, *а* и *б* приводятся примеры такой программы,

(а) Паскаль

```
PROGRAM SORT(INPUT,OUTPUT);
VAR J,K,L,M : INTEGER;
    LIST : ARRAY[1..20] OF INTEGER;
BEGIN
(* ВВОД НЕУПОРЯДОЧЕННЫХ ЧИСЕЛ *)
    FOR J := 1 TO 20 DO READLN (LIST[J]);
(* СОРТИРОВКА ЧИСЕЛ *)
    FOR L := 1 TO 19 DO BEGIN
        FOR K := L+1 TO 20 DO BEGIN
            IF (LIST[L] > LIST[K]) THEN BEGIN
                M := LIST[L];
                LIST[L] := LIST[K];
                LIST[K] := M;
            END;
        END;
    END;
(* ПЕЧАТЬ УПОРЯДОЧЕННЫХ ЧИСЕЛ *)
    FOR J := 1 TO 20 DO WRITELN (LIST[J]);
END.
```

(б) Фортран

```
      PROGRAM SORT
      INTEGER LIST(1:20), J, K, L, M
C ВВОД НЕУПОРЯДОЧЕННЫХ ЧИСЕЛ
      READ *, (LIST(J), J=1,20)
C СОРТИРОВКА ЧИСЕЛ
      DO 20 L=1,19
        DO 10 K=L+1,20
          IF(LIST(L).GT.LIST(K)) THEN
            M=LIST(L)
            LIST(L)=LIST(K)
            LIST(K)=M
          END IF
10      CONTINUE
20    CONTINUE
C ПЕЧАТЬ УПОРЯДОЧЕННЫХ ЧИСЕЛ
      PRINT *, (LIST(J), J=1,20)
      STOP
      END
```

Рис. 7.1. Программа сортировки

написанной на Паскале и Фортране. В программе осуществляется ввод списка из 20 чисел, выполняется сортировка этих чисел по возрастанию и в конце упорядоченный список выводится на печать. Обратите внимание, что программа разделяется на три части: ввод чисел, сортировка и печать. Эти три части программы выделены комментариями. Используемый в программе метод сортировки представляет собой один из вариантов известного алгоритма сортировки выбором. Если читатель еще не знаком с методом сортировки, ему предлагается в качестве упражнения подробно разобрать эти программы.

Первый оператор в обоих вариантах программы – это невыполняемый оператор, сообщающий компилятору, сколько ячеек памяти должно быть выделено для массива LIST.

Как было показано ранее, в программе на языке ассемблера необходимо зарезервировать ячейки памяти, предназначенные для данных, что обычно делается с помощью директивы ассемблера .BLKL1. В следующем разделе будет рассмотрено выделение памяти под массивы.

Обратите внимание, что в программах на языках Фортран и Паскаль в различных операциях основной части программы идентификатор переменной LIST используется вместе с *индексом*, который указывает на определенный элемент массива. Это требуется потому, что обращение все время осуществляется к отдельному элементу массива, а не ко всему массиву. Это справедливо и для языка ассемблера. Хотя некоторые вычислительные машины, в частности и ЭВМ семейства VAX, имеют инструкции, позволяющие выполнить пересылку всего массива целиком, большинство операций нижнего уровня работают в один момент времени только с одной ячейкой памяти. Поэтому необходимы средства для указания отдельного элемента массива. Об этом и будет рассказано в следующем разделе.

7.2. МАССИВЫ В ПРОГРАММАХ НА ЯЗЫКЕ АССЕМБЛЕРА

РАСПРЕДЕЛЕНИЕ ПАМЯТИ

Как было сказано выше, при работе с массивами необходимы некоторые средства для резервирования ряда ячеек памяти. Хотя это можно сделать по-разному, проще всего использовать блок последовательно расположенных ячеек памяти. Ячейки в блоке соответствуют последовательно расположенным элементам массива.

В языке ассемблера VAX для резервирования блока памяти используются директивы .BLKL, .BLKW или .BLKB. Ранее в поле операндов этих директив указывалось значение 1, которое показывало, что резервируется одно длинное слово, слово или байт. Для массива, чтобы зарезервировать больший блок памяти, в директиве может быть указано большее число. Тогда ассемблер в этом месте программы зарезервирует пространство для заданного числа элементов данных. Например, по директиве .BLKL 10 было бы выделено десять длинных слов, размещенных одно после другого (см. рис. 7.2).

Для того чтобы можно было обращаться к массиву, массив должен иметь символическое имя. Для этого перед директивой .BLKL помещается метка XYZ. Как описывалось в гл. 4, это символическое имя (символический адрес) XYZ вводится в таблицу имен, определяемых пользователем, вместе с текущим значением счетчика адресов. В результа-

```
XYZ:    .BLKL   10

ПО ДАННОЙ ДИРЕКТИВЕ
РЕЗЕРВИРУЕТСЯ ПАМЯТЬ
```

эквивалентно

```
XYZ:    .BLKL   1       ;XYZ(1)
        .BLKL   1       ;XYZ(2)
        .BLKL   1       ;XYZ(3)
        .BLKL   1       ;XYZ(4)
        .BLKL   1       ;XYZ(5)
        .BLKL   1       ;XYZ(6)
        .BLKL   1       ;XYZ(7)
        .BLKL   1       ;XYZ(8)
        .BLKL   1       ;XYZ(9)
        .BLKL   1       ;XYZ(10)
```

Рис. 7.2. Директива .BLKL

те символический адрес XYZ является адресом первого длинного слова массива. Директива ассемблера .BLKL 20 вызывает 20-кратное прибавление к содержимому счетчика адресов числа 4, т.е. увеличивает его на 80.

АДРЕСНЫЕ ВЫРАЖЕНИЯ

После того, как выделена память под массив, возникает проблема адресации определенного элемента этого массива. С первым элементом массива все просто, так как имя массива указывает адрес первого элемента массива. Поэтому первый элемент массива XYZ можно было бы очистить, выполняя инструкцию CLRL XYZ. Однако это не решает проблему доступа к другим элементам массива, таким как второй, третий и т. д.

Одним из решений этой проблемы является использование *адресных выражений*. Поскольку длинное слово занимает 4 байта, адрес второго элемента массива XYZ на четыре больше адреса XYZ. Аналогично третий и четвертый элементы имеют адреса на 8 и 12 больше, чем адрес XYZ. Следовательно, можно ссылаться на эти адреса, используя символические выражения XYZ+4, XYZ+8 и XYZ+12.

Очистить содержимое первых четырех элементов массива XYZ можно с помощью следующих инструкций:

```
CLRL    XYZ
CLRL    XYZ+4
CLRL    XYZ+8
CLRL    XYZ+12
```

Обратите внимание, что в выражениях языка ассемблера символическое имя XYZ всегда означает адрес XYZ и никогда – содержимое ячейки с этим адресом, что очень важно для понимания инструкций типа MOVL XYZ+4, A. Внешне это похоже на операторы языков Паскаль или Фортран A:=XYZ+4; или A=XYZ+4. Однако по сути это разные вещи, так как в данном случае не подразумевается сложение содержимого XYZ с числом 4, а нужен адрес ячейки, который на 4 больше адреса XYZ. Поэтому здесь аналогия может быть проведена с оператором языков Паскаль и Фортран A:=XYZ[2]; или A=XYZ(2).

Заметим, что адресные выражения могут использоваться в языке ассемблера везде, где может быть символическое имя или число. Выражение может быть довольно сложным, но в результате его вычисления должно получаться осмысленное значение – адрес или числовая константа. Подробнее это будет описано в гл. 10. Пока ограничимся наиболее часто встречающимися выражениями – адрес плюс-минус одно или более чисел или символических имен, представляющих собой числовые константы.

Использование адресных выражений очень важно при работе с массивами, но существует серьезное ограничение. Эти выражения вычисляются во время ассемблирования и поэтому не могут включать какие-либо числа, которые могут быть изменены во время выполнения программы. Это означает, что хотя с их помощью можно обращаться к определенным элементам массива, таким как XYZ[1], XYZ[2] и т. д., но нельзя обратиться к неизвестному заранее (произвольному) элементу массива, такому как XYZ[J]. Следовательно, хотя можно было бы очистить массив, выполнив последовательность инструкций CLRL, но пока еще не рассматривались средства, чтобы сделать это в цикле.

РЕГИСТРОВЫЙ КОСВЕННЫЙ РЕЖИМ

Необходимо иметь средства, по действию аналогичные адресным выражениям, позволяющие вычислять адрес во время выполнения программы. Для этого большинство современных ЭВМ имеют специальные режимы адресации. Адрес, который используется для выборки или занесения информации в память, называется *исполнительным адресом*. Например, в режиме относительной адресации для получения исполнительного адреса значение смещения складывается с содержимым программного счетчика. Архитектура ЭВМ VAX позволяет вычислять исполнительный адрес с помощью разных режимов адресации.


```
XYZ:    .BLKL   50
        . . .                   ;МАССИВ ЭЛЕМЕНТОВ

CLEAR:  MOVAL   XYZ,R0
        CLRL    R1              ;R0 - РЕГИСТР УКАЗАТЕЛЬ АДРЕСА
10$:    CLRL    (R0)            ;R1 - СЧЕТЧИК ЦИКЛА
        ADDL2   #4,R0           ;ОЧИСТИТЬ ЭЛЕМЕНТ МАССИВА
        AOBLSS  #50,R1,10$      ;УВЕЛИЧИТЬ УКАЗАТЕЛЬ
                                ;ПОВТОРЯТЬ В ЦИКЛЕ, ПОКА НЕ БУДУТ
                                ;ОЧИЩЕНЫ 50 ЭЛЕМЕНТОВ МАССИВА
```

Рис. 7.3. Режим регистровой косвенной адресации

Одним из простейших является *регистровый косвенный* режим. Для применения такого режима адресации значение адреса вычисляется во время выполнения программы и помещается в регистр. Затем в случае регистровой косвенной адресации значение, находящееся в регистре, используется как адрес операнда, т.е. исполнительный адрес.

В ЭВМ VAX 15 из 16 регистров общего назначения (а именно R0 – R11, AP, FP и SP) могут использоваться для регистровой косвенной адресации. (Регистр 15 – программный счетчик – нельзя применять в этом режиме.) Для обозначения режима регистровой косвенной адресации в языке ассемблера операнд в инструкции записывается как имя регистра, заключенное в круглые скобки. Например, в инструкции CLRL (R3) задана косвенная адресация с использованием регистра 3. Исполнительным адресом является значение, содержащееся в этом регистре общего назначения. Если регистр 3 содержит адрес ^X00001000, то будет очищено содержимое длинного слова, начинающееся с адреса ^X1000.

Рассмотрим для примера действие инструкции CLRL (R3) на уровне машинного языка. Инструкция после ее трансляции в машинный код будет занимать только 2 байта – один для кода операции и один для спецификатора операнда. В машинном коде инструкция будет выглядеть следующим образом:

63 D4

Инструкция CLRL имеет код операции ^XD4. Байт ^X63 обозначает косвенную адресацию с использованием регистра R3. (Код ^X6 задает регистровый косвенный режим, а ^X3 – регистр 3.)

Теперь, прежде чем описать действие этой инструкции, необходимо знать содержимое регистра R3. Предположим, что в момент выполнения этой инструкции регистр R3 содержит ^X00001E84. Поскольку исполнительный адрес равен ^X00001E84, инструкция CLRL очищает (сбрасывает в 0) содержимое длинного слова с адресом ^X1E84. Надо отметить, что содержимое регистра R3 не изменяется при выполнении инструкции CLRL и после выполнения этой инструкции в регистре R3 остается значение ^X00001E84.

На рис. 7.3 приведен фрагмент программы, в котором очищается массив из 50 длинных слов. Обратите внимание:

1. В регистр R0 загружается начальный адрес массива XYZ с помощью новой инструкции MOVAL (переслать адрес длинного слова).

2. При каждом проходе цикла к содержимому регистра R0 прибавляется число 4, полученный адрес указывает на следующее длинное слово массива.

3. Еще один регистр, R1, используется для счета числа повторений. Это может показаться избыточным, и в самом деле, как будет показано, второй регистр не понадобится, когда будут рассмотрены дополнительные режимы адресации.

СЕМЕЙСТВО ИНСТРУКЦИЙ MOVAL

Инструкция MOVAL (см. программу на рис. 7.3) – это только один представитель семейства инструкций, разработанных специально для загрузки адресов, используемых впоследствии в режимах адресации, таких как регистровая косвенная. Инструкция MOVAL отличается от инструкции MOVL тем, что MOVL пересылает *содержимое* длинного слова по

указанному месту назначения, а MOVAL пересылает *адрес* длинного слова. Инструкция MOVAL – это один член семейства инструкций *пересылки адреса*, которое включает в себя:

Мнемоника	Значение
MOVAB	Переслать адрес байта
MOVAW	Переслать адрес слова
MOVAL	Переслать адрес длинного слова
MOVAQ	Переслать адрес 64-битового квадраслова
MOVAO	Переслать адрес 128-битового октаслова

Поскольку адрес на ЭВМ VAX представляет собой 32-битовое двоичное число, любая из этих пяти инструкций выполняет пересылку 32-битового значения по адресу второго операнда, имеющего формат длинного слова. Может показаться странным, зачем нужно пять таких инструкций, если адрес всегда остается адресом, независимо от того, указывает он на байт, слово, длинное слово и т. д.

Фактически программа на рис. 7.3 будет работать так же, если инструкцию MOVAL заменить на инструкцию MOVAB. Однако, когда позднее в этой главе будут рассмотрены более сложные режимы адресации, станет ясно, зачем нужно столько инструкций.

Чтобы понять смысл инструкций пересылки адреса, необходимо рассмотреть, что произойдет при выполнении следующих трех инструкций:

```
1. MOVL    A, R0
2. MOVL    #A, R0
3. MOVAL   A, R0
```

Предположим, что символическое имя A определено в ассемблерной программе следующим образом и при ассемблировании символическому адресу A был назначен адрес ^X00000444:

```
A:      .LONG   ^X00112233
```

Если компоновщик настроит программный модуль, содержащий имя A, так, что адресом начала этого модуля станет ^X0200, то во время выполнения символическому имени A будет соответствовать адрес ^X00000444 плюс ^X0200 или ^00000644 (см. о перемещении адресов в гл. 4). В машинном коде длинное слово с адресом A могло бы выглядеть следующим образом:

Содержимое	Адрес
00112233	00000644

По первой инструкции, MOVL A, R0, будет выполняться пересылка содержимого длинного слова A в регистр R0, так что в результате в регистре R0 будет содержаться значение ^X00112233. Поскольку эта инструкция использует одну из форм относительной адресации для операнда A, то при перемещении программы на адрес ^X0200 не требуется никакой модификации программы. (Если адрес A и инструкция MOVL A, R0 находятся в одном модуле, то расстояние между ними не меняется при перемещении модуля.)

Во второй инструкции MOVL #A,R0 используется непосредственная адресация для "константы" A, которой соответствует в таблице имен значение ^X00000444. Поэтому длинное слово ^X00000444 становится частью инструкции. Когда компоновщик будет настраивать программу на адрес ^X0200, то он прибавит ^X0200 к значению длинного слова, содержащемуся в инструкции, так что оно станет равно ^X00000644. В результате эта инструкция будет пересылать значение ^X00000644 (адрес A во время выполнения) в регистр R0.

По третьей инструкции, MOVAL A,R0, также пересылается значение ^X00000644 в регистр R0. Однако эта инструкция использует одну из форм относительной адресации для вычисления адреса A, поэтому при настройке программы эту инструкцию не надо модифицировать.

На многих ЭВМ, таких как ЭВМ семейства PDP-11, инструкции, подобные MOVL #A,R0, используются для получения адресов внутри модуля. Все такие адреса должны модифицироваться, если программа перемещается. В отличие от этого архитектура ЭВМ VAX ориентирована на то, чтобы упростить разработку *позиционно независимых* программ, которые могут без всякой модификации размещаться в любом месте памяти. Именно для этого в набор инструкций VAX включена группа инструкций MOVAL. Поэтому считается, что использование непосредственной адресации с символическими адресами, как в инструкции MOVL #A,R0, является плохим стилем программирования.

ПРОГРАММА СОРТИРОВКИ

В качестве последнего примера этого раздела перепишем программу рис. 7.1 на языке ассемблера (рис. 7.4). Этот пример не включает ввод и вывод, а содержит только программу сортировки. В программе на языке ассемблера регистры R3 и R4 используются как счетчики циклов L и K соответственно. Однако в программе на языке высокого уровня переменная L принимает значения 1–19, а содержимое регистра R3 в ассемблированной программе изменяется от 0 до 18. То же замечание справедливо для K и R4. В ассемблерной программе регистр R2, подобно переменной M в программе на языке высокого уровня, служит как временная память для перестановки местами двух элементов. Для выборки элементов массива применяется регистровая косвенная адресация с использованием регистров R0 и R1. Обратите внимание, что для загрузки адреса LIST в регистр R0 используется инструкция MOVAL.

```
SIZE=20
LIST:   .BLKL   SIZE            ;ПАМЯТЬ ВЫДЕЛЕННАЯ ДЛЯ МАССИВА
        .
        .
        .
SORT:   MOVAL   LIST,R0         ;R0 - УКАЗАТЕЛЬ НА ЛЕВЫЙ ЭЛЕМЕНТ МАССИВА
        CLRL    R3              ;R3 - СЧЕТЧИК ВНЕШНЕГО ЦИКЛА, L
10$:    MOVL    R0,R1           ;ЗНАЧЕНИЕ ПРАВОГО УКАЗАТЕЛЯ В R1 РАВНО
        ADDL2   #4,R1           ;ЗНАЧЕНИЮ ЛЕВОГО УКАЗАТЕЛЯ  + 4
        MOVL    R3,R4           ;R4 - СЧЕТЧИК ВНУТРЕННЕГО ЦИКЛА, K
        INCL    R4              ;ЕГО НАЧАЛЬНОЕ ЗНАЧЕНИЕ L+1
20$:    CMPL    (R0),(R1)       ;СРАВНИТЬ ЛЕВЫЙ И ПРАВЫЙ ЭЛЕМЕНТЫ МАССИВА
        BLEQ    30$             ;ЕСЛИ МЕНЬШЕ ИЛИ РАВНО, ПРОДОЛЖИТЬ ЦИКЛ
        MOVL    (R0),R2         ;ИНАЧЕ ПОМЕНЯТЬ МЕСТАМИ
        MOVL    (R1),(R0)       ;ДВА ЭЛЕМЕНТА МАССИВА LIST,
        MOVL    R2,(R1)         ;ИСПОЛЬЗУЯ R2 КАК ВРЕМЕННУЮ ПАМЯТЬ, M
30$:    ADDL2   #4,R1           ;УВЕЛИЧИТЬ ПРАВЫЙ УКАЗАТЕЛЬ
        AOBLSS  #SIZE,R4,20$    ;ПОВТОРЯТЬ В ЦИКЛЕ ДО ЗАВЕРШЕНИЯ
        ADDL    #4,R0           ;УВЕЛИЧИТЬ ЛЕВЫЙ УКАЗАТЕЛЬ
        AOBLSS  #SIZE-1,R3,10$  ;ПОВТОРЯТЬ В ЦИКЛЕ ДО ЗАВЕРШЕНИЯ
```

Рис. 7.4. Программа сортировки на языке ассемблера

УПРАЖНЕНИЯ 7.1

1. Пусть регистры R0, R1 и R2 содержат следующие значения, представленные в шестнадцатеричном виде:

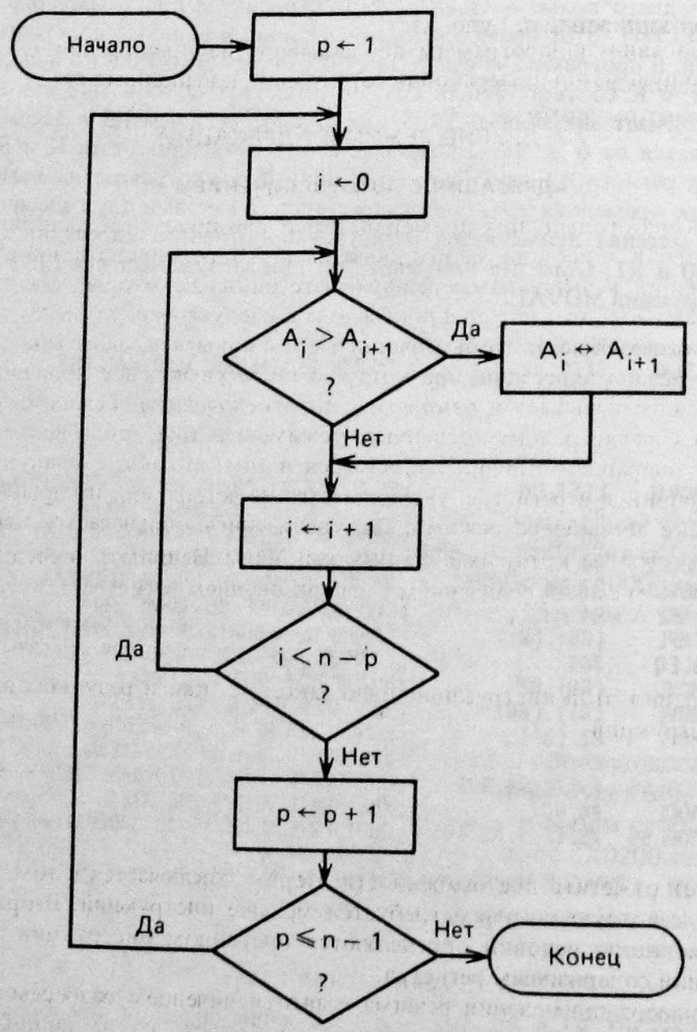
```
R0      ^X00001111
R1      ^X00000E24
R2      ^X0000FFC4
```

а символические имена ABC, PQW и XYZ соответствуют следующим адресам:

```
ABC     ^X00001000
PQW     ^X0000C4E0
XYZ     ^XFFFFFFF8
```

Какими будут исполнительные адреса в следующих (взятых по отдельности) инструкциях, т.е. какие ячейки будут очищены при их выполнении?

```
а.  CLRL    PQW          б.  CLRL    PQW+24
в.  CLRL    ABC+^XE4     г.  CLRL    ABC+300
д.  CLRL    (R0)         е.  CLRL    XYZ+1046
ж.  CLRL    (R1)         з.  CLRL    XYZ+^X300
и.  CLRL    (R2)         к.  CLRL    XYZ+100
```



* N – число элементов, подлежащих сортировке
* A – массив данных, подлежащих сортировке

Структурная схема алгоритма пузырьковой сортировки

2. Преобразуйте каждую из инструкций п. 1 упр. 7.1 в машинный код, считая, что инструкция расположена, начиная с адреса ^X00000346.

3. Преобразуйте фрагмент программы рис. 7.4 в полностью законченную программу сортировки. Используйте подпрограммы RNUM и PNUM из приложения Б для организации ввода и вывода. Выполните программу с произвольными исходными данными.

4. Блок-схема на стр. 138 описывает алгоритм сортировки, который называют методом пузырьковой сортировки.

   а. Как работает эта программа сортировки? Приведите пример с четырьмя или пятью элементами.

   б. Чем отличается эта программа от программы сортировки рис. 7.4?

   в. Какой из методов более эффективен и в чем их различие?

   г. Какие преимущества и недостатки можно выделить при сравнении этих двух методов?

5. Напишите программу на языке ассемблера (аналогичную рис. 7.4), реализующую алгоритм пузырьковой сортировки п. 1 упр. 7.1.

6. Напишите (добавив подпрограммы ввода-вывода из приложения Б) полную программу, использующую метод пузырьковой сортировки, и выполните ее.

7.3. ДРУГИЕ РЕЖИМЫ АДРЕСАЦИИ

АДРЕСАЦИЯ С АВТОУВЕЛИЧЕНИЕМ

Как было видно из предыдущих примеров, когда при просмотре массива применялась регистровая косвенная адресация, необходимо было корректировать значение указателя в регистре, прибавляя к нему на очередном шаге цикла некоторое число. Если массив состоит из длинных слов, то следует для перехода к следующему элементу массива увеличивать значение указателя на 4. Чтобы облегчить это и повысить эффективность программ, ЭВМ VAX имеют режим адресации, при котором такое увеличение производится автоматически. Такой режим называется *режимом с автоувеличением*. Режим с автоувеличением весьма сходен с регистровым косвенным режимом в том, что в некотором регистре содержится адрес операнда. Отличие заключается в том, что после выполнения инструкции адрес, содержащийся в регистре, увеличивается на четыре или на число байтов в длинном слове. В языке ассемблера режим с автоувеличением обозначается как заключенное в скобки имя регистра, за которыми следует знак плюс. Например, в следующей инструкции CLRL задан режим с автоувеличением с использованием регистра 0.

```
CLRL    (R0)+
```

Результат выполнения этой инструкции точно такой же, как и результат выполнения следующей пары инструкций:

```
CLRL    (R0)
ADDL2   #4,R0
```

Однако следует отметить две особенности. Первая заключается в том, что при использовании режима с автоувеличением требуется меньше инструкций. Вторая – в том, что при этом режиме коды условий определяются операндом инструкции, а не операцией автоувеличения или содержимым регистра.

В качестве примера применения режима с автоувеличением разберем программу, выполняющую суммирование элементов массива, состоящего из 20 длинных слов. На рис. 7.5 показано два варианта реализации этой программы – с регистровым косвенным режимом и режимом с автоувеличением. Как уже говорилось, отличие между этими двумя программами состоит в том, что при использовании режима с автоувеличением не требуется инструкция ADDL2 #4,R0. Это делает программу короче, яснее, более простой

(а) Вариант с регистровой, косвенной адресацией

```
SIZE=20
SUM:    .BLKL   1
DATA:   .BLKL   SIZE
        . . .

SUMV1:  MOVAL   DATA,R0
        CLRL    R1
        CLRL    SUM
10$:    ADDL2   (R0),SUM
        ADDL2   #4,R0
        AOBLSS  #SIZE,R1,10$
```

(б) Вариант с использованием режима с автоувеличением

```
SIZE=20
SUM:    .BLKL   1
DATA:   .BLKL   SIZE
        . . .

SUMV2:  MOVAL   DATA,R0
        CLRL    R1
        CLRL    SUM
10$:    ADDL2   (R0)+,SUM

        AOBLSS  #SIZE,R1,10$
```

Рис. 7.5. Две программы суммирования 20 чисел

в написании и значительно эффективнее. (Обратите внимание, что уменьшается число инструкций в цикле.)

При выполнении инструкции ADDL2 (R0)+,SUM адрес, содержащийся в регистре R0, увеличивается на 4, поскольку длинное слово занимает 4 байта. Чтобы использовать режим с автоувеличением для работы с массивами данных любого типа, значение увеличения определяется кодом операции. Например, в инструкции ADDB2 (R0)+,SUM адрес в регистре R0 увеличивается на 1, а в инструкции ADDW2 (R0)+,SUM адрес в регистре R0 увеличивается на 2. Аналогично в инструкциях, работающих с квадрасловами или октасловами, адрес в регистре увеличивается на восемь или шестнадцать соответственно.

Этим, в частности, объясняется, почему имеется несколько инструкций MOVA. Инструкции семейства MOVA могут использоваться с любым режимом адресации. Таким образом, допустима инструкция MOVAL (R0)+,R3. По этой инструкции в регистр R3 будет занесен *адрес*, содержащийся в регистре R0 (а не содержимое длинного слова, расположенного по этому адресу). Предположим, например, что до выполнения этой инструкции в регистре R0 содержится адрес A некоторого длинного слова. После выполнения инструкции MOVAL (R0)+,R3, в регистр R3 будет помещен адрес A указанного длинного слова, а в регистр R0 – адрес A+4 следующего длинного слова. Чтобы обеспечить выполнение таких адресных вычислений в соответствии с различными форматами данных в инструкциях семейства MOVA, при автоувеличении используются следующие приращения:

Инструкция		Приращение R0
MOVAB	(R0)+,R3	1
MOVAW	(R0)+,R3	2
MOVAL	(R0)+,R3	4
MOVAQ	(R0)+,R3	8
MOVAO	(R0)+,R3	16

Может показаться, что в инструкциях семейства MOVA режим с автоувеличением будет применяться редко. Однако одним из примеров такого применения является генерация списков адресов для подпрограмм. Это будет рассмотрено в гл. 9.

АДРЕСАЦИЯ С АВТОУМЕНЬШЕНИЕМ

Часто бывает необходимо получать доступ к элементам массива в порядке, обратном их расположению. Для этого в архитектуре ЭВМ семейства VAX предусмотрен еще один режим адресации, который называется *режимом с автоуменьшением*. Режим с автоуменьшением по сути то же самое, что и режим с автоувеличением: адрес содержащийся в регистре общего назначения уменьшается на четыре, два или единицу при адресации длинных слов, слов или байтов соответственно. Другое отличие вытекает из необходимости симметрии между этими двумя режимами. В режиме с автоувеличением адрес в регистре общего назначения увеличивается *после* его использования. В режиме с автоуменьшением адрес в регистре уменьшается *перед* его использованием. Это означает, что инструкция с автоуменьшением выполняет противоположную адресную операцию по отношению к инструкции с автоувеличением. Данное свойство очень важно для реализации *стека*. Как уже говорилось в гл. 5, аналогией стека является стопка тарелок, в которой удаляемая тарелка – это самая последняя из добавленных в стопку. (Если тарелки добавляются и убираются с верхушки стопки, то верхняя тарелка – всегда последняя из добавленных.) Как будет показано в гл. 9, существующая симметрия между режимами с автоувеличением и автоуменьшением позволяет программистам легко реализовать стек.

В языке ассемблера режим с автоуменьшением обозначается следующим образом:

```
CLRL    -(R3)
```

При выполнении инструкции значение в регистре R3 уменьшается на четыре, а затем ячейка, адрес которой содержится в R3, очищается. Обратите внимание на то, что знак "минус" должен стоять перед операндом, в отличие от режима с автоувеличением, в котором знак "плюс" стоит после операнда. Это должно напоминать программисту, что уменьшение адреса производится до его использования, а увеличение – после. Действие предыдущей инструкции дает тот же результат, что и две следующие инструкции:

```
SUBL2   #4,R3
CLRL    (R3)
```

В качестве примера применения режима с автоуменьшением рассмотрим программу на рис. 7.6, в которой осуществляется пересылка массива A, состоящего из 20 длинных слов, в массив B с изменением порядка следования элементов (длинных слов) на обратный. Обратите внимание, что в инструкции MOVAL #B+BYTES, R1 в регистр R1 загружается адрес длинного слова, следующего за последним длинным словом массива B. (Если 20 длинных слов массива имеют адреса ^X0400, ^X0404, ^X0408, . . . , ^X044C, то адрес, загруженный в регистр R1, равен ^X0400 плюс константа BYTES. Поскольку BYTES равна 20, умноженному на 4, т.е. 80 десятичное или ^X50, то в регистр R1 загружается адрес ^X0400 плюс ^X50 или ^X0450. Однако поскольку в инструкции MOVL используется режим с автоуменьшением, то значение в регистре R1 уменьшается до ^X00000450 минус ^X4 или ^X0000044C перед тем, как будет осуществлена первая пересылка элемента в массив B.

```
SIZE=20                         ;МАССИВ СОДЕРЖИТ 20 ДЛИННЫХ СЛОВ
BYTES=SIZE*4                    ;КОТОРЫЕ ЗАНИМАЮТ 80 БАЙТОВ
A:      .BLKL   SIZE
B:      .BLKL   SIZE
        . . .
REVERSE:
        MOVAL   A,R0            ;УКАЗАТЕЛЬ НАЧАЛА МАССИВА A
        MOVAL   B+BYTES,R1      ;УКАЗАТЕЛЬ КОНЦА МАССИВА B
        CLRL    R2              ;СЧЕТЧИК ЦИКЛА (ОТ 0 ДО SIZE-1)
10$:    MOVL    (R0)+,-(R1)
        AOBLSS  #SIZE,R2,10$
```

Рис. 7.6. Программа переупорядочения элементов массива

На рис. 7.7 приведен пример программы, в которой каждый элемент массива (кроме последнего) устанавливается равным следующему по порядку элементу. Эта операция соответствует удалению из массива первого элемента.

На рис. 7.8,*а* показано, как эта программа может быть реализована на языке ассемблера с применением адресации в режиме с автоуменьшением. Обратите внимание, что используется три регистра. Регистр R0 содержит адрес элемента A[I], регистр R1 содержит адрес элемента A[I+1], а регистр R2 используется как счетчик цикла, значение которого изменяется от нуля до VSIZE – 1. Отметим, что VSIZE – адрес длинного слова – является переменным значением, а не константой, так что для завершения цикла используется инструкция AOBLSS VSIZE,R2,10$, а не инструкция AOBLSS #VSIZE,R2,10$. Поскольку значения в регистрах R0 и R1 увеличиваются согласованно, с разницей на четыре, то эта разница сохраняется на каждом шаге цикла.

Чтобы избежать использования двух или большего числа регистров в таких случаях, ЭВМ семейства VAX имеют набор *режимов адресации со смещением*. В режимах адресации со смещением учитывается фиксированное смещение относительно адреса, содержащегося в регистре. В языке ассемблера операнд, адресуемый в таком режиме, обозначается как некоторое число (или символическое имя, определяющее число), за которым следует имя регистра, заключенное в скобки. Например, по инструкции CLRL 16(R4) выполняется очистка содержимого длинного слова, смещенного на 16 байтов относительно адреса, содержащегося в R4. Если регистр R4 содержит ^X000003A4, то будет очищено содержимое длинного слова по адресу ^X000003B4.

На рис. 7.8,*б* показано, как ассемблерная программа рис. 7.8,*а* может быть написана с использованием режима адресации со смещением, что позволяет уменьшать число используемых регистров. Одним из источников смущения в такой программе является инструкция MOVL 4(R0), (R0)+, в которой используется регистр R0 в двух операндах (один из которых адресуется в режиме с автоувеличением). Смущение вызывается тогда, когда точно происходит увеличение содержимого регистра R0. Основное правило состоит в том, что операнды инструкции обрабатываются один за другим в том порядке, в каком они указаны в ней. Следовательно, хотя автоувеличение выполняется для второго операнда, оно не влияет на вычисление исполнительного адреса первого операнда.

Паскаль

```
FOR I := 1, VSIZE-1
BEGIN
 A[I] := A[I+1];
END;
```

Фортран

```
      DO I=1, VSIZE-1
        A(I)=A(I+1)
10    CONTINUE
```

Рис. 7.7. Программа пересылки данных в массиве

(а) Вариант с использованием режима с автоувеличением

```
      MOVAL   A, R0
      MOVAL   A+4, R1
      CLRL    R2
      SUBL3   #1, VSIZE, R3
10$:  MOVL    (R1)+, (R0)+
      AOBLSS  R3, R2, 10$
```

(б) Вариант с использованием режима со смещением

```
      MOVAL   A, R0

      CLRL    R2
      SUBL3   #1, VSIZE, R3
10$:  MOVL    4(R0), (R0)+
      AOBLSS  R3, R2, 10$
```

Рис. 7.8. Использование режима адресации со смещением

Следующий пример представляет несколько причудливый вариант программы рис. 7.8,*б*, в котором для адресации первого операнда используется режим с автоувеличением:

```
      MOVAL   A+4, R0
      CLRL    R2
      SUBL3   #1, VSIZE, R3
10$:  MOVL    (R0)+, -8(R0)
      AOBLSS  R3, R2, 10$
```

Обратите внимание, что для второго операнда должен указываться адрес, на четыре меньший чем адрес первого операнда. Так как содержимое регистра R0 будет уже увеличено на четыре, смещение адреса для второго операнда должно быть –8. Этот пример показывает, что смещение представляет собой число в дополнительном коде и может быть отрицательным.

Для экономии памяти предусмотрены три режима адресации со смещением. В первом смещение задается в формате байта, в диапазоне значений –128...+127. Во втором – смещение задается в формате слова в диапазоне –32768...+32767. В третьем режиме смещение указывается в формате длинного слова и может адресовать любую ячейку памяти.

Если значение смещения достаточно мало и может поместиться в формате байта (т.е. находиться в диапазоне –128...+127), то ассемблер для указания такого смещения выберет формат байта. Если значение смещения выходит за границы этого диапазона, то для задания смещения будет выбран формат слова или длинного слова. Если неизвестно точно значение смещения, ассемблер будет использовать для задания смещения формат длинного слова. (Это может произойти, если символическое имя определяется в программе позже.) Можно отменить автоматический выбор формата смещения, если перед полем адреса поместить служебный символ, а именно

B^	Смещение, заданное в формате байта
W^	Смещение, заданное в формате слова
L^	Смещение, заданное в формате длинного слова

Например, инструкция

```
      CLRL    W^X(R3)
```

будет ассемблироваться со смещением, заданным в формате слова, независимо от того, помещается ли значение X в формате слова. Обычно это приводит к ошибкам, если формат для смещения выбран неправильно. Поэтому широкое использование приведенных выше служебных символов не рекомендуется.

В последнем примере на стр. 000 адрес A+4 помещается в регистр R0 с помощью инструкции MOVAL A+4,R0. Это сделано для того, чтобы избежать искушения объединить два режима: режим со смещением и режим с автоувеличением – в одном спецификаторе операнда типа 4(R0)+. В ЭВМ семейства VAX такого режима адресации нет и операнд, подобный 4(R0)+ вызовет ошибку ассемблирования. Такие выражения, как A+4, и режимы типа –8(R0) могут показаться сложными; на самом деле в последней приведенной программе выполняется то же самое, что и в программе рис. 7.8,*б*. Отметим, что выражения типа A+4 вычисляются в процессе ассемблирования и не требуют времени при выполнении программы.

ИНДЕКСНЫЙ РЕЖИМ АДРЕСАЦИИ

Хотя режимы со смещением, регистровый, косвенный, с автоувеличением и автоуменьшением обеспечивают богатый выбор способов обработки массивов, каждый из них имеет

свои недостатки. Регистровый косвенный режим, режимы с автоувеличением и автоуменьшением удобны для последовательной выборки элементов одного массива. Однако эти режимы неудобны для произвольной выборки элементов массива и еще более неудобны для связанной обработки нескольких массивов. Например, при реализации на языке ассемблера с использованием регистрового косвенного режима адресации следующих аналогичных программ на языках Паскаль и Фортран:

```
Паскаль                                  Фортран

I := ПРОИЗВОЛЬНОЕ ЗНАЧЕНИЕ;              I = ПРОИЗВОЛЬНОЕ ЗНАЧЕНИЕ
. . .                                    . . .
A[I] := B[I] + C[I];                     A(I) = B(I) + C(I)
```

потребовалось бы отдельно вычислить адреса A(I), B(I), C(I) и поместить их в три разных регистра, чтобы можно было выполнить инструкцию

```
ADDL3   (R2),(R3),(R1)
```

Заметим, что значение I должно быть умножено на четыре, если A, B и C описаны как массивы типа INTEGER. В результате эквивалентная ассемблерная программа будет иметь следующий вид:

```
MULL3   I,#4,R4          ;СМЕЩЕНИЕ (БАЙТ) ПОМЕСТИТЬ В R4
MOVAL   A-4,R1           ;АДРЕС A[I] ПОМЕСТИТЬ В R1
ADDL2   R4,R1
MOVAL   B-4,R2           ;АДРЕС B[I] ПОМЕСТИТЬ В R2
ADDL2   R4,R2
MOVAL   C-4,R3           ;АДРЕС A[I] ПОМЕСТИТЬ В R3
ADDL2   R4,R3
ADDL3   (R2),(R3),(R1)   ;A[I] := B[I] + C[I]
```

В инструкциях MOVAL от значения символических адресов A, B, C отнимается четыре. Это следствие того, что и в Паскале, и в Фортране индекс младшего элемента массива обычно равен 1. Поэтому в программе на языке ассемблера символическое имя A определяет адрес элемента A[1] или A(1). Если индекс I равен 1, то по инструкции MULL3 вычисляется смещение I*4 или ^X00000004 и помещается в регистр R4, а затем с помощью трех инструкций ADDL2 формируются в регистрах R1, R2 и R3 адреса элементов в массивах A, B и C соответственно. Адрес A–4 называется *базовым адресом* массива A, так как для получения адреса конкретного элемента массива значение этого адреса складывается со значением смещения. Аналогично адреса B–4 и C–4 являются базовыми для массивов B и C.

Инициализация трех отдельных регистров требует больших затрат времени и памяти. Ясно, что режимы с автоувеличением и с автоуменьшением не помогут в решении этой проблемы, если на последующих шагах программы не будет выполняться последовательное сканирование массивов A, B и C. Эту задачу можно решить при использовании еще одного режима адресации, называемого *индексным*. Программу можно переписать следующим образом, применяя индексный режим адресации:

```
MOVL    I,R0
ADDL3   B-4[R0],C-4[R0],A-4[R0]
```


Обратите внимание, что форма записи для обозначения индексного режима похожа на запись, применяемую в языках высокого уровня. В инструкции ADDL3 обозначение каждого операнда состоит из базового адреса, такого как B–4 или C–4, за которым следует имя регистра, заключенное в квадратные скобки, например [R0].

Исполнительный адрес вычисляется так: к смещению, равному умноженному на четыре значению, содержащемуся в индексном регистре (в данном случае – в регистре R0), прибавляется значение базового адреса. Поэтому в программе нет необходимости умножать I на четыре. Обратите внимание, что содержимое регистра R0 умножается на четыре, потому что инструкция ADDL3 работает с длинными словами. Если индексный режим используется для адресации операндов, имеющих формат байтов или слов, происходит автоматическое умножение на один или два, а если операнд – квадраслово, то происходит умножение на восемь.

В отличие от остальных режимов адресации, индексный режим всегда используется в сочетании с одним из других режимов. Другие режимы используются для вычисления базового адреса, к которому прибавляется смещение, вычисляемое в зависимости от указываемого индекса. В этом случае используется одна из форм относительной адресации для получения базовых адресов A–4, B–4 и C–4. В результате инструкция ADDL3 становится позиционно независимой. Более того, если массивы A, B и C размещены в памяти достаточно близко к инструкции ADDL3, то ассемблер для A–4, B–4 и C–4 будет использовать относительную адресацию со смещением, заданным в формате байта или слова. В результате инструкция может занимать 10–13 байтов.

Как ранее говорилось, индексный режим используется в сочетании с другими режимами. Например, можно написать такую инструкцию, как

```
CLRL    (R2)+[R0]
```

При этом исполнительный адрес вычисляется так: значение в регистре R0, умноженное на четыре, складывается со значением адреса в регистре R2, после чего производится очистка ячейки, расположенной по вычисленному адресу. Затем адрес в регистре R2 увеличивается на четыре.

На рис. 7.9 показано, как может быть переписана программа рис. 7.5 с использованием индексного режима адресации. Как видно из рис. 7.9, нет особой разницы между двумя вариантами программы, за исключением того, что в программе, в которой используется индексный регистр, значение в регистре R0 увеличивается на единицу, а не на четыре. Однако последний вариант концептуально проще, поскольку регистр R0 содержит индекс, а не адрес.

(а) Вариант с регистровой косвенной адресацией

```
SIZE=20
SUM:    .BLKL   1
DATA:   .BLKL   SIZE
        . . .

ADDUP:  MOVAL   DATA,R0
        CLRL    R1
        CLRL    SUM
10$:    ADDL2   (R0),SUM
        ADDL2   #4,R0
        AOBLSS  #SIZE,R1,10$
```

(б) Вариант с использованием индексной адресации

```
SIZE=20
SUM:    .BLKL   1
DATA:   .BLKL   SIZE
        . . .

ADDUP:  MOVAL   DATA,R0
        CLRL    R1
        CLRL    SUM
10$:    ADDL2   DATA[R0],SUM
        INCL    R0
        AOBLSS  #SIZE,R1,10$
```

Рис. 7.9. Суммирование 20 чисел с использованием индексного режима адресации.

Другое преимущество индексного режима может быть обнаружено, если проследить за изменением содержимого регистров R0 и R1. В программе рис. 7.9,*а* регистр R0 содержит адрес, а регистр R1 используется как счетчик цикла. Во втором примере (рис. 7.9,*б*), регистр R0 содержит индекс, а регистр R1 опять используется как счетчик цикла. При детальном рассмотрении второго варианта программы выясняется, что регистры R0 и R1 содержат одинаковое значение. Содержимое обоих регистров сбрасывается в 0 перед началом цикла и увеличивается на 1 при каждом проходе цикла. Поскольку регистры содержат одно и то же значение, то один из них оказывается лишним и программа может быть переписана следующим образом:

```
ADDUP:  CLRL    R0
        CLRL    SUM
10$:    ADDL2   DATA[R0],SUM
        AOBLSS  #SIZE,R0,10$
```

7.4. РЕЖИМЫ АДРЕСАЦИИ С ИСПОЛЬЗОВАНИЕМ РЕГИСТРОВ ОБЩЕГО НАЗНАЧЕНИЯ

ОБЗОР

В табл. 7.1 приведены режимы адресации, описанные в данной главе. (Обратите внимание, что регистровый режим адресации также включен в таблицу.) Все эти режимы относятся к режимам адресации с использованием регистров общего назначения (за исключением программного счетчика – регистра R15). Слева в таблице перечислены названия режимов. Справа на примере инструкции CLRL с использованием регистра общего назначения R3 показано обозначение этих режимов в языке ассемблера.

Для интерпретации содержимого средней колонки табл. 7.1 вспомним, что каждая инструкция в машинном коде состоит из числового кода операции, за которым следует соответствующее число спецификаторов операндов. Каждый спецификатор операнда состоит из байта спецификатора операнда, который может сопровождать дополнительная информация, такая как смещение, зависящая от значения байта спецификатора операнда. Байт спецификатора операнда представляется двумя шестнадцатеричными цифрами. Вторая шестнадцатеричная цифра задает номер регистра общего назначения, используемого при определении адреса операнда. Первая шестнадцатеричная цифра в байте спецификатора операнда задает код *режима адресации*, указывающий, как этот регистр будет использоваться при определении адреса операнда.

Таблица 7.1. **Режимы адресации с использованием регистров общего назначения**

Режим адресации	Байт спецификатора	Код операции	Пример на языке ассемблера
Регистровый		53 D4	CLRL R3
Регистровый косвенный		63 D4	CLRL (R3)
С автоуменьшением		73 D4	CLRL -(R3)
С автоувеличением		83 D4	CLRL (R3)+
Со смещением (байт)	xx	A3 D4	CLRL D(R3)
Со смещением (слово)	xxxx	C3 D4	CLRL D(R3)
Со смещением (длинное слово)	xxxxxxxx	E3 D4	CLRL D(R3)
Относительный индексный (смещение – байт)	xx AF	43 D4	CLRL A[R3]
Относительный индексный (смещение – слово)	xxxx CF	43 D4	CLRL A[R3]
Относительный индексный (смещение – длинное слово)	xxxxxxxx EF	43 D4	CLRL A[R3]

Примечание. Символ A обозначает адрес, символ D обозначает смещение. Формат смещения – байт, слово или длинное слово выбирается ассемблером в зависимости от значения смещения.

В средней колонке табл. 7.1 в поле кода операции помещается код операции ^XD4, соответствующий инструкции CLRL. Слева от него под заголовком "Спецификатор операнда" расположен байт спецификатора операнда. Во всех примерах второй цифрой байта спецификатора операнда является ^X3, так как используется регистр общего назначения R3. Первая цифра определяет код режима адресации. Как видно из таблицы, код ^X5 обозначает режим регистровой адресации, код ^X6 задает режим регистровой косвенной адресации и т. д.

Заметим, что регистровый косвенный режим, режимы с автоувеличением и автоуменьшением (код режима ^X6, ^X7 и ^X8 соответственно) весьма схожи между собой в том, что в регистре содержится адрес операнда. В регистровом косвенном режиме содержимое регистра не меняется. В режиме с автоуменьшением адрес в регистре уменьшается перед его использованием, в режиме с автоувеличением адрес в регистре увеличивается после его использования.

Коды ^XA, ^XC и ^XE обозначают режимы адресации со смещением, заданным соответственно в формате байта, слова и длинного слова. Ассемблер формирует байт, слово или длинное слово смещения и располагает его в памяти непосредственно за байтом спецификатора операнда. В табл. 7.1 значение смещения обозначено символами xx, xxxx и xxxxxxxx.

Напомним, что во время выполнения инструкции программный счетчик указывает на разные байты этой инструкции. После завершения выполнения инструкции программный счетчик должен указывать на начало следующей инструкции. Чтобы обеспечить это в режимах адресации со смещением, заданным в формате байта, слова и длинного слова, к содержимому программного счетчика автоматически прибавляется 1, 2 или 4 соответственно, чтобы "пропустить" нужное количество байтов, отведенных в данной инструкции под смещение. Увеличение содержимого программного счетчика происходит автоматически, независимо от содержимого регистра общего назначения, используемого для адресации со смещением. Например, в инструкции

```
CLRL    L^X(R3)
```

указан режим со смещением, заданным в формате длинного слова. При выполнении инструкции к содержимому программного счетчика будет прибавлено число 4, чтобы "пропустить" 4-байтовое смещение. В результате программный счетчик будет указывать на начальный адрес следующей инструкции. (Обратите внимание, что содержимое регистра R3 при этом не меняется.) Это увеличение не должно беспокоить программиста, поскольку во всех режимах адресации со смещением увеличение значения программного счетчика осуществляется автоматически.

В последних трех строчках табл. 7.1 приведены примеры индексной адресации (режим 4). В отличие от остальных режимов режим индексной адресации должен использоваться в сочетании с некоторым другим режимом адресации. Использование в инструкциях только индексной адресации недопустимо, например нотация CLRL [R3] является неправильной. В режиме индексной адресации регистр содержит значение индекса, которое используется для вычисления смещения. Второй режим адресации, в сочетании с которым используется индексная адресация, обеспечивает формирование адреса, с которым складывается значение смещения. Регистровый, литеральный и непосредственный режимы адресации не могут использоваться в сочетании с индексной адресацией, поскольку они не формируют адрес операнда, к которому можно было бы прибавить значение смещения.

В табл. 7.1 индексная адресация используется в сочетании с относительной адресацией[1]. Байт спецификатора операнда индексного режима с регистром R3 в качестве индексного регистра содержит код ^X43. Непосредственно за ним следует байт спецификатора операнда для относительной адресации со смещением, заданным в формате байта, слова или длинного слова (коды ^XAF, ^XCF или ^XEF соответственно). Режимы относительной адресации указывают базовый адрес массива, а режим индексной адресации указывает индекс требуемого элемента в массиве.

ЛИТЕРАЛЬНЫЙ РЕЖИМ АДРЕСАЦИИ

В табл. 7.1 приведены восемь режимов адресации, имеющие коды режима 4, 5, 6, 7, 8, A, C и E. Очевидно, что незадействованные коды можно использовать для других целей. Как уже говорилось в гл. 5, байт спецификатора операнда, имеющий значения от ^X00 до ^X3F, используется для задания *литерального режима.* Отличительным признаком этого режима является то, что два старших бита в байте спецификатора операнда содержат нули. В этом случае оставшиеся шесть битов рассматриваются как заданное непосредственно 6-битовое число без знака. Это позволяет описывать в одном байте непосредственно заданные операнды, имеющие значения в диапазоне 0–63. Значение литерала должно быть положительным. Ассемблер автоматически выбирает литеральный режим для небольших по значению положительных непосредственно заданных операндов. Например, при ассемблировании следующей инструкции:

```
MOVL    #5,R0
```

могут быть получены следующие три байта машинного кода:

```
50      05      D0
```

Инструкция с большими по значению непосредственно заданными операндами, такая как

```
MOVL    #64,R0
```

ассемблируется с использованием режима непосредственной адресации и будет занимать 7 байтов:

```
50      00000040      8F      D0
```

Поскольку литерал не имеет знака, то желательно избегать непосредственного задания небольших отрицательных операндов. Это можно сделать, если правильно выбрать инструкцию. Например, инструкция MNEGL #5, R0 более предпочтительна, чем инструкция MOVL #–5,R0, так как в машинном коде первая инструкция будет занимать три байта, а вторая – семь байтов.

РЕЖИМЫ КОСВЕННОЙ АДРЕСАЦИИ

Как уже было показано в разд. 7.2, существует режим адресации, называемый регистровым косвенным режимом. В этом режиме содержимое регистра не используется как операнд, а рассматривается как адрес операнда. Существует еще четыре режима косвенной адресации, которые могут использоваться в сочетании с режимом с автоувеличением и с тремя режимами адресации со смещением (все они приведены в табл. 7.2).

В каждом случае такой адресации адрес вычисляется точно так же, как это делается в случае режимов некосвенной адресации. По этому адресу выбирается длинное слово, и содержимое этого длинного слова используется как *адрес* операнда инструкции. В случае косвенной адресации с автоувеличением значение содержимого регистра увеличивается на четыре. Это делается независимо от типа инструкции, потому что в регистре всегда содержится некоторый адрес, а адрес всегда занимает длинное слово. Такие режимы косвенной адресации используются в основном для работы с таблицами адресов. Адреса в таблице являются адресами операндов или более сложных структур данных.

Таблица 7.3 содержит полный список режимов адресации с использованием регистров общего назначения, реализованных в ЭВМ семейства VAX.

[1] В режиме относительной адресации смещение определяется относительно текущего значения программного счетчика – регистр 15, или ^XF. – *Прим. ред.*

Таблица 7.2. **Режимы косвенной адресации**

Режим адресации	Байт спецификатора	Код операции	Пример на языке ассемблера
Косвенный с автоувеличением	93	D4	CLRL @(R3)+
Косвенный со смещением (байт)	B3	D4	CLRL @D(R3)
Косвенный со смещением (слово)	D3	D4	CLRL @D(R3)
Косвенный со смещением (длинное слово)	F3	D4	CLRL @D(R3)

Примечание. Символ D обозначает смещение. Ассемблер автоматически в зависимости от его значения выберет формат смещения – байт, слово или длинное слово.

Таблица 7.3. **Режимы адресации с использованием регистров общего назначения в ЭВМ семейства VAX**

Режим адресации	Байт спецификатора	Код операции	Пример на языке ассемблера
Литеральный	50 05	D0	MOVL #5,R0*
Индексный	.... xx 43	D4**	CLRL A[R3]
Регистровый	53	D4	CLRL R3
Регистровый косвенный	63	D4	CLRL (R3)
С автоуменьшением	73	D4	CLRL -(R3)
С автоувеличением	83	D4	CLRL (R3)+
Косвенный с автоувеличением	93	D4	CLRL @(R3)+
Со смещением (байт)	xx A3	D4	CLRL D(R3)
Косвенный со смещением (байт)	xx B3	D4	CLRL @D(R3)
Со смещением (слово)	xxxx C3	D4	CLRL D(R3)
Косвенный со смещением (слово)	xxxx D3	D4	CLRL @D(R3)
Со смещением (длинное слово)	xxxxxxxx E3	D4	CLRL D(R3)
Косвенный со смещением (длинное слово)	xxxxxxxx F3	D4	CLRL @D(R3)

* Литеральный режим разрешен только для операнда-источника.

** Обозначение . . . xx указывает соответствующий режим адресации для A.

Примечание. Символ D обозначает смещение. Формат смещения – байт, слово или длинное слово выбирается ассемблером в зависимости от значения смещения.

7.5. РЕЖИМЫ АДРЕСАЦИИ С ИСПОЛЬЗОВАНИЕМ ПРОГРАММНОГО СЧЕТЧИКА

В предыдущих главах был описан целый ряд режимов адресации, использующих программный счетчик. Эти режимы называются *режимами адресации с использованием программного счетчика* и перечислены в табл. 7.4. Непосредственная адресация иллюстрируется инструкцией MOVL, а для других режимов адресации в качестве примера приводится инструкция CLRL. Обратите внимание на символы @# во второй строке: они используются в инструкции CLRL @#A для указания режима абсолютной адресации, ко-

Таблица 7.4. Режимы адресации с использованием программного счетчика

Режим адресации	Байт спецификатора	Код операции	Пример на языке ассемблера
Непосредственный	50 xxxxxxxx 8F	D0	MOVL #K,R0*
Абсолютный	xxxxxxxx 9F	D4	CLRL @#A
Относительный (смещение — байт)	xx AF	D4	CLRL A
Относительный (смещение — слово)	xxxx CF	D4	CLRL A
Относительный (смещение — длинное слово)	xxxxxxxx EF	D4	CLRL A

* Непосредственная адресация разрешена только для операнда-источника.

Примечание. Символ А обозначает адрес, символ К обозначает константу. Формат смещения — байт, слово или длинное слово выбирается ассемблером в зависимости от значения смещения адреса А.

торый рассматривался в гл. 3. Байты спецификатора операнда в этой таблице содержат коды ^X8F, ^X9F, ^XAF, ^XCF и ^XEF. Везде код ^XF указывает, что для определения адреса операнда используется программный счетчик. Символы x, следующие за байтом спецификатора операнда, показывают, сколько шестнадцатеричных цифр, обозначающих дополнительную информацию, должно стоять после байта спецификатора операнда.

Использовать программный счетчик точно так же, как и регистры общего назначения, нежелательно, так как это может привести к непредсказуемым результатам. Однако существует множество подходящих способов его использования, показанных в табл. 7.4. В процессе декодирования инструкции процессор ЭВМ семейства VAX увеличивает значение содержимого программного счетчика на 1 каждый раз, когда происходит выборка очередного байта. Это означает, что на любом этапе декодирования инструкции программный счетчик указывает на следующий байт, выборка которого должна осуществляться. Как это происходит, можно увидеть, рассмотрев выполнение инструкции

```
MOVL    #100,R0
```

В машинном коде инструкция выглядит следующим образом:

```
50      00000064      8F      D0
```

Байт спецификатора операнда ^X8F указывает, что применяется режим адресации с автоувеличением с использованием регистра ^XF, т.е. программного счетчика.

Предположим, что эта инструкция размещается, начиная с адреса ^X00000300. Тогда расположение байтов в памяти будет следующим:

Содержимое	Адрес
D0	0300
8F	0301
64	0302
00	0303
00	0304
00	0305
50	0306

Перед началом выполнения инструкции в программном счетчике будет находиться адрес ^X0300. После выборки кода операции ^XD0 значение содержимого программного счетчика увеличивается на 1 и становится равным ^X0301. Код ^XD0 опознается как код операции инструкции MOVL. Эта инструкция имеет два операнда. Процессор продолжит выборку первого операнда, по адресу ^X0301 выбирается байт ^X8F, после чего значение содержимого программного счетчика увеличивается на 1 и становится равным ^X0302. Выбранный байт — это байт спецификатора операнда. В байте спецификатора операнда код ^X8 обозначает режим адресации с автоувеличением, а ^XF обозначает, что используется регистр 15 (или программный счетчик). Поскольку программный счетчик содержит ^X0302, выбирается длинное слово по адресу ^X0302. Таким образом, первым операндом инструкции MOVL является значение ^X00000064. Однако, поскольку задан режим адресации с автоувеличением, значение содержимого программного счетчика увеличивается теперь на 4 и становится равным ^X0306. После этого декодирование инструкции продолжается с адреса ^X0306, по которому выбирается байт ^X50. Это — байт спецификатора второго операнда. Он указывает, что применяется регистровая адресация с использованием регистра общего назначения R0. Поэтому значение ^X00000064 пересылается в регистр R0.

Как видно из этого примера, непосредственная адресация — это просто режим адресации с автоувеличением, в котором вместо регистра общего назначения используется программный счетчик. Аналогичное исследование других режимов адресации показывает, что не во всех этих режимах можно применять программный счетчик. Например, инструкция в машинном коде

```
50      00000064      6F      D0
```

отличается от только что рассмотренной тем, что в ней используется регистровая косвенная адресация (режим 6) вместо адресации с автоувеличением. Будет произведена выборка операнда ^X00000064, но содержимое программного счетчика не будет увеличено надлежащим образом, чтобы указывать на следующий байт спецификатора операнда (^X50). Адресация с автоуменьшением (режим 7) не позволит даже осуществить выборку операнда ^X00000064, потому что содержимое программного счетчика будет уменьшено до того, как будет произведена выборка операнда. По этим и другим причинам режимы адресации, имеющие код с 4-го по 7-й, не используются с программным счетчиком.

Однако программный счетчик можно применять в режимах адресации, имеющих код с 8-го по 15-й (с ^X8 по ^XF). Ниже показано, как реализуются эти пять режимов адресации с использованием программного счетчика (регистр PC), приведенные в табл. 7.4, с помощью режимов адресации с использованием регистров общего назначения при замене регистра общего назначения на регистр PC.

Режим адресации с использованием регистра PC	Реализация
Непосредственный	Режим с автоувеличением с использованием регистра PC
Абсолютный	Режим косвенной адресации с автоувеличением с использованием регистра PC
Относительный (смещение — байт)	Режим адресации со смещением, заданным в формате байта, с использованием регистра PC
Относительный (смещение — слово)	Режим адресации со смещением, заданным в формате слова, с использованием регистра PC
Относительный (смещение — длинное слово)	Режим адресации со смещением, заданным в формате длинного слова, с использованием регистра PC

Для понимания того, как режим относительной адресации со смещением, заданным в формате байта, реализуется с помощью режима адресации со смещением, заданным в формате байта, в котором вместо регистра общего назначения используется регистр PC, пред-

положим, что, начиная с адреса ^X0300, расположена следующая инструкция, занимающая три байта памяти:

Содержимое	Адрес
D4	0300
AF	0301
F1	0302

Когда содержимое байта по адресу ^X0300 выбирается и интерпретируется как код инструкции, обнаруживается, что это инструкция CLRL, имеющая один операнд. Байтом спецификатора операнда является байт ^XAF. Этот код указывает на использование режима адресации со смещением, заданным в формате байта, относительно регистра R15, т.е. программного счетчика. Затем выбирается байт смещения ^XF1. В дополнительном коде это значение соответствует числу –15, которое надо прибавить к содержимому программного счетчика, чтобы получить исполнительный адрес. Поскольку содержимое программного счетчика увеличивалось при выборке каждого байта, то в настоящий момент он содержит адрес ^X0303, т.е. на единицу больше адреса байта смещения, выбранного последним[1]. Следовательно, исполнительный адрес вычисляется сложением ^XF1 (с распространенным знаковым разрядом) и ^X0303, что дает в результате

```
^X00000303
^XFFFFFFF1
^X000002F4
```

Таким образом очищается содержимое длинного слова, расположенного по адресу ^X02F4. Это обычная схема адресации при выполнении инструкции

```
CLRL    A
```

Относительная адресация со смещением, заданным в формате слова и длинного слова, выполняется аналогично. Ассемблер обычно автоматически выбирает наилучший вариант относительной адресации, если значение адреса известно. Если же адрес неизвестен, то ассемблер использует на всякий случай смещение, заданное в формате длинного слова. Возможны две причины, по которым ассемблер может не знать адрес. Первая связана с настройкой адресов, описанной в гл. 9. Вторая заключается в том, что символическое имя, используемое для задания адреса, или в адресном выражении, не было пока еще определено в программе. По этой причине обычно рекомендуется описывать все области данных в начале программы. Это поможет сократить число ссылок вперед, которые приводят в чрезмерному использованию смещений, задаваемых в формате длинного слова.

В режимах косвенной адресации со смещением также можно использовать программный счетчик. В этих случаях содержимое адресуемого длинного слова будет рассматриваться как адрес, а не как операнд. В следующей инструкции применяется режим косвенной адресации со смещением

```
CLRL    @A
```

В данном случае очищается не длинное слово с адресом A, а ячейка, адрес которой содержится в длинном слове с адресом A.

В табл. 7.5 приводится полный список режимов адресации с использованием программного счетчика, реализованных на ЭВМ семейства VAX.

[1] Как отмечалось ранее, в режиме адресации со смещением адрес в программном счетчике всегда увеличивается с учетом числа байтов, в которых содержится значение смещения, независимо от того, какой из регистров (от ^X0 до ^XF) указан в байте спецификатора операнда.

Таблица 7.5. **Режимы адресации с использованием программного счетчика в ЭВМ семейства VAX**

Режим адресации	Байт спецификатора	Код операции	Пример на языке ассемблера	
Непосредственный	50	xxxxxxxx 8F	D0	MOVL #K,R0*
Абсолютный	xxxxxxxx 9F	D4	CLRL @#A	
Относительный (смещение — байт)	xx AF	D4	CLRL A	
Относительный косвенный (смещение — байт)	xx BF	D4	CLRL @A	
Относительный (смещение — слово)	xxxx CF	D4	CLRL A	
Относительный косвенный (смещение — слово)	xxxx DF	D4	CLRL @A	
Относительный (смещение — длинное слово)	xxxxxxxx EF	D4	CLRL A	
Относительный косвенный (смещение — байт)	xxxxxxxx FF	D4	CLRL @A	

* Непосредственная адресация разрешена только для операнда-источника.

Примечание. Символ A обозначает адрес, символ K — число. Формат смещения — байт, слово, или длинное слово выбирается ассемблером в зависимости от значения смещения адреса A.

7.6. МНОГОМЕРНЫЕ МАССИВЫ

РАЗМЕЩЕНИЕ МАССИВОВ В ПАМЯТИ

Практика работы с языками Паскаль и Фортран показывает, что часто требуются многомерные массивы. Наиболее часто массивы представляются в виде матрицы (см. рис. 7.10).

Поскольку память ЭВМ имеет линейную организацию, более сложные структуры данных, такие как матрицы, должны быть отображены или преобразованы в линейный формат. Для матриц и подобным им структур данных это обычно делается разбиением матрицы на некоторое число одномерных массивов или строк[1]. Матрицу можно развернуть либо по строкам — расположив в памяти строки подряд одну за другой, либо по столбцам — расположив подряд столбцы. В принципе эти два варианта мало чем отличаются.

Большинство компиляторов Фортрана размещают в памяти матрицы по столбцам. Компилятор Паскаля для ЭВМ семейства VAX размещает матрицы по строкам. В табл. 7.6 показано, как матрица размером 5 × 7, элементами которой являются целые числа в формате длинного слова, размещается в памяти ЭВМ семейства VAX, начиная с адреса ^X00003C00, согласно правилам, принятым для языка Фортран. Для языка Паскаль такое размещение в памяти будет соответствовать матрице размера 7X5 и другому порядку индексов, т.е.

$$\begin{matrix} A_{11} & A_{12} & A_{13} & \dots & A_{1n} \\ A_{21} & A_{22} & A_{23} & \dots & A_{2n} \\ A_{31} & A_{32} & A_{33} & \dots & A_{3n} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ A_{m1} & A_{m2} & A_{m3} & \dots & A_{mn} \end{matrix}$$

Рис. 7.10. Матрица размера m X n

[1] Иногда одномерные структуры называют векторами. – *Прим. перев.*

Таблица 7.6. **Размещение в памяти матрицы размера 5 × 7**

Адрес	Элемент матрицы	Адрес	Элемент матрицы	Адрес	Элемент матрицы
^X00003C00	A_{11}	^X00003C30	A_{33}	^X00003C60	A_{55}
^X00003C04	A_{21}	^X00003C34	A_{43}	^X00003C64	A_{16}
^X00003C08	A_{31}	^X00003C38	A_{53}	^X00003C68	A_{26}
^X00003C0C	A_{41}	^X00003C3C	A_{14}	^X00003C6C	A_{36}
^X00003C10	A_{51}	^X00003C40	A_{24}	^X00003C70	A_{46}
^X00003C14	A_{12}	^X00003C44	A_{34}	^X00003C74	A_{56}
^X00003C18	A_{22}	^X00003C48	A_{44}	^X00003C78	A_{17}
^X00003C1C	A_{32}	^X00003C4C	A_{54}	^X00003C7C	A_{27}
^X00003C20	A_{42}	^X00003C50	A_{15}	^X00003C80	A_{37}
^X00003C24	A_{52}	^X00003C54	A_{25}	^X00003C84	A_{47}
^X00003C28	A_{13}	^X00003C58	A_{35}	^X00003C88	A_{57}
^X00003C2C	A_{23}	^X00003C5C	A_{45}		

элемент A(I, J) в Фортране становится элементом A[J, I] в Паскале. Применительно к Паскалю в приведенных ниже пояснениях следует взаимозаменить понятия "строка" и "столбец".

Из табл. 7.6 видно, что адрес любого элемента матрицы $A_{i,j}$ может быть определен при сложении соответствующего смещения элемента с базовым адресом ^X00003C00. Смещение вычисляется по формуле $4[i-1+5(j-1)]$. Например, если требуется выбрать элемент матрицы A_{36}, то $i=3$ и $j=6$ и смещение для этого элемента

$$4[3-1+5(6-1)] = 108_{10} = \text{^X6C}.$$

В результате получается адрес ^X00003C6C. Рассмотрим, как найдена эта формула. Матрица размещена в памяти так, что первый столбец занимает первые пять длинных слов памяти, второй занимает следующие пять длинных слов и т.д. Таким образом начало j-го столбца смещено на $5(j-1)$ длинных слов от начала массива. От начала этого столбца i-е длинное слово в столбце смещено на $i-1$ длинных слов или на $i-1+5(j-1)$ длинных слов от начала массива. (Очевидно, что первый элемент столбца имеет нулевое смещение от начала этого столбца.) Умножение на четыре требуется потому, что адреса длинных слов меняются с дискретностью четыре.

ОБРАБОТКА МАССИВОВ В ЭВМ СЕМЕЙСТВА VAX

Массивы чаще всего обрабатываются по строкам, столбцам или по диагонали. В этих случаях зачастую можно обойтись без значительных адресных вычислений. Ясно, что если матрица сканируется по столбцам, то это то же самое, что сканирование вектора из 35 элементов в случае матрицы 5X7. Поэтому можно применять многие методы, описанные в предыдущих разделах.

Довольно часто (и это отличает матрицы от одномерных массивов) при сканировании матрицы по столбцам необходимо выполнять специальные операции в конце каждого столбца. Например, можно было бы найти сумму всех чисел каждого столбца. Это легко может быть сделано, если ввести второй счетчик для проверки конца столбца. На рис. 7.11 показана программа просмотра по столбцам матрицы размером 5X7.

Как можно видеть в строчке программы с меткой 20$, для выборки элементов матрицы в программе используется индексный режим адресации. Поскольку в этом режиме индекс автоматически умножается на длину операнда, данная программа может использоваться не только для массивов с элементами, имеющими формат длинного слова, но и

для массивов с элементами другого формата – байтов, слов и квадраслов. Это справедливо для всех примеров данного раздела.

На первый взгляд сканирование матрицы по строкам может показаться более сложным, чем по столбцам. Однако это ничуть не сложнее, а при использовании некоторых приемов – даже проще. Сканирование строки означает просто увеличение адреса на соответствующее число. Обратившись к табл. 7.5, можно увидеть, что элементы A_{11}, A_{12}, A_{13}, A_{14} и т.д. расположены по адресам ^X00003C00, ^X00003C14, ^X00003C28, ^X00003C3C и т.д. Разница между соседними адресами составляет 20_{10}, или ^X14. (Число ^X14 соответствует десятичному 20 или числу 4, умноженному на число строк в матрице, что равно числу байтов в столбце. Напомним, что длинное слово содержит четыре байта.)

```
ROWS=5
COLS=7
SIZE=ROWS*COLS
        . . .

        CLRL    R1                  ;ИНДЕКСНЫЙ РЕГИСТР
10$:    ОБРАБОТКА НАЧАЛА СТОЛБЦА
        . . .

        CLRL    R2                  ;РЕГИСТР СЧЕТЧИК
20$:    ОБРАБОТКА ЭЛЕМЕНТА МАССИВА A[R1]
        . . .

        INCL    R1                  ;УВЕЛИЧЕНИЕ ИНДЕКСА
        AOBLSS  #ROWS,R2,20$        ;КОНЕЦ СТРОКИ?
        ЗАВЕРШЕНИЕ ОБРАБОТКИ СТОЛБЦА
        . . .

        CMPL    R1,#SIZE
        BLSS    10$
```

Рис. 7.11. Программа сканирования матрицы размера 5 × 7 по столбцам

```
ROWS=5
COLS=7
SIZE=ROWS*COLS
        . . .

        CLRL    R2                  ;ИНИЦИАЛИЗАЦИЯ ВСПОМОГАТЕЛЬНОГО ИНДЕКСА
10$:    НАЧАЛО ОБРАБОТКИ СТРОКИ
        . . .

        MOVL    R2,R1               ;ИНИЦИАЛИЗАЦИЯ ОСНОВНОГО ИНДЕКСА
20$:    ОБРАБОТКА ЭЛЕМЕНТА МАТРИЦЫ A[R1]
        . . .

        ADDL2   #ROWS,R1            ;ПРИБАВИТЬ 5 ДЛЯ ПЕРЕХОДА К СЛЕДУЮЩЕМУ
        CMPL    R1,#SIZE            ;ЭЛЕМЕНТУ СТРОКИ И ПРОВЕРИТЬ
        BLSS    20$                 ;НЕ ДОСТИГНУТ ЛИ КОНЕЦ СТРОКИ
        ЗАВЕРШЕНИЕ ОБРАБОТКИ СТРОКИ
        . . .

        AOBLSS  #ROWS,R2,10$        ;УВЕЛИЧИТЬ ВСПОМОГАТЕЛЬНЫЙ ИНДЕКС
```

Рис. 7.12. Программа сканирования матрицы размера 5 × 7 по строкам

```
ROWS=5
COLS=7
SIZE=ROWS*COLS
        . . .

        CLRL    R1              ;ИНДЕКСНЫЙ РЕГИСТР
10$:    НАЧАЛО ОБРАБОТКИ СТРОКИ
        . . .

20$:    ОБРАБОТКА ЭЛЕМЕНТА МАТРИЦЫ A[R1]
        . . .

        ADDL    #ROWS,R1        ;ПРИБАВИТЬ 5 ДЛЯ ПЕРЕХОДА К СЛЕДУЮЩЕМУ
        CMPL    R1,#SIZE        ;ЭЛЕМЕНТУ СТРОКИ И ПРОВЕРИТЬ
        BLSS    20$             ;НЕ ДОСТИГНУТ ЛИ КОНЕЦ СТРОКИ
        ЗАВЕРШЕНИЕ ОБРАБОТКИ СТРОКИ
        . . .

        SUBL    #SIZE-1,R1      ;КОНЕЦ СТРОКИ, ВЫЧЕСТЬ 34
        CMPL    R1,#ROWS        ;ИЗ ЗНАЧЕНИЯ ИНДЕКСА И ПРОВЕРИТЬ
        BLSS    10$             ;ВСЕ ЛИ СТРОКИ ОБРАБОТАНЫ
```

Рис. 7.13. Усовершенствованная программа сканирования матрицы размера 5 × 7 по строкам

Поскольку в режиме индексной адресации индекс будет автоматически умножаться на 4, необходимо только позаботиться о том, чтобы каждый раз перед этим содержимое индексного регистра увеличивалось на 5. Проблема заключается в том, как инициализировать регистр-указатель в начале каждой строки. Один из способов заключается в том, чтобы иметь вспомогательный регистр-счетчик индекса, с помощью которого будет осуществляться сканирование первого столбца. На рис. 7.12 приведен пример такого подхода.

Однако существует простой арифметический прием, который делает ненужным вспомогательный счетчик. Этот прием показан на рис. 7.13. Первая строка матрицы обрабатывается при последовательном прибавлении числа 5 к значению индекса в регистре R1. Поскольку матрица размера 5×7 состоит из 35 элементов, то в результате значение в регистре станет равно 35. Когда значение в регистре R1 станет равным 35, это будет означать, что первая строка просмотрена. Если теперь вычесть из этого значения число 34, в регистре R1 окажется 1, т.е. индекс первого элемента второй строки.

ИНСТРУКЦИЯ INDEX

Для упрощения доступа к элементам двухмерных массивов в ЭВМ VAX имеется сложная инструкция INDEX. Эта инструкция не только помогает в вычислении формулы индексации $[i-1+5(j-1)]$, но также задает различные начальные значения индексов и проверяет, находятся ли значения индексов в диапазоне, допустимом для данного массива. Инструкцию можно использовать для работы с многомерными массивами любого размера. Инструкция INDEX имеет шесть операндов – длинных слов.

```
INDEX   SUB,LOW,HIGH,SIZE,IXIN,IXOUT
```

При выполнении инструкций производится вычисление

```
IXOUT := (IXIN+SUB)*SIZE
```

и происходит прерывание по ошибке, если

```
SUB < LOW   ИЛИ   SUB > HIGH
```



```
ROWS=5
COLS=7
        . . .

        CLRL    R1
10$:    CLRL    R2                          ;НОМЕР СТРОКИ (ОТ 0 ДО ROWS-1)
20$:    INDEX   R2,#0,#COLS-1,#ROWS,#0,R0   ;НОМЕР СТОЛБЦА (ОТ 0 ДО COLS-1)
        INDEX   R1,#0,#ROWS-1,#1,R0,R0      ;R0:=R2*ROWS
        РАБОТА С ЭЛЕМЕНТОМ МАТРИЦЫ A[R1]    ;R0:=(R0+R1)*1
        . . .

        AOBLSS  #COLS,R2,20$
        AOBLSS  #ROWS,R1,10$
```

Рис. 7.14. Использование инструкции INDEX для сканирования матрицы размера 5 × × 7 по строкам

На рис. 7.14 показано, как программа на рис. 7.13 должна быть переписана с использованием инструкции INDEX. Для сканирования массива по столбцам, а не по строкам необходимо поменять между собой начальные значения содержимого регистров R1 и R2 и значения приращения адреса.

Преимущество инструкции INDEX заключается в том, что она может использоваться для доступа к элементам матрицы в произвольном порядке. Если регистр R1 содержит индекс строки I, изменяющийся от 0 до ROWS−1, а регистр R2 содержит индекс столбца J, изменяющийся от 0 до COLS−1, то для выборки элемента A[I, J] или A(I, J) могут использоваться следующие две инструкции:

```
20$:    INDEX   R2,#0,#COLS-1,#ROWS,#0,R0   ;R0:=R2*ROWS
        INDEX   R1,#0,#ROWS-1,#1,R0,R0      ;R0:=(R0+R1)*1

        ОБРАБОТКА ЭЛЕМЕНТА МАТРИЦЫ A[R1]
```

УПРАЖНЕНИЯ 7.2

1. Пусть регистр R0 содержит число ^X00001200, а в памяти размещена следующая информация:

Содержимое	Адрес
^X0000120C	^X000011F4
^X000011F8	^X000011F8
^X000011F4	^X000011FC
^X0000120C	^X00001200
^X00001208	^X00001204
^X00001200	^X00001208
^X000011F4	^X0000120C

Как изменится содержимое памяти и регистра R0 после выполнения каждой из следующих инструкций? (Предполагается, что указанное состояние памяти является начальным для *каждой* инструкции)

а.	CLRL	R0	б.	CLRL	(R0)
в.	CLRL	(R0)+	г.	CLRL	-(R0)
д.	MOVL	R0, (R0)	е.	MOVL	(R0), R0
ж.	MOVL	(R0)+, (R0)+	з.	MOVL	(R0)+, (R0)
и.	MOVL	(R0)+, -(R0)	к.	MOVL	-(R0), -(R0)
л.	MOVL	@(R0)+, R0	м.	MOVL	@(R0)+, @(R0)+

2. Переведите каждую из инструкций п. 1 упр. 7.2 в машинный код.

3. Перепишите программы п. 3, 5 или 6 из предыдущего набора упр. 7.1 так, чтобы максимально использовать режимы адресации с автоувеличением или автоуменьшением.

4. Перепишите программы п. 3, 5 или 6 из предыдущего набора упр. 7.1 так, чтобы максимально использовать индексный режим адресации.

5. Напишите программу, которая осуществляет ввод 35 чисел в порядке, соответствующем последовательному просмотру матрицы размером 5X7 по строкам. Напечатайте эти числа в том порядке, в котором бы они следовали при просмотре этой матрицы по столбцам.

*6. Двумерный массив можно сканировать по строкам или по столбцам. Каким образом можно выполнить сканирование трехмерного массива? Напишите программу, которая осуществляет ввод 60 чисел, формирует массив размера 3X4X5 и затем печатает эти 60 чисел в порядке, соответствующем **всем** возможным способам сканирования. (Сканирование в обратном направлении не считается новым способом.)

ГЛАВА 8. СИМВОЛЬНАЯ ИНФОРМАЦИЯ

8.1. ПРЕДСТАВЛЕНИЕ СИМВОЛЬНОЙ ИНФОРМАЦИИ

ВВЕДЕНИЕ

До сих пор обсуждались вопросы, касающиеся того, как выполняются числовые вычисления, и ничего не было сказано о символьной информации, хотя совершенно очевидно, что какой-то способ обработки символьных данных должен существовать. В конце концов, и ассемблер, и операционная система, и компиляторы Паскаля и Фортрана – все имеют дело с операторами, представляющими собой строки текстовых символов. Более того, поскольку все эти обрабатывающие программы на самом деле являются обычными машинными программами, то любая программа может быть способна выполнять обработку символьных данных. Сразу же возникает вопрос о том, каким образом символьная информация представляется в машине. Ответ заключается в следующем: текстовые символы кодируются в виде двоичных чисел, дающих уникальное представление каждого символа алфавита. Поскольку следует иметь в виду еще цифры и знаки пунктуации, нередко используют термин *алфавитно-цифровые* символы. Фактически используемое кодирование может быть произвольным. Другими словами, интерпретация символьного кода полностью определяется разработчиком устройства ввода-вывода, применяемого для считывания и распечатки символов. В ЭВМ VAX используется код, который первоначально был стандартным кодом для телеграфных аппаратов-телетайпов, но в настоящее время применяется для всех типов устройств ввода-вывода. Этот код называется ASCII, что означает *Американский стандартный код для обмена информацией*. Другой символьный код называется EBCDIC, он произошел от кода для перфокарт Холлерита и будет рассмотрен позднее.

Настоящая глава посвящена основным способам обработки символьной информации, а также обработке отдельных битов, составляющих эту информацию. Символьные строки рассматриваются как массивы байтов, а для их обработки используются способы, описанные в предыдущих главах. В гл. 13 будет показано, что в архитектуру ЭВМ VAX включены инструкции, которые могут за одну операцию выполнять обработку целиком всей символьной строки.

СИСТЕМА КОДОВ ASCII

Как только что было сказано, система кодов ASCII возникла при использовании телетайпов. Поэтому каждый код представляет нажатие некоторой комбинации клавиш на клавиатуре, подобной клавиатуре пишущей машинки. (Заметим, что, как и на большинстве пишущих машинок, одновременное нажатие нескольких клавиш обычно является запрещенным или бессмысленным[1]. Исключение составляют клавиши SHIFT ("Смена регистра") и CONTROL ("Управление"), которые сами кода не порождают, но используются в комбинации с другими клавишами. Каждое нажатие клавиши пишущей машинки заставляет печатающий механизм выполнить какое-то действие, т.е. напечатать знак, сделать пробел, перевести каретку и т.п. Аналогично этому каждый отрабатываемый код ASCII вызывает подобное действие печатающего механизма телетайпа. Чтобы напечатать на телетайпе какое-либо сообщение, необходимо последовательно нажимать на клавиши. Чтобы напечатать сообщение, программа должна выдать на телетайп некоторую последовательность кодов ASCII.

В настоящее время общепринятым широко распространенным стандартом является 7-битовая система ASCII. В ней используются 2^7, или 128, кодов, из которых 95 кодов – для печатных символов, 33 – для операций управления, таких как "Возврат каретки" или "Перевод строки". В число 95 печатных символов входят:

26 прописных букв;
26 строчных букв;
10 цифр;
1 пробел;
32 знака пунктуации.

К знакам пунктуации относятся следующие символы:

```
!"#$%&'()*+,-./<>:;=?@[\]_^`{|}~
```

Здесь следует заметить, что некоторые более старые или менее дорогие телетайпы и устройства печати способны распечатывать только 64 символа:

26 прописных букв;
10 цифр;
1 пробел;
27 знаков пунктуации (исключены знаки `{|}~).

Хотя подобные терминалы устарели, многие из них все еще продолжают существовать. Именно поэтому в командном языке операционной системы VAX/VMS не делается различия между прописными и строчными буквами и не используются пять знаков пунктуации, отсутствующих на этих терминалах.

В новейшем стандарте ASCII вместо семи используются восемь битов: 8-битовый код ASCII способен представлять 256 различных кодов, что открывает возможность представления значительно более богатого алфавита. Дополнительные 128 кодов позволяют представлять алфавиты иностранных языков и графические символы.

Из 33 управляющих символов 7-битового кода ASCII обычно используются всего лишь несколько. Наиболее употребительными являются:

BEL – сигнальный звонок телетайпа;

[1] Некоторые терминалы обладают способностью обрабатывать одновременное нажатие клавиш так, что оно воспринимается как последовательность одиночных нажатий.

BS – возврат на шаг назад[1];
HT – горизонтальная табуляция[1];
LF – перевод строки; передвигает бумагу на одну строку;
VT – вертикальная табуляция[2];
FF – перевод страницы; передвигает бумагу на следующую страницу[1];
CR – возврат каретки; перемещает печатающий механизм в начало строки.

Читатель вправе удивиться, для чего же предназначены остальные 26 управляющих символов? В большинстве случаев они не находят применения в оборудовании вычислительной техники (многие из них используются как управляющие коды при коммутации сообщений и передаче телеграмм, т.е. для выполнения тех функций, для которых первоначально применялись телетайпы). Неиспользуемые управляющие символы обычно аппаратурой игнорируются. Например, телетайп, в котором не реализована возможность горизонтальной табуляции, не производит никакого действия, когда получает символ горизонтальной табуляции. Как следствие, такие символы могут использоваться для программных функций. Например, операционная система для ЭВМ VAX может заменять символ горизонтальной табуляции знаками пробела, что позволяет программам эффективно имитировать операцию табуляции. Такая функция является факультативной; она может быть выбрана теми пользователями, устройства вывода которых не обладают встроенной функцией горизонтальной табуляции.

В 7-битовом коде ASCII каждый символ представляется 7-битовым двоичным числом. Первые 32 числа используются для 32 из 33 символов управления; это коды ^X00 – ^X1F. Пробел имеет код ^X20; десятичные цифры 0–9 – коды ^X30 – ^X39; прописным буквам назначены коды ^X41 – ^X5A; строчным буквам – коды ^X61 – ^X7A; под знаки пунктуации некоторым специальным образом выделяются остальные неиспользованные коды ^X21 – ^X7E. Один управляющий символ, называемый Rub Out или DEL (забой, удаление), ассоциируется с кодом ^X7F. Этот символ имеет особую значимость, когда вы вручную набиваете, например, текст статьи, с тем чтобы он оказался занесенным на перфоленту. Если вы делаете ошибку при вводе текста с клавиатуры, то можете передвинуть перфоленту на шаг назад, однако стереть уже пробитые на ленте отверстия вы не сможете. Но вместо этого можно пробить отверстия во всех позициях кадра

Таблица 8.1. **Набор символов кода ASCII**

^X00	NUL	^X10	DLE	^X20	SP	^X30	0	^X40	@	^X50	P	^X60	`	^X70	p
^X01	SOH	^X11	DC1	^X21	!	^X31	1	^X41	A	^X51	Q	^X61	a	^X71	q
^X02	STX	^X12	DC2	^X22	"	^X32	2	^X42	B	^X52	R	^X62	b	^X72	r
^X03	ETX	^X13	DC3	^X23	#	^X33	3	^X43	C	^X53	S	^X63	c	^X73	s
^X04	EOT	^X14	DC4	^X24	$	^X34	4	^X44	D	^X54	T	^X64	d	^X74	t
^X05	ENQ	^X15	NAK	^X25	%	^X35	5	^X45	E	^X55	U	^X65	e	^X75	u
^X06	ACK	^X16	SYN	^X26	&	^X36	6	^X46	F	^X56	V	^X66	f	^X76	v
^X07	BEL	^X17	ETB	^X27	'	^X37	7	^X47	G	^X57	W	^X67	g	^X77	w
^X08	BS	^X18	CAN	^X28	(	^X38	8	^X48	H	^X58	X	^X68	h	^X78	x
^X09	HT	^X19	EM	^X29	)	^X39	9	^X49	I	^X59	Y	^X69	i	^X79	y
^X0A	LF	^X1A	SUB	^X2A	*	^X3A	:	^X4A	J	^X5A	Z	^X6A	j	^X7A	z
^X0B	VT	^X1B	ESC	^X2B	+	^X3B	;	^X4B	K	^X5B	[	^X6B	k	^X7B	{
^X0C	FF	^X1C	PS	^X2C	,	^X3C	<	^X4C	L	^X5C	\	^X6C	l	^X7C	|
^X0D	CR	^X1D	GS	^X2D	-	^X3D	=	^X4D	M	^X5D	]	^X6D	m	^X7D	}
^X0E	SO	^X1E	RS	^X2E	.	^X3E	>	^X4E	N	^X5E	^	^X6E	n	^X7E	~
^X0F	SI	^X1F	US	^X2F	/	^X3F	?	^X4F	O	^X5F	_	^X6F	o	^X7F	DEL

[1] Некоторые устройства не могут выполнять эту функцию.
[2] Большинство устройств не в состоянии выполнять эту функцию.

перфоленты, и такие кадры обычно игнорируются при обработке вводимой информации. Именно по этой причине и был введен специальный символ Rub Out или DEL, код которого состоит из всех единиц (^X7F), в результате чего пробиваются отверстия во всех позициях кадра перфоленты. Учитывая такое традиционное применение, в операционной системе для ЭВМ VAX этот символ используется для возврата на шаг назад с целью исправления ошибок. В табл. 8.1 показано шестнадцатеричное представление всего кода ASCII, состоящего из 128 символов.

РАСПОЛОЖЕНИЕ СИМВОЛОВ КОДА ASCII НА КЛАВИАТУРЕ

Из-за большого размера алфавита кода ASCII (128 символов) было бы непрактично выделять отдельную клавишу под каждый символ. Поэтому обычно для получения некоторых символов используют определенные комбинации клавиш. Например, клавишу смены регистра SHIFT используют для переключения между прописными и строчными буквами. Клавиша SHIFT служит также для переключения между некоторыми знаками пунктуации и цифрами, как это делается на большинстве обычных пишущих машинок. Поэтому, если даже терминал имеет только лишь прописные буквы, у него все равно будет клавиша смены регистра.

На рис. 8.1 показан вид клавиатуры обычного терминала. Обратите внимание, что для управляющих символов эта клавиатура располагает всего лишь несколькими клавишами; обычно это клавиши возврата каретки (RETURN), перевода строки (LINE FEED), табуляции (TAB), возврата на шаг (BACK SPACE), управляющей последовательности кода (ESC) и удаления (DEL). Первые четыре необходимы, так как выполняют функции, постоянно используемые при обычной работе на пишущей машинке. Последние две не выполняют аппаратно-реализованных функций, но широко используются программным обеспечением, например, как упоминалось ранее, для удаления или забоя.

Читатель может задать вопрос, каким же образом порождаются другие управляющие символы. Обратите внимание, что в левом нижнем углу рис. 8.1 имеется клавиша, помеченная буквами CTRL (от CONTROL – управление). Эта клавиша очень похожа на клавишу смены регистра SHIFT тем, что сама по себе не выполняет какой-то определенной функции, но нажимается совместно с какой-либо другой клавишей. Основная функция клавиши управления CTRL заключается в принудительном сбросе в нуль двух определенных битов знакового кода. Примеры:

CTRL/@ эквивалентно ^X00 или NUL;
CTRL/A эквивалентно ^X01 или SOH;
CTRL/B эквивалентно ^X02 или STX;
CTRL/C эквивалентно ^X03 или ETX.

Обратите внимание, что таким способом могут быть получены все управляющие символы, исключая символ DEL. Сюда входят и возврат каретки (CTRL/M), и перевод строки (CTRL/J). Символ DEL, код которого состоит из всех единиц, очевидно, не может быть

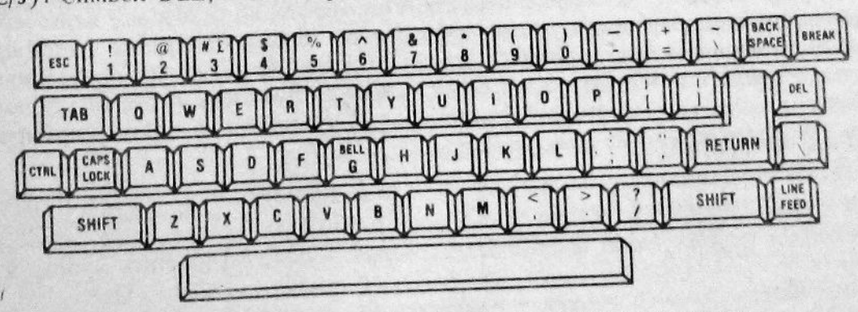


Рис. 8.1. Типовое расположение символов кода ASCII на клавиатуре

получен с помощью клавиши управления, сбрасывающей биты в нуль. Поэтому обычно под этот символ отводится отдельная клавиша.

Многие клавиатуры располагают также специальной клавишей верхнего регистра, которая может иметь маркировку ALL CAPS. Нажатие на эту клавишу приводит к тому, что клавиатура ведет себя как имеющая лишь прописные буквы. Большая часть существующего ныне программного обеспечения разработана так, чтобы сохранить совместимость со старыми терминалами, способными обрабатывать только 64 печатных символа. Клавиша верхнего регистра позволяет использовать терминал с 95-символьной клавиатурой так, как если бы он имел 64-символьную клавиатуру (без этой возможности пришлось бы удерживать клавишу смены регистра большую часть времени при работе с программным обеспечением, ориентированным на старые терминалы). Из табл. 8.1 можно заметить, что код каждой строчной буквы больше кода соответствующей прописной буквы на величину ^X20. Например, код для буквы a есть ^X61, а код для буквы A – ^X41. При нажатии на клавишу верхнего регистра фактически происходит вычитание ^X20 из кодов строчных букв. Вычитание выполняется обычно сбросом бита 5 в двоичном коде строчных букв. Операционная система для ЭВМ VAX способна имитировать действие клавиши верхнего регистра программными средствами, что во многих случаях освобождает от необходимости использования этой клавиши.

УСТРОЙСТВА, ОТЛИЧНЫЕ ОТ ТЕЛЕТАЙПА

Код ASCII является *последовательным* (символ за символом) кодом, так как первоначально он предназначался для работы с телетайпами, которые представляют собой посимвольные печатающие устройства. Но, даже будучи таковым, он также пригоден для построчно-печатающих устройств и видеотерминалов или дисплеев. Последние иногда называют электронно-лучевыми терминалами (CRT), поскольку основным компонентом в них служит электронно-лучевая трубка.

Даже если подобные устройства разрабатываются для быстродействующей групповой обработки символов, для упрощения их подключения информация обычно передается к ним посимвольно. Как правило, построчно-печатающие устройства и видеотерминалы выводят информацию одновременно по целой строке или странице и в отличие от телетайпов не имеют, как таковой, каретки, которую надо возвращать на начало строки. Но тем не менее для совместимости с обыкновенным кодом ASCII символы возврата каретки и перевода строки традиционно используются для завершения одной строки и продвижения к следующей. Поэтому в большинстве случаев подобные устройства можно рассматривать как обычные телетайпы.

Некоторые устройства обладают особыми возможностями. Устройства печати могут позволять программисту запрограммировать продвижение бумаги на начало новой страницы. Видеотерминалы могут иметь такие функции, как стирание экрана, управление курсором, использование режима прокрутки вместо страничного режима. *Курсор* – это мерцающий маркер, который показывает на экране место, где должен вводиться текст. Обычно по мере ввода текста курсор перемещается вправо, но некоторые видеотерминалы позволяют позиционировать курсор в любой точке экрана. *Прокрутка* (scrolling) дает возможность пользователю добавлять строки в нижней части экрана, прокручивая остальной текст вверх с потерей верхних строк. Такие специальные возможности достигаются обычно с помощью некоторого протокола, построенного на основе управляющих символов. Например, строка из семи символов

(esc) [5; 39f

заставляет терминал типа VT-100 или любой другой терминал, реализующий то, что называют *стандартной управляющей последовательностью* ANSI[1], переместить курсор в 39-ю

[1] American National Staudards Institute – Американский национальный институт стандартов. – *Прим. ред.*

знаковую позицию 5-й строки экрана. Конкретная информация о таких последовательностях приводится в руководствах для подобных терминалов.

ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СИМВОЛОВ В ЭВМ СЕМЕЙСТВА VAX

В ЭВМ семейства VAX отдельные символы хранятся как байты. При использовании 7-битового кода ASCII операционная система VAX/VMS, как правило, сбрасывает в нуль старший бит каждого байта. Строка символов обычно размещается в памяти как массив байтов. Следовательно, если желательно распечатать на одной строчке сообщение HELLO!!!, то потребуются символы H, E, L, L, O, !, ! и !. Для этих восьми символов нужен массив из восьми байтов. Если этот массив называется STRING и расположен, начиная с адреса ^X00000400, то он будет выглядеть так, как показано на рис. 8.2. В программу на языке ассемблера эти данные могут быть введены с помощью директивы .BYTE следующим образом:

```
STRING: .BYTE    ^X48, ^X45, ^X4C, ^X4C, ^X4F, ^X21, ^X21, ^X21
```

Специального замечания по поводу байтовых массивов заслуживает ситуация, когда байты или еще меньшие элементы данных упаковываются в элементы данных большего объема, такие как длинные слова; при этом в первую очередь заполняются их младшие биты. Вследствие этого байты представляются упорядоченными в обратном порядке. Например, если приведенный массив из 8 байтов рассматривать как упакованный в массив из двух длинных слов, то он выглядел бы так:

Содержимое	Адрес	Символический адрес
^X4C4C4548	^X0400	STRING
^X2121214F	^X0404	STRING+4

Именно по этой причине представление этих байтов в объектном коде на листинге ассемблера читается справа налево. Таким образом, порожденный по предыдущей директиве .BYTE список байтов имел бы в листинге ассемблера следующий вид:

```
21  21  21  4F  4C  4C  45  48    0400.
```

Всем понятно, что такое представление неудобно, а для работы с ним требуются определенные навыки. Но подобная проблема в той или иной форме существует у всех байт-ориентированных ЭВМ, и нотация, принятая для ЭВМ VAX, вероятно, сбивает с толку в меньшей степени, чем многие другие представления.

Содержимое	Адрес	Символ	Символический адрес
		H	STRING
^X48	^X0400	E	STRING+1
^X45	^X0401	L	STRING+2
^X4C	^X0402	L	STRING+3
^X4C	^X0403	O	STRING+4
^X4F	^X0404	!	STRING+5
^X21	^X0405	!	STRING+6
^X21	^X0406	!	STRING+7
^X21	^X0407		

Рис. 8.2. Строка символов в коде ASCII

БАЙТЫ И ИНСТРУКЦИИ РАБОТЫ С БАЙТАМИ

Во внутреннем представлении ЭВМ VAX коды символов – это просто множество чисел от 0 до 255 (если они интерпретируются как числа без знака), ничем не отличающихся от каких-либо других байтов. Обратясь к рис. 8.2, можно увидеть, что для обработки символов в строке могут быть применены различные байтовые инструкции, такие как MOVB, INCB, DECB, ADDB2 и т. п. Например, по инструкции

```
MOVB    STRING+5, STRING+2
```

код ^X21, расположенный по адресу STRING+5 или ^X405, будет помещен по адресу STRING+2 или ^X402. После этого содержимое байтов массива строки стало бы таким, как показано на рис. 8.3. При выводе на печать была бы сформирована строка **HE!LO!!!**. Аналогично выполнение инструкции

```
ADDB2   #5, STRING+3
```

приводит к тому, что вместо числа ^X4C по адресу STRING+3 будет находиться число ^X51. Это число является кодом ASCII для буквы **Q**. Следовательно, если снова распечатать строку, то она будет иметь вид **HE!QO!!!**.

Важно не забывать о том, что когда речь идет о процессоре ЭВМ VAX, то любая символьная строка есть не что иное, как массив 8-битовых чисел. Интерпретация этих чисел как кодов символов является функцией устройств ввода-вывода и программного обеспечения. Следовательно, ответственность за корректность операций, выполняемых над символьными строками, ложится на программиста. Отличить корректные операции от не имеющих смысла бывает не так уж легко. Все зависит от конкретной ситуации. Вернемся к предыдущему примеру. Число 5 было прибавлено к букве L, что дало букву Q. В этой ситуации такая операция не представляется полезной, но фактически к этому и не стремились. Однако операции подобного рода могут пригодиться при шифровании или при преобразовании кодов символов из кода ASCII в какой-то другой код для устройства ввода-вывода, не способного работать с кодом ASCII. (См. раздел "Другие представления символов" в конце данной главы.)

Содержимое	Адрес	Символический адрес
^X48	^X0400	STRING
^X45	^X0401	STRING+1
^X21	^X0402	STRING+2
^X4C	^X0403	STRING+3
^X4F	^X0404	STRING+4
^X21	^X0405	STRING+5
^X21	^X0406	STRING+6
^X21	^X0407	STRING+7

Рис. 8.3. Представление символьной строки в памяти

СОГЛАШЕНИЯ ЯЗЫКА АССЕМБЛЕРА ДЛЯ СИМВОЛЬНЫХ СТРОК

В предыдущем разделе было показано, как можно вставить символьную строку в текст программы с помощью директивы .BYTE. Чтобы ввести символьную строку таким образом, необходимо отыскать каждый символ вводимого текста в табл. 8.1. Ясно, что это может стать обременительным, если программа содержит большое число символьных

строк. Для упрощения задания символьных строк в программе, в ассемблере VAX-MACRO имеется директива .ASCII. Эта директива позволяет вставлять в программу символьные строки. Вставляемые символьные строки заключаются в кавычки, как это обычно делается во многих языках высокого уровня. Например, символьная строка, которая ранее вводилась как

```
STRING: .BYTE   ^X48, ^X45, ^X4C, ^X4C, ^X4F, ^X21, ^X21, ^X21
```

может быть введена так:

```
STRING: .ASCII  "HELLO!!!"
```

В обоих случаях содержимое памяти получается одним и тем же.

Если строка, которую необходимо ввести, содержит непечатные символы, такие как символ табуляции (^X09), возврата каретки (^X0D), звонка (^X07) и т. п., возникает проблема. Хотя некоторые непечатные символы возможно включить в определяемую строку, это плохая программистская практика, так как в таком случае трудно бывает определить, какие именно символы в действительности находятся в строке. Для разрешения этой проблемы в директиве .ASCII допускаются байтовые выражения, заключенные в угловые скобки < и >. Предположим, например, что строке **HELLO!!!** должны предшествовать два символа табуляции, а вслед за ней должен располагаться символ "звонок". При передаче такой строки на терминал слово **HELLO!!!** появится, начиная со второй позиции табуляции, а после вывода слова прозвенит сигнальный звонок терминала. Для создания этой строки может быть использована следующая директива .ASCII:

```
ALARM_STRING:
        .ASCII  <^X09><^X09>"HELLO!!!"<^X07>
```

Обратите внимание, что массив ALARM_STRING содержит 11 символов кода ASCII.

Конечно, лучший стиль программирования состоял бы в определении для символов табуляции и звонка символических имен и в использовании этих имен вместо фактических шестнадцатеричных чисел. Это могло бы выглядеть так:

```
TAB = ^X09
BEL = ^X07
. . .

ALARM_STRING:
        .ASCII  <TAB><TAB>"HELLO!!!"<BEL>
```

Еще одна очевидная проблема заключается в том, что невозможно включить знак " между кавычками, ограничивающими строку. Ассемблер VAX – MACRO дает возможность решить эту проблему, позволяя использовать альтернативные знаки кавычек. Считается, что первый печатный символ, отличный от знака <, определяет начало кавычек. Второе появление того же самого символа означает окончание кавычек. Следовательно, предыдущая строка может быть введена и следующим образом:

```
ALARM_STRING:
        .ASCII  <TAB><TAB>/HELLO!!!/<BEL>
```

или

```
ALARM_STRING:
        .ASCII  <TAB><TAB>%HELLO!!!%<BEL>
```

Ясно, что при выборе какого-то конкретного символа для кавычек в строку можно ввести любые другие символы, кроме него самого. Однако для стандартизации многие программисты ограничиваются применением только нескольких символов в качестве кавычек, таких как ", ' или /.

При передаче на терминал последовательных строк необходимо отделять их друг от друга управляющими символами возврата каретки (^X0D) и перевода строки (^X0A). В большинстве случаев операционная система VAX/VMS сама вместо пользователя вставляет эти символы. Поэтому чаще всего не нужно включать их в символьные строки. Но во многих других вычислительных системах пользователи должны явным образом вставлять эти управляющие символы. Форматирование строк подробно рассматривается в гл. 11.

Чтобы увидеть, как могут использоваться символьные строки, рассмотрим некий программный сегмент, в котором каждый символ строки STRING передается подпрограмме PROCESS. Символы передаются каждый раз по одному через регистр R0. Не обязательно знать, что именно делает подпрограмма PROCESS, за исключением того, что она принимает по одному символу при каждом обращении к ней и выполняет какое-то действие. Возможно, она выводит символ на печать, возможно, помещает его в буфер. Используя полученные значения, можно написать следующую программу:

```
STRING:   .ASCII    "HELLO!!!"
          . . .

PASS:     CLRL      R1
10$:      MOVB      STRING[R1],R0
          JSB       PROCESS
          AOBLSS    #8,R1,10$
```

Одна проблема, связанная с этой программой, состоит в том, что инструкция AOBLSS содержит непосредственный операнд #8, который сообщает, сколько символов содержится в массиве STRING. Это порождает трудности, если программист когда-либо модифицирует символьную строку, ибо при этом он обязан всегда помнить о необходимости замены операнда #8 при указании длины новой строки.

Одно решение этой проблемы состоит в использовании символического выражения для вычисления длины строки. Обратите внимание. Если вычесть адрес начала массива строки из адреса следующей за массивом доступной ячейки, то разность даст число байтовых ячеек в массиве. Например, возвращаясь к рис. 8.3, можно увидеть, что байтовый массив расположен в памяти, занимая адреса ^X400 – ^X407. Значит, следующей доступной ячейкой является ячейка с адресом ^X408, а разность между адресом этой ячейки и адресом первой ячейки ^X400 равна 8, что и есть длина массива. В предыдущую программу все это можно включить следующим образом:

```
STRING:   .ASCII    "HELLO!!!"
STRINGEND:
          . . .

PASS:     CLRL      R1
10$:      MOVB      STRING[R1],R0
          JSB       PROCESS
          AOBLSS    #STRINGEND-STRING,R1,10$
```

В результате ассемблирования этой программы будет получен такой же двоичный код, как и в ее предыдущем варианте. Однако, поскольку цифра 8 заменена выражением STRINGEND – STRING, ассемблер автоматически занесет длину строки в инструкцию AOBLSS. Следовательно, можно модифицировать массив STRING, не внося какие-либо изменения в программу. Например, если подлежащая обработке строка заменяется на **GOOD BYE**!!!, то две первых строки станут такими:


```
STRING:   .ASCII    "GOOD BYE!!!"
STRINGEND:
```

Теперь строка имеет длину 11. Если метке STRING был назначен, как и прежде, адрес ^X400, метке STRINGEND будет назначен адрес ^X40B. Тогда их разность равна ^XB, или 11 в десятичном представлении.

Другой способ решения этой проблемы заключается в использовании какого-либо специального признака для обозначений конца строки. Для символьных строк в качестве такого признака подходит значение 0, поскольку 0 – это код символа NULL кода ASCII, который в большинстве систем просто игнорируется. В ассемблере ЭВМ VAX-11 данную возможность обеспечивает директива .ASCIZ, равносильная директиве .ASCII, за исключением того, что к концу строки добавляется нулевой байт. Следовательно, оператор

```
STRING: .ASCIZ    "HELLO!!!"
```

эквивалентен оператору

```
STRING: .ASCII    "HELLO!!!"<0>
```

В следующей программе показано, как можно использовать признак конца строки – нулевой байт, достигая при этом такого же результата, как и в предыдущем примере, вызывая подпрограмму PROCESS и передавая ей каждый символ строки через регистр R0 как байт:

```
STRING: .ASCIZ    "HELLO!!!"
        . . .

PASS:   MOVAL     STRING,R1
10$:    MOVB      (R1)+,R0
        BEQL      20$
        JSB       PROCESS
        BRB       10$
20$:
```

Этот способ обладает очевидными преимуществами, когда используются строки переменной длины или строки, порождаемые внутри программы. Кроме того, он позволяет применять режим адресации с автоувеличением в инструкции MOVB.

Существует все же еще один способ работы с символьными строками. Хотя данный способ несколько сложен, он имеет преимущества при передаче символьных строк подпрограммам и применяется на ЭВМ VAX для языков высокого уровня как стандартный способ обработки символьных строк. Более детальные пояснения даются в гл. 9. Этот способ требует присоединения к символьной строке дескриптора. Дескриптор состоит из двух длинных слов. Первое длинное слово разбивается на два слова. Первое из этих слов задает длину строки в байтах. Второе слово содержит некоторые коды, которые для системного программного обеспечения идентифицируют информацию как символьную строку. Во второе длинное слово помещается адрес первого байта символьной строки. Формат дескриптора порождается с помощью директивы .ASCID. Например, такая директива:

```
STRING: .ASCID    "HELLO!!!"
```

эквивалентна следующему фрагменту программы, составленному из ранее рассмотренных директив:

```
STRING: .WORD    20$-10$          ;ДЛИНА = 8
        .WORD    ^X010E           ;СПЕЦИАЛЬНЫЙ КОД
        .ADDRESS 10$              ;АДРЕС СТРОКИ
10$:    .ASCII   "HELLO!!!"       ;СИМВОЛЬНАЯ СТРОКА
20$:
```

Директива .ADDRESS используется для занесения некоторого адреса в длинное слово памяти. По своему действию эта директива очень близка директиве .LONG, за исключением того, что директиву .LONG следует использовать только для определения констант, а директива .ADDRESS может применяться для задания переместимых адресов. Обе директивы поместили бы в длинное слово адрес 10$, но здесь для переместимых адресов следует использовать директиву .ADDRESS, так как она обеспечивает дополнительную информацию для ассемблера и компоновщика.

В следующей программе показано, как можно переписать предыдущий пример, адресуя элементы строки через дескриптор вместо непосредственного обращения к ним:

```
STRING: .ASCID   "HELLO!!!"
        . . .

PASS:   CLRL     R1
        MOVZWL   STRING,R2
10$:    MOVB     @STRING+4[R1],R0
        JSB      PROCESS
        AOBLSS   R2,R1,10$
```

Обратите внимание, что уровень косвенности при адресации массива STRING в инструкции MOVB на один уровень выше, чем в предыдущем примере. Выражение STRING+4 определяет адрес длинного слова, содержащего адрес начала строки в коде ASCII, а @STRING+4 – адрес первого байта строки. Поэтому первый операнд инструкции MOVB имеет вид @STRING+4[R1]. Следует также отметить, что инструкция MOVZWL применяется для преобразования формата 16-битового слова, расположенного по адресу STRING, в формат длинного слова, пересылаемого в регистр R2 при интерпретации их содержимого как числа без знака. Это преобразование необходимо для инструкции AOBLSS, в которой требуется, чтобы первые два операнда имели формат длинного слова.

УПРАЖНЕНИЯ 8.1

1. Используя телетайп или видеотерминал, работающий в автономном режиме (т.е. не подключенный к ЭВМ, а передающий символы самому себе), выясните действие всех его клавиш. Используйте также клавиши смены регистра (SHIFT) или управления (CTRL) совместно со всеми другими клавишами и опишите, что происходит. В частности, обратите внимание на следующие вопросы:

а) Имеет ли ваш терминал прописные и строчные буквы? Если нет, то что происходит при смене регистра? Возможно ли напечатать все буквы при нажатой клавише смены регистра? Если есть строчные буквы, то имеется ли клавиша блокировки смены регистра или клавиша верхнего регистра? Какое между ними различие?

б) Каково действие клавиши управления? Какие управляющие символы вызывают какое-либо видимое действие, а если вызывают, то какое именно? Всегда ли вы замечаете какой-то видимый эффект?

г) Некоторые терминалы реагируют на символ управляющей последовательности ко-

да (ESC), сопровождаемый другими символами. Посмотрите, сможете ли вы заметить какую-либо реакцию.

2. Повторите все шаги п. 1 упр. 8.1 на терминале, подключенном к вычислительной системе VAX. Как вы объясните различия в получаемых результатах при выполнении п. 1 и п.2?

8.3. УПРОЩЕННЫЙ ВВОД-ВЫВОД

Тема ввода-вывода в целом весьма сложна и в существенной мере выходит за пределы настоящего изложения. Некоторые детали ввода-вывода рассматриваются в гл. 11. Поскольку необходимо все же каким-то образом осуществлять взаимодействие с выполняемыми программами, материал данного раздела предназначается для того, чтобы дать простейшую схему, по которой пользователи смогут считывать информацию с терминала и выводить ее на него. Такие операции выполняются с помощью трех подпрограмм, называемых IOINIT, RLINE и PLINE.

Эти подпрограммы входят в пакет подпрограмм, введенный в гл. 5 и описанный в приложении Б. Фактически IOINIT – это та самая подпрограмма, которая использовалась для инициализации подпрограмм RNUM и PNUM. Она инициализирует также подпрограммы RLINE и PLINE и должна быть вызвана перед вызовом любой из них. Подпрограмма инициализации не должна вызываться более одного раза, даже если вызываются все четыре подпрограммы ввода-вывода.

Точно так же, как подпрограммы RNUM и PNUM используются для ввода и распечатки чисел, подпрограммы RLINE и PLINE применяются для ввода и распечатки символьных строк. Обе подпрограммы считают, что основная программа имеет буфер или массив для хранения символов. При работе с подпрограммой RLINE в этот массив будут пересылаться вводимые символы, в случае подпрограммы PLINE этот массив содержит символы, подлежащие распечатке. Кроме того, подпрограмме PLINE необходимо знать, сколько должно распечатываться символов, а подпрограмма RLINE должна знать объем доступного для ввода пространства в массиве. Наконец, после возврата из подпрограммы RLINE основная программа должна быть в состоянии определить, сколько фактически было введено символов. Эта информация передается между программами через регистры R0 и R1. Регистр R0 содержит адрес буфера символов, а R1 – информацию о длине введенной строки.

Для распечатки, например, строки **HELLO!** можно использовать следующую последовательность операторов:

```
STRING: .ASCII   "HELLO!!!"
STRINGEND:
        . . .

        MOVAL    STRING,R0
        MOVZWL   #STRINGEND-STRING,R1
        JSB      PLINE
```

Обратите внимание, что длина строки символов загружается в регистр R1 как слово. Делается это таким образом, потому что именно в таком формате она используется в подпрограмме PLINE. Инструкция MOVZWL применяется по двум причинам. Во-первых, оставлять мусор в старшей половине регистра – это плохая программистская практика, а по инструкции MOVZWL происходит очистка старших 16 битов регистров. Во-вторых, ассемблер выпаст предупреждающее сообщение, если значение STRINGEND – STRING окажется больше, чем 65535 или $2^{16} - 1$.

Обратите также внимание, что к выводимой строке не добавляются символы возврата каретки или перевода строки. Поскольку подпрограмма PLINE распечатывает только целиком строку, управление переводом строк автоматически поддерживается обычными

средствами ввода-вывода операционной системы VAX/VMS. Если все же символы возврата каретки и перевода строки включаются в текст, это приводит к выводу дополнительных строк.

Вызывающая последовательность подпрограммы PLINE хорошо согласуется с использованием дескрипторов строк, что можно увидеть из альтернативного способа распечатки той же самой строки:

```
DSTRING:        .ASCID   "HELLO!!!"
        . . .

        MOVL    DSTRING+4, R0       ;СОДЕРЖИМЫМ DSTRING+4 ЯВЛЯЕТСЯ АДРЕС
        MOVZWL  DSTRING, R1         ;В СЛОВЕ DSTRING СОДЕРЖИТСЯ ДЛИНА
        JSB     PLINE
```

Подпрограмма RLINE работает аналогично. Основное отличие состоит в том, что она ожидает, что ей будет сообщена максимальная длина буфера. Это необходимо, чтобы программа была защищена от приема строк, превосходящих по длине размер буфера и перекрывающих программную область. Примером работы подпрограммы RLINE может служить следующий фрагмент, предназначенный для считывания вводимой строки в буфер, вмещающий 80 символов:

```
BSIZE=80
BUFFER: .BLKB    BSIZE
        . . .

        MOVAL    BUFFER, R0    ;ПОМЕСТИТЬ АДРЕС БУФЕРА В R0
        MOVZWL   #BSIZE, R1    ;ПОМЕСТИТЬ ДЛИНУ БУФЕРА В R1
        JSB      RLINE
```

После возврата из подпрограммы RLINE произойдет следующее:

1. Введенные символы будут помещены в байтовый массив BUFFER. Поскольку эта подпрограмма работает построчно, никаких управляющих символов типа возврата каретки или перевода строки в буфер не заносится.

2. Число фактически прочитанных символов будет помещено в формате слова в регистр R1. Оно может быть равно 0, если была нажата только клавиша "Возврат каретки", или любому числу вплоть до 80, если было напечатано столь много символов. Но ни в коем случае это число не будет превосходить то значение, которое было в регистре R1 первоначально, независимо от фактического числа вводимых символов.

3. Код ошибки передается вызывающей программе в формате байта через регистр R0. Множество причин может вызывать ошибки при вводе. Три наиболее частые из них: было введено больше символов, чем может вместить буфер; была попытка чтения после достижения признака конца вводимых данных (особенно вероятно в системе пакетной обработки); какой-либо тип программной ошибки, например не была вызвана подпрограмма IOINIT. Поскольку сообщается код ошибки, то программа может проверить, не случилось ли что-либо подобное. Например, некоторые программы разрабатываются так, что выполняют чтение до тех пор, пока не кончатся все данные. Код ошибки, порождаемой при достижении конца данных, может использоваться для завершения работы программы. Код ошибки распознается так: если в регистре R0 содержится число 1 (в формате слова), то программа работает нормально и ошибки нет; любое другое значение обозначает какую-то ошибку. Это является стандартной процедурой обработки ошибок для программ, выполняемых в среде операционной системы VAX/VMS. Число различных кодов ошибок весьма значительно, поэтому лучшее, что можно сделать, – это позволить операционной системе самой описать для вас ошибку. Это можно сделать, передав системе код ошибки, поместив его в строку $EXIT_S. В данном случае это выглядит как $EXIT _SR0.

Код ошибки следует проконтролировать в программе с помощью таких инструкций:

```
        . . .

        JSB      RLINE
        CMPW     R0, #1
        BEQL     10$
        $EXIT_S  R0
10$:    СЛЕДУЮЩАЯ ИНСТРУКЦИЯ
```

Заметим, что пара инструкций CMPW R0, #1 и BEQL 10$ может быть заменена одной инструкцией, которая будет рассмотрена в следующем разделе. Заметим также, что подпрограмма RLINE – единственная из пяти подпрограмм ввода-вывода в этом пакете, возвращающая код ошибки. Все другие подпрограммы ошибки игнорируют. Более подробное описание кодов ошибок и того, что с ними делать, приведено в гл. 11.

8.4. ИНСТРУКЦИИ ПОБИТОВОЙ ОБРАБОТКИ

НЕОБХОДИМОСТЬ РАБОТЫ С БИТАМИ

Зачастую бывает удобно упаковывать информацию в словах или байтах в виде битовых групп разного размера. Ниже приводится несколько примеров отдельных элементов слов или байтов, ранее уже обсуждавшихся, имеющих особое значение.

1. Бит 31 в длинном слове, бит 15 в слове или бит 7 в байте определяют знак числа.

2. Бит 0 слова или байта сообщает о том, является ли представляемое число четным или нечетным.

3. Биты 5 и 6 символьного кода ASCII сообщают, является ли данный символ прописной или строчной буквой, управляющим символом или же входит в основную группу знаков пунктуации и цифр.

СЕМЕЙСТВА ИНСТРУКЦИЙ BIS, BIC И BIT

Для упаковки и распаковки информации на ЭВМ VAX имеются три семейства инструкции. Эти инструкции предназначены для установки или очистки определенных битов слова или же для проверки состояния отдельных битов. Это следующие три семейства инструкций:

Мнемоника	Назначение
BIS	Установить биты (BIt Set)
BIC	Сбросить биты (BIt Clear)
BIT	Проверить биты (BIt Test)

Каждое семейство инструкций включает инструкции для работы с операндами различного формата, а инструкции семейства BIS и BIC могут работать с двумя или тремя операндами. Полный набор инструкций:

Мнемоника	Код операции	Операнды	Операция
BISB2	88	mask, dst	dst <- dst OR mask
BISW2	A8		
BISL2	C8		
BISB3	89	mask, src, dst	dst <- src OR mask
BISW3	A9		
BISL3	C9		
BICB2	8A	mask, dst	dst <- dst AND (NOT mask)
BICW2	AA		
BICL2	CA		

BICB3 BICW3 BICL3	8B AB CB	mask, src, dst	dst <- src AND (NOT mask)
BITB BITW BITL	93 B3 D3	mask, src	src AND mask

Первый операнд каждой инструкции – это *маска,* используемая для выделения некоторых битов во втором операнде. В инструкциях семейств BIS и BIC, работающих с двумя операндами, результат запоминается во втором операнде. В инструкциях с тремя операндами результат помещается в третий операнд. Это похоже на инструкцию ADDL2 в сравнении с инструкцией ADDL3. Например, инструкция BISW2 приводит к тому, что каждый бит второго операнда, имеющего формат слова, устанавливается в 1, если установлен в 1 соответствующий бит слова маски. Биты во втором операнде, которым соответствуют нулевые биты маски, не изменяются. Как видно из описания, это представляет собой побитовую операцию **ИЛИ**, выполняемую над маской и вторым операндом. Рассмотрим, например, 16-битовые двоичные числа до и после выполнения инструкции BISW2:

X = 1101 1110 0001 0001
Y = 1100 0101 0110 1000 До выполнения BISW2 X, Y
X = 1101 1110 0001 0001
Y = 1101 1111 0111 1001 После выполнения BISW2 X, Y

Обратите внимание, что маска не меняется. Для варианта инструкции с тремя операндами – BISW3 X, Y, Z значение Y остается неизменным, а результат помещается в Z.

Инструкция BICW2 подобна приведенной выше, за исключением того, что происходит очистка (сброс) битов второго операнда, если установлены в 1 соответствующие биты маски. Эта операция совпадает с логической операцией **И**, выполняемой над поразрядным дополнением маски (операция **НЕ**) и вторым операндом (см. гл. 2). Повторим теперь предыдущий пример, используя инструкцию BICW2:

X = 1101 1110 0001 0001
Y = 1100 0101 0110 1000 До выполнения BICW2 X, Y
X = 1101 1110 0001 0001
Y = 0000 0001 0110 1000 После выполнения BICW2 X, Y

Семейство инструкций BIT отличается от двух рассмотренных семейств; оно схоже с семейством инструкций CMP в том, что вычисленный результат не сохраняется. Оба семейства используются лишь для установки кодов условий. Для семейства инструкций BIT вычисленный результат определяется так: единица в данной битовой позиции результата будет только тогда, когда *оба* операнда имеют 1 в соответствующей позиции. Это логическая операция **И**. Если в вычисленном результате не окажется ни одной 1, будет установлен код условия Z, который может быть проверен с помощью инструкций BEQL и BNEQ. Если равен 1 старший бит вычисленного результата, устанавливается код условия N, который можно проверить инструкциями BLSS и BGEQ. Поскольку ни переполнения, ни переноса произойти не может, эти коды условий обычно не используются. Однако, чтобы сохранить совместимость, так как многие программисты могут захотеть воспользоваться инструкциями переходов, бит V всегда очищается, а бит C остается без изменения. Инструкции семейств BIC и BIS устанавливают коды условий аналогично, за исключением того, что биты N и Z зависят от значения, сохраняемого в операнде получателя, а не от значения временного результата.

Покажем пример работы инструкции BITW, используя прежние значения X и Y:

X = 1101 1110 0001 0001
Y = 1100 0101 0110 1000 До и после выполнения BITW X, Y
врем. = 1100 0100 0000 0000 После BITW X, Y (врем. не сохраняется)

Инструкция BITW X, Y оставляет значения X и Y без изменения. Временный результат, который не сохраняется, используется для установки следующих кодов условий:

N = 1
Z = 0
V = 0
C = Сохраняет предыдущее значение до выполнения инструкции BITW

Обратите внимание, что использование бита Z может приводить к путанице. Этот бит устанавливается в 1 (значение "истина"), если равен 0 временный результат, но это означает лишь то, что в исходном операнде выделенные по маске биты равны 0.

ИНСТРУКЦИЯ XOR

Похожим на предыдущие семейства инструкций побитовой обработки является семейство XOR. Инструкции этого семейства вычисляют побитовое **ИСКЛЮЧАЮЩЕЕ_ИЛИ** двух операндов и запоминают результат или во втором, или в третьем операнде в зависимости от того, используется ли инструкция с двумя или тремя операндами. Результат операции **ИСКЛЮЧАЮЩЕЕ_ИЛИ**, выполняемой над двумя битами, равен 1 тогда, и только тогда, когда равен 1 какой-либо один из битов, *но не оба* вместе. Это можно увидеть из следующей таблицы:

0 **ИСКЛЮЧАЮЩЕЕ_ИЛИ** 0 = 0
0 **ИСКЛЮЧАЮЩЕЕ_ИЛИ** 0 = 1
1 **ИСКЛЮЧАЮЩЕЕ_ИЛИ** 0 = 1
1 **ИСКЛЮЧАЮЩЕЕ_ИЛИ** 1 = 0

Используя прежние значения X и Y из примеров предыдущего раздела, можем посмотреть, как работает инструкция XORW2:

X = 1101 1110 0001 0001
Y = 1100 0101 0110 1000 До выполнения XORW2 X, Y
X = 1101 1110 0001 0001
Y = 0001 1011 0111 1001 После выполнения XORW2 X, Y

Рекомендуем обратить внимание на сходство и различие двух семейств BIS и XOR, инструкции семейства BIS выполняют операцию **ИЛИ**, тогда как инструкции семейства XOR выполняют операцию **ИСКЛЮЧАЮЩЕЕ ИЛИ**.

Семейство инструкций XOR представлено ниже полностью:

Мнемоника	Код операции	Операнды	Операция
XORB2 XORW2 XORL2	8C AC CC	mask, dst	dst <- dst XOR mask
XORB3 XORW3 XORL3	8D AD CD	mask, src, dst	dst <- src XOR mask

ИНСТРУКЦИИ MCOM

Последнее множество инструкций побитовой обработки представляет семейство MCOM (Move COMplement – переслать дополнение), инструкции которого просто инвертируют все биты исходного операнда и запоминают их в операнде назначения. Другими словами, единичные биты заменяются нулевыми и наоборот. Например, инструкция MCOMW X, Y произведет следующее:

X = 1101 1110 0001 0001 До выполнения MCOMW X, Y,
Y = 0010 0001 1110 1110 После выполнения MCOMW X, Y.

Семейству MCOM аналогично семейство инструкций MNEG, используемое для изменения знака числа, представленного в дополнительном коде. Инструкции семейства MCOM могут использоваться для изменения знака числа, представленного в обратном коде, как описано в гл. 2. Однако в VAX, где числа со знаком представляются в дополнительном, а не в обратном коде, инструкции семейства MCOM обычно используются только для работы с битами. В действительности отрицательное число в дополнительном коде можно получить, прибавив единицу к представлению этого числа в обратном коде. Три инструкции этого семейства выглядят так:

Мнемоника	Код операции	Операнды	Операция
MCOMB MCOMW MCOML	88 A8 C8	src, dst	dst <- NOT src

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ПОБИТОВОЙ ОБРАБОТКИ

На рис. 8.4 и 8.5 показаны два программных сегмента, в которых используются инструкции побитовой обработки. В первом, на рис. 8.4, из длинного слова N извлекается третья шестнадцатеричная цифра, которая помещается при возврате управления как один байт символьного кода в регистр R0.

В программном сегменте на рис. 8.5 вычисляется то, что называют *битом четности*. Как отмечалось ранее, код ASCII, которым мы пользуемся, является 7-битовым. Однако для обнаружения ошибок при передаче символов кода ASCII между ЭВМ и устройствами ввода-вывода иногда добавляют восьмой бит. Этот бит называется *битом четности*, он устанавливается в 1 или 0 так, чтобы при этом общее число единиц в 8-битовом коде стало четным. Это называют *проверкой на четность* (even parity). Если теперь из-за помех или неисправности произойдет изменение одной из единиц в нуль или одного из нулей в единицу, общее число единиц станет нечетным (*проверка на нечетность* – odd parity), и эта ошибка обнаруживается. (*Замечание.* В некоторых системах производится проверка на нечетность, и тогда четное число единиц означает ошибку. Технические и программные средства ЭВМ VAX позволяют осуществлять связь с терминалами, производящими проверку на четность, нечетность или не выполняющими проверки по четности вообще. Поскольку биты четности обрабатываются техническими и программными средствами ЭВМ VAX без участия пользователя, пользователям обычно даже нет нужды знать об их существовании.

В программном сегменте на рис. 8.5 7-битовый код ASCII выбирается из регистра R0 и определяется, является ли нечетным или четным общее число единиц в этом коде. Битом четности является дополнительный бит в позиции 7, с помощью которого производится проверка байта на нечетность.

В последнем примере используются различные особенности инструкций побитовой обработки и сдвига при работе с символьными строками. Эта программа (показанная

```
N:          .BLKL   21
C_DIG_3:    MOVL    N,R0                ;ПОМЕСТИТЬ ЧИСЛО В R0
            BICL    #^XFFFFF0FF,R0      ;ОЧИСТИТЬ ВСЕ РАЗРЯДЫ, КРОМЕ РАЗРЯДА 3
            ASHL    #-8,R0,R0           ;СДВИНУТЬ ВПРАВО НА 8
            BISB    #^X30,R0            ;ВСТАВИТЬ КОД ASCII
            CMPB    R0,#^X39            ;ДЕСЯТИЧНАЯ ЦИФРА ?
            BLEQ    10$                 ;ДА, ОСТАВИТЬ КАК ЕСТЬ
            ADDB2   #^X41-^X30-10,R0    ;НЕТ, ПРЕОБРАЗОВАТЬ В ШЕСТНАДЦАТЕРИЧНУЮ
10$         RSB                         ;ВОЗВРАТ ИЗ ПОДПРОГРАММЫ
```

Рис. 8.4. Программа для распечатки третьей шестнадцатеричной цифры слова


```
;
;ПОДПРОГРАММА ODDGN: СОПРОВОЖДЕНИЕ БАЙТА КОНТРОЛЬНЫМ БИТОМ ЧЕТНОСТИ
;
MASK:   .BLKB   1
TEST:   .BLKB   1
ODDGN:  BICB    #^X80,R0            ;СБРОСИТЬ БИТ ЧЕТНОСТИ
        MOVB    #1,MASK             ;СФОРМИРОВАТЬ ПЕРВУЮ МАСКУ
        MOVL    #7,R1               ;В КОДЕ 7 БИТ
        CLRB    TEST                ;TEST БУДЕТ ПОКАЗЫВАТЬ ЧЕТНОСТЬ
10$:    BITB    MASK,R0             ;ПРОВЕРИТЬ ЗНАЧЕНИЕ БИТА
        BEQL    20$                 ;ПРОПУСТИТЬ, ЕСЛИ 0
        INCB    TEST                ;УВЕЛИЧИТЬ TEST НА ЕДИНИЦУ
20$:    ASHB    #1,MASK,MASK        ;СДВИНУТЬ БИТ МАСКИ
        SOBGTR  R1,10$              ;ОТСЧИТАТЬ 7 БИТ
        BITB    #1,TEST             ;ПРОВЕРИТЬ ЧЕТНОСТЬ
        BNEQ    30$                 ;ЕСЛИ КОД ЧЕТНЫЙ, ЗАВЕРШИТЬ ОБРАБОТКУ
        BISB    #^X80,R0            ;ИНАЧЕ УСТАНОВИТЬ БИТ ЧЕТНОСТИ
30$:    RSB                         ;ВОЗВРАТ УПРАВЛЕНИЯ, РЕЗУЛЬТАТ В R0
```

Рис. 8.5. Подпрограмма формирования байта с проверкой по нечетности

```
;
;ЧТЕНИЕ ДЕСЯТИЧНОГО ЧИСЛА, РЕЗУЛЬТАТ ПОМЕЩАЕТСЯ В R0
;
BSIZE=80
BUFFER: .BLKB   BSIZE
DATA:   .BLKB   1
READ:   MOVAL   BUFFER,R0           ;ВВЕСТИ СТРОКУ ТЕКСТА
        MOVZWL  #BSIZE,R1           ;С ПОМОЩЬЮ RLINE
        JSB     RLINE               ;В R1 БУДЕТ ДЛИНА СТРОКИ
        MOVAL   BUFFER,R2           ;ПОЛУЧИТЬ АДРЕС ВХОДНОЙ СТРОКИ
        CLRL    DATA                ;НАЧАЛЬНЫЙ РЕЗУЛЬТАТ 0
10$:    DECL    R1                  ;ПРОВЕРИТЬ ЧИСЛО СИМВОЛОВ
        BLSS    20$                 ;ЕСЛИ ДОСТИГНУТ КОНЕЦ СТРОКИ, ЗАКОНЧИТЬ
        MOVB    (R2)+,R0            ;ПОЛУЧИТЬ СИМВОЛ
        BICL    #^XFFFFFFF0,R0      ;ОТСЕЧЬ ЛИШНИЕ БИТЫ
        MULL2   #10,DATA            ;DATA=DATA*10
        ADDL2   R0,DATA             ;СЛОЖИТЬ СО ЗНАЧЕНИЕМ ЦИФРЫ
        BRB     10$                 ;ПОЛУЧИТЬ СЛЕДУЮЩУЮ ЦИФРУ (ЕСЛИ ЕСТЬ)
20$:    MOVL    DATA,R0             ;ЗАПОМНИТЬ РЕЗУЛЬТАТ
        RSB                         ;ВОЗВРАТИТЬ УПРАВЛЕНИЕ
```

Рис. 8.6. Подпрограмма ввода десятичного числа

на рис. 8.6) считывает десятичное число без знака, ввод числа выполняется с помощью подпрограммы ввода строки RLINE. Значение десятичного числа помещается в ячейку DATA. Поскольку число вводимых символов не контролируется, результат может переполнять длинное слово. Не выполняется также и контроль ошибок. Если пользователь вводит с терминала символы, отличные от десятичных цифр, результат будет бессмысленным.

Обратите внимание на то, как работает эта программа. В начале работы содержимое ячейки DATA равно 0. Затем каждый раз, когда берется очередная цифра введенного числа, число в ячейке DATA умножается на десять и к нему прибавляется значение этой цифры. Предположим, например, что вводится число 573. Число в DATA есть 0, а первая цифра есть 5. Число из DATA, умноженное на 10, остается по-прежнему 0, плюс 5 дает 5. Следующая цифра – 7. Число в DATA, умноженное на 10, дает 50, прибавив 7, получаем 57. Последняя цифра – это 3. Умножение на 10 числа в DATA дает 570, а после прибавления 3 получаем 573. После достижения последней цифры числа процесс завершается.

Описание этого алгоритма можно найти в гл. 2. Примеры, показанные на рис. 8.4 – 8.6, могут быть переписаны более эффективно. Например, на рис. 8.6 инструкции

```
MULL2   #10, DATA       ;DATA=DATA*10
ADDL2   R0, DATA        ;СЛОЖИТЬ СО ЗНАЧЕНИЕМ ЦИФРЫ
```

можно заменить одной инструкцией EMUL, которая уже приводилась в гл. 6. Это и другие улучшения мы оставляем читателям в качестве упражнений.

ИНСТРУКЦИИ BLBS И BLBC

Как видно из примера с умножением в конце гл. 6 и примера на рис. 8.5, нередко возникает желание осуществить проверку младшего разряда числа. Поэтому с этой целью введены две специальные инструкции:

Мнемоника	Значение
BLBS	Перейти, если младший бит установлен (Branch if Low Bit is Set)
BLBC	Перейти, если младший бит сброшен (Branch if Low Bit is Clear)

Эти инструкции проверяют младший бит байта, слова, длинного слова или квадраслова; переход по некоторому адресу осуществляется, если бит установлен в 1 – в случае инструкции BLBS или если он сброшен в 0 – в случае инструкции BLBC. Обычно не имеет значения, является ли адресуемый элемент словом или байтом или чем-то иным, поскольку младший бит находится в младшем байте, а байты, слова, длинные слова и квадраслова адресуются заданием адреса их младшего байта. (Однако, если используется для индексации регистр или режим адресации с автоувеличением, операнд рассматривается как длинное слово в том смысле, что содержимое регистра умножается или увеличивается на четыре). Пример использования инструкции BLBS: по инструкции

```
BLBS    X, 50$
```

будет осуществлен переход на метку 50$, если самый младший бит операнда X равен 1. Инструкция BLBS может быть применена в программе на рис. 8.5, что даст экономию одной инструкции. Заметим, что последние четыре инструкции программы таковы:

```
        BITB    #1, TEST
        BNEQ    30$
        BISB    #^X80, R0
30$:    RSB
```

Их можно изменить так:

```
        BLBS    TEST, 30$
        BISB    #^X80, R0
30$:    RSB
```

И это экономит одну инструкцию. Обратите внимание, что в этой инструкции применяется 8-битовое смещение, ограничивающее диапазон переходов так же, как в других инструкциях переходов.

Другое применение этих инструкций связано с кодами системных ошибок, описанными в разделе, который посвящен подпрограмме RLINE. Там отмечалось, что кодом нормального завершения является 1. Однако все другие коды ошибок представляют собой четные числа. Это позволяет использовать инструкции BLBS и BLBC для проверки ошибок.

Таким образом, текст программы контроля ошибок, показанный на стр. 171, может быть упрощен следующим образом:

Первоначальный вариант

```
        . . .
        JSB     RLINE
        CMPW    R0, #1
        BEQL    10$
        $EXIT_S R0
10$:    . . .
```

Вариант с использованием BLBS

```
        . . .
        JSB     RLINE
        BLBS    R0, 10$
        $EXIT_S R0
10$:    . . .
```

8.5. ДРУГИЕ СПОСОБЫ ПРЕДСТАВЛЕНИЯ СИМВОЛЬНОЙ ИНФОРМАЦИИ

КОД ХОЛЛЕРИТА

Хотя наиболее распространенным символьным кодом, применяемым на ЭВМ VAX, является код ASCII, имеет смысл упомянуть и другие способы кодирования. Особое значение имеет кодирование, применяемое в перфокарточном оборудовании. Оно известно также как *код Холлерита*, получивший свое название в честь Германа Холлерита, который ввел в употребление перфокарты для табуляторов во время переписи населения США в 1890 г.

Носителем информации, представленной в коде Холлерита, является бумажная карта, совпадающая по высоте и ширине с долларовой банкнотой тех времен. На карте (см. рис. 8.7) могут пробиваться прямоугольные отверстия в любой из 80 позиций (колонок) по горизонтали и в любой из 12 позиций (рядов) по вертикали.

Каждая колонка предназначена для перфорации одного символа; следовательно, перфокарта может содержать 80 закодированных символов. Поскольку имеется 12 рядов и так как предположительно может быть составлена любая возможная комбинация из 12 пробивок, перфокарточный код способен вместить алфавит, содержащий до $2^{12} = 4096$ различных символов. На практике, однако, если пробить слишком много отверстий, перфокарта становится похожа на сетку и теряет физическую жесткость. Это вынудило ограни-

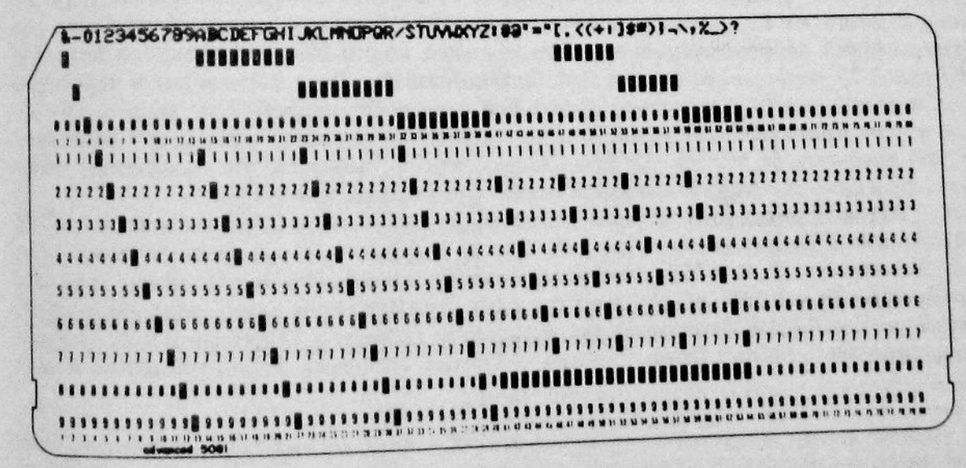


Рис. 8.7. Перфокарта Холлерита

A	1–12	J	1–11	Не использ.	1–0
B	2–12	K	2–11	S	2–0
C	3–12	L	3–11	T	3–0
D	4–12	M	4–11	U	4–0
E	5–12	N	5–11	V	5–0
F	6–12	O	6–11	W	6–0
G	7–12	P	7–11	X	7–0
H	8–12	Q	8–11	Y	8–0
I	9–12	R	9–11	Z	9–0

Рис. 8.8. Код Холлерита для алфавита

чить кодовые комбинации лишь такими, в которых обычно в одной колонке содержится не более трех пробивок[1].

Ряды перфокарты нумеруются сверху вниз в следующем порядке: 12, 11, 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Коды цифр от 0 до 9 представляются одной пробивкой в соответствующем ряду (т.е. код цифры 5 – это пробивка в ряду 5, или *пробивка пять*). Коды букв формируются из комбинации одной пробивки в рядах 1–9 (*цифровая пробивка*) и второй пробивки в ряду 12, 11 или 0 (*зонная пробивка*). Это дает $9 \times 3 = 27$ возможных комбинаций, из которых используются 26. Таблица кодов алфавита дана на рис. 8.8. Обратите внимание, что единственный незадействованный код 0–1 расположен в середине таблицы. Холлерит не задействовал этот код, поскольку опасался, что его машина могла бы порвать перфокарты с двумя пробивками в соседних рядах. Помимо этого одиночные пробивки в рядах 12 и 11 и отсутствие перфорации вообще используются для представления знаков &, –, пробела.

СОВРЕМЕННЫЕ ПЕРФОКАРТОЧНЫЕ КОДЫ

По мере усложнения обработки данных возникла потребность в большем разнообразии знаков пунктуации. Она была удовлетворена путем расширения поля цифровых пробивок с 9 до 15 при использовании пробивки 8 в комбинации с одной из пробивок от 2 до 7. Например, перфорация 8–5 имеет числовое значение 13. Кроме того, более развитое оборудование позволило использовать комбинацию 0–1. Это обеспечило возможность представления 64 символов.

Соответствие расположения пробивок знакам пунктуации весьма произвольно, и фактически существуют несколько различных назначений. На рис. 8.9 показан наиболее распространенный перфокарточный код. Его называют **кодом 029**, поскольку он появился на модели 29 перфоратора фирмы IBM. Однако название "код 29" вводит в некотором смысле в заблуждение, поскольку фирма IBM производит перфораторы модели 29 с целым рядом различных назначений. В частности, приведенное назначение кодов свойственно для перфоратора модели 029-ЕН. Следует, однако, заметить, что назначения кодов для цифр и букв стандартны, меняются только коды знаков пунктуации.

Код 029, содержащий 64 символа, является подмножеством 8-битового кода EBCDIC (расширенный двоично-десятичный код обмена информацией). Восемь битовых позиций обеспечивают 2^8, или 256, возможных кодовых комбинаций. В дополнение к 64 символам кода 029 код EBCDIC содержит строчные буквы, а также целый ряд управляющих символов. Код EBCDIC показан на рис. 8.10 в виде таблицы с 32 рядами и 8 колонками. Ряд определяет 5 правых битов, колонка – 3 левых. Например, прописная буква A зани-

[1] Из этого правила существует несколько исключений, касающихся перфокарт специального назначения, таких как перфокарты конца файла, перфокарты с двоичным кодом, а также некоторых кодов расширенного алфавита, которые могут включать строчные буквы и управляющие символы.

Пробел	Нет пробивки	&	12	–	11–0	0			
1	1	A	12–1	J	11–1	/		0–1	
2	2	B	12–2	K	11–2	S		0–2	
3	3	C	12–3	L	11–3	T		0–3	
4	4	D	12–4	M	11–4	U		0–4	
5	5	E	12–5	N	11–5	V		0–5	
6	6	F	12–6	O	11–6	W		0–6	
7	7	G	12–7	P	11–7	X		0–7	
8	8	H	12–8	Q	11–8	Y		0–8	
9	9	I	12–9	R	11–9	Z		0–9	
:	8–2	¢	12–8–2	!	11–8–2	Не использ.		0–8–2	
#	8–3	.	12–8–3	$	11–8–3	,		0–8–3	
@	8–4	<	12–8–4	*	11–8–4	%		0–8–4	
'	8–5	(	12–8–5	)	11–8–5	–		0–8–5	
=	8–6	+	12–8–6	;	11–8–6	>		0–8–6	
"	8–7	|	12–8–7	¬	11–8–7	?		0–8–7	

Рис. 8.9. Код перфоратора 029-ЕН фирмы IBM

Младшие 5 битов	Старшие 3 бита							
	000	001	010	011	100	101	110	111
00000	NUL	DS	Sp					
00001	SOH	SOS			a		A	
00010	STX	FS			b	s	B	S
00011	ETX				c	t	C	T
00100	PF	BYP			d	u	D	U
00101	HT	LF			e	v	E	V
00110	LC	ETB			f	w	F	W
00111	DEL	ESC			g	x	G	X
01000					h	y	H	Y
01001					i	z	I	Z
01010	SMM	SM	¢					
01011	VT	CU2	.	.				
01100	FF		<	%				
01101	CR	ENQ	(	–				
01110	SO	ACK	+	>				
01111	SI	BEL	|	?				
10000	DLE		&					0
10001	DC1				j		J	1
10010	DC2	SYN			k		K	2
10011	TM				l		L	3
10100	RES	PN			m		M	4
10101	NL	RS			n		N	5
10110	BS	UC			o		O	6
10111	IL	EOT			p		P	7
11000	CAN				q		Q	8
11001	EM				r		R	9
11010	CC		!	:				
11011	CU1	CU3	$	#				
11100	IFS	DC4	*	@				
11101	IGS	NAK	)	'				
11110	IRS		;	=				
11111	IUS	SUB	¬	"				

Рис. 8.10. Код EBCDIC

мает вторую позицию в 7-й колонке. Таким образом, двоичный код буквы A состоит из битов 110 (из колонки), за которыми следуют биты 00001 (из ряда), что дает код 11000001. Обратите внимание, что многие комбинации не назначены каким-либо символам или управляющим символам. Код EBCDIC является стандартным на больших ЭВМ фирмы IBM, а также на ЭВМ, выпускаемых другими производителями. Промежутки между кодами букв I и J и кодами букв R и S предназначены для обеспечения совместимости с использовавшимися ранее кодами, являющимися производными от перфокарточного кода Холлерита.

УПРАЖНЕНИЯ 8.2

1. Задано, что содержимым ячеек A, B, C и D является следующее:

```
A   ^X0F0F0F0F
B   ^XF0F0F0F0
C   ^X12345678
D   ^X87654321
```

Что произойдет с содержимым ячеек и кодами условий N, Z, C и V в результате выполнения следующих инструкций над первоначальным содержимым этих ячеек?

```
а.  BISCL2  A,B        б.  BISCL2  A,C
в.  BISCL2  C,A        г.  BISCL2  B,C
д.  BITL2   C,D        е.  BITL2   C,C
```

2. Напишите и выполните программу, которая осуществляет ввод символов с терминала и распечатывает каждый символ семь раз на строчке. Программа завершает работу, когда считывает пустую (нулевой длины) строку.

3. Напишите программу, которая считывает строку и затем распечатывает ее попеременно в прямом и обратном порядке семь раз. Длина строки может изменяться, но не должна превышать 80 символов.

4. Напишите программу, предназначенную для выполнения в режиме пакетной обработки, которая считывает символы с перфокарты. Она распечатывает символы, заменяя все управляющие символы (включая "Возврат каретки" и "Перевод строки") символом "звездочка" и литерой, клавишу которой надо было бы нажать вместе с клавишей управления CTRL для получения данного управляющего символа. Каждую комбинацию CTRL/J, или *J, сопроводите символами "Возврат каретки" и "Перевод строки". Какие управляющие символы вставляются системой пакетной обработки в ваши данные? Можно ли управляющие символы перфорировать на перфокартах? Если да, то как?

5. Напишите программу, которая считывает 32-битовые двоичные числа как последовательности из 32 единиц и нулей в коде ASCII, сопровождаемой символами возврата каретки и перевода строки. Подпрограмма передает двоичное значение через регистр R0.

6. Напишите подпрограмму, которая берет значение из регистра R0 и распечатывает его как 32-битовое двоичное число. Печатайте по одному числу на строке при каждом вызове подпрограммы.

7. Скомбинируйте подпрограммы из пп. 5 и 6 с основной программой, которая проверяет работу подпрограмм, вызывая их несколько раз.

8. Напишите подпрограмму, которая распечатывает десятичные числа со знаком. Модифицируйте подпрограмму на рис. 8.6 для считывания десятичных чисел со знаком. Напишите основную программу, которая проверяет эти подпрограммы, вызывая их несколько раз. Подпрограмма вывода на печать должна обеспечивать подавление незначащих нулей в начале числа.

9. Перепишите программы на рис. 8.4, 8.5 и 8.6 так, чтобы они стали более эффективными, воспользовавшись следующими рекомендациями в дополнение к любым другим, которые вы в состоянии придумать:

а) на рис. 8.4 задачу решите с помощью индексации массива из шестнадцати символов из строки "0123456789ABCDEF";

б) на рис. 8.5 используйте инструкции BLBS и BLBC для проверки битов в байте. Если важнее быстродействие, чем занимаемая память, то можно ли получить выигрыш, используя таблицу из 256 байтов?

в) на рис. 8.6 воспользуйтесь инструкцией EMUL для замены инструкций MULL2 и ADDL2.

10. Каждый из следующих программных сегментов выполняет некоторую простую операция. Однако применяются логические инструкции и выполняемая операция не обязательно очевидна. Определите, что выполняется в каждом сегменте?

```
а.  XORL2   A,B
    XORL2   B,A
    XORL2   A,B
```

(сопоставьте новые значения A и B с первоначальными);

```
б.  MOVL    A,B
    XORL2   B,B
```

(значение B);

```
в.  MOVL    A,B
    DECL    B
    COML    B
```

(сопоставьте B и A);

```
г.  MOVL    A,B
    XORL2   #-1,B
    INCL    B
```

(сопоставьте B и A);

```
д.  10$:    XORL3   A,B,C
            MCOML   A,A
            BICL2   A,B
            MOVL    C,A
            ASHL    #1,B,B
            BNEQ    10$
```

(сопоставьте C с первоначальным значением A и B).

*11. Напишите программу, которая считывает некоторый фрагмент текста. Затем программа распечатывает число печатных символов, встретившихся в тексте. Числа должны распечатываться в десятичном представлении с помощью такой подпрограммы, какая была написана для п. 8. Ваша распечатка должна выглядеть примерно так:

```
A  ВСТРЕТИЛОСЬ   129   РАЗ
B  ВСТРЕТИЛОСЬ    17   РАЗ
C  ВСТРЕТИЛОСЬ    18   РАЗ
```

ГЛАВА 9. ПОДПРОГРАММЫ

9.1. ВВЕДЕНИЕ

В предыдущих главах было показано использование простых подпрограмм для выполнения часто повторяющихся операций, таких как ввод или распечатка строк или чисел. Подпрограммы играют очень важную роль в структуре машинных программ. В настоящей главе будет детально рассмотрено, как осуществляются вызов и возврат из подпрограмм. Читатель узнает также, как подпрограммы получают доступ к данным основной программы и как могут быть связаны вместе в сложную программную структуру отдельные подпрограммы и объединены с программами, написанными на языках высокого уровня. Последний вопрос, наиболее важен. Одно из самых важных применений языка ассемблера состоит в том, что он расширяет возможности таких языков, как Паскаль или Фортран, позволяя выполнять операции, которые трудно или невозможно реализовать только с помощью языка высокого уровня. Кроме того, в операционной системе VAX/VMS из основных программ, написанных на языке ассемблера, могут вызываться подпрограммы, написанные на языках высокого уровня. Это позволяет ассемблерным программам легко получать доступ к таким функциям языков высокого уровня, как форматный ввод и вывод. В настоящей главе будут обсуждены стандартизированные способы вызова, которые позволяют программам, написанным на различных языках, включая язык ассемблера, взаимодействовать друг с другом.

9.2. ВЫЗОВ ПОДПРОГРАММЫ

ИНСТРУКЦИЯ JSB

В гл. 5 было сказано, что подпрограмма может быть вызвана с помощью инструкции

```
JSB     ADDR
```

Возврат из подпрограммы к основной программе выполняется по инструкции

```
RSB
```

В пояснении, данном в гл. 5, говорилось, что инструкция JSB использует указатель стека (регистр 14, или SP) для сохранения программного счетчика (регистр 15, или PC) в области памяти, называемой стеком. Значение программного счетчика, сохраняемое в стеке, есть адрес инструкции, следующей за инструкцией JSB. Это та инструкция, которая будет выполняться после возврата из подпрограммы.

Пояснение верно, но не полно. Например, мы уже встречались с программами, в которых одни подпрограммы вызывали другие. При вложенных вызовах подпрограмм необходимо сохранять весь массив адресов возврата. Тогда возникает вопрос, каким образом каждая инструкция возврата RSB получает из стека правильный адрес возврата?

ИСПОЛЬЗОВАНИЕ СТЕКА

Чтобы можно было использовать массив для хранения адресов возврата, необходим указатель, показывающий, в каком месте этого массива запоминаются данные. Для этой цели всегда применяется регистр общего назначения 14, или SP (Stack Pointer – указатель стека). Теперь читателю должно быть ясно, почему были сделаны предостережения против использования регистра 14. Произвольное применение этого регистра может приводить к тому, что при следующем вызове подпрограммы или возврате из подпрограммы будут происходить очень странные вещи.

Массив, доступ к которому осуществляется путем последовательного добавления и удаления данных описанным ниже способом, называют *стеком*. Действие стека может быть

продемонстрировано на примере стопки тарелок в кафе. Чтобы сохранить элемент информации, откуда-нибудь берут новую тарелку, пишут на ней 32-битовое двоичное число и помещают тарелку на вершину стопки тарелок (стека). Чтобы извлечь элемент из стойки (стека), с вершины стопки удаляют тарелку и смотрят, какое число на ней записано. Обратите внимание, что тарелки добавляются и удаляются всегда на *вершине* стопки (стека). Поэтому удаляемая тарелка – это всегда та, которая была добавлена последней. Например, ниже показано, как три элемента, помеченные буквами A, B и C, сначала сохраняются в стеке, а затем удаляются из него:

Пустой стек	A Добавление элемента	B A Добавление элемента	C B A Добавление элемента	B A Удаление верхнего элемента	A Удаление верхнего элемента	Удаление верхнего элемента

Хотя последним в стек был помещен элемент C, он – первый кандидат на удаление из стека. Такая последовательность "последним вошел – первым вышел" часто обозначается сокращением LIFO (Last In, First Out).

Регистр 14 называется *указателем стека,* поскольку он указывает на вершину стека. Например, предположим, что первое длинное слово стека хранится в памяти по адресу ^X7FFC8FFC. Поскольку первое длинное слово будет заноситься по адресу ^X7FFC8FFC, то регистр 14 будет содержать адрес ^X7FFC9000, когда стек пуст:

	Адрес	Содержимое
	^X7FFC8FF4	??????
	^X7FFC8FF8	??????
	^X7FFC8FFC	??????
регистр 14 ^X7FFC9000	→^X7FFC9000	??????

Чтобы сохранить в стеке длинное слово, нужно вычесть число 4 из содержимого регистра 14 и записать элемент по полученному в результате вычитания адресу. Если в стеке сохраняется длинное слово A, то результат будет таким:

	Адрес	Содержимое
	^X7FFC8FF4	??????
	^X7FFC8FF8	??????
регистр 14 ^X7FFC8FFC	→^X7FFC8FFC	A
	^X7FFC9000	??????

Чтобы сохранить в стеке длинное слово B, вычитаем число 4 из содержимого регистра 14 и помещаем длинное слово B по полученному адресу:

	Адрес	Содержимое
	^X7FFC8FF4	??????
регистр 14 ^X7FFC8FF8	→^X7FFC8FF8	B
	^X7FFC8FFC	A
	^X7FFC9000	??????

Извлечение элемента стека совершается в обратном порядке по отношению к сохранению элемента. Извлекаем содержимое ячейки памяти, адрес которой содержится в ре-

гистре 14 и затем прибавляем число 4 к содержимому регистра 14. После извлечения длинного слова B стек будет выглядеть так:

	Адрес	Содержимое
	^X7FFC8FF4	??????
	^X7FFC8FF8	??????
регистр 14 ^X7FFC8FFC	→^X7FFC8FFC	A
	^X7FFC9000	??????

Обратите внимание, что длинное слово с адресом ^X7FFC8FF8, которое использовалось для хранения B, теперь содержит ??????. Ячейки памяти, используемые для хранения помещаемых в стек элементов, после извлечения элемента из стека могут содержать бессмысленную информацию. Программист должен исходить из того, что это отвечает истине на большинстве (если не на всех) вычислительных систем. Конкретная реализация стека может включать аппаратную или программную перезапись ячеек памяти стека после извлечения элементов.

Удаление еще одного элемента из стека приводит к извлечению элемента A и опустошению стека.

	Адрес	Содержимое
	^X7FFC8FF4	??????
	^X7FFC8FF8	??????
	^X7FFC8FFC	??????
регистр 14 ^X7FFC9000	→^X7FFC9000	??????

Поскольку адреса возврата сохраняются в стеке, подпрограммы могут вызываться одна из другой без затруднений. Например, основная программа с именем ALPHA может вызывать подпрограмму под названием BETA. В свою очередь, подпрограмма BETA может вызывать подпрограмму GAMMA, а та может вызывать подпрограмму DELTA. К моменту, когда выполнится подпрограмма DELTA, в стеке сохранено три адреса возврата – адреса возврата к ALPHA, BETA и GAMMA. Благодаря организации стека по принципу "последним вошел – первым вышел" адресом возврата, который будет извлечен из стека после завершения подпрограммы DELTA, окажется адрес возврата к подпрограмме GAMMA. Аналогично подпрограмма GAMMA возвратит управление подпрограмме BETA, а подпрограмма BETA возвратит управление программе ALPHA. Как будет показано далее в этой главе, использование стека для хранения адресов возврата позволяет подпрограмме даже вызывать саму себя. Такие подпрограммы называются *рекурсивными.*

Длинное слово может быть помещено в стек по инструкции

```
MOVL    X,-(SP)
```

Аналогично извлечение длинного слова из стека может быть сделано по инструкции

```
MOVL    (SP)+,X
```

Фактически пользователи могут применять эти две инструкции для помещения данных в стек, используя тем самым стек для сохранения временных данных. Представим себе для примера подпрограмму, которая использует регистры R2, R3 и R4 для внутренних вычислений. Но предположим, что основная программа пользуется теми же регистрами и нежелательно, чтобы подпрограмма изменяла их содержимое. Тогда подпрограмма должна сохранять содержимое регистров R2, R3 и R4 и восстанавливать его при возврате управления. Для этой цели обычно используют стек. Хотя для сохранения регистра можно воспользоваться почти любой доступной ячейкой, применение стека дает важные преимущества. Когда стек используется в качестве временного хранилища, ячейки памяти в стеке занимаются лишь до удаления элементов из стека, после чего эта память становится доступной для использования в других целях. Экономятся также память и время, так как инструкция MOVL R2,-(SP) является короткой, а инструкция MOVL R2, SAVE – длинной, поскольку в нее входит адрес. По этим и другим причинам, которые станут яснее позже, хороший стиль программирования на ЭВМ VAX требует, чтобы для указанных целей использовался стек.

На рис. 9.1 показана подпрограмма, которая сохраняет в стеке содержимое регистров R2, R3 и R4. Обратите внимание на метку RETN. Чтобы осуществить возврат, необходимо либо произвести переход на эту метку, либо выполнить инструкции, дающие такой же эффект. Выполнение инструкции возврата из подпрограммы RSB без восстановления содержимого регистров привело бы к катастрофе. Инструкция RSB берет в качестве адреса возврата верхний элемент стека. Если же регистры не были восстановлены, то вершина стека будет содержать сохраненное содержимое регистра R4, а *не* адрес возврата. Общее правило состоит в том, что подпрограммы могут использовать стек для сохранения данных, *но* всякий раз, когда в стек что-либо добавляется, требуется не больше и не меньше, как удалить это из стека до выполнения инструкции RSB.

В действительности включение данных в стек и особенно сохранение в стеке содержимого регистров имеет столь важное значение, что существует несколько инструкций, специально предназначенных для облегчения таких операций. Первая из них – это инструкция PUSHL (PUSH Long – включить в стек длинное слово):

```
PUSHL   X
```

Фактически эта инструкция дает тот же самый эффект, что и инструкция

```
MOVL    X,-(SP)
```

Однако инструкция PUSHL яснее, работает несколько более эффективно и занимает на один байт меньше памяти. По этим причинам она является предпочтительной.

Две другие инструкции работы со стеком предназначаются для одновременного сохранения в стеке содержимого сразу нескольких регистров общего назначения. Такие инструкции ссылаются на регистры с помощью 16-битовой маски, осуществляющей выбор подмножества из 16 регистров общего назначения. Это инструкции

```
  PUSHR   MASK
и
  POPR    MASK
```

```
START:  MOVL    R2,-(SP)        ;СОХРАНИТЬ СОДЕРЖИМОЕ
        MOVL    R3,-(SP)        ;РЕГИСТРОВ R2, R3 И R4
        MOVL    R4,-(SP)        ;В СТЕКЕ
        .
        .
        .
RETN:   MOVL    (SP)+,R4        ;ВОССТАНОВИТЬ РЕГИСТРЫ
        MOVL    (SP)+,R3        ;ОТМЕТЬТЕ ОБРАТНЫЙ ПОРЯДОК
        MOVL    (SP)+,R2        ;ПОСЛЕДНИЙ ИЗВЛЕКАЕТСЯ ПЕРВЫМ
        RSB                     ;ВОЗВРАТ ИЗ ПОДПРОГРАММЫ
```

Рис. 9.1. Использование стека для сохранения содержимого регистров

Операнд, обозначенный как MASK, – 16-битовое слово. Каждый бит в слове определяет некоторый регистр общего назначения. Бит 0 относится к регистру R0, бит 1 – к регистру R1, и т.д. вплоть до бита 14, соответствующего регистру 14 (SP). Поскольку регистр 15 (PC) является программным счетчиком, он обычно в стеке не сохраняется, исключение составляет только инструкция вызова подпрограмм. По этой причине бит 15 слова маски игнорируется и всегда рассматривается как нулевой этими инструкциями. Кроме того, особое внимание следует уделить регистрам AP, FP и SP (регистры 12, 13 и 14 соответственно). Попытка сохранить содержимое этих регистров в стеке и непосредственно восстановить их значения из стека, вероятно, необратимо нарушит поток управления в программе. Эти регистры имеют специальное назначение, и если необходимо их сохранение, то оно должно делаться специально предписанными способами. Более полные пояснения будут даны далее в разделе, посвященном инструкциям CALLS и CALLG настоящей главы.

Инструкция PUSHR работает таким образом, что в стек помещается содержимое каждого регистра, для которого соответствующий бит в маске равен 1. Например, инструкция

```
PUSHR   #^X0047
```

приведет к сохранению в стеке содержимого регистров R0, R1, R2 и R6, так как число ^X0047 имеет единицы в разрядах 0, 1, 2 и 6, что видно из его двоичного представления:

0000 0000 0100 0111

Следовательно, инструкция PUSHR #^X0047 эквивалентна следующей инструкции:

```
PUSHL   R6
PUSHL   R2
PUSHL   R1
PUSHL   R0
```

Обратите внимание, что содержимое регистров включается в стек в порядке, обратном номерам регистров. Инструкция POPR делает противоположное тому, что делает инструкция PUSHR. Следовательно, инструкция

```
PUSHR   #^X0047
```

эквивалентна совокупности инструкций

```
MOVL    (SP)+,R0
MOVL    (SP)+,R1
MOVL    (SP)+,R2
MOVL    (SP)+,R6
```

Очевидная трудность при использовании этой инструкции связана с неудобством задания слова маски. Для упрощения этой процедуры ассемблер VAX MACRO располагает специальным оператором для генерации масок. Это оператор ^M. Он выглядит так:

^M⟨список регистров⟩

Список регистров представляет собой просто список имен регистров, разделенных запятыми. При использовании этого оператора инструкция PUSHR #^X0047 может быть переписана так:

```
PUSHR   #^M<R0,R1,R2,R6>
```

Ясно, что такой вид более предпочтителен, так как регистры, подлежащие сохранению или восстановлению, здесь идентифицируются явным образом.

Следует заметить, что до сих пор примеры работы со стеком затрагивали включение и извлечение длинных слов. И хотя возможно использовать стек для хранения данных любого формата, например для байтов или слов, ЭВМ семейства VAX работают значительно эффективнее, если указатель стека устанавливается на границе длинного слова. Иначе говоря, значение указателя стека должно быть кратно числу 4. Поскольку система всегда начинает выполнение программы со значением указателя стека, установленным на границе длинного слова, то это значение будет оставаться на границе длинного слова, если в стек будут включаться или из стека извлекаться только длинные слова. И тем не менее все же существует возможность использования стека для хранения байтов или слов с сохранением выравнивания на границе длинного слова. Осуществить это можно, воспользовавшись инструкциями преобразования формата данных и инструкциями пересылки с добавлением незначащих нулей. Например, байт можно сохранить в стеке с помощью инструкции

```
MOVZBL  BYTE,-(SP)
```

Байт может быть восстановлен инструкцией

```
CVTLB   (SP)+,BYTE
```

Эти инструкции описаны в гл. 6. Обратите внимание, режимы с автоувеличением и автоуменьшением используются с операндом в формате длинного слова, так что указатель стека увеличивается или уменьшается на четыре.

Поскольку стек – это просто некоторый массив в памяти, то возникает вопрос о том, где же он находится: все зависит от операционной системы. В ОС VAX/VMS программы обычно загружаются со стартового адреса ^X00000200. Стек начинается с одного из старших адресов, такого как ^X7FFC9000, и располагается в памяти по направлению в сторону уменьшения адресов. Действительный стартовый адрес определяется операционной системой VAX/VMS.

ПЕРЕДАЧА ИНФОРМАЦИИ МЕЖДУ ОСНОВНОЙ ПРОГРАММОЙ И ПОДПРОГРАММАМИ

В большинстве случаев функция подпрограммы состоит в выполнении некоторых вычислений над одним или несколькими данными. Результаты этих вычислений могут также быть представлены как одно или несколько данных. Эти данные должны передаваться от основной программы подпрограмме и обратно.

В гл. 5 для подпрограмм RNUM и PNUM использовался очень простой способ связи. Число, передаваемое между подпрограммами, помещалось в регистр R0. Этот способ применим тогда, когда между подпрограммами надо передать лишь несколько (не более чем 12) длинных слов информации. Первое длинное слово помещается в регистр R0, второе – в регистр R1, и т.д. вплоть до регистра R11, если необходимо. Подпрограмма должна составляться так, что ищет информацию в соответствующем регистре. Таким же образом можно вернуть до 12 результатов.

Очевидно, что этот способ передачи информации налагает серьезные ограничения, когда приходится иметь дело с большим количеством данных или массивами. Ограничение объема передаваемых данных только 12 регистрами можно преодолеть путем использования специально выделенной области памяти, предназначенной для связи. Однако в действительности при больших объемах данных такой способ работает не очень хорошо, поскольку требуются загрузка и сохранение содержимого всех ячеек, в которых находятся эти данные при каждом вызове подпрограммы, что может повлечь излишние затраты на вычисления.

Одно решение этой проблемы особенно пригодно при работе с массивами. Вместо того чтобы передавать значения всех элементов массива, основная программа передает адрес массива. Подпрограмма использует этот адрес для доступа к данным массива. Этот

```
SIZE=100
SUMUP:  PUSHR   #^M<R1,R2>      ;СОХРАНИТЬ R1 И R2
        MOVL    #SIZE,R2        ;В R2 - КОЛИЧЕСТВО ЧИСЕЛ
        CLRL    R0              ;ОБНУЛИТЬ СУММУ
1$:     ADDL2   (R1)+,R0        ;ПРИБАВИТЬ ОЧЕРЕДНОЕ ЧИСЛО
        DECL    R2              ;УМЕНЬШИТЬ СЧЕТЧИК
        BNEQ    1$              ;ПОВТОРЯТЬ В ЦИКЛЕ ДО ЗАВЕРШЕНИЯ
        POPR    #^M<R1,R2>      ;ВОССТАНОВИТЬ СОДЕРЖИМОЕ РЕГИСТРОВ
        RSB                     ;ВОЗВРАТ ИЗ ПОДПРОГРАММЫ
```

Рис. 9.2. Подпрограмма суммирования элементов массива

способ применялся в гл. 8 в подпрограммах RLINE и PLINE. В качестве еще одного примера использования этого способа на рис. 9.2 представлена подпрограмма, в которой выполняется сложение 100 чисел массива. Основная программа передает подпрограмме адрес массива через регистр R1. *Значение* результата возвращается обратно, для чего используется регистр R0. Таким образом, здесь применяется комбинация двух способов. Последовательность действий при вызове этой программы будет следующей:

```
MOVAL   ARRAY,R1
JSB     SUMUP
MOVL    R0,ANS
```

В данном случае ARRAY – это метка (символический адрес) массива чисел, подлежащих сложению, ANS – ячейка, в которую в конце концов помещается результат.

Специального замечания здесь заслуживает тот факт, что помимо уже отмеченных преимуществ передача адресов вместо данных обладает еще одним: она является намного более общим способом связи. Адреса могут использоваться двунаправленно. Другими словами, если подпрограмма располагает адресом ячейки в главной программе, она может воспользоваться этим адресом как для доступа к данным, так и для пересылки результатов назад к главной программе. Так было сделано в случае подпрограмм RLINE и PLINE. Подпрограмма RLINE возвращает данные, тогда как подпрограмма PLINE осуществляет доступ к данным.

Подводя итог, можно сказать, что передача значений – это быстрый и простой способ связи, который весьма пригоден для подпрограмм, имеющих дело только с несколькими данными. Передача адресов необходима для таких подпрограмм, которые работают с большим числом данных. Эти способы называют соответственно *вызовом по значению* и *вызовом по ссылке*. Позднее в этой главе читатель познакомится с еще более общим способом передачи данных, который называют *вызовом по дескрипторам*.

УПРАЖНЕНИЯ 9.1

Для пп. 1, 2 и 3 следует предположить, что регистр R0 содержит ^X00000256, регистр R1 – ^XFFFFF624, регистр R2 – ^XFFFFFFFC, а следующие ячейки памяти – приведенные ниже числа:

Адрес	Содержимое
^X7FFC8FF0	^X12345678
^X7FFC8FF4	^X000002FC
^X7FFC8FF8	^X00000005
^X7FFC8FFC	^X00004212
^X7FFC9000	^XFFFFFFFE

Также предполагается, что SP содержит ^X7FFC8FF8 и что первоначальное содержимое этого регистра было ^X7FFC9000. Значение PC есть ^X4F3. Адрес SUB равен ^X67F.

1. Какие значения содержатся в стеке?

2. Содержимое каких ячеек памяти изменится и каким станет новое содержимое этих ячеек после выполнения каждой из следующих инструкций (используйте значения, определенные ранее для каждой инструкции):

```
а.  PUSHL   R0              б.  MOVL    (SP)+,R1
в.  CMPL    (SP)+,(SP)+     г.  JSB     SUB
д.  JSB     @R0             е.  RSB
ж.  ADDL2   R2,SP           з.  MOVL    (SP),-(SP)
и.  PUSHR   #^M<R0,R1>      к.  POPR    #^M<R0,R1>
```

3. Какие из следующих инструкций представляют обычное использование стека? Какие являются неправильными и могут приводить к непредсказуемым результатам из-за того, что операционная система может периодически прерывать выполнение вашей программы и модифицировать ячейки, расположенные "ниже", чем ячейка, адрес которой содержится в указателе стека? Какие инструкции являются катастрофическими и с большой вероятностью вызовут аварийное завершение программы? Какие инструкции вызовут ошибки при ассемблировании? Поясните ваши ответы. Используйте значения, описанные в п. 1 для каждой инструкции, и покажите результирующие изменения содержимого.

```
а.  MOVL    R0,(SP)+        б.  CMPL    -(SP),-(SP)
в.  JSB     (SP)+           г.  JSB     @(SP)+
д.  JSB     SP              е.  INCL    SP
ж.  DECL    SP              з.  MOVL    4(SP),R0
и.  MOVL    (SP)+,PC
```

4. Напишите подпрограмму, аналогичную подпрограмме SUMUP, показанной на рис.9.2. Но ваша программа будет иметь второй параметр, передаваемый через регистр R2, задающий длину массива, подлежащего суммированию. Затем напишите основную программу, которая вводит 10 чисел, распечатывает их и вызывает вашу подпрограмму для суммирования этих чисел, после чего делает то же самое для 20 чисел.

5. Напишите программу, которая считывает переменное число чисел (не более 100), сортирует их и распечатывает отсортированный массив. Эту программу следует разбить на различные подпрограммы, по одной подпрограмме на каждую основную функцию. Адреса массивов, их длина и все другие данные должны передаваться через регистры общего назначения при каждом вызове подпрограммы.

9.3. РАЗДЕЛЬНОЕ АССЕМБ[illegible]ОВАНИЕ И ГЛОБАЛЬНЫЕ ИМЕНА

НЕОБХОДИМОСТЬ РАЗДЕЛЬНОГО АССЕМБЛИРОВАНИЯ

При написании какой-либо довольно большой программы весьма важно разбить эту программу на ряд фрагментов приемлемого размера, называемых *модулями*. Чем более независимы такие модули, тем легче писать, тестировать и отлаживать каждый из них. Отлаживать всю программу целиком разумно лишь тогда, когда будет отлажен каждый ее модуль.

Чтобы программировать таким образом, важно иметь возможность обработки каждого модуля как отдельной программы. Это означает, что каждая из этих небольших программ должна обеспечивать возможность ее самостоятельного ассемблирования. Следовательно, как и в любой другой программе, все используемые программой ячейки должны полностью определяться внутри модуля. Иначе говоря, не должно быть каких-либо неопределенных имен.

С другой стороны, эти программные модули предназначены для того, чтобы, сочетаясь между собой, образовывать единую программу. Это означает, что, хотя модули могут

быть весьма независимыми, они не могут быть независимы полностью. Значит, должен существовать некоторый вид связи между программами. Такая связь выполняется с помощью *глобальных имен.*

ГЛОБАЛЬНЫЕ ИМЕНА

Глобальное имя – это символический адрес, который определяется в одной программе, но доступен и из других раздельно ассемблируемых программ. Хотя глобальные имена могут использоваться почти для всех целей, для которых используются обычные имена, чаще всего они применяются для имен подпрограмм.

Например, программный модуль может содержать подпрограмму с именем READ. Если эта подпрограмма должна вызываться одним из других программных модулей, то имя READ должно быть глобальным. Определение глобальных имен осуществляется посредством директивы .ENTRY, директивы .GLOBAL или при завершении имени, используемого в качестве метки, двумя двоеточиями (: :). В директиве .GLOBAL указывается одно или несколько имен, отделенных запятыми. Действие директивы состоит в том, что эти имена помечаются как глобальные в таблице имен, определяемых пользователем, создаваемой при ассемблировании. Например, директива

```
.GLOBAL START, PRINT, SORT_TABLE
```

приведет к тому, что имена START, PRINT и SORT_TABLE будут помечены как глобальные и могут быть доступны другим программным модулям.

Директива .ENTRY, которой мы пользуемся, начиная с гл. 1, автоматически объявляет определяемое в ней имя как глобальное. Помимо этого символические адреса могут определяться как глобальные, если при появлении их в поле метки они заканчиваются не одним, а двумя двоеточиями. Таким образом, следующие строчки программы являются эквивалентными способами задания начала программы и определения метки START как глобального имени, которое может быть доступно из других раздельно ассемблируемых программных модулей:

```
1.          .ENTRY    START,0
2.          .GLOBAL   START
   START:   .WORD     0
3. START::  .WORD     0
```

Если такое имя, как START, используется в некотором модуле, но определяется где-то в другом месте, то оно должно быть объявлено в модуле как внешнее глобальное имя. Это делается с помощью директивы .EXTERNAL. Например, директива

```
.EXTERNAL PRINT, GET_SYMBOL, REMOVE
```

отмечает имена PRINT, GET_SYMBOL и REMOVE как внешние глобальные имена, которые используются в текущем модуле, но определены где-то еще. Это препятствует тому, чтобы ассемблер порождал сообщение об ошибке при обнаружении неопределенного имени. Вместо выдачи сообщения об ошибке ассемблер в файле объектного модуля формирует специальный код, который показывает, что значения этих определенных вне данного модуля адресов должны быть получены тогда, когда все модули будут объединяться компоновщиком для образования единой программы.

ОБРАБОТКА ГЛОБАЛЬНЫХ ИМЕН ПРИ КОМПОНОВКЕ

Процесс объединения всех программных модулей вместе и разрешения глобальных имен выполняется системной программой, называемой *компоновщиком*[1] (Linker). Компоновщик делает две вещи. Во-первых, он настраивает каждый программный модуль на последующий блок памяти. Отметим, что адрес настройки будет разным для каждого модуля; он зависит от того, как много памяти потребовалось для предыдущих модулей.

Во-вторых, компоновщик разрешает глобальные ссылки. Здесь компоновщик завершает тот процесс, который ассемблер не мог завершить из-за наличия внешних глобальных имен. Как и ассемблеру, компоновщику требуется дважды просмотреть файлы объектных модулей. На первом проходе строится таблица имен для всех определенных глобальных имен. На втором проходе заполняются отсутствующие адреса, соответствующие внешним глобальным именам, путем поиска их в таблице имен, которая была построена на первом проходе. Если глобальные имена остаются неопределенными, это означает, что произошла одна из нескольких возможных ошибок. Программист мог забыть объявить имя как глобальное; в имени могла быть допущена опечатка: мог быть не включен какой-то модуль и т. п. Компоновщик выдаст сообщение об ошибке для каждого неопределенного имени.

Помимо этого большинство систем обеспечивают возможность ведения библиотек подпрограмм. Библиотека – это файл, состоящий из нескольких объектных модулей, в котором компоновщик осуществляет поиск, пытаясь разрешить неопределенные глобальные имена. Модуль не будет загружаться из библиотеки до тех пор, пока он не потребуется, чтобы удовлетворить какое-либо другое неопределенное глобальное имя.

Рисунки 9.3 и 9.4 иллюстрируют использование глобальных имен. В подпрограмме (рис. 9.4) два двоеточия используются для объявления метки READ как глобального имени, на которое ссылаются из какого-то другого модуля (в данном случае из основной программы). Если бы метка READ в подпрограмме не была объявлена как глобальное имя, не произошло бы никакой ошибки ассемблирования. Однако, пытаясь разрешить

```
.TITLE    MAIN                ОСНОВНОЙ ПРОГРАММНЫЙ МОДУЛЬ
.EXTERNAL READ                ;READ ИСПОЛЬЗУЕТСЯ, НО НЕОПРЕДЕЛЕН
.ENTRY    START,0
...

JSB       READ                ;ОБРАЩЕНИЕ К READ
...

.END      START               ;УКАЗАН НАЧАЛЬНЫЙ АДРЕС
```

Рис. 9.3. Модуль основной программы

```
          .TITLE   SUB1          МОДУЛЬ ПОДПРОГРАММЫ
;ОТМЕТЬТЕ, ЧТО  READ::  ОБЪЯВЛЯЕТ СИМВОЛ READ ГЛОБАЛЬНЫМ
READ::    PUSHR    #^M<R0>       ;НАЧАЛО ПОДПРОГРАММЫ READ
          ...

          POPR     #^M<R0>
          RSB
          .END                   ;ОТМЕТЬТЕ, ЧТО НАЧАЛЬНЫЙ АДРЕС
                                 ;НЕ УКАЗЫВАЕТСЯ
```

Рис. 9.4. Модуль подпрограммы

[1] Это тот самый компоновщик, который обсуждался в разд. 4.7. По поводу концепции перемещения или настройки модулей отсылаем читателя к этому разделу.

ссылки на метку READ в основной программе, компоновщик мог осуществить просмотр библиотеки подпрограмм в поиске объектного модуля с именем READ. И если библиотека содержит подпрограмму READ, результат при выполнении программы может оказаться непредсказуемым. Ведь кто знает, что может делать модуль по имени READ, написанный совсем для других целей. Программисту найти ошибку такого рода очень трудно. Чтобы избежать неприятности, в библиотеках подпрограмм операционной системы VAX/VMS используются глобальные имена, содержащие знак доллара. Например, подпрограмма вычисления квадратного корня называется MTH$SQRT, а программа, вызываемая с помощью макроинструкции $EXIT_S, называется SYS$EXIT. Вот почему мы предостерегали от использования знака доллара в символических именах, задаваемых пользователем.

В основной программе (рис. 9.3) имя READ объявляется директивой .EXTERNAL как внешнее глобальное имя. Это показывает компоновщику, что он должен подставить числовой адрес READ в инструкцию JSB READ. Если бы директива .EXTERNAL была опущена, ассемблер пометил бы READ как неопределенное имя. Именно по этой причине в предыдущем варианте программы была включена строчка .DISABLE GLOBAL. Без этой строчки ассемблер автоматически объявляет все неопределенные имена внешними глобальными именами. Оставить строчку .DISABLE GLOBAL в вашей программе полезно, ибо автоматическое объявление глобальных имен задержит нахождение символических имен, в которых была допущена опечатка.

Разница между операторами .END на рис. 9.3 и рис. 9.4 довольно значительна. В модуле основной программы в директиве .END должен присутствовать адрес передачи управления или начальный адрес, но в других модулях директива .END должна использоваться без параметров, поскольку во всей программе не может быть более одной начальной точки.

9.4. ВЫЗОВ ПРОЦЕДУРЫ

ИНСТРУКЦИИ CALLG И CALLS

До сих пор вызов подпрограмм осуществлялся с помощью инструкций JSB, BSBW и BSBB, а возврат – с помощью инструкции RSB. Это простой способ вызова подпрограмм, но ЭВМ семейства VAX поддерживают и намного более сложную процедуру вызова подпрограмм, которая позволяет с помощью единственной инструкции вызова и единственной инструкции возврата автоматически выполнять следующие действия:

1. Обычное управление программным счетчиком, позволяющее осуществлять вызов и возврат так, как это делается с помощью инструкций JSB и RSB.
2. Передачу списков аргументов вызываемым подпрограммам.
3. Выборочное сохранение и восстановление регистров и кодов условий.
4. Сохранение в стеке информации, необходимой для вложенных вызовов подпрограмм.
5. Отметку критических точек в стеке, которые позволяют выполнять обратную трассировку подпрограммы и другие процедуры обработки ошибок.
6. Полное восстановление состояния стека при возврате управления из подпрограммы, что в существенной степени снижает опасность, вызванную тем, что в стеке остаются посторонние данные.
7. Очистку стека таким способом, который позволяет помещать в стек список аргументов при вызове подпрограммы и автоматически удалять его при возврате.

Чтобы все это выполнить, в дополнение к программному счетчику (PC, или регистр 15) и указателю стека (SP, или регистр 14) имеются два специальных регистра. Это *указатель аргументов* (Argument Pointer – AP, или регистр 12) и *указатель блока вызова* (Frame Pointer – FP, или регистр 13).

Указатель аргументов используется для указания на массив, который составлен из не-

которого числа длинных слов, передаваемых от вызывающей программы к вызываемой. Эти длинные слова могут содержать данные любого типа, но обычно они содержат адреса, которые, в свою очередь, указывают на элементы данных или массивы, передаваемые подпрограмме. Как уже упоминалось, передача адреса обладает тем преимуществом, что позволяет передавать данные как к подпрограмме, так и от нее. Кроме того, по причинам, которые станут ясны в дальнейшем, первое длинное слово списка аргументов обычно представляет число, которое говорит о том, сколько длинных слов содержится в списке. Помимо прочего это позволяет подпрограммам иметь переменное число аргументов как у функции MAX0 в Фортране или у функции CONCAT в Паскале фирмы UCSD.

Указатель блока вызова используется для сохранения значения указателя стека, чтобы возврат мог быть выполнен, даже если подпрограмма оставит данные в стеке. Вспомним, что подпрограммы, вызванные с помощью инструкции JSB, должны удалять все, что они включили в стек, до выполнения инструкции RSB. Наличие указателя блока вызова снимает это ограничение для подпрограмм, вызываемых с помощью инструкций CALLS и CALLG. **Указатель блока вызова используется также операционной системой для раскручивания (обратной трассировки) списка вызовов подпрограмм, чтобы вернуться обратно к основной программе в случае ошибки.**

Как отмечалось ранее, программы пользователя не должны модифицировать указатель аргументов и указатель блока вызова никакими другими способами, кроме описываемых в данном разделе. Инструкции, описанные в данном разделе, автоматически сохраняют и восстанавливают содержимое этих регистров стандартным образом. Любое другое обращение с ними может приводить к непредсказуемым результатам.

Чтобы использовать рассматриваемый способ вызова подпрограмм, подпрограмму вызывают с помощью инструкции CALLG и CALLS. Возврат из подпрограммы к основной или вызывающей программе осуществляется с помощью инструкции RET. Инструкции CALLG и CALLS сохраняют в стеке содержимое всех нужных регистров, а также устанавливают регистры AP и FP. Вызываемая подпрограмма должна сообщить инструкции CALLG или CALLS, какие регистры из группы R0 – R11 желательно сохранить. Инструкции CALLG и CALLS всегда сохраняют регистры AP, FP, SP и PC (регистры 12, 13, 14 и 15). Подпрограмма показывает, какие регистры желательно сохранить, с помощью битов, устанавливаемых в 16-битовом слове, которое называется *маской входа*. Эта маска фактически располагается по адресу точки входа в подпрограмму. Первая инструкция подпрограммы, которая будет выполняться, имеет адрес старше на два байта, чем адрес, указанный в инструкции CALLG или CALLS.

Работа с маской входа осуществляется во многом так же, как и работа с маской регистров в инструкциях PUSHR или POPR. Главное различие состоит в том, что могут быть сохранены только регистры R0 – R11. Каждый бит маски соответствует некоторому регистру. Как и в случае инструкций PUSHR и POPR, для создания маски входа может использоваться оператор ^M. Например, оператор ^M<R2,R3> указывает, что должны сохраняться регистры R2 и R3. Поскольку регистры 12–15 сохраняются автоматически, биты 12–15 служат другим целям. Биты 12 и 13 не используются и должны быть нулевыми. Биты 14 и 15 вызывают установку соответственно флагов разрешения прерываний по переполнению целых чисел и по переполнению десятичных чисел. Эти прерывания используются для обнаружения ошибок. Если происходит определенная ошибка и соответствующий бит, управляющий прерыванием, установлен, то управление передается операционной системе VAX/VMS. Более полное описание дано в гл. 15. Пока же все четыре бита сбрасываются в 0.

Рассмотрим пример. Чтобы осуществить вызов подпрограммы с помощью инструкции CALLG, в вашей программе должен быть образован массив длинных слов, представляющий список аргументов подпрограммы. Тогда последовательность действий при вызове подпрограммы будет выглядеть так:

```
        .EXTERNAL  SUBR
        . . .

ARGLST: .LONG      3              ;ИМЕЕТСЯ 3 АРГУМЕНТА
        .ADDRESS   A              ;A - АДРЕС ПЕРВОГО АРГУМЕНТА
        .ADDRESS   B              ;B - АДРЕС ВТОРОГО АРГУМЕНТА
        .ADDRESS   C              ;C - АДРЕС ТРЕТЬЕГО АРГУМЕНТА
        . . .

        CALLG      ARGLST, SUBR
        . . .
```

Обратите внимание, что директива .LONG используется для помещения константы 3 в длинное слово, а директива .ADDRESS – для помещения в длинные слова адресов A, B и C.

Подпрограмма может быть написана следующим образом:

```
SUBR::  .WORD      ^M<R2, R3, R4>  ;СОХРАНЯТЬ И ВОССТАНАВЛИВАТЬ R2-R4
        MOVL       4(AP), R2       ;В R2 АДРЕС ПЕРВОГО АРГУМЕНТА
        MOVL       8(AP), R3       ;В R3 АДРЕС ВТОРОГО АРГУМЕНТА
        MOVL       12(AP), R4      ;В R4 АДРЕС ТРЕТЬЕГО АРГУМЕНТА
        . . .

        RET
```

Обратите внимание, что первое слово подпрограммы является не инструкцией, а словом, в котором содержится маска входа. Нотация ^M<...> действует так же, как и в случае инструкций PUSHR и POPR.

При входе в эту программу регистр AP указывает на список аргументов, а 4(AP) – на первый аргумент (не считая длинного слова со счетчиком аргументов). Аналогично 8(AP) указывает на второй аргумент, а 12(AP) – на третий. Следовательно, инструкция MOVL 4(AP),R2 поместит в регистр R2 первый элемент списка аргументов. Однако следует заметить, что первым элементом списка аргументов является *адрес* ячейки A. Если желательно получить содержимое A (вероятно, это 16-битовое слово), то можно использовать с регистром R2 косвенную адресацию, например MOVW (R2), . . . , или можно, не заботясь об адресе, осуществить непосредственный доступ к данным с помощью инструкции MOVW @4(AP), . . . , вообще не привлекая регистр R2.

Директива .ENTRY, которая применялась для обозначения точки входа основной программы, может быть использована для любой программы или подпрограммы, которая вызывается инструкцией CALLS или CALLG. Например, директива

```
.ENTRY     START, 0
```

заставляет ассемблер выполнять следующее:

1. Помещать символический адрес START в таблицу имен, определяемых пользователем, как адрес следующей доступной ячейки.
2. Объявлять метку START глобальным именем.
3. Порождать слово кода, используя второй аргумент для формирования маски входа.

Поскольку операционная система не "полагается" на то, что программа сохранит регистры, мы всегда использовали нуль для маски входа. Следует, однако, отметить, что мы использовали эту директиву в начале программы по той причине, что операционная система VAX/VMS рассматривает нашу основную программу как подпрограмму, которую она вызывает с помощью инструкции CALLS или CALLG.

Можно использовать директиву .ENTRY в предыдущей подпрограмме, чтобы получить следующий эквивалентный текст программы:


```
.ENTRY     SUBR, ^M<R2, R3, R4>
MOVL       4(AP), R2       ;В R2 АДРЕС ПЕРВОГО АРГУМЕНТА
MOVL       8(AP), R3       ;В R3 АДРЕС ВТОРОГО АРГУМЕНТА
MOVL       12(AP), R4      ;В R4 АДРЕС ТРЕТЬЕГО АРГУМЕНТА
. . .

RET
```

Инструкция CALLS аналогична инструкции CALLG, однако с целью облегчения использования изменяемых списков аргументов в этой инструкции предполагается, что список аргументов загружается в стек. Инструкция CALLS помещает на вершину списка аргументов число аргументов, и весь список аргументов будет удален из стека при выполнении инструкции RET. Следовательно, вызывающей программе не нужно заботиться об удалении слов из стека. Вызов подпрограммы из предыдущего примера при использовании инструкции CALLS может выглядеть так:

```
. . .

PUSHAL     C
PUSHAL     B
PUSHAL     A
CALLS      #3, SUBR
. . .
```

Инструкция PUSHAL применяется для помещения в стек адресов A, B и C. Как инструкция

```
PUSHL      X
```

эквивалентна по действию инструкции

```
MOVL       X, -(SP)
```

так и инструкция

```
PUSHAL     A
```

эквивалентна инструкции

```
MOVAL      A, -(SP)
```

Она используется как более эффективное средство помещения адресов в стек. Обратите внимание, что обычная инверсия в отношении стека приводит к тому, что третий аргумент включается в стек первым.

В гл. 7 было показано, что инструкция MOVAL является частью семейства инструкций, включающего инструкции MOVAB, MOVAW, MOVAL, MOVAQ и MOVAO. Аналогично инструкция PUSHAL есть член следующего семейства инструкций:

Мнемоника	Значение
PUSHAB	Включить в стек адрес байта
PUSHAW	Включить в стек адрес слова
PUSHAL	Включить в стек адрес длинного слова
PUSHAQ	Включить в стек адрес квадраслова
PUSHAO	Включить в стек адрес октаслова

9.5. СВЯЗЬ МЕЖДУ ПРОГРАММАМИ НА ЯЗЫКЕ АССЕМБЛЕРА И НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

НЕОБХОДИМОСТЬ ОБЪЕДИНЕНИЯ ПРОГРАММ НА ЯЗЫКЕ АССЕМБЛЕРА И ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

Языки высокого уровня обычно предусматривают три типа программных модулей. Это основные программы, подпрограммы или процедуры, и функции. В среде операционной системы VAX/VMS такие программные модули могут раздельно компилироваться или

транслироваться в объектные модули. Связь между объектными модулями выполняется посредством глобальных имен подпрограмм и функций. Поскольку объектные модули, порождаемые компиляторами операционной системы VAX/VMS, имеют точно тот же формат, что и объектные модули, порождаемые ассемблером, возможно осуществление взаимосвязи основных программ, подпрограмм и функций, написанных на языках высокого уровня, с модулями, написанными на языке ассемблера, при условии соблюдения соглашений по программированию, принятых в операционной системе VAX/VMS.

Целый ряд причин приводит к желанию писать подпрограммы на языке ассемблера, которые будут вызываться из программы на языке высокого уровня. Во-первых, существуют операции, которые не так легко реализовать в некоторых языках высокого уровня. Сюда входят побитовые операции над словами (маскирование, упаковка и т.п.) и операции арифметики с многократно увеличенной точностью, отличающиеся от тех, которые обеспечиваются в этих языках. Во-вторых, некоторые операции исключаются из языков высокого уровня. Сюда входят нестандартный ввод-вывод, интерфейс с нестандартными устройствами и доступ к абсолютным ячейкам памяти.

Аналогично существует целый ряд причин для вызова подпрограмм на языках высокого уровня из языка ассемблера. Например, большинство языков высокого уровня обеспечивает упрощенные средства для выполнения ввода и вывода. В приложении В приведены подпрограммы ввода-вывода, написанные на языках высокого уровня, которые могут вызываться из основных программ на языке ассемблера. Кроме того, языки высокого уровня позволяют легко вычислять сложные математические выражения.

СОГЛАШЕНИЯ О СВЯЗЯХ В ОПЕРАЦИОННОЙ СИСТЕМЕ VAX/VMS

Соглашения о связях в операционной системе VAX/VMS – это стандартный способ передачи управления и информации между вызывающей программой и вызываемой. Этот стандарт используется в языках высокого уровня; реализованных как часть операционной системы. Сюда входят языки Бейсик, Блисс, Си, Кобол, Фортран, Паскаль и ПЛ/1. Эти соглашения используются также для получения сервиса, предоставляемого операционной системой.

Стандарты для некоторых языков, таких как Бейсик и Паскаль, не обеспечивают раздельную компиляцию подпрограмм и функций. Но поскольку раздельная компиляция является очень важной концепцией программирования, многие реализации таких языков обеспечивают расширение языка, позволяющее вызывать отдельно скомпилированные или ассемблированные программные модули. В частности, в реализации перечисленных выше языков в операционной системе VAX/VMS такие расширения, где необходимо, включены. Так как для стандартного Фортрана не требуется каких-либо расширений, то мы воспользуемся фортрановскими модулями для введения в соглашения о связях в операционной системе VAX/VMS. Будут также описаны расширения, необходимые для использования модулей Паскаля. По поводу других языков следует обратиться к справочному руководству по соответствующему языку, реализованному в среде операционной системы VAX/VMS.

СОГЛАШЕНИЯ О СВЯЗЯХ В ЯЗЫКЕ ФОРТРАН

При написании на языке ассемблера подпрограммы, предназначенной для совместной работы с программами на Фортране, необходимо пользоваться соглашениями о вызове и возврате, которые являются частью стандарта вызова процедур в операционной системе VAX/VMS. Нужно также знать, как передаются данные между подпрограммами. Это делается в основном с помощью списков аргументов, описанных для инструкций CALLS и CALLG.

Простейший случай представляет подпрограмма, не имеющая аргументов. Такая подпрограмма может вызываться в Фортране с помощью такого оператора, как

```
CALL      XSUB
```


Объектный код, генерируемый компилятором языка Фортран, в точности совпадает с тем, который ассемблер сгенерировал бы для следующего текста:

```
.EXTERNAL XSUB
. . .
CALLS     #0, XSUB
```

В инструкции CALLS присутствует операнд #0 потому, что подпрограмма не имеет аргументов. Это означает, что подпрограмма должна иметь простую структуру, показанную на рис. 9.5.

Следующая тема, которой следует заняться, касается связи по данным между программами на Фортране. Существует два способа, какими основная программа на Фортране может связываться с подпрограммой. Первый – с помощью списка аргументов; второй – через общие блоки. Общие блоки будут рассмотрены в отдельном разделе.

В программах на Фортране списки аргументов передаются путем передачи адресов, как это было описано ранее в разделе данной главы по инструкциям CALLG и CALLS. Например, программа на Фортране, имеющая оператор CALL YSUB(A, B, C), создает где-либо в памяти массив из четырех длинных слов: по одному длинному слову на каждый из трех адресов и длинное слово в начале массива, в котором задается число аргументов.

На рис. 9.6,*а* представлен эквивалент на языке ассемблера для этого оператора Фортрана. В такой реализации подразумевается, что переменные A, B и C объявляются в программе на Фортране как тип данных, занимающий длинное слово. Например, оператор INTEGER A, B, C объявляет переменные A, B и C целыми со знаком, каждое из которых занимает длинное слово (см. Справочное руководство по Фортрану операционной системы VAX/VMS). На рис. 9.6,*б* показано, как та же самая вызывающая последовательность могла быть написана при использовании инструкции CALLS. Отличие состоит в том, что список аргументов строится в стеке каждый раз, когда вызывается подпрограмма. Число аргументов дается в списке аргументов, так что подпрограмма может иметь дело с переменным числом аргументов. Это делает вызов с помощью инструкции CALLG совместимым с вызовом с помощью инструкции CALLS. (Другие вычислительные системы используют другие соглашения по определению числа аргументов, например применяют специальное значение для пометки конца списка аргументов.)

Рисунок 9.7 иллюстрирует подпрограмму, которая складывает элементы массива A и помещает сумму в S. Обратите внимание, что программа изменяет содержимое регистров R0 и R1. Однако все правильно, так как соглашения о связях в операционной системе VAX/VMS позволяют это делать. Стандартные процедуры вызова дают возможность подпрограммам модифицировать содержимое регистров R0 и R1. Причина этого будет разъяснена в разделе, посвященном функциям в Фортране и Паскале. Все другие регистры должны сохраняться. Следовательно, если подпрограмма использует регистры R2, R3 и R5, в слове маски входа должны быть установлены биты 2, 3 и 5 с помощью оператора .ENTRY SUB, ^M<R2, R3, R5>. В программе на рис. 9.7 должно быть сохранено только содержимое регистра R2.

```
.TITLE  XSUB                ПОДПРОГРАММА-ПРИМЕР
.ENTRY  XSUB, ^M<. . .>
. . .

RET
.END
```

Рис. 9.5. Подпрограмма, вызываемая из программы на языке Фортран

```
            .EXTERNAL  YSUB      ;ВСЕ ПОДПРОГРАММЫ ГЛОБАЛЬНЫЕ
            . . .

ARGS:       .LONG      3         ;ЧИСЛО АРГУМЕНТОВ
            .ADDRESS   A         ;A - АДРЕС ПЕРВОГО АРГУМЕНТА
            .ADDRESS   B         ;B - АДРЕС ВТОРОГО АРГУМЕНТА
            .ADDRESS   C         ;C - АДРЕС ТРЕТЬЕГО АРГУМЕНТА
            . . .

A:          .BLKL      1         ;ОБЛАСТЬ ПАМЯТИ ДЛЯ ПЕРВОГО АРГУМЕНТА
B:          .BLKL      1         ;ОБЛАСТЬ ПАМЯТИ ДЛЯ ВТОРОГО АРГУМЕНТА
C:          .BLKL      1         ;ОБЛАСТЬ ПАМЯТИ ДЛЯ ТРЕТЬЕГО АРГУМЕНТА
            . . .

            CALLG      ARGS, YSUB ;ВЫЗОВ ПОДПРОГРАММЫ
```

```
            .GLOBAL    YSUB      ;ВСЕ ПОДПРОГРАММЫ ГЛОБАЛЬНЫЕ
            . . .

A:          .BLKL      1         ;ОБЛАСТЬ ПАМЯТИ ДЛЯ ПЕРВОГО АРГУМЕНТА
B:          .BLKL      1         ;ОБЛАСТЬ ПАМЯТИ ДЛЯ ВТОРОГО АРГУМЕНТА
C:          .BLKL      1         ;ОБЛАСТЬ ПАМЯТИ ДЛЯ ТРЕТЬЕГО АРГУМЕНТА
            . . .

            PUSHAL     C         ;ПОСТРОИТЬ СПИСОК АРГУМЕНТОВ
            PUSHAL     B         ;ЗАМЕТЬТЕ, ЧТО ПОСЛЕДНИЙ АРГУМЕНТ
            PUSHAL     A         ;НАХОДИТСЯ НА ВЕРШИНЕ СТЕКА
            CALLG      #3, YSUB  ;ВЫЗОВ ПОДПРОГРАММЫ
```

Рис. 9.6. Эквивалент на языке ассемблера для оператора CALL YSUB(A, B, C) при использовании инструкции CALL (*а*); эквивалент на языке ассемблера для оператора CALL YSUB(A, B, C) при использовании инструкции CALL (*б*)

Подпрограмма на Фортране

```
      SUBROUTINE SUM(S, A)
      INTEGER S, I, A(10)
      S=0
      DO 10 I=1,10
        S=S+A(I)
10    CONTINUE
      RETURN
      END
```

Эквивалентная программа на ассемблере

```
            .TITLE     SAMPLE
            .ENTRY     SUM, ^M<R2>
            MOVL       8(AP), R0     ;ПЕРЕСЛАТЬ В R0 АДРЕС A
            CLRL       R2            ;В R2 БУДЕТ СУММА
            CLRL       R1            ;ОБНУЛИТЬ ИНДЕКС
10$:        ADDL2      (R0)+, R2     ;R2: =R2+A(I)
            AOBLSS     #10, R1, 10$  ;ВЫПОЛНЯТЬ В ЦИКЛЕ
            MOVL       R2, @4(AP)    ;ЗАПОМНИТЬ РЕЗУЛЬТАТ В S
            RET                      ;ВОЗВРАТ ИЗ ПОДПРОГРАММЫ
            .END
```

Рис. 9.7. Соотношение Фортрана и языка ассемблера

Инструкция CALLG для указания на список аргументов использует регистр AP. Следовательно, нотации 4(AP) и 8(AP) ссылаются на вторую и третью ячейки списка. Эти ячейки содержат адреса аргументов (S и A в программе на языке Фортран или Паскаль). Поскольку нотация 4(AP) ссылается на адрес S, запись @4(AP) отсылает к содержимому S. Поэтому при выполнении инструкции MOVL R2,@4(AP) содержимое регистра R2 копируется в ячейку S.

СОГЛАШЕНИЯ О СВЯЗЯХ В ПАСКАЛЕ

Обычный способ объявления процедуры в Паскале заключается в использовании декларации вида

```
PROCEDURE ИМЯ_ПРОЦЕДУРЫ (ОБЪЯВЛЕНИЕ АРГУМЕНТОВ);
BEGIN
  ТЕЛО ПРОЦЕДУРЫ
END
```

Эти операторы, конечно,должны располагаться в теле программы, вызывающей процедуру. Чтобы сообщить компилятору Паскаля, что тело процедуры отсутствует, но при этом ожидается, что процедура будет внешней подпрограммой, то все, что нужно сделать, – это надо заменить описание тела процедуры вместе с его операторными скобками BEGIN и END ключевым словом EXTERNAL, которое является расширением стандартного языка Паскаль. Следовательно, предыдущая процедура может быть переписана так:

```
PROCEDURE ИМЯ_ПРОГРАММЫ (ОБЪЯВЛЕНИЕ АРГУМЕНТОВ);
EXTERNAL ;
```

Обратите внимание, что описания аргументов все-таки должны быть включены, хотя и представляется, что эта информация не будет использоваться в самой программе на Паскале. Однако компилятор языка Паскаль выполняет контроль ошибок и проверку совместимости типов и в силу этого ему нужна такая информация.

В качестве примера того, как это можно использовать, рассмотрим программу на Паскале, которая вызывает внешнюю процедуру, называемую YSUB и имеющую три целочисленных аргумента. Программа может быть написана так, как показано на рис. 9.8. Эта программа будет порождать точно такую же вызывающую последовательность, как и в программе на Фортране, в иллюстрациях, данных на рис. 9.6,*а* и *б*[1]. Поэтому программа на языке ассемблера, написанная для этой вызывающей последовательности, будет одинаково хорошо работать и в случае программы на Фортране или Паскале.

```
PROGRAM MAIN(INPUT, OUTPUT)
VAR
 A, B, C : INTEGER;
 . . .

PROCEDURE YSUB(VAR X, Y, Z : INTEGER);
EXTERNAL ;
 . . .

 YSUB(A, B, C)
 . . .
```

Рис. 9.8. Вызов из программы на языке Паскаль внешней процедуры

[1] Здесь мы говорим о действии этих инструкций. Фактическая последовательность инструкций может варьироваться, но будет оставаться эквивалентной, как эквивалентны инструкции на рис. 9.6,*а* и *б*.

Важный момент различия, который следует отметить, состоит в том, что Паскаль хранит двумерные массивы по строкам, тогда как Фортран хранит их по столбцам (см. гл. 7). К многомерным массивам применяются те же принципы. В программе на Паскале первым меняется последний индекс массива, в программе же на Фортране первым меняется первый индекс.

ФУНКЦИИ В ЯЗЫКАХ ФОРТРАН И ПАСКАЛЬ

Из описаний, приводимых в большинстве руководств по языкам Фортран и Паскаль, может сложиться представление, что подпрограммы-функции совершенно отличаются от подпрограмм. Фактически же существует только одно реальное отличие — функции возвращают единственное значение, которое доступно для использования в выражениях. Это значение может занимать от одного байта до двух длинных слов, для типов данных двойной точности или комплексных. Эти слова всегда возвращаются в регистрах R0 и R1, причем R1 используется только тогда, когда требуется более одного длинного слова. Списки аргументов для функций реализуются точно так же, как и для подпрограмм или процедур. Таким образом, можно видеть, что подпрограмма SUM на рис. 9.7 или ее эквивалент на Паскале может быть переписана в виде функции, как показано на рис. 9.9. Обратите внимание, поскольку функция имеет тип INTEGER и, следовательно, возвращает 32-битовое двоичное число, необходим только регистр R0, чтобы результат стал доступен вызывающей программе. Отличие в основной программе сводится к тому, что вместо оператора CALL SUM(S,A) или SUM(S,A) будет оператор вида S=ISUM(A) или S:=ISUM(A);. В функции на языке Паскаль оператор MODULE является расширением стандартного Паскаля, позволяющим осуществлять раздельную компиляцию функций и процедур. (См. Справочное руководство по языку Паскаль операционной системы VAX/VMS.)

Функция на Фортране

```
      INTEGER FUNCTION ISUM(A)
      INTEGER A(1:10),J
      ISUM=0
      DO 10 J=1,10
        ISUM=ISUM+A(J)
10    CONTINUE
      RETURN
      END
```

Функция на Паскале

```
MODULE SAMPLE(INPUT,OUTPUT)
TYPE
  NMBS = ARRAY[1..10] OF INTEGER;
FUNCTION ISUM(VAR A : NMBS):INTEGER;
VAR
  J : INTEGER;
BEGIN
  ISUM := 0;
  FOR J:=1 TO 10 DO ISUM := ISUM + A[J];
  END;
END.
```

Эквивалентная программа на ассемблере

```
      .TITLE   SAMPLE          ПРИМЕР ФУНКЦИИ
      .ENTRY   ISUM,^M<R2,R3>
      MOVL     4(AP),R3        ;ПЕРЕСЛАТЬ В R3 АДРЕС A
      CLRL     R2              ;РЕГИСТР R2 СЧЕТЧИК
      CLRL     R0              ;ОБНУЛИТЬ СУММУ
10$:  ADDL2    (R0)+,R2        ;СЛОЖИТЬ A(J) С СУММОЙ
      AOBLSS   #10,R2,10$      ;ПОВТОРИТЬ В ЦИКЛЕ ДЕСЯТЬ РАЗ
      RET                      ;ВОЗВРАТ
      .END
```

Рис. 9.9. Эквивалент функции в программе на языке Фортран или Паскаль

ОБЩИЕ БЛОКИ ФОРТРАНА

В языке Фортран существует два способа передачи данных между основными программами и подпрограммами. Первый, как мы видели, осуществляется через списки аргументов. Другой способ, который программы на Фортране могут применять для связи, основывается на использовании общих блоков. Если в основной программе и в подпрограмме содержится оператор

```
COMMON  /DBLK/X,Y,Z
```

где переменные X, Y и Z ссылаются на одни и те же ячейки в обеих программах[1], то связь может осуществляться без передачи списков аргументов или массивов адресов. Этот способ работает при объявлении имени общего блока как глобального имени и образовании специального массива, достаточно большого для размещения переменных общего блока. Такое пространство устанавливается в каждой программе, но компоновщик разработан так, что перекрывает блоки с одним и тем же глобальным именем (т.е. общим блоком в разных программах назначается одна и та же область памяти). Таким образом гарантируется, что все программы ссылаются на одни и те же ячейки.

На рис. 9.10 показан эквивалент на языке ассемблера для оператора COMMON Фортрана. Директива .PSECT определяет, что все следующие за ней инструкции должны помещаться в определенную программную секцию. Эту определенную секцию идентифицирует имя DBLK. Остальные девять параметров показывают, что секция (блок):

1. Позиционно независима. Поскольку здесь предполагаются числовые данные, фактически занимаемые адреса не важны.
2. Перекрываемая, в противоположность размещаемой последовательно. Переменные X, Y и Z в этом блоке должны занимать те же самые ячейки, что и переменные X, Y и Z в блоках с тем же именем в других программах. Секции, размещаемые последовательно, располагались бы одна за другой и использовали раздельные области памяти.
3. Переместимая, в противоположность абсолютной. В языке ассемблера возможны абсолютные программные секции, но в языке Фортран все определенное пользователем — переместимо.
4. Глобальная, в противоположность локальной. Если необходимо, чтобы эта секция была доступна другим программным модулям, имя DBLK должно быть глобальным.
5. Разделяемая. Эти данные разделяемы с другими процессами.
6. Неисполняемая. Это данные, а не исполнимый код.
7. Разрешен доступ к чтению этой секции.
8. Разрешена запись или модификация.

Фортран

```
INTEGER X,Y(25),Z
COMMON /DBLK/X,Y,Z
```

Эквивалентное объявление на ассемблере

```
      .PSECT   DBLK,PIC,OVR,REL,GBL,SHR,NOEXE,RD,WRT,LONG
X:    .BLKL    1
Y:    .BLKL    25
Z:    .BLKL    1
      .PSECT
```

Рис. 9.10. Эквивалент общего блока на языке ассемблера

[1] Предполагается, что переменные X, Y и Z в обеих программах объявлены согласованно.

9. Выровнена по границе длинного слова. Это может повысить эффективность выполнения.

Хотя некоторые из этих вариантов в настоящее время могут быть и не реализованы (как 6 и 7), они должны быть указаны или будут использоваться значения по умолчанию. Если компоновщик обнаружит конфликтующие атрибуты для программных секций с одним и тем же именем, появится сообщения об ошибках.

После строки .PSECT идет последовательность помеченных блоков (вводимых директивой .BLKL) для переменных общего блока. Можно использовать и операторы, в которых задается значение данных (такие как .LONG). Это имело бы такое же действие, как и модуль BLOCK DATA Фортрана. В ячейках блока будут загружены данные, но следует избегать перезаписи данных, определенных в другом модуле. Чтобы не сталкиваться с этой проблемой, в стандартном Фортране запрещено использование операторов DATA для переменных в общих блоках, исключая модуль BLOCK DATA.

Блок заканчивается тогда, когда будет встречена другая директива .PSECT или когда появится директива .END в конце программы. Если директива .PSECT не имеет аргументов, это показывает, что снова выполняется ассемблирование обычной программной секции. С другой стороны, последующая директива .PSECT может иметь аргументы, показывающие на то, что определяется другой общий блок.

СИМВОЛЬНЫЙ ТИП ДАННЫХ ФОРТРАНА-77

Поскольку объекты символьного типа данных в языке Фортран имеют произвольную длину, они должны обрабатываться как-то отлично от объектов фиксированной длины, таких как вещественные или целые числа. Чтобы решить проблему получения информации о длине, приходится усложнять процедуру вызова для аргументов символьного типа по сравнению с описанными в предыдущих разделах для обычных переменных. Вместо списка аргументов, содержащего адрес данных, такая процедура вызова содержит адрес *дескриптора.* Как отмечалось в гл. 8, дескриптор – это блок данных, состоящий из двух длинных слов, который содержит информацию о длине и адрес, по которому расположены данные. Существует еще код, идентифицирующий объект как символьные данные. Это позволяет использовать дескрипторы и для других типов данных, хотя стандартное применение в Фортране – только для символьных данных.

Полное описание дескрипторов – дело весьма сложное: выше уровня данной книги. Однако пример на рис. 9.11 показывает, как дескрипторы используются для передачи подпрограмме простой переменной символьного типа.

```
Фортран                          Ассемблер

CHARACTER*20 A      A:      .BYTE    20        ;СИМВОЛЬНАЯ СТРОКА
. . .                       . . .

CALL XYZ(A)         ARGS:   .LONG    1         ;СПИСОК АРГУМЕНТОВ
                            .ADDRESS ADESC
                            . . .

                    ADESC:  .WORD    20        ;ДЛИНА СТРОКИ
                            .BYTE    14        ;ТИП CHARACTER
                            .BYTE    1         ;НЕ ПОЯСНЯЕТСЯ
                            .ADDRESS A         ;РАСПОЛОЖЕНИЕ СТРОКИ
                            . . .

                            CALLG    ARGS, XYZ
```

Рис. 9.11. Передача символьных данных

Дескриптор содержит: 20 в 16-битовом слове – длину символьной строки; 14 в 8-битовом байте – код, говорящий о том, что это дескриптор символьной строки; 1 в 8-битовом байте, которая означает, что это неиндексированная переменная (дескрипторы массивов намного более сложные, они дают все данные о структуре массива); адрес символьной строки в длинном слове.

Очевидно, что регистры R0 и R1 не могут быть использованы для получения результатов функции символьного типа, ведь результатом может быть и очень длинная строка. Поэтому функции символьного типа рассматриваются как подпрограммы. Другими словами, строка-результат добавляется как дополнительный аргумент в начале списка аргументов. Таким образом, вызов символьной функции CFUN, который может выглядеть так:

```
A=CFUN(B, C, D)
```

будет трактоваться, как если бы он был написан следующим образом:

```
CALL CFUN(A, B, C, D)
```

9.6. РЕКУРСИВНЫЕ ФУНКЦИИ

Для некоторого типа задач, таких как обработка лингвистической информации и искусственный интеллект, важны рекурсивные функции. Говоря коротко, рекурсивная функция – это такая функция, которая вызывает саму себя. Рекурсивные функции происходят от рекурсивных определений в математике.

В качестве примера рекурсивной функции рассмотрим функцию факториала. Инженер может удовлетвориться определением n факториала (или $n!$) как произведения целых чисел от 1 до n (случай $0! = 1$ рассматривается как специальный). Однако, хотя такого определения достаточно для вычисления, в качестве формы математического доказательства оно не очень полезно. Математик предпочел бы следующее определение:

$$0! = 1$$
$$n! = n * (n-1)!, \text{ если } n > 0$$

Первая строка показывает, что нуль факториал (0!) равен единице, вторая строка говорит, что n-факториал ($n!$) равен n, умноженному на $(n-1)$ факториал. Такие определения приемлемы как форма доказательства, называемая *математической индукцией.* Индуктивные доказательства широко используются в информатике для проверки правильности программ.

Хотя в стандартном Фортране не обеспечена поддержка рекурсивных функций, в большинстве более новых языков высокого уровня они допустимы. Например, языки Агол, АПЛ и Паскаль попадают в число тех, которые разрешают подпрограммам или подпрограммам-функциям вызывать самих себя. На рис. 9.12 показана рекурсивная функция на языке Паскаль, вычисляющая факториалы. Поскольку эта внешняя функция отвечает соглашениям о связях в операционной системе VAX/VMS, она может вызываться из основной программы на Паскале, Фортране или языке ассемблера.

Чтобы рекурсивная программа могла работать, она должна сохранять все необходимые данные, когда вызывает саму себя. Следует отметить, что для сохранения такой информации не могут использоваться фиксированные ячейки, так как информация окажется перезаписанной в процессе следующего рекурсивного вызова. Поскольку рекурсивная программа вызывает саму себя, то она должна быть в состоянии также и вернуть себе управление. Обратите также внимание, что должен существовать конечный путь, проходящий через последовательность вызовов функций, который в конце концов приводит к первоначально вызвавшей ее программе. В противном случае получится бесконечный цикл.

Стек ЭВМ VAX дает пользователям языка ассемблера очень важный инструмент для рекурсивного программирования. Чтобы рекурсивные функции могли работать, в сте-

```
MODULE EXTFUN(INPUT,OUTPUT);
  FUNCTION FAC(N : INTEGER) : INTEGER;
    BEGIN
      IF N=0 THEN
        FAC := 1
      ELSE
        FAC := N*FAC(N-1);
    END;

END.
```

Рис. 9.12. Программа вычисления факториала на языке Паскаль

```
        .TITLE  ПРОГРАММА ВЫЧИСЛЕНИЯ ФАКТОРИАЛА
TEMP:   .BLKL   1               ;В TEMP ЗАПОМИНАЕТСЯ N-1
        .ENTRY  FAC,0
        MOVL    @4(AP),R0       ;N ПОМЕЩАЕТСЯ В R0
        BNEQ    10$             ;ПРОПУСТИТЬ, ЕСЛИ НЕ 0
        MOVL    #1,R0           ;ФАКТОРИАЛ НУЛЯ РАВЕН 1
        RET
        PUSHL   R0              ;СОХРАНИТЬ N В СТЕКЕ
10$:    SUBL3   #1,R0,TEMP      ;N-1 ПОМЕСТИТЬ В TEMP
        PUSHAL  TEMP            ;СФОРМИРОВАТЬ СПИСОК АРГУМЕНТОВ
        CALLS   #1,FAC
        MULL2   (SP)+,R0        ;ВЫЧИСЛИТЬ  N*(N-1) ФАКТОРИАЛ
        RET                     ;И ВЕРНУТЬ ЕГО ЗНАЧЕНИЕ В R0
        .END
```

Рис. 9.13. Рекурсивная подпрограмма вычисления факториала на языке ассемблера

ке должны сохраняться необходимые данные каждый раз, когда подпрограмма вызывает себя. При возврате подпрограммы к самой себе элементы данных будут удалены из стека. На рис. 9.13 показано, как рекурсивная функция для вычисления факториала, представленная на рис. 9.12, может быть переписана на языке ассемблера. Обратите внимание, что в этой программе имеются два элемента, которые должны сохраняться в стеке. Первый элемент — это аргумент N, второй — адрес возврата и блок вызова, которые, конечно, сохраняются в стеке автоматически.

При входе в эту функцию значение N помещается в регистр R0, а N – 1 запоминается во временной ячейке TEMP. Затем программа проверяет, не равно ли N нулю; если это так, результат 0! = 1 заносится в R0. Если N не равно нулю, то адрес ячейки TEMP включается в стек, чтобы функция FAC могла вычислить (N – 1)!. Здесь требуется проявить осторожность, ибо рекурсивный вызов функции FAC уничтожит значение TEMP. Обычно, рекурсивные программы сохраняют подобные ячейки в стеке и восстанавливают их при возврате. Однако, поскольку ячейка TEMP в дальнейшем не используется, ее содержимое сохранять не нужно. После возврата управления значение (N – 1)! будет находиться в регистре R0. Оно умножается на значение N, которое было сохранено в стеке, и результат, равный N* (N – 1)! или N!, заносится в регистр R0.

УПРАЖНЕНИЯ 9.2

1. Напишите подпрограмму, аналогичную подпрограмме SUMUP, показанной на рис. 9.2. Но ваша программа будет иметь второй входной аргумент, задающий длину массива, подлежащего суммированию. Затем напишите основную программу, которая считывает 10 чисел, распечатывает их, вызывает вашу подпрограмму для их суммирования и затем

проделывает то же самое с 20 числами. Каждая подпрограмма, включая RNUM и PNUM (если используются), должна ассемблироваться раздельно как отдельные модули, которые компонуются вместе с помощью глобальных имен. (см. п. 4 упр. 9.1).

2. Напишите программу, которая считывает переменное число чисел (до 100 включительно), сортирует их и распечатывает отсортированный массив. Эта программа должна быть разбита на несколько ассемблируемых раздельно подпрограмм, по одной на каждую основную функцию. Отдельные программные модули следует связать, используя глобальные имена. Адреса массивов, их длина и т. п. должны передаваться в регистрах общего назначения при каждом вызове подпрограммы. (См. п. 5 упр. 9.1.)

3. Ниже предлагается метод генерации положительных случайных чисел в диапазоне 1 – 32767:

а) Начать с любого положительного числа из указанного диапазона;

б) Для каждого генерируемого числа: 1) сдвинуть первоначальное число один раз влево; 2) если два старших бита оба равны 1 или 0, сбросить младший бит в 0. Если один из битов равен 1, а другой 0, установить младший бит в 1. Другими словами, младший бит получает значение результата операции **ИСКЛЮЧАЮЩЕЕ_ИЛИ** над знаковым битом и битом 14 сдвинутого слова; 3) затем сбросить знаковый бит в 0.

Напишите подпрограмму, вызываемую из программы на Фортране или Паскале, для генерации таких псевдослучайных чисел. Протестируйте свою подпрограмму, вызывая ее несколько сотен раз и распечатывая результаты. Можете ли вы придумать какие-либо другие средства проверки случайности этих чисел? Если да, включите их в свою программу.

4. Напишите подпрограмму, вызываемую из программы на Фортране или Паскале, когда вызов осуществляется такими операторами:

Фортран	Паскаль
CALL LOCS(A)	LOCS(A)

Здесь A — это массив из трех элементов типа INTEGER. Ваша подпрограмма заполняет массив A следующим образом:

A(1) – адрес инструкции, к которой произойдет возврат при выполнении инструкции RET.

A(2) — адрес списка параметров.

A(3) – адрес самого массива A.

Напишите основную программу на языке Фортран или Паскаль, которая тестирует эту подпрограмму. Проверьте результаты работы программы по картам загрузки памяти, таблицам имен и т. п., причем настолько быстро, насколько сможете.

5. Рекурсивная функция, известная как функция Аккермана, определяется через неотрицательные целые числа следующим образом:

$$A(0,n) = n + 1 \text{ для } n > 0$$
$$A(m,0) = A(m-1,1) \text{ для } m > 0$$
$$A(m,n) = A[m-1, A(m, n-1)] \text{ для } m \text{ и } n > 0$$

Напишите рекурсивную подпрограмму на языке ассемблера для вычисления функции Аккермана.

6. Напишите функцию, которую можно вызывать из программы на Фортране или Паскале, которая вызывает рекурсивную функцию Аккермана из п. 5, и протестируйте ее с помощью основной программы, вызывающей эту функцию с различными значениями. (Предостережение: не используйте для аргументов числа больше 3.)

7. Напишите программу сортировки, такую, как описано в п. 2, за исключением того, что:

а) подпрограммы ввода и вывода и основная программа должны быть написаны на языке Фортран;
б) программа сортировки должна быть написана на языке ассемблера;
в) все аргументы должны передаваться через общие блоки.

ГЛАВА 10. МАКРОИНСТРУКЦИИ И УСЛОВНОЕ АССЕМБЛИРОВАНИЕ

10.1. ВВЕДЕНИЕ

В данной главе рассматриваются некоторые особенности ассемблера, которые мало связаны с основным процессом ассемблирования, заключающимся в генерации машинного кода. Макроинструкции являются средством порождения больших и сложных фрагментов текста программы, осуществляемого с помощью всего лишь нескольких простых операторов. Условное ассемблирование позволяет получать различные варианты генерируемого кода в зависимости от управляющих параметров, посредством которых можно влиять на структуру всей программы в целом или на расширение макроинструкций. Фактически средства макрообработки включены во все современные языки ассемблера. По сути макроинструкции представляют собой поименованные блоки текста программы, в которых допускается параметрическая подстановка и которые могут быть вставлены в различные места программы. Возможность макрообработки не является особенностью, свойственной только для языка ассемблера, поскольку любой язык программирования может быть функционально расширен для работы с поименованными блоками текста программы. Тем не менее макрообработка включена лишь в некоторые языки высокого уровня, поэтому она чаще всего ассоциируется с языком ассемблера.

В этой главе читателю будут даны основы написания макроинструкций и использования условного ассемблирования. Однако эти темы весьма сложные и материал данной главы является только начальным их изложением. В гл. 11 будет показано, что имеется большое число макроинструкций, написанных для того, чтобы помочь программистам в использовании *системных сервисных средств* (system services) операционной системы VAX/VMS. Системные сервисные средства представляют собой программы и подпрограммы, позволяющие пользователю выполнять системно-ориентированные функции, например ввод и вывод. Макроинструкции, позволяющие обращаться к системным сервисным средствам, или *системные макроинструкции* содержатся в библиотеке, доступной для программистов, работающих с языком ассемблера. Чаще всего эти макроинструкции применяются для генерации вызывающих последовательностей и списков аргументов системных сервисных программ. Одной из таких макроинструкций, уже использованной ранее в тексте книги, является $EXIT_S, с помощью которой формируется вызов подпрограммы SYS $EXIT.

10.2. ПОВТОРЯЮЩИЕСЯ ФРАГМЕНТЫ ПРОГРАММЫ

ПРЕИМУЩЕСТВО ПОВТОРЕНИЯ ФРАГМЕНТОВ ПРИ АССЕМБЛИРОВАНИИ

При программировании задачи на языке ассемблера часто обнаруживается, что большие фрагменты программы неоднократно повторяются. Первое, что при этом приходит в голову, – избавиться от повторений, применяя циклы и подпрограммы. Хотя в принципе это верно, есть ситуации, когда циклы и подпрограммы – не лучшее решение. Ниже перечислены некоторые возможные соображения, по которым иногда нежелательно использование циклов и подпрограмм.

1. Программы, содержащие циклы и подпрограммы, будут выполняться медленнее, чем программы с эквивалентными повторяющимися фрагментами кода. Это происходит потому, что при организации циклов требуются дополнительные инструкции для подсчета числа повторений, индексации, проверки условий завершения цикла и передачи управления на начало цикла. Аналогично при использовании подпрограмм должен осуществляться их вызов, а затем передача управления обратно; подпрограмме должны передаваться аргументы, а также должно выполняться сохранение содержимого регистров. По этим причинам при критичном времени выполнения программы может возникнуть необходимость в том, чтобы отказаться от применения циклов и подпрограмм (особенно на самых нижних уровнях вложенности программы).

2. В некоторых случаях требуется повторение фрагмента, но оно не должно быть точным. Это возможно при применении подпрограммы со сложной структурой аргументов, однако часто такое решение неприемлемо. Если в подпрограмме выполняются простейшие действия, то объем дополнительного кода, необходимого для вызова подпрограммы и передачи ей аргументов, может оказаться больше, чем объем кода, получаемого при обычном повторении фрагмента программы.

3. Наконец, повторяемый фрагмент программы может включать данные, а не инструкции. Предположим, например, что массив должен быть целиком заполнен пятерками. Это можно сделать с помощью программы инициализации, которая заносит число 5 во все элементы массива. Но иногда неудобно использовать программы инициализации и более предпочтительной является инициализация массива, осуществляемая на этапе ассемблирования. Значение 5 заносится в элементы массива при неоднократном повторении директивы .LONG 5.

Выше приведены далеко не все возможные соображения, но и они позволяют получить некоторое представление о том, почему предпочтительнее повторение фрагментов программы, чем использование циклов. Именно эти соображения явились предпосылками для создания *макроассемблеров*.

БЛОКИ ПОВТОРЕНИЯ

В языке ассемблера простейший тип повторения представляет блок повторения. Блок повторения – это фрагмент текста программы, повторяемый несколько раз без каких-либо изменений. Примером того, когда в этом возникает необходимость, является случай, приведенный выше, – массив, заполненный пятерками. Его можно сформировать, поместив одну за другой несколько строчек .LONG 5.

Для удобства пользователя в языке ассемблера ЭВМ семейства VAX имеется специальная директива указания повторений. Директива .REPT используется в контексте

```
.REPT   ВЫРАЖЕНИЕ
.
        БЛОК КОДА
.
.ENDR
```

Фрагмент текста программы будет повторен много раз, причем количество повторений определяется значением выражения, следующего за директивой .REPT. На рис. 10.1,*а* показано, каким образом можно сформировать блок из семи длинных слов, содержа-

(а) Блок повторения

```
.REPT   7
.LONG   5
.ENDR
```

(б) Эквивалентный код

```
.LONG   5
.LONG   5
.LONG   5
.LONG   5
.LONG   5
.LONG   5
.LONG   5
```

Рис. 10.1. Использование блока повторения

щих число 5, используя директиву .REPT. На рис. 10.1,*б* представлен эквивалентный текст программы. Заметим, что, хотя в примере между директивами .REPT и .ENDR расположена только одна строка, число строк не ограничивается[1] и фрагмент текста программы может иметь любую длину.

Обратите также внимание на то, что строки в блоке повторения повторяются без изменений. Никакие изменения не вносятся в строки текста программы. Это *не означает*, однако, что генерируемый машинный код также не может быть изменен. Для генерации различного машинного кода можно использовать выражения и определения, что и будет показано в следующем разделе.

ИСПОЛЬЗОВАНИЕ В БЛОКАХ ПОВТОРЕНИЯ СЧЕТЧИКА АДРЕСА

Предположим, что программисту требуется создать массив из 100 пар длинных слов. Первое длинное слово в каждой паре содержит начальный адрес следующей пары длинных слов. Второе длинное слово первоначально содержит 0. Такая организация данных известна как *однонаправленный список* и часто используется для данных, подлежащих переупорядочиванию. Для переупорядочивания данных в этом случае требуется переместить только некоторые указатели (адреса). Сами слова данных остаются на своих местах.[2]

Такая структура данных может быть создана с помощью операторов

```
10$:    .ADDRESS    20$
        .LONG   0
20$:    .ADDRESS    30$
        .LONG   0
30$:    .ADDRESS    40$
        .LONG   0
40$:    .ADDRESS    50$
        .LONG   0
50$:    . . .
```

Очевидно, что здесь имеет место повторяющаяся структура, однако это не точное повторение, поскольку все адреса различны. Для того чтобы реализовать указанную структуру с помощью блоков повторения, используется специальный знак . (*точка*). Точка обозначает *значение счетчика адреса*. Как упоминалось в гл. 4, счетчик адреса ассемблера содержит адрес следующей доступной ячейки памяти. Поскольку адрес ячейки постоянно меняется, то изменяется и значение, приписываемое точке. Например, при ассемблировании длинных слов значением, присвоенным точке, является адрес длинного слова, ассемблируемого в текущий момент. Адрес следующего длинного слова в памяти может быть обозначен выражением .+4. Следующее затем длинное слово будет иметь адрес (.+8). Как можно сформировать структуру такого связного списка, показано на рис. 10.2.

```
        .REPT   100
        .ADDRESS    .+8
        .LONG   0
        .ENDR
```

Рис. 10.2. Структура связного списка

[1] Единственным ограничением является то, что может произойти переполнение рабочей области памяти, используемой ассемблером.

[2] В ЭВМ семейства VAX имеется ряд инструкций, предназначенных специально для обработки данных в связных списках. Однако они довольно сложны и не относятся к рассматриваемой тематике. Подробности использования см. в гл. 13 и в руководстве "VAX Architecture Handbook".

ИСПОЛЬЗОВАНИЕ ИМЕН В БЛОКАХ ПОВТОРЕНИЯ

Несмотря на то, что часто переменные данные можно представить в виде сложных выражений, включающих знак точки, есть случаи, когда это сделать трудно, почти невозможно. Тогда может оказаться полезным другой способ определения переменных данных. Он связан с использованием знака равенства (=). Знак равенства уже употреблялся ранее при определении параметров в выражениях вида SIZE=10. Большей частью определения имен, заданных с помощью знаков = и :, подобны. Но есть и существенное различие. Если имя определяется более одного раза с использованием знака :, то имеет место ошибка повторного определения; в этом случае программа будет сопровождаться сообщением об ошибке. В отличие от этого имена, определенные знаком =, могут быть переопределены в дальнейшем очередным знаком =. Эти определения повторяются на двух проходах ассемблера, и новое значение имени будет распространяться на строки программы, следующие после его переопределения.

На рис. 10.3 показано, как структура рис. 10.2 может быть реализована при переопределении имени с использованием знака = вместо знака .. Хотя этот пример может показаться в чем-то более сложным, его преимущество заключается в большей общности. Рассмотрим, например, заполнение массива из восьми длинных слов значениями факториалов от 1 до 8. Переопределение в блоке повторения является простым и эффективным способом создания такого массива (рис. 10.4).

Отметим, что имена K и F на рис. 10.4 не являются ни ячейками памяти, ни даже их именами. Это просто имена, которым соответствуют числовые значения в таблице имен. Они не используются в качестве адреса инструкции или элемента данных. Заметим также, что вычисления типа K=K+8, K=K+1 или F=F*K производятся во время ассемблирования программы, а не во время ее выполнения. Данные строки программы не вызывают порождение какого-либо выполняемого кода.

Эта особенность языка ассемблера может вносить путаницу, поскольку в языках высокого уровня, таких как Фортран или Паскаль, выражения K=K+8 или K:=K+8; рассматриваются как выполняемые операторы, которые транслируются в инструкции, такие как

```
Блок повторения                 Эквивалент                      Эквивалент
                                блока повторения                с вычисляемым K

K=0                             K=0
A:      .REPT   100             A:                              A:      .ADDRESS    A+8
K=K+8                           K=K+8                                   .LONG   0
        .ADDRESS    A+K                 .ADDRESS    A+K                 .ADDRESS    A+16
        .LONG   0                       .LONG   0                       .LONG   0
        .ENDR                   K=K+8   .ADDRESS    A+K                 .ADDRESS    A+24
                                        .LONG   0                       .LONG   0
                                K=K+8   .ADDRESS    A+K                 . . .
                                        .LONG   0
                                        . . .
```

Рис. 10.3. Построение связного списка с переопределением имен

```
K=1
F=1
        .REPT   8
        .LONG   F
K=K+1
F=F*K
        .ENDR
```

Рис. 10.4. Массив факториалов

ADDL2 #8,K. В отличие от этого в языке ассемблера оператор K=K+8 задает операцию над значениями, содержащимися в таблице имен, осуществляемую во время ассемблирования и не требующую вычислений во время выполнения программы.

Это отличие легче понять в том случае, когда имена представляют собой адреса. В языке высокого уровня результатом выполнения оператора вида K=L+4 (или K:=L+4) будет то, что *содержимое* ячейки с адресом K станет равно содержимому ячейки с адресом L+4. В языке ассемблера результатом вычисления K=L+4 будет то, что K станет адресом длинного слова, расположенного после длинного слова L.

10.3. СИМВОЛИЧЕСКИЕ ВЫРАЖЕНИЯ

ОСОБЕННОСТИ ФОРМИРОВАНИЯ ВЫРАЖЕНИЙ

В предшествующих главах использовались символические выражения простейшего вида. Обычно такие выражения сводились к выражениям вида A−6 или A+6, означающим адрес ячейки, расположенной на шесть ячеек перед или после ячейки A. При использовании блоков повторения, макроинструкций и условного ассемблирования нужны более сложные выражения.

Прежде всего следует вспомнить, что необязательно применять имена только для задания адресов. Символические имена могут применяться для обозначения чисел, которые могут быть использованы для любых целей.

Программист, пишущий программы на языке ассемблера, должен хорошо уяснить связь между самим процессом ассемблирования и выполнением программы. Например, как было упомянуто выше, строка ассемблера K=K+1 не означает, что во время выполнения программы значение K будет увеличено на 1. Как уже было сказано, для этого оператора не будет сгенерирован какой-либо исполняемый машинный код. В данном случае будет увеличено на 1 значение, соответствующее имени K в таблице имен ассемблера. Не следует путать оператор K=K+1 с инструкцией INCL K. Это совершенно разные вещи.

Очень важно отметить, что существует разница между именами для значений, изменяющихся при перемещении программы, именами для значений, которые при этом не изменяются, и глобальными именами, определенными в других модулях. Например, именам, обозначающим адреса в текущем модуле, соответствуют значения, которые должны измениться при настройке программы на область памяти, где она будет выполняться. Такие имена называются *перемещаемыми*. Значения имен, применяемых для обозначения чисел, при перемещении не меняются. Кроме того, имеются адреса, значение которых фиксированно. Эти адреса называются *абсолютными адресами* и используются операционной системой для доступа к ячейкам специального назначения, которые описываются в гл. 15. Имена, соответствующие значениям, не изменяемым при перемещении программы, называются *абсолютными*. Имена в таблице имен делятся на переместимые и абсолютные. К именам третьего типа относятся глобальные имена, но поскольку они обрабатываются специальным образом, то рассматриваться не будут.

ПРАВИЛА ФОРМИРОВАНИЯ ВЫРАЖЕНИЙ

В языке ассемблера выражения могут использоваться практически во всех случаях, когда требуется задание числа или значения. Выражения формируются в виде комбинации имен и чисел со знаками математических операций +, −, * и /, почти так же, как в языках Паскаль или Фортран. Допускается использование скобок, за тем исключением, что в качестве левых и правых скобок используются угловые скобки (< и >). Важно отметить, что нет приоритетности операций. Выражения вычисляются слева направо, при этом скобки действуют обычным образом. Например, выражение X+Y*<A+<5*B/K>> является допустимым выражением. Поскольку вычисления ведутся слева направо, то выражение 1+2*3 равно 9, а не 7. (В ассемблерах некоторых других ЭВМ не придерживаются этих правил.)

Выражения имеют те же типы, что и имена. Вместо того, чтобы перечислять исчерпывающий набор правил для определения типов выражений, далее дается набор правил на уровне здравого смысла, которыми можно руководствоваться всегда, за исключением наиболее нестандартных ситуаций[1].

1. В качестве значения выражения должно быть получено нечто такое, что можно поместить в таблицу имен, т.е. в результате вычисления должно получиться следующее:

а) число или абсолютный адрес;

б) адрес ячейки в программе или адрес ячейки, имеющей фиксированное смещение от заданной ячейки программы, например ячейки, смещенной на 100 байтов после конца вашей программы;

в) глобальный адрес плюс или минус фиксированное число.

2. Выражение, полностью состоящее из чисел и абсолютных имен будет абсолютным.

3. Переместимое имя плюс или минус абсолютное значение будет переместимым выражением. Это объясняется тем, что ячейка, смещенная на фиксированное расстояние от которой ячейки в программе, при перемещении программы будет также перемещаться (и поэтому является переместимой).

4. Разность между адресами двух переместимых имен, определенных в одной программной секции, является *абсолютной*. Это объясняется тем, что разность между двумя адресами представляет собой число ячеек, расположенных между этими двумя адресами. Если оба адреса являются переместимыми и расположены в одной программной секции, то при перемещении программной секции они изменятся на одну и ту же величину. Поэтому число ячеек между ними остается неизменным или фиксированным.

Приведенные выше правила могут, разумеется, применяться и к частям сложных выражений. Кроме того, существуют два правила, относящиеся к операциям умножения и деления.

5. В случае умножения хотя бы один из операндов должен быть абсолютным. Тогда умножение может рассматриваться как многократное сложение. Например, в случае, когда имена A, B и C являются переместимыми, рассмотрим выражение <3*A> − <2*B> − C. Это выражение может быть переписано в виде A+A+A−B−B−C, что эквивалентно A−B+A−B+A−C. Заметим, что каждая разность является разностью двух переместимых адресов и является поэтому абсолютным выражением. Следовательно, все выражение будет абсолютным.

6. В случае деления оба операнда должны быть абсолютными. Трудно вообразить, что может быть допущено иное. Тем не менее это не мешает использовать деление в сложных выражениях, включающих переместимые части. Например, если разность между двумя адресами требуется выразить в длинных словах, а не в байтах, то можно использовать выражение <A−B>/4. Заметим, что это выражение в действительности является частным от деления двух абсолютных выражений (предполагается, что A и B являются переместимыми). Отметим также, что данное выражение более предпочтительно, чем эквивалентное <A/4> − <B/4>. Хотя такое выражение на ЭВМ семейства VAX может быть вычислено правильно при условии, что значение выражения определяется компоновщиком, оно может вызывать затруднения или привести к ошибке в случае, если значение выражения определяется на этапе ассемблирования.

ПРИМЕР ИСПОЛЬЗОВАНИЯ ВЫРАЖЕНИЙ

Обычно при программировании на языке ассемблера важно иметь возможность использовать выражения и имена вместо чисел, поскольку это дает программисту более простой способ модификации программы, а также облегчает понимание и документирование про-

[1] Выражения, не подчиняющиеся этим простым правилам или включающие глобальные имена, помечаются ассемблером ЭВМ семейства VAX как *сложные*. Сложные выражения вычисляются компоновщиком.

```
;ПОДПРОГРАММА MPRINT - ПЕЧАТЬ СООБЩЕНИЯ В РАМКЕ
;
        .TITLE  MPRINT          ПЕЧАТЬ СООБЩЕНИЯ
        .EXTERNAL               PLINE
MARK=^A"*"
BOXES=3
MSG:    .ASCII  "ВЫВОД   СООБЩЕНИЯ"
EMSG:
MSG_LENGTH=EMSG-MSG+<2*BOXES>
MSG_BUF:
        .BLKB   MSG_LENGTH
;
        .ENTRY  MPRINT,^M<R2>
        BSBW    HORIZ           ;ПЕЧАТЬ ГОРИЗОНТАЛЬНОГО ОБРАМЛЕНИЯ
        MOVAB   MSG_BUF,R1      ;ПОМЕСТИТЬ В БУФЕР
        BSBW    VERT            ;НАЧАЛО ВЕРТИКАЛЬНОГО ОБРАМЛЕНИЯ
        MOVAB   MSG,R0
        MOVAB   EMSG,R2         ;R2 - АДРЕС КОНЦА СТРОКИ
10$:    MOVB    (R0)+,(R1)+     ;ПЕРЕСЫЛКА СИМВОЛОВ ИЗ MSG В BUFFER
        CMPL    R0,R2           ;ПОВЕРКА НА КОНЕЦ СТРОКИ
        BNEQ    10$
                                ;ПОМЕСТИТЬ В БУФЕР
        BSBW    VERT            ;КОНЕЦ ВЕРТИКАЛЬНОГО ОБРАМЛЕНИЯ
        BSBW    PBUF            ;ПЕЧАТЬ СТРОКИ СООБЩЕНИЯ
        BSBW    HORIZ           ;ПЕЧАТЬ ГОРИЗОНТАЛЬНОГО ОБРАМЛЕНИЯ
        RET
;
HORIZ:  MOVL    #BOXES,R2           ;R2 - СЧЕТЧИК СТРОК
10$:    CLRL    R0                  ;R0 - ИНДЕКС В БУФЕРЕ
20$:    MOVB    #MARK,MSG_BUF[R0]   ;ПОМЕСТИТЬ СИМВОЛ MARK В БУФЕР
        AOBLSS  #MSG_LENGTH,R0,20$  ;ПОВТОРЯТЬ ДО ЗАПОЛНЕНИЯ БУФЕРА
        BSBW    PBUF                ;ПЕЧАТЬ СТРОКИ С ЗАПОЛНИТЕЛЕМ MARK
        SOBGTR  R2,10$              ;ПЕЧАТЬ #BOXES СТРОК
        RSB
;
VERT:   MOVL    #BOXES,R0       ;ВСТАВИТЬ #BOXES СИМВОЛОВ MARK
10$:    MOVB    #MARK,(R1)+     ;R1 - УКАЗАТЕЛЬ БУФЕРА
        SOBGTR  R0,10$          ;ПОВТОРЯТЬ ПОКА, НЕ ВСТАВЯТСЯ ВСЕ #BOXES
        RSB
;
PBUF:   PUSHR   #^M<R0,R1>      ;СОХРАНИТЬ R0 И R1 В СТЕКЕ
        MOVAL   MSG_BUF,R0      ;R0 - УКАЗАТЕЛЬ БУФЕРА
        MOVZWL  #MSG_LENGTH,R1  ;R1 СОДЕРЖИТ ДЛИНУ
        JSB     PLINE           ;ПЕЧАТЬ СТРОКИ
        POPR    #^M<R0,R1>      ;ВОССТАНОВИТЬ R0 И R1
        RSB
;
        .END
```

Рис. 10.5. Подпрограмма печати сообщений

граммы. В качестве примера рассмотрим программу, которая распечатывает сообщение и окружает его тройной рамкой из звездочек:

```
*************************
*************************
*************************
*** ВЫВОД   СООБЩЕНИЯ ***
*************************
*************************
*************************
```

Это делается для того, чтобы сообщение сразу бросалось в глаза. На рис. 10.5 представлена программа печати такого сообщения.

Если использовать имена и выражения, то внесение изменений в программу становится довольно простым. Например, если сообщение должно быть окружено пятью рамками из звездочек вместо трех, то единственное, что требуется изменить, – это шестая строка программы, которая теперь будет выглядеть так: BOXES=5. Если требуется заменить звездочки на знаки @, то достаточно заменить пятую строку программы на MARK=^A"@". Оператор ^A используется для того, чтобы определить значение в коде ASCII для следующей строки символов. Таким образом, выражение ^A"@" задает значение кода ASCII для знака @, равное ^X40. Подразумевается, что в программе используется глобальная подпрограмма PLINE для распечатки строки символов, адрес строки передается ей через регистр R0.

ОСОБЕННОСТИ ВЫЧИСЛЕНИЯ ВЫРАЖЕНИЙ ПРИ АССЕМБЛИРОВАНИИ

Последний момент, связанный с выражениями и их применением, который будет рассмотрен, касается очередности определения и использования имен. Вспомним, что два прохода в процессе ассемблирования требуются потому, что в инструкции может быть ссылка на символический адрес, который определен в программе позднее. Однако имеются такие случаи, когда значение должно быть известно на первом проходе. Тогда имена, используемые в выражениях для вычисления таких значений, должны быть определены заранее.

По сути поскольку таблица имен создается на первом проходе, то все имена, влияющие на значения в таблице имен, должны быть определены до своего использования. Простым примером такой ситуации является случай непосредственного определения, который обычно приводит к ошибке. Например, при

```
A=B
B=5
```

переменная A останется неопределенной после завершения первого, а также в начале второго прохода. Разумеется, эта проблема будет легко устранена, если эти две строки поменять местами.

Более сложная проблема возникает, когда выражение может повлиять на объем программы. Если в таком выражении содержатся неопределенные имена, то это вызывает ошибку в вычислении всех последующих адресов в остальной части программы. Например,

```
X:      .BLKL   A
Y:      .BLKL   1
A=5
```

Поскольку переменная A не определена на первом проходе, то ассемблеру не будет известно, сколько длинных слов нужно зарезервировать под массив X. За неимением лучшего ассемблер присваивает для неопределенных имен нулевое значение. Поэтому адрес Y будет таким же, как адрес X. Однако на втором проходе будет известно, что значение A равно 5, так что адрес Y окажется на 20 больше, чем адрес X. Это расхождение будет отмечено сообщением типа Expression is not absolute ("Выражение не является абсолютным"). Причиной появления такого сообщения является то, что в ассемблере подразумевается, что неопределенные выражения представляют собой глобальные или перемещаемые имена, которые будут определены в дальнейшем.

Кроме того, всякий раз, когда выражение используется таким образом, что его значение должно быть известно во время ассемблирования, это выражение должно не только состоять целиком из предварительно определенных имен, но и быть настолько простым, чтобы ассемблер мог вычислить его значение. Очевидно, что поэтому не могут быть использованы сложные выражения, которые должны вычисляться компоновщиком.

УПРАЖНЕНИЯ 10.1

1. Покажите результат ассемблирования (машинный код) следующих блоков повторения. Предполагается, что загрузка производится с ячейки ^X1200.

```
а.  .REPT    5
    .ADDRESS .+5
    .ENDR

в.  K=0
    .REPT    4
    K=K+4
    .ADDRESS .+K
    .ENDR
```

```
б.  .REPT    7
    .LONG    5
    .ADDRESS .-4
    .ENDR
г.  K=0
    .REPT    5
    K=K+1
    .REPT    K
    .LONG    K*3
    .ENDR
    .ENDR
```

2. Напишите блок повторения, который генерирует массив из 100 длинных слов с меткой NUMBS. Этот массив должен быть заполнен числами от 1 до 100.

а) решите задачу, используя выражения со знаком точки;

б) решите задачу, используя знак переопределения =.

3. Предположим, что в выражениях I=2, J = 3 и K = 5 определяются значения, присвоенные именам I, J и K в программе на ассемблере. Определите значения следующих выражений в шестнадцатеричном представлении:

```
а.  I*J+K
в.  J/I+2
д.  I+<J*K>/K
ж.  I+J*<I+J>
```

```
б.  I+J*K
г.  I+J/K
е.  I+<<J*K>+K>
з.  <I+J>*I+J
```

4. Предположим, что A, B и C являются переместимыми именами (т.е. метками, указывающими ячейки программы). Кроме того, I, J и K являются абсолютными именами (т.е. именами, определяющими фиксированные числа). Какие из перечисленных ниже выражений являются абсолютными, переместимыми или сложными?

```
а.  A+I
в.  A-B
д.  A+K-B
ж.  <A-K>+<A-J>
```

```
б.  B-K
г.  A+B
е.  <A-B>+<A-C>
з.  <A+K>*<<B-C>*4>
```

5. В приведенной ниже программе появится ряд ошибок из-за неправильной очередности определений. Укажите, какие операторы неверно расположены и какие будут отмечены в сообщении об ошибке. (Заметим, что, хотя приведенный фрагмент не является программой, поскольку отсутствуют выполняемые операторы, переупорядочивание операторов приведет к тому, что ассемблирование пройдет без ошибок.)

```
I=3
A:      .BLKL   K
B:      .BLKL   D-C
C:      .BLKL   I
D:      .BLKL   K+I
K=J
J=I+5
        .END
```

6. Измените порядок операторов в программе п. 5 так, чтобы ее ассемблирование проходило без ошибок.

10.4. МАКРОИНСТРУКЦИИ

ПРОСТЫЕ МАКРОИНСТРУКЦИИ

Макроинструкции — это сложные блоки программного текста, которые могут быть многократно повторены в программе. В отличие от блоков повторения повторяющийся код, генерируемый с помощью макроинструкций, не обязательно должен весь находиться в одном месте, а может быть помещен в различных точках программы. К тому же в отличие от блоков повторений макроинструкции позволяют вносить существенные изменения в повторяющийся код.

В сущности, макроинструкции являются поименованными блоками текста программы. Везде, где в поле операции указывается имя макроинструкции, в тело программы будет вставлен (скопирован) соответствующий блок программного текста. Кроме того, в макроинструкции могут присутствовать *параметры* или имена, которые модифицируются или подставляются всякий раз, когда макроинструкция используется или, другими словами, происходит ее *вызов*.

Для примера заметим, что на ЭВМ семейства VAX не предусмотрена инструкция сложения 64-разрядных целых чисел. Предположим, что при решении некоторой задачи потребуется арифметика с двойной точностью (см. гл. 6). Поэтому при сложении квадраслов необходимо иметь эквивалент следующим строкам:

```
ADDL3   A, B, C
MOVL    A+4, C+4
ADWC    B+4, C+4
```

Было бы удобно, чтобы такой блок генерировался при появлении имени ADDQ3. Для этого сначала нужно определить этот блок как макроинструкцию с именем ADDQ3. Имена A, B и C будут использоваться как имена параметров. Можно применять любые произвольные символические имена, за исключением тех, которые уже задействованы в программе.

После того как будут выбраны имена, требуется определить макроинструкцию. Описание начинается с директивы .MACRO ADDQ3 A,B,C, за которой идут строки текста программы, которые будут генерироваться всякий раз при вызове макроинструкции. Для обозначения конца макроопределения используется директива .ENDM ADDQ3. В строке директивы .ENDM записывается имя макроинструкции ADDQ3, что помогает ассемблеру и программисту в использовании вложенных макроинструкций, рассмотренных ниже. На рис. 10.6 представлено полное макроопределение.

```
.MACRO  ADDQ3   A, B, C
ADDL3   A, B, C
MOVL    A+4, C+4
ADWC    B+4, C+4
.ENDM   ADDQ3
```

Рис. 10.6. Простое макроопределение

МАКРООПРЕДЕЛЕНИЕ И МАКРОРАСШИРЕНИЕ

Само по себе макроопределение не приводит к появлению в программе какого-либо нового текста, текст генерируется только при *вызове* макроинструкции. Макровызов состоит из имени макроинструкции, используемого так, как если бы это был код операции, после которого идут аргументы, которые должны быть подставлены вместо пара-

	Макровызов		Генерируемый код	
а.	ADDQ3	X, Y, Z	ADDL3 MOVL ADWC	X, Y, Z X+4, Z+4 Y+4, Z+4
б.	ADDQ3	A, B, C	ADDL3 MOVL ADWC	A, B, C A+4, C+4 B+4, C+4
в.	ADDQ3	COST, PROFIT, PRICE	ADDL3 MOVL ADWC	COST, PROFIT, PRICE COST+4, PRICE+4 PROFIT+4, PRICE+4
г.	ADDQ3	X+4, Y+8, Z+12	ADDL3 MOVL ADWC	X+4, Y+8, Z+12 X+4+4, Z+12+4 Y+8+4, Z+12+4

Рис. 10.7. Простые макровызовы и макрорасширения

метров макроопределения. На рис. 10.7 представлено несколько примеров макровызовов и порождаемого при их вызове программного кода. На рис. 10.7, п. *а* показан простой и понятный макровызов, в котором имена X, Y и Z подставляются вместо параметров A, B и C. На рис. 10.7, п. *б* имена A, B, C подставляются вместо самих себя, это не приводит ни к каким недоразумениям.

На рис. 10.7, *в* имена COST, PROFIT и PRICE подставляются вместо параметров A, B и C, демонстрируя тем самым, что число алфавитно-цифровых символов в имени аргумента не обязательно должно быть таким же, как в имени параметра. Ключевым моментом при этом является то, что производится подстановка символьных строк, а не адресов. Было бы неверно сказать, что адрес PRICE подставляется вместо адреса A. В действительности имя A может даже никогда не использоваться как адрес. Имеет место то, что макропроцессор подставляет вместо параметров *символьные строки* – имена аргументов. Полученный в результате текст обрабатывается ассемблером, как обычный текст программы на языке ассемблера.

На рис. 10.7, п. *г* показано, как осуществляется подстановка символьной строки. При этом символьные строки X+4, Y+8 и Z+12 подставляются вместо параметров A, B и C. Отметим, что подстановка является в чистом виде подстановкой символьной строки. Строка X+4 подставляется вместо параметра A даже в выражении A+4. Поэтому в результате подстановки будет получено X+4+4, а не X+8.

По тому как эта конкретная макроинструкция была записана (см. рис. 10.6), аргументы должны быть адресами или выражениями для вычисления адресов. Другие режимы адресации, например R0, @8(AP) и X[R1], использовать нельзя, поскольку могут быть порождены выражения следующего вида, недопустимые в рассматриваемом контексте:

```
R0+4
@8(AP)+4
X[R1]+4
```

Макроинструкция, представленная на рис. 10.8, является измененным вариантом макроинструкции рис. 10.6, который можно использовать с большинством режимов адресации.

Вследствие того что операнды инструкций помещаются для выполнения арифметических операций в стек, имена, используемые в качестве аргументов макровызова, не должны появляться в выражениях. Более того, они появляются только один раз в инструкциях пересылки квадраслова. Это важно в том случае, если подставляемый аргумент имеет вид


```
.MACRO   ADDQ3   A, B, C
MOVQ     B, -(SP)
MOVQ     A, -(SP)
ADDL2    (SP)+, 4(SP)
ADWC     (SP)+, 4(SP)
MOVQ     (SP)+, C
.ENDM    ADDQ3
```

Рис. 10.8. Другой вариант макроинструкции, приведенной на рис. 10.6

(R5)+ или даже X[R1]. Используя при вызове макроинструкции аргументы R0, @8(AP) и X[R1], можно записать

```
ADDQ3    R0, @8(AP), X[R1]
```

В результате получено следующее макрорасширение:

```
MOVQ     @8(AP), -(SP)
MOVQ     R0, -(SP)
ADDL2    (SP)+, 4(SP)
ADWC     (SP)+, 4(SP)
MOVQ     (SP)+, X[R1]
```

Использование стека в этом примере является специфическим приемом и требует тщательной проверки при адресации операндов (SP), (SP)+ и 4(SP). Читателю рекомендуется разобрать самостоятельно несколько примеров, наблюдая за содержимым стека при выполнении каждой инструкции (см. п. 5 упр. 10.2). Заметим, что данная макроинструкция не будет работать надлежащим образом, если при адресации операнда используется указатель стека SP. Тем не менее она будет работать со всеми другими режимами адресации.

МАКРОИНСТРУКЦИИ И ПАРАМЕТРЫ

Важно уяснить, что при вызове макроинструкций происходит их *расширение*, которое заключается в замещении каждого вхождения параметра символьной строкой соответствующего ему аргумента. Заметим, что параметры могут находиться в любом месте

(а) Макроопределение

```
        .MACRO   TEST   ABC, DEF, HIJ
ABC:    DEF      HIJ
        INCL     HIJ
        .ENDM    TEST
```

(б) Макровызов

```
        TEST     LOOP1, CLRL, COUNT
```

(в) Макрорасширение

```
LOOP1:  CLRL     COUNT
        INCL     COUNT
```

Рис. 10.9. Макроинструкция с замещаемыми параметрами

макроопределения и идентифицируются по имени, выделенному с помощью знаков пунктуации. На рис. 10.9 представлены макроопределение и макрорасширение, которые иллюстрируют, как производится замена параметров в различных полях макроопределения.

Заметим, во-первых, что строки текста в макроопределении сами по себе не являются корректным исходным текстом программы на ассемблере. В первой строке в поле кода операции содержится код DEF, инструкция с таким кодом отсутствует в ЭВМ семейства VAX. Однако DEF представляет собой замещаемый параметр и в макрорасширении не появляется, а замещается на код CLRL, который является допустимым кодом операции ЭВМ семейства VAX.

Другой важный момент, который демонстрируется в этом примере, заключается в том, что допускается подстановка меток. Действительно, почти всегда имеется необходимость в том, чтобы метки, присутствующие в макроопределении, были замещаемыми параметрами. Предположим, что метка ABC не является замещаемым параметром. Тогда если вызов макроинструкции производится два или более раз, то имя ABC появится в качестве метки более одного раза, что приведет к ошибке повторного определения.

10.5. УСЛОВНОЕ АССЕМБЛИРОВАНИЕ

ОПРЕДЕЛЕНИЕ УСЛОВНОГО АССЕМБЛИРОВАНИЯ

Часто, в особенности в больших программах, потребность в появлении некоторых фрагментов текста программы возникает не всегда, а только при определенных условиях. Примерами этого являются:

1) части программы, используемые для отладки, надобность в которых отпадает при эксплуатации программы;

2) макроинструкции и подпрограммы, которые требуются для одной модели ЭВМ, но не нужны или существенно отличаются для другой модели;

3) программный код, необходимый для поддержки дополнительных возможностей вычислительной системы;

4) части макроинструкции, которые должны быть включены в макрорасширение при одних значениях аргументов, но не нужны при других.

Ненужную часть текста для экономии памяти можно удалить из программы. Однако это не всегда легкая задача. Редактирование большой программы в лучшем случае – рискованное занятие. При этом всегда имеется опасность либо удалить лишнее, либо не то, что нужно. К тому же всегда бывает очень трудно восстановить первоначальный вариант, если вы отказались от изменения программы. Все эти проблемы можно решить с помощью *условного ассемблирования.*

Условное ассемблирование позволяет включить или проигнорировать при ассемблировании некоторый идентифицированный фрагмент текста программы. Такой фрагмент текста называется *блоком условного ассемблирования* и ограничивается двумя директивами ассемблера. Начало блока отмечается директивой .IF, а конец – директивой .ENDC. Директива .IF содержит описание логического условия. Если условие выполняется, фрагмент текста, находящийся в блоке условного ассемблирования, включается ассемблером в программу. В противном случае весь блок пропускается, так будто его не существовало.

ПРИМЕР УСЛОВНОГО АССЕМБЛИРОВАНИЯ

На рис. 10.10 показан простой пример блока условного ассемблирования. Значение, соответствующее имени TEST, определяет, будет или нет ассемблироваться данный блок. Если значение TEST *не равно* 0, то выполняется ассемблирование блока, в противном случае блок будет пропущен.

Следует особо отметить тот факт, что в блоке содержится определение метки XCODE. Следовательно, если этот блок будет пропущен, то ссылки на имя XCODE приведут к выдаче сообщения об ошибке появления неопределенного имени. Чтобы избежать этого, необходимо:

либо собрать все ссылки на метку XCODE в блоках условного ассемблирования, которые будут также *пропускаться*, когда значение символа TEST равно 0;

либо дать альтернативное определение метки XCODE, которое вставляется в том случае, когда значение TEST равно 0.


```
        .IF     NOT_EQUAL, TEST
XCODE:  MOVL    X, Y
        INCL    R0
        ADDL2   W, R1
        .ENDC
```

Рис. 10.10. Блок условного ассемблирования

Заметим также, что в данном блоке условного ассемблирования будут порождаться инструкции, что влияет на счетчик адреса. Поэтому имя TEST должно быть определено в программе заранее. Обычно имена, которые участвуют в управлении блоками условного ассемблирования, определяются с помощью знака равенства = в начале программы, где их легко обнаружить, если потребуется внести изменения.

На рис. 10.10 используется операция отношения NOT_EQUAL для проверки условия *не равно.* Как и следовало ожидать, можно использовать все шесть арифметических операций отношения: EQUAL, NOT_EQUAL, LESS_THEN, LESS_EQUAL, GREATER_EQUAL и GREATER (для операций отношения равно, не равно, меньше, меньше или равно, больше или равно и больше). Для всех этих отношений выполняется сравнение значения, заданного как аргумент выражения, с нулем. Синтаксис директивы .IF дан ниже:

```
.IF     ОТНОШЕНИЕ, ВЫРАЖЕНИЕ
```

Например, директива

```
.IF     GREATER_EQUAL, X-5
```

приведет к пропуску фрагмента текста программы, если только значение выражения X – 5 не будет больше или равно 0. Иначе говоря значение X должно быть больше или равно 5 для того, чтобы выполнилось ассемблирование фрагмента текста программы, помещенного в блок условного ассемблирования.

ДРУГИЕ КОДЫ УСЛОВНОГО АССЕМБЛИРОВАНИЯ

Дополнительно к описанным выше арифметическим условиям существует шесть символических условий:

1. Условие DEFINED. Истинно, если указанное как аргумент имя определено. Например, директива .IF DEFINED, XYZW приводит к порождению фрагмента текста, если имя XYZW определено и содержится в таблице имен.

2. Условие NOT_DEFINED. Противоположно условию DEFINED.

3. Условие BLANK. Истинно, если следующий за ним параметр заменяется пробелами. Например, рассмотрим следующую макрокоманду:

```
.MACRO  NULL    X, Y, Z
.IF     BLANK, Y
.LONG   X, Z
.ENDC
.ENDM   NULL
```

При макровызове NULL 5,ABC,6 не происходит порождения кода, тогда как при макровызове NULL 5, ,6 будет образовано два длинных слова. Заметим, что если в директиве .LONG имеется более одного аргумента, то происходит генерация нескольких длинных слов.

4. Условие NOT_BLANK. Это условие противоположно условию BLANK.

5. Условие IDENTICAL. Истинно, если следующие затем два параметра после подстановки аргументов макроинструкции окажутся идентичными строками символов.

6. Условие DIFFERENT. Противоположно условию IDENTICAL.

Очевидно, четыре последние директивы условного ассемблирования предназначены для использования в макроопределениях. Действительно, условное ассемблирование в основном используется в макроинструкциях. Условное ассемблирование может быть использовано для управления генерацией строк текста программы, которые ассемблируются в зависимости от значений аргументов или даже от числа обращений к макроинструкции. Например, рассмотрим макроинструкцию, предназначенную для генерации вызывающей последовательности для подпрограммы с именем SUBR. Подпрограмма имеет три аргумента, которые передаются в списке аргументов согласно стандартной вызывающей последовательности операционной системы VAX/VMS, которая была описана в предыдущей главе. Второй и третий аргументы в подпрограмме являются необязательными. Тем не менее подпрограмме SUBR все же нужен список аргументов, занимающий три длинных слова. Если аргумент передается в явном виде, то его адрес должен быть внесен в этот список; если аргумент опущен, то в список должен быть помещен фиктивный нулевой адрес. На рис. 10.11 показано, как можно написать такую макроинструкцию.

Заметим, что в этой макроинструкции проверяется, не пропущены ли аргументы, соответствующие параметрам B и C. Если это так, то в список аргументов заносится нулевое значение. В противном случае в стек помещается соответствующий адрес. Заметим, что первое должно быть сделано, при выполнении условия BLANK, второе -- при выполнении условия NOT_BLANK, что очень похоже на структуру IF-THEN-ELSE, используемую в языках высокого уровня. В действительности в ассемблере ЭВМ семейства VAX имеется аналог оператора ELSE для условного ассемблирования — это директива .IF_FALSE. Данная директива вызывает ассемблирование расположенного за ней фрагмента текста, если предыдущий фрагмент был пропущен, или вызывает пропуск фрагмента, если предыдущий фрагмент был ассемблирован. Имеется еще две похожие директивы условного ассемблирования: директива .IF_TRUE, которая ставит условное ассемблирование снова в зависимость от первоначального условия, и директива .IF_TRUE_FALSE, которая приводит к тому, что ассемблирование производится независимо от условия. На рис. 10.12 показано, каким образом можно переписать макроинструкцию, представленную на рис. 10.11, более кратко с использованием директивы .IF_FALSE; в результате отпадает необходимость в анализе противоположного условия.

```
.MACRO  DO_SUBR  A,B,C
.EXTERNAL  SUBR
.IF     NOT_BLANK,C
PUSHAL  C
.ENDC                   ;ПОМЕСТИТЬ В СТЕК АДРЕС C
.IF     BLANK,C
PUSHL   #0
.ENDC                   ;ИЛИ 0
.IF     NOT_BLANK,B
PUSHAL  B
.ENDC                   ;ПОМЕСТИТЬ В СТЕК АДРЕС B
.IF     BLANK,B
PUSHL   #0
.ENDC                   ;ИЛИ 0
PUSHAL  A
CALLS   #3,SUBR         ;ПОМЕСТИТЬ В СТЕК ТРЕБУЕМЫЙ АДРЕС A
.ENDM   DO_SUBR         ;И ВЫЗВАТЬ ПОДПРОГРАММУ
```

Рис. 10.11. Макроинструкция для вызова подпрограммы


```
.MACRO  DO_SUBR  A,B,C
.EXTERNAL  SUBR
.IF     NOT_BLANK,C
PUSHAL  C
.ENDC                   ;ПОМЕСТИТЬ В СТЕК АДРЕС C
.IF_FALSE
PUSHL   #0
.ENDC                   ;ИЛИ 0
.IF     NOT_BLANK,B
PUSHAL  B
.ENDC                   ;ПОМЕСТИТЬ В СТЕК АДРЕС B
.IF_FALSE
PUSHL   #0
.ENDC                   ;ИЛИ 0
PUSHAL  A
CALLS   #3,SUBR         ;ПОМЕСТИТЬ В СТЕК ТРЕБУЕМЫЙ АДРЕС A
.ENDM   DO_SUBR         ;И ВЫЗВАТЬ ПОДПРОГРАММУ
```

Рис. 10.12. Другой вариант макроинструкции для вызова подпрограммы

10.6. ВЛОЖЕННОСТЬ И РЕКУРСИЯ

Не вдаваясь в подробности, отметим, что блоки условного ассемблирования могут быть вложены в другие блоки условного ассемблирования, а внутри макроинструкции могут появляться макроопределения. Директивы .IF_.ENDC и .MACRO–.ENDM–, подобно скобкам должны использоваться парами. Такое объединение в пары позволяет создавать сложные структуры для построения комбинированных условий и для написания макроинструкций, в которых после их вызова определяются другие макроинструкции. Помимо этого использование имени макроинструкции в директиве .ENDM позволяет сделать структуру вложений более ясной.

Отметим также, что в макрорасширении может происходить вызов других макроинструкций. Это не создает никаких трудностей, поскольку после расширения макроинструкции управление передается собственно ассемблеру, который допускает появление дополнительных макровызовов. В результате будет создан дополнительный код, который просто добавится к уже имеющемуся.

Интересно, что при этом макроинструкция может вызывать сама себя. Это приводит к образованию цикла особого рода, в котором последовательно повторяется расширение

```
;
;МАКРОИНСТРУКЦИЯ ДЛЯ ГЕНЕРАЦИИ ТАБЛИЦЫ ПОСЛЕДОВАТЕЛЬНЫХ ЧИСЕЛ
;
        .MACRO  TABLE  N
TABK=1                          ;ИНИЦИАЛИЗАЦИЯ TABK
        TAB     N               ;НАЧАЛО ОБРАБОТКИ
        .ENDM   TABLE
;
;РЕКУРСИВНАЯ МАКРОИНСТРУКЦИЯ, ИСПОЛЬЗУЕМАЯ В TABLE
;
        .MACRO  TAB    N
        .IF     LESS_EQUAL,TABK-<N> ;ЗАВЕРШИТЬ ГЕНЕРАЦИЮ ЧИСЕЛ?
        .LONG   TABK                ;НЕТ, ОБРАЗОВАТЬ НОВОЕ ЧИСЛО
TABK=TABK+1                     ;РЕКУРСИВНОЕ ОБРАЩЕНИЕ
        TAB     N
        .ENDC   TABLE
        .ENDM   TAB
```

Рис. 10.13. Рекурсивная макроинструкция

макроинструкции. Однако, как и в обычных программных циклах, должен быть способ завершения цикла. Таким образом, если макроинструкция осуществляет вызов самой себя, то этот вызов должен находиться внутри блока условного ассемблирования, который в конце концов будет пропущен. Макроинструкции такого вида называются *рекурсивными макроинструкциями*, поскольку их свойства во многом аналогичны свойствам рекурсивных подпрограмм, описанных в гл. 9.

На рис. 10.13 представлена рекурсивная макроинструкция, предназначенная для генерации таблицы чисел от 1 до N, генерация таблицы осуществляется с помощью макровызова TABLE N. Заметим, что макроинструкция TABLE обращается к макроинструкции TAB, которая является рекурсивной. В строке с директивой .IF число N заключено в скобки, поскольку вместо него может быть подставлено выражение.

УПРАЖНЕНИЯ 10.2

1. Пусть даны макроопределения

```
.MACRO  ORD    A,B
MOVL    A,B
CLRL    A
SUBL    B,A
.ENDM   ORD
.MACRO  SPEC   A,B,C
MOVL    #A,AA
B       A,C
CLRB    C+6
.ENDM   SPEC
```

Покажите, как будут выглядеть макрорасширения, полученные в результате следующих макровызовов:

```
а.  ORD    SUM,TOTAL
в.  ORD    A[R0],B
А.  SPEC   XMAX,MOVL,W
ж.  SPEC   X,ORD,1000
б.  ORD    R0,(R1)+
г.  ORD    B,A
е.  SPEC   1000,ADDL2,C+6
з.  SPEC   B,ORD,A
```

2. Исходя из предположения, что имена A и B были определены как A = 5 и B = 7, покажите, будут ли ассемблированы следующие фрагменты текста программы, помещенные в блоки условного ассемблирования, а если будут, то какие:

```
а.  .IF    EQUAL,A-3
    MOVL   X,Y
    .ENDC
в.  .IF    GREATER,B
    MOVL   R,R0
    .ENDC

А.  C=0
    C=1
           .IF    EQUAL,B
           .IF    EQUAL,C
           MOVL   U,V
           .ENDC
           MOVL   L,M
           .ENDC
```

```
б.  .IF    NOT_EQUAL,A-3
    MOVL   #3,W
    .ENDC
г.  C=0
           .IF    LESS_THAN,B-4
    C=1
           .ENDC
           .IF    EQUAL,C
           MOVL   H,Q
           .ENDC
е.  C=0
           .IF    NOT_EQUAL,B
    C=1
           .IF    EQUAL,C-1
           MOVL   I,J
           .ENDC
           MOVL   G,F
           .ENDC
```


3. Напишите вариант макроинструкции ADDQ3, представленной на рис. 10.6, имеющей два параметра. Эта макроинструкция называется ADDQ2 и предназначается для сложения квадраслов; первый аргумент складывается со вторым, так же как в инструкции ADDL2 складывалось содержимое двух длинных слов. Какие режимы адресации допускаются в вашей макроинструкции? Можно ли добиться того, чтобы макроинструкция работала со всеми режимами адресации?

4. Переписать макроинструкцию из п. 3 таким образом, чтобы она включала директивы условного ассемблирования, которые выявляли бы случай, когда первый аргумент равен #1. При этом порождаемый код может быть значительно упрощен. Например, вместо инструкции ADDL2 может быть использована инструкция INCL и не нужна будет обработка первого операнда.

5. На рис. 10.8 было дано следующее макроопределение:

```
.MACRO   ADDQ3    A,B,C
MOVQ     B,-(SP)
MOVQ     A,-(SP)
ADDL2    (SP)+,4(SP)
ADWC     (SP)+,4(SP)
MOVQ     (SP)+,C
.ENDM    ADDQ3
```

Напишите расширение для макровызова

ADDQ3 X,Y,Z

Последовательно разберите инструкции, сгенерированные в макрорасширении, в том порядке, в каком они будут следовать на стадии выполнения. Покажите, как меняется содержимое квадраслов X, Y и Z, а также стека при выполнении каждой инструкции.

6. Вообразим, что имеется простая ЭВМ с архитектурой, подобной архитектуре ЭВМ семейства VAX, память ЭВМ состоит из 32-битовых длинных слов, содержимое которых интерпретируется как числа в дополнительном коде. В этой ЭВМ имеется только один регистр общего назначения, который называется *сумматором*. Все операции должны выполняться с использованием сумматора, всего имеется десять инструкций, а именно:

LDA	M	Загрузить содержимое ячейки памяти M в сумматор
STA	M	Занести содержимое сумматора в ячейку памяти M
ADDA	M	Сложить содержимое ячейки M с содержимым сумматора
SUBA	M	Вычесть содержимое ячейки M из содержимого сумматора
JUMP	M	Перейти по адресу M
JMI	M	Перейти по адресу M, если содержимое сумматора отрицательно
JZ	M	Перейти по адресу M, если содержимое сумматора равно 0
READ		Ввести числа и загрузить в сумматор
PRINT		Напечатать содержимое сумматора
STOP		Останов

Следующая программа распечатывает первые десять степеней числа два, вычисляя выражение 2X как X+X:

```
        .TITLE  ПРОГРАММА ГИПОТЕТИЧЕСКОЙ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ
START:  LDA     MTEN     ;ИНИЦИАЛИЗИРОВАТЬ СЧЕТЧИК
        STA     COUNT    ;ЗНАЧЕНИЕМ -10
        LDA     ONE      ;ИНИЦИАЛИЗИРОВАТЬ POWER
        STA     POWER
LOOP:   LDA     POWER    ;УМНОЖИТЬ POWER НА 2
        ADDA    POWER
        STA     POWER
        PRINT            ;ПЕЧАТЬ POWER
```

```
        LDA     COUNT       ;ИЗМЕНИТЬ СОДЕРЖИМОЕ СЧЕТЧИКА
        ADDA    ONE
        STA     COUNT
        JMI     LOOP        ;ПОКА ОТРИЦАТЕЛЬНОЕ, ПОВТОРЯТЬ В ЦИКЛЕ
        STOP                ;ЗАВЕРШИТЬ ВЫПОЛНЕНИЕ
MTEN:   .LONG   -10         ;НАЧАЛЬНОЕ ЗНАЧЕНИЕ СЧЕТЧИКА
ONE:    .LONG   1           ;КОНСТАНТА ONE
COUNT:  .BLKL   1           ;ОБЛАСТЬ ПЕРЕМЕННЫХ
POWER:  .BLKL   1           ;ОБЛАСТЬ ПЕРЕМЕННЫХ
        .END    START
```

Напишите набор макроинструкций, которые имитировали бы работу такой гипотетической машины, заменяя каждую инструкцию инструкциями ЭВМ VAX с эквивалентным действием. Проверьте действие макроинструкций на приведенном примере программы.

7. Напишите программу для гипотетической машины, описанной в п. 6 упр. 10.2, которая бы считывала два числа со знаком и перемножала их между собой, получая в результате число со знаком. Можно ли написать эффективную программу, несмотря на то, что в гипотетической машине отсутствует инструкция сдвига? Проверьте работу программы с помощью макроинструкций, которые были написаны для п. 6.

10.7. БОЛЕЕ СЛОЖНЫЕ ПРИЕМЫ НАПИСАНИЯ МАКРОИНСТРУКЦИЙ

КЛЮЧЕВЫЕ ПАРАМЕТРЫ

В ассемблере ЭВМ семейства VAX используются два способа идентификации аргумента в макровызове. Более простым является тот, который уже был рассмотрен. При этом способе аргумент идентифицируется по его позиции в макровызове. Параметры, обеспечивающие передачу аргументов таким образом, называются *позиционными.*

Например, в следующих макроопределении и макровызове:

```
.MACRO  MAC     A,B,C
. . .
.ENDM   MAC
. . .
MAC     XXX,YYY,ZZZ
```

произойдет подстановка аргумента XXX вместо параметра A, YYY – вместо B, а ZZZ – вместо C. То, как устанавливается соответствие, зависит от порядка следования аргументов, указанных в макровызове.

Другой способ называется передачей аргументов с помощью *ключевых параметров.* В этом случае соответствие параметров, используемых в исходном макроопределении, аргументам макровызова устанавливается с помощью знака равенства. Например, для предыдущего примера следующий вызов:

```
MAC     B=XXX,C=YYY,A=ZZZ
```

снова приведет к подстановке аргумента XXX вместо параметра A, YYY – вместо B и ZZZ – вместо C. Поскольку подстановка идентифицируется с помощью имени или ключевого слова, очередность указания аргументов не имеет значения и на самом деле в данном примере порядок следования аргументов намеренно нарушен. Цена, которую приходится платить за возможность нарушать очередность, такова, что приходится вводить с терминала больше символов. Но часто это дает такие преимущества, которые перевешивают необходимость набора дополнительных символов.

Основная причина использования макровызовов такого типа заключается в том, что сложные макроинструкции, подобные системным макроинструкциям, описанным в гл. 11,

часто имеют большое количество параметров и трудно запомнить очередность, в которой они были специфицированы в исходном макроопределении. Это усугубляется тем, что программное обеспечение часто обновляется, в результате чего возникает необходимость в модификации таких макроинструкций. Другая причина заключается в том, что очень часто многие из параметров сложной макроинструкции являются необязательными, и их нужно задавать только в особых случаях. В результате при позиционной форме записи потребовалось бы большое количество запятых для обозначения позиции неиспользуемых или опущенных аргументов. Для примера рассмотрим макроинструкцию с именем GENERATE, в которой имеется 15 замещаемых параметров. Если в конкретном макровызове задаются только два аргумента, 6-й и 13-й, то при позиционном способе записи макровызов имел бы вид

```
GENERATE  ,,,,,FTAB,,,,,,XLOC
```

Заметим, что поставлено меньше 14 запятых, так как XLOC подставляется вместо 13-го параметра, а поскольку ничего не нужно подставлять вместо 14-го и 15-го параметров, то последние запятые не нужны. Однако если 6-му параметру было присвоено имя FUNC, а 13-му – VALUE, то макровызов с явным указанием имен параметров можно было бы осуществить следующим образом:

```
GENERATE  FUNC=FTAB,VALUE=XLOC
```

При таком способе записи нет необходимости знать, сколько параметров было в исходном макроопределении и в каком порядке они были записаны. А использование в качестве ключевых слов мнемонических обозначений типа FUNC и VALUE позволяет сделать программу более понятной.

Поэтому в руководствах для системного программиста операционной системы VAX/VMS обычно предпочитают определять системные макроинструкции с использованием ключевых, а не позиционных параметров. Другие примеры того, как это делается, приведены в следующей главе, в которой описываются системные макроинструкции ввода-вывода.

ПОДСТАНОВКА ЗНАЧЕНИЙ ПАРАМЕТРОВ ПО УМОЛЧАНИЮ

Как видно из предыдущего раздела, не столь уж редки случаи, в особенности для системных макроинструкций, когда в макровызовах пропущен аргумент. Обычно пропуск аргумента приводит к тому, что вместо соответствующего ему параметра не происходит ***никакой* подстановки. Имя параметра уничтожается, и на место ничего не помещается.** Например, рассмотрим макроопределение

```
.MACRO  ADDX    A,B,C
ADDL3   A,B,C
.ENDM   ADDX
```

Если макровызов имеет вид

```
ADDX    SUM,,TOTAL
```

то расширением будет

```
ADDL3   SUM,,TOTAL
```

что является ошибочным (конечно, если только не было произведено переопределение имени ADDL3 в качестве макроинструкции). В некоторых случаях может потребоваться, чтобы макроинструкция, подобная этой, при пропуске аргумента приобретала особый смысл. Например, такую макроинструкцию можно определить таким образом, чтобы в случае пропуска среднего аргумента она порождала бы инструкцию, которая прибавляет к первому аргументу 1. Этого можно достичь с помощью условного ассемблирования с использованием директивы .IF BLANK, . . . Однако при этом обычно требуется добавить

в макроопределение небольшое количество дополнительных строк. Например, предыдущее макроопределение примет вид

```
.MACRO  ADDX    A,B,C
.IF     BLANK,B
ADDL3   A,#1,C
.IF_FALSE
ADDL3   A,B,C
.ENDC
.ENDM   ADDX
```

Очевидно, что использование этого приема для сложной макроинструкции с несколькими параметрами, имеющими особый смысл в случае пропуска аргументов, может потребовать очень большой объем дополнительного кода. Если для каждого пропущенного аргумента удваивается объем кода, то может потребоваться 2^{15} строк текста программы.

Существует более простой способ – *подстановка значения параметра по умолчанию*. При такой подстановке для каждого параметра заранее задается строка символов. Если в макровызове аргумент опущен, то вместо него по умолчанию используется эта строка. Строки, подставляемые по умолчанию, помещаются в список параметров макроопределения и помечаются знаком равенства. Чтобы продемонстрировать этот способ, снова рассмотрим макроинструкцию ADDX, которую можно переопределить следующим образом:

```
.MACRO  ADDX    A,B=#1,C
ADDL3   A,B,C
.ENDM   ADDX
```

В данном случае параметру B для подстановки по умолчанию назначена строка #1. Поэтому для макровызова

```
ADDX    SUM,,TOTAL
```

будет порождено макрорасширение

```
ADDL3   SUM,#1,TOTAL
```

Однако если аргумент, соответствующий параметру B, **указан, то** подстановка происходит обычным способом, так что макровызов

```
ADDX    SUM,FIVE,TOTAL
```

порождает макрорасширение

```
ADDL3   SUM,FIVE,TOTAL
```

ПОРОЖДЕНИЕ ЛОКАЛЬНЫХ ИМЕН

В этом подразделе описывается другой вид подстановки по умолчанию в случае пропущенных или пустых аргументов. Напомним, что выше в этой главе рекомендовалось соблюдать осторожность при использовании меток внутри макроопределения. Причина этого заключается в том, что, если обращение к макроинструкции производится в программе несколько раз, обращение к такой метке вызывает ошибку повторного определения. Одно из решений проблемы – сделать так, чтобы метка стала замещаемым параметром. В этом случае программист при вызове макроинструкции всякий раз мог бы назначать для метки свое собственное уникальное символическое имя. Например, рассмотрим следующую макроинструкцию, производящую очистку массива из N длинных слов:

```
        .MACRO  CLEAR   A,N,L
        CLRL    R0
L:      CLRL    A[R0]
        AOBLSS  N,R0,L
        .ENDM   CLEAR
```


В случае макровызова

```
CLEAR   ARRAY,#20,30$
```

макрорасширение примет вид

```
        CLRL    R0
30$:    CLRL    ARRAY[R0]
        AOBLSS  #20,R0,30$
```

Этот прием действует безотказно. Однако при этом требуется, чтобы программист постоянно заботился о таких элементах макроинструкции, которые сами по себе несущественны с точки зрения общего назначения макроинструкции, а именно о подборе уникального имени для метки L. При выборе подходящего для этой цели аргумента нужно соблюдать осторожность. В данном случае, например, важно, чтобы символическое имя 30$ не использовалось бы в качестве метки в другом месте данного блока локальных имен.

Решение этой проблемы заключается в том, чтобы возложить на ассемблер функцию генерации локальных имен. Этого можно добиться, помещая знак вопроса перед именем параметра в строке макроопределения. В частности, вышеупомянутая макроинструкция может быть переопределена следующим образом:

```
        .MACRO  CLEAR   A,N,?L
        CLRL    R0
L:      CLRL    A[R0]
        AOBLSS  N,R0,L
        .ENDM   CLEAR
```

Если параметр помечен знаком вопроса, а в макровызове опускается соответствующий аргумент, то ассемблером будет сгенерировано собственное локальное имя. Все генерируемые локальные имена записываются в виде большого числа (не менее 30 000), после которого следует знак доллара. Ассемблер начинает с числа 30 000 и использует последовательно числа 30 001, 30 002, . . . каждый раз, когда требуется сгенерировать новое имя. Например, макровызов

```
CLEAR   ARRAY,#20
```

порождает макрорасширение

```
        CLRL    R0
30015$: CLRL    ARRAY[R0]
        AOBLSS  #20,R0,30015$
```

Из этого следует, что перед этим было сгенерировано 15 локальных имен и таким образом счет был доведен до 30 015.

Разумеется, могут быть случаи, когда программисту потребуется получить доступ к метке, находящейся вне макроинструкции. Тогда задание имени метки в явном виде подавляет генерацию локальных имен. Так, для макровызова

```
CLEAR   ARRAY,#20,30$
```

макрорасширение будет таким же, как и в предыдущем случае, однако вместо L будет подставлена метка 30$ и не произойдет генерации локального имени.

ОБРАБОТКА СИМВОЛОВ В МАКРОИНСТРУКЦИЯХ

Обычно имена, используемые для замещаемых параметров в макроинструкциях, идентифицируются по отделяющим их пробелам или с помощью специальной пунктуации. Так, например, в макроинструкции

```
.MACRO  EVER    A,B
MOVB    A,B
.ENDM   EVER
```

замена буквы B в слове MOVB производиться не будет, а параметр B, указанный в поле операндов этой инструкции, будет замещен.

В большинстве случаев желаемый результат может быть получен и таким способом, но иногда бывает полезным способ создания символических имен путем *конкатенации*, или объединения символьных строк. Этот способ реализуется в макроинструкциях с использованием апострофа, который выполняет в макрорасширениях особую функцию. Апостроф является знаком пунктуации и, следовательно, ограничителем для имени параметра. Однако в макрорасширении апострофы, примыкающие к имени замещаемого параметра, уничтожаются. Следующая макроинструкция поясняет использование апострофа:

```
.MACRO  SLIDE   A,B,X
MOV'X   A,B
.ENDM   SLIDE
```

Апостроф в символьной строке MOV'X – ограничитель для X, который является замещаемым параметром, и, поскольку апостроф примыкает к замещаемому параметру, он будет уничтожен. Это означает, что в результате макровызова

```
SLIDE   #5,DATA,L
```

будет получено макрорасширение

```
MOVL    #5,DATA
```

Иногда желательно сохранить апостроф в макрорасширении. Это возможно, поскольку апостроф, не примыкающий к замещаемому параметру, в макрорасширении сохраняется. При этом иногда может потребоваться использовать в макроопределении два или три апострофа, чтобы в результате был сохранен один. Например, макроопределение и макровызов

```
.MACRO  STRING  X,Y
.ASCII  /X'''Y/
.ENDM   STRING
STRING  DON,T
```

порождают макрорасширение

```
.ASCII  /DON'T/
```

Использование апострофов продиктовано желанием исключить лишние знаки пунктуации. Иногда возникает обратная задача. В макроинструкцию требуется передать строку-аргумент, содержащую знаки пунктуации, например запятые. Это вызывает определенные трудности, поскольку символьная строка A, B воспринимается как два фактических параметра, а не как один. Решением является использование угловых скобок 〈 и 〉 , в которые заключается аргумент в макровызове; при этом угловые скобки удаляются, а любая строка, находящаяся внутри скобок, используется в качестве строки подстановки макроинструкции. Строка может содержать пробелы, запятые и даже парные угловые скобки. В качестве примера использования этого приема рассмотрим макроинструкцию

```
.MACRO  WHAT    A,B
A       B
.ENDM   WHAT
```

Если ее вызов осуществляется следующим образом:

```
WHAT    MOVL,<SRC,DEST>
```


то макрорасширение будет иметь вид

```
MOVL    SRC,DEST
```

Заметим, что один уровень вложенности скобок убирается, но если внутри имеются другие скобки, они сохраняются. Это можно продемонстрировать на примере макровызова

```
WHAT    WHAT,<ADDL3,<R0,#2,R5>>
```

Расширением для него будет

```
WHAT    ADDL3,<R0,#2,R5>
```

что, в свою очередь, приведет к такому расширению:

```
ADDL3   R0,#2,R5
```

Имеется возможность назначения альтернативных ограничителей, чтобы в результате в строке-аргументе можно было использовать непарные угловые скобки. (Подробности см. в руководстве "*VAX/VMS Macro Language Referense Manual*".)

СТРОКИ ПРОДОЛЖЕНИЯ

Как следует из предыдущего раздела, макроопределения и макровызовы могут содержать большие обьемы информации. Может оказаться, что в строке не хватает места для размещения всех параметров, значений, принимаемых по умолчанию, либо аргументов в макроопределениях или макровызовах. Решением этой проблемы является использование строк продолжения, которые выполняют те же самые функции, что и строчки продолжения в программах на Фортране. Способ формирования строчек продолжения в языке ассемблера ЭВМ семейства VAX сводится к тому, что если последний аргумент, указанный в строке, состоит только из знака "минус", то следующая строка рассматривается в качестве строки продолжения. Допускается наличие нескольких строк продолжения, каждая из которых имеет в конце знак "минус". Знаки "минус" при конечной интерпретации строки кода уничтожаются. Например, макровызов

```
DOMAC   A=ARG1,B=ARG2,C=ARG3,D=ARG4
```

может быть записан в виде

```
DOMAC   A=ARG1,-
        B=ARG2,-
        C=ARG3,-
        D=ARG4
```

Точка с запятой, которая используется перед комментарием, действует как маркер конца строки. Поэтому комментарии могут быть записаны в любой из строк продолжения. Например, предыдущий макровызов может быть переписан в следующем виде:

```
DOMAC   A=ARG1,-                ;ПЕРВЫЙ АРГУМЕНТ
        B=ARG2,-                ;ВТОРОЙ АРГУМЕНТ
        C=ARG3,-                ;И Т.Д.
        D=ARG4
```

УПРАЖНЕНИЯ 10.3

1. Пусть макроопределение начинается со строки

```
.MACRO  PROB EXP,TOTAL,SUM,MAX
```

Покажите, как можно переписать каждый из последующих макровызовов, используя способ передачи аргументов с помощью ключевых параметров вместо способа передачи аргументов с помощью позиционных параметров:

```
а.  PROB    A,B,C,D
б.  PROB    A+5,B(R5)[R3],-(SP),#27
в.  PROB    A,,,D
г.  PROB    ,R5,<A,B>,LONGSYMBOL
д.  PROB    ,,(SP)+
е.  PROB    <A,B,C,D>
```

2. Пусть макроопределение начинается со строки

```
.MACRO  SECOND CHAN,ADS,ORK,W
```

Преобразовать следующие макровызовы с ключевыми параметрами в эквивалентные макровызовы с позиционными параметрами:

```
а.  SECOND  CHAN=A,ADS=B,ORK=C,W=D
б.  SECOND  ADS=A,ORK=B,CHAN=C,W=D
в.  SECOND  ADS=ORK,ORK=W,CHAN=ADS,W=CHAN
г.  SECOND  ADS=(R5+,W=-(SP)
д.  SECOND  CHAN=<A,B>
е.  SECOND  W=<A,B>
```

3. Пусть дано макроопределение

```
        .MACRO  TEST    ARRAY=BLOCK,SIZE=#100,ANS=R0,?LABEL
        CLRB    ANS
        PUSHR   #^M<R1,R2>
        MOVAB   ARRAY,R1
        CLRL    R2
LABEL:  XDRB    (R1)+,ANS
        AOBLSS  SIZE,R2,LABEL
        POPR    #^M<R1,R2>
        .ENDM   TEST
```

Покажите, как будут выглядеть макрорасширения для следующих макровызовов, если в качестве генерируемого локального имени будет использоваться 30 000$:

```
а.  TEST    ARRAY=LIST
б.  TEST    SIZE=#25
в.  TEST    ARRAY=BLOCK,SIZE=SIZE
г.  TEST
д.  TEST    A,B,C,D
е.  TEST    LABEL=X,ANS=CSUM
```

4. Пусть дано макроопределение

```
.MACRO  WHATEVER  A
A
.ENDM   WHATEVER
```

Покажите, как будет выглядеть окончательное расширение для последующих макровызовов (регистрируйте все промежуточные шаги):


```
а.  WHATEVER  HALT
б.  WHATEVER  <ADDL3  A,B,C>
в.  WHATEVER  <3$:  ADDL2 1,R0>
г.  WHATEVER
д.  WHATEVER  WHATEVER
е.  WHATEVER  <WHATEVER <CLRL R0>>
```

5. Напишите макроинструкцию, расширение которой содержит инструкции для сложения четырех длинных слов A, B, C и D, а результат помещается в ANS. Макроинструкция должна иметь пять замещаемых параметров A, B, C, D и ANS. Для каждого из этих параметров предусмотрены следующие значения, принимаемые по умолчанию:

```
A       X
B       #100
C       (R1)+
D       #0
ANS     R0
```

ГЛАВА 11. СИСТЕМНЫЙ ВВОД И ВЫВОД

11.1. ВВЕДЕНИЕ

В гл. 8 была описана упрощенная форма ввода-вывода, которая позволяла программистам вводить и выводить символьные строки на свой терминал. В настоящей главе будет более детально рассмотрено использование системных сервисных средств ввода-вывода операционной системы VAX/VMS, позволяющих осуществлять полное управление форматами данных, обеспечивать работу с различными периферийными, а также с логическими устройствами. Кроме того, читатель познакомится со структурой файлов, операциями работы с файлами, методами обработки ошибок, которые могут встретиться в процессе ввода-вывода, а также с другими системными операциями.

Существует ряд средств, с помощью которых такая развитая операционная система, как VAX/VMS, предоставляет пользователям возможность выполнения ввода-вывода. Эти средства предназначены для работы на различных уровнях в зависимости от потребностей пользователя. На верхнем уровне располагаются средства ввода-вывода, применяемые в языках высокого уровня, таких как Фортран с его операторами READ, WRITE, FORMAT и др. Предполагается, что читатель в той или иной степени уже знаком с такой формой ввода-вывода. На нижнем уровне находятся операции ввода-вывода, реализованные на машинном языке, что связано с установкой битов и полей в специальных регистрах, являющихся составной частью каждого периферийного устройства. Фактически выполнение ввода и вывода на любом уровне в конце концов сводится к выполнению операций на самом нижнем уровне. Однако в многопользовательских системах, таких как VAX/VMS, имеются механизмы защиты, предотвращающие непосредственное обращение пользователей к нижнему уровню ввода-вывода, поскольку оно могло бы помешать выполнению операций ввода-вывода других пользователей. В результате только сама операционная система может иметь доступ к регистрам устройств. Как работают некоторые из этих функций, описано в гл. 15.

Если программисты, работающие с языком ассемблера ЭВМ семейства VAX, не имеют системных привилегий, они могут выполнять операции ввода-вывода на двух уровнях, занимающих промежуточное положение между операциями, реализуемыми на машинном языке и языках высокого уровня. Первый из этих методов основан на использовании подсистемы *ввода-вывода с очередями*, называемой QIO, второй метод – на использовании *службы управления записями*, называемой RMS. Подсистема QIO работает на более низком из этих двух уровней и позволяет осуществлять ввод неструктурированной информации с устройства или вывод на него. Подсистема RMS работает на более высо-

ком уровне и обеспечивает выполнение операций с файлами. В данной главе будут рассмотрены некоторые основные операции обеих систем, однако читателю следует помнить, что эти темы весьма сложны и наше рассмотрение ни в коей мере не является исчерпывающим. За более подробной информацией следует обратиться к соответствующим руководствам ("VAX-11 Record Manager Services Reference Manual", "VAX/VMS System Services Reference Manual", "VAX/VMS I/O Users Guide".)

11.2. ИСПОЛЬЗОВАНИЕ СИСТЕМНОГО СЕРВИСА

Обычный способ использования большинства системных функций, таких как операции ввода-вывода, заключается в вызове системных сервисных подпрограмм. В операционной системе VAX/VMS системные сервисные подпрограммы могут вызываться как из программ, написанных на языках высокого уровня, так и из программ на языке ассемблера. Поскольку достаточно часто вызывающие последовательности бывают весьма сложными, для упрощения работы с этими подпрограммами предусмотрены системные макроинструкции. Макроопределения этих макроинструкций находятся в библиотечном файле, в котором осуществляется поиск всякий раз, когда в программе встречается макроинструкция, не определенная внутри самой этой программы. Чтобы отличить системные макроинструкции от макроинструкций, определенных пользователем, имена системных макроинструкций начинаются со знака доллара. Примером такой системной макроинструкции, уже применяемой ранее, является $EXIT_S. В данном случае сочетание _S показывает, что ее макрорасширение будет содержать инструкцию CALLS. Существует аналогичная макроинструкция $EXIT_G, в которой используется инструкция CALLG. Большинство описанных в данной главе макроинструкций имеют формы _S и _G. Макроинструкции формы _S имеют обычно большие списки аргументов, которые описывают то, что должно быть занесено в стек перед выполнением инструкции CALLS. Макроинструкции формы _G обычно имеют только один аргумент, который является указателем на список аргументов, но при этом подразумевается, что список аргументов должен быть сформирован отдельно. Для простоты в этой главе будет использоваться более легкая в обращении, хотя иногда и менее эффективная, форма _S макроинструкций.

Напомним об употреблении знака подчеркивания. В программах на языке ассемблера и знак доллара, и знак подчеркивания могут присутствовать в метках, символических именах и именах макроинструкций. Эти знаки входят в алфавит языка ассемблера. Программисты могут пользоваться ими как знаками псевдопунктуации, разделяя тем самым длинные символические имена на несколько частей, несущих смысловую нагрузку. Знак подчеркивания можно применять не только в именах системных программ. Его применение для разбиения длинных имен даже поощряется, так как повышается их удобочитаемость. Например, символическое имя INPUT_OUTPUT_ROUTINE предпочтительнее имени INPUTOUTPUTROUTINE. С другой стороны, знак доллара зарезервирован для системного использования и не должен применяться в именах, определенных пользователем, так как может возникнуть путаница.

Большинство системных макроинструкций при их расширении образуют вызывающие последовательности для системных подпрограмм. Конечно, вызывающие последовательности могут быть сформированы и без применения макроинструкций, но все-таки макроинструкции существенно упрощают работу программиста. В вызывающих последовательностях используется стандартная форма списков аргументов, описанная в гл. 9, и, следовательно, эти подпрограммы могут вызываться программами, написанными на языках высокого уровня, таких как Фортран или Паскаль. Список аргументов может быть помещен либо в стек, либо в массив в фиксированной области памяти.

Хотя большинство макроинструкций используется для генерации вызывающих последовательностей для системных подпрограмм, некоторые системные макроинструкции предназначены для генерации таблиц данных. Двумя из таких макроинструкций являются $FAB и $RAB, применяемые для определения атрибутов файлов и записей при работе

с подсистемой RMS, описанной далее в этой главе. Другие макроинструкции позволяют сформировать списки аргументов, когда применяются макроинструкции формы _G.

Кроме формирования таблиц данных и вызывающих последовательностей для подпрограмм, многие системные макроинструкции представляют собой длинные списки определений символических имен. Эти символические имена программисты могут использовать вместо числовых кодов для обозначения операций или указания относительного расположения полей данных в таблицах. Например, символическое имя IO$_READVBLK может использоваться в макровызове $QIOW_S для обозначения того, что должна быть выполнена функция 010 — чтение виртуального блока.

11.3. ЗАПРОСЫ ВВОДА-ВЫВОДА С ОЧЕРЕДЯМИ

Запросы ввода-вывода с очередями используются для выполнения простых функций ввода-вывода неструктурированной информации, поступающей с устройства ввода или на устройство вывода. Термин *устройство* применяется для обозначения источника ввода или приемника вывода, подключенного к ЭВМ, например телетайпа, устройства печати, магнитного диска или ленты. В данном разделе работа с устройствами ввода-вывода будет рассмотрена только на примере работы с терминалом пользователя.

НАЗНАЧЕНИЕ КАНАЛА ВВОДА-ВЫВОДА

Конкретные устройства идентифицируются определенными в системе именами. Имена SYS$INPUT и SYS$OUTPUT обычно применяются для обозначения терминала пользователя. Имя SYS$INPUT используется для обозначения входного потока данных, а имя SYS$OUTPUT – выходного потока данных. Когда системе передается имя устройства, она должна найти информацию, описывающую параметры, необходимые для доступа к указанному устройству. Ясно, что было бы очень неэффективно осуществлять поиск этой информации при каждом обращении к устройству. Поэтому система разработана так, что поиск выполняется только один раз, а информация помещается во внутренний массив и становится доступной в дальнейшем. Каждая запись в этом массиве содержит информацию, необходимую для работы с устройством, или указатель на место, где эта информация может быть найдена. Такие записи в массиве называются *каналами*, а их относительное местоположение в массиве определяет *номер канала.* Для того чтобы система осуществила поиск информации об устройстве и назначила номер канала, применяется системная макроинструкция $ASSIGN_S. Эта макроинструкция имеет два обязательных ключевых параметра DEVNAM и CHAN. Параметр DEVNAM – это адрес дескриптора символьной строки, которая задает имя устройства (см. гл. 8 и 9 по поводу описания директивы .ASCID – дескриптора символьной строки). Параметр CHAN – это адрес длинного слова, в которое будет помещен номер канала, назначенный системой. На рис. 11.1 показано, как в программе может быть выполнено назначение канала для устройства SYS$INPUT, в данном случае система поместит номер канала в длинное слово CH1. В дополнение к двум обязательным параметрам, показанным в макровызове $ASSIGN_S на рис. 11.1, имеются два необязательных. Они служат для задания режима доступа и имени почтового ящика.

ЭВМ семейства VAX имеют четыре режима доступа: пользователя, супервизора, управления и ядра. Параметр "режим доступа" позволяет программе изменять привилегии опе-

```
CH1:      .BLKL     1
CHNAME:   .ASCID    "SYS$INPUT"
          .
          .
          .
          $ASSIGN_S   DEVNAM=CHNAME,CHAN=CH1
```

Рис. 11.1. Назначение номера канала

рации для назначенного канала в сторону уменьшения привилегий режима доступа. Таким образом обеспечиваются средства защиты от системных программ, вызываемых для выполнения операций программами пользователей, имеющих меньшие привилегии доступа. Но поскольку наиболее вероятно, что читатели книги имеют привилегии самого нижнего уровня, т.е. доступ в режиме пользователя, то отпадает необходимость в указании режима доступа.

Параметр "имя почтового ящика" используется для назначения почтового ящика для приема сообщений об ошибках или каких-либо других сообщений о канале. Почтовые ящики – это псевдоустройства ввода-вывода, которые позволяют обеспечить асинхронный обмен сообщениями между программами или процессами. В таком случае почтовый ящик должен быть настроен таким образом, чтобы он смог зарегистрировать изменения состояния канала, происходящие даже тогда, когда в этот момент времени не выполняются операции ввода-вывода. Например, предположим, что назначенное каналу устройство – это телефонная линия связи. Почтовый ящик может использоваться в данном случае для информирования программы о том, что абонент разорвал соединение. Предположительно, что рассоединение линии связи может произойти в любой момент времени, но программа незамедлительно получит уведомление об этом через почтовый ящик. Однако вопросы использования почтовых ящиков выходят далеко за рамки этой книги и больше здесь рассматриваться не будут.

ФУНКЦИИ ВВОДА-ВЫВОДА С ОЧЕРЕДЯМИ

Операции ввода-вывода осуществляются с помощью инструкций $QIO_S и $QIOW_S. Эти макроинструкции помещают запросы на выполнение требуемых операций ввода-вывода в очередь, которая используется системой для определения того, какие операции предстоит выполнять. В большинстве случаев операции ввода-вывода выполняются существенно дольше, чем внутренние машинные операции. Например, ЭВМ семейства VAX может выполнить миллионы инструкций за то время, которое требуется пользователю для ввода команды с терминала. Поэтому операционная система ЭВМ семейства VAX, как правило, только инициирует операции ввода-вывода, а затем продолжает выполнять другую обработку до тех пор, пока не будет завершен ввод-вывод.

Действие макроинструкций $QIO_S и $QIOW_S основано на этих же принципах. Подпрограмма, вызываемая при расширении макроинструкции $QIO_S, возвращает управление вызывающей программе сразу же после инициации запроса. После этого программа может свободно выполнять вычисления, не связанные с запрошенными операциями ввода-вывода. Существует ряд способов, позволяющих определить, завершился ли запрошенный ввод-вывод. И программа должна воспользоваться одним из них, чтобы удостовериться, что уже может выполнять операции, связанные с этим вводом или выводом.

Макроинструкция $QIOW_S используется, если в программе ничего не может быть выполнено до тех пор, пока не станут доступными запрашиваемые данные. В таком случае она должна ждать завершения операций ввода или вывода. Обратите, однако, внимание, что в отношении ЭВМ это не есть реальное ожидание. В любой многопользовательской системе, такой как VAX/VMS, пока программа ожидает завершения запроса ввода-вывода, могут обычно выполняться программы других пользователей.

Обе макроинструкции имеют один и тот же набор параметров. Два из них являются обязательными. Это:

1. CHAN – номер канала;
2. FUNC – код функции ввода-вывода.

Номер канала обычно получают из области, которая была загружена с помощью макроинструкции $ASSIGN_S. Функция ввода-вывода задается числовым кодом, который идентифицирует функцию. Независимо от того, вызывается ли макроинструкция $QIO_S или $QIOW_S, для идентификации всех этих функций определяется набор символических имен. Ссылка на функции по их символическим именам вместо указания их фактического числового кода имеет два преимущества. Во-первых, символические имена, используемые для обозначения функции, обычно содержат информацию о сути операции и делают программу самодокументируемой. Во-вторых, при доработке ОС VAX/VMS, возможно, могут применяться другие числовые коды для функций. При использовании символических имен повторное ассемблирование программы автоматически скорректирует коды функций. Некоторые коды функций:

1. IO$_READLBLK Читать виртуальный блок
2. IO$_READLBLK Читать логический блок
3. IO$_READPBLK Читать физический блок
4. IO$_WRITEVBLK Записать виртуальный блок
5. IO$_WRITELBLK Записать логический блок
6. IO$_WRETIPBLK Записать физический блок
7. IO$_SETMODE Установить режим
8. IO$_SETCHAR Установить характеристики

Более подробно коды функций и их возможные модификации будут описаны в этом разделе далее. В зависимости от того, какой код функции используется в макроинструкции, может быть до шести параметров, которые обозначаются именами P1, P2, ... ,P6. Например, при чтении и записи P1 – это адрес массива или буфера, куда будут помещены данные или откуда они будут извлекаться. Параметр P2 задает размер буфера в байтах. На рис. 11.2 показано, как можно вывести сообщение на устройство SYS$OUTPUT. Макровызов $ASSIGN_S – тот же самый, что использовался на рис. 11.1.

Обратите внимание, что перед кодом функции и длиной сообщения стоит знак номера. Это сделано потому, что макрорасширение включает инструкции PUSHL FUNC и PUSHL P2. Это позволяет подставлять на место параметров или число с предшествующим ему знаком номера #, или адрес ячейки, в которой содержится значение, или даже имя некоторого регистра или указать режим адресации, определяющий местоположение значения.

Обратите также внимание на то, что строка сообщения содержит символы перевода строки и возврата каретки. Это необходимо потому, что при такой вызывающей последовательности все заданные в строке сообщения символы выводятся без какого-либо управления форматом. Обычно же операционная система VAX/VMS осуществляет ввод и вывод информации записями, которые соответствуют строчкам распечатки и не требуют управляющих символов. В большинстве случаев предпочтительно форматирование запи-

```
CR=^X0D
LF=^X0A
CH1:            .BLKL       1
CHNAME:         .ASCID      "SYS$OUTPUT"
MESSAGE:        .ASCII      <LF>"ЗДЕСЬ ПОМЕЩЕНО СООБЩЕНИЕ"<CR>
MESSAGE_END:
MESSAGE_SIZE=MESSAGE_END-MESSAGE
                .
                .
                .
                $ASSIGN_S   DEVNAM=CHNAME,CHAN=CH1
                .
                .
                .
                $QIOW_S     CHAN=CH1,FUNC=#IO$_WRITEVBLK,-
                            P1=MESSAGE,P2=MESSAGE_SIZE
                .
                .
                .
```

Рис. 11.2. Вывод сообщения

сей, поскольку оно более эффективно. Однако пока наше обсуждение ограничивается рассмотрением форматирующих символов, чтобы сконцентрировать внимание на особенностях работы терминала. Макроинструкции $QIO имеют параметр P4, который может использоваться для выполнения автоматического форматирования; более детально мы обсудим это позднее.

В рассматриваемой программе для выполнения операции вывода используется также функция IO$_WRITEVBLK. Эта операция осуществляет запись *виртуального блока* информации. Если устройством вывода является диск, расположение виртуальных блоков определяется относительно начала файла, тогда как расположение *логических блоков* определяется относительно самого устройства. *Физические блоки* определяются как физические блоки на магнитной ленте или диске и могут отличаться по размеру от логических блоков. Для совместимости длина логических блоков на устройствах с файловой структурой единообразно устанавливается равной 512 байтам, даже если на физическом носителе применяется другое блокирование информации. Вообще запись физических и логических блоков необходима только для реализации системных функций и безопасность системы может быть нарушена, если обычные пользователи смогут осуществлять доступ к физическим или логическим блокам на системном диске. Вследствие этого функции IO$_READLBLK, IO$_WRITELBLK, IO$_READPBLK и IO$_WRITEPBLK могут быть использованы только привилегированными пользователями.

11.4. ВВОД И ВЫВОД С ТЕРМИНАЛА

ОСОБЕННОСТИ ОРГАНИЗАЦИИ ВВОДА С ОЧЕРЕДЯМИ

Ввод с терминала имеет одну сложность, которая обычно не встречается при выводе. Во время распечатки выводимой информации программа в точности определяет, сколько символов находится в буфере вывода. Это справедливо для вывода одиночной строки, для многострочного вывода или даже если должна быть выведена лишь часть строки. Однако при вводе число символов в строке определяется обычно не в программе, а работающим за терминалом человеком. Поэтому программа должна быть способна определить, сколько символов было напечатано оператором терминала. Существует ряд способов для определения этого, тем не менее их можно свести всего лишь к трем.

Ввод целой строки. Первые два способа основаны на предположении, что одновременно вводится вся строка целиком. Третий способ используется в программах, которые вводят за один раз по одному символу. Этот последний способ ввода применяется тогда, когда каждый вводимый с терминала символ может выполнять функцию управляющего символа, приводящего к каким-либо действиям. Примером такого типа программы может служить экранный редактор, например EDT (см. приложение Д).

Если при вводе происходит считывание одновременно всей строки, то для хранения строки должен быть предусмотрен буфер длиной не меньше самой длинной допустимой при вводе строки. Если терминал пользователя имеет экран шириной 80 символов, то разумно будет ограничить длину вводимых строк длиной строки, отображаемой на экране терминала. Тогда подходящим окажется буфер длиной 81 байт. Заметим, что дополнительный байт необходим для того, чтобы в него помещался символ возврата каретки при 80-символьной строке. В двух первых способах ввода используется такой буфер. Реально же память обычно не так уж ограничена, поэтому, чтобы не рисковать, предпочтительно иметь буфер несколько большего размера. По первому способу сначала считывается строка целиком, а затем посимвольно просматривается содержимое буфера для нахождения признака конца строки. Признаком конца строки служит обычно символ возврата каретки (код ASCII^X0D). Однако при вводе к завершению строки обычно приводит любой управляющий символ, который может быть введен (за исключением символов табуляции,

перевода строки, перевода страницы, вертикальной табуляции и возврата на шаг назад)[1]. Для изменения списка символов, которые интерпретируются как признаки конца строки, может использоваться параметр P4. Может быть выбран любой набор символов алфавита, однако потребность в использовании символов, отличающихся от стандартного набора, возникает редко; для более детального изучения этих функций читатель может обратиться к руководству "VAX/VMS Input/Output User's Guide"

Поскольку существует ряд возможных символов – признаков конца строки, проверка каждого символа на его принадлежность к множеству всех допустимых признаков конца строки потребовала бы очень много времени. И это приводит ко второму способу определения длины вводимой строки. По мере ввода символов операционная система должна контролировать каждый символ, проверяя, не является ли он завершающим, и подсчитывать число занятых байтов в буфере; поэтому существует счетчик вводимых символов, доступный для программиста через квадраслово, называемое *блоком состояния ввода-вывода* (IOSB). Второе слово в этом квадраслове является счетчиком числа переданных байтов без учета завершающего символа. На рис. 11.3 показаны строчки программы, необходимые для ввода строки с терминала и для того, чтобы сделать счетчик байтов строки доступным по адресу BCOUNT. Остальные три слова в блоке IOSB содержат информацию о состоянии, требуемую для более сложной обработки данных. Вновь следует читателю обратиться за более подробной информацией к руководствам фирмы DEC.

Будет ли показанная на рис. 11.3 программа работать по первому или второму способу, зависит от того, что программа делает с информацией, получаемой с помощью операции $QIOW_S. Если она просматривает буфер в поисках признака конца строки, то это будет первый способ; если же используется счетчик BCOUNT для получения числа байтов, то это – второй способ. Поскольку для первого способа не нужны BCOUNT и SBLOK, то определяющие их строчки текста могут быть исключены из программы на рис. 11.3.

Посимвольный ввод. Третий способ ввода заключается в считывании за один раз по одному символу. Наипростейший путь выполнить такое считывание – сделать длину буфера равной одному байту. Тогда, как только человек за терминалом напечатает один символ, буфер заполнится и ввод будет завершен, как если бы был введен завершающий строку символ. После завершения операции, заданной в макровызове, $QIOW_S, программе будет передан этот одиночный символ и программа сможет обработать его соответствующим

```
CH1:        .BLKL     1
CHNAME:     .ASCID    "SYS$INPUT"
BUFFER:     .BLKB     100
SBLOK:      .BLKQ     1
BCOUNT=SBLOK+2              ;ВТОРОЕ СЛОВО IOSB
            .
            .
            .
            $ASSIGN_S   DEVNAM=CHNAME,CHAN=CH1
            .
            .
            .
            $QIOW_S     CHAN=CH1,FUNC=#IO$_READVBLK,-
                        P1=BUFFER,P2=#100,IOSB=SBLOK
            .
            .
            .
```

Рис. 11.3. Пример ввода с терминала

[1] Благодаря своему специальному назначению управляющие символы CTRL/Q, CTRL/S, CTRL/Y и некоторые другие не являются обычными в том смысле, что они не попадают в символьную строку, которая будет считываться в нормальном режиме.

```
CR = ^X0D
BS = ^X08
CTLU = ^X15
DEL = ^X7F
BSB:      .ASCII  <BS>" "<BS>
BUFFER:   .BLKB   100
CHBUF:    .BLKB   1
          .
          .
          .

READ:     MOVL    R5,-(SP)
          CLRL    R5              ;ИНИЦИАЛИЗАЦИЯ УКАЗАТЕЛЯ БУФЕРА
10$:      $QIOW_S CHAN=CH1,FUNC=#IO$_READVBLK!IO$M_NOFILTR,-
                  P1=CHBUF,P2=#1
          MOVB    CHBUF,BUFFER[R5] ;ВЗЯТЬ СИМВОЛ
          INCL    R5              ;И ПЕРЕМЕСТИТЬ УКАЗАТЕЛЬ
          CMPB    CHBUF,#CR       ;ВОЗВРАТ КАРЕТКИ?
          BEQL    20$
          CMPB    CHBUF,#DEL      ;УДАЛЕНИЕ СИМВОЛА?
          BEQL    30$
          CMPB    CHBUF,#CTLU     ;CTRL/U?
          BEQL    40$
          CMPL    R5,#100         ;В БУФЕРЕ ЕСТЬ ЕЩЕ МЕСТО?
          BLSSU   10$             ;ДА
          JMP     ERROR           ;НЕТ
20$:      MOVL    R5,R0           ;ВОЗВРАТИТЬ ДЛИНУ СТРОКИ В R0
          MOVL    (SP)+,R5        ;ВОССТАНОВИТЬ СОДЕРЖИМОЕ R5
          RSB                     ;ВОЗВРАТ

30$:      DECL    R5              ;ПЕРЕМЕСТИТЬ УКАЗАТЕЛЬ НАЗАД
          BEQL    10$             ;ЗАКОНЧИТЬ, ЕСЛИ БУФЕР ПУСТ
          BSBB    60$             ;УДАЛИТЬ СИМВОЛ
          BRB     10$             ;И ЗАКОНЧИТЬ ОБРАБОТКУ УДАЛЕНИЯ
40$:      DECL    R5
50$:      TSTL    R5              ;БУФЕР ПУСТ?
          BEQL    10$             ;ДА, ОБРАБОТКА ЗАВЕРШЕНА
          BSBB    60$             ;НЕТ, УДАЛИТЬ СИМВОЛ
          BRB     50$             ;ПОВТОРИТЬ ЕЩЕ РАЗ
60$:      DECL    R5              ;УДАЛИТЬ СИМВОЛ
          $QIOW_S CHAN=CH2,FUNC=#IO$_WRITEVBLK,-
                  P1=BSB,P2=#3    ;СТЕРЕТЬ СИМВОЛ С ЭКРАНА
          RSB                     ;ВОЗВРАТ
```

Рис. 11.4. Посимвольный ввод данных

образом. Это дает программе возможность обрабатывать каждый раз по одному символу и немедленно ответить, основываясь на посимвольной обработке, как это делает редактор EDT в экранном режиме. Однако в этом режиме возникает проблема – клавиша удаления DEL и комбинация CTRL/U не могут быть использованы как обычно. Чаще всего по мере поступления вводимая информация сохраняется в массиве операционной системы, который называется *буфером клавиатуры*. В какой-то момент символ удаления и CTRL/U предписывают операционной системе удалить символ или строку из буфера. При активизировании операции ввода, информация будет пересылаться из буфера клавиатуры в буфер пользователя всякий раз, когда буфер заполнится полностью или когда встретится завершающий символ. Если буфер пользователя имеет длину 1 байт, то, как только буфер клавиатуры принимает одиночный символ, этот символ сразу же пересылается программе пользователя. Следовательно, в буфере клавиатуры не будет каких-либо данных, которые могли бы удаляться при вводе символа удаления или CTRL/U, поэтому эти управляющие символы никакого действия не оказывают.

В результате в режиме посимвольного ввода обычно необходимо, чтобы в самой программе выполнялось все редактирование входной информации, включая обработку символов удаления и CTRL/U. Здесь кроется одна проблема, так как при нормальной работе символы удаления и CTRL/U программе пользователя не передаются. Следовательно, программа не знает о том, что пользователь нажал клавишу DEL для удаления символа или комбинацию клавиш CTRL/U для удаления целой строки. Это может вводить в заблуждение, поскольку изображение на экране терминала может быть таким, как если бы на самом деле отрабатывались эти управляющие символы. Одно решение этой проблемы состоит в том, чтобы перейти к модифицированному режиму работы, когда символы удаления и CTRL/U не обрабатываются системой, а передаются программе, как обычные символы. Это можно сделать, воспользовавшись кодом модификации IO$M_NOFILTR. Биты этого кода комбинируются с битами кода IO$_READVBLK с помощью логической операции ИЛИ (!), так что функция записывается следующим образом:

```
FUNC=#IO$_READVBLK!IO$M_NOFILTR
```

На рис. 11.4 показан фрагмент программы, в котором выполняется посимвольный ввод, введенные символы помещаются в буфер. Символ возврата каретки является символом завершения строки. Если нажата клавиша удаления, то один символ (если он есть) удаляется из буфера, а для стирания символа с экрана терминала выдается последовательность "возврат на шаг – пробел – возврат на шаг". Если нажата комбинация клавиш CTRL/U, то функция удаления повторяется, пока буфер не будет пуст (соответствующие символы стираются с экрана). В программе подразумевается, что каналы CH1 и CH2 были назначены устройствам SYS$INPUT и SYS$OUTPUT, как в более ранних примерах.

ИНТЕРАКТИВНЫЙ РЕЖИМ РАБОТЫ С ТЕРМИНАЛОМ

Когда программа поддерживает интерактивный режим работы с терминалом, важно, чтобы программа и терминал могли обеспечить некоторый вид диалога, в котором сохранялась бы постоянная синхронизация пользователя и программы. Нередко попытка написать программу, предназначенную для интерактивного режима работы, терпит неудачу. Программист вводит команду, запуская программу на выполнение, но ничего не происходит. Кажется, что ЭВМ ничего не делает. Вскоре программист убеждается, что программа все-таки работает, но ожидает ввода. Решение, конечно, состоит в том, чтобы сопроводить каждую операцию ввода в программе выдачей какого-то наводящего сообщения, которое предупреждало бы пользователя о том, что ожидается ввод с терминала.

Наводящие сообщения могут быть простыми или сложными в зависимости от того, насколько много информации необходимо знать пользователю, чтобы правильно ответить на сообщение. Например, операционная система VAX/VMS перед началом сеанса работы предлагает пользователю сообщения User: иPassword:, на которые ожидаются очень конкретные ответы. После начала сеанса работы обычно используется более простое сообщение $, которое следует понимать как приглашение к вводу любой команды операционной системы. В пользовательских программах применяются свои собственные наводящие сообщения, отличающие одну программу от других. Многие редакторы текста, например TECO и EDIT, для запроса команды используют приглашение "*". Приглашение отладчика имеет вид DBG>.

Для облегчения использования наводящих сообщений при вводе, макроинструкции $QIO поддерживают дополнительную функцию чтения IO$_READPROMPT. Эта функция имеет два дополнительных параметра P5 и P6. Параметр P5 определяет адрес строки сообщения, а параметр P6 задает длину этой строки. Строкой наводящего сообщения может быть любая символьная строка, но, как правило, одна должна начинаться с символов возврата каретки и перевода строки, чтобы гарантировать появление наводящего сообщения на новой строке. На рис. 11.5 показан фрагмент программы, в котором выполняется ввод

```
CR = ^X0D
LF = ^X0A
PROMPT:          .ASCII     <CR><LF>"ВВЕДИТЕ ДАННЫЕ - "
PROMPT_END:
PROMPT_LOC:
                 .ADDRESS   PROMPT
BUFFER:          .BLKB      100
SBLOK:           .BLKQ      1
BCOUNT=SBLOK+2
                 .
                 .
                 .
                 $QIOW_S    CHAN=CH1,FUNC=#IO$_READPROMPT,-
                            P1=BUFFER,P2=#100,IOSB=SBLOK,-
                            P5=PROMPT_LOC,P6=PROMPT_END-PROMPT
                 .
                 .
                 .
```

Рис. 11.5. Ввод с наводящим сообщением

в 100-байтовый буфер после вывода наводящего сообщения "Введите данные –". За исключением вывода наводящего сообщения, функция IO$_READPROMPT работает точно так же, как и функция IO$_READVBLK. В программе на рис. 11.5 предполагается, что, как и в предыдущих примерах, по адресу CH1 будет находиться некоторый номер канала. Отметим также использование параметра IOSB, описанного выше.

Следует заметить, что выдача наводящего сообщения может быть осуществлена и без использования функции IO$_READPROMTP, поскольку операция ввода всегда может сопровождаться в программе обычной операцией вывода, выполняемой функцией IO$_WRITEVBLK. Однако использование функции ввода с наводящим сообщением дает несколько преимуществ. Во-первых, здесь несколько выше эффективность, так как требуется только одно обращение к $QIO. Во-вторых, поскольку наводящее сообщение связывается со вводом, то операционная система знает, что необходимо преодолеть действие управляющего символа CTRL/O и даже повторить сообщение, если пользователь нажал комбинацию клавиш CTRL/U.

11.5. ДОПОЛНИТЕЛЬНЫЕ СООБРАЖЕНИЯ ПО ОРГАНИЗАЦИИ ВВОДА-ВЫВОДА С ОЧЕРЕДЯМИ

ОШИБКИ

До сих пор во всех примерах предполагалось, что все шло прекрасно и не было необходимости беспокоиться об ошибках. Но любому человеку, имеющему опыт работы с ЭВМ, известно, что это плохая посылка. Ошибки могут возникать по самым разным причинам. В действительности даже в случае простых программ крайне редко бывает так, чтобы программа заработала при первом же запуске. Поэтому тот, кто использует системные сервисные средства ввода-вывода, должен быть готов иметь дело с ошибками.

Некоторые типы ошибок, которые могут случиться: в имени устройства может быть опечатка; может использоваться неназначенный номер канала; может осуществляться попытка выполнить функцию, для использования которой пользователь не имеет привилегии; может произойти ошибка ввода-вывода на используемом устройстве. Во многих случаях, когда происходит ошибка в какой-либо системной подпрограмме, подпрограмма выводит сообщение об ошибке и прекращает выполнение программы. Это должно быть знакомо большинству работавших с языками высокого уровня. Однако подпрограммы, вызываемые с помощью системных макроинструкций, не работают таким образом. Здесь

заложена та философия, что опытные программисты, которые используют эти подпрограммы, будут располагать собственными средствами обработки ошибок. Например, программа может искать какое-то доступное устройство, пытаясь назначать различные устройства до тех пор, пока не достигнет успеха. Здесь нужно, чтобы подпрограмма, которая не смогла успешно выполнить запрошенную функцию, прекращала свои действия и возвращала управление к вызвавшей ее программе с индикацией ошибки.

Все такие системные подпрограммы используют для возвращения информации об ошибке регистр R0. Если самый младший бит (бит 0) регистра R0 равен 1, это означает успешное завершение подпрограммы. Если он равен 0, то была обнаружена ошибка. Состояние самого младшего бита регистра R0 легко можно проверить с помощью инструкции BLBC или BLBS. Следовательно, чтобы проверить, успешно ли осуществлено назначение устройства, можно использовать следующую последовательность инструкций:

```
                 $ASSIGN_S  DEVNAM=NAME,CHAN=CH1
                 BLBS       R0,10$    ;ОШИБКА?
                 JMP        ERROR     ;ДА, ЗАВЕРШИТЬ ОБРАБОТКУ
10$:                                  ;НЕТ, ПРОДОЛЖИТЬ
```

Если младший бит регистра R0 не равен 1, то три младших бита показывают степень серьезности ошибки, а остальные биты содержат код, идентифицирующий ошибку. Каждой ошибке соответствует определенное символическое имя. Например, код ошибки, обозначенный как SS$NOPRIV, сигнализирует о том, что пользователь не имеет достаточных привилегий для операции, которую он пытается выполнить. Но в большинстве случаев единственной причиной, по которой необходимо знать фактический код ошибки, является желание распечатывать сообщение об ошибке, информирующее пользователя о причине ошибки. Поскольку существуют сотни различных кодов ошибок, написание программы для распечатки всех возможных сообщений об ошибках потребовало бы значительных усилий. Однако сообщения об ошибках можно запросить у операционной системы, указав аргумент в макровызове $EXIT_S. Если это сделано, операционная система будет проверять значение аргумента и, если младший бит нулевой, распечатывает сообщение об ошибке, поясняющее причину ошибки. Таким образом, вместе с тремя предыдущими строчками программы для обработки ошибок может быть использована следующая:

```
ERROR:   $EXIT_S  R0
```

Если в регистр R0 помещено значение SS$NOPRIV, то на терминале пользователя будет распечатано или занесено в файл регистрации при пакетной обработке сообщение вроде следующего:

```
SYSTEM-F-NOPRIV, No privilege for attempted operation, reason . . .
```

ЛОГИЧЕСКИЕ ИМЕНА УСТРОЙСТВ

При написании программы не очень хорошо помещать конкретное имя какого-то физического устройства в код программы по той причине, что в зависимости от разных пользователей фактически используемое устройство может меняться. Например, с помощью команды SHOW TERMINAL пользователь может определить, что конкретным физическим портом связи, через который он начал сеанс работы, является TTC3. Это означает, что при назначении канала для ввода-вывода с терминала должно использоваться имя устройства TTC3.

Однако было бы неразумно писать программу, в которой имя устройства было бы задано следующим образом:

```
.ASCID          "TTC3"
```

Проблема заключается в том, что после окончания сеанса работы сегодня, при новом сеансе работы завтра (сегодня же, но позднее) порт, назначенный вашему терминалу, может быть уже не TTC3, а TTD4, поскольку порт TTC3, вероятно, был уже выделен какому-то другому пользователю. И эта программа не сможет больше работать и вызовет ошибку нарушения привилегий.

Одно возможное решение состоит в том, чтобы вместо включения имени устройства в программу в виде константы пользователь вводил имя устройства как входные данные. С таким решением сопряжены две проблемы. Первая – это неудобство, что пользователи вынуждены вводить эти данные, так как пользователь может и не знать, с какого именно устройства он начал сеанс работы, и ему придется воспользоваться командой SHOW, чтобы узнать это. Вторая же проблема представляет логическую дилемму. Как может программа взаимодействовать с терминалом, чтобы определить имя устройства, если она не знает имени терминала? Решение этой проблемы состоит в использовании логических имен вместо имен физических устройств.

Логическое имя устройства – это символьная строка, которая назначается вместо имени физического устройства. Такие имена могут изменяться пользователем и назначаться устройством с помощью команды ASSIGN операционной системы VAX/VMS. Например, если пользователь хочет назначить для терминала имя SCOPE, может быть введена следующая команда:

```
ASSIGN TTC3: SCOPE
```

Это команда операционной системы, которая вводится с терминала в ответ на приглашение $.

Если такое назначение было сделано, программа может воспользоваться именем SCOPE для назначения канала для устройства TTC3. Программа может быть написана так:

```
NAME:       .ASCID      "SCOPE"
CH:         .BLKL       1
            .
            .
            .

            $ASSIGN_S       DEVNAM=NAME,CHAN=CH
```

Таким образом решается вторая проблема, связанная с невозможностью в программе назначить канал, не зная физического имени терминала; но все же необходимо знать это имя, иначе пользователь не сможет ввести команду назначения. Одно решение состоит в автоматическом назначении нескольких логических имен устройствам всякий раз, когда пользователь начинает сеанс работы. В одном примере были использованы логические имена SYS$INPUT и SYS$OUTPUT. Данные имена автоматически назначаются соответствующим устройствам ввода и вывода для выполнения обычного ввода и вывода. Для пользователей, работающих в интерактивном режиме, эти имена назначаются для терминала пользователя. В режиме пакетной обработки они назначаются для входного и выходного потоков пакета.

Поэтому улучшенная реализация вышеприведенного примера,может быть такой:

```
NAME:       .ASCID      "SYS$INPUT"
CH:         .BLKL       1
            .
            .
            .

            $ASSIGN_S       DEVNAM=NAME,CHAN=CH
```

Канал CH будет автоматически назначен для терминала пользователя.

Одной из более трудных задач программирования является подготовка выходной информации в подходящем формате, облегчающем чтение. Сюда входят формирование сообщений и преобразование чисел из внутренней формы представления в какую-либо другую подходящую форму представления – обычно десятичную, иногда – шестнадцатеричную. Большинство языков высокого уровня включают некоторые средства управления форматом вывода. В Фортране для этих целей используется оператор FORMAT, в Коболе – оператор PICTURE, а в Паскале – информация о формате содержится в списках вывода.

В языке ассемблера форматирование ввода и вывода обычно требует, чтобы программист писал пространные программы символьной обработки. Чтобы упростить эту задачу, в операционную систему VAX/VMS включены некоторые подпрограммы для форматирования выходной информации, которые могут вызываться с помощью системной макроинструкции $FAO_S. Действие этой макроинструкции во многом такое же, как и оператора Фортрана FORMAT; здесь также имеется управляющая строка, которая описывает формат вывода. Однако при этом используется совершенно другой набор кодов. В оставшейся части этого раздела описаны некоторые наиболее полезные управляющие коды для макроинструкции $FAO_S. Полное описание читатель может найти в руководстве "VAX/VMS System Services Reference Manual".

Макроинструкция $FAO_S может быть вызвана несколькими способами. Простейший способ включает три обязательных параметра и множество необязательных параметров, обязательными параметрами являются: CTRSTR, OUTBUF и OUTLEN . Параметр CTRSTR – это адрес дескриптора символьной строки, содержащей управляющую строку. Вероятно, что эта строка определяется директивой .ASCID. Параметр OUTBUF – это адрес дескриптора байтового массива, куда помещается преобразованная выходная информация. При ассемблировании дескриптора образуется два длинных слова. Одно содержит длину байтового массива, а другое – адрес байтового массива, в который будет помещаться выходная информация. Параметр OUTLEN – это адрес слова, в которое будет помещена длина преобразованной символьной строки. Обратите внимание, что поле длины дескриптора OUTBUF определяет максимальный размер выводимой строки. Реальная строка, сформированная с помощью макроинструкции $FAO_S, может быть меньше. Значение фактической длины строки будет находиться в слове OUTLEN. Заметим, что если подпрограммы, вызываемые при расширении макроинструкции $FAO_S, завершаются успешно, то это значение всегда будет меньше или равно длине, заданной в дескрипторе буфера. При попытке сформировать строку более длинную, чем вмещает буфер, будет возвращена усеченная строка вместе с соответствующим значением ошибки, помещенным в регистр R0.

Управляющая строка содержит последовательность символьных строк и спецификаторов формата, определяющих формат вывода. Любая строка, которая не распознается как директива управления форматом, обрабатывается, как символьная строка, которая должна быть выведена. Такие строки не заключаются, как в операторе Фортрана FORMAT, в кавычки. Все директивы управления форматом начинаются с восклицательного знака. Поэтому любая символьная строка, которая не начинается с восклицательного знака, помещается в буфер вывода без какой-либо обработки.

После восклицательного знака следуют значение длины поля вывода, которое может быть опущено, и код спецификатора формата, состоящий из одного или двух символов. Например, код SL задает преобразование содержимого длинного слова в десятичное число со знаком. Следовательно, следующей управляющей строкой можно воспользоваться для распечатки ответа:

```
.ASCID  "ПОЛУЧЕН ОТВЕТ !10SL(ДЕСЯТИЧНОЕ)."
```

Это эквивалентно такому оператору формата Фортрана:

```
100   FORMAT('ПОЛУЧЕН ОТВЕТ ',I10,'(ДЕСЯТИЧНОЕ).')
```

На рис. 11.6 представлен неполный список управляющих кодов, используемых с макроинструкцией $FAO_S, который включает наиболее важные коды управления форматом. Полную информацию об использовании макроинструкции $FAO_S читатель может найти в руководстве "VAX/VMS System Services Reference Manual".

Каждая директива управления форматом может включать длину поля вывода, помещенную между восклицательным знаком и двухсимвольным кодом спецификатора формата. Длина поля вывода – это десятичное число, определяющее число символов, которые могут в нем разместиться. Если указана, например, длина !6SL, то преобразованное значение будет расширено или усечено так, чтобы оно занимало точно 6 позиций. Если длина поля не указана, то она будет равна минимальной длине поля, необходимого для представления числа. Заметим, однако, что если длина буфера недостаточно велика, то произойдет усечение строки.

Следует также отметить, что каждому спецификатору формата должно быть поставлено в соответствие подлежащее преобразованию значение, как, например, в Фортране, где каждой спецификации формата I6 должна соответствовать переменная. Это же делается и при работе с макроинструкцией $FAO_S с помощью параметров P1, P2, P3, . . . Может быть задано до двенадцати параметров. Каждому параметру соответствует адрес ячейки, содержащей подлежащее преобразованию значение. Следовательно, если ячейки A, B и C содержат

Код	Функция
!XB	Преобразовать байт в шестнадцатеричное
!XW	Преобразовать слово в шестнадцатеричное
!XL	Преобразовать длинное слово в шестнадцатеричное
!SB	Преобразовать байт в десятичное со знаком
!SW	Преобразовать слово в десятичное со знаком
!SL	Преобразовать длинное слово в десятичное со знаком
!%T	Занести в буфер значение времени
!%D	Занести в буфер дату и время

Рис. 11.6. Коды управления форматом для макроинструкции $FAO_S

```
CR=^X0D
LF=^X0A
CSTR:   .ASCID   <LF>"БАЙТ !SB СЛОВО !SW ДЛИННОЕ СЛОВО !SL "<CR>
BDES:   .LONG    80,BUF
BUF:    .BLKB    80
BSIZ:   .BLKL    1
A:      .BLKB    1
B:      .BLKW    1
C:      .BLKL    1
        .
        .
        .
        $FAO_S   CTRSTR=CSTR,OUTBUF=BDES,OUTLEN=BSIZ,-
                 P1=A,P2=B,P3=C
```

Рис. 11.7. Пример форматного вывода с помощью макроинструкции $FAO_S

числа в формате байта, слова и длинного слова, то для их преобразования в десятичное представление может использоваться фрагмент программы, показанный на рис. 11.7.

Очевидно, что не только для таких спецификаторов формата, как !SW, требуется наличие соответствующих им параметров, но и для спецификаторов, задающих преобразование значений даты и времени суток. Это связано с тем, что эти спецификаторы могут применяться не только для преобразования текущей даты и времени, но также и для преобразования предварительно сохраненных значений даты и времени, которые могут потребоваться для идентификации версий файлов или систем. Простейшее применение таких преобразований – распечатка текущей даты и времени. В подобных случаях параметр должен заменяться значением, равным #0. Поэтому для распечатки текущей даты и времени может применяться следующая последовательность операторов:

```
TSTR:   .ASCID   <LF>"ВРЕМЯ=!%T."<CR>
        .
        .
        .
        $FAO_S   CTRSTR=TSTR,OUTBUF=BDES,OUTLEN=BSIZ,-
                 P1=#0
        $QIO_S   CHAN=CH1,FUNC=#IO$_WRITEVBLK,-
                 P1=BUF,P2=BSIZ
```

Предполагается, что на этот фрагмент программы распространяется действие определений имен, приведенных на рис. 11.3 и 11.7.

УПРАЖНЕНИЯ 11.1

1. Напишите программу, используя макроинструкцию $QIOW_S для считывания строки, вводимой с клавиатуры терминала, и для последующей распечатки этой строки на терминале 5 раз.

2. Напишите программу, которая бы считывала символьную строку длиной до 30 символов, а затем пыталась открыть канал ввода-вывода, используя эту строку как имя устройства. Используйте подпрограмму PNUM для печати назначенного номера канала, если удалось сделать назначение, или в противном случае – кода ошибки, помещаемого в регистр R0.

3. Используйте команду ASSIGN операционной системы VAX/VMS, чтобы назначить логическое имя XXX для вашего текущего терминала. Например, если ваш терминал имеет имя TTA3:, введите команду

```
ASSIGN   TTA3:   XXX
```

Затем назначьте имя YYY для имени XXX и имя ZZZ для имени YYY с помощью команд

```
ASSIGN   XXX   YYY
ASSIGN   YYY   ZZZ
```

Используйте программу, которую вы написали для п. 2 упр. 11.1, и попытайтесь назначить номера канала для вашего терминала и для устройств с логическими именами XXX,YYY и ZZZ. Все ли назначения выполняются? Если нет, то сколько уровней логических имен допустимо?

4. Перепишите подпрограммы IOINIT, RNUM и PNUM, описанные в приложении Б, так, чтобы использовалась макроинструкция $QIOW_S и связанные с ней подпрограммы вместо $PUT и $GET, как это сделано в указанных подпрограммах.

5. Повторите п. 4, но модифицируйте алгоритмы так, чтобы отсутствовал массив для хранения вводимой строки, а ввод осуществлялся по одному символу. Это потребует также, чтобы программа включала обработку символов DEL и CTRL/U.

ВНЕШНИЕ ЗАПОМИНАЮЩИЕ УСТРОЙСТВА БОЛЬШОЙ ЕМКОСТИ

Чтобы ЭВМ была практически полезной, она должна обладать способностью хранить большие объемы информации на каких-либо *внешних запоминающих устройствах большой емкости.* В больших вычислительных системах применяются разнообразные устройства подобного рода. Наиболее распространенными внешними запоминающими устройствами большой емкости, применяемыми для ЭВМ семейства VAX, являются магнитные диски и магнитные ленты. Информация на таких устройствах записывается в виде последовательно организованной совокупности блоков. Привилегированные пользователи могут осуществлять доступ к дискам и лентам по назначенным для диска или ленты каналам поблочно, используя операции ввода-вывода с очередями (макроинструкция $QIO). Но непривилегированные пользователи для доступа к информации, расположенной на этих устройствах, будут пользоваться другими методами. В оставшейся части главы будут описаны эти последние методы ввода-вывода.

Хотя и диски, и ленты имеют последовательную блочно-организованную структуру данных, физические характеристики этих двух типов устройств совершенно различны, что обусловливает радикально разные способы их использования. Ленты представляют собой длинные узкие полосы магнитного материала, на который производится запись, намотанного на катушки. Длина 2400 футов при ширине ленты половина дюйма – это стандартные параметры для промышленно совместимых полномерных катушек. Блоки информации размещаются один за другим по длине ленты. На рис. 11.8 показано, как располагается информация на ленте.

Чтение или запись на ленту производятся при ее перемещении мимо закрепленной неподвижно *магнитной головки,* подобно тому как это делается в бытовых магнитофонах. Чтобы прочитать информацию, расположенную в конце ленты, приходится перематывать вперед 2400 футов ленты. Если затем потребуется информация, расположенная в начале ленты, то снова придется перемотать назад те же самые 2400 футов. Понятно, что это занимает значительное время (возможно, несколько минут). По этой причине доступ к данным на магнитной ленте обычно осуществляется не в произвольном порядке, а во вполне определенной последовательности.

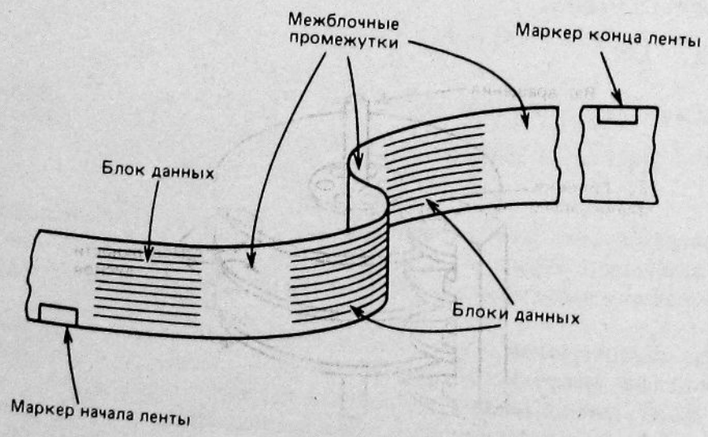


Рис. 11.8. Структура магнитной ленты

Диски собираются из одной или нескольких круглых пластин, поверхности которых покрыты магнитным материалом, нанесенным на пластины в виде кольца. Эти кольца образуют поверхности записи диска. Информация записывается на концентрических дорожках, на которые разбивается кольцо магнитного материала на каждой поверхности. На каждой поверхности может быть более чем 1000 дорожек, а на каждой дорожке записываются несколько блоков фиксированной длины, называемых *секторами.* На магнитных лентах, отвечающих промышленному стандарту, расположение блоков не фиксировано, при записи они занимают изменяющееся по объему пространство. Положение секторов дисков на дорожках фиксировано. Это достигается с помощью физических маркеров, таких как прорези или отверстия на диске, или с помощью меток, записываемых на самих дорожках. Первый метод называют жесткой разметкой секторов, второй – программной разметкой секторов. Прежде чем диск с программной разметкой секторов может быть использован, он должен быть *сформатирован* посредством записи на каждой дорожке маркеров, отмечающих местоположение секторов. Типичное устройство пакетного диска, состоящего из нескольких пластин, показано на рис. 11.9.

Благодаря рассмотренной выше организации каждый блок информации на диске может быть найден при указании специального адреса, имеющего следующие части: 1) номер пластины; 2) поверхность пластины (верхняя или нижняя); 3) номер дорожки на поверхности; 4) номер сектора на дорожке. Считывание информации с дисков осуществляется обычно системой головок чтения-записи. Как правило, имеется по одной головке на поверхность, хотя в некоторых дисковых системах их может быть и больше. Эти головки перемещаются вперед и назад и устанавливаются (позиционируются) над дорожкой, с которой должно быть произведено считывание. Сам диск постоянно вращается, поэтому для доступа к какому-либо сектору приходится ждать, пока нужный сектор не пройдет под головкой чтения-записи.

Магнитные головки перемещаются очень быстро, так что только доли секунды занимает перемещение с самых внутренних на самые внешние дорожки. Кроме того, и диски вращаются весьма быстро, совершая 5–100 полных оборотов в секунду. В результате этого дисковая система может осуществлять доступ к любому блоку информации на диске за доли секунды. Сравните это с магнитной лентой, где могут потребоваться минуты для перехода с одного конца ленты на другой. Поэтому магнитные ленты имеет смысл использовать только для *последовательного доступа,* продвигаясь от одного конца ленты к другому. Для дисков может использоваться *произвольный доступ,* т.е. доступ к блокам осуществляется прямо, независимо от их положения на диске.

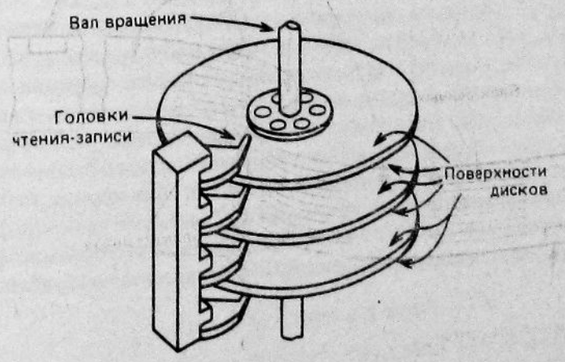


Рис. 11.9. Устройство магнитного диска

Как еще более важный момент, касающийся специфики организации информации на лентах и дисках, рассмотрим запись информации. Как уже упоминалось, на дисках имеются специальные маркеры, которые точно определяют местоположение блоков информации на диске. Обычные магнитные ленты таких меток не имеют. Следовательно, информацию на ленту можно записывать только последовательно. Более того, как правило, невозможно перемотать назад и перезаписать блок информации в середине ленты без того, чтобы не испортить всю следующую далее информацию. С другой стороны, любой сектор диска может быть перезаписан, не затрагивая информацию в любых других секторах.

ФАЙЛЫ

Обычно объем информации, который способно вместить внешнее запоминающее устройство большой емкости, намного превосходит потребности отдельного приложения. Поэтому общепринято разделять информацию на таких устройствах на совокупности данных, называемых *файлами*[1].

Благодаря последовательной организации информации на магнитной ленте файлы на ленте состоят из совокупности смежных блоков. Начало и конец файла помечаются специальными блоками меток, называемыми просто метками. Помимо прочей информации метки содержат имена файлов, используемые программистами для идентификации файлов. По отмеченным в предыдущем разделе причинам последовательная природа операций ввода-вывода, выполняемых с магнитной лентой, позволяет добавлять на ленту метки и файлы, но как только файл записан на ленту, он уже не может быть ни модифицирован, ни удален.

Диски же позволяют выполнять операции ввода и вывода с произвольной выборкой информации. Поэтому файл на диске может состоять из произвольно расположенных блоков информации, разбросанных по разным секторам диска. При записи файла система ввода-вывода должна для блоков файла отыскать неиспользованные секторы. Дисковые файлы могут модифицироваться или удаляться. Если файл удаляется с диска, его блоки становятся доступными, как неиспользованные секторы диска. Чтобы отслеживать все эти операции, требуется система *каталогов*. *Каталог* – это некоторая область диска, которая содержит информацию о расположении файлов на диске, расположении блоков файлов и т. п. Кроме того, диски большой емкости используются для обслуживания большого числа пользователей. Поскольку каждый пользователь может иметь свой собственный каталог файлов, могут даже существовать каталоги каталогов.

Сказанное оставляет впечатление нечто сложного, но это сделано преднамеренно. Написание программ для обслуживания файлов на диске – это не только крайне сложное дело, но и опасное. Диск, содержащий 100 тыс. блоков информации, может быть приведен в полную непригодность при потере одного блока информации в каталоге. Поскольку информация рассеяна по диску, каталог следует вести очень тщательно, обеспечивая его максимальную защиту, чтобы можно было отыскать какую-либо информацию. Как следствие этого, пользователям системы редко приходится программировать операции такого рода. В действительности существуют механизмы защиты, препятствующие любым программам, кроме самых привилегированных, выполнять любые операции, которые могли бы испортить справочную информацию на диске. По этой причине доступ пользователей к дисковым файлам обычно осуществляется с помощью системных программ, работающих на более высоком уровне, чем операции подсистемы $QIO, описанные в начале этой главы. В частности, в операционной системе VAX/VMS файловый ввод-вывод выполняется с помощью *службы управления записями* (подсистема RMS), описание которой дается в разд. 11.7.

[1] В некоторых приложениях нужно так много информации, что она не помещается на отдельном диске или ленте. Тогда информация может быть распространена на несколько таких устройств. При этом говорят о *многотомных файлах*.

ЗАПИСИ

Так же как диски и ленты подразделяются на файлы, обычно необходимо файлы разделять на более мелкие порции информации, которые называют *записями*. Исторически сложилось так, что записи служили синонимами блоков, так как блок – это наименьшая единица информации, которая может быть считана за одну физическую операцию устройства. Однако быстро пришло понимание того, что единицы информации, с которыми удобно иметь дело в программных приложениях, не совпадают с размерами физических блоков. В результате этого или очень неэффективно работали программы, так как небольшие порции информации запоминались в блоках, способных вместить больший объем информации, или программист вынужден был изменять структуру программ, чтобы достичь эффективного использования внешних запоминающих устройств большой емкости. В большинстве случаев программист хотел бы переложить эту работу на кого-нибудь другого. Поэтому во многих развитых системах ввода-вывода файлы разбивают на записи, которые могут быть приспособлены под потребности конкретного приложения. Затем эти записи упаковывают так, чтобы они эффективно укладывались в физическую блочную структуру конкретного устройства ввода-вывода.

Следовательно, *запись* представляет собой такую порцию информации, содержащейся в файле, которую удобно обработать за один прием. Наиболее общим типом записи является вводимая или выводимая строчка текста. Чаще всего записи считываются и записываются одна за другой. Это называется *последовательным доступом*. Однако для некоторых приложений требуется существование возможности считывать записи из любого места файла. Это называется *произвольным* или *прямым доступом*. Большинство операционных систем обеспечивает прямой доступ к записям в файлах, хранимым на устройствах прямого доступа.

Для того чтобы использовать стандартное системное программное обеспечение для выполнения операций чтения и записи, программист должен определить структуру файла. Некоторые характеристики файла, которые должны быть определены:

1) тип записи: записи фиксированной или переменной длины;
2) длина записи (для записей фиксированной длины);
3) тип доступа: последовательный или прямой доступ;
4) тип прямого доступа (для файлов прямого доступа);
5) другие характеристики, такие как длина файла, содержит ли файл данные в коде ASCII или двоичные и т. п.

В операционной системе VAX/VMS эти операции выполняются подсистемой RMS, описание которой дается в следующем разделе.

11.7. ПОДСИСТЕМА УПРАВЛЕНИЯ ЗАПИСЯМИ (RMS)

Служба управления записями (**RMS**) операционной системы VAX/VMS предназначена для выполнения операции ввода-вывода при работе с файлами. Это совокупность системных подпрограмм, которые необходимы большинству программистов для работы с файлами. Как и в случае ввода-вывода с очередями, операции подсистемы RMS выполняются с помощью ряда системных подпрограмм, обращение к которым осуществляется с помощью макроинструкций. По существу, имеется два типа макроинструкций, применяемых для RMS. Макроинструкции одного типа порождают блоки данных, которые содержат всю необходимую информацию о файле или о конкретных записях файла. Макроинструкции другого типа порождают исполняемый программный код, предназначенный для вызова подпрограммы, необходимой для выполнения операции подсистемы RMS.

БЛОКИ ДОСТУПА К ФАЙЛУ

Блоки доступа к файлу – это блоки данных, содержащие всю информацию о некотором файле, которую должен обеспечить пользователь для выполнения операции ввода-вывода при работе с файлом. Среди прочего блок содержит имя файла и информацию об организа-

ции файла, методах доступа и типах записей. Блок доступа к файлу формируется с помощью макроинструкции **$FAB**. Если файл относится к наиболее распространенному типу файлов последовательного доступа с записями переменной длины, то большая часть блока будет определена по умолчанию, так что необходимо указать только имя файла.

Имена файлов представляют собой символьные строки, которые однозначно идентифицируют файлы. Полное имя файла имеет следующие части:

1. Имя узла – определяет конкретную ЭВМ VAX в сети вычислительных маших семейства VAX.
2. Имя устройства – определяет, какой используется диск, какая лента и т. п.
3. Имя каталога – определяет используемый каталог или подкаталог, поскольку на диске может быть много каталогов или подкаталогов.
4. Имя файла – определяет конкретный файл в совокупности с типом файла (см. п. 5) и номером версии (см. п. 6).
5. Тип файла – определяет предназначение файла, сообщая, является ли файл программой на макроассемблере, программой на Фортране, объектным файлом или чем-то другим.
6. Номер версии – определяет версию файла, изменяющуюся при модификации или перезаписи файла.

Эти части имени файла отделяются друг от друга различными знаками пунктуации, такими как : , [ ] и ; . Поэтому полное имя может выглядеть так:

```
TEMPVAX::DPA2:[KANDS.PROGS]EXAMPLE.MAR;4
```

Части этого описания:

1) имя узла – TEMPVAX;
2) имя устройства – DPA2;
3) подкаталог – PROGS в каталоге KANDS;
4) имя файла – EXAMPLE;
5) тип файла – MAR;
6) номер версии – 4.

Поскольку полное имя файла может быть очень длинным, большинство частей, если они не указаны, подразумеваются по умолчанию. Обычная спецификация файла, используемая по умолчанию, такова:

1. Имя узла – узел сети, в котором выполняется программа.
2. Имя устройства – диск, назначаемый по умолчанию для пользователя в начале сеанса работы.
3. Каталог или подкаталог – каталог, устанавливаемый по умолчанию для пользователя в начале сеанса работы.
4. Имя файла – нет значения по умолчанию.
5. Тип файла – нет значения по умолчанию.
6. Номер версии – самый старший для этого файла.

Большинство этих умолчаний управляются посредством логических имен, автоматически назначаемых для пользователя в начале сеанса работы. Эти умолчания могут быть изменены с помощью команды DEFINE. Кроме того, специальные значения по умолчанию могут быть заданы в блоке доступа к файлу. В результате, если местонахождение вышеупомянутого файла удовлетворяет всем условиям умолчания, то он может быть назван просто как

```
EXAMPLE.MAR
```

Аналогично можно сформировать блок доступа к файлу, который позволит осуществлять чтение и запись файла, используя последовательный доступ, с помощью следующего макровызова:


```
F1:     $FAB    FNM=<EXAMPLE.MAR>
```

Обратите внивание! Угловые скобки являются необходимой частью синтаксиса!

Этот макровызов генерирует две структуры. Первая – это сам блок доступа к файлу. Однако в действительности блок доступа к файлу не содержит имени файла, поскольку этот блок имеет фиксированную длину и может содержать только данные фиксированной длины. Поэтому в блоке доступа к файлу содержатся адрес и длина строки имени файла, помещенные соответственно в длинном слове и слове. Реальное имя файла хранится, как строка в коде ASCII, в отдельной программной секции с именем **$RMSNAM**. Заметим, что подсистема RMS требует, чтобы блоки доступа к файлу и блоки доступа к записи (которые будут обсуждаться в следующем разделе) были выровнены по границе длинного слова. То есть адрес начала блока должет быть кратен четырем. Чтобы гарантировать, что блоки доступа к файлу и блоки доступа к записи выровнены по границе длинного слова, лучше всего организовать их как отдельные программные секции. Например, можно использовать следующую последовательность операторов:

```
        .PSECT  IOBLOCKS,LONG
F2:     $FAB    FNM=<FILE1.TYP>
F3:     $FAB    FNM=<FILE2.TYP>
        .
        .
        .
        .PSECT
```

Заметим, что программа может иметь любое число блоков доступа к файлам. Однако в любой конкретный момент времени для каждого файла должен использоваться только один блок.

Заметим также, что имена файлов, применяемые с блоком доступа к файлу, могут содержать как логические, так и реальные имена для частей или для всей спецификации файла. Например, вызов макроинструкции $FAB может быть сделан по-разному:

```
F4:     $FAB    FNM=<SYS$INPUT>
F5:     $FAB    FNM=<USER:FILE.TYP>
```

Использование логических имен SYS$INPUT и SYS$OUTPUT позволяет направлять ввод-вывод на терминал пользователя. Если программа выполняется в пакетном режиме, то эти логические имена относятся к входному и выходному потокам пакета.

БЛОКИ ДОСТУПА К ЗАПИСИ

Блоки доступа к записи используются для того, чтобы обеспечить всю информацию, необходимую для чтения или записи какой-либо отдельной записи файла. С блоком доступа к записи связана следующая информация:

1. Ссылка на блок доступа к файлу для файла, с которым выполняются операции ввода-вывода.
2. Местоположение в памяти буфера записи.
3. Длина записи.
4. Длина буфера.
5. Информация, необходимая для обнаружения записи, для файла прямого доступа.
6. Наводящие сообщения для терминального ввода.
7. Информация для синхронизации, позволяющая организовать одновременную работу с несколькими файлами.

Часть этой информации должна передаваться подсистеме RMS программой пользователя. Другая часть информации поступает в программу пользователя от подсистемы RMS.

Какая и куда информация передается, зависит от того, выполняется ввод или вывод. Например, при записи записей переменной длины фактическая длина записи определяется программой пользователя. Однако при чтении записей переменной длины длина определяется тем, что реально имеется в файле.

Как и в случае блоков доступа к файлу, когда осуществляются чтение или запись последовательных файлов, бóльшая часть информации в блоке доступа к записи может быть задана по умолчанию. Для последовательного файла с записями переменной длины необходимы и обычно достаточны следующие три элемента:

1) файл; 2) местоположение буфера; 3) длина буфера в байтах. Используя эту информацию, можно применить макроинструкцию $RAB для формирования блока доступа к записи. Чтобы вывести запись, можно воспользоваться следующим макровызовом:

```
R1:     $RAB     FAB=OUTFILE,RBF=OUTBUF,RSZ=OBS
```

Указанные аргументы имеют следующий смысл:

Аргумент	Значение
OUTFILE	Метка блока доступа к файлу, в который производится запись
OUTBUF	Метка строчки программы с директивой .BLKB или директивой .ASCII, определяющая место, где хранятся данные, подлежащие записи
OBS	Символическое имя, или число, или выражение, определяющее длину записываемой строки (записи)

Для того чтобы считать запись, необходим макровызов следующего типа:

```
R1:     $RAB     FAB=INFILE,UBF=INBUF,USZ=IBS
```

В этом случае аргументы имеют следующий смысл:

Аргумент	Значение
INFILE	Метка блока доступа к файлу, из которого производится чтение
INBUF	Метка блока памяти, куда помещается считываемая запись
IBS	Длина буфера ввода

Обратите внимание, что IBS задает не длину записи, которая может быть больше или меньше, а максимальный объем памяти, доступной для размещения записи. Поэтому, если делается попытка чтения записи большей длины, информация будет теряться. Если такое произошло, в регистр R0 будет возвращен код предупреждения, которому соответствует символическое имя RMS$_RTB. (см. в предыдущих разделах о возврате кода ошибки и в последующем разделе о работе подсистемы RMS).

```
        .PSECT  IOBLOK,LONG
F1:     $FAB    FNM=<SYS$INPUT>
F2:     $FAB    FNM=<SYS$OUTPUT>
R1:     $RAB    FAB=F1,UBF=IBUF,USZ=80
R2A:    $RAB    FAB=F2,RBSF=MSGA,RSZ=MSGAE-MSGA
R2B:    $RAB    FAB=F2,RBF=MSGB,RSZ=MSGBE-MSGB
R2C:    $RAB    FAB=F2,RBF=OBUF
        .PSECT
MSGA:   .ASCII  "СООБЩЕНИЕ А"
MSGAE:
MSGB:   .ASCII  "СООБЩЕНИЕ В"
MSGBE:
IBUF:   .BLKB   80
OBUF:   .BLKB   132
```

Рис. 11.10. Типовое определение блоков доступа к файлу и записям

Заметим, что для вывода используются параметры RBF и RSZ, а для ввода – параметры UBF и USZ. Почему это так, станет ясно позднее.

Как и блоки доступа к файлу, блоки доступа к записи должны быть выровнены по границе длинного слова. Легче всего это сделать, если поместить их все в одну программную секцию, выровненную по границе длинного слова. На рис. 11.10 показан фрагмент программы, в котором определяются блоки доступа к файлу и записи для вывода на терминал.

Обратите внимание, что блок доступа к записи с меткой R2C: не содержит параметра RSZ. Это сделано потому, что, как случается в большинстве программ, длина выводимой строки не известна до тех пор, пока выполнение программы не дойдет до того места, когда должна быть осуществлена операция записи. Имеются средства для получения отсутствующей информации во время выполнения программы. Это относится как к блокам доступа к файлу, так и к блокам доступа к записи и позволяет учесть изменения любой информации, включая имена файлов, адреса буферов, длину записей и т. п.

11.8. ИСПОЛЬЗОВАНИЕ ПОДСИСТЕМЫ RMS

ОПЕРАЦИИ ПОДСИСТЕМЫ RMS

Предназначение блоков доступа к файлу и записи состоит в том, чтобы обеспечить информацию, необходимую для выполнения операций ввода-вывода при работе с файлами. Основными операциями ввода-вывода при работе с файлами являются чтение и запись. Однако для того, чтобы использовать данные операции, необходимо выполнить несколько взаимосвязанных операций для инициализации и завершения этих операций. В простейших случаях ввод-вывод, осуществляемый с помощью подсистемы RMS, может быть выполнен, если использовать шесть макроинструкций, которые порождают вызовы соответствующих подпрограмм. Эти макроинструкции предназначены для следующего:

1. $OPEN – Выполняет поиск по каталогу и другие операции, необходимые для того, чтобы дать возможность подсистеме RMS прочитать записи из уже существующего файла[1].
2. $CREATE – Подготавливает пространство на диске и в каталоге для нового файла, который будет записан на диск[1]. Используется вместо $OPEN для создания новых выходных файлов.
3. $CONNECT – Подключает файл к потоку записей, что завершает процесс открытия файла для операций над записями и связывает блок доступа к записи с блоком доступа к файлу и с потоком записей. Эта операция должна выполняться для каждого файла после того, как он был открыт или создан, но перед тем, как могут быть выполнены любые другие операции, использующие блок доступа к записи. Если для файла используется более одного блока доступа к записи, то единственным активным блоком будет подключенный самым последним. Поэтому, прежде чем новый блок доступа к записи может быть использован, он должен быть подключен. Предыдущий блок будет отключен и не может быть повторно использован до тех пор, пока не будет подключен повторно. Макроинструкция $CONNECT заставляет систему установить внутренние буферы, соответствующие структурам физических блоков и логических записей файла.
4. $GET – Осуществляет считывание записи в буфер ввода.
5. $PUT – Осуществляет вывод записи из буфера вывода.

[1] Аналогичные операции имеют место, если файл располагается на магнитной ленте, терминала или осуществляется ввод-вывод с некоторых других устройств.

6. $CLOSE — Завершает работу с файлом и позволяет использовать блок доступа к файлу для ввода и вывода из другого файла. Макроинструкция $EXIT_S обеспечивает автоматическое закрытие всех открытых файлов. Однако лучшая программистская практика рекомендует закрывать открытые файлы явным образом. Хотя операционная система VAX/VMS очень хороша для пользователя в смысле закрытия файлов, иногда информация, записанная в выходной файл, может быть потеряна, если программа прекращает работу прежде, чем файл будет закрыт. Именно закрытие файла завершает формирование записи о файле в каталоге.

В простейших случаях необходимо, чтобы в макровызовах указывались только блок доступа к файлу и блок доступа к записи, поскольку в блоках доступа содержится вся необходимая информация. Так как в макроинструкциях $OPEN, $CREATE и $CLOSE выполняются операции с файлами, в их макровызовах должен быть указан блок доступа к файлу. Аналогично в макровызовах $CONNECT, $GET и $PUT указывается блок доступа к записи.

Если предположить, что блоки доступа к файлу и блоки доступа к записи были определены так, как показано на рис. 11.10, то операции ввода с устройства SYS$INPUT могут быть выполнены с помощью следующих шагов программы:

```
$OPEN     FAB=F1    ;ОТКРЫТЬ ФАЙЛ
$CONNECT  RAB=R1    ;ПОДКЛЮЧИТЬ R1 К СВОЕМУ FAB
.
.
.

$GET      RAB=R1    ;ЭТОТ ОПЕРАТОР МОЖЕТ БЫТЬ
.                   ;ПОВТОРЕН, ИЛИ ВЫПОЛНЯТЬСЯ В ЦИКЛЕ
.
.

$CLOSE    FAB=F1    ;ЗАКРЫТИЕ ВЫПОЛНЯЕТСЯ
$EXIT_S             ;В КОНЦЕ ПРОГРАММЫ
```

Аналогично вывод на устройство SYS$OUTPUT можно выполнить следующим образом:

```
$CREATE   FAB=F2    ;ОТКРЫТЬ ФАЙЛ ДЛЯ ВЫВОДА
$CONNECT  RAB=R2A   ;ПОДКЛЮЧИТЬ R2A
.                   ;ЗАМЕТЬТЕ, ЧТО R2B ИЛИ R2C НЕ МОГУТ
.                   ;ИСПОЛЬЗОВАТЬСЯ ДО ТЕХ ПОР ПОКА
.                   ;НЕ ВЫПОЛНЕНО ПОДКЛЮЧЕНИЕ

$PUT      RAB=R2A   ;ЭТИ ОПЕРАТОРЫ МОГУТ
.                   ;ВЫПОЛНЯТЬСЯ НЕОДНОКРАТНО
.
.

$PUT      RAB=R2B   ;В ЛЮБОМ ПОРЯДКЕ
.                   ;ИЛИ В ЦИКЛАХ
.
.

$PUT      RAB=R2C
.
.
.

$CLOSE    FAB=F2    ;ЗАКРЫТИЕ ВЫПОЛНЯЕТСЯ
$EXIT_S             ;В КОНЦЕ ПРОГРАММЫ
```


ИСПОЛЬЗОВАНИЕ БЛОКОВ ДОСТУПА В ПРОГРАММЕ

Как упоминалось ранее, для формирования постоянной части блоков доступа к файлу и записи используются макроинструкции $FAB и $RAB. Однако эти блоки могут иметь переменные части, такие как адреса буферов, длина записей и строки имен файлов, которые не могут быть определены во время ассемблирования и с которыми поэтому приходится иметь дело во время выполнения. Чтобы осуществить это, необходимо просто определить местонахождение соответствующих длинных слов, слов, байтов или битов в блоках доступа и изменить или проверить их значения. Помня основные положения о том, как это делается в операционной системе VAX/VMS, все поля в блоках доступа идентифицируются символически, вместо того чтобы указывать их действительное местонахождение.

При вызове макроинструкций $FAB и $RAB происходит определение ряда символических имен, задающих расположение различных частей блока доступа относительно начала блока. Символические имена, соответствующие уже обсуждавшимся параметрам, таковы:

Символическое имя	Значение
FAB$L_FNA	Длинное слово, в котором содержится адрес строки имени файла в коде ASCII
FAB$B_FNS	Байт, в котором содержится длина строки имени файла.
RAB$L_FAB	Длинное слово, в котором содержится адрес блока доступа к файлу в блоке доступа к записи
RAB$L_RBF	Длинное слово, в котором содержится адрес буфера записи
RAB$W_RSZ	Слово, в котором содержится длина буфера записи в байтах
RAB$L_UBF	Длинное слово, в котором содержится адрес буфера пользователя
RAB$W_USZ	Слово, в котором содержится длина буфера пользователя в байтах

В качестве примера того, как это может использоваться, давайте представим себе подпрограмму для записи символьной строки в файл. Подпрограмма будет вызываться с помощью стандартных вызывающих последовательностей, описанных в гл. 9. Подпрограмма будет иметь два аргумента. Первым аргументом будет адрес дескриптора символьной строки, подлежащей записи в файл. Вторым аргументом будет адрес для блока доступа к предварительно открытому файлу, в который будет осуществляться вывод информации. Эта подпрограмма показана на рис. 11.11. Вспомним, что дескриптор символьной строки состоит из двух длинных слов. Первое слово в первом длинном слове содержит длину строки. Второе длинное слово содержит адрес символьной строки.

Заметим, что определенные выше символические имена дают адреса относительно начала блока доступа. Поэтому, чтобы адресовать в блоке доступа поле, в которое заносится значение параметра RSZ, необходимо вычислить исполнительный адрес RBLOCK + + RAB$W_RSZ.

```
        .TITLE   ПРОГРАММА ВЫВОДА СТРОКИ
        .PSECT   SWIOBLOCK,LONG
                                  ;ПУСТОЙ RAB ДОЛЖЕН БЫТЬ ЗАПОЛНЕН
RBLOCK: $RAB
        .PSECT
        .ENTRY   STRINGWRITE,0
        MOVL     8(AP),RBLOCK+RAB$L_FAB ;ЗАГРУЗИТЬ АДРЕС RAB
                                  ;ПОЛУЧИТЬ УКАЗАТЕЛЬ ДЕСКРИПТОРА
        MOVL     4(AP),R0
        MOVL     4(R0),RBLOCK+RAB$L_RBF ;ЗАГРУЗИТЬ АДРЕС СТРОКИ
        MOVW     (R0),RBLOCK+RAB$W_RSZ  ;ЗАГРУЗИТЬ ДЛИНУ СТРОКИ
        $CONNECT RAB=RBLOCK       ;ПОДКЛЮЧИТЬ RBLOCK
        $PUT     RAB=RBLOCK       ;ВЫВЕСТИ СТРОКУ
        RET
        .END
```

Рис. 11.11. Подпрограмма вывода строки

Вообще для распечатки сообщений предпочтительно использовать такой прием, как приведенный выше. Если для каждого сообщения используется отдельный блок доступа к записи, то теряется полезная память, причем в значительных объемах.

УПРАВЛЕНИЕ ТЕРМИНАЛАМИ И УСТРОЙСТВАМИ ПЕЧАТИ

На терминале или устройстве печати запись обычно имеет вид строчки текста. Накопленный нами опыт показывает, что для ограничения концов строк используются управляющие символы, такие как возврат каретки и перевод строки. И хотя во многих операционных системах эти управляющие символы также применяются для ограничения конца записей, в подсистеме RMS обычно это не делается. Причина заключается в том, что подсистема RMS предназначена для ввода-вывода информации более общего вида. Если информацию в общем случае рассматривать как произвольные строки байтов, то числовые значения, такие как ^X0D, являются простым элементом данных в строке. Однако, поскольку ^X0D является также и кодом возврата каретки, будет возникать путаница, если это значение будет интерпретироваться и как управляющий символ конца записи. Чтобы избежать этого, во внутреннем формате записей, записанных подсистемой RMS, содержится дополнительное слово-счетчик байтов, которое дает число байтов информации в записи. Пользователь обычно не видит этого счетчика байтов как части записи. Напротив, при выполнении операции ввода $GET счетчик байтов помещается по относительному адресу RAB$W_RSZ в блоке доступа к записи.

Одно исключение из этого правила относится к вводу с терминала. Поскольку ввод с терминала производится человеком, печатающим на клавиатуре, использование счетчиков байтов было бы весьма неудобным. Поэтому обычно для завершения строк применяют символ возврата каретки. Однако, чтобы сделать ввод с терминала совместимым с вводом с любого другого устройства, подсистема RMS удаляет символ возврата каретки и возвращает число введенных байтов, помещая его по относительному адресу RAB$W_RSZ, как и при вводе с других устройств. Поскольку для завершения ввода записей с терминала могут применяться многие управляющие символы, пользователю может потребоваться узнать, какой из них был введен в действительности[1], и в самом ли деле оператор нажал клавишу возврата каретки, или же была нажата комбинация клавиш CTRL/W, или CTRL/G, или что-либо иное? Это можно определить, так как система RMS сохраняет фактический символ завершения строки в младшем байте слова, расположенного по относительному адресу RAB$W_STV0.

При создании файла, подлежащего распечатке, допускается включение управляющих символов в символьные строки. Эти управляющие символы, такие как "Возврат каретки", "Перевод строки" и "Перевод формата", будут выполнять свои обычные функции. Однако включение символов завершения строки в выводимые записи ведет к несовместимости вывода с вводом, при котором эти символы удаляются. Существует способ избежать такой несовместимости. В действительности имеется три варианта.

Чтобы избежать присутствия символов завершения строки в явном виде в выводимой записи, в блок доступа к файлу помещается параметр, показывающий, что этот файл в конечном счете предназначается для распечатки. Тогда при распечатке файла или выводе его на терминал на устройство будут передаваться соответствующие управляющие символы. Три способа осуществления этого имеют разную степень сложности, зависящую от сложности требуемого вывода. Первый способ состоит в том, что между строками помещаются символы возврата каретки и перевода строки. При втором способе используются символы управления кареткой, как это принято в Фортране. При третьем способе применяются более сложные управляющие коды, которые выходят за рамки этой главы. Чтобы выбрать один из этих способов, следует в макровызове $FAB для файла указать один из следующих параметров типа RAT (определяющих атрибуты записи):

1. $FAB FNM= ⟨SYS$OUTPUT⟩, RAT = ⟨CR⟩ – для вставки символов возврата каретки и перевода строки между строками. При распечатке этого файла каждая строка будет начинаться с символа перевода строки, а завершать строку будет символ возврата каретки.
2. $FAB FNM= ⟨SYS$OUTPUT⟩, RAT = ⟨FTN⟩ – для символов управления кареткой, используемых в Фортране. Первый символ каждой строки удаляется из распечатки и используется как управляющий для принтера. Нормально воспринимаемые на ЭВМ семейства VAX символы управления кареткой:
   пробел – перевод на следующую строчку перед печатью;
   0 – пропуск строки перед печатью;
   1 – перевод на новую страницу перед печатью;
   + – печать без продвижения на следующую строку (надпечатка строки);
   $ – в конце строки не выполняется возврат каретки (используется для наводящих сообщений при выводе с терминала).
3. $FAB FNM= ⟨SYS$OUTPUT⟩, RAT = ⟨PRN⟩ – для сложного управления устройством печати. Дальнейшую информацию об управлении этого типа вы можете найти в руководстве "VAX-11 Record Management Services Reference Manual".

НАВОДЯЩИЕ СООБЩЕНИЯ ПРИ ВВОДЕ С ТЕРМИНАЛА

Как было сказано в первой половине этой главы, при вводе с терминала обычно требуются некоторые наводящие сообщения. Без таких сообщений пользователь за терминалом может оказаться в неведении и сомнении по поводу того, ожидает ли программа ввода или нет. Такие сообщения могут, конечно, выводиться и обычным образом на устройство SYS$OUTPUT. Однако выдаче наводящих сообщений таким способом свойственно несколько недостатков:

1. Если сделано переназначение устройства SYS$OUTPUT, чтобы сохранить то, что обычно выводилось на терминал, то сообщение появится в несоответствующем месте.
2. Если программа выполняется в пакетном режиме, наводящие сообщения не нужны и будут лишь загромождать вывод.
3. Поскольку каждый ввод становится двойной операцией, это усложняет программу.

Чтобы избежать этих проблем, наводящие сообщения можно выдавать автоматически с помощью макровызова $GET. Для выполнения таких действий нужно внести три дополнения в блок доступа к записи. Первое – это установить флаг для указания того, что желателен вывод наводящего сообщения. Второе и третье дополнения определяют строку наводящего сообщения и ее длину. Эти три элемента могут быть помещены в блок доступа к записи с помощью следующих параметров:

1. ROP = ⟨RMT⟩	Флаг запроса наводящего сообщения.
2. PBF = адрес строки	Идентифицирует строку наводящего сообщения.
3. PSZ = длина строки	Задает длину строки сообщения.

Для примера следующая последовательность операторов может использоваться в программе для выдачи наводящего сообщения "Вводите данные –" всякий раз, когда данные должны считываться с терминала:

```
RSIZE=8
          .PSECT    IOBLOCK,LONG
INFILE:   $FAB      FNM=<SYS$INPUT>
INREC:    $RAB      FAB=INFILE,UBF=IBLK,USZ=RSIZE,-
                    ROP=<PMT>,PBF=PRMSG,PSZ=PRMSGE-PRMSG
          .PSECT
```

[1] Существует возможность выбора программистом своего собственного набора символов завершения строки. Однако обычный набор завершающих символов включает все управляющие символы, исключая те, которые используются в текстах, такие как "Табуляция", "Перевод строки", "Возврат на шаг назад", "Вертикальная табуляция" и специальные управляющие символы, такие как ^C, ^O, ^Q, ^R, ^S, ^U, ^Y и символ удаления.

```
PRMSG:   .ASCII   "ВВЕДИТЕ ДАННЫЕ - "
PRMSGE:
IBLK:    .BLKB    RSIZE
```

Тогда каждый раз при выполнении макровызова $GET RAB = INREC будет выдаваться наводящее сообщение и программа будет ожидать ввод данных с клавиатуры.

Как и в случае других параметров, значения параметров PBF и PSZ могут быть загружены в блок доступа к записи во время выполнения. Как и прежде, соответствующие этим параметрам поля в блоке доступа определяются с помощью символических имен, задающих адреса относительно начала блока доступа к записи. Символические имена, определяющие поля для значений параметров PBF и PSZ, таковы:

Символическое имя	Значение
RAB$L_PBF	Длинное слово, содержит адрес наводящего сообщения
RAB$B_PSZ	Байт, содержит длину сообщения

Обратите внимание, что поле длины сообщения состоит только из одного байта. Так сделано из предположения, что наводящие сообщения не будут очень длинными и, следовательно, одного байта достаточно.

ОШИБКИ ВВОДА-ВЫВОДА

Как и многие другие системные подпрограммы, подпрограммы, вызываемые подсистемой RMS, возвращают информацию об ошибках через регистр R0. Код ошибки может быть обнаружен по тому факту, что младший бит регистра R0 не равен 1. Как и прежде, самый простой способ проверки этого бита – с помощью инструкции BLBC. В большинстве случаев при обнаружении ошибки программа должна просто передать управление операционной системе. Макровызов $EXIT_S R0 позволяет вывести сообщение об ошибке, если код ошибки будет находиться в регистре R0.

Но один из кодов ошибок может быть полезным при работе с файлами. Его символическое имя RMS$_EOF. Значение помещается в регистр R0 при попытке осуществить чтение после того, как обнаружен конец файла. Этим можно воспользоваться при чтении файла, длина которого неизвестна. Программа продолжает чтение до тех пор, пока в регистре R0 не появится значение RMS$_EOF, после чего переходит к другой обработке. Для примера того, как это работает, можно воспользоваться следующей последовательностью операторов:

```
$GET    RAB=INFILE
CMPL    R0,#RMS$_EOF
BEQ     DONE
BLBC    R0,ERROR
```

Выполнение программы будет продолжено с метки DONE, если будет обнаружен конец файла; в случае же любой другой ошибки будет сделан переход на метку ERROR. При отсутствии ошибок выполнение программы будет продолжено со следующей инструкции.

ОБРАЗЕЦ ПРОГРАММЫ

В качестве примера работы подсистемы RMS на рис. 11.12 показана программа, которая копирует один файл в другой подобно тому, как это осуществляется по команде операционной системы COPY FILEA FILEB. Программа выдает сообщения пользователю о том, что необходимо ввести имена для входного и выходного файлов.

```
.TITLE    ПРОГРАММА КОПИРОВАНИЯ ФАЙЛА
NSIZE=40                              ;MAX ДЛИНА ИМЕНИ
RSIZE=80                              ;MAX ДЛИНА ЗАПИСИ
.PSECT    IOBLOCK,LONG
```



```
TRMFAB:   $FAB    FNM=<SYS$INPUT>                ;ДЛЯ ВВОДА ИМЕН ФАЙЛОВ
INFAB:    $FAB                                   ;ИМЕНА ФАЙЛОВ БУДУТ
OUTFAB:   $FAB    RAT=<CR>                       ;ПОЛУЧЕНЫ ПОЗДНЕЕ
TRMRAB:   $RAB    FAB=TRMFAB,ROP=PMT,USZ=NSIZE   ;ДРУГИЕ ПАРАМЕТРЫ
INRAB:    $RAB    FAB=INFAB,UBF=RBUFFER,USZ=RSIZE ;БУДУТ ПОЛУЧЕНЫ
OUTRAB:   $RAB    FAB=OUTFAB,RBF=RBUFFER         ;ДЛИНА БУДЕТ ПОЛУЧЕНА
          .PSECT
PMT1:     .ASCII  "ВВЕДИТЕ ИМЯ ФАЙЛА ВВОДА -"
PMT1S=.-PMT1                                     ;ВЫЧИСЛЕНИЕ ДЛИНЫ СООБЩЕНИЯ
PMT2:     .ASCII  "ВВЕДИТЕ ИМЯ ФАЙЛА ВЫВОДА -"
PMT2S=.-PMT2                                     ;ВЫЧИСЛЕНИЕ ДЛИНЫ СООБЩЕНИЯ
NAME1:    .BLKB   NSIZE
NAME2:    .BLKB   NSIZE
RBUFFER:  .BLKB   NSIZE
;
          .ENTRY  START,0
          $OPEN   FAB=TRMFAB                     ;ОТКРЫТЬ И ПОДКЛЮЧИТЬСЯ
          $CONNECT RAB=TRMRAB                    ;К ТЕРМИНАЛУ
          MOVAB   NAME1,TRMRAB+RAB$L_UBF         ;ПОДГОТОВКА БУФЕРА ИМЕН
          MOVAB   PMT1,TRMRAB+RAB$L_PBF          ;СТРОКИ ПОДСКАЗКИ
          MOVB    @PMT1S,TRMRAB+RAB$B_PSZ        ;И ДЛИНЫ ПОДСКАЗКИ
          $GET    RAB=TRMRAB                     ;ПОЛУЧИТЬ ИМЯ ФАЙЛА ВВОДА
          MOVAB   NAME1,INFAB+FAB$L_FNA          ;ПОМЕСТИТЬ ИМЯ В FAB
          MOVB    TRMRAB+RAB$W_RSZ,INFAB+FAB$B_FNS ;УСТАНОВИТЬ ДЛИНУ
          $OPEN   FAB=INFAB                      ;ОТКРЫТЬ ФАЙЛ ВВОДА
          BLBS    R0,10$                         ;ПРОВЕРКА НАЛИЧИЯ ОШИБКИ
          JMP     ERROR
10$:      $CONNECT RAB=INRAB                     ;И ПОДКЛЮЧИТЬ ФАЙЛ ВВОДА
          BLBS    R0,20$                         ;ПРОВЕРКА НАЛИЧИЯ ОШИБКИ
          JMP     ERROR
20$:      MOVAB   NAME2,TRMRAB+RAB$L_UBF         ;УСТАНОВКИ ДЛЯ
          MOVAB   PMT2,TRMRAB+RAB$L_PBF          ;ВВОДА ИМЕНИ
          MOVB    @PMT2S,TRMRAB+RAB$B_PSZ        ;ФАЙЛА ВЫВОДА
          $GET    RAB=TRMRAB
          MOVAB   NAME2,OUTFAB+FAB$L_FNA         ;И ОТКРЫТИЯ
          MOVB    TRMRAB+RAB$W_RSZ,OUTFAB+FAB$B_FNS ;ФАЙЛА
          $CREATE FAB=OUTFAB                     ;ВЫВОДА
          BLBS    R0,30$                         ;ПРОВЕРКА НАЛИЧИЯ ОШИБКИ
          JMP     ERROR
30$:      $CONNECT RAB=OUTRAB
          BLBS    R0,40$                         ;ПРОВЕРКА НАЛИЧИЯ ОШИБКИ
          JMP     ERROR
;
;В ЭТОМ МЕСТЕ В ЦИКЛЕ ВЫПОЛНЯЕТСЯ КОПИРОВАНИЕ СОДЕРЖИМОГО ФАЙЛА
;
40$:      $GET    RAB=INRAB                      ;СЧИТАТЬ ЗАПИСЬ
          CMPL    R0,#RMS$_EOF                   ;КОНЕЦ ФАЙЛА?
          BEQL    50$                            ;ДА
          BLBC    R0,ERROR                       ;НЕТ, ДРУГИЕ ОШИБКИ?
          MOVW    INRAB+RAB$W_RSZ,OUTRAB+RAB$W_RSZ
                                                 ;ПОЛУЧИТЬ ДЛИНУ ЗАПИСИ
          $PUT    RAB=OUTRAB                     ;ВЫВЕСТИ ЗАПИСЬ
          BLBS    R0,40$                         ;ПОВТОРЯТЬ В ЦИКЛЕ,
                                                 ;ЕСЛИ НЕТ ОШИБОК
                                                 ;ДО КОНЦА ФАЙЛА
50$:      $CLOSE  FAB=INFAB                      ;ЗАКРЫТЬ ФАЙЛЫ
          $CLOSE  FAB=OUTFAB                     ;ЗАВЕРШИТЬ ВЫПОЛНЕНИЕ
          $EXIT_S
;                                                ;ЗАВЕРШЕНИЕ ПО ОШИБКЕ
ERROR:    $EXIT_S R0
;         .END    START
```

Рис. 11.12. Программа ввода-вывода

УПРАЖНЕНИЯ 11.2

1. Ниже даны относительные адреса полей в блоках доступа к файлу и записи; для каждого из этих полей ответьте на следующие вопросы:

Для чего оно используется?
Появляется ли оно в блоке доступа к файлу или в блоке доступа к записи?
Длина этого поля байт, слово или длинное слово?
Значение является обязательным или нет?
Значение задается в программе? Если да, когда оно используется?
Значение возвращается программе системой? Если да, то когда?

а.	FAB$L_FNA	б.	FAB$B_FNS
в.	RAB$L_PBF	г.	RAB$B_PSZ
д.	RAB$L_RBF	е.	RAB$B_RSZ
ж.	RAB$L_UBF	з.	RAB$B_USZ

2. Напишите программу, которая считывает с терминала некоторое число записей, используя макровызовы подсистемы RMS, и распечатывает их на терминале в обратном порядке. Например, если вы вводите с клавиатуры "HELLO, HOW ARE YOU?", программа распечатает "?UOY ERA WOH, OLLEH". Программа будет циклически осуществлять чтение и распечатку до тех пор, пока не будет считана запись нулевой длины, после чего программа передает управление операционной системе. Используйте форматный вывод, добавив в выводимые записи символы возврата каретки и перевода строки.

3. Напишите программу, которая будет выполнять чтение файла на диске, записи в котором имеют в начале каждой строки символы управления кареткой, используемые в Фортране. Содержимое файла будет выводиться на системное устройство с логическим именем SYS$PRINT. Выведенная информация будет иметь обычную страничную организацию, присущую всем распечаткам (если устройство SYS$PRINT недоступно для начинающих пользователей, ваш наставник может переназначить вывод на устройство SYS$OUTPUT или в какой-либо другой файл вывода). Подсказка: используйте параметр атрибутов записи (RAT).

4. Повторите п. 3, но не используйте параметр RAT. Вместо этого ваша программа должна удалять и интерпретировать символы управления кареткой и вставлять управляющие символы кода ASCII CR, LF и FF в соответствующие места для получения распечатки желаемого вида.

5. Рассмотрим базу данных служащих предприятия, в которой имеются сведения о 20 служащих, хранящиеся в одном файле. Запись о каждом сотруднике состоит из пяти частей:

а) номер служащего – 2 цифры;
б) имя служащего – 25 символов;
в) адрес служащего – 100 символов (может включать символы возврата каретки и перевода строки);
г) номер страхового полиса служащего – 9 цифр, 2 тире;
д) оклад служащего – 8 цифр и десятичная точка.

Нарисуйте структурную схему системы, которая принимает и выполняет следующие однобуквенные команды:

A – создать новый файл с 20 пустыми записями;
B – найти и распечатать запись по ее номеру;
C – позволяет пользователю ввести или изменить поле имени;
D – позволяет пользователю ввести или изменить поле адреса;
E – позволяет пользователю ввести или изменить поле номера страхового полиса;
F – позволяет пользователю ввести или изменить поле оклада;
G – перезаписать старую запись новой информацией;
H – закончить работу и закрыть файл(ы).

*6. Используя дополнительную информацию, которой нет в этой главе, но которая имеется в реководстве "VAX-11 Record Manager Reference Manual", напишите программу на языке ассемблера, реализующую систему, которая описана в п. 5. В программе следует использовать файл прямого доступа, содержащий 20 записей, которые могут считываться и записываться в любом порядке.

ГЛАВА 12. ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ

12.1. ВВЕДЕНИЕ

В гл. 6 рассматривались представление целых чисел и арифметические операции над этими числами. Представление целых чисел является примером представления чисел *с фиксированной точкой*, так как подразумевается, что десятичная точка находится на строго определенном месте. Если целое число содержится в 32-битовом длинном слове, то подразумевается, что десятичная точка находится справа от крайнего правого бита длинного слова. Хотя представление чисел с фиксированной точкой пригодно для многих приложений, оно является неудобным для тех задач, в которых приходится иметь дело как с очень малыми дробными, так и с очень большими значениями. Таких задач существует много в науке и технике, а также в других областях.

Представление чисел *с плавающей точкой* было разработано для эффективного представления как дробных, так и очень больших значений чисел. Представление чисел с плавающей точкой сходно с экспоненциальным форматом записи чисел в том, что числа представляются как последовательности значащих цифр, умноженных на какое-либо число, возведенное в степень.

В этой главе читатель познакомится с тем, как вообще осуществляется обработка дробных значений и как это делается, в частности, на ЭВМ семейства VAX. Будут изучены различные представления чисел с плавающей точкой, применяемые в ЭВМ семейства VAX, и инструкции для работы с этими числами.

12.2. ЧИСЛА С ФИКСИРОВАННОЙ И С ПЛАВАЮЩЕЙ ТОЧКОЙ

Для того чтобы эффективно использовать числа с фиксированной и с плавающей точкой, важно понять принципы работы с этими числами. Чтобы упростить изложение, числа *с фиксированной* и *плавающей* точкой будут рассматриваться на основе десятичного представления. После этого будет рассмотрено представление этих чисел в ЭВМ семейства VAX и показано, какие искусные приемы применяются для того, чтобы такое представление было эффективным.

Существуют разнообразные способы выражения дробных значений. Наиболее общий способ основывается на использовании пар целых чисел, таких как 1/2, 5/12 или 537/8946. Этот способ представления дробей довольно легко может быть реализован на ЭВМ. Однако обычно он не применяется в вычислениях из-за свойственного ему неудобства. Вместо него более предпочтительным является ограничение дроби множеством стандартных знаменателей, таких как десятки, сотни, тысячи и т. д. Поскольку эти знаменатели представляют собой степени числа 10, образованные таким образом дроби называются десятичными. Обычно они изображаются с десятичной точкой или запятой в позиционном представлении чисел, например 5.4, 7.2 или 0.093. Существуют также и другие системы построения дробей, в которых используются знаменатели, такие как 12, 14, 16, 32 и 60. И хотя некоторые из них вышли из употребления с введением метрической системы, оставшиеся еще находят применение. В большинстве ЭВМ используются или десятичные, или двоичные дроби (которые будут рассмотрены позже).

ЧИСЛА С ФИКСИРОВАННОЙ ТОЧКОЙ

Для работы с нецелочисленными значениями в ЭВМ широко применяются два способа представления чисел: с фиксированной и с плавающей точкой. Наиболее простым из них является представление чисел с фиксированной точкой. Этот способ представления основан на том, что часто имеется такая наименьшая дробь, которая достаточна для представления наименьшего значения и при которой не надо рассматривать еще какие-либо меньшие значения. Хорошей иллюстрацией этого являются финансовые расчеты. В большинстве финансовых расчетов в США используется такая денежная единица, как доллар, а в качестве наименьшей единицы рассматривается один цент, или 1/100 долл.[1].

Поскольку нет нужды иметь дело с долями центов, в постановке задачи можно было бы перейти к целому числу центов. Например, 537.23 долл. – это то же самое, что 53 723 центов. Однако, поскольку большинство людей путаются, когда им приходится иметь дело с большим числом центов, а не долларов, более предпочтительно о том же самом говорить так, что число 53 723 имеет подразумеваемую десятичную точку, отделяющую две позиции справа. Поэтому это число представляет 537.23 долл. Это называется представлением чисел с фиксированной точкой, так как десятичная точка подразумевается на некотором фиксированном месте в числе.

Представление чисел с фиксированной точкой очень удобно для финансовых расчетов; именно поэтому оно широко используется в предназначенных для экономических приложений языках, таких, например, как Кобол. Числа с фиксированной точкой могут также применяться в научных расчетах, при приведении данных к соответствующей шкале единиц. Поэтому в научных задачах очень часто используются целочисленные значения, такие, как миллиамперы, микросекунды, сантиметры, килограммы, тонны. Последние два элемента списка представляют вариацию описанной ранее проблемы долларов и центов, поскольку десятичная точка эффективно переносится вправо, за пределы числа. Следовательно, вместо подсчета долей некоторой единицы можно подсчитать значения, кратные какой-либо единице. Иногда это необходимо, чтобы избежать вычисления с многократно увеличенной точностью, к которым приходится прибегать при работе с очень большими значениями. Например, если вес супертанкера выражать в миллиграммах, то потребовались бы многозначные числа. Следовательно, должны использоваться тонны или даже килотонны.

Проблема приведения величин к соответствующей шкале единиц может быть довольно сложной и была фактически одной из самых трудных при программировании на некоторых первых моделях ЭВМ. Трудность заключается в том, что во многих научных задачах приходится иметь дело как с очень большими, так и с очень маленькими числами. Например, при рассмотрении корабля с ядерной силовой установкой могут использоваться килотонны в отношении массы корабля и миллиграммы в отношении массы гранул топлива, так что единицы массы в рамках одной задачи оказываются несогласованными.

ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ

Решение вышеназванной проблемы состоит в использовании чисел *с плавающей точкой*. Число с плавающей точкой в действительности состоит из двух частей. Одна часть содержит знак и цифры числа, другая часть показывает, в каком месте предполагается нахождение десятичной точки. Аналогичный способ часто используется учеными для того, чтобы их вычисления могли выполняться в стандартных единицах. В применяемой ими записи числа представляются в виде нескольких значащих цифр, умноженных на некоторую степень числа 10. Например,

$$-5.347 \times 10^{15} = -5347000000000000.$$

или

[1] В некоторых вычислениях процента налога и прибыли имеют дело с долями цента, такой, как, например 1/1000 долл. Для этих случаев описанная здесь схема представления чисел с фиксированной точкой должна быть модифицирована.

$$4.92 \times 10^{-9} = 0.000000004912$$

В действительности это есть $\pm a \times 10^{\pm b}$, где a представляет значащие цифры числа, а b представляет степень числа 10 – порядок числа. Обратите внимание, что оба числа, как a, так и b, имеют знак.

При представлении в ЭВМ чисел с плавающей точкой требуется где-то хранить два числа со знаком -. Это $\pm a$ и $\pm b$. Требуется также понять, что эти числа означают. Хотя оба числа можно хранить в двух отдельных ячейках памяти, это неэффективно, поскольку b имеет тенденцию быть относительно небольшим числом, и было бы расточительно отводить для хранения этих чисел одинаковый объем памяти. Вместо этого обычно используют некоторые средства для совместной упаковки обоих чисел. Предположим, что имеется вычислительная машина, слово которой вмещает 8-разрядное число со знаком, например +73214694 (напомним, что временно используется десятичное представление).

Хорошим компромиссом для упаковки чисел a и b в это слово было бы отвести два разряда слова для b и остальные шесть разрядов – для a. Это дает шесть *значащих* цифр в числе и диапазон значений, определяемый множителем 10^{99}. Таким образом, если в машинном слове содержится значение +51314159, то его часть +314159 соответствует a, а 51 соответствует b. Возникают два вопроса. Первый: "Где в числе a подразумевается положение десятичной точки?" Хотя положение точки может быть произвольным, большинство производителей ЭВМ помещают ее в самой левой позиции. Следовательно, a = +0.314159. Второй вопрос: "Что произошло со знаком для числа b?" Машинное слово имеет только один знаковый разряд, который использовался для знака числа a. Однако, если требуется представлять и малые, и большие числа, необходимо иметь знак также и для числа b.

Обычный метод состоит в том, чтобы в двух разрядах, отведенных для представления b, хранить число, которое больше действительного значения b на некоторое строго определенное значение. Если такое фиксированное значение равно 50, то заданное в этих разрядах число 50 означает, что число b равно 0, число 51 означает, что число b равно 1, число 52 означает b = 2, и т. д. Числа, меньшие 50, служат для представления отрицательных порядков. Например, число 49 означает, что b равно –1, а число 41 означает, что b равно –9. Поскольку число 51 соответствует b, равному 1, содержимое слова +51314159 представляет число $+0.314159 \times 10^1$ – или просто 3.14159. Чтобы отличить два представления порядка, значение b будем называть *истинным* значением порядка, а значение b + 50 – *порядком с избытком* 50 или просто порядком с избытком. На рис. 12.1 показаны примеры чисел с плавающей точкой и их эквиваленты.

Как видно из рис. 12.1, имеются три явно выраженных части слова с плавающей точкой: знаковый разряд для a; разряды, в которых представлен порядок b; разряды, отведенные

Число с плавающей точкой	Экспоненциальная форма	Десятичное представление
–46134926	-0.134926×10^{-4}	–0.0000134926
+50934821	$+0.934821 \times 10^{0}$	0.934821
+50999999	$+0.999999 \times 10^{0}$	0.999999
+51100000	$+0.100000 \times 10^{1}$	1.00000
–53426910	-0.426910×10^{3}	–426.910

Рис. 12.1. Представление десятичных дробей

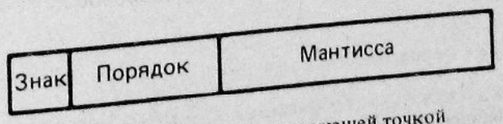


Рис. 12.2. Формат числа с плавающей точкой

для представления значения a. Последние обычно называют *дробной частью* или *мантиссой* вследствие подразумеваемого положения десятичной точки в a (как отмечалось выше, разряды, в которых представлено значение b, называют порядком с избытком). На рис. 12.2 показаны составные части формата числа с плавающей точкой.

НОРМАЛИЗОВАННЫЕ ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ

Последнее замечание по поводу представления чисел с плавающей точкой касается *нормализации*. Из предыдущего обсуждения можно видеть, что содержимое слова +51100000 представляет собой $0.1 \times 10^1 = 1$. Аналогично +54000100 представляет $0.0001 \times 10^4 = 1$. Следовательно, в любом случае +51100000 и +54000100 представляют одно и то же число. Чтобы предотвратить возможную путаницу в большинстве систем представления чисел с плавающей точкой, требуется, чтобы число было приведено к такому виду, при котором самый левый разряд мантиссы не был бы равен 0 (как в +51100000). Это называют *нормализованным* представлением числа с плавающей точкой. Такое требование помимо устранения путаницы важно прежде всего потому, что нормализованные числа с плавающей точкой сохраняют максимальное число значащих цифр. При работе с ненормализованными числами, такими как +54000100, может теряться точность. Одно исключение из правила нормализации касается числа 0, которое имеет нормализованное представление +00000000.

Посмотрим теперь, какой диапазон значений возможен при таком представлении чисел с плавающей точкой. Диапазон возможных значений приведен на рис. 12.3. Обратите внимание, что существует разрыв между значениями 10^{-51} и 0. Это означает, что следует избегать употребления слишком маленьких чисел, так как информационной емкости слова может оказаться недостаточно для их представления.

Наименьшее число	−99999999	$= -0.999999 \times 10^{49} \approx -10^{49}$	
Наибольшее отрицательное число	−00100000	$= -0.100000 \times 10^{-50} = -10^{-51}$	
Нуль	+00000000	$= 0.000000 \times 10^{0} = 0$	
Наименьшее положительное число	+00100000	$= 0.100000 \times 10^{-50} = +10^{-51}$	
Наибольшее число	+99999999	$= 0.999999 \times 10^{49} \approx +10^{49}$	

Рис. 12.3. Диапазон нормализованных десятичных дробей

12.3. ОПЕРАЦИИ НАД ЧИСЛАМИ С ПЛАВАЮЩЕЙ ТОЧКОЙ

СЛОЖЕНИЕ И ВЫЧИТАНИЕ

Сама по себе возможность представления чисел в формате с плавающей точкой по сути бесполезна, если только не существует какого-либо способа выполнения операций над такими числами. На ЭВМ обычно можно выполнять операции сложения, вычитания, умножения и деления (другие математические операции и функции являются производными от этих четырех). В этом разделе будет показано на примере десятичных чисел, как эти операции могут выполняться над числами с плавающей точкой, представление которых было описано в предыдущем разделе.

Сначала рассмотрим сложение и вычитание (эти операции очень близки и различаются только тем, как обрабатывается знак числа). Рассмотренное в предыдущем разделе правило гласит, что первый шаг сложения десятичных чисел состоит в выравнивании положения точек. Таким образом,

$$\begin{array}{r} 573.426 \\ +8.93425 \\ \hline \end{array} \quad \text{должно быть перезаписано как} \quad \begin{array}{r} 573.426 \\ +8.93425 \\ \hline 582.36025 \end{array}$$

После этого выполняется простое сложение значений каждого разряда чисел. Аналогичное

правило применимо к числам, представленным как в экспоненциальной форме, так и с плавающей точкой. Представленные в такой форме два числа нельзя складывать до тех пор, пока не будут равны их порядки. Следовательно, для аналогичного примера

$$\begin{array}{r} .573426 \times 10^3 \\ +.893425 \times 10^1 \\ \hline \end{array} \quad \text{должно быть переписано как} \quad \begin{array}{r} .573426 \times 10^3 \\ +.00893425 \times 10^3 \\ \hline .58236025 \times 10^3 \end{array}$$

Проследим теперь шаг за шагом обработку тех же самых чисел, представленных в формате с плавающей точкой, и посмотрим, как такая обработка может быть осуществлена в ЭВМ.

Шаг 1. Выровнять два числа одно под другим:

+53573426
+51893425

Шаг 2. Распаковать числа, чтобы отделить друг от друга мантиссу и порядок:

53 +573426
51 +893425

Шаг 3. Расположить числа так, чтобы первым было число с наибольшим порядком, переставив числа, если необходимо:

53 +573426
51 +893425

Шаг 4. Вычислить разность порядков:

53 +573426
−51 +893425
2

Шаг 5. Сдвинуть второе число вправо на число разрядов, равное вычисленной разности, и сделать порядки чисел равными:

53 +573426
53 +008934 | 25^1

Шаг 6. Сложить мантиссы. Порядок результата остается тем же самым:

53 +573426
53 +008934
53 +582360

Шаг 7. Упаковать результат в формат с плавающей точкой:

+53582360

Результат сложения равен 582.36.

СЛОЖНОСТИ, СВЯЗАННЫЕ СО СЛОЖЕНИЕМ И ВЫЧИТАНИЕМ

Алгоритм, рассмотренный выше, без сомнения правильно работает для приведенного примера. Однако могут возникнуть две сложности, которые требуют двух дополнительных шагов. Первая проблема состоит в том, что сумма двух чисел может занимать больше разрядов, чем имеется у первоначальных чисел. Например, предположим, что складываются числа в формате с плавающей точкой +53573426 и 53698421. После выполнения шагов 1 – 5 обнаруживается, что сдвиг не нужен, и будет получено

53 +573426
53 +698421

[1] Эти цифры теряются, если только операция не выполняется с двойной точностью. Иначе можно было бы округлить число, если самая левая из этих цифр 5 или больше. Такое усечение или округление приводит к неизбежным погрешностям в вычислениях.

Теперь перейдем к шагу 6

53 +573426
53 +698421
+1271847

Отметим, что результирующая дробная часть числа занимает более шести разрядов, а это не позволит упаковать его в восьмиразрядное слово. Решение состоит в том, чтобы сдвинуть мантиссу на один разряд вправо и прибавить единицу к порядку числа:

Шаг 6а.

54 +127184|7[1]

Шаг 7а. Упаковать результат в формат с плавающей точкой.

+54127184

Вторая проблема в некотором смысле противоположна первой. Она возникает при вычитании чисел с одинаковыми знаками или при сложении чисел с противоположными знаками. Опять шаги 1 – 5 – те же самые. Но при выполнении вычитания на шаге 6 результат может занимать менее шести разрядов. Например, если складываются числа +53573426 и –53573213, то на шаге 6 будет следующее:

53 +573426
53 –573213
53 +000213

Отметим, что если упаковать это число, результат окажется ненормализованным. Поэтому необходимо нормализовать результат и получить шаг 6б.

Шаг 6б.

50 +213000

Заметим, что нули справа означают потерю точности. Это обычно происходит при вычитании очень близких по значению чисел. На шаге 6б нужно иметь в виду, что результат может быть нулевым. В этом случае при нормализации будет получено

00 +000000

УМНОЖЕНИЕ И ДЕЛЕНИЕ

Правила умножения и деления чисел с плавающей точкой прямо вытекают из правил, применяемых к числам в экспоненциальной форме. При умножении порядки складываются, при делении порядки вычитаются. Дробные части чисел или умножаются, или делятся. Например,

$(0.5 \times 10^{15}) \times (0.8 \times 10^{4}) = (0.5 \times 0.8) \times 10^{15+4} = 0.4 \times 10^{19}$.

Аналогично

$(0.4 \times 10^{19}) / (0.5 \times 10^{15}) = (0.4/0.5) \times 10^{19-15} = 0.8 \times 10^{4}$.

Правила умножения чисел с плавающей точкой представлены ниже:

Шаг 1. Расположить два числа одно под другим:

+65500000
+54800000

Шаг 2. Распаковать числа, как при сложении:

65 +500000
54 +800000

Шаг 3. Теперь умножаются мантиссы. Для мантиссы или порядков никакой подстройки не требуется (замечаем, где в результате оказывается десятичная точка, т. е. 0.50 × 0.80 равно 0.4000):

65 +0.500000
54 × +0.800000
+0.400000

Шаг 4. Складываются порядки. Но каждый порядок имеет избыток 50, так что результат будет иметь избыток 100. Следовательно, необходимо вычесть 50:

65
+ 54
119
– 50
69 + 400000

Шаг 5. Теперь нужно упаковать результат.

+69400000

Поскольку дробные части всегда меньше 1, произведение двух мантисс должно быть меньше 1. Следовательно, проблема, которая существовала при сложении на шаге 6а, здесь не возникает. Однако произведение может быть меньше чем 0.1, поэтому иногда необходима нормализация. Она выполняется так же, как на шаге 6б, при сложении. Например, если число +51150000 умножается на +52200000, то будут выполнены

Шаги 3 и 4:

51 + +0.150000
+ 52 × +0.200000 (заметим, что 0.15 x 0.20 равно 0.0300)
103 0.030000
– 50
53 +030000

Необходима нормализация, чтобы получить

Шаг 4а:

52 +300000

Это дает результат, равный +52300000. Обратите внимание, что в примере фактически выполняется – 1.5 x 20 равно 30.

Операция деления в существенной степени аналогична операции умножения. Подробности выполнения операции деления читателю предлагается разобрать самостоятельно в качестве упражнения.

ОБСУЖДЕНИЕ

Представление чисел с плавающей точкой, рассмотренное выше, может показаться неудобным. Слева направо оно состоит из разряда знака мантиссы, двух разрядов порядка, представленного с избытком 50, и шестиразрядной мантиссы. Такой формат был выбран, чтобы упростить сравнение двух чисел с плавающей точкой.

Предположим, что желательно определить, больше ли число с плавающей точкой X, чем число с плавающей точкой Y. Это можно сделать, выполнив вычитание этих чисел Y – X и проверив, отрицателен ли результат. Однако существует более быстрый и легкий способ. Нужно рассмотреть представления чисел X и Y, как если бы они были восьмиразрядными целыми со знаком, и вычислить Y – X, используя *целочисленную* арифметику. Если полученное в результате целое число отрицательно, то X (как число с плавающей точкой) больше, чем Y. Например, предположим, что X равно 10.0, а Y равно 1.0. Представление этих чисел в формате с плавающей точкой для X и Y будет соответственно +52100000 и +51100000. Вычитание +52 100 000 из +51 100 000 как целых со знаком даст –01 000 000, а это пока-

[1] Эта цифра теряется. Она может быть сохранена при выполнении операци с двойной точностью или использована для округления результата до +127185. Это аналогично тому, что случилось на шаге 5, и также может отразиться на погрешности вычислений.

зывает, что X(10.0) больше, чем Y(1.0) (в данном случае оба числа X и Y положительны и порядок числа X превосходит порядок числа Y на 1, так что X больше).

Предлагаем читателю проверить справедливость этого положения для любой пары чисел с плавающей точкой при условии, что числа нормализованы. Чтобы такой краткий вариант сравнения работал, порядок должен располагаться между знаком мантиссы и самой мантиссой. Этим объясняется и использование представления порядка с избытком (в данном случае 50). На ЭВМ, не имеющих инструкций для сравнения чисел с плавающей точкой, варианты этого краткого способа используются для создания эффективного программного обеспечения для сравнения чисел с плавающей точкой. На ЭВМ, имеющих такие инструкции, этот способ может использоваться для упрощения конструкции аппаратуры.

ОШИБКИ ПРИ ОПЕРАЦИЯХ НАД ЧИСЛАМИ С ПЛАВАЮЩЕЙ ТОЧКОЙ

В процессе вычислений с плавающей точкой могут возникнуть различные ошибки. Например, попытка разделить число с плавающей точкой на 0 приводит к ошибке *деления на нуль числа с плавающей точкой*. Возможно также, что операции над числами с плавающей точкой могут приводить к *переполнению числа с плавающей точкой*. Как показано на рис. 12.3, в приведенном формате с плавающей точкой могут быть представлены только числа от -10^{-49} до 10^{49}. Попытка образовать число, не попадающее в этот диапазон, приводит к переполнению числа с плавающей точкой. Рассмотрим, например, следующее умножение:

+98800000
× +54400000

Мантисса результата – это .800000, умноженное на .400000, или .320000. Порядок с избытком 50 равен 98 плюс 54 минус 50 или 102. Правильный результат умножения есть $.32 \times \times 10^{52}$ (порядок с избытком 50, равный 102, представляет истинный порядок 52). Поскольку порядок с избытком 50 больше 99, он не может уместиться в двухразрядном поле порядка и происходит переполнение числа с плавающей точкой.

Рассмотрим следующее умножение:

+02800000
× +40400000

Мантисса результата – это опять .800000, умноженное на .400000, или .320000. Однако порядок с избытком 50 есть 02 плюс 40 минус 50 или –08. Правильным результатом умножения будет $.32 \times 10^{-58}$ (порядок –08 с избытком 50 представляет истинный порядок –58). Поскольку порядок с избытком 50 меньше 0, в двухразрядном поле порядок не умещается. Такая ошибка называется *потерей значимости числа с плавающей точкой*.

Обратите внимание, что переполнение и потеря значимости числа с плавающей точкой весьма отличаются друг от друга. Переполнение числа с плавающей точкой означает, что абсолютное значение результата операции над числами с плавающей точкой слишком велико и не может быть представлено в данной разрядной сетке. Потеря значимости числа с плавающей точкой означает, что результат слишком близок к нулю и поэтому также не может быть представлен. Потеря значимости может происходить при вычитании двух небольших, очень близких по значению чисел, при умножении двух небольших по значению чисел или при делении небольшого числа на большое число.

В описанном представлении чисел с плавающей точкой порядок 00 с избытком 50 показывает, что истинный порядок есть –50. Например, число в формате с плавающей точкой

+00123456

представляет число $.123456 \times 10^{-50}$. Нулевой порядок не имеет какого-то специального значения. Представление чисел с плавающей точкой, используемое на целом ряде ЭВМ, следует этому соглашению. Однако на других ЭВМ, куда входят и ЭВМ семейств VAX и PDP-11, нулевой порядок рассматривается как некий специальный случай. В ЭВМ семейства VAX положительная мантисса при нулевом порядке интерпретируется как значение 0 независимо от действительного значения мантиссы. То есть любое число с плавающей точкой, аналогичное следующему:

+00??????

интерпретируется как 0. Причина, по которой это сделано именно так, станет ясной из описания различных форматов чисел с плавающей точкой, используемых на ЭВМ семейства VAX.

Когда нулевой порядок рассматривается таким образом, возникает вопрос о том, как интерпретировать числа с плавающей точкой, аналогичные следующему:

–00??????

Такие числа могли бы интерпретироваться как отрицательный 0. Но иметь два варианта 0 неудобно и это может приводить к ошибкам программирования. Вместо этого на ЭВМ семейства VAX такие числа рассматриваются как *резервные операнды*. Это попросту означает, что числа, аналогичные –00??????, рассматриваются как недопустимые числа с плавающей точкой, а попытка использования таких чисел в операциях над числами с плавающей точкой приводит к возникновению ошибки. Резервный операнд может оказаться и довольно полезным. Например, транслятор языка высокого уровня может инициализировать все переменные, имеющие формат числа с плавающей точкой, недопустимыми значениями. Если во время выполнения программа пытается обратиться к переменной с плавающей точкой без предварительной ее инициализации,то это немедленно вызовет ошибку по испольэованию резервного операнда.

УПРАЖНЕНИЯ 12.1.

1. Преобразуйте следующие числа в нормализованное представление, десятичное представление и в представление с плавающей точкой, как описано в этой главе:

а. 5
б. 374
в. 3.14159
г. 0.0005
д. 0.8035×10^{23}
е. 0.4923×10^{-15}
ж. 8.496×10^{18}
з. 954.2×10^{-12}

2. Преобразуйте следующие числа в формате с плавающей точкой в экспоненциальную и в обычную десятичную форму записи (без порядка):

а. +51300000
б. –53742000
в. +50894026
г. +45805216
д. –56293465
е. –57100000
ж. +38950125
з. –64790881

3. Выполните показанные операции над следующими парами чисел в формате с плавающей точкой. Покажите, какие шаги должны быть при этом выполнены. Выразите свои результаты в виде нормализованных чисел в формате с плавающей точкой.

а. (+53215904) + (+53116895)
б. (+52159099) + (+49889621)
в. (+50912065) – (+54891126)
г. (–52998046) + (–50479138)
д. (–53885304) – (–53885034)
е. (+57900000) × (+48800000)

ж. (+51426931) × (−44357926)
з. (−43250000) × (−41250000)
и. (−55255000) / (+51500000)
к. (−41800000) / (−44200000)

4. Перечислите все шаги, необходимые для выполнения деления чисел с плавающей точкой. Выполнение каких шагов зависит от конкретных условий? Является ли проблемой нормализация (в предположении, что операнды нормализованы)? Происходит ли переполнение в мантиссе числа, как на шаге 6а сложения? Как велико может быть переполнение в предположении, что операнды нормализованы?

5. Популярный метод вычисления квадратных корней на ЭВМ основан на использовании так называемой формулы Ньютона–Рафсона. Чтобы вычислить квадратный корень N, вы задаете некоторое приближенное к ответу число (назовем его X). Затем применяете формулу

$$X_{нов} = 0.5 \times (X_{стар} + N/X_{стар}).$$

Новое значение X будет много ближе к правильному корню квадратному, чем старое значение. Формула может применяться повторно для получения ответа желаемой точности. Скорость сходимости метода зависит от числа повторных вычислений по формуле, а оно, в свою очередь, зависит от того, насколько близко значение первоначально заданного числа к искомому результату, а также от желаемой точности результата:

а) с точки зрения математики, что представляет собой корень квадратный из числа при его записи в экспоненциальной форме? То есть как соотносятся порядок и мантисса корня квадратного с порядком и мантиссой числа, из которого извлекается корень?

б) каким образом ответ на пункт а) может обеспечить простой метод удачного задания приближенного значения для корня квадратного, извлекаемого из числа с плавающей точкой?

в) насколько сильно отличается в наихудшем случае заданное вами приближенное число от правильного ответа?

г) если ориентироваться на наихудшие случаи, то сколько итераций для формулы Ньютона–Рафсона потребуется для получения ответа с точностью до шести значащих цифр (для проверки наихудших случаев воспользуйтесь калькулятором или ЭВМ)?

*6. Напишите программу на языке ассемблера ЭВМ семейства VAX, которая вводит символьную строку, представляющую десятичное число со знаком. Строка будет состоять из знака, десятичных цифр и десятичной точки, например +89.462, –0.009461 и т. п. Затем программа должна распечатать нормализованное представление в формате с плавающей точкой, такое как +52894620 и –48946100 соответственно. Программа должна в цикле обработать по крайней мере 20 различных чисел. Некоторые подсказки для решения этой задачи:

а) игнорируйте ведущие нули в числе (за исключением ситуаций, отмеченных в п. в);
б) шесть цифр разместите в 6-байтовом массиве;
в) подсчитайте цифры до десятичной точки или ведущие нули после десятичной точки для определения порядка;

*7. Повторите п. 6, но теперь ваша программа должна воспринимать любую допустимую в языке Фортран вещественную константу типа REAL, включая формат записи типа E.

12.4. ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ В ЭВМ СЕМЕЙСТВА VAX

ДВОИЧНЫЕ ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ

Операции над числами с плавающей точкой в ЭВМ семейства VAX осуществляются во многом так же, как и с числами, описанными в предыдущем разделе. Однако в ЭВМ VAX числа с плавающей точкой кодируются в двоичном, а не десятичном виде. Это означает,

что мантисса выражается как двоичное число и что порядок есть степень числа 2, а не степень числа 10.

Вспомним из гл. 2, что двоичное число 11011 интерпретируется так:

$$11011 = (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) =$$
$$= 16 + 8 + 0 + 2 + 1 = 27.$$

Двоичные дроби обрабатываются аналогично, но с отрицательными порядками. Следовательно, двоичная дробь 0.100011 интерпретируется так:

$$0.100011 = (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (0 \times 2^{-4}) +$$
$$+ (1 \times 2^{-5}) + (1 \times 2^{-6}) =$$
$$= 1/2 + 0 + 0 + 0 + 1/32 + 1/64 = 35/64.$$

Двоичные дроби могут показаться странными, но фактически они используются как обычно. Дюймы, принятые в английской системе измерений, обычно делятся на двоичные доли. Нет ничего непривычного в том, что механик может пользоваться сверлом диаметром 35/64 дюйма. В большинство домашних наборов плотничьего инструмента входят сверла диаметром от 1/64 до 1/4 дюйма.

Следует отметить, что не все дроби можно точно выразить в виде двоичных дробей. И такая ситуация знакома по десятичным дробям. Известно, что одна треть не может быть выражена в десятичном виде. Лучшее, что можно сделать, это написать что-то вроде 0.333333, но такая запись неточна. Можно улучшить ее, добавив тройки, но никакое конечное число троек не сделает число точным. Как и следовало ожидать, одну треть невозможно точно выразить и в двоичном виде. Но более удивительно то, что невозможно точно выразить в двоичном виде и дробь 1/5. В десятичном виде можно записать 1/5 = 0.2, но нельзя сделать так же в двоичном виде. Следующая таблица показывает, как можно приблизительно выразить дробь 1/5 двоичными дробями, но она никогда не будет равна ни одной из них:

1/4 > 1/5 > 1/8
1/4 > 1/5 > 3/16
7/32 > 1/5 > 3/16
13/64 > 1/5 > 3/16
13/64 > 1/5 > 25/128

ПРЕДСТАВЛЕНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Как и в случае десятичных чисел, двоичные числа с плавающей точкой имеют порядок и мантиссу. Посмотрим, как эти части числа размещаются в различных форматах. В ЭВМ семейства VAX применяются четыре формата с плавающей точкой: формат типа F, формат типа D, формат типа G и формат типа H. Требуемый для них объем памяти изменяется от 32-битового длинного слова (формат F) до 128-битового октаслова (формат H). Кроме того, форматы типов D и G, каждый занимающий по 64 бита, различаются числом битов, отведенных для порядка и мантиссы. Чтобы познакомить читателя с форматами чисел с плавающей точкой ЭВМ семейства VAX, воспользуемся форматом F.

Прежде всего необходим знак для мантиссы. Для этого достаточно одного бита. Затем нужен порядок числа. В формате F на ЭВМ VAX могут быть представлены числа в диапазоне $2^{-128} - 2^{+127}$. Это примерно эквивалентно десятичному диапазону $10^{-38} - 10^{+38}$, что вполне достаточно для многих задач науки, техники и других областей. Поскольку порядок, представленный числами в диапазоне –128 – +127, может принимать 256 возможных значений, для его представления необходимо 8 битов.

На ЭВМ семейства VAX для задания знака и порядка используется способ, близкий к описанному в предыдущих разделах способу представления с избытком 50. Но здесь избыток равен 128 в десятичном виде (10000000 в двоичном виде). Следовательно, имеется следующая таблица порядков:

Десятичный порядок	Двоичное представление	Шестнадцатеричное представление
+127	11111111	FF
+1	10000001	81
0	10000000	80
−1	01111111	7F
−128	00000000	00

Числа с плавающей точкой в формате F хранятся в 32-битовом длинном слове. Поскольку для знака нужен один бит, для порядка – 8 битов, то для мантиссы остается 23 бита (32 – 9). Рассмотрим, например, представление десятичного числа .75, или $3/4 \times 2^0$. Поскольку 3/4 – это 1/2 плюс 1/4, мантисса в двоичном представлении есть .110000 . . . 0000, порядок с избытком 128 в двоичном виде есть 10000000, что является истинным нулевым порядком. Таким образом, число .75 может быть представлено в виде двоичного числа с плавающей точкой следующим образом:

Знак	Порядок	Мантисса
0	10000000	11000000000000000000000

Однако 23-битовая мантисса обеспечивает лишь минимально необходимую для большинства вычислений точность. Поэтому в ЭВМ семействах VAX используют очень удачный способ добавления к мантиссе дополнительного бита, что позволяет получать 24-битовую мантиссу. Вспомним, что в нормализованном представлении числа с плавающей точкой ведущая цифра мантиссы никогда не может быть нулем. В двоичной системе имеются только две цифры – 0 и 1. Следовательно, если исключается 0, то ведущей цифрой должна быть 1. Если ведущая цифра всегда 1, то нет нужды указывать ее явно в числе. Поэтому этот бит в записи мантиссы опускается; его называют скрытым битом. В результате число .75 представляется так:

Знак	Порядок	Мантисса
0	10000000	10000000000000000000000
		(Ведущая единица опущена)

Единственная проблема, связанная с предположением, что некоторый бит всегда 1, состоит в невозможности представления нуля. Другой способ оценить эту проблему дает рассмотрение числа $.5 \times 2^{-128}$. Поскольку число положительное, знаковый бит равен 0. В силу того, что истинный порядок есть −128, порядок с избытком 128 будет 00000000. Двоичное представление дроби 1/2 есть .10000000 . . . 0000. Однако после того, как опускается скрытый бит, все оставшиеся 23 бита будут нулевыми. В результате представление числа $.5 \times 2^{-128}$ состоит из 32 двоичных нулей, а это то же самое, что и представление нуля.

Чтобы решить эту проблему, ЭВМ семейства VAX интерпретирует любое число с плавающей точкой с нулевым знаковым битом и порядком с избытком, равным 0, как представление нуля (это означает, что число $.5 \times 2^{-128}$ не может быть представлено в виде 32-битового числа с плавающей точкой в формате F). Другими словами, любое число в формате F вида

Знак	Порядок	Мантисса
0	00000000	?????????????????????????

есть представление нуля в формате F.

Как отмечалось в разд. 12.3, это приводит к вопросу об интерпретации чисел с отрицательным знаком и нулевым порядком, т. е. чисел вида

Знак	Порядок	Мантисса
1	00000000	?????????????????????????

На ЭВМ семейства VAX такие числа обрабатываются, как резервные операнды. Любая попытка использования такого числа в операции над числами в формате F будет порождать ошибку, называемую ошибкой по использованию *резервного операнда с плавающей точкой*. Как отмечалось ранее, это может применяться в языках высокого уровня для обнаружения во время выполнения программы переменных с плавающей точкой, значения которых не определены. Программисты, работающие с языком ассемблера, могут воспользоваться резервными операндами для той же цели.

В качестве последнего примера рассмотрим представление в формате F числа $+35/64 \times 2^9$:

знаковый бит для положительного числа есть 0 (1 для отрицательного числа);

порядок для +9 равен 10001001;

мантисса есть 0.10001100000000000000000, так как 35/64 = 1/2 + 0/4 + 0/8 + 0/16 + + 1/32 + 1/64.

Все это при упаковке в 32 бита дает следующее:

Знак	Порядок	Мантисса
0	10001001	00011000000000000000000
		(Ведущая единица опущена)

Все вместе, как 32-битовая двоичная строка или длинное слово, это выглядит так:

0100 0100 1000 1100 0000 0000 0000 0000

или, в шестнадцатеричном представлении, – как ^X448C0000.

Можно было бы предположить, что теперь все закончено и можно просто поместить каждое число в формате F в длинное слово. Однако и здесь существует еще одна проблема. ЭВМ семейства VAX разрабатывалось так, чтобы обеспечивалась некоторая совместимость с ЭВМ семейства PDP-11, ЭВМ семейства PDP-11 – 16-битовые машины; и 32-битовые, и 64-битовые числа с плавающей точкой в памяти этих ЭВМ размещаются как последовательности 16-битовых слов, причем первой располагается старшая значащая часть числа. Для ЭВМ семейства VAX это неудобно, так как здесь почти повсеместно принято помещать первой младшую значащую часть числа. Числа с плавающей точкой представляют основное исключение из этого правила. Как следствие, когда число в формате F помещается в длинное слово, приходится менять местами первые 16 битов числа с последними 16 битами. Результат выглядит довольно странно:

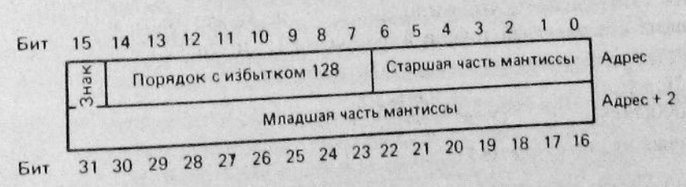
Двоичное представление	Шестнадцатеричное представление
0000 0000 0000 0000 0100 0100 1000 1100	^X0000448C

Обратите внимание, что знаковый бит оказывается в середине длинного слова.

Чтобы избежать этого неудобства, числа с плавающей точкой могут записываться в виде последовательности 16-битовых слов. Приведенное выше число может быть представлено в шестнадцатеричном виде следующим образом:

Содержимое	Адрес
448C	начальный адрес
0000	начальный адрес +2

В этом формате положительный знак, порядок и мантисса выглядят так:



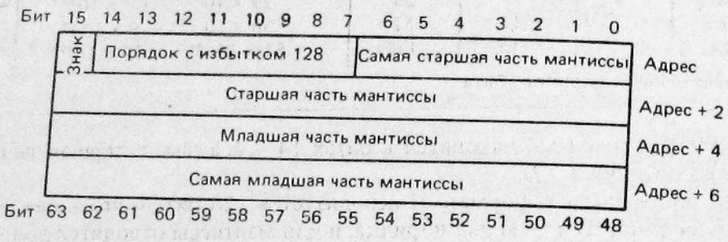
Внутри каждого 16-битового слова биты мантиссы расположены справа налево по возрастанию значимости.

ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ В ФОРМАТАХ ТИПА D, G И H

Числа с плавающей точкой в формате F иногда называют числами с плавающей точкой одинарной точности. Порядок числа, заданный в 8-битовом поле, позволяет представлять

числа приблизительно от $.29 \times 10^{-38}$ до 1.7×10^{38}. Точность представления числа составляет примерно одну часть от 2^{23}, или примерно семь значащих десятичных цифр.

Числа в формате D, которые иногда называют числами с плавающей точкой *двойной точности*, имеют 64-битовую длину и занимают поэтому квадраслово. Формат D идентичен формату F, за исключением того, что поле мантиссы расширяется от 23 до 55 битов (добавляются 32 бита; скрытый бит не считается). Число в формате D выглядит так:



Дополнительные биты поля мантиссы допускают представление чисел с точностью, примерно равной одной части от 2^{55}, или примерно 16 значащих десятичных цифр. Число $+35/64 \times 2^9$ в формате D будет таким:

Содержимое	Адрес
448C	начальный адрес
0000	начальный адрес + 2
0000	начальный адрес + 4
0000	начальный адрес + 6

Как квадраслово это будет ^X000000000000448C.

Представление чисел с плавающей точкой в форматах типов F и D, используемое в ЭВМ семейства VAX, совпадает с представлением таких чисел на ЭВМ семейства PDP-11, выпускавшихся фирмой DEC. Применение одного и того же представления чисел с плавающей точкой помогло сохранить программную совместимость этих двух семейств ЭВМ. Существует, однако, новый стандарт для представления чисел с плавающей точкой, который был одобрен Институтом инженеров по электротехнике и радиоэлектронике (IEEE).

Стандартное представление IEEE чисел с плавающей точкой базируется на той же философии, что и стандарт фирмы DEC. Однако разбиения по числу битов в полях отличаются. Для согласования с ожидаемыми стандартами на языки высокого уровня, ориентированными на стандарт IEEE, в ЭВМ семейства VAX введены два новых формата чисел с плавающей точкой, отвечающие требованиям стандартов IEEE. Они называются форматами типов G и H. Для представления числа в формате G используется 64 бита, как и в формате D. Однако порядок занимает более длинное поле, а мантисса – соответственно поле меньшей длины. Это означает, что формат G несколько менее точный, но охватывает значительно больший диапазон чисел, чем формат D (см. диапазоны и точность представления в табл. 12.1). Числа в формате G имеют следующий вид:

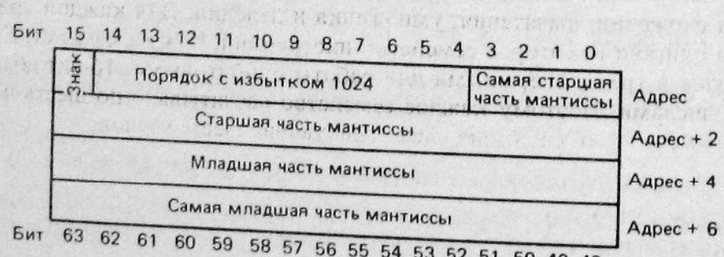


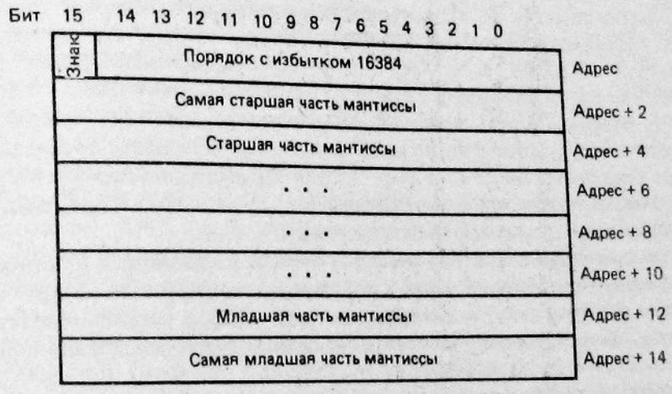
Таблица 12.1. **Представление чисел с плавающей точкой в ЭВМ семейства VAX**

Тип формата	Длина поля, бит				Диапазон значений	Число значащих цифр
	Полная	Знак	Порядок	Мантисса*		
F	32	1	8	23	$.29 \times 10^{-38} - 1.7 \times 10^{38}$	7
D	64	1	8	55	$.29 \times 10^{-38} - 1.7 \times 10^{38}$	16
G	64	1	11	52	$.56 \times 10^{-308} - .9 \times 10^{308}$	15
H	128	1	15	112	$.84 \times 10^{-4932} - .59 \times 10^{4932}$	33

*Не учитывается наличие скрытого бита.

Поле порядка с избытком 1024 находится в битах 14 – 4, а самая старшая часть мантиссы занимает битовые позиции 3 – 0.

Для представления чисел в формате H используется 128 битов, что в два раза больше, чем для форматов типов D и G. И для порядка, и для мантиссы отводятся дополнительные биты, что обеспечивает расширенный диапазон значений и значительно большую точность. Число в формате H имеет такой вид:



Как и форматы типов F и D, форматы типов G и H имеют однобитовое поле знака числа, используют представление порядка с излишком 2^{C-1} (где C – число битов порядка) и скрытый бит в мантиссе. Напомним, что пропуск скрытого бита дает один дополнительный бит точности по сравнению с теми представлениями, где явно указывается каждый бит мантиссы. Особенности четырех типов форматов с плавающей точкой приведены в табл. 12.1.

12.5. ОПЕРАЦИИ С ПЛАВАЮЩЕЙ ТОЧКОЙ В ЭВМ СЕМЕЙСТВА VAX

В гл. 6 был дан полный список инструкций целочисленной арифметики. В основном это были операции сложения, вычитания, умножения и деления. Для каждой арифметической операции было описано некоторое семейство инструкций. Каждое семейство включало инструкции с двумя и тремя операндами для работы с 8-битовыми, 16-битовыми и 32-битовыми целыми числами. Поэтому каждое семейство насчитывало по шесть членов. Например, семейство инструкций сложения имеет следующие шесть членов:

```
ADDB2     ADDB3
ADDW2     ADDW3
ADDL2     ADDL3
```

Когда рассматриваются операции с плавающей точкой, к каждому семейству инструкций добавляется еще по восемь членов. Это инструкции с двумя и тремя операндами для выполнения арифметических операций с числами в формате типов F, D, G и H. В результате получается 32 дополнительные инструкции:

```
ADDF2  ADDF3  SUBF2  SUBF3  MULF2  MULF3  DIVF2  DIVF3
ADDD2  ADDD3  SUBD2  SUBD3  MULD2  MULD3  DIVD2  DIVD3
ADDG2  ADDG3  SUBG2  SUBG3  MULG2  MULG3  DIVG2  DIVG3
ADDH2  ADDH3  SUBH2  SUBH3  MULH2  MULH3  DIVH2  DIVH3
```

Эти инструкции работают так, как описано ранее, выполняя указанные операции над парами 32-битовых, 64-битовых или 128-битовых чисел с плавающей точкой. Буквы F, D, G и H в мнемонике инструкций определяют соответствующие форматы операндов.

Формат инструкций в машинном коде, работающих с операндами типов F и D, совпадает с форматом арифметических инструкций для работы с операндами в формате байта, слова и длинного слова. Единственное отличие заключается в использовании других кодов операций. Однако формат инструкций для операндов типов G и H несколько отличается. Эти инструкции были введены в архитектуру ЭВМ семейства VAX после того, как большинство 8-битовых кодов операций уже были задействованы для других инструкций. В результате этого большинство кодов операций для операций над числами с плавающей точкой в форматах G и H требуют двух 8-битовых байтов. Поэтому, хотя, например, код операции для инструкции ADDF2 есть ^X40, а код для инструкции ADDD2 есть ^X60, код операции для инструкции ADDG2 равен ^X40FD, а для инструкции ADDH2 – ^X60FD. Фактически все коды операции для инструкций с операндами типов G и H содержат в младших восьми битах код ^XFD. Это признак расширения кода. Вспомним, что младшие 8 битов анализируются первыми. Сам по себе код ^XFD не является допустимым кодом операции для какой-либо инструкции, он лишь предписывает процессору извлечь следующий байт, чтобы сформировать расширенный 16-битовый код операции.

Чтобы вместе с инструкциями для работы с числами с плавающей точкой можно было использовать инструкции условных переходов, при выполнении этих инструкций производят установку кодов условий. Если результат отрицательный, устанавливается бит N, если нулевой – бит Z. Бит V устанавливается, если происходит переполнение или потеря значимости.

Как отмечалось ранее, *переполнение числа с плавающей точкой* происходит при попытке вычислить такое число, значение которого слишком велико, чтобы его можно было представить в заданном формате. Для чисел в форматах F и D это значение приблизительно равно 1.7×10^{38}. Многие операционные системы прекращают выполнение программы, если происходит переполнение числа с плавающей точкой. В результате программа может не обнаружить, что установлен бит V при переполнении с плавающей точкой.

Потеря значимости числа с плавающей точкой происходит при попытке вычислить такое число, которое не является нулем, но имеет настолько маленькое значение, что не может быть представлено в данном формате числа с плавающей точкой (для чисел в форматах F и D наименьшее ненулевое значение с плавающей точкой равно приблизительно $.29 \times 10^{-38}$). В большинстве случаев потеря значимости дает нулевой результат. Однако слишком большое число повторений ошибки потери значимости может служить признаком того, что выбранная для данной задачи шкала единиц не обеспечивает необходимую значимость. По этой причине в большинстве операционных систем при обнаружении потери значимости результат операции обнуляется, но при этом также выдается предупреждающее сообщение или производится останов по ошибке, если за время выполнения программы случается больше чем несколько (вероятно, 10) ошибок такого рода. Фактическое значение максимально допустимого числа ошибок может быть параметром, который пользователь может модифицировать при запуске программы.

До сих пор обсуждались биты N, Z и V, а что можно сказать о бите C? Поскольку операции над числами с плавающей точкой не порождают переноса в том смысле, как это делают

при целочисленных операциях, бит C не используется. Поэтому при выполнении операций над числами с плавающей точкой бит C сбрасывается в 0.

Все перечисленные выше арифметические инструкции вызывают прерывание по ошибке, называемой *ошибкой по использованию резервного операнда с плавающей точкой*, если делается попытка использовать число с плавающей точкой со знаковым битом, равным 1, и порядком с избытком, равным 0. Фактически почти все инструкции для работы с числами с плавающей точкой ведут себя таким образом (единственное исключение касается специального семейства инструкций POLY, которое будет описано позднее).

В дополнение к основным арифметическим операциям сложения, вычитания, умножения и деления существуют еще несколько вспомогательных операций. Это операции пересылки, очистки, проверки и сравнения. Они выполняются с помощью следующих инструкций:

```
MOVF    CLRF    TSTF    CMPF
MOVD    CLRD    TSTD    CMPD
MOVG    CLRG    TSTG    CMPG
MOVH    CLRH    TSTH    CMPH
```

Как и ожидалось, эти инструкции выполняют операции пересылки числа с плавающей точкой из одной области памяти в другую, очистки области памяти для числа с плавающей точкой, проверки того, является ли число отрицательным или равно нулю, и сравнения значений двух чисел с плавающей точкой.

Читателю может показаться, что поскольку число в формате F занимает 4 байта и столько же занимает длинное слово, то можно заключить, что MOVF – это та же самая инструкция, что и MOVL, поскольку они обе пересылают четыре байта из одного места в другое. Однако имеются некоторые тонкие отличия.

С одной стороны, разные правила, которые используются для установки кодов условий. Для инструкции MOVL бит Z устанавливается тогда, и только тогда, когда все 32 бита содержат значение 0. Для числа с плавающей точкой бит Z устанавливается, если равны 0 как бит знака числа, так и порядок с избытком (значение мантиссы в расчет не принимается). Бит N устанавливается по-разному, поскольку бит знака числа расположен в разных позициях длинного слова. Знаковым битом целочисленного длинного слова является бит 31 (самый левый бит), тогда как знаковый бит числа с плавающей точкой – это всегда бит 15 (старший бит второго байта). Наконец, в инструкциях весьма по-разному обрабатываются резервные операнды. Для инструкции MOVL резервный операнд не имеет никакого особого значения, но попытка пересылки резервного операнда с помощью инструкции MOVF будет порождать ошибку по использованию резервного операнда с плавающей точкой. Как следствие, инструкции MOVF и MOVL имеют разные коды операций и являются разными инструкциями.

С другой стороны, сброс в 0 содержимого целочисленной ячейки требует тех же самых шагов, что и сброс в нуль содержимого ячейки с числом в формате F. Поэтому CLRF – это та же самая инструкция, что и CLRL. То есть для мнемоники CLRF и CLRL порождается один и тот же код операции. Аналогичные замечания относятся и к другим форматам чисел с плавающей точкой. Например, инструкции MOVQ (переслать квадрослово), MOVD и MOVG – разные инструкции, но CLRQ, CLRD и CLRG имеют один и тот же код операции и в действительности это просто разные имена одной и той же инструкции.

Излишние мнемоники языка ассемблера встроены в ассемблер исключительно с целью облегчить жизнь программистов, работающих с языком ассемблера. Во-первых, программист освобождается от необходимости помнить, какие инструкции различны, какие – нет (MOVF отличается от MOVL, но CLRF – это то же самое, что CLRL). Во-вторых, согласованное использование мнемоник языка ассемблера обеспечивает хорошую документируемость программ.

ДРУГИЕ ИНСТРУКЦИИ ДЛЯ РАБОТЫ С ЧИСЛАМИ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Кроме перечисленных выше инструкций имеется еще несколько дополнительных инструкций, двойники которых использовались в предыдущих главах для работы с целыми числами. Они снова группируются по четыре инструкции – для каждого формата чисел с плавающей точкой. Первый набор инструкций такой:

```
MNEGF    MNEGD    MNEGG    MNEGH
```

Эти инструкции выполняют пересылку отрицательного значения исходного операнда по адресу назначения, как это делают инструкции пересылки целых чисел с изменением знака. Поскольку изменение знака числа с плавающей точкой состоит просто в изменении на обратное значения знакового бита, эти инструкции отличаются от своих целочисленных дубликатов, которые имеют дело с дополнением до двух.

Другое семейство инструкций предназначено для адресных операций. Это такие инструкции:

```
MOVAF    MOVAD    MOVAG    MOVAH
PUSHAF   PUSHAD   PUSHAG   PUSHAH
```

Поскольку адрес 4-байтового блока данных не зависит от того, для чего эти данные предназначаются, нет никакого реального различия между адресацией 32-битового числа с плавающей точкой и длинного слова. Поэтому инструкция MOVAF идентична инструкции MOVAL, а инструкция PUSHAF идентична инструкции PUSHAL. Аналогичные замечания относятся и к остальным шести инструкциям.

КОНСТАНТЫ И НЕПОСРЕДСТВЕННЫЕ ЗНАЧЕНИЯ

Инструкции для работы с числами с плавающей точкой могут использовать непосредственные операнды в существенной степени так же, как это делается в инструкциях для работы с целыми числами. Так, например, по инструкции ADDF2 #3.1415927, X выполняется сложение приблизительного значения числа π с числом с плавающей точкой X, имеющим формат F. Синтаксис написания констант с плавающей точкой, хотя и несколько менее строгий, в существенной степени совпадает с тем, который используется в Фортране. Вместо описания здесь точных правил достаточно будет сказать, что любое допустимое представление вещественных констант в Фортране так же допустимо и в языке ассемблера ЭВМ семейства VAX. Сюда входит, конечно, и экспоненциальный формат E. (Более подробную информацию см. в руководстве "VAX-11 Macro Language Reference Manual".)

В инструкциях для работы с числами с плавающей точкой могут использоваться литеральные операнды. Как и в случае целочисленных инструкций, если первые два бита байта спецификатора операнда нулевые, то остальные шесть битов используются для ограниченных значений непосредственного операнда. Для целых чисел ограничение состоит в том, что число должно попадать в диапазон 0 – 63. Для операций над числами с плавающей точкой ограничение заключается в том, что число с плавающей точкой должно быть представимо в специальном 6-битовом формате. Очевидно, что такое число, как 3.1415927, в шести битах представлено быть не может. Поэтому в инструкции ADDF2 #3.1415927, X потребуется полный, заданный в формате длинного слова непосредственный операнд. Однако более простые константы с плавающей точкой, такие как 1.0 (которая встречается очень часто) могут быть представлены в 6-битовом формате. Следовательно, инструкция ADDF2 #1.0, X после ее ассемблирования будет занимать на четыре байта меньше.

На рис. 12.4 показаны 64 числа с плавающей точкой, которые могут быть представлены как литералы. 6-битовый литерал с плавающей точкой состоит из 3-битового порядка и 3-битовой мантиссы со скрытым битом. Представление отрицательных значений и отрицательных порядков не поддерживается. Поэтому диапазон дробей простирается от 1000 до 1111 (в двоичном виде). (Заметим, что ведущая единица мантиссы опущена так, что для

Биты порядка	Биты мантиссы (ведущий бит скрыт)							
	000	001	010	011	100	101	110	111
000	$\frac{1}{2}$	$\frac{9}{16}$	$\frac{5}{8}$	$\frac{11}{16}$	$\frac{3}{4}$	$\frac{13}{16}$	$\frac{7}{8}$	$\frac{15}{16}$
001	1	$1\frac{1}{8}$	$1\frac{1}{4}$	$1\frac{3}{8}$	$1\frac{1}{2}$	$1\frac{5}{8}$	$1\frac{3}{4}$	$1\frac{7}{8}$
010	2	$2\frac{1}{4}$	$2\frac{1}{2}$	$2\frac{3}{4}$	3	$3\frac{1}{4}$	$3\frac{1}{2}$	$3\frac{3}{4}$
011	4	$4\frac{1}{2}$	5	$5\frac{1}{2}$	6	$6\frac{1}{2}$	7	$7\frac{1}{2}$
100	8	9	10	11	12	13	14	15
101	16	18	20	22	24	26	28	30
110	32	36	40	44	48	52	56	60
111	64	72	80	88	96	104	112	120

Рис. 12.4. Литералы с плавающей точкой

мантиссы требуется только 3 бита памяти.) Поскольку подразумевается, что точка в позиционном представлении числа находится слева, эти дроби позволяют представить значения в диапазоне 1/2 – 15/16 с шагом по 1/16. Значения порядка числа находятся в диапазоне 0 – 7, так что числа могут быть представлены с ограниченной точностью от $1/2 \times 2^0$, или 1/2, до $15/16 \times 2^7$, или 120.

В дополнение к непосредственным константам в инструкциях иногда полезно иметь константы, размещенные в области данных программы. Это, в частности, полезно для создания массивов констант. В случае целых чисел применяются директивы ассемблера .BYTE, .WORD, .LONG и .OUAD. Для вещественных констант применяются директивы .F_FLOATING, .D_FLOATING, .G_FLOATING и .H_FLOATING[1]. Эти директивы могут иметь переменное число аргументов, синтаксис констант совпадает с принятым в Фортране. Например, по следующей директиве:

```
.F_FLOATING    1,0,2,0,3,0
```

будет сформирован массив длинных слов:

```
^X00004080
^X00004100
^X00004140
```

ИНСТРУКЦИИ ПРЕОБРАЗОВАНИЯ

В гл. 6 рассматривались инструкции пересылки байтов в слова, слов в длинные слова и т. п. Теперь это множество инструкций будет расширено, будет разрешено не только преобразование операндов с плавающей точкой одного типа (таких как одинарной точности – формат F) в операнды другого типа (таких как двойной точности – формат D), но также всех комбинаций преобразований целых чисел в числа с плавающей точкой и наоборот. На рис. 12.5 показано семейство инструкций преобразования.

Как и следовало ожидать, при пересылке вещественного числа в целочисленное поле дробная часть числа теряется. Для инструкций на рис. 12.5 это делается с помощью усечения. Также возможно, что вещественное число может оказаться слишком большим для размещения в поле байта, слова или даже длинного слова при его преобразовании в целое. В таком случае может произойти переполнение. Для индикации переполнения используется бит V, как это делается в случае инструкций, преобразующих большие форматы целых чисел в меньшие, таких как CVTWB. Обратите внимание, что при употреблении инструкции

[1] Директива .FLOAT дает тот же эффект, что и директива .F_FLOATING, а директива .DOUBLE – тот же эффект, что директива .D_FLOATING.

Тип операнда источника	Тип операнда получателя						
	Байт	Слово	Длинное слово	Формат F	Формат D	Формат G	Формат H
Байт		CVTBW	CVTBL	CVTBF	CVTBD	CVTBG	CVTBH
Слово	CVTWB		CVTWL	CVTWF	CVTWD	CVTWG	CVTWH
Длинное слово	CVTLB	CVTLW		CVTLF	CVTLD	CVTLG	CVTLH
Формат F	CVTFB	CVTFW	CVTFL		CVTFD	CVTFG	CVTFH
Формат D	CVTDB	CVTDW	CVTDL	CVTDF			CVTDH
Формат G	CVTGB	CVTGW	CVTGL	CVTGF			CVTGH
Формат Н	CVTHB	CVTHW	CVTHL	CVTHF	CVTHD	CVTHG	

Рис. 12.5. Семейство инструкций преобразования

CVTLF может происходить потеря точности, поскольку в длинном слове значение числа занимает 31 бит, но только 24 бита занимает мантисса числа в формате F.

Заметим, что отсутствуют инструкции для преобразования чисел между форматами D и G. Чтобы преобразовать формат D в формат G, необходимо преобразовать 64-битовое число в формате D в 128-битовое число в формате H, а затем преобразовать число в формате H в 64-битовое число в формате G. Аналогичные замечания относятся и к преобразованию из формата G в формат D. Преобразования могут выполняться с помощью следующих макроинструкций:

```
.MACRO   CVTDG   D_FLT,G_FLT        .MACRO   CVTGD   G_FLT,D_FLT
CVTDH    D_FLT,-(SP)               CVTGH    G_FLT,-(SP)
CVTHG    (SP)+,G_FLT               CVTHD    (SP)+,D_FLT
.ENDM    CVTDG                     .ENDM    CVTGD
```

Как отмечалось ранее, инструкции на рис. 12.5, осуществляющие преобразование формата с плавающей точкой в целочисленный формат, делают это с усечением дробной части. В ЭВМ семейства VAX имеются четыре дополнительные инструкции, которые при преобразовании округляют дробную часть до ближайшего целого числа. Это такие инструкции:

```
CVTRFL   CVTRDL   CVTRGL   CVTRHL
```

Например, инструкция CVTRFL (преобразовать число в формате F в целое в формате длинного слова с округлением) будет округлять число с плавающей точкой до ближайшего целого числа и помещать результат в длинное слово.

12.6. СЛОЖНЫЕ ИНСТРУКЦИИ ДЛЯ РАБОТЫ С ЧИСЛАМИ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Программы, выполняющие большой объем вычислений с числовыми дробями, зачастую называют "перемалывателями" чисел. Однако даже такие программы много своего времени (если не бóльшую часть) тратят на выполнение логических и целочисленных операций. Сюда входят адресные вычисления для массивов, подсчет числа итераций цикла и определение того, куда должен быть осуществлен переход. Чем ниже уровень, на котором выполняются эти логические операции, тем быстрее будет работать программа. Поэтому, например, циклы, требуемые для выполнения умножения, деления и операций с плавающей точкой, на быстродействующих ЭВМ, уже давно реализуются аппаратно. Современной тенденцией является вычисление полиномов. Простой полином может быть определен так:

$$A_n X^n + A_{n-1} X^{n-1} + \ldots + A_2 X^2 + A_1 X^1 + A_0,$$

где $A_0, \ldots, A_n$ образуют массив постоянных коэффициентов, а X – это переменная. Поскольку n – наивысший показатель степени, это говорит о том, что полином имеет порядок n. Полиномы очень важны для численных вычислений, поскольку большинство функций, таких как SIN, COS и EXP, возвращающих вещественное значение, могут быть аппроксимированы полиномами. Например, функцию e^X можно аппроксимировать полиномом

$$1 + X + X^2/2! + X^3/3! + X^4/4! + \ldots + X^n/n!$$

Вычисляя такой полином достаточно высокого порядка, можно достичь любой желаемой точности.

Для вычисления полиномов предназначаются инструкции POLVF, POLYD, POLYG и POLYH. Эти инструкции имеют три операнда: переменная X в формате числа с плавающей точкой; целочисленное значение порядка полинома; адрес массива, который содержит коэффициенты, имеющие формат числа с плавающей точкой. Начиная с коэффициента члена наивысшего порядка для записи результата вычисления операнда нет. Вместо этого для результата всегда используются регистры общего назначения R0, R1, R2,.... В этих инструкциях регистры общего назначения также используются для сохранения промежуточных значений во время вычисления. Поэтому после завершения выполнения инструкций содержимое некоторых из этих регистров будет перезаписано. Инструкция POLYF помещает результат в регистр R0, но также перезаписывает регистры R1, R2 и R3. Инструкции POLYD и POLYG помещают старшие 32 бита результата в регистр R0, а младшие 32 бита – в регистр R1. Они также изменяют содержимое регистров от R2 до R5. Инструкция POLYH помещает 128-битовый результат в регистры R0 – R3, причем старшие биты заносятся в регистр R0, а самые младшие – в регистр R3. Эта инструкция также изменяет содержимое регистров R4 и R5.

Заметим, что использование регистров R0 и R1 для запоминания результата двойной точности является обычным способом, применяемым для всех инструкций типа D или G, и, когда один или несколько операндов находятся в регистрах общего назначения, указанный в инструкции регистр будет содержать старшие 32 бита числа, а регистр со следующим номером будет содержать младшие 32 бита. Таким образом, за инструкцией POLYD может следовать инструкция MOVD R0, ANS или даже инструкция ADDD2 R0, TOTAL.

В качестве примера работы этих инструкций рассмотрим приведенный выше полином для аппроксимации функций e^X. И хотя точность не будет очень высокой, если только значение X не малое, будем считать, что для аппроксимации вполне достаточно иметь полином пятого порядка. Тогда нам нужна таблица из шести коэффициентов:

```
CTAB:   .F_FLOATING   0.8333333E-2   ;1/5!
        .F_FLOATING   0.04166667     ;1/4!
        .F_FLOATING   0.166667       ;1/3!
        .F_FLOATING   0.5            ;1/2!
        .F_FLOATING   1.0            ;1/1!
        .F_FLOATING   1.0            ;1/0!
```

Затем по инструкции POLYF X, #5, CTAB можно будет вычислить функцию, помещая результат в регистр R0.

Семейство инструкций EMOD (расширенное умножение и отделение целой части) используется для вычисления значения периодических функций, таких как тригонометрические функции (как поясняется ниже, буквы MOD происходят от слова модуль). Рассмотрим синус угла, заданного в градусах. По определению функции синуса имеем

$$SIN(X) = SIN(X + 360) = SIN(X + 720) = SIN(X + N \times 360),$$

где N – любое целое число. При вычислении синуса угла числа, кратные 360, являются избыточными. Для вычисления значений периодических функций в программах обычно осуществляется приведение к заданному интервалу, которое позволяет уменьшать значение аргумента функции так, чтобы оно попадало в некоторый ограниченный интервал. Например, для функции синуса приведение к заданному интервалу может быть выполнено делением значения аргумента на 360 (а это то же самое, что умножение на 1/360) с последую-

щим отбрасыванием целой части результата, после чего вычисление синуса производится для дробной части результата (в данном примере число 360 является *модулем*, по которому вычисляется остаток, что и нашло отражение в мнемонике инструкции).

Однако при сокращении интервала может происходить потеря значащих цифр. Например, если операнды являются числами в формате F и выполняется сокращение для SIN(360001), то 360001/360 равно 1000 плюс 1/360. Первоначально точность числа в формате F определяется примерно 23 битами. Отбрасывание целой части – 1000 отнимает примерно 10 битов точности. Это означает, что точность дробной части определяется примерно как 23 – 10 = 13 битов.

Семейство инструкций EMOD позволяет разрешить проблему. Рассмотрим, например, следующую инструкцию этого семейства:

```
EMODF    MULT, MULT_EXT, ARG, INT_PART, FRAC_PART
```

Первый, третий и пятый операнды (MULT, ARG и FRACT_PART) – это числа в формате F. Второй операнд, MULT_EXT, – это байт, используемый для 8 битов дроби дополнительно к 24 битам дроби в операнде MULT. В вышеприведенном примере программист для числа 1/360 мог задать 32 значащие цифры (двоичные). Старшие 24 бита становятся мантиссой числа MULT в формате F, а младшие 8 бит попадают в байт MULT_EXT.

При выполнении инструкции EMODF аргумент ARG в формате F умножается на расширенный множитель (MULT, расширенный байтом MULT_EXT). Целая часть результата помещается в целое слово INT_PART, а приведенный к заданному интервалу результат помещается по адресу FRACT_PART, как число формата F (после чего можно воспользоваться инструкцией POLYF, использующей FRACT_PART, для вычисления синуса). Инструкции EMOD, EMODG и EMODH работают аналогично. Однако длина операндов (включая MULT_EXT и INT_PART) отличается.

УПРАЖНЕНИЯ 12.2

1. Покажите, как следующие цифры могут быть представлены в 32-битовом формате типа F. Ответы представьте в шестнадцатеричном виде:

а) 1.0 б) –10.0 в) 100.0
г) 0.5 д) 0.1 е) –0.01

2. Для заданных ниже 32-битовых чисел в формате F покажите, используя форму записи, принятую в Фортране, какими будут их значения в десятичной системе.

а. ^X00004080 б. ^X00004000
в. ^X00003F80 г. ^X00006080
д. ^X0000C080 е. ^XCCCC3F4C

3. Покажите, как числа из пп. 1 и 2 могут быть представлены в формате типов D, G и H. Свои ответы дайте в шестнадцатеричном виде.

4. Как обсуждалось в п. 5 из упр. 12.1, корень квадратный из числа N может быть аппроксимирован формулой

$0.5 \times (X + N/X)$,

где X – начальное приближение заданного значения квадратного корня из N. Если начальное значение X оказывается достаточно близким к корню квадратному из N, то формула даст значение, намного более близкое к правильному. В результате этого, если алгоритм обеспечивает удачное определение первоначального значения, несколько итераций этой формулы позволяют получить результат с точностью до 15 – 30 знаков (десятичных). Один простой алгоритм для задания начального приближенного значения работает следующим образом:

а) представление числа с плавающей точкой таково:
$f \times 2^C$, где $1/2 \leqslant f < 1$;

б) если из C вычитается единица, чтобы сделать его четным, то формула становится такой:
$\sqrt{f} \times 2^{C'}$, где $1/2 \leqslant f < 2$;

в) теперь квадратный корень из этого будет
$\sqrt{f'} \times 2^{C/2}$;

г) диапазон значений для корня из $\sqrt{f'}$ будет примерно таким:
$0.707 \leqslant \sqrt{f'} \leqslant 1.41$;

д) в результате удачным начальным значением для $\sqrt{f'}$ может быть 1, и это будет отличаться от точного значения не более чем на 41 %;

е) поскольку мантисса числа с плавающей точкой не может быть единицей, используйте наиболее близкое к ней возможное значение; такое значение представляют двоичные единицы во всех разрядах мантиссы.

Напишите подпрограмму, которую можно вызывать из программы на Фортране и которая в качестве входного аргумента имеет одну переменную в формате D, а в качестве выходного – массив из десяти переменных в формате D. Эта подпрограмма будет применять описанный выше алгоритм для получения первоначального значения корня квадратного из входной переменной. Затем она выполняет девять итераций по формуле $0.5 \times (X + N/X)$ и помещает каждое полученное значение в выходной массив. Напишите программу на Фортране, которая вызывает подпрограмму с различными входными значениями. Программа на Фортране должна для сравнения распечатывать эти результаты вместе с результатами, полученными с помощью функции DSQRT из библиотеки Фортрана. (Подсказка: используйте маскирование и операции сдвига для деления показателя степени на два при определении первоначального значения.)

5. Косинус X можно вычислить с помощью бесконечного ряда

$$\cos(x) = 1 - X^2/2! + X^4/4! - X^6/6! + X^8/8! - \ldots$$

Этот ряд может использоваться в ЭВМ для вычисления функции косинуса, если берется достаточное число членов ряда, чтобы гарантировать необходимую точность. Т. к. от члена к члену ряда знак меняется, то как только члены начнут уменьшаться по абсолютному значению, ошибка будет меньше первого отбрасываемого члена:

а) сколько членов ряда необходимо для достижения точности формата D или G, если значение X находится в интервале $-\pi \leqslant X \leqslant \pi$?

б) напишите подпрограмму или функцию, которую можно вызывать из программы на Фортране и которая вычисляет косинус, используя приведенный выше полином со значениями, заданными в формате D или G. Сравните свои результаты с теми, которые дает библиотечная функция Фортрана DCOS.

Коэффициенты не должны храниться в массиве, а должны вычисляться в процессе вычисления полинома. В этом упражнении нельзя пользоваться инструкцией POLYD или POLYG.

6. Повторите п. 5, за исключением того, что ваша программа должна содержать массив постоянных коэффициентов и использовать инструкцию POLYD или POLYG. Обратите внимание, что поскольку ряд косинуса не содержит нечетных степеней, то подразумевается, что все другие коэффициенты нулевые. Это означает, что половина умножений и сложений выполняется с нулем. Можно ли что-либо сделать, чтобы избавиться от этих неэффективных операций?

7. Повторите п. 5 и 6, но ограничьте интервал значений X до $-\pi/2 \leqslant X \leqslant \pi/2$. Сколько теперь нужно членов ряда для достижения точности формата D или G? Существует ли какой-либо простой способ подстройки значений входного аргумента с помощью тригонометрических тождественных преобразований так, чтобы можно было вычислить косинус любого угла, даже если интервал ограничен? Если да, то используйте его в вашей программе.

8. Предположим, что не существует инструкции CVTFL. Напишите программу, которая использует сдвиги и маскирование для преобразования числа из формата F в формат длинного слова.

9. Повторите п. 8, но для инструкции CVTLF.

10. Повторите п. 8, но для инструкции CVTDG.

ГЛАВА 13. ИНСТРУКЦИИ ОБРАБОТКИ СИМВОЛЬНЫХ, ДЕСЯТИЧНЫХ И ДРУГИХ ДАННЫХ

13.1. ВВЕДЕНИЕ

В этой главе описаны некоторые из более сложных инструкций, которые являются составной частью архитектуры ЭВМ семейства VAX. В архитектуру входят, например, инструкции обработки символьных строк. С помощью этих инструкций можно выполнять такие операции, как пересылка символьной строки, сравнение двух символьных строк или поиск в символьной строке конкретной подстроки. Многие из этих методов были описаны в гл. 8, однако инструкции, описываемые в настоящей главе, позволяют выполнять подобные функции с помощью одной инструкции.

Другое множество инструкций имеет дело с десятичными числами и десятичной арифметикой. Для представления десятичных чисел в архитектуру ЭВМ семейства VAX включено представление чисел, называемое *упакованным десятичным представлением*. Число в таком представлении хранится в виде последовательности байтов, в которой каждый байт представляет две десятичные цифры. Имеются инструкции для выполнения арифметических операций над упакованными десятичными числами, которые особенно важны для обработки экономической информации.

Еще одно множество инструкций поддерживает форматирование числовой информации для ввода и вывода. Числовая строка – это символьная строка, которая содержит числовую информацию в одном из нескольких форматов, обеспечиваемых архитектурой ЭВМ семейства VAX. Имеются инструкции для преобразования упакованных десятичных чисел в числовые строки и обратно. Гибкий способ преобразования десятичных чисел в символьные строки обеспечивает инструкция EDIT. Наконец, архитектура ЭВМ VAX включает инструкции обработки битовых полей и двунаправленных списков.

Многие описанные в этой главе инструкции были включены в архитектуру ЭВМ семейства VAX для поддержки языков высокого уровня. Однако большинство этих инструкций выполняют такие функции, которые вообще полезны в любой среде программирования. Их использование в программах на языке ассемблера может обеспечить сохранение значительного количества процессорного времени и пространства памяти.

13.2. ИНСТРУКЦИИ СЕМЕЙСТВА MOVC

ИНСТРУКЦИЯ MOVC3

Как было показано в гл. 8, символьная строка – это последовательность байтов памяти, которая определяется двумя атрибутами: адресом первого байта строки и длиной строки. На рис. 13.1 приведен фрагмент программы для копирования исходной символьной строки в целевую символьную строку. Исходная строка начинается с адреса SOURCE, целевая строка начинается с адреса DEST, а длину строки определяет символическое имя LENGTH. Поскольку для загрузки длины строки в регистр R0 используется инструкция MOVZWL, длина – это целое число без знака в формате слова, а исходная и целевая строки могут иметь длину от 0 до 65535 байтов. После завершения цикла регистр R0 будет содержать 0, в регистре R1 будет находиться адрес байта, следующего за последним байтом исходной строки, а в регистре R3 – адрес байта, следующего за последним байтом целевой строки.

В ЭВМ семейства VAX имеется инструкция MOVC3, которая в существенной степени дает тот же эффект, что и фрагмент программы на рис. 13.1. Например, инструкция


```
MOVC3   #LENGTH, SOURCE, DEST
```

скопирует #LENGTH байтов, начиная с адреса SOURCE, по адресу DEST. Кроме того, инструкция MOVC3 использует регистры R0, R1 и R3 так же, как и фрагмент программы на рис. 13.1. После выполнения инструкции MOVC3 регистры R0 – R5 будут содержать следующее:

R0 нуль,
R1 адрес первого байта, следующего за исходной строкой,
R2 нуль,
R3 адрес первого байта, следующего за целевой строкой,
R4 нуль,
R5 нуль.

Обратите внимание, что инструкция MOVC3 модифицирует содержимое регистров R0 – R5, даже если эти регистры не указаны в качестве операндов инструкции. Большинство описываемых в этой главе инструкций неявно используют некоторые или все регистры от R0 до R5 (инструкции семейства POLY, описанные в гл. 12, также неявно используют регистры R0 – R5). Поскольку эти инструкции уничтожают предыдущее содержимое регистров R0 – R5, программист при работе с такими регистрами должен проявлять особое внимание.

Для описания инструкций и используемых ими регистров будем пользоваться следующей нотацией:

```
MOVC3       len,        srcadr,     dstadr
            R0=0        R1=a        R3=a    (R2=R4=R5=0)

a = адрес байта, следующего за последним байтом строки
```

Имя len обозначает операнд, задающий длину строки, имена srcadr и dstadr представляют адреса исходной и целевой строк. Символы R0 = 0 под именем len показывают, что значение длины строки помещается в регистр R0 и что после завершения инструкции этот регистр содержит нуль. Символы (R2 = R4 = R5 = 0) показывают, что инструкция очищает эти регистры, даже если они прямо и не ассоциируются с каким-либо операндом.

```
LENGTH= ЛЮБОЕ ЧИСЛО ОТ 0 ДО 65535
SOURCE: .BLKB   LENGTH
DEST:   .BLKB   LENGTH
        . . .
        MOVAB   SOURCE,R1       ;УКАЗАТЕЛЬ НА СТРОКУ SOURCE
        MOVAB   DEST,R3         ;УКАЗАТЕЛЬ НА СТРОКУ DEST
        MOVZWL  #LENGTH,R0      ;ДЛИНА - ЦЕЛОЕ БЕЗ ЗНАКА
        BEQL    20$:            ;ДЛЯ СТРОКИ С НУЛЕВОЙ ДЛИНОЙ
10$:    MOVB    (R1)+,(R3)+
        SOBGTR  R0,10$
20$:    . . .
```

Рис. 13.1. Копирование символьной строки

ИНСТРУКЦИЯ MOVC5

Имеется инструкция под названием MOVC5, подобная инструкции MOVC3, которая предназначена для пересылки исходной строки в целевую строку в случае, когда строки имеют разную длину. Формат этой инструкции такой:

```
MOVC5      srclen, srcadr, fill, dstlen, dstadr
           R0=a    R1=б          R2=0    R3=б                (R4=R5=0)

а - число байтов оставшихся непересланными, если длина
    строки-источника (srclen) меньше длины строки-получателя (dstlen);
    в противном случае 0
б - адрес байта, следующего за последним байтом строки
```

Первый и четвертый операнды (srclen и dstlen) – это 16-битовые слова, содержащие длину исходной и целевой строк. Если srclen больше, чем dstlen, то из исходной строки в целевую пересылаются только первые dstlen байтов. Если srclen меньше, чем dstlen, то из исходной строки в целевую пересылается srclen байтов. Оставшиеся dstlen-srclen байтов целевой строки заполняются символом-заполнителем, который определен третьим операндом.

Например, положим, что исходная и целевая строки определены так:

```
SOURCE:   .ASCII     /КОРОТКАЯ СТРОКА/
SOURCE_END:
SRC_LEN= SOURCE_END-SOURCE
DST_LEN= 25
DEST:     .BLKB      DST_LEN
```

Тогда выполнение инструкции

```
MOVC5    #SRC_LEN, SOURCE, #^A"*", #DST_LEN, DEST
```

установит в целевой строке следующее:

```
КОРОТКАЯ СТРОКА**********
```

Третий операнд, #^A"*", иллюстрирует, как в качестве операнда можно использовать строку в коде ASCII. Это операнд эквивалентен операнду #42 или #^X2A, поскольку 42 или ^X2A – это код ASCII символа *. Так как исходная строка имеет длину 14 байтов, а целевая строка – 25 байтов, последние 11 байтов целевой строки будут содержать символы-заполнители, в данном случае*.

В зависимости от соотношения длины исходной и целевой строк инструкция MOVC5 устанавливает коды условий. В частности,

- N устанавливается, если длина исходной строки меньше, чем длина целевой строки, как число со знаком;
- Z устанавливается при равной длине строк;
- V сбрасывается;
- C устанавливается, если длина исходной строки меньше, чем длина целевой строки, как число без знака.

За исключением бита V, коды условий устанавливаются так, как если бы выполнялась инструкция

```
CMPW     srclen, dstlen
```

вслед за инструкцией MOVC5. Поэтому после инструкции MOVC5 можно воспользоваться инструкцией условного перехода по сравнению длин двух строк как чисел без знака. Для обеспечения согласованности с инструкцией MOVC5 инструкция MOVC3 устанавливает коды условий так, чтобы отразить сравнение двух строк равной длины. Таким образом, инструкция MOVC3 сбрасывает биты N, V и C и устанавливает бит Z.

Инструкция MOVC5 дает предпочтительный способ заполнения блока памяти символами-заполнителями. Это можно сделать, если задать нулевую длину исходной строки. Например, положим, что следующая область памяти должна быть инициализирована символами пробела (^X20):

```
BUFLEN = любое число от 0 до 65535
BUFFER:  .BLKB     BUFLEN
```

Пробелы могут быть пересланы в буфер с помощью инструкции

```
MOVC5     #0, любой_адрес, #^X20, #BUFFLEN, BUFFER
```

Второй операнд,"любой адрес", является адресом начала исходной символьной строки. Поскольку длина исходной строки нулевая, этот адрес никогда не используется, и, следовательно, допустим любой адрес, включая нуль. Одно решение состоит в применении символа значения счетчика адреса (.) в качестве второго операнда. Но лучшее решение заключается в том, чтобы в качестве второго операнда использовать указатель стека (SP), так как при этом требуется только один байт.

Инструкции MOVC3 и MOVC5 содержат операнды трех типов: 32-битовые адреса, 16-битовые длины строк как числа без знака и в случае инструкции MOVC5 – 8-битовый символ-заполнитель. Для операндов длины и символа-заполнителя могут использоваться любые режимы адресации, описанные в гл. 7. Но адресные операнды требуют таких режимов адресации, которые образуют 32-битовый адрес. Таким образом, регистровая, литеральная и непосредственная адресации не могут быть использованы с адресными операндами.

В обеих инструкциях,MOVC3 и MOVC5, исходная и целевая строки могут частично накладываться друг на друга. Рассмотрим, например, следующее:

```
STRING:  .ASCII    /ABCDEF/
         . . .

         MOVC3     #5, STRING, STRING+1
```

Если бы инструкции семейства MOVC выполняли пересылку по одному байту, как показано на рис. 13.1, то байт STRING был бы переслан в байт STRING+1, новое значение байта STRING+1 было переслано в байт STRING+2, и т. д. В результате символ, размещенный по адресу STRING, был бы распространен на остальные пять байтов и после выполнения строка примет вид "AAAAAA"[1].

Однако реализация инструкций семейства MOVC на ЭВМ семейства VAX такова, что исходная строка как бы сначала копируется во временную память, а затем пересылается в целевую строку. В результате после выполнения инструкции MOVC3 строка будет иметь вид "AABCDE".

13.3. ИНСТРУКЦИИ ОБРАБОТКИ СИМВОЛЬНЫХ СТРОК

Инструкции обработки символьных строк, такие как MOVC3 и MOVC5, могут эффективно реализовать операции над символьными строками в языках высокого уровня. Однако слово "символ" в названии инструкции вводит в заблуждение, т. к. применение этих инструкций не ограничивается каким-либо конкретным символьным кодом или даже символами любого рода. Это инструкции общего назначения для обработки байтовых массивов.

Предположим, например, что программа содержит два следующих массива длинных слов:

---

[1] В ЭВМ серии IBM 360 и последующих моделях этого ряда инструкцией, аналогичной инструкции MOVC3, является MVC (переслать символы). Она реализована так, что осуществляет пересылку по одному символу и используется для распространения байта по строке.

```
ELEMENTS= любое число от 0 до 16383
ARRAY_ONE:    .BLKL  ELEMENTS
ARRAY_TWO:    .BLKL  ELEMENTS
```

Инструкция

```
MOVC5     #0,(SP),#0,#4*ELEMENTS,ARRAY_ONE
```

может быть использована для очистки всех длинных слов массива ARRAY_ONE (поскольку первый операнд указывает, что длина первой символьной строки нулевая, адрес первой строки не имеет значения; в результате для второго операнда применяется регистровая косвенная адресация с использованием указателя стека (SP)).

Длинные слова массива ARRAY_ONE могут быть скопированы в массив ARRAY_TWO по инструкции

```
MOVC3     #<4*ELEMENTS>,ARRAY_ONE,ARRAY_TWO
```

(Инструкция MOVC3 рекомендуется для пересылки блока данных из одного места в памяти в другое.) Аналогично инструкция

```
MOVC3     #<<4*ELEMENTS>-4>,ARRAY_ONE+4,ARRAY_ONE
```

устанавливает элементы массива ARRAY_ONE [I] равными элементам ARRAY_ONE [I+1] при изменении I от 1 до ELEMENTS-1. (Операнд #⟨⟨4*ELEMENTS⟩ −4⟩ эквивалентен операнду #4*ELEMENTS−4. Угловые скобки, которые обычно применяются для изменения порядка вычисления выражения, включены здесь для улучшения читабельности.)

Все инструкции обработки символьных строк используют некоторые регистры от R0 до R5. Как будет показано, инструкции обработки символьных строк могут иметь до трех строк в качестве операндов, что требует указания адресов этих строк и их длины. Когда начинается выполнение инструкции обработки символьных строк, процессор производит выборку операндов и загружает длину строк в регистры от R0 до R5 (поскольку выборка операндов осуществляется до использования каких-либо регистров общего назначения, сами операнды могут находиться в этих регистрах, включая регистры R0 – R5). В регистры R0 – R5 загружается следующая информация:

R0 – длина первой строки;
R1 – начальный адрес первой строки;
R2 – длина второй строки (если требуется);
R3 – начальный адрес второй строки (если требуется);
R4 – длина третьей строки (если требуется);
R5 – начальный адрес третьей строки (если требуется).

В зависимости от числа строк инструкции обработки символьных строк могут использовать регистры R0 и R1, регистры R0 – R3 или регистры R0 – R5.

В начале выполнения инструкции происходит загрузка значений в регистры R0 и R1, R0 – R3 или R0 – R5, что стирает их предыдущее содержимое. Во время выполнения такой инструкции процессор увеличивает адреса в регистрах с нечетными номерами и уменьшает длину строк в регистрах с четными номерами в соответствии с тем, как это предусмотрено для конкретной ситуации. После завершения выполнения инструкции содержимое регистров может использоваться для дальнейшей обработки.

Есть еще одна причина, по которой инструкции обработки символьных строк задействуют регистры общего назначения. Как поясняется в гл. 14, выполнение программы может быть прервано, чтобы процессор смог ответить на событие, имеющее более высокий приоритет, чем текущая программа. Большинство инструкций ЭВМ семейства VAX имеют короткое время выполнения, и процессор обычно, прежде чем начать обработку прерывания, ожидает завершения выполнения текущей инструкции. Однако некоторые инструкции, включая инструкции обработки символьных строк, могут иметь продолжительное время

выполнения (даже на такой вычислительной системе, как VAX, пересылка 65 535 байтов может занять значительную долю секунды). Чтобы дать возможность ЭВМ семейства VAX быстро отвечать на внешние события, инструкции с продолжительным временем выполнения разработаны так, что они могут быть прерваны, не дожидаясь их завершения. Поскольку инструкции обработки символьных строк в качестве временной памяти используют регистры R0 – R5, обработка прерываний происходит быстро и эффективно. Когда такие инструкции прерываются в середине выполнения, большая часть информации, необходимой для возобновления выполнения инструкции, находится в регистрах общего назначения.

13.4. ИНСТРУКЦИИ СЕМЕЙСТВА CMPC

Семейство инструкций CMPC (сравнить символьные строки) состоит из двух инструкций, CMPC3 и CMPC5. Подобно всем инструкциям сравнения, инструкции семейства CMPC просто устанавливают коды условий. Инструкции CMPC3 и CMPC5 имеют тот же формат, что и инструкции MOVC3 и MOVC5. Например, инструкция

```
CMPC3     #10,ALPHA,BETA
```

выполняет последовательно десять сравнений байтов:

```
CMPB      ALPHA,BETA
CMPB      ALPHA+1,BETA+1
CMPB      ALPHA+2,BETA+2
. . .

CMPB      ALPHA+9,BETA+9
```

Последовательность сравнений заканчивается, если обнаруживаются два неравных байта или после сравнения всех байтов выясняется, что строки совпадают. В результате коды условий показывают, является ли первая строка большей, равной или меньшей, чем вторая символьная строка.

Инструкция CMPC3 модифицирует содержимое регистров от R0 до R3. Когда строки равны, регистр R1 содержит адрес байта, следующего за последним байтом первой строки, регистр R3 содержит адрес байта, следующего за последним байтом второй строки, а содержимое регистров R0 и R2 сбрасывается. Однако, когда строки не равны, регистры R1 и R3 содержат адреса несовпавших при сравнении байтов, а регистр R0 содержит число байтов, оставшихся в каждой строке, включая байт, на котором это произошло. Рассмотрим, например, следующее:

```
GAMMA:    .ASCII   /ABCDEF/
DELTA:    .ASCII   /ABCDXY/
          . . .

          CMPC3    #6,GAMMA,DELTA
```

После выполнения инструкции регистр R0 содержит число 2, а это показывает, что в каждой строке оставалось еще 2 байта, когда было обнаружено различие. Регистр R1 содержит адрес байта, в котором находится символ E, регистр R3 содержит адрес байта с символом X, а регистр R0 содержит нуль.

Инструкция CMPC5 предназначена для сравнения двух строк, которые могут иметь разную длину. Она имеет такой же формат, как инструкция MOVC5. Инструкция CMPC5 работает так, как если бы более короткая из двух сравниваемых строк была бы дополнена символами-заполнителями до достижения длины более длинной строки. Рассмотрим для примера следующие строки:

```
ONE:      .ASCII   /ABC/
TWO:      .ASCII   /ABC**!/
```

Строки, сравниваемые следующими инструкциями, изображены справа от каждой инструкции:

Инструкция		Сравнение строк	Причина	N	Z	V	C
CMPC5	#3, ONE, #^A"$", #6, TWO	ABC$$$ < ABC**!	$ < *	1	0	0	1
CMPC5	#3, ONE, #^A"*", #6, TWO	ABC*** > ABC**!	* > !	0	0	0	0
CMPC5	#6, TWO, #^A"$", #3, ONE	ABC**! > ABC$$$	* > $	0	0	0	0
CMPC5	#6, TWO, #^A"*", #3, ONE	ABC**! < ABC***	! < *	1	0	0	1

Если символьные строки равны, бит Z устанавливается в 1, а биты N и C сбрасываются в нуль. Если строки не равны, биты N и C устанавливаются в зависимости от байтов, вызвавших несовпадение. Бит N устанавливается, если значение байта первой строки меньше значения байта второй строки, как число со знаком, тогда как бит C устанавливается, если значение первого байта меньше значения второго байта как число без знака.

В начале выполнения инструкции CMPC5длины двух строк помещаются в регистры R0 и R2, а адреса строк – в регистры R1 и R3. Если строки равны, то при завершении инструкции регистры R0 и R2 будут содержать 0, а регистры R1 и R3 будут содержать адреса байтов, непосредственно следующих за последним байтом соответствующей строки.

Если строки не равны, регистры R0 и R2 показывают число байтов, оставшихся в соответствующих строках, включая байт, на котором произошло несовпадение (если несовпавший байт является одним из тех, в которые были добавлены символы-заполнители, то соответствующий регистр будет содержать нуль). Регистры R1 и R3 содержат адреса байтов, вызвавших несовпадение (если несовпавшим байтом является один из тех, в которые были добавлены символы-заполнители, то соответствующий регистр будет содержать адрес байта, следующий за последним байтом строки).

Символ-заполнитель используется в инструкции MOVC5, только если первая строка короче второй. В инструкции CMPC5 символ-заполнитель может использоваться независимо от того, какая из строк длиннее. В результате этого выполнение последовательности инструкции MOVC5 и CMPC5 с идентичными операндами не обязательно приводит к совпадению результатов. Например:

```
THREE:  .ASCII  "ABCDEF"
FOUR:   .BLKB   3
        . . .

        MOVC5   #6, THREE, #^A"*", #3, FOUR
        CMPC5   #6, THREE, #^A"*", #3, FOUR
```

Инструкция MOVC5 перешлет ABC в строку FOUR. Однако инструкция CMPC5 будет сравнивать строку ABCDEF со строкой ABC***, что даст несовпадение строк.

Как и в семействе инструкций MOVC, в инструкциях CMPC3 и CMPC5 строки могут частично перекрываться. Например, инструкция

```
CMPC3   #10, STRING, STRING+1
```

обнаружит совпадение, если только байты, начиная со STRING+1 по STRING+9, равны байту с адресом STRING (если строка содержит AAAAABCDEF, регистр R1 будет содержать адрес пятой буквы A, а регистр R3 будет содержать адрес буквы B после завершения выполнения). Аналогично инструкция

```
CMPC5   #SIZE, BYTE_ARRAY, #128, #0, (SP)
```

будет осуществлять сканирование байтового массива BYTE_ARRAY, оставляя по завершении в регистре R1 адрес первого байта, содержимое которого не равно десятичному числу 128 (поскольку вторая строка имеет нулевую длину, операнд (SP) используется как адрес второй строки).

13.5. ДРУГИЕ ИНСТРУКЦИИ ОБРАБОТКИ СИМВОЛЬНЫХ СТРОК

ИНСТРУКЦИИ SKPC, LOOC И MATCHC

Как отмечалось в предыдущем разделе, инструкция CMPC5 может использоваться для пропуска в символьной строке символов, равных символу-заполнителю. Например, рассмотрим следующее:

```
STRING:   .ASCII      "    НАЧИНАЕТСЯ С ПЯТИ ПРОБЕЛОВ"
STRING_END:
LENGTH=STRING_END-STRING
        . . .

        CMPC5       #LENGTH, STRING, #^A" ", #0, (SP)
```

Поскольку символом-заполнителем является пробел, а вторая строка имеет нулевую длину, инструкция CMPC5 впервые обнаружит неравные байты при сравнении символа-заполнителя с буквой Н в слове НАЧИНАЕТСЯ. После завершения инструкции регистр R1 будет содержать буквы Н.

Такой же эффект можно получить с помощью инструкции SKPC (пропустить символ), включенной в архитектуру ЭВМ семейства VAX. Например, инструкция

```
SKPC        #^A" ", #LENGTH, STRING
```

также приведет к тому, что регистр R1 будет содержать адрес буквы Н. Регистр R0 будет содержать 26 – число байтов, оставшихся в строке (включая букву "Н") после нахождения символа, отличного от пробела. В противоположность инструкции CMPC5 инструкция SKPC всегда сбрасывает в нуль биты N, V и C. Бит Z устанавливается в соответствии со значением в регистре R0. Если в строке обнаруживается символ, отличный от символа-заполнителя, регистр R0 будет содержать число оставшихся в строке байтов, а бит Z сбрасывается. Если все символы строки равны символу-заполнителю, то регистр R0 содержит 0, а бит Z устанавливается. Регистры R0 и R1 – единственные регистры, используемые инструкцией SKPC как временная память.

Инструкция LOCC (обнаружить символ) идентична инструкции SKPC, за исключением того, что она завершается по совпадению байтов вместо их несовпадения. Например, инструкция

```
STRING:   .ASCII      "НАЧИНАЕТСЯ БЕЗ ПРОБЕЛОВ"
STRING_END:
LENGTH=STRING_END-STRING
        . . .

        LOCC        #^A" ", #LENGTH, STRING
```

по завершению оставит в регистре R1 адрес пробела, следующего за буквой Я в слове НАЧИНАЕТСЯ. Как и инструкция SKPC, инструкция LOCC устанавливает бит Z в зависимости от значения в регистре R0. В приведенном выше примере регистр R0 будет содержать 13 после завершения инструкции, поскольку в строке остается 13 байтов (включая символ пробела). Таким образом, бит Z будет сброшен.

Обнаружение какого-либо заданного символа в символьной строке – весьма частая операция. Например, инструкция

```
LOCC        #^A":", #LENGTH, STRING
```

может быть использована для поиска символа :, которым оканчиваются метки в программе на языке ассемблера. Та же самая инструкция может использоваться и для поиска двоеточия в спецификации файла в ОС VAX/VMS. Фактически подпрограмма, которая осуще-

ствляет синтаксический разброс спецификации файла, может воспользоваться несколькими инструкциями LOCC для обнаружения таких символов, как : , [,] , . и : . Аналогично подпрограмма, выполняющая синтаксический разброс выражений в языках высокого уровня, может использовать инструкции LOCC для обнаружения таких символов, как = , : , (,) , + , – , * и / .

Инструкции SKPC и LOCC могут применяться для обработки строк, состоящих из слов, разделенных одним или несколькими символами-разделителями. В следующем примере в качестве символа разделителя используется звездочка * (конечно, в обычном тексте разделителем может служить символ пробела):

```
STRING:   .ASCII"**СЛОВА*РАЗДЕЛЯЮТСЯ***СИМВОЛОМ*****ЗВЕЗДОЧКА**"
STRING_END:
LENGTH=STRING_END-STRING
          . . .

          SKPC    #^A"*", #LENGTH, STRING  ;R1 УКАЗЫВАЕТ НА "С" В "СЛОВА"
          LOCC    #^A"*", #R0, R1          ;R1 УКАЗЫВАЕТ НА "*" ПОСЛЕ "СЛОВА"
          SKPC    #^A"*", #R0, R1          ;R1 УКАЗЫВАЕТ НА "Р" В "РАЗДЕЛЯЮТСЯ"
          LOCC    #^A"*", #R0, R1          ;R1 УКАЗЫВАЕТ НА "*" ПОСЛЕ "РАЗДЕЛЯЮТСЯ"
          SKPC    #^A"*", #R0, R1          ;R1 УКАЗЫВАЕТ НА "С" В "СИМВОЛОМ"
```

Комментарии в каждой строчке программы сообщают о том, адрес какого байта содержит регистр R1 после выполнения инструкции. Обратите внимание, как результаты, оставляемые в регистрах R0 и R1 одной инструкцией, используются как входная информация для следующей инструкции. Слова строки могут быть обработаны с помощью следующего цикла:

```
          MOVZWL  #LENGTH, R0      ;R0 - ДЛИНА СТРОКИ
          MOVAB   STRING, R1       ;R1 - АДРЕС СТРОКИ
10$:      SKPC    #^A"*", R0, R1   ;ПРОПУСТИТЬ "*", НАЧАТЬ СО СЛЕДУЮЩЕГО СЛОВА
          BEQL    20$              ;ЕСЛИ В R0 НУЛЬ, ТО КОНЕЦ СТРОКИ
           ОБРАБОТКА, ВЫПОЛНЯЕМАЯ ПРИ ОБНАРУЖЕНИИ НАЧАЛА СЛОВА
          LOCC    #^A"*", #R0, R1  ;НАЙТИ "*" В КОНЦЕ СЛОВА
          BEQL    20$              ;ЕСЛИ В R0 НУЛЬ, ТО КОНЕЦ СТРОКИ
           ОБРАБОТКА, ВЫПОЛНЯЕМАЯ ПРИ ДОСТИЖЕНИИ КОНЦА СЛОВА
          BRB     10$
20$:      . . .
```

Инструкция MATCHC (найти соответствие символьной подстроки) близка инструкции LOCC. Однако LOCC обнаруживает первое появление в строке конкретного символа, а инструкция MATCHC обнаруживает первое появление в строке конкретной подстроки. Рассмотрим, например, следующее:

```
STRING:          .ASCII    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
LENGTH=26
SUB_STRING:      .ASCII    "DEF"
SUB_LENGTH=3
                 . . .

                 MATCHC    #LENGTH, STRING, #SUB_LENGTH, SUB_STRING
```

Заметим, что два первых операнда задают длину и начальный адрес строки, а третий и четвертый операнды определяют длину и начальный адрес подстроки. В этом примере внутри строки будет найдена подстрока DEF. Инструкция использует для временного запоминания данных регистры R0 – R3. В данном случае после выполнения инструкции регистры R0 – R3 будут содержать следующее:

R0 – 23 (число байтов, оставшихся в строке, начиная с первого совпадающего символа);
R1 – адрес первого совпадающего символа;
R2 – 3 (длина подстроки);
R3 – адрес подстроки.

Коды условий N, V и C сбрасываются, а бит Z устанавливается в соответствии со значением в регистре R0. В данном случае регистр R0 содержит число 23, поэтому бит Z будет сброшен.

Если подстрока не найдена, регистр R0 будет содержать нуль, а регистр R1 будет содержать адрес байта, следующего за последним байтом строки. Кроме того, регистр R2 будет содержать длину подстроки (в данном случае 3), а регистр R3 будет содержать адрес начала подстроки. Поскольку содержимое регистра R0 равно 0, бит Z будет установлен.

ИНСТРУКЦИИ ПРЕОБРАЗОВАНИЯ КОДА СИМВОЛОВ

Четыре дополнительные инструкции напоминают инструкции MOVC, LOCC и SKPC, за исключением того, что они используют добавочную таблицу или строку из 256 байтов. Инструкции MOVC, LOCC и SKPC последовательно производят выборку байтов из исходной строки (инструкция MOVC эти байты пересылает в целевую строку; инструкции LOCC и SKPC сравнивают их с символом, определяемым третьим операндом). Инструкции, описываемые в этом разделе, работают аналогично, за исключением того, что после выборки каждого байта из исходной строки он используется как индекс в таблице из 256 байтов. Соответствующий байт извлекается из таблицы и употребляется вместо байта из исходной строки.

Ниже показаны операнды инструкции MOVTC (переслать преобразованные символы). Операнды инструкции MOVC5 также показаны для того, чтобы показать сходство этих двух символов инструкций.

```
MOVC5     srclen, srcadr, fill, dstlen, dstadr
MOVTC     srclen, srcadr, fill, tbladr, dstlen, dstadr
```

Обратите внимание, что единственное различие между инструкциями MOVC5 и MOVTC состоит в добавлении операнда tbladr, который задает адрес начала 256-байтовой таблицы.

Инструкция MOVTC может использоваться для выполнения преобразований символов из кода ASCII в код EBCDIC или обратно. Она может применяться для преобразования строчных букв в прописные или для преобразования пробелов в нули или наоборот. Но ею можно пользоваться и для других целей. Рассмотрим следующий простой пример, который иллюстрирует работу инструкции MOVTC:

```
CNT=1
TABLE:
          .REPT   255
          .BYTE   CNT
CNT=CNT+1
          .ENDR
          .BYTE   0
          . . .

ARRAY:    .BYTE   27, 255, 0, 100, 1
          . . .

          MOVTC   #5, ARRAY, #0, TABLE, #5, ARRAY
```

256-байтовая таблица содержит числа 1, 2, 3, . . . , 254, 255, 0. Таким образом, значение в BYTE[I] равно I+1 (последний, 256-й байт рассматривается как специальный признак и поэтому содержит 0). Инструкция MOVTC извлекает каждый байт из байтового массива ARRAY и использует его значение как индекс байта в таблице (который содержит значение, на 1 большее номера байта в массиве), выбранный из таблицы байт снова помещается в массив. В результате инструкция MOVTC как бы прибавляет 1 к каждому байту массива, так что после завершения инструкции массив содержит числа 28, 0, 1, 101, 2.

В отношении использования регистров MOVTC рассматривает таблицу как третью свободную строку. Таким образом, регистры R0 и R1 используются для хранения длины и

адреса исходной строки, регистры R2 и R3 – для длины (256) и адреса таблицы, а регистры R4 и R5 – для длины и адреса целевой строки. После завершения инструкции MOVTC регистры содержат следующее:

```
MOVTC       srclen, srcadr, fill, tbladr,   dstlen, dstadr
            R0=a    R1=б          R3=адр    R4=0    R5=в          (R2=0)

а - srclen-dstlen, если srclen > dstlen; в противном случае 0
б - адрес байта, следующего за последним перекодированным байтом
в - адрес байта, следующего за последним байтом строки-получателя
```

Как и в случае инструкции MOVC5, инструкция MOVTC устанавливает коды условий так, как если бы выполнена пара инструкций:

```
MOVTC       srclen, srcadr, fill, tbladr, dstlen, dstadr
CMPW        srclen, dstlen
```

за исключением того, что бит Z сбрасывается.

Инструкция MOVTUC (пересылать преобразованные символы до переключения) работает так же, как инструкция MOVTC, за исключением того, что выполнение завершается, когда из таблицы извлекается определенный символ, называемый символом переключения кода (esc). Описание этой инструкции таково:

```
MOVTUC      srclen, srcadr, esc, tbladr,   dstlen, dstadr
            R0=a    R1=б         R3=адр    R4=в    R5=г          (R2=0)

MOVTC       srclen, srcadr, fill, tbladr, dstlen, dstadr

а - если встречен символ-переключатель, то число оставшихся в строке байт,
    включая символ-переключатель; в противном случае 0
б - адрес байта, выборка которого была осуществлена последней,
    или адрес байта, следующего за последним байтом строки
в - число байтов, оставшихся в строке-получателе
г - адрес перого немодифицированного байта
```

Как показывает сравнение с инструкцией MOVTC, символ-заполнитель заменяется символом переключения кода. Поскольку символ-заполнитель отсутствует, инструкция завершается, если в исходной строке исчерпаны все байты; при этом регистр R1 содержит адрес первого байта, следующего за последним байтом строки. Инструкция MOVTUC устанавливает коды условий так же, как инструкция MOVTC (CMPW srclen dstlen), за исключением того, что бит V устанавливается, если инструкция завершается при извлечении из таблицы символа, соответствующего символу переключения кода.

Инструкции SCANC и SPANC подобны инструкциям LOCC и SKPC, за исключением того, что завершение происходит, когда обнаруживается один из некоторого набора символов (SCANC) или когда обнаруживается символ, не входящий в заданный набор (SPANC). Формат этих инструкций такой:

```
SCANC       srclen, srcadr, tbladr, mask
            R0=a    R1=б    R3=адр                          (R2=0)
SPANC       srclen, srcadr, tbladr, mask
            R0=a    R1=б    R3=адр                          (R2=0)

а - число оставшихся в строке байт, включая байт на котором было
    завершено выполнение инструкции
б - адрес байта на котором завершилось выполнение инструкции или
    srcadr+srclen
```


Байты из исходной строки (определенной первыми двумя операндами) последовательно используются как индексы таблицы (определенной третьим операндом). Над каждым байтом, извлеченным из таблицы, выполняется логическая операция **И** с байтом, заданным четвертым операндом (маска). Инструкция SCANC завершается, когда результат выполнения операции **И** отличен от нуля или исчерпаны все символы исходной строки. Инструкция SPANC завершается, когда результат выполнения операции **И** равен 0 или когда исчерпаны все символы в исходной строке.

Восьмибитовая маска обеспечивает большую гибкость. Например, символы могут быть классифицированы по восьми двоичным атрибутам, таким как цифра или нецифра, прописная буква или любой другой символ, строчная буква или любой другой символ и т. п. Вход в таблицу для каждого символа базируется на восьми двоичных атрибутах. При выборе подходящей маски инструкция SCANC может искать символ, обладающий одним или несколькими из этих атрибутов. (Инструкция SPANC может использоваться для пропуска символов, обладающих одним или несколькими атрибутами.)

УПРАЖНЕНИЯ 13.1

1. Опишите, как можно реализовать инструкцию MOVC3 таким образом, что не нужна будет временная память для символов, даже если строки перекрываются.
2. Напишите макроинструкцию, имитирующую инструкцию MOVC3, используя только инструкции, описанные в гл. 8 (ваша макроинструкция должна уметь справляться с перекрывающимися строками). Напишите основную программу, которая тестирует вашу макроинструкцию, сравнивая полученные результаты с результатами инструкции MOVC3.
3. Напишите программу, которая считывает предложения некоторого текста и создает отсортированный список слов из этого текста. Вы можете полагать, что слова в тексте разделяются одним или несколькими пробелами и что нет никаких других знаков пунктуации. Сделайте предположение, что слова не могут быть длиннее 20 символов.
4. Перепишите программу п. 3 упр. 13.1, полагая, что текст содержит реальную пунктуацию, включая точки, запятые, вопросительные знаки, тире, апострофы и т. п.
5. Напишите программу, которая считывает текст, как в п. 3 упр. 13.1, и распечатывает текст в виде правильно сформатированного параграфа. (Между словами должно быть только по одному пробелу и в каждой строке должно размещаться максимальное число слов.) Заметим, что входной текст может содержать между словами много пробелов.
6. Модифицируйте программу п. 5 так, чтобы между словами добавлялось достаточное для выравнивания текста в строчке по правой границе число пробелов.

*7. Напишите программу, которая считывает любую программу на языке ассемблера и распечатывает все символические имена, встречающиеся в этой программе. (Заметим, что при этом требуется частичный синтаксический разбор программы на языке ассемблера.)

*8. Скомбинируйте п. 3 и 7 для создания программы, которая распечатывает полную таблицу имен для программы на языке ассемблера (без значений имен).

13.6. АРИФМЕТИКА УПАКОВАННЫХ ДЕСЯТИЧНЫХ ЧИСЕЛ

УПАКОВАННЫЕ ДЕСЯТИЧНЫЕ ЧИСЛА

При обработке экономической информации программисты предпочитают обозначать числа в виде последовательности десятичных цифр фиксированной длины с фиксированной позицией десятичной точки. В языках, предназначенных для обработки экономической информации, таких как Кобол, подобный вид представления чисел используется очень широко. Несколько причин применения десятичных чисел:

1. Ввод с перфокарт и вывод на печать делается в виде десятичных полей фиксированной длины.
2. Большинство числовых значений при обработке экономической информации используется в простых вычислениях, и преобразование десятичных чисел в двоичные числа

для выполнения вычислений было бы расточительным.

3. Фиксированное число цифр помогает предотвращать ошибки, такие как выдача чека на 1 000 000.00 долл. на мелкие расходы.
4. Десятичные дроби, такие как 0.20, не могут быть точно представлены в двоичном виде.
5. Представление чисел с плавающей точкой может не обеспечить желаемую точность. Например, большинство финансовых расчетов выполняется с округлением до ближайшего цента, и если при таких вычислениях сохраняются дробные части цента, то это может приводить к бухгалтерским ошибкам.

Чтобы избежать этих и других проблем, первые ЭВМ (первого поколения), разрабатываемые для обработки экономической информации, позволяли представлять числа в виде строк десятичных цифр. Как универсальные вычислительные машины, предназначенные как для обработки данных, так и для научных расчетов, ЭВМ семейства VAX включают поддержку десятичных чисел и десятичной арифметики. Основной идеей является упаковка двух десятичных цифр в каждом байте (отсюда и название *упакованные десятичные числа*). Например, десятичное число 12345678 может быть представлено в четырех байтах, начиная с адреса ^X1000, так:

Содержимое	Адрес
12	1000
34	1001
56	1002
78	1003

Каждый 8-битовый байт составляется из двух 4-битовых полубайтов, а каждый полубайт содержит десятичную цифру. Например, в байте ^X1000 левый полубайт содержит ^X1, что представляет десятичную цифру 1, а правый полубайт содержит ^X2, что представляет десятичную цифру 2.

Если содержимое байтов ^X1000 – ^X1003 поместить на одной строке, то результат будет таким:

Содержимое	Адрес
78 56 34 12	1000

Поскольку старшие цифры числа (в данном случае 12) запоминаются в байте с меньшим адресом (в данном случае ^X1000), упакованные десятичные числа могут казаться неудобными при чтении дампа памяти. (Обратите внимание, что содержимое длинного слова, начинающегося с адреса ^X1000, есть ^X78563412 и оно является представлением упакованного десятичного числа 12345678.)

Описанное выше представление упакованного десятичного числа должно быть модифицировано так, чтобы зарезервировать место для знака числа. В ЭВМ семейства VAX придерживаются соглашения, что знак числа запоминается в правом полубайте байта с самым старшим адресом. Это означает, что самый старший байт содержит в левом полубайте самую младшую цифру (цифру разряда единиц) и знак в правом полубайте. Хотя в ЭВМ семейства VAX допускается, чтобы положительный знак представлялся шестнадцатеричными цифрами A, C, E или F, а отрицательный знак – цифрами B или D, предпочтительными являются представление ^XC для плюса и ^XD для минуса. Таким образом, десятичное число 12345678 представляется так:

Содержимое	Адрес
01	1000
23	1001
45	1002
67	1003
8C	1004

В листинге это может выглядеть следующим образом:

Содержимое	Адрес
8C 67 45 23 01	1000

Десятичное число –12345678 было бы представлено в пяти байтах так:

Содержимое	Адрес
8D 67 45 23 01	1000

В ЭВМ семейства VAX упакованные десятичные числа задаются двумя атрибутами: начальным адресом числа и длиной. Однако в противоположность символьным строкам, длина упакованного десятичного числа задается числом десятичных цифр, а не байтов. Таким образом, длина упакованного десятичного числа 12345678 (или –12345678) есть восемь, хотя это число занимает 5 байтов.

Заметим, что представление в памяти девятизначного упакованного десятичного числа 012345678 точно совпадает с представлением восьмизначного числа 12345678. Оба числа будут представлены в 5 байтах, содержащими 8C, 67, 45, 23 и 01. Однако два числа могут различаться по длине (девять или восемь цифр). В общем случае длина упакованного десятичного числа может изменяться от 0 до 31-й десятичной цифры. Заметим, что любое упакованное десятичное число четной длины (такое как 12345678) будет иметь незадействованный полубайт в байте старшего порядка (байт с младшим адресом). Десятичная строка нулевой длины состоит из одного байта, содержащего знак в правом полубайте и неиспользованный левый полубайт. Десятичная строка длиной 31 состоит из 16 байтов.

В языке ассемблера для резервирования памяти и инициализации упакованных десятичных чисел используется директива .PACKED. Следующий листинг ассемблера демонстрирует способ использования директивы .PACKED. В первых четырех строчках определяются упакованные десятичные числа, которые были описаны ранее. В строчке 5 за значением 12345678 следует символическое имя L_PACKE. В результате этого директива .PACKED определяет имя L_PACKE как длину упакованного десятичного числа PACKE. В данном случае имя L_PACKE будет занесено в таблицу имен со значением 8. В строке 6 имя L_PACKF будет занесено в таблицу имен со значением 9. Наконец, в строке 7 показано, как может быть зарезервировано (без инициализации) место в памяти для упакованного десятичного числа PACKG, имеющего то же самое число байтов, что и PACKE.

```
8C 67 45 23 01   0000     1 PACKA:  .PACKED  12345678
8D 67 45 23 01   0005     2 PACKB:  .PACKED  -12345678
                 0D 000A  3 PACKC:  .PACKED  -0
                 0C 000B  4 PACKD:  .PACKED  +0
8C 67 45 32 01   000C     5 PACKE:  .PACKED  12345678,L_PACKE
8C 67 45 32 01   0011     6 PACKF:  .PACKED  012345678,L_PACKF
      0000001A   0016     7 PACKG:  .BLKB    <L_PACKE+1>/2+1
                 001A     8         .END
```

ИНСТРУКЦИИ ДЛЯ РАБОТЫ С УПАКОВАННЫМИ ДЕСЯТИЧНЫМИ ЧИСЛАМИ

В архитектуру ЭВМ семейства VAX включены инструкции для пересылки, сравнения и выполнения арифметических операций над упакованными десятичными числами. Форматы инструкций для упакованных десятичных чисел аналогичны форматам инструкций для целых чисел и чисел с плавающей точкой, за исключением того, что должны быть включены операнды, определяющие длину упакованных десятичных чисел. Это такие инструкции:

Мнемоника	Операнды	Описание
MOVP	len, srcadr, dstadr	dst <- src
CMPP3	len, srcadr, dstadr	tmp <- src - dst
CMPP4	srclen, srcadr, src2len, src2adr	tmp <- src : dst
ADDP4	srclen, srcadr, dstlen, dstadr	dst <- dst + src
SUBP4	srclen, srcadr, dstlen, dstadr	dst <- dst - src
ADDP6	srclen, srcladr, src2len, src2adr, dstlen, dstadr	dst <- src2 + srcl
SUBP6	srclen, srcladr, src2len, src2adr, dstlen, dstadr	dst <- src2 - srcl
MULP	srclen, srcladr, src2len, src2adr, dstlen, dstadr	dst <- src2*srcl
DIVP	srclen, srcladr, src2len, src2adr, dstlen, dstadr	dst <- src2/srcl

Инструкции для упакованных десятичных чисел подразумевают использование регистров общего назначения от R0 до R5 точно так же, как и в инструкциях обработки символьных строк. Регистры R0, R2 и (если необходимо) R4 используются для запоминания длины упакованных десятичных чисел, а регистры R1, R3 и (если необходимо) R5 – для адресов этих чисел. После выполнения инструкции регистры с четными номерами содержат нуль, а регистры с нечетными номерами содержат адреса начала соответствующих десятичных чисел. Например, после выполнения инструкции MOVP (переслать упакованное десятичное) регистры R0 и R2 содержат нуль, регистры R1 и R3 указывают на адреса операндов источника и получателя, а регистры R4 и R5 не используются и не модифицируются. Адреса в регистрах с нечетными номерами могут применяться в качестве входной информации следующей инструкции.

Эти инструкции устанавливают коды условий N, Z и V вполне предполагаемым образом. Бит V устанавливается, если происходит переполнение десятичного числа. Поскольку не существует упакованных десятичных чисел без знака, бит C просто сбрасывается, за исключением инструкции MOVP, которая сохраняет значение бита C.

Имеется одна тонкость в отношении установки битов N и Z. В зависимости от способа представления в упакованном десятичном числе знакового разряда, существует упакованное десятичное представление как +0, так и –0 (так как длина упакованного десятичного числа может изменяться от 0 до 31, фактически существует 64 различных представления нуля). Существование представлений как для +0, так и для –0 в некоторых ЭВМ более старой архитектуры порождало трудности (инструкция перехода по условию "если меньше чем 0" могла бы вызвать переход при появлении числа –0, но переход не будет осуществлен в случае числа +0, хотя обе эти величины математически эквивалентны). Чтобы избежать этого в архитектуре ЭВМ семейства VAX приняты два решения. Во-первых, нулевой результат всегда представляется как +0. Например, если исходным операндом инструкции MOVP является –0, то в операнд-получатель будет переслан +0. Но существует возможность получения результата –0, если происходит переполнение десятичного числа. Вторая особенность решает эту проблему. Если результат есть –0, ЭВМ семейства VAX просто устанавливает биты N и Z так, как если бы результат был +0. Таким образом, при получении результата +0 или –0 биты будет сбрасываться и установится бит Z. Поэтому программистам не нужно беспокоиться о существовании представления числа –0, за исключением весьма необычных обстоятельств.

ИНСТРУКЦИИ ASHP, CVTLP И CVTPL

Чтобы можно было обрабатывать дробные значения, упакованные десятичные числа рассматриваются как числа с фиксированной точкой. Как пояснялось в гл. 10, это означает, что нахождение десятичной точки подразумевается в некоторой фиксированной позиции внутри числа. Следить за позицией подразумеваемой десятичной точки – обязанность программиста. В частности, расположение десятичных точек должно выравниваться перед сложением или вычитанием. (Как мы видели в гл. 10, та же самая проблема возникает при вычислениях с плавающей точкой. Однако необходимое там выравнивание осуществляется автоматически инструкциями для чисел с плавающей точкой.) Выравнивание упакованных десятичных чисел может выполняться путем умножения или деления чисел на степени числа 10. Однако умножение или деление упакованного десятичного числа можно быстрее выполнить с помощью сдвига полубайтов. Такой сдвиг выполняется инструкцией ASHP (сдвинуть арифметически упакованное десятичное число).

Инструкция ASHP требует шесть операндов. В дополнение к длинам и адресам для исходной и целевой строки первый операнд инструкции, имеющий формат байта, задает величину сдвига в десятичных разрядах. Положительное значение этого операнда приводит к эффективному умножению исходного операнда на 10^n и размещению результата в целевой строке. Отрицательное значение приводит к делению исходного операнда на 10^n. Четвертый операнд – это операнд, имеющий формат байта, который позволяет выполнять деление с округлением вместо усечения. Например, следующая инструкция преобразует упако-

ванное десятичное число, соответствующее величине, выраженной в долларах, в число, соответствующее величине, выраженной в центах:

```
ASHP    #2, #DOLLAR_LEN, DOLLAR, #0, #CENTS_LEN, CENTS
```

Величина, выраженная в центах, может быть преобразована в доллары с помощью следующей инструкции:

```
ASHP    #-2, #CENTS_LEN, CENTS, #0, #DOLLAR_LEN, DOLLAR
```

Эта инструкция осуществит усечение при пересылке упакованного десятичного числа CENTS в DOLLAR. Округление может быть получено с помощью инструкции:

```
ASHP    #-2, #CENTS_LEN, CENTS, #5, #DOLLAR_LEN, DOLLAR
```

Четвертый операнд #5 перед делением на 100 прибавляет 50 к упакованному десятичному числу, расположенному по адресу CENTS. Инструкция ASHP использует регистры R0 – R3, как это ожидается.

Инструкция MOVP предназначена для пересылки исходного упакованного числа в область памяти, занимающую то же число байтов. Для пересылки упакованных десятичных чисел между полями разной длины может использоваться инструкция ASHP (с операндами сдвига и округления, равными 0). Например, инструкция

```
ASHP    #0, #4, SOURCE, #0, #10, DEST
```

может использоваться для пересылки четырехзначного исходного упакованного десятичного числа в область памяти, вмещающую десятизначное число.

В архитектуре ЭВМ семейства VAX имеется одна пара инструкций для преобразования упакованных десятичных чисел в целые числа со знаком и обратно. Это инструкции CVTLP (преобразовать длинное слово в упакованное) и CVTPL (преобразовать упакованное число в длинное слово). Описание этих инструкций:

```
CVTLP     longsrc, dstlen, dstadr
            R2=0   R3=a                          R0=0, R1=0

CVTPL     srclen, srcadr, longdst
          R0=0   R1=a                            R2=0, R3=0
```

a = адрес байта, следующего за последним байтом упакованного десятичного числа.

Инструкция преобразования длинного слова легко может быть использована для преобразования между целыми числами в формате байта и слова и упакованными десятичными числами. Например, следующие инструкции преобразуют 16-битовое целое со знаком, расположенное по адресу WORD, в пятизначное упакованное десятичное число:

```
CVTWL     WORD, LONG          ;ПРЕОБРАЗОВАТЬ СЛОВО В ДЛИННОЕ СЛОВО
CVTLP     LONG, #5, PACK_ADR  ;ПРЕОБРАЗОВАТЬ ДЛ. СЛОВО В УПАКОВ. ДЕСЯТИЧНОЕ
```

Обратите внимание, что переполнения никогда не произойдет, поскольку 16-битовое целое со знаком всегда помещается в пятизначном десятичном поле. Аналогично следующие операторы будут преобразовывать пятизначное упакованное десятичное число в 16-битовое двоичное целое. Заметим, что возможно переполнение.

```
CVTPL     #5, PACK_ADR, LONG  ;ПРЕОБРАЗОВАТЬ УПАКОВ. ДЕСЯТИЧНОЕ В ДЛ. СЛОВО
CVTLW     LONG, WORD          ;ПРЕОБРАЗОВАТЬ ДЛИННОЕ СЛОВО В СЛОВО
                              ;(ВОЗМОЖНО ПЕРЕПОЛНЕНИЕ)
```

Преобразование между упакованными десятичными числами и целыми со знаком в формате квадраслова является более сложным. Следующие операторы преобразовывают целое со знаком в формате квадраслова с адресом QUAD в двадцати-значное упакованное десятичное число с адресом PACKED_ANS:

```
PACKED_CONST:
        .PACKED  4294967286              ;КОНСТАНТА 2**32
        . . .

        MOVL     QUAD,LONG_LOW           ;РАЗБИТЬ НА ДВА ДЛИННЫХ СЛОВА
        MOVL     QUAD+4,LONG_HIGH
        CVTLP    LONG_LOW,#10,PACKED_LOW ;ПРЕОБРАЗОВАТЬ В ДЕСЯТИЧНОЕ
        CVTLP    LONG_HIGH,#10,PACKED_HIGH
;ПОМЕСТИТЬ ДВА УПАКОВ. ДЕСЯТИЧНЫХ ДЛИНОЙ 10 В PACKED_ANS С ДЛИНОЙ 20
        MULP     #10,PACKED_CONST,#10,PACKED_HIGH,#20,PACKED_ANS
        ADDP4    #10,PACKED_LOW,#20,PACKED_ANS
        TSTL     LONG_LOW
        BGEQ     10$
        ADDP4    #10,PACKED_CONST,#10,PACKED_ANS
10$:    . . .
```

Первые две инструкции разбивают 64-битовое квадраслово на два 32-битовых длинных слова LONG_LOW и LONG_HIGH. Если оба длинных слова положительны (т. е. бит 31 нулевой), то преобразование очевидно. Длинные слова преобразуются в десятизначные упакованные десятичные числа, упакованное десятичное число, выражающее старшие порядки, умножается на 2^{32} и пересылается в PACED_ANS, а упакованное десятичное число, выражающее младшие порядки, прибавляется к PACED_ANS.

Если одно или оба числа в формате длинного слова отрицательны, алгоритм более сложен. Например, предположим, что бит 31 в LONG_LOW равен 1, а бит 31 в LONG_HIGH нулевой. Бит 31 в LONG_LOW представляет 2^{31}-битовую позицию в 64-битовом квадраслове. Однако инструкция CVTLP обрабатывает этот бит как знаковый, и операнд PACKED_LOW становится отрицательным числом. После выполнения первой инструкции ADDP4 ответ неверен по двум причинам. Во-первых, число в PACKED_LOW в действительности вычитается из числа в PACKED_ANS, а не прибавляется к нему. Во-вторых, поскольку бит 31 на самом деле является битом данных, но не обрабатывается как бит данных, необходимо прибавить к ответу число 2^{31}. Ситуация такова, что комбинированное влияние этих двух факторов всегда дает в результате ответ, который меньше правильного на 2^{32}. Последние три инструкции прибавляют 2^{32} к ответу, если число в PACKED_LOW отрицательно. В качестве упражнения предлагаем читателю продемонстрировать работу этого алгоритма для ряда чисел.

13.7. ЧИСЛОВЫЕ СТРОКИ

ФОРМАТЫ ЧИСЛОВЫХ СТРОК

До сих пор было рассмотрено четыре различных представления чисел: двоичные целые без знака, двоичные целые со знаком, вещественные или числа с плавающей точкой и упакованные десятичные числа. Арифметические вычисления могут выполняться над числами в любом из этих представлений. Кроме того, числа, имеющие любое из этих представлений, могут быть записаны или считаны с внешних запоминающих устройств, таких как магнитный диск или магнитная лента. Однако они не могут быть выведены или считаны с устройств, использующих символьное представление информации в коде ASCII, таких как клавиатура терминалов, экраны дисплеев или устройства печати. Подобные устройства требуют, чтобы числа были представлены в виде строк символов в коде ASCII.

Для того чтобы поддерживать работу устройств, использующих символьный формат, в архитектуру ЭВМ семейства VAX включено пятое представление чисел, называемое *числовыми строками*. По существу числовая строка – это не что иное, как символьная строка, в которой символы ограничены набором цифр в коде ASCII от 0 до 9. Другими словами, содержимое байтов в строке ограничивается значениями цифр в коде ASCII, а именно от ^X30 до ^X39.

Числа представляются в виде символьных строк со времени появления ЭВМ. Вследствие различий между ЭВМ (включая различия в используемых символьных кодах) был разработан ряд представлений знака числа. Поэтому архитектура ЭВМ семейства VAX поддерживает несколько методов представления знака числовой строки. Самое простое представление – это *числовая строка с раздельным ведущим знаком*. Название этого представления дано вследствие того, что код ASCII для знака плюс (^X2B) или знака минус (^X2D) заканчивается в отдельном байте перед цифрой старшего порядка числа. Например, ниже показано, как можно определить несколько числовых строк с раздельным ведущим знаком в языке ассемблера:

```
STRING_1:  .ASCII   "+1234"      ;5 БАЙТ, НО ДЛИНА ЧИСЛОВОЙ СТРОКИ 4
STRING_2:  .ASCII   "-12345678"  ;9 БАЙТ, НО ДЛИНА ЧИСЛОВОЙ СТРОКИ 8
STRING_3:  .ASCII   "+"          ;1 БАЙТ, НО ДЛИНА ЧИСЛОВОЙ СТРОКИ 0
STRING_4:  .ASCII   "-"          ;ПРЕДСТАВЛЕНИЕ "ОТРИЦАТЕЛЬНОГО НУЛЯ"
STRING_5:  .ASCII   "-0000"      ;ТОЖЕ, НО ДЛИНА ЧИСЛОВОЙ СТРОКИ 4
STRING_6:  .ASCII   " 1234"      ;АНАЛОГИЧНО STRING_1,
                                 ;НО С ПРОБЕЛОМ ДЛЯ ЗНАКА
```

Числовая строка с раздельным ведущим знаком задается двумя атрибутами: адресом начала строки (который в действительности является адресом знакового байта) и длиной строки. Однако длина определяет число цифр в строке, которое из-за знакового байта на 1 меньше числа байтов в строке. Как показано в шестой строке, знак плюс может быть заменен символом пробела (^X20). Однако использование знака "плюс" предпочтительнее.

Правила для числовых строк с раздельным ведущим знаком весьма строги. Например, каждое из следующих представлений недопустимо по указанной причине:

```
BAD_1:  .ASCII   "1234"      ;ОТСУТСТВУЕТ ЗНАК
BAD_2:  .ASCII   ""          ;ТРЕБУЕТСЯ ЗНАК
BAD_3:  .ASCII   " +1234"    ;НЕДОПУСТИМ ПРОБЕЛ ПЕРЕД ЗНАКОМ
BAD_4:  .ASCII   "+123,456"  ;В СТРОКЕ НЕДОПУСТИМА ЗАПЯТАЯ
BAD_5:  .ASCII   "-1234.56   ;ТОЧКА НЕДОПУСТИМА
```

Чтобы сохранить согласование с упакованными десятичными числами, длина числовой строки с раздельным ведущим знаком должна быть от 0 до 31 цифры (из-за знакового байта число байтов изменяется от 1 до 32).

На ЭВМ семейства VAX поддерживается второй формат для числовых строк, называемый *числовыми строками с ведомым знаком*. Этот формат был образован от кода Холлерита для перфокарт. Как отмечалось в гл. 9, перфокарта холлеритовского стандарта содержит 12 рядов, называемых соответственно рядами 12, 11, 0, 1, 2, . . . , 8 и 9. Пробивки в рядах 0 – 9 используются для представления цифр от 0 до 9, пробивка в ряду 12 обозначает знак "плюс", а пробивка в ряду 11 обозначает знак "минус". Вместо того чтобы отводить под знак отдельный столбец, знак включается в тот же столбец, где находится и цифра разряда единиц. Например, для представления числа –12345 в первых четырех столбцах делаются соответственно пробивки в рядах 1, 2, 3 и 4, а в пятом столбце делается две пробивки в рядах 5 и 11. Поскольку пробивка знака располагается в одном столбце с цифрой разряда единиц, это представление называют форматом с дополнительной пробивкой или *зонным форматом 2*.

Когда перфокарточные коды были расширены включением алфавитных символов, букве N был назначен перфокарточный код, состоящий из комбинации пробивок в рядах 11 и 5. Чтобы ввести число –12345, опытный оператор перфоратора может напечатать 1234N. Аналогично, поскольку код буквы E состоит из комбинации пробивок 12 и 5, число +12345 может быть введено на перфораторе как 1234E.

В результате следующие представления числовых строк с ведомым знаком допустимы в ЭВМ семейства VAX:

```
TRAIL_1: .ASCII    "1234N"        ; +12345 В ЗОННОМ ФОРМАТЕ 2
TRAIL_2: .ASCII    "1234E"        ; -12345 В ЗОННОМ ФОРМАТЕ 2
```

В этом формате последний байт содержит как знак, так и цифру разряда единиц. Левый полубайт байта определяет знак числа, а правый полубайт – цифру разряда единиц. Для большей совместимости с представлением символов в коде EBCDIC (см. гл. 8) ЭВМ семейства VAX поддерживают также второй формат для числовых строк с ведомым знаком, называемый *зонным числовым форматом* или *зонным форматом 1*. В этом формате для представления знаков "плюс" и "минус" просто используются различные битовые комбинации в левом полубайте последнего байта.

Чтобы справляться при обработке с различными способами кодирования знака и цифры разряда единиц в числовых строках с ведомым знаком, в инструкциях ЭВМ семейства VAX для обработки числовых строк с ведомым знаком используется 256-байтовая таблица для обработки байта, содержащего цифру разряда единиц и знак. За более подробной информацией по форматам числовых строк с ведомым знаком заинтересованного читателя отсылаем к книге "VAX Architecture Handbook". В оставшейся части раздела будут использоваться более легкие для понимания числовые строки с раздельным ведущим знаком.

ИНСТРУКЦИИ ПРЕОБРАЗОВАНИЯ ЧИСЛОВЫХ СТРОК

В архитектуре ЭВМ семейства VAX нет каких-либо инструкций для выполнения арифметических операций над числовыми строками. Однако в нее входят инструкции для преобразования числовых строк в упакованные десятичные числа и обратно. Это инструкции CVTPS (преобразовать упакованное десятичное в числовую строку с ведущим знаком) и CVTSP (преобразовать числовую строку с ведущим знаком в упакованное десятичное). Описание этих инструкций:

```
CVTPS       src_len, src_adr, dst_len, dst_adr
            R0=0     R1=a     R2=0      R3=a

CVTSP       src_len, src_adr, dst_len, dst_adr
            R0=0     R1=a     R2=0      R3=a

a - адрес байта, следующего за последним байтом строки
```

Биты N и Z устанавливаются в зависимости от знака и значения упакованного десятичного операнда. Бит V устанавливается, если происходит переполнение, а бит C всегда сбрасывается.

В архитектуру ЭВМ семейства VAX включены также инструкции для преобразования числовых строк с ведомым знаком. Инструкции CVTPT (преобразовать упакованное десятичное в числовую строку с ведомым знаком) и CVTTP (преобразовать числовую строку с ведомым знаком в упакованное десятичное) имеют аналогичный формат операндов и работают аналогично инструкциям CVTPS и CVTSP. Поскольку существует ряд способов представления знака для числовых строк с ведомым знаком инструкции CVTPT и CVTTP имеют дополнительный операнд. Этот операнд задает адрес 256-байтовой таблицы, которая определяет представление знака. (См. детали в книге "VAX Architecture Handbook".)

Заметим, что инструкция CVTSP будет завершаться по ошибке использования резервного операнда, если исходная строка содержит недопустимый символ. Исходная строка должна содержать код ASCII для знаков "плюс", "минус" или "пробел" в первом байте и коды ASCII для цифр 0 – 9 во всех последующих байтах. Если подлежащие преобразованию числа содержат ведущие пробелы, для их пропуска можно воспользоваться инструкцией SKPC.

ИНСТРУКЦИЯ EDITPC

Программист может иметь желание писать числа в самых разнообразных форматах. Предположим, например, что в поле упакованного десятичного числа содержится число 12345. Предположим далее, что это число должно быть преобразовано в символьную строку, содержащую десять байтов. Следующий список показывает различные возможные форматы получаемой в результате символьной строки. Символ подчеркивания (_) используется для обозначения символа пробела.

```
+000012345    _____12345    0000012345    ____+12345
$000012345    $____12345    ____$12345    $****12345
0012345 CR    0000123.45    ____123.45    ___+123.45
$***123.45    0123.45 CR    $___123.45    ___$123.45
```

Десятичные точки включены постольку, поскольку при представлении чисел с фиксированной точкой, описанной в гл. 10, подразумевается, что десятичная точка расположена где-то внутри упакованного десятичного числа.

Для формирования первой выведенной строки (+000012345) может быть использована инструкция CVTPS. Кроме того, для формирования различных символьных строк могут быть полезны различные инструкции преобразования кода символов. Однако проблема на самом деле сложнее, чем может показаться. Рассмотрим, например, формат, по которому образуется символьная строка _____123.45. Если число, подлежащее преобразованию в символьную строку, есть нуль, то результат _____0.00 предпочтительнее результата _____.____.

Чтобы упростить задачу преобразования упакованных десятичных чисел в символьные строки, в архитектуру ЭВМ семейства VAX включена инструкция EDITPC (отредактировать упакованное десятичное). Формат этой инструкции:

```
EDITPC      srclen,   srcadr, pattern, dstadr
            R0=длина  R1=a    R3=б     R5=в                    (R2=R4=0)

а - первый байт строки-источника
б - последний байт строки-источника или шаблона
в - первый байт, следующий за последним байтом строки получателя
```

Первые два операнда – это длина и адрес упакованного десятичного числа. Третий операнд – шаблон, представляет начальный адрес последовательности операторов шаблона редактирования. Каждый оператор шаблона редактирования – это один или два байта, показывающие, как должны располагаться символы в формируемой символьной строке. Последний операнд – это адрес начала символьной строки. Заметим, что длина символьной строки не является операндом. Так сделано потому, что длина упакованных десятичных чисел вместе с операторами шаблона редактирования однозначно определяют длину символьной строки – результата.

Точное кодирование операторов шаблона редактирования описано в книге "VAX Architecture Handbook". Для облегчения использования операторов шаблона редактирования в макробиблиотеке ЭВМ семейства VAX содержится набор макроинструкций для образования операторов шаблона. Рассмотрим, например, следующее:

```
PACKED:   .PACKED          0012345
PATTERN:
          EO$LOAD_FILL     #^A" "
          EO$MOVE          7
          EO$END
CHAR_7:   .BLKB            7
          . . .

          EDITPC           #7, PACKED, PATTERN, CHAR_7
```

Шаблон содержит три макроинструкции: EO$LOAD_FILL, EO$MOVE и EO$END. Сначала опишем действие макроинструкции EO$MOVE. Она образует оператор шаблона, называемый EO$MOVE, который приводит к тому, что некоторое число десятичных цифр извлекается из упакованного десятичного числа, преобразуется в символы кода ASCII и запоминается в целевой строке. В данном случае как аргумент указано число 7, так что все семь цифр из упакованного десятичного PACKED будут преобразованы. В результате можно ожидать, что в CHAR_7 будет запомнена строка 0012345.

Однако во многих случаях желательно подавить ведущие нули в начале числа. Чтобы это сделать, инструкция EDITPC использует однобитовую область памяти, называемую *битом значимости*. Пока извлекаемые из упакованного десятичного числа цифры остаются нулями, бит значимости сброшен. Но как только извлекается цифра, отличная от нуля, бит значимости устанавливается (устанавливается значимость) и остается установленным, даже если в последующем будут еще извлекаться нули (эти нули значащие, так как они следуют за ненулевой цифрой). Инструкция EDITPC использует также 8-битовый регистр, который содержит символ, называемый символом-заполнителем. Назначением макроинструкции EO$LOAD_FILL является загрузка символа-заполнителя в регистр заполнения.

Оператор EO$LOAD_FILL в предыдущем примере загружает в регистр заполнения символ пробела. (Операнд, указанный в макроинструкции, есть #^A" * ". Символ # задает непосредственный операнд. Префикс ^A показывает, что непосредственный операнд определяется как символ кода ASCII, а " " обозначает, что задан символ пробела.) Оператор EO$MOVE извлекает цифры из исходного упакованного десятичного числа так, как описано выше. Однако, если бит значимости не установлен (поскольку не была извлечена отличная от нуля цифра), цифры из исходной строки замещаются символом-заполнителем (в данном случае – пробелом). В результате после завершения выполнения, по адресу CHAR_7 будет находиться строка __12345. Заметим, что если внести изменение в оператор EO$LOAD_FILL #^A"*", то в результате будет образована строка **12345.

Если бы число PACKED было 0000000, бит значимости никогда бы не установился, символ-заполнитель заменил бы все семь цифр числа и результат был бы _____. Чтобы скорректировать это, следует изменить шаблон:

```
PATTERN:
        EO$LOAD_FILL    #^A" "      ;СИМВОЛ-ЗАПОЛНИТЕЛЬ ПРОБЕЛ
        EO$MOVE         6           ;ДЛИНА ПОЛЯ 6 РАЗРЯДОВ, В СВОБОДНЫЕ
                                    ;РАЗРЯДЫ ПОМЕЩАЕТСЯ ЗАПОЛНИТЕЛЬ
        EO$SET_SIGNIF               ;ВЫБИРАЕТСЯ НЕ НУЛЕВОЙ РАЗРЯД
        EO$MOVE         1           ;ОБЯЗАТЕЛЬНО ПРЕОБРАЗОВАТЬ СЕДЬМОЙ РАЗРЯД
        EO$END
```

Макроинструкция EO$SET_SIGNIF устанавливает бит значимости, даже если все цифры, извлеченные из исходной строки, были нулями. В результате этот шаблон дает такой же эффект, как и предыдущий, за исключением того, что нулевое исходное число преобразуется в _____0.

Для того чтобы позволить распечатывать знак числа, инструкция EDITPC использует второй 8-битовый регистр, предназначенный для хранения символа знака. Этот знаковый регистр подобен регистру заполнения, за исключением того, что для загрузки символа в знаковый регистр используются другие операторы шаблона. Макроинструкция EO$LOAD_SIGN осуществляет безусловную загрузку знакового регистра, макроинструкция EO$LOAD_PLUS загружает знаковый регистр тогда, и только тогда, когда упакованное десятичное число положительное, а макроинструкция EO$LOAD_MINUS загружает знаковый регистр, если упакованное десятичное отрицательно. Оператор EO$STORE_SIGN используется для вставки знака числа в формируемую строку. Рассмотрим, например, следующее:


```
PACKED: .PACKED 0012345
PATTERN:
        EO$LOAD_FILL    #^A" "      ;СИМВОЛ-ЗАПОЛНИТЕЛЬ ПРОБЕЛ
        EO$LOAD_PLUS    #^A"+"      ;ЕСЛИ ДЕСЯТИЧНОЕ > 0, ТО ДОБАВИТЬ ЗНАК "+"
        EO$LOAD_MINUS   #^A"-"      ;ЕСЛИ ДЕСЯТИЧНОЕ < 0, ТО ДОБАВИТЬ ЗНАК "-"
        EO$STORE_SIGN               ;ЗНАКОВЫЙ РАЗРЯД ОБЯЗАТЕЛЕН
        EO$MOVE         6           ;ДЛИНА ПОЛЯ 6 РАЗРЯДОВ, В СВОБОДНЫЕ
                                    ;РАЗРЯДЫ ПОМЕЩАЕТСЯ ЗАПОЛНИТЕЛЬ
        EO$SET_SIGNIF               ;УСТАНОВИТЬ ЗНАЧИМОСТЬ, ЕСЛИ ОНА ОТСУТСТВУЕТ
        EO$MOVE         1           ;ОБЯЗАТЕЛЬНО ПРЕОБРАЗОВАТЬ ПОСЛЕДНИЙ РАЗРЯД
        EO$END
CHAR_7: .BLKB           8
        . . .

        EDITPC          #7,PACKED,PATTERN,CHAR_7
```

Будет порождена строка +____12345. Заметим, что если бы упакованное десятичное число было нулем, результат был бы +____0.

Макроинструкции EO$FLOAT и EO$END_FLOAT используются для перемещения знака вправо так, чтобы он оказался перед первой значащей цифрой числа. Макроинструкция EO$FLOAT – такая же, как и EO$MOVE, за исключением того, что когда обнаруживается первая значащая цифра, то перед цифрой помещается знак числа. Для обработки случая, когда не найдено ни одной значащей цифры, последовательность из одного или нескольких операторов EO$FLOAT должна завершаться одним оператором EO$END_FLOAT, который помещает знак числа в строку, только, если знак не был уже помещен в строку оператором EO$FLOAT. Например, если предыдущую строку шаблона заменить на следующую строку, будет сформирована строка ____+12345. Если упакованное число было нулевым, сформировано будет _____+0.

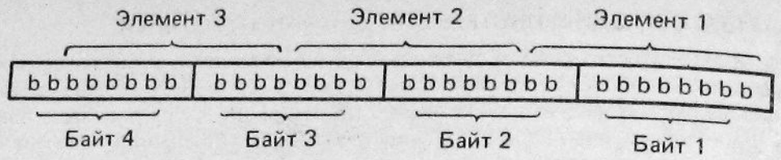
```
        EO$LOAD_FILL    #^A" "      ;СИМВОЛ-ЗАПОЛНИТЕЛЬ ПРОБЕЛ
        EO$LOAD_PLUS    #^A"+"      ;ЕСЛИ ДЕСЯТИЧНОЕ > 0, ТО ДОБАВИТЬ ЗНАК "+"
        EO$LOAD_MINUS   #^A"-"      ;ЕСЛИ ДЕСЯТИЧНОЕ < 0, ТО ДОБАВИТЬ ЗНАК "-"
        EO$FLOAT        6           ;ДЛИНА ПОЛЯ 6 РАЗРЯДОВ, В СВОБОДНЫЕ
                                    ;РАЗРЯДЫ ПОМЕЩАЕТСЯ ЗАПОЛНИТЕЛЬ
        EO$END_FLOAT    6           ;ВЫВЕСТИ ЗНАК, ЕСЛИ ТРЕБУЕТСЯ
        EO$SET_SIGNIF               ;УСТАНОВИТЬ ЗНАЧИМОСТЬ, ЕСЛИ ОНА ОТСУТСТВУЕТ
        EO$MOVE         1           ;ОБЯЗАТЕЛЬНО ПРЕОБРАЗОВАТЬ ПОСЛЕДНИЙ РАЗРЯД
        EO$END
```

Оператор шаблона EO$INSERT может использоваться для занесения в формируемую строку какого-то конкретного символа (операндом является вставляемый символ). Таким образом, оператор EO$INSERT может быть использован для занесения в выходную строку запятых (1,234,567), десятичных точек (12345.67) или того и другого (12,345.67). Если не обнаружено значащих цифр, этот оператор вставляет символ-заполнитель, так что нуль может быть распечатан без ненужных запятых (_____0, а не __,___0). (Если оператору EO$INSERT предшествует оператор EO$SET_SIGNIF, то символ-заполнитель будет вставлен безусловно.) Дополнительную информацию по этим и другим операторам шаблона см. в книге "VAX Architecture Handbook".

13.8. БИТОВЫЕ ПОЛЯ ПЕРЕМЕННОЙ ДЛИНЫ

Иногда желательно адресовать группы битов, которые не обязательно начинаются с границы байта. Предположим, например, что телевизионное изображение высокого разрешения преобразуется в цифровую форму в виде матрицы 1000 на 1000 элементов, где каждый элемент матрицы – это 10-битовое число, показывающее уровень яркости в данной точке. Каждое 10-битовое число может быть представлено в 16-битовом слове, но при этом остались бы неиспользованными 6 битов. Однако неиспользованные биты составляют 6 млн. битов, или 750 тыс. байтов потерянной памяти.

Чтобы избежать потери битов, 10-битовые числа могут быть упакованы вместе следующим образом:



Выборку и занесение 10-битовых полей с помощью описанных до сих пор инструкций делать неудобно.

Для решения этой и других проблем в архитектуру ЭВМ семейства VAX включен тип данных, называемый *битовым полем переменной длины.* Битовое поле задается тремя атрибутами:

1. Адрес байта, который определяет адрес начала области памяти, используемой для битовых полей (адрес байта 1 в примере).
2. Битовое смещение, которое определяет интервал в битах между адресом начала области памяти и адресом начала битового поля (0 для элемента 1, 10 для элемента 2, 20 для элемента 3 и т. д. в примере).
3. Длина поля, которая определяет длину битового поля в битах (10 в примере).

Начальный адрес таблицы может быть задан любым режимом адресации, который дает 32-битовый адрес (кроме того, может использоваться регистровая адресация). Это означает, что группа битов может находиться в одном из регистров. Битовое смещение задается в формате 32-битового длинного слова как целое со знаком. Заметим, что это смещение можно представлять как 29-битовое байтовое смещение, определяющее байт, в котором начинается битовое поле, за которым следует 3-битовое число, задающее местоположение этого поля внутри байта. Длина поля – это целое без знака в формате байта, значение которого находится в диапазоне 0 – 32. Если поле находится в регистре общего назначения, то значение смещения битового поля должно быть в диапазоне 0 – 31.

В архитектуру ЭВМ семейства VAX включены инструкции для извлечения (выборки), сравнения и вставки (занесения) битовых полей. Форматы этих инструкций:

```
EXTV    disp,len,start,longdst
CMPV    disp,len,start,longsrc
INSV    longsrc,disp,len,start
```

В полях операндов обозначение start определяет адрес начала блока памяти, используемого для хранения битовых полей, disp определяет 32-битовое смещение, а len – это байтовый операнд, определяющий длину поля. Инструкция EXTV извлекает группу битов и помещает ее в длинное слово (longdst), причем младший бит группы попадает в младший бит длинного слова. Битовое поле рассматривается как число со знаком, так что значение старшего бита поля распространяется по старшим битам длинного слова (альтернативная инструкция EXTZV вместо знакового бита распространяет нуль и поэтому используется для битовых полей, интерпретируемых как числа без знака). Аналогичным образом инструкция CMPV расширяет битовое поле, приводя его к формату длинного слова, перед сравнением с содержимым длинного слова (инструкция CMPZV распространяет нуль вместо знакового бита). Биты кодов условий устанавливаются так, как и ожидается, за исключением того, что инструкция INSV оставляет коды условий без изменения.

Работа этих инструкций очевидна. В следующем примере инструкция EXTV извлекает биты с 20-й по 25-й из регистра R2 и помещает результат с распространенным значением знакового разряда в регистр R4. Инструкция INSV помещает 6-битовую строку в битовые позиции 15 – 20 регистра R10.

```
EXTV    #20,#6,R2,R4
INSV    R4,#15,#6,R10
```


13.9. ОЧЕРЕДИ И ИНСТРУКЦИИ ДЛЯ РАБОТЫ С НИМИ

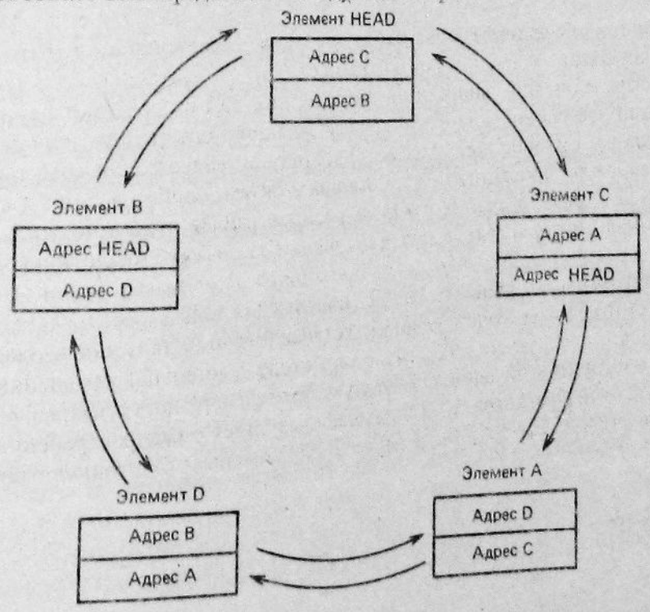
Архитектура ЭВМ семейства VAX поддерживает структуры данных, называемых *абсолютными очередями*. Очередь – это разновидность двунаправленного списка, в котором каждый элемент очереди содержит указатель на предыдущий элемент в этом списке, а также указатель на последующий элемент списка. В очереди последний элемент списка указывает на первый элемент списка, формируя тем самым циклическую структуру.

Рассмотрим, например, элементы HEAD, A, B, C и D.

```
HEAD:   .ADDRESS   C      ;ССЫЛКА ВПЕРЕД НА ПОСЛЕДУЮЩИЙ ЭЛЕМЕНТ
        .ADDRESS   B      ;ССЫЛКА НАЗАД НА ПРЕДЫДУЩИЙ ЭЛЕМЕНТ
        . . .

A:      .ADDRESS   D      ;ССЫЛКА ВПЕРЕД НА ПОСЛЕДУЮЩИЙ ЭЛЕМЕНТ
        .ADDRESS   C      ;ССЫЛКА НАЗАД НА ПРЕДЫДУЩИЙ ЭЛЕМЕНТ
        . . .

B:      .ADDRESS   HEAD   ;ССЫЛКА ВПЕРЕД НА ПОСЛЕДУЮЩИЙ ЭЛЕМЕНТ
        .ADDRESS   D      ;ССЫЛКА НАЗАД НА ПРЕДЫДУЩИЙ ЭЛЕМЕНТ
        . . .

C:      .ADDRESS   A      ;ССЫЛКА ВПЕРЕД НА ПОСЛЕДУЮЩИЙ ЭЛЕМЕНТ
        .ADDRESS   HEAD   ;ССЫЛКА НАЗАД НА ПРЕДЫДУЩИЙ ЭЛЕМЕНТ
        . . .

D:      .ADDRESS   B      ;ССЫЛКА ВПЕРЕД НА ПОСЛЕДУЮЩИЙ ЭЛЕМЕНТ
        .ADDRESS   A      ;ССЫЛКА НАЗАД НА ПРЕДЫДУЩИЙ ЭЛЕМЕНТ
```

Каждый элемент начинается с двух адресов, заданных в формате длинного слова (любые данные, относящиеся к этому элементу, могут располагаться вслед за этими двумя адресами). Первый адрес в каждой паре – это указатель на последующий элемент. Поскольку адрес HEAD содержит адрес C, элемент C следует за элементом HEAD. Аналогично адрес C содержит адрес следующего элемента A, так что элемент A следует за элементом C. В прямом направлении порядок следования элементов таков: HEAD, C, A, D, B и затем – обратно HEAD. Второй адрес в каждом элементе указывает на предыдущий элемент. Поскольку по адресу HEAD+4 содержится адрес предшествующего элемента, элемент B предшествует элементу HEAD. В обратном направлении порядок такой: B, D, A, C, HEAD.

Эта структура может быть представлена следующим образом:



Набор инструкций ЭВМ семейства VAX содержит инструкции INSQUE (вставить элемент в очередь) и REMQUE (удалить элемент из очереди). Описание этих инструкций:

```
INSQUE    elemadr, headadr
REMQUE    headadr, elemadr
```

Например, инструкция INSQUE A, HEAD вставит элемент A как следующий за элементом HEAD (элемент, следовавший ранее за элементом HEAD, будет следовать за элементом A). Инструкция REMQUE HEAD, E удалит элемент HEAD из очереди и поместит адрес элемента в длинное слово E. Например, показанная выше структура из пяти элементов может быть создана так:

```
HEAD:   .ADDRESS   HEAD
        .ADDRESS   HEAD
        . . .

A:      .BLKL      2
        . . .
B:      .BLKL      2
        . . .
C:      .BLKL      2
        . . .
D:      .BLKL      2
        . . .

        INSQUE     B, HEAD      ;ОЧЕРЕДЬ СОДЕРЖИТ HEAD, B
        INSQUE     D, HEAD      ;ОЧЕРЕДЬ СОДЕРЖИТ HEAD, D, B
        INSQUE     A, HEAD      ;ОЧЕРЕДЬ СОДЕРЖИТ HEAD, A, D, B
        INSQUE     C, HEAD      ;ОЧЕРЕДЬ СОДЕРЖИТ HEAD, C, A, D, B
```

Обратите внимание, что пустая очередь состоит из одного головного элемента (HEAD), в котором оба адреса указывают на головной элемент. Структура из пяти элементов HEAD, C, A, D, B обычно рассматривается как имеющая головной элемент и четыре элемента данных.

Хотя головной элемент рассматривается как особый элемент, он имеет ту же структуру, что и другие элементы. Поэтому инструкция INSQUE может использоваться для вставки элемента после любого заданного элемента при условии, что доступен адрес данного элемента. Например, в структуре HEAD, C, A, D, B адрес HEAD+4 содержит адрес B. Как описано в гл. 7, операнд @HEAD+4 определяет адрес, содержащийся в HEAD+4. В результате этого инструкция

```
INSQUE    E, @HEAD+4
```

рассматривает элемент B как головной элемент списка и вставляет элемент E между элементами B и HEAD, порождая структуру HEAD, C, A, D, B, E.

Аналогичные методы адресации могут применяться и с инструкцией REMQUE. Например:

```
REMQUE    @HEAD, R0        ;УДАЛИТЬ ЭЛЕМЕНТ СЛЕДУЮЩИЙ ЗА HEAD
REMQUE    @HEAD+4, R0      ;УДАЛИТЬ ЭЛЕМЕНТ ПРЕДШЕСТВУЮЩИЙ HEAD
```

Очередь – это такая структура данных, в которой первый добавленный элемент первым же и удаляется (используется стратегия FIFO – "первым вошел – первым вышел"). Очереди легко реализуются с помощью инструкций INSQUE item, HEAD и REMQUE HEAD+4, dist, которые вставляют элементы после головного элемента и удаляют элементы, предшествующие головному элементу. Очередь противоположна стеку, в котором используется другая организация (LIFO – "последний вошел, первым вышел"). Однако реализация очередей в ЭВМ VAX дает более мощную структуру данных, чем обычная очередь (или

стек), поскольку она основана на двунаправленном списке. В результате этого включенные в структуру элементы могут быть произвольно расположены в памяти, данные, связанные с каждым элементом, могут иметь любую длину и легко могут быть изменены, а элементы могут легко вставляться или удаляться из любого места структуры данных.

Кроме абсолютных очередей ЭВМ семейства VAX поддерживают *относительные очереди*, в которых указатели являются относительными, а не абсолютными адресами. Имеются инструкции для вставки или удаления элементов, предшествующих или следующих за головным элементом очереди. Дополнительные подробности см. в книге "VAX Architecture Handbook".

УПРАЖНЕНИЯ 13.2

1. Используя формулу Ньютона-Рафсона, описанную в гл. 12, вычислите корень квадратный из числа 2 с точностью не меньше 25 десятичных знаков. Однако вычисления должны выполняться, с использованием упакованных десятичных чисел, а не числа с плавающей точкой. Используйте инструкцию CVTPS для распечатки последовательных приближений.

2. В разд. 13.7 приведен фрагмент программы, в котором число со знаком в формате квадраслова преобразуется в 20-значное упакованное десятичное число. Покажите, как этот алгоритм работает с каждым из следующих квадраслов:

а). ^X0000001000000001 б). ^X0000000080000000
в). ^X8000000000000000 г). ^X8000000080000000
д). ^X0000000180000001 е). ^X8000000180000001
ж). ^XC0000000C0000000 з). ^XC0000001C0000001

3. Напишите подпрограмму, которая преобразует целое число со знаком в формате длинного слова в числовую строку с раздельным ведущим знаком. Строка должна иметь длину 10, а это означает, что она занимает 11 байтов. (Подсказка: Используйте инструкции CVTLP и CVTPS.)

4. Модифицируйте подпрограмму из п. 3 так, чтобы подавлялись ведущие нули, а знак перемещался вправо таким образом, чтобы он оказывался перед первой значащей цифрой (положительный нуль должен распечатываться так: _____+0).

5. Напишите подпрограмму, которая распечатывает число F_FLOAT по формату F10.0. Фортрана (число в формате F10.0 состоит из знака, шести десятичных цифр, десятичной точки и двух цифр после десятичной точки). Однако вместо знака "плюс" используется пробел, ведущие нули подавляются, а знак переносится к первой значащей цифре. (Подсказка: умножьте число F_FLOAT на 100 и затем используйте инструкции CVTFL, CVTLP и EDITPC.)

6. Напишите программу, которая считывает числа, представленные как символьные строки длиной 20, и преобразует число в формат G с плавающей точкой. Символьная строка начинается со знака "плюс" или "минус" и содержит десятичную точку, вставленную где-то между 18 десятичными цифрами (например, +00000000012345.6789). (Подсказка: удалите десятичную точку и разбейте символьную строку на две 9-значные числовые строки с раздельным ведущим знаком. Преобразуйте в два упакованных десятичных числа с помощью инструкции CVTSP, а затем преобразуйте в два целых числа в формате длинного слова с помощью инструкции CVTPL. Используйте инструкцию CVTLG, применив подходящее масштабирование, для преобразования длинных слов в формат G с плавающей точкой.)

7. Как в п. 3 из упр. 13.1, напишите программу, которая составляет алфавитный указатель слов заданного текста. Однако в вашей программе должна использоваться такая структура данных, в которой каждое слово текста представляется элементом очереди.

ГЛАВА 14. ЭЛЕМЕНТЫ АРХИТЕКТУРЫ ЭВМ СЕМЕЙСТВА VAX

14.1. ВВЕДЕНИЕ

Под *архитектурой* вычислительной машины понимается ее организация с точки зрения программиста. Понятие архитектуры включает в себя такие особенности организации ЭВМ, как набор инструкций, типы данных, режимы адресации и пр. Специфическое устройство элементов аппаратуры, которые реализуют эти особенности, называется *исполнением* вычислительной машины. Любая заданная архитектура может быть реализована различными способами. Программа, которая может быть выполнена вычислительной машиной с одним исполнением данной архитектуры, должна выполняться вычислительными машинами с любым другим исполнением этой архитектуры.

Например, ЭВМ семейства VAX, такие как VAX-11/730, VAX-11/750, VAX-11/780 и др., представляют разные исполнения архитектуры VAX. Различные модели ЭВМ семейства VAX отличаются по стоимости и производительности. С изменением технологии появляются новые исполнения архитектуры VAX. Поскольку новые модели ЭВМ будут поддерживать ту же архитектуру, то на этих ЭВМ можно будет выполнять программы, разработанные для более ранних моделей.

В предыдущих главах были описаны те части архитектуры VAX, с которыми приходится сталкиваться при написании обычных пользовательских программ. В данной главе рассматриваются особенности архитектуры, используемые операционной системой. Кроме того, будет рассмотрена реализация некоторых элементов архитектуры. В главе обсуждаются такие понятия, как виртуальная память, кэш-память, системные привилегии, а также структура системных интерфейсов и физический уровень ввода-вывода.

14.2. УПРАВЛЕНИЕ ПАМЯТЬЮ И ВИРТУАЛЬНАЯ ПАМЯТЬ

ВВЕДЕНИЕ

С первых дней существования вычислительной техники память вычислительных машин разделялась на память с высоким и низким быстродействием. Наличие двух типов памяти объясняется тем, что память с низким быстродействием может быть на несколько порядков дешевле, чем быстродействующая память. С развитием технологии происходит замена памяти с низким быстродействием на новые варианты ее исполнения, обладающие более высоким быстродействием. В настоящее время в большинстве вычислительных машин в качестве быстродействующей памяти применяется полупроводниковая память, а в качестве памяти с низким быстродействием – дисковые запоминающие устройства. Типичная вычислительная система на базе ЭВМ VAX-11/780 может иметь 8 Мбайт быстродействующей полупроводниковой памяти и 1000 Мбайт более медленной дисковой памяти. (С введением технологических улучшений оба типа памяти становятся все более дешевыми, поэтому объем памяти для типичных вычислительных систем будет увеличиваться.)

Как правило, быстродействующая, или *основная память* используется для запоминания работающих программ и данных, к которым эти программы активно обращаются. Более медленная, или *внешняя память большой емкости* применяется для хранения *файлов* программ и данных, сохраняемых для дальнейшего использования. Встречаются две трудности, усложняющие эту простую схему распределения памяти.

Первая состоит в том, что часто объем памяти, требуемый для больших задач, превышает имеющийся в наличии объем основной памяти. Одним из первых решений этой проблемы была организация *оверлейных программ*. Это означает, что написанные специальным образом программы располагаются во внешней памяти и по частям загружаются в основную память для выполнения. После завершения выполнения одной программной секции в основную память вызывается другая программная секция, которая перекрывает область основной памяти, использованную первой секцией. Для повышения эффективности такого процесса программист должен тщательно разделить программу на отдельные секции, которые могли бы успешно выполняться, не требуя многочисленных обменов какого-либо рода между секциями. Некоторые первые попытки автоматизировать процесс организации перекрытия привели к созданию того, что теперь получило название *виртуальная память*.

Вторая трудность возникает при использовании вычислительных машин в мультипрограммном режиме. Для повышения эффективности работы вычислительной системы желательно иметь не одну, а несколько программ, одновременно загруженных в память. Если одна из программ находится в состоянии ожидания, что случается при выполнении довольно медленных операций ввода-вывода, то в это время может выполняться другая программа, время простоя процессора при этом сокращается. Например, устройство построчной печати может распечатывать 600 строк в минуту, или 10 строк в секунду. Пока одна программа ожидает завершения печати очередной строки, в другой программе может быть выполнено 100 тыс. и более инструкций. При осуществлении ввода-вывода с терминала в интерактивном режиме работы время ожидания еще больше, и, пока пользователь решает, какой символ ему напечатать следующим, могут быть выполнены миллионы инструкций. Такое одновременное выполнение программ различных пользователей называется *режимом разделения времени*. Поэтому терминалы вычислительной системы, используемые в интерактивном режиме, часто называют терминалами, работающими в режиме разделения времени.

Как было рассмотрено в предыдущих главах, обычная программа пользователя занимает некоторое множество ячеек памяти. На ЭВМ семейства VAX программы обычно компонуются так, что располагаются в памяти, начиная с адреса ^X200. Если бы адреса программы являлись адресами реальной (физической) памяти, то возникла бы проблема. Невозможно иметь две разные программы, занимающие одну и ту же область памяти одновременно. Для решения этого на ЭВМ семейства VAX используется некая система отображения, позволяющая отображать адреса программы пользователя в некоторые другие адреса реальной памяти. Например, адреса программы ^X200, ^X201, ^X202, . . . могли быть отображены в адреса ^X3C600, ^X3C601, ^X3C602, . . . физической памяти ЭВМ семейства VAX. Поскольку программа другого пользователя могла бы иметь адреса ^X200, . . . отображенные в другие адреса памяти ЭВМ семейства VAX, то здесь конфликта не возникает.

Алгоритм отображения адреса может быть достаточно простым. Один из способов состоит в использовании регистра смещения, который содержит значение смещения. Всякий раз при обращении программы к адресу значение смещения автоматически прибавляется к этому адресу для вычисления исполнительного адреса. В приведенном выше примере в регистре смещения могло бы находиться значение ^X3C400. Поэтому если программа обращается к адресу ^X202, то к нему прибавляется смещение ^X3C400 и производится выбор исполнительного адреса ^X3C602. Это сложение выполняется автоматически, поэтому в программе пользователя не требуется специально выполнять какие-либо действия. Что касается программы, то она запросила содержимое ячейки с адресом ^X202, а затем и получила содержимое ячейки с адресом ^X202. Для программы адрес ^X202 является явным "видимым" адресом. Такие адреса называются *виртуальными*, подобно *виртуальному изображению*, которое мы видим в зеркале. Адрес ^X202 – виртуальный адрес, а адрес ^X3C602 – *реальный* или *абсолютный* (физический) адрес. При выполнении другой программы в регистре смещения будет находиться другое значение, поэтому различным будет и отображение виртуальных адресов в реальные адреса.

Некоторые устаревшие вычислительные системы имели только один регистр смещения. Таким образом, все адреса программы отображались в некоторую непрерывную область реальной памяти. Существует проблема, связанная с тем, что не все программы занимают одинаковый объем памяти, а при завершении одной программы ее место в памяти должно быть занято другой программой. В результате между программами могут образоваться неиспользуемые участки памяти. Это получило название *фрагментации памяти*. Одним из решений является *страничная организация памяти*. Например, память ЭВМ семейства VAX разбивается на страницы по 512 байтов. Для каждой страницы имеется регистр смещения. Каждая страница (или 512 байтов) виртуального адресного пространства программы мо-

жет быть отображена в некоторую страницу реальной памяти. Страницы в реальной памяти размещаются необязательно непрерывно; размещение страниц в памяти осуществляется операционной системой с помощью специального алгоритма нахождения свободных страниц.

В добавление к проблемам отображения адресов и фрагментации памяти существует еще проблема защиты памяти. Если несколько пользователей работают на ЭВМ в режиме разделения времени, решая свои несвязанные с другими пользователями задачи[1], то должен быть обеспечен способ защиты информации, находящейся в области памяти, выделенной одному пользователю, от попыток со стороны другого пользователя изменить или даже просто получить доступ к этой информации. Кроме того, должна быть защищена от изменения и/или доступа со стороны всех пользователей область памяти, зарезервированная для операционной системы.

Целиком весь процесс, включающий распределение памяти, отображение виртуальных адресов программы и реализацию защиты памяти, объединен под общим названием – *управление памятью*. В управлении памятью участвуют как технические, так и программные средства. Распределение памяти и загрузка программ в память требуют применения сложных алгоритмов и являются в основном функциями, реализованными программно. Для того чтобы программа могла выполняться с максимальной скоростью, процесс отображения адресов реализуется с помощью специальной аппаратуры. К техническим средствам системы управления памятью относятся такие устройства, как регистр или регистры смещения, а также сумматор, автоматически складывающий содержимое регистра смещения с каждым виртуальным адресом, формируемым программой. Кроме того, технические средства системы управления памятью обеспечивают защиту памяти, ограничивая или предотвращая доступ пользователей к определенным страницам.

ВИРТУАЛЬНАЯ ПАМЯТЬ

Для того чтобы рассматриваемая система управления памятью была полной системой *виртуальной памяти*, необходима еще и автоматическая смена страниц. В предыдущем разделе обсуждалось, как большие программы с оверлейной организацией могут по частям загружаться в память, перекрывая выделенную область памяти, или выгружаться из памяти, когда это необходимо. Также рассматривалось, как программа может разделяться на страницы, причем каждой странице отводится свой собственный регистр смещения. Эти два способа сочетаются в системе виртуальной памяти.

Для операционной системы VAX/VMS вместе со всеми выполняемыми в определенный момент времени программами пользователей требуется набор страниц памяти. Этот набор страниц можно представить себе как одну очень большую программу. В многопользовательской вычислительной системе обычно не хватает реальной памяти, чтобы одновременно загрузить в память все страницы. В результате необходим аппарат перекрытия или смены страниц, чтобы пользователи могли получать в основную память страницы своих программ.

В какой-то момент времени некоторые страницы набора могут на самом деле находиться в быстродействующей памяти. Ссылки к этим страницам преобразуются с помощью регистров смещения, описанных ранее. Страницы, не поместившиеся в памяти, размещаются на таких внешних запоминающих устройствах большой емкости, как магнитные диски. Если программа обращается к такой странице, то система управления памятью должна загрузить в реальную память страницу и занести в регистр смещения страницы соответствующее значение. Для того чтобы найти неиспользуемую страницу в основной памяти, системе управления памятью может потребоваться переслать некоторые другие страницы

[1] Иногда пользователи, работая в режиме разделения времени, занимаются решением одной общей задачи, такой, например, как резервирование авиабилетов. В этих случаях между программами осуществляется управляемое взаимодействие.

из основной памяти на внешнее запоминающее устройство большой емкости. Таким образом система виртуальной памяти обеспечивает смену страниц.

Например, программа во время выполнения обращается к различным ячейкам памяти. Если адреса этих ячеек относятся к странице, которая в текущий момент расположена в памяти, то производятся выборка или занесение информации по адресу, вычисляемому с помощью регистра смещения. Если требуемая страница отсутствует в памяти, то выполнение операций выборки или занесения прерывается. Эта ситуация называется *прерыванием по отсутствию страницы* (page fault). Когда происходит прерывание по отсутствию страницы, система управления памятью производит определенную последовательность действий для загрузки соответствующей страницы в память и инициализации ее регистра смещения, чтобы можно было адресовать ячейки этой страницы.

При возникновении прерывания по отсутствию страницы соответствующая страница должна быть считана с внешнего запоминающего устройства большой емкости и помещена в основную, или *реальную*, память. Если в реальной памяти еще имеются неиспользуемые страницы, то может быть назначена одна из этих страниц, но при большой загрузке вычислительной машины, по всей вероятности, неиспользуемых страниц в основной, или реальной, памяти может и не быть. Это означает, что система управления памятью должна найти страницу и предоставить ее программе. Чтобы сделать это, операционная система применяет некоторый алгоритм для выбора страницы, которая, вероятно, не потребуется в ближайшее время. Выбранная страница выделяется программе. Однако, перед тем как выделить страницу, операционная система сохраняет старое содержимое страницы на внешнем запоминающем устройстве большой емкости, так что сохраненная страница несколько позже может быть возвращена обратно той программе, к которой она относится. Затем с внешнего запоминающего устройства большой емкости считывается требуемая страница и помещается на выделенную страницу памяти. Соответствующий регистр смещения инициализируется так, чтобы можно было адресовать эту страницу, и страница помечается как доступная. После этого управление возвращается программе, которая была прервана по отсутствию страницы.

Рассмотрим специфический пример, иллюстрирующий работу виртуальной памяти при выполнении программы. При загрузке программы загрузчик определяет, сколько памяти требуется для программы[1]. Если диапазон адресов программы ограничен ^X200 – ^X21FF, то программа занимает ^X2000 = 8192 байтов, или ^X10, иначе – 16 страниц памяти. Отметим, что это является распределением *виртуальной* памяти, поскольку не гарантируется какое-либо частное распределение страниц реальной памяти. В связи с тем, что виртуальные страницы могут пересылаться в реальную память и обратно, на запоминающем устройстве большой емкости сохраняется самая последняя копия содержимого 16 страниц.

Предположим, что во время выполнения программы пользователь попытался обратиться к содержимому ячейки памяти с адресом ^X8C4. Возможно, что в программе выполняется инструкция MOVL A,R0 и оказалось, что символическому адресу A соответствует адрес ^X8C4. Предположим далее, что страница, содержащая ячейку с адресом ^X8C4, была выгружена из памяти на внешнее запоминающее устройство большой емкости или страница просто еще не была загружена. В любом случае в реальной памяти нет страницы, назначенной для виртуальных адресов в диапазоне ^X800 – ^X9FF. В итоге при выполнении инструкции MOVL A,R0 произойдет прерывание по отсутствию страницы, для обработки которого управление будет передано операционной системе.

Поскольку адрес ^X8C4 является допустимым для данной программы, операционная система должна найти страницу реальной памяти. Операционная система с помощью какого-то алгоритма (алгоритм будет рассмотрен позже) может принять решение, что страница

[1] В ОС VAX/VMS процесс загрузки программы тесно связан с реализацией виртуальной памяти. Поэтому программа, которую можно было бы назвать загрузчиком, называется *активатором образа* (см. гл. 15).

реальной памяти, начинающаяся с адреса ^X000A3C00, подходит для того, чтобы выделить ее программе. Весьма вероятно, что страница уже используется, может быть данной программой, или может быть какой-либо другой.

Предположим, что страница реальной памяти, начинающаяся с адреса ^X0003AC00, была назначена виртуальной странице некоторой программы. Тогда перед тем, как страница реальной памяти может быть использована, ее содержимое должно быть сохранено на внешнем запоминающем устройстве большой емкости. Как только старое содержимое страницы сохранится, страница памяти становится доступной.

Для того чтобы обработка прерывания по отсутствию страницы была завершена, требуемая информация должна быть загружена с внешнего запоминающего устройства большой емкости на доступную теперь страницу памяти. В данном примере страница информации, находящаяся на внешнем запоминающем устройстве большой емкости, могла быть скопирована на страницу памяти, начинающуюся с адреса ^X000A3C00. Регистр смещения страницы для данной программы устанавливается так, чтобы отобразить виртуальную страницу с адресом ^X800 на страницу реальной памяти, начинающуюся с адреса ^X000A3C000. В то же время регистр смещения страницы для предыдущей программы должен быть установлен так, чтобы показать, что страница предыдущей программы больше не существует в реальной памяти. Операционная система может затем вернуть управление рассматриваемой программе, и инструкция MOVL A, R0 может быть выполнена успешно.

Не приходится и говорить, что процесс обмена страниц может стать достаточно сложным. Рассмотрим, например, такую инструкцию, как ADDL3 A, B, C. Для выполнения этой инструкции процессор должен обратиться к четырем данным – к самой инструкции, а также ее операндам A, B и C. Поскольку все эти четыре элемента могут находиться на несуществующей в реальной памяти странице, во время выполнения инструкции может произойти ряд прерываний по отсутствию страницы.

В действительности ситуация сложнее, чем кажется. Например, если операнд A, имеющий формат длинного слова, расположен по виртуальному адресу ^X9FE, то одна половина длинного слова находится на одной странице, а его вторая половина – на другой. В результате обращение к операнду A может в худшем случае вызвать два прерывания по отсутствию страницы. Аналогичная ситуация возникает и когда сама инструкция пересекает границу страницы. Прерывание по отсутствию страницы может произойти при выборке инструкции. В любом случае создание системы виртуальной памяти, работающей корректно и эффективно, – чрезвычайно сложная задача, включающая разработку технического и программного обеспечения.

АЛГОРИТМЫ УПРАВЛЕНИЯ СТРАНИЦАМИ

В предыдущем разделе был упомянут алгоритм для выбора страницы, выгружаемой на внешнее запоминающее устройство большой емкости при возникновении прерывания по отсутствию страницы. Когда принимается решение о том, какая страница должна быть выгружена, важно выбрать страницу, которая вряд ли потребуется сразу. Неудачный подбор страниц для выгрузки может привести к тому, что при выполнении каждой инструкции будет происходить прерывание по отсутствию страницы. Эта ситуация может отразиться на выполнении программы; выполнение будет настолько медленным, как если бы внешнее запоминающее устройство большой емкости выступало в роли основной памяти.

Идеальным выбором страницы, предназначенной для выгрузки, могла быть страница памяти, которая не понадобится еще долгое время. К сожалению, это невозможно предсказать в масштабе реального времени. Поэтому алгоритмы базируются на эвристических методах. Одним из таких алгоритмов является алгоритм выбора дольше всего не использовавшейся страницы. Предположение строится на том, что если страница давно не использовалась, то вряд ли она потребуется в ближайшее время. Могут быть случаи, когда это предложение не оправдывается, но в среднем посылка правильная. Однако этот алгоритм весьма дорогой как с точки зрения технических средств, так и программного обеспечения.

Для определения того, какая страница давно не использовалась, может потребоваться, чтобы каждый регистр страницы имел таймер, автоматически устанавливающийся при очередном обращении к странице. Кроме того, должен осуществляться поиск регистра страницы с наиболее ранней установкой таймера.

Один простой алгоритм для выбора выгружаемой страницы состоит в том, что производится случайный выбор. Другой простой алгоритм основан на последовательном выборе страниц. Практически простой алгоритм может быть не намного хуже более сложного и дорогого алгоритма. (Следует отметить, что простые алгоритмы могут быть улучшены, если повысить вероятность предпочтительного выбора страниц программ с меньшим приоритетом.)

КЭШ-ПАМЯТЬ

Основная цель организации виртуальной памяти, описанной в предыдущих разделах, состоит в том, что доступная программе память может быть значительно увеличена за счет обмена страниц между дорогой быстродействующей основной памятью и более медленной и дешевой дисковой памятью. При соответствующем использовании программа может выполняться в системе виртуальной памяти с подкачкой страниц так же быстро, как если бы она выполнялась, находясь целиком в основной памяти. Возникает вопрос, может ли быть сделана подобная оценка при сравнении основной памяти и более дорогой сверхоперативной памяти. Эффект мог быть почти таким же, как если бы программа выполнялась в более быстрой памяти.

Действительно, такая система памяти может быть реализована; она получила название *кэш-памяти* или быстродействующей буферной памяти. Кэш-память состоит из быстродействующих элементов памяти и имеет блочную организацию. Блоки кэш-памяти называются *фреймами* (чтобы отличать их от страниц виртуальной памяти). Обычно фреймы меньше, чем страницы. В зависимости от исполнения размер фрейма может изменяться от десятков до сотен байтов. В противоположность этому длина страницы может изменяться от сотен до тысяч байтов.

По мере того как информация из основной памяти используется программой, в кэш-память загружаются фреймы с этой информацией. Поэтому, если требуются другие данные из загруженного фрейма, они считываются уже из кэш-памяти; таким образом сокращается время доступа. Когда программа записывает информацию в память, то она заносится и в кэш-память, и в основную память, так что позже будет доступна из любой памяти этих двух типов. Это называется *сквозной записью.*

В то время как основная память может быть на несколько порядков быстрее, чем дисковая память, кэш-память быстрее основной памяти только в несколько раз. В результате нужны быстрые и простые алгоритмы загрузки фреймов, сквозной записи, выбора фрейма для перезаписи и прежде всего для определения, имеются ли данные в кэш-памяти. Эти алгоритмы должны быть реализованы непосредственно аппаратурой, поскольку прерывание выполнения программ и обработка прерываний программными средствами могут свести на нет любое преимущественное быстродействие. При такой реализации применение кэш-памяти может повысить быстродействие вычислительной мащины в 2 – 3 раза.

14.3. УПРАВЛЕНИЕ ПАМЯТЬЮ НА ЭВМ СЕМЕЙСТВА VAX

В предыдущем разделе рассматривались основные принципы организации управления памятью, виртуальной адресации, виртуальной памяти и кэш-памяти. Эти принципы организации не являются свойственными исключительно ЭВМ семейства VAX, они могут быть отнесены ко многим разработкам вычислительных машин. В данном разделе будут рассмотрены некоторые подробности того, как осуществляется управление памятью на ЭВМ семейства VAX. Изучение управления памятью включает как особенности архитектуры, которые одинаковы для всех моделей ЭВМ семейства VAX, так и особенности исполнения, которые могут отличаться для различных моделей. Например, тогда как старшие модели ЭВМ семейства VAX имеют большую кэш-память, младшие модели семейства имеют ма-

ленькую кэш-память или не имеют ее вовсе. Поскольку охватить все особенности конкретных моделей достаточно трудно и изложение было бы долгим, в этой главе мы ограничимся обсуждением только особенностей, свойственных большинству или всем моделям ЭВМ семейства VAX.

Одной из особенностей организации виртуальной памяти на ЭВМ семейства VAX является деление виртуального адресного пространства на области. Виртуальное адресное пространство на ЭВМ VAX составляет 4 Гбайта. Это виртуальное пространство разбивается на страницы, имеющие такую же длину, как блоки или физические записи на диске. Фирма DEC стандартизовала длину блока, сделав ее равной 512 байтам, или 1/2 Кбайта. Для того чтобы отобразить всю виртуальную память, потребовалось бы приблизительно

4 Гбайта/0,5 Кбайт на страницу = 8 млн. страниц

или

8 млн. длинных слов (элементов в таблице страниц).

Эти 8 млн. элементов в таблице страниц, имеющих формат длинного слова, могут занять 8 × 4 = 32 Мбайта памяти. Это не только намного больше того объема реальной памяти, который имеют большинство ЭВМ семейства VAX, но и превышает максимальный объем памяти, которая может быть установлена на многих моделях ЭВМ семейства VAX. Невозможно сохранять всю информацию об отображении виртуальных адресов в реальной памяти. Однако, по счастью, большинство приложений имеют более скромные требования к памяти. В любом случае таблицы страниц памяти для программ пользователей в действительности хранятся в виртуальной памяти. Это позволяет разбивать на страницы и сами таблицы страниц.

Для того чтобы упростить нахождение смещения для страницы и других ее атрибутов[1], вся такого рода информация, относящаяся к странице, сохраняется в непрерывных таблицах с последовательной организацией. Элемент в таблице страниц может быть найден с помощью прямой индексации, индексом элемента в таблице служат старшие биты виртуального адреса. В простейшем виде это могло ограничить программу использованием множества последовательных виртуальных адресов, начинающихся с адреса ^X00000000. Для большей гибкости, чтобы можно было иметь дело с фиксированными системными областями, а также с такими заранее не определенными параметрами, как длина программы и стека, 4 Гбайта виртуального адресного пространства разделяются на две области – по 2 Гбайта каждая, называемые *пространством процесса* и *пространством системы*. Пространство процесса занимает диапазон виртуальных адресов ^X00000000 – ^X7FFFFFFF. Пространство системы занимает оставшуюся часть виртуального пространства, начинающуюся с адреса ^X80000000 и оканчивающуюся адресом ^XFFFFFFFF. Однако в системах VAX последнего поколения используется только первая половина пространства системы, т. е. ограниченная диапазоном адресов ^X8000000 – ^XBFFFFFFF. Пространство системы отводится для постоянно находящихся в памяти системных подпрограмм, области стека для системных подпрограмм, а также функций ввода-вывода, описанных в этой главе ниже. Хотя пользователи при обращении к некоторым системным подпрограммам могут иметь доступ с правом на чтение, эта область защищается от пользователей по записи.

Пространство процесса выделяется для каждого процесса пользователя или, иначе, множества выполняемых программ. Таблица страниц необходима для каждой области, выделенной для процесса пользователя. При передаче управления от одного процесса пользователя к другому происходит переключение таблиц страниц. Переключение таблиц страниц является частью того, что называется *переключением контекста*, и будет описываться далее.

Одна из проблем, возникающая, когда пользователю выделяется непрерывная область памяти, состоит в том, что объем памяти, требуемый программе, может во время ее выполнения увеличиваться. На ЭВМ семейства VAX эта проблема решается добавлением дополнительной области для увеличения памяти к концу программы, так что общее пространство памяти, необходимое программе, остается непрерывным при ее увеличении.

В то время как требуемая программе память увеличивается по мере увеличения адресов памяти, увеличение области стека, интенсивно используемой в программах на ЭВМ семейства VAX, происходит по направлению "вниз" – в сторону младших адресов памяти. Для разрешения этой проблемы на ЭВМ семейства VAX пространство процесса также разделяется на две области. Первая называется пространством P0 или *областью программы* и используется в основном для программ и данных. Вторая называется пространством P1 или *областью управления* и предназначается в основном для организации стека и для некоторых системных функций, связанных с обслуживанием программ пользователя.

Пространство P0 начинается с виртуального адреса ^X00000000 и кончается адресом ^X3FFFFFFF. Это пространство является отображающимся пространством, заданным в таблице страниц, начиная с адреса ^X00000000 и выше. Если в программе пользователя происходит обращение к адресу, принадлежащему пространству P0, но оказавшемуся старше любого отображаемого в таблице страниц адреса, произойдет прерывание по отсутствию страницы. Если операционная система решает, что программа имеет право на большее пространство, то таблица страниц расширяется и программе выделяются дополнительные страницы виртуальной памяти.

Поскольку по мере занесения информации в стек значение указателя стека уменьшается, отображение пространства P1 противоположно пространству P0. Пространство P1 начинается с виртуального адреса ^X40000000 и кончается адресом ^X7FFFFFFF, но выделение памяти из этого пространства происходит в обратном направлении – в сторону уменьшения адресов; памяти выделяется столько, сколько необходимо для стека.

БУФЕР ТРАНСЛЯЦИИ

Для того чтобы ускорить преобразование виртуальных адресов в реальные адреса, применяется быстродействующее устройство памяти, называемое *буфером трансляции*. Буфер трансляции предназначен для хранения наиболее часто используемых адресов. Работа этого быстродействующего устройства памяти для процесса трансляции адресов аналогична работе кэш-памяти. Так же как и кэш-память, буфер трансляции является весьма дорогой частью аппаратуры ЭВМ и может быть включен или отсутствовать в различных моделях ЭВМ семейства VAX. Однако, подобно кэш-памяти, наличие буфера трансляции *прозрачно* для программиста, т. е. является для него скрытым. Программист не может сказать, имеет ли ЭВМ кэш-память или буфер трансляции, наличие этих устройств отражается только на увеличении скорости выполнения программы.

Особенностью внутреннего устройства, присущей в различной форме как кэш-памяти, так и буферу трансляции является *ассоциативная память*. Обычная память с произвольной выборкой информации имеет дешифратор адреса, который преобразует адрес так, чтобы можно было выбрать одну из N ячеек памяти. Например, память объемом 4 млн. длинных слов могла бы иметь дешифратор адреса, который позволил бы выбрать одну ячейку памяти из 4 млн. ячеек. Дешифратор работает так просто потому, что число ячеек памяти в точности соответствует числу возможных доступных адресов.

Напротив, ассоциативная память имеет относительно небольшое число ячеек, которые соответствуют адресам из значительно большего адресного пространства. Это означает, что адресов намного больше, чем ячеек этой памяти. Как следствие, некоторые адреса не соответствует какой-либо ячейке. Иначе говоря, не существует полного отображения адресов к ячейкам памяти.

Это такая же ситуация, которая возникает и с буфером трансляции, поскольку ограниченный объем буфера позволяет вместить только наиболее часто используемые адреса. Если бы буфер трансляции был реализован программно, алгоритм поиска требовал бы включения многочисленных сравнений. Такой поиск был бы медленным и буфер трансляции не

[1] В предыдущем разделе говорилось, что вся эта информация содержится в регистре смещения страницы. Термин *"регистр смещения страницы"* использовался авторами как условный для упрощения изложения. Не следует понимать, что для каждой страницы виртуальной памяти на самом деле имеется реальный физический регистр. – *Прим. ред.*

оправдал бы своего назначения. Для решения данной проблемы буфер трансляции адреса страницы реализуется аппаратно с помощью большого числа параллельно работающих электронных схем сравнения, обеспечивая высокую скорость доступа к хранимым данным. Буфер (или память), реализованный с помощью параллельных схем сравнения для выполнения очень быстрого поиска, называется *ассоциативной памятью.* Методы проектирования ассоциативной памяти применяются при разработке разнообразных частей больших вычислительных машин, включая в том числе буферы трансляции адреса страницы и кэш-память.

КОНТЕКСТ ПРОЦЕССА

Когда пользователь начинает сеанс работы с системой, ему для выполнения программ выделяются ресурсы. Выполнение программ включает выполнение не только программ пользователя, но и любых системных программ, вызываемых командами операционной системы, такими как COPY, DELETE, MACRO, LINK и т. д. Любая выполняющаяся по команде пользователя программа называется *процессом* пользователя. Что касается операционной системы, то она обслуживает ряд процессов. Некоторые их этих процессов являются процессами пользователей, некоторые являются системными процессами. Для каждого процесса необходимы следующие ресурсы вычислительной машины:

1. Некоторое назначение виртуальных адресов адресам реальной памяти.
2. Набор регистров процессора (R0, R1, R2, . . .).
3. Некоторые другие регистры процессора, используемые для управления процессом.

Перечисленные ресурсы называются *контекстом процесса* и выделяются персонально для каждого пользователя.

Очевидно, ячейка виртуальной памяти с адресом ^X8E4 одной программы – это совсем иная ячейка, чем ячейка виртуальной памяти ^X8E4 какой-либо еще программы. Каждая программа должна иметь свое собственное отображение виртуальных адресов, так чтобы каждая программа имела доступ к различным адресам реальной памяти. Аналогично одна программа использует регистры R0, R1 и т. д., и те же регистры используют другие программы, выполняющиеся в то же самое время. При переключении операционной системой управления между различными процессами операционная система для данного процесса должна загрузить в регистры R0, R1 и т. д. корректные значения и установить регистры – указатели таблиц страниц и регистры длины таблиц страниц (определяющие местонахождение и длину таблиц страниц пространства P0 и P1), чтобы задать соответствующие назначения виртуальных адресов для этого процесса. Все значения сохраняются в системной области памяти, а операция сохранения контекста и загрузки другого называется *переключением контекста.* ЭВМ семейства VAX имеют инструкции, позволяющие эффективно реализовывать переключение контекста.

Поскольку переключение контекста или любая операция, включающая отображение памяти или другие системные функции, может оказать влияние на всех пользователей, эти операции могут выполняться только привилегированными программами. Различного рода защита и привилегии, имеющиеся на ЭВМ семейства VAX, обсуждаются далее в данной главе.

УПРАЖНЕНИЯ 14.1.

1. Дайте краткие определения следующим терминам:
   а. Оверлей.
   б. Виртуальный адрес.
   в. Виртуальная память.
   г. Кэш-память.
   д. Страница.
   е. Фрейм.
   ж. Контекст процесса.
2. а. Как распределяется память для пространства P0 и P1?
   б. Какой смысл имеет различная направленность этих пространств?
   в. Как называются пространства P0 и P1?
   г. Как распределено пространство системы?

3. Выберите некоторую конкретную модель ЭВМ семейства VAX, возможно, ту, которую вы используете. Применительно к вашей модели ЭВМ ответьте на следующие вопросы:
   а. Сколько реальной физической памяти имеет данная модель?
   б. Какой максимальный допустимый объем памяти реально может иметь модель?
   в. Имеет ли ваша модель кэш-память? Если это так, то каков объем кэш-памяти и какова ее организация?
   г. Имеет ли ваша модель буфер трансляции адресов страниц? Если имеет, то каков объем буфера и как он устроен?
4. а. Предположим, что вы имеете программу, занимающую 1000 страниц виртуальной памяти на ЭВМ, имеющей только 100 страниц реальной памяти. Промоделируйте запросы к странице, используя генерацию случайных чисел от 1 до 1000. Напишите программу на любом языке программирования, которая будет моделировать такую систему для 10 тыс. обращений к памяти. Ваша программа должна моделировать обмен страниц при возникновении ситуации отсутствия страницы и распечатывать число таких прерываний, а также среднее число обращений к памяти между прерываниями.

6. Выполните повторно написанную вами программу, изменив вероятность выбора страниц таким образом, чтобы вероятность выбора страниц, уже отображенных в реальную память, была в 10 раз выше, чем при выборе страниц, вызывающих прерывание по отсутствию страниц. Как изменится в таком случае среднее число прерываний.

14.4. ЗАЩИТА И ПРИВИЛЕГИИ

В любой многопользовательской системе для исключения случайного или преднамеренного влияния пользователей друг на друга должны быть различные системы защиты. В то же время, для хорошо отлаженных системных процессов необходимо дать возможность превышать защиту и другие ограничения, так чтобы они могли выполнять такие действия, как загрузка и выгрузка процессов пользователей, или даже позволять процессам пользователей взаимодействовать между собой.

РЕЖИМЫ ДОСТУПА

Для того чтобы упростить разделение привилегий, которые может иметь программа, по группам, на ЭВМ семейства VAX имеются четыре уровня привилегий, действие которых проявляется при выполнении программы. Эти уровни привилегий называются *режимами доступа*, поскольку они предназначены для контроля за тем, к какого рода операциям программа имеет доступ. Ниже перечислены режимы доступа в порядке снижения старшинства:

режим ядра,
режим управления,
режим супервизора,
режим пользователя.

Как уже отмечалось, в длинных словах смещения страницы имеются некоторые дополнительные биты, применяемые для защиты памяти. Отдельные страницы могут использоваться беспрепятственно, другие могут быть открыты для чтения, но защищены от записи, а к третьим вовсе может быть запрещен доступ. Эта организация защиты распространяется на все режимы доступа, так что для различных режимов могут существовать и различные типы защиты. Таким образом, для любой страницы возможно установить биты защиты так, чтобы программы, выполняющиеся в режиме пользователя, не имели к ней доступа, программы, выполняющиеся в режиме супервизора, имели доступ по чтению, а программы, выполняющиеся в режимах управления и ядра, имели доступ по чтению и записи. Отметим, что режимы доступа образуют строгую иерархию. Поэтому программа, выполняющаяся в старших режимах, имеет доступ любой привилегии младших режимов доступа. Если программа в режиме пользователя имеет доступ к странице по чтению, то и все программы в старших режимах доступа могут иметь такой доступ. **Привилегия доступа по записи** подразумевает наличие привилегии доступа по чтению. В табл. 14.1 показано, как использу-

Таблица 14.1. Коды защиты памяти по доступу

Код	Режимы: Ядра	Управления	Супервизора	Пользователя
0000	нет	нет	нет	нет
0001	---------- не используется ----------			
0011	Чт/Зап	только Чт	нет	нет
0100	только Чт	нет	нет	нет
0101	Чт/Зап	Чт/Зап	Чт/Зап	Чт/Зап
0110	Чт/Зап	Чт/Зап	нет	нет
0111	Чт/Зап	только Чт	нет	нет
1000	Чт/Зап	Чт/Зап	Чт/Зап	нет
1001	Чт/Зап	Чт/Зап	только Чт	нет
1010	Чт/Зап	только Чт	только Чт	нет
1011	только Чт	только Чт	только Чт	нет
1100	Чт/Зап	Чт/Зап	Чт/Зап	только Чт
1101	Чт/Зап	Чт/Зап	только Чт	только Чт
1110	Чт/Зап	только Чт	только Чт	только Чт
1111	только Чт	только Чт	только Чт	только Чт

Обозначения: Чт/Зап - разрешен доступ по Чтению/Записи
только Чт - разрешен доступ только по Чтению
нет - все виды доступа запрещены

ются биты 27 – 30 элемента таблицы страниц для задания прав доступа к странице для различных режимов доступа. Обратите внимание! В таблице нет ни одной записи, которая бы позволила программам, выполняющимся в младших режимах доступа, иметь большее право доступа, чем программам, выполняющимся в старших режимах.

Кроме того, что режимы доступа влияют на доступ к памяти, они также влияют и на использование инструкций. Например, *резервные инструкции* для чтения и записи содержимого регистра управления процессора и для загрузки или сохранения контекста процесса могут выполняться только в режиме ядра. Любая попытка выполнить эти инструкции в любом другом режиме вызовет *прерывание по резервной инструкции*. Резервные инструкции являются инструкциями, которые для своего использования требуют специальных привилегий.

Операционная система VAX/VMS устроена так, что операции разного рода выполняются в различных режимах доступа. Следующий список дает представление об использовании режимов доступа.

Режим	Применение
Пользователя	Программы пользователей, утилиты, компиляторы, отладчики
Супервизора	Интерпретатор командного языка
Управления	Вызов системных сервисных подпрограмм и функций подсистемы RMS
Ядра	Управление страницами виртуальной памяти, планирование процессов, физический уровень ввода-вывода

ИНСТРУКЦИЯ PROBE

Инструкция PROBE позволяет программе проверить допустимость выполнения операций, связанных с обращением к памяти, перед выполнением этих операций. Имеется ряд соображений, почему такая проверка может оказаться полезной. Во-первых, это защита операционной системы. Программа пользователя может вызывать процедуры подсистемы управления записями (RMS), которые выполняются в более привилегированном режиме, таком как режим управления. Программа пользователя может в качестве аргументов передавать некоторые адреса. Предположим, что данные находящиеся по этим адресам, были защищены от доступа в режиме пользователя, но не от доступа в режиме супервизора. Тогда процедуры подсистемы RMS могли бы успешно осуществить запрещенный для программы пользователя доступ при их вызове этой программой.

Для защиты от подобных действий более привилегированной процедуре следует проверить все адреса, передаваемые в качестве аргументов, при ее вызове из менее привилегированной программы, чтобы сделать нечто такое, в результате чего менее привилегированная программа имела бы разрешенный для нее доступ при обращении к этим адресам.

Имеется две инструкции PROBE. Одна инструкция служит для проверки права доступа по чтению (PROBER), а другая – для проверки права доступа по записи (PROBEW). Данные инструкции имеют три аргумента:

1. Режим доступа.
2. Базовый адрес.
3. Длина.

Аргумент "Режим доступа" определяет, какой режим доступа должен быть проверен:

0 – режим ядра.
1 – режим управления.
2 – режим супервизора.
3 – режим пользователя.

ЭВМ семейства VAX сохраняют информацию о предыдущем режиме, когда вызов программы с более привилегированным режимом изменяет режим доступа. Инструкции PROBE определяют, какой из режимов более привилегированный: указанный в инструкции или предыдущий режим. Отметьте, что режимы со старшими номерами являются менее привилегированными.

Базовый адрес и длина позволяют специфицировать массив байтов. Однако контролируется только первый и последний адреса, так что массив, занимающий больше чем две страницы, следует проверить по частям.

Например, инструкция

```
PROBER    #3, #50, A
```

могла бы использоваться для контроля того, имеет ли программа пользователя право на доступ по чтению 50 байтов, начинающихся с адреса A. Аналогично инструкция

```
PROBEW    #3, #4, B
```

могла бы проконтролировать, имеет ли программа пользователя право на доступ по записи для обращения к длинному слову с адресом B. Для любой из этих инструкций бит Z принимается для проверки права на доступ. Если бит Z = 1, доступ является запрещенным; если бит Z = 0, доступ разрешен. Поэтому после инструкций PROBE для завершения проверки следует поместить инструкцию типа

```
BEQL      NO_ACCESS
```

или

```
BNEQ      ACCESS_LEGAL
```

Отметим, что здесь было сделано несколько допущений. Первое: доступ по записи предполагает наличие доступа по чтению, а запрет на чтение предполагает запрет по записи. Второе: режим #3, или режим пользователя, является наименее привилегированным. Поэтому предыдущие инструкции PROBE проверяли права режима доступа пользователя, невзирая на значение предыдущего режима.

РЕГИСТРЫ ПРОЦЕССОРА

Кроме 16 регистров общего назначения процессора (R0, R1, . . .) на ЭВМ семейства VAX имеется ряд регистров специального назначения. Эти регистры служат для управления внутренней работой ЭВМ семейства VAX. В их число входят такие регистры, как указатели

таблиц страниц виртуальной памяти. Отметим, что, поскольку большинство из этих регистров доступны только программам, выполняющимся в режиме ярда, здесь будут рассмотрены только наиболее общие принципы их использования. Помимо упомянутых регистров имеются некоторые регистры, использование которых отличается на разных моделях ЭВМ; такие регистры могут быть у одних моделей ЭВМ, но отсутствовать у других.

Один регистр должен быть упомянут специально, он называется *длинное слово состояния процессора* PSL. Этот регистр содержит поля битов, в которые заносится информация различного рода о состоянии выполняющегося процесса. Например, два бита определяют уровень доступа процесса 0, 1, 2 или 3 для режима ядра, управления, супервизора или пользователя соответственно. Еще 2 бита сохраняют информацию о предыдущем режиме, данная информация использовалась инструкциями PROBE. Еще 4 бита содержат коды условий – биты N, C, Z и V.

При выполнении программы биты регистра PSL изменяются. Например, любая инструкция, оказывающая влияние на коды условий, будет заносить информацию в поле битов кодов условий. Чтобы изменить режим доступа, необходимо занести в регистр PSL соответствующую информацию. И, конечно, для этого требуются некоторые привилегии. Однако можно прочитать содержимое регистра PSL с помощью инструкции MOVPSL. По инструкции MOVPSL содержимое регистра PSL копируется в указанное длинное слово. Инструкцию MOVPSL можно применять для определения текущего или предыдущего режима доступа, а также для получения некоторой другой информации о состоянии процессора. В приведенном ниже списке показано назначение отдельных битов регистра PSL, более подробную информацию можно получить в книге "VAX Architecture Handbook".

Биты	Назначение
0	Бит C
1	Бит V
2	Бит Z
3	Бит N
4	Бит трассировки, может использоваться для эмуляции пошагового выполнения программы при отладке
5	Бит разрешения прерывания по переполнению целого числа
6	Бит разрешения прерывания по потере значимости числа с плавающей точкой
7	Бит разрешения прерывания по переполнению десятичного числа
8 – 15	Должны содержать 0
16 – 20	Уровень приоритета прерывания
21	Должен содержать 0
22 – 23	Предыдущий режим работы процессора
24 – 25	Текущий режим работы процессора
26	Флаг стека прерываний
27	Флаг выполнения первой части инструкции; некоторые инструкции, занимающие при выполнении много времени, могут быть прерваны на середине (признак неоконченного выполнения)
28 – 29	Должны содержать 0
30	Бит задержки трассировки
31	Бит режима совместимости, если он установлен, то процессор будет выполнять инструкции ЭВМ серии PDP-11

Отметим, что младшие 16 битов (биты 0 – 15) не содержат привилегированной информации и поэтому могут быть модифицированы программой пользователя (за исключением битов 8 – 15, которые не используются и всегда должны иметь значение 0). Младшие 16 битов иногда называются *словом состояния процессора* PSW. Младшие 8 битов PSW могут модифицироваться; установка и сброс этих битов осуществляются с помощью инструкций BISPSW или BICPSW. Например, инструкция

```
BISPSW   #^X20
```


устанавливает бит разрешения прерывания по переполнению целого числа. Если затем какая-либо инструкция программы установит при выполнении бит V, то прерывание по переполнению целого числа приведет к прекращению выполнения программы и будет выдано сообщение об ошибке. Установка этого бита может помочь при определении корректности целочисленных вычислений, но может и создать проблему для каких-либо программ, в которых используются арифметические операции над целыми числами без знака.

Для ускорения обмена между программами с различными привилегиями и помощи операционной системе в организации защиты в системе имеется пять различных стеков. Этими стеками являются: стек режима пользователя, стек режима супервизора, стек режима управления, стек режима ядра и стек прерываний. Подразумевается, что имеется также пять регистров указателей стека. При работе в одном из режимов доступа, всякий раз когда инструкция обращается к регистру SP явным или не явным образом, используется соответствующий режиму указатель стека. Выбор первых четырех указателей стека определяется текущим режимом доступа. Указатель стека прерываний используется, если установлен бит 26 регистра PSL. Обычно программа может получить доступ только к своему собственному указателю стека – стеку режима пользователя. Тем не менее имеются две инструкции MTPR и MFPR (переслать в регистр процессора и переслать из регистра процессора), которые позволяют программам, выполняющимся в режиме ядра, получить доступ ко всем пяти регистрам-указателям стека.

Эти инструкции также используются для доступа к регистрам базы и длины, которые определяют таблицы страниц виртуальных адресов для пространства P0, пространства P1 и пространства системы. Кроме того, существует еще целый ряд регистров, к которым можно получить доступ с помощью этих инструкций. Чтобы узнать, какие регистры имеет конкретная модель ЭВМ семейства VAX, читатель должен обратиться к книге "VAX Architecture Handbook".

14.5. СИСТЕМНЫЕ ИНТЕРФЕЙСЫ И ИХ СТРУКТУРА

Системный интерфейс типа *шина* представляет собой совокупность электрических проводников, используемых для подключения к процессору вычислительной машины его памяти, а также периферийных устройств. Обычно шину составляет большое число проводников – – линий связи, для каждого передаваемого или принимаемого бита информации выделяется своя линия. Это позволяет обеспечивать одновременную передачу всех битов информации, что намного быстрее, чем передача информации по одному биту, осуществляемая по одной линии, как делается в системах последовательной передачи данных. Процессор передает по шине адрес и управляющую информацию, а устройство памяти или периферийное устройство, которое опознает этот адрес, в ответ будет передавать или принимать данные по линиям данных шины.

Обычно даже простейшие ЭВМ имеют несколько различных шин. Однако наличие некоторых шин может быть прозрачно для пользователя, т. е. скрытым от него. Например, процессор ЭВМ семейства VAX имеет внутреннюю шину, соединяющую между собой регистры процессора. Такие шины могут быть полностью размещены на одной печатной плате или даже находиться внутри одной микросхемы.

С другой стороны, имеются шины, предназначенные для передачи данных между различными частями вычислительной системы. Пользователям важно немного знать об этих шинах, поскольку они влияют на конфигурацию системы и на использование периферийных устройств. Разные модели ЭВМ семейства VAX имеют ряд шин различного рода. Большинство ЭВМ семейства VAX имеют несколько шин различного назначения. В следующих разделах будут кратко описаны шины, применяемые в ЭВМ семейства VAX, а также будет показано, как они используются в различных моделях ЭВМ VAX.

Процессоры старших моделей ЭВМ семейства VAX, таких как VAX-11/780, VAX-11/782 и VAX-11/785, подключаются к памяти и к некоторым контроллерам периферийных устройств посредством высокоскоростной синхронной шины, называемой *синхронной магистралью межблочных соединений* или шиной SBI. Работа синхронной шины синхронизируется тактовыми импульсами, и все устройства, подключенные к шине, должны быть способны ответить на запросы в строго определенном интервале времени. Для шины SBI период повторения тактовых импульсов составляет 200 нс (1 нс = 10^{-9} с). Шина обеспечивает одновременную передачу 32 битов информации, кроме того, возможна и двойная передача информации, которая позволяет передавать по одному адресу две посылки по 32 бита каждая. Это повышает скорость передачи шины SBI и дает номинальную скорость передачи 13,3 млн. байтов/с.

Высокоскоростная синхронная шина, такая как SBI, может использоваться только на очень коротком расстоянии и только для подключения устройств, способных работать со скоростью синхронной передачи. Если скорость устройств различается, то синхронная шина должна работать со скоростью самого медленного подключенного к шине устройства. Для разрешения этой проблемы могут применяться асинхронные шины. Асинхронные шины соединяют устройства, которые отвечают на запросы с той скоростью, с которой они могут это сделать, а затем сигнализируют о завершении передачи. Такая сигнализация между устройствами, подтверждающая завершение какой-либо операции, называется *квитированием.*

АСИНХРОННЫЕ СИСТЕМНЫЕ ИНТЕРФЕЙСЫ

Для связи процессора с периферийными устройствами в ЭВМ семейства VAX применяются несколько асинхронных шин. На младших моделях ЭВМ семейства VAX одна из этих асинхронных шин может выступать в качестве основной системной шины, но на старших моделях ЭВМ семейства VAX эти шины подключаются к шине SBI с помощью *адаптеров*. Адаптеры имеют небольшого объема быстродействующую буферную память обратного магазинного типа (дисциплина обслуживания "первым пришел – первым вышел"), называемую SILO. Наличие буферной памяти позволяет адаптерам передавать данные шине SBI 64-битовыми порциями и тем самым извлекать преимущество двойной передачи по шине SBI. Первоначально асинхронные шины были разработаны для различных моделей ЭВМ серии PDP-11 или LSI-11. В связи с тем, что объем основной памяти на этих машинах меньше, чем на ЭВМ семейства VAX, адаптеры имеют возможность выполнять отображение адресов. Ниже приводятся основные характеристики асинхронных шин.

Шина **MASSBUS** – высокоскоростная шина с 16-битовым трактом данных. Передача управляющей информации осуществляется асинхронно, а данные передаются синхронно. Шина была первоначально разработана для ЭВМ PDP-11/70, в настоящее время используется на некоторых старших моделях ЭВМ семейства VAX для подключения высокоскоростных дисковых устройств, таких как диски типов RP и RM. Скорость передачи данных шины MASSBUS может достигать 2 млн. байтов/с.

Шина **UNIBUS** – полностью асинхронная шина с 16-битовым трактом данных. Первоначально применялась на всех ЭВМ серии PDP-11, для ЭВМ серии LSI-11 не использовалась. Шина UNIBUS позволяет подключить к ЭВМ семейства VAX любые периферийные устройства, которые могут быть подключены к ЭВМ серии PDP-11. Шина используется для подключения большинства устройств ввода-вывода, за исключением высокоскоростных дисков, подключаемых к шине MASSBUS. Младшие модели ЭВМ семейства VAX, такие как VAX-11/730, применяют шину UNIBUS для подключения всех устройств ввода-вывода. Скорость передачи данных шины UNIBUS может достигать 1,5 млн. байтов/с.

Шина **Q-BUS** – дешевая, полностью асинхронная шина с 16-битовым трактом данных. Первоначально была разработана для ЭВМ серии LSI-11, как более дешевый альтернативный вариант шине UNIBUS. Стоимость шины снижается за счет использования одних и тех же линий как для передачи данных, так и для передачи младших 16 битов адреса. Это означает, что данные и адреса должны передаваться в разное время. Конечно, это несколько замедляет работу пользователя, но при этом сокращается количество микросхем для каждого периферийного контроллера почти наполовину. ЭВМ серии VAX-11 непосредственно не используют шину Q-BUS. Тем не менее старшие модели, такие как VAX-11/780, имеют установленную внутри центральной стойки ЭВМ серии LSI-11, которая служит для управления консолью оператора и начальной загрузки системы. Поэтому, хотя эти модели ЭВМ семейства VAX номинально и включают шину Q-BUS, она не используется во время нормальной работы системы. Однако ЭВМ серии MicroVAX были спроектированы так, что шина Q-BUS была единственной внешней шиной, используемой и для подключения памяти, и для подключения периферийных устройств.

14.6. ФИЗИЧЕСКИЙ УРОВЕНЬ ВВОДА-ВЫВОДА НА ЭВМ СЕМЕЙСТВА VAX

В отличие от многих вычислительных машин ЭВМ семейства VAX не имеют специальных инструкций для выполнения операций ввода-вывода. Вместо этого на ЭВМ семейства VAX для выполнения операций ввода-вывода применяются обычные инструкции обработки данных, такие как семейства инструкций MOV, ADD, INC и т. д., которые активизируют операции ввода-вывода при обращении к специально зарезервированным адресам.

В конце верхней части реального (в противоположность виртуальному) адресного пространства для устройств ввода-вывода зарезервирован 1 Мбайт пространства, данная область адресов не используется как обычная память. Адреса в этой области относятся к регистрам контроллеров устройств, подключенных к шинам MASSBUS или UNIBUS. (Для ЭВМ серии MicroVAX адреса относятся к регистрам контроллеров ввода-вывода, подключенных к шине Q-BUS.) Биты или комбинации битов в регистрах устройств служат для активизации периферийных устройств или для получения информации о состоянии устройства. Например, загруженный в регистр команд код может быть указанием выполнить перемотку магнитной ленты. В регистре состояния один из битов может использоваться как признак завершения операций ввода-вывода, а какой-либо еще бит использоваться для индикации ошибки. Каждое устройство имеет свой собственный набор регистров и свой собственный способ интерпретации битов.

Следует отметить, что адреса регистров устройств являются адресами реальной памяти, поэтому, чтобы программа могла получить к ним доступ, на них должны быть отображены виртуальные адреса. Однако поскольку для операционной системы существует потенциальная опасность ее разрушения при неправильном обращении к устройствам на физическом уровне, то эти адреса помечаются как доступные только в режиме ядра. (Если бы пользователь имел доступ на физическом уровне к такому устройству, как магнитный диск, то он бы мог читать, модифицировать или удалять файлы и каталоги на диске, минуя таким образом систему защиты файлов ОС VAX/VMS.)

ПРЯМОЙ ДОСТУП К ПАМЯТИ

Для достижения высокой производительности вычислительной системы необходимо минимизировать для программ требуемое число обращений к регистрам устройств ввода-вывода при работе с этими устройствами. Например, если для передачи каждого байта с диска требовалось бы выполнение инструкции MOVB, то это сильно бы загрузило процессор. В действительности, ЭВМ семейства VAX не настолько быстрые машины, чтобы сохранить свою производительность, работая подобным образом с высокоскоростными дисками. Чтобы избежать таких потерь, большинство устройств ввода-вывода ЭВМ семейства VAX работают *в режиме прямого доступа к памяти*. Это означает, что, начав выполнение операции, устройство способно самостоятельно напрямую осуществлять чтение или запись информации в память ЭВМ семейства VAX, не требуя при этом какого-либо вмешательства со стороны программы; операция выполняется до тех пор, пока не будет передан полностью весь блок информации.

Прямой доступ к памяти выполняется частично с помощью контроллера устройства ввода-вывода и частично с помощью адаптера системной шины. Поскольку обе шины MASSBUS и UNIBUS имеют 16-битовый тракт данных, они способны передавать только слова и байты. Передача таких небольших блоков информации неэффективна для ЭВМ семейства VAX, для которых предпочтительнее использовать 32-битовую передачу или 64-битовую двойную передачу. Чтобы извлечь преимущество из такой более эффективной передачи, адаптеры системных шин снабжаются буферной памятью (SILO), в которой хранятся передаваемые данные. Весь обмен в режиме прямого доступа к памяти адаптеры пытаются по возможности выполнить 64-битовыми посылками. (Если длина передаваемого блока информации не кратна 64 битам, то для завершения передачи блока должна быть сделана лишняя передача.) Буферная память адаптера работает обратным образом по отношению к тому, как работает обычный стек. В случае стека элемент, помещенный в стек последним, извлекается из стека первым. В буферной памяти адаптера первый помещенный элемент первым же и извлекается. У применяемых в настоящее время адаптеров объем буферной памяти составляет 64 байта. Этот объем буферной памяти вполне достаточен для передачи по шине SBI данных с минимальной задержкой, а тогда как шина SBI будет занята передачей данных других устройств – для обработки полученных данных.

ПРЕРЫВАНИЯ

При прямом доступе к памяти или при выполнении других процессов ввода-вывода процессор ЭВМ был бы сильно загружен, если бы он непрерывно контролировал завершение процесса ввода-вывода. Для решения этой проблемы в современных вычислительных машинах используется аппарат прерываний. Прерывание – это сигнал, который заставляет процессор отложить выполнение текущей программы и начать выполнение специальной программы, называемой *процедурой обработки прерывания.* При прерывании программы сохраняется достаточно информации, чтобы управление могло быть передано обратно прерванной программе и она могла продолжить выполнение так, как будто она и не прерывалась. На ЭВМ семейства VAX в стеке прерываний сохраняется содержимое программного счетчика (регистр PC) и содержимое длинного слова состояния процессора (регистр PSL). Любой другой регистр, используемый процедурой обработки прерывания, должен сохраняться самой процедурой.

Местонахождение процедур обработки прерываний определяют длинные слова, называемые *векторами прерываний*, каждое из которых содержит начальный адрес частной процедуры обработки прерываний. Это называется *векторной обработкой прерываний* или обработкой прерываний по вектору. На ЭВМ семейства VAX векторы прерывания и процедуры обработки прерываний находятся в защищенном пространстве памяти системы; эти процедуры выполняются в режиме ядра. Однако операционная система VAX/VMS обеспечивает способы для вызова процедур обработки прерываний, написанных пользователями. Эти способы называются асинхронными системными прерываниями (AST). При соответствующей установке программа пользователя может иметь процедуры обработки завершения ввода-вывода, которые активизируются системными прерываниями. Это позволяет программе продолжать вычисление, вместо того чтобы ожидать, пока будет завершена операция ввода-вывода. Чтобы получить дополнительную информацию по использованию асинхронных системных прерываний, читателю следует просмотреть гл. 11 или обратиться к системному руководству "VAX/VMS System Services Reference Manual".

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Прерывания по исключительным ситуациям подобны прерываниям ввода-вывода, однако отличаются от них тем, что вызываются не внешними событиями, такими как завершение ввода-вывода, а внутренними, такими как переполнение, использование запрещенных или резервных инструкций или операндов, или обращением к запрещенной области памяти. В большинстве случаев при возникновении исключительной ситуации выполнение

программы завершается и выдается сообщение об ошибке. Тем не менее возможно иметь написанные пользователем процедуры, активизируемые при возникновении некоторых исключительных ситуаций. Такие процедуры называются *процедурами обработки исключительных ситуаций.*

Исключительные ситуации разделяются в основном на две категории – *ловушки* (trap) и *отказы* (fault). В случае исключительной ситуации типа ловушки прерывание происходит после завершения выполнения инструкции. Сохраненное значение программного счетчика является адресом следующей инструкции, которая должна выполняться. Примером такой исключительной ситуации является прерывание по переполнению целого числа. Если в регистре PSL установлен бит разрешения прерывания по переполнению целого числа, то прерывание произойдет в том случае, если при выполнении какой-либо инструкции обработки целых чисел, такой как ADDL3, будет установлен бит V. Операция, вызвавшая установку бита V, будет завершена, а результат (который может быть корректным при работе с числами без знака) будет помещен по адресу операнда результата.

Исключительная ситуация типа отказа возникает, если выполнение инструкции не может быть завершено. В этом случае в регистре PC будет восстановлено значение, которое находилось в нем перед выполнением инструкции. Это позволяет процедуре обработки исключительной ситуации исследовать инструкцию и определять, в чем причина. Вполне возможно, что процедура обработки исключительной ситуации сможет устранить эту причину. Например, на моделях ЭВМ семейства VAX, не имеющих инструкций для работы с числами с плавающей точкой типов G и H, для программной эмуляции этих инструкций можно использовать обработку прерывания по резервной инструкции, возникающего при отказе выполнения инструкции, отсутствующей в наборе инструкций. При этом с точки зрения пользователя программа будет выполняться корректно, хотя и медленнее, чем когда инструкции реализуются аппаратно.

ПРЕРЫВАЕМЫЕ ИНСТРУКЦИИ

Когда поступает запрос на прерывание, процессор будет занят выполнением очередной инструкции. Обычно, перед тем как передать управление процедуре обработки прерывания, процессор завершает выполнение инструкции. Однако сложные инструкции, такие как инструкции обработки символьных строк, описанные в гл. 13, представляют потенциальную проблему. Такие инструкции могут выполняться довольно продолжительное время, а многие высокоприоритетные события не могут ждать, пока завершится выполнение таких инструкций. (Даже на вычислительных системах VAX пересылка 65536 байтов может занять значительную долю секунды.)

Для решения проблемы инструкции с потенциально большим временем выполнения разрабатывались так, что они могут прерываться до завершения своего выполнения. Такие инструкции называются *прерываемыми инструкциями.* При возврате управления прерванной программе, выполнение прерванной инструкции продолжается с той точки, на которой оно было приостановлено.

При прерывании прерываемой инструкции во время ее выполнения необходимо сохранить промежуточные результаты, так чтобы ее выполнение возобновилось с той точки, где она была прервана. (Если было бы необходимо начинать выполнение инструкции каждый раз с начала, то серии прерываний могли бы задержать завершение инструкции на неопределенное время.) Поскольку прерываемые инструкции во время выполнения используют регистры R0 – R5 для хранения промежуточных результатов, то восстановление после прерывания является быстрым и эффективным. К прерываемым инструкциям относятся инструкции семейства POLY, описанные в гл. 12, а также инструкции обработки символьной информации и упакованных десятичных чисел, описанные в гл. 13.

БЛОКИРОВКИ

В случаях, когда несколько процессов или устройств модифицируют одну информацию, существует опасность неправильной корректировки информации. Это верно для любых запоминающих устройств, включая основную память или магнитные диски. Например, пред-

положим, что вы должны университету сумму 2001 долл., из них 2000 долл. за обучение и 1 долл. пени за библиотеку. Если случится так, что оба платежных листа будут обрабатываться приблизительно в одно и то же время, то может произойти следующее:

1. Процесс "библиотека" осуществит выборку вашего текущего баланса – 2001 долл.
2. Процесс "казначей" осуществит выборку вашего текущего баланса – 2001 долл.
3. Процесс "казначей" вычислит ваш новый баланс как 1 долл. (2001 долл. минус 2000 долл. казначею) и запишет его на устройство памяти.
4. Процесс "библиотека" вычислит ваш новый баланс как 2000 долл. (2001 долл. минус 1 долл. выплаты библиотеке) и запишет его на устройство памяти перезаписав при этом сумму (1 долл.), только что записанную процессом "казначей".

В итоге вы произвели выплаты 1 долл. и 2000 долл., по счету 2001 долл., но на устройстве памяти осталась информация, которая показывает, что вы должны 2000 долл.

Те же самые ошибки могут происходить, когда процесс и устройство модифицируют одну и ту же область памяти. Избежать такой ошибки с помощью только системного программного обеспечения может быть трудно. Решением проблемы является *блокировка* информации, чтобы она не была доступна одному процессу в то время, пока другой процесс модифицирует информацию. В приведенном примере, если ваш баланс был бы блокирован, то процесс "казначей" не мог бы иметь к нему доступ, пока процесс "библиотека" модифицировал ваш баланс.

Для поддержки работы операционной системы и работы параллельно выполняемых процессов в набор инструкций ЭВМ семейства VAX включены следующие семь инструкций:

Режим	Применение
Пользователя	Программы пользоваателя, утилиты, компиляторы, отладчики
Супервизора	Интерпретатор команд.
Управления	Вызовы системных сервисных программ и функций RMS.
Ядра	Управление страницами виртуальной памяти, планирование процессов, ввод и вывод на физическом уровне.

УПРАЖНЕНИЯ 14.2

1. Дайте краткие определения для следующих терминов:
   а. Привилегии. б. Режим доступа.
   в. Регистры процессора. г. Шина.
   д. UNIBUS. е. MASSBUS.
   ж. Q-BUS. з. Прямой доступ к памяти.
   и. Прерывание. к. Ловушка.
   л Отказ.
2. Игнорируя все наложенные ограничения, ответьте на вопрос: сколько может быть возможных комбинаций прав доступа типов "запрещен", "только чтение", "чтение-запись" для четырех режимов доступа? Показаны ли в табл. 14.1 все возможные комбинации прав доступа, которые предполагают, что старшие режимы доступа могут иметь такие же права, какие имеют младшие? Если нет, то какие другие возможности имеются?
3. Ответьте относительно модели ЭВМ семейства VAX, с которой вы работаете, на следующие вопросы: какие системные шины имеются на данной модели? Какие устройства ввода-вывода подключены к этим шинам? Существуют ли другие шины, доступные как дополнительное оборудование? Если да, то какие это шины и какое дополнительное оборудование необходимо для их установки в системе?

4. Используйте инструкции PROBER и PROBEW, чтобы определить, какие области памяти доступны пользователям для чтения и какие доступны для записи. Распечатайте в сжатом виде карту памяти всего виртуального пространства памяти. Будьте осторожны при форматировании информации, поскольку при распечатке только одной строки для каждой страницы памяти всего потребуется 8 млн. строк.
5. Обратитесь к руководству "VAX/VMS System Services Reference Manual", чтобы узнать, как сгенерировать AST прерывания, которые будут прерывать выполнение вашей программы каждые 10 с. Напишите программу, которая каждые 10 с будет подавать звуковой сигнал на вашем терминале, но которая бы при этом не тратила зря время процессора, выполняясь в непрерывном цикле.

ГЛАВА 15. ЯЗЫК АССЕМБЛЕРА БОЛЕЕ ПОДРОБНО

15.1. ВВЕДЕНИЕ

При изучении языка ассемблера проявляется соотношение между программами и данными, которое часто бывает скрыто в языках высокого уровня. Как было показано в предыдущих главах, инструкции (программа) и данные используют одну и ту же память ЭВМ. На самом деле это справедливо не только для ЭВМ семейства VAX, но также и для большинства ЭВМ универсального назначения, как для больших, так и для малых[1].

По сути это означает, что между программами и данными имеется весьма незначительное различие. В данной главе будут продемонстрированы расхождения такого рода, кроме того, будет показано, как программы, написанные пользователями, обрабатываются различными компонентами операционной системы. В действительности программу можно рассматривать как данные, которые считываются какой-либо обрабатывающей программой. Такие обрабатывающие программы упоминались в предыдущих главах, но подробно не разбирались – это будет сделано в данной главе. В дополнение будут рассмотрены некоторые вопросы, связанные с созданием программ, такие как написание позиционно независимого кода.

15.2. ФОРМАТ ПРОГРАММЫ

ОСНОВНЫЕ ВИДОИЗМЕНЕНИЯ ПРОГРАММЫ

Прежде чем программы, написанные на языке ассемблера или языке Фортран, могут быть выполнены, они обычно проходят различные стадии преобразования. На каждой стадии программа воспринимается как данные некоторого определенного вида. Далее будут рассмотрены форматы этих данных и их структура, а также различные обрабатывающие программы, используемые на каждой стадии преобразования.

На ЭВМ семейства VAX программы, написанные на языке ассемблера, и большинство программ на языках высокого уровня могут находиться в следующем виде:

1. **Исходный текст.** Исходный текст программы состоит из символьных строк, образующих операторы языка программирования.
2. **Объектный код.** Представляет собой оттранслированную программу, операторы которой конвертированы в двоичный машинный код ЭВМ VAX. Однако не все адреса в программе транслируются в двоичный код. Это происходит потому, что ассемблер (или компилятор) не определил значения для глобальных имен, а также потому, что может понадобиться настройка адресов программы. Вследствие этого для компоновщика должна быть

[1] Некоторые микроЭВМ специального назначения, такие как TMS-1000 фирмы Texas Instruments, имеют раздельные области памяти для программ и данных. Однако многие из описанных здесь принципов относятся и к ним.

обеспечена соответствующая информация, на основе которой он мог бы завершить преобразование программы.

3. **Файл выполняемого образа.** Компоновщик берет один или несколько объектных файлов и перемещает программные модули, производя настройку адресов так, чтобы каждый из них занимал свою отдельную область памяти. Глобальным именам присваиваются двоичные значения. В результате образуется файл выполняемого образа, содержащий двоичный код, который может быть загружен в память часто без какой-либо дальнейшей модификации.

4. **Активизированная программа.** Представляет собой готовую к выполнению программу, помещенную в определенную область памяти ЭВМ. Отличие между активизированной программой и файлом выполняемого образа состоит в том, что активизированной программе назначается область выполнения в виртуальной памяти, тогда как файл выполняемого образа просто хранится на некотором внешнем запоминающем устройстве большой емкости. В операционной системе VAX/VMS имеется программа, называемая *активатором образа*, которая назначает программе, находящейся в файле выполняемого образа, пространство в виртуальной памяти. То, как это делается, тесно связано с работой виртуальной памяти и описывается более подробно в разделе, посвященном загрузке в виртуальную память.

ИСХОДНЫЙ ТЕКСТ ПРОГРАММЫ

На ЭВМ семейства VAX исходный текст (или код) программы состоит из строк символов в коде ASCII, имеющих произвольную длину. Эти символьные строки разбиваются на части, называемые *операторами*, или *строками* текста, или *записями*. Точный формат строки исходного текста программы зависит от конкретного языка программирования, такого как язык ассемблера, Паскаль или Фортран. В структуре записей файла каждая строка отделяется от других с помощью управляющей информации. Длина строки текста обычно ограничивается сверху некоторым фиксированным числом символов, например 80 символами.

Имеется целый ряд обрабатывающих программ, предназначенных для работы с исходным текстом. Обрабатывающие программы – это программы, воспринимающие исходный текст как данные, т. е. как некоторую строку символов, чтобы выполнить над ней некоторые операции. Одни программы выполняют относительно простые операции, как, например, программ, осуществляющая копирование файла из одного каталога в другой. Эта программа вызывается по команде COPY (см. приложение Г). Другие программы, такие как редакторы текста (см. приложение Д) или ассемблер, выполняют специфические непосредственно над самим исходным текстом.

РЕДАКТИРОВАНИЕ ИСХОДНОГО ТЕКСТА

Редакторы текста, такие как EDT и TECO, в основном используются для создания, модификации исходного текста, а также для внесения в него добавлений. Однако применение редакторов текста не ограничивается каким-либо конкретным языком программирования; они могут работать с *любым* текстом в коде ASCII, имеющим строчно-ориентированную структуру, включая текст, совершенно не связанный с программированием, например литературный. Редакторы текста распознают разделители записи и страницы как служебные символы, но оставшиеся символы в записи обрабатываются, как обычные данные. Редакторы текста работают следующим образом: копируют или весь текст целиком, или большой блок текста в память как один большой массив символов. (Большой текстовый файл может не поместиться в выделенной памяти и должен быть разбит на страницы, каждая из которых редактируется отдельно. Некоторые редакторы текста, подобные редактору EDT, позволяют перемещаться со страницы на страницу как в прямом, так и в обратном направлении. Другие, подобно редактору TECO, обеспечивают перемещение по страницам строго в одном направлении – вперед.) Разнообразные команды редактирования дают редактору указание обнаружить определенное место в массиве символов и вставить или удалить символы из массива. Вставка и удаление часто бывают медленными, так как вся оставшаяся до конца массива часть может целиком перекопироваться, чтобы освободить или занять пустое пространство. Тем не менее, так как ЭВМ семейства VAX имеют достаточное быстродействие, пользователь обычно не замечает каких-либо задержек по времени, если только вставки и удаления не повторяются. После завершения редактирования содержимое памяти копируется обратно в файл, расположенный на внешнем запоминающем устройстве.

ПРОЦЕСС АССЕМБЛИРОВАНИЯ

Читатель уже немного знаком с процессом ассемблирования, поскольку он рассматривался в различных главах книги, начиная с гл. 4. Однако если взглянуть на этот процесс с точки зрения обработки данных, то прояснятся некоторые детали. Как и большинство обрабатывающих программ, ассемблер считывает данные из входного файла и записывает их в выходные файлы[1]. В простейшем случае имеется один входной файл, в котором содержится исходный текст программы. Имеется *два* выходных файла: первый, в который помещается оттранслированная программа в двоичном формате, называется *объектным файлом*; во второй файл выводится листинг программы. Подобная информация выводится в оба файла, но есть некоторые различия. В то же время формат этой информации полностью различается, что будет показано в следующем разделе.

Как было сказано выше, на ЭВМ VAX процесс ассемблирования является двухпроходным. Это значит, что входной файл должен быть полностью считан ассемблером дважды, так как ассемблер не может назначить адреса ячейкам, которые определяются позже в программе. Это известно как *ссылки вперед*, такие ссылки могут появиться в ситуациях, аналогичных ситуации, показанной на рис. 15.1. Ссылка на имя NEXT содержится в инструкции JSB. Однако для этой инструкции не может быть сгенерирован код, так как ассемблер еще не считал строку программы, в которой определяется имя NEXT. Хотя в этом примере может показаться, что ассемблеру для разрешения ссылки вперед надо просмотреть текст программы лишь не намного вперед, следует запомнить: ссылки вперед часто могут быть в большой программе найдены только через много страниц.

Решением, принятым на ЭВМ VAX, является просмотр программы целиком, выполняемый один раз. Во время первого прохода ассемблера создается таблица имен. На втором проходе имена заменяются числами, чтобы сгенерировать машинный код программы. Замечательным является то, что в ассемблере ЭВМ VAX процессы обработки, выполняемые на первом и втором проходе, почти идентичны. Основное различие состоит в том, что во время первого прохода вывод информации подавляется. Если бы генерация машинного кода осуществлялась на первом проходе, то для каждой ссылки вперед могли бы появляться сообщения об ошибке, вызванные появлением неопределенного имени.

Во время второго прохода повторяется аналогичная обработка. Но теперь уже существует полная таблица имен, созданная на первом проходе. Таблица имен будет содержать значения всех разрешенных ссылок вперед. Во время второго прохода могут быть образованы объектный файл и файл листинга, так как каких-либо неопределенных имен больше не будет, если только программист сам не оставил их неопределенными.

Следует заметить, что использование на ЭВМ VAX двухпроходного ассемблера не является единственным решением проблемы ссылок вперед. Имеются и другие решения. Одно из них – умышленное избегание ссылок вперед программистом. Большинству пользователей такое решение не нравится. Другое решение состоит в том, что генерация кода осуществляется на первом проходе, а обнаруженные ссылки вперед помечаются. Затем, когда ссылки вперед будут разрешены, объектный файл будет скорректирован в соответствии

[1] Здесь файл будет рассматриваться как совокупность или последовательная цепочка данных, поступающих с, или выводимых на некоторое устройство ввода-вывода, такие как устройство ввода с перфокарт, устройство построчной печати, магнитные диски или ленты.

```
        JSB   NEXT
        . . .
NEXT:
```

Рис. 15.1. Ссылка вперед

с отмеченными ссылками. Такой ассемблер иногда называется *полуторапроходным*. Есть и еще решение – оставить разрешение ссылок вперед другой программе, компоновщику, загрузчику или непосредственно самому процессору ЭВМ с помощью некоторой схемы косвенной адресации.

ДАННЫЕ ДЛЯ АССЕМБЛЕРА

Как уже упоминалось, входные и выходные файлы ассемблера на самом деле представляют собой входные и выходные данные для программы. Два из трех файлов данных, а именно файл с исходным текстом программы и файл листинга, предназначены для пользователей. Поэтому эти файлы являются *символьными*, а их содержимое может быть распечатано постранично, как обычный текст. Читатель уже должен быть знаком с тем, как выглядит содержимое этих файлов. Но человек воспринимает информацию, содержащуюся в этих файлах, совсем не так, как ЭВМ. Слова, строки, страницы воспринимаются человеком как целое. Восприятие ЭВМ намного более ограничено; таким образом, символьные данные должны обрабатываться программой символ за символом или в лучшем случае небольшими группами символов (вероятно, до восьми символов одновременно при выполнении на ЭВМ VAX операций над квадрасловами).

По мере того как ассемблер считывает информацию из файла с исходным текстом программы, он должен быть способен различать в строке *поля*. Такими полями являются поле метки, поле кода операции, поле операндов и комментарии. Есть два распространенных способа распознавания полей. Первый основан на использовании полей фиксированной длины, начинающихся с определенных позиций символов на строке. Например, операторы Фортрана занимают на строке позиции (колонки) с 7-й по 72-ю. По сути это правило применяется в Фортране вследствие того, что первоначально язык был реализован на вычислительных системах, в которых использовался ввод и вывод с перфокарт. Аналогично первые ассемблеры для ЭВМ фирмы IBM были ориентированы на обработку полей, имеющих фиксированное расположение.

Второй способ распознавания полей основывается на использовании разделителей – знаков пунктуации, таких как двоеточие, точка с запятой, запятые, пробелы и т. д., как это делается в языке ассемблера для ЭВМ VAX. Для большинства современных приложений данный способ является более подходящим, поскольку ввод осуществляется с терминала, где позицию символа определить труднее, чем на перфокарте. Как следствие, многие ассемблеры используют комбинацию из отдельных фиксированных полей и разделителей. Например, в ассемблере для ЭВМ CDC COMPAS требуется, чтобы метки начинались с колонки 1, а коды операций начинались с любой колонки, кроме первой. Следовательно, чтобы отличать метки от кодов операций и операндов, не требуется двоеточие.

Теперь можно приблизительно описать функции, выполняемые различными частями ассемблера для ЭВМ VAX на первом и втором проходе. На каждом цикле своей работы ассемблер осуществляет выборку строки исходного текста и анализирует ее символ за символом[1]. Начальные пробелы в строке игнорируются[2]. Предполагается, что первый отличный от пробела символ является началом имени. Ассемблер накапливает последовательно

[1] Инструкции обработки символьных строк, описанные в гл. 13, эффективно работают в режиме посимвольной обработки.

[2] Символы горизонтальной табуляции обрабатываются в основном так же, как пробелы.

поступающие символы до тех пор, пока не будет обнаружен знак пунктуации. Обычно это пробел, двоеточие или знак равенства.

Если первый отличный от пробела символ является двоеточием, это означает, что накопленные символы представляют собой метку, и в таблице имен подается соответствующая запись. Если первый отличный от пробела символ – знак равенства, то ассемблер предполагает, что далее следует выражение, которое должно быть вычислено, а его значение занесено в таблицу имен. Если это какой-нибудь другой символ, ассемблер считает, что накопленные символы являются кодом операции, директивой ассемблера или именем макроинструкции. Коды операций и директивы ассемблера группируются по классам. Например, инструкции ADDL2, MOVL и SUBL2 подобны так же, как между собой подобны инструкции BRB, BEQL и BNEQ. Для обработки кодов операций каждого класса есть специальные обрабатывающие подпрограммы, которые в соответствии с кодом операции обрабатывают поле операндов и генерируют объектный код. Строка исходного текста завершается или символом признака конца строки, или точкой с запятой. Комментарии, признаком начала которых является точка с запятой, ассемблером игнорируются.

Если выполняется второй проход, то генерируемый объектный код (если он есть) записывается в объектный файл и образуется строка листинга программы. В строку листинга программы заносятся шестнадцатеричные значения, представляющие объектный код, и адреса (переместимые (настраиваемые) адреса помечаются специальной отметкой), затем следуют номер строки и исходный текст в том виде, в каком он был во введенной строке исходного текста программы. Этот полный цикл повторяется и в конце концов завершается при обработке директивы .END.

РАСШИРЕНИЕ МАКРОИНСТРУКЦИЙ

Макропроцессор фактически представляет собой отдельную обрабатывающую программу, являющуюся на самом деле расширением языка ассемблера. Как было описано в гл. 10, макрообработка складывается из двух частей: определения макроинструкций и вызова или расширения макроинструкций. Когда встречается макроопределение, строки, расположенные между директивой .MACRO и соответствующей ей директивой .ENDM, копируются в область макроопределений.

При вызове макроинструкции ее имя помещается в поле кода операции вводимой строки. При обнаружении макровызова начинает работать подпрограмма обработки макрорасширений. Эта подпрограмма копирует строки исходного текста, находящиеся в области макроопределений, в область расширения. Во время копирования замещаемые параметры заменяются аргументами, указанными в макровызове. В большинстве случаев осуществляется замена одной строки символов на другую без интерпретации.

После завершения расширения управление передается подпрограмме, отвечающей за выполнение проходов ассемблирования. При наличии макровызовов процесс ассемблирования несколько отличается от обычного. Если в области макрорасширения имеется какой-либо исходный код, то выборка строк программы осуществляется из этой области, а не из файла с исходным текстом программы. В остальном процесс ассемблирования осуществляется так же, как и раньше. Выборка строк из области макрорасширения будет продолжаться до тех пор, пока не будут выбраны все строки из этой области, после чего строки снова будут браться из файла с исходным текстом. Такое действие одинаково для двух проходов.

Если одна из строк в макрорасширении является макровызовом, происходит такой же процесс расширения, за одним исключением: расширенная макроинструкция всегда добавляется перед тем, что может оставаться в области расширения. Это позволяет из макроинструкций вызывать макроинструкции, внутри которых вызываются другие макроинструкции, и т. д. На вложенные вызовы макроинструкций нет никаких ограничений, кроме ограничения по памяти, накладываемого на область расширения. Поскольку область макрорасширений устроена как стек, макроинструкции могут быть даже рекурсивными и вы-

зывать самих себя. Заметим, что в рекурсивных макроинструкциях вызов макроинструкции самой себя должен находиться в блоке условия, так чтобы процесс в конечном итоге был завершен.

УПРАЖНЕНИЯ 15.1

1. Напишите программу, которая распечатывает по байтам содержимое всех своих ячеек (как инструкции, так и данные). Она должна распечатывать также содержимое 16 регистров общего назначения. Содержимое ячеек и регистров распечатывается в шестнадцатеричном и двоичном представлении. Например, распечатка может иметь следующий вид:

Адрес	Шестнадцатеричное представление	Двоичное представление
00000200	AF	10101111
00000201	D0	10110000
...	...	...

а) определите, какие части вашей программы (если они есть) представляют только инструкции, только данные или инструкции и данные? Почему?

б) имеются ли в вашей распечатке какие-либо странности, например несовпадение шестнадцатеричных и двоичных значений? Почему? Можно ли это исправить?

2. Напишите простую программу-редактор, которая работала бы подобно встроенному редактору для языка Бейсик. Пусть:

а) каждой строке предшествует шестизначный десятичный номер;

б) ввод строчек может осуществляться в любом порядке, но распечатываться строки будут в порядке увеличения их номеров;

в) если номер вводимой строки совпадает с номером записанной ранее строки, то прежняя строка будет удалена.

г) строка с номером 999999 используется для завершения ввода и распечатки отредактированных данных. Например, при вводе строк:

```
000100   ПЕРВАЯ СТРОКА
000070   ВТОРАЯ СТРОКА
000120   ТРЕТЬЯ СТРОКА
000130   ЧЕТВЕРТАЯ СТРОКА
000120   ПЯТАЯ СТРОКА
999999
```

будет распечатано следующее:

```
000070   ВТОРАЯ СТРОКА
000100   ПЕРВАЯ СТРОКА
000120   ПЯТАЯ СТРОКА
000130   ЧЕТВЕРТАЯ СТРОКА
```

3. Приведенные ниже инструкции имеют различную структуру операндов и могли бы быть отнесены к различным группам:

```
CLRL    ADDL2   BRB     JMP
CALLS   RET     MOVAB   ACBL
```

а) в наиболее общем виде опишите операнды каждой инструкции;

б) опишите, что бы делал ассемблер при обработке операндов этих инструкций.

4. Опишите, для чего предназначены приведенные ниже директивы ассемблера? Какие действия могут быть выполнены на каждом проходе ассемблирования при обработке этих директив?

```
.LIST   .WORD   .REPT   .ASCII
```


5. Вручную проделайте по шагам ассемблирование следующей программы; при обработке каждой строки покажите, что будет находиться в области макроопределений и в области макрорасширений:

```
        .MACRO  FAC     A,N
        .IF     EQUAL,N
A=1
        .IF_FALSE
        FAC     A,N-1
A=A*N
        .ENDC
        .ENDM
        FAC     NUM,3
        .END
```

15.3. ОБЪЕКТНЫЙ КОД

ДВОИЧНЫЕ ФАЙЛЫ И СТРУКТУРА ЗАПИСЕЙ

Поскольку исходный текст программ и листинги предназначены для человека, то они записываются в файлы в символьном виде – в коде ASCII. Однако объектный код обычно не просматривается человеком, он считывается ЭВМ во время компоновки. Поэтому лучше использовать формат данных, наиболее подходящий для ЭВМ (лучше, чем код ASCII). Информация в символьных файлах представлена в некомпактном виде, так как обычно включает пробелы, знаки табуляции и другие знаки пунктуации, чтобы сделать данные удобными для чтения. Например, 16-битовое двоичное число может быть записано как четырехзначное шестнадцатеричное число, отделенное одним пробелом от последующих чисел. Для такого представления потребовалось бы пять символов в коде ASCII или пять байтов данных. Тем не менее во внутреннем представлении 16-битовое число занимает только два байта. Следовательно, данные получаются намного более компактными и могут передаваться намного эффективнее, если они сохраняют внутреннюю двоичную форму представления. Файлы, содержащие информацию в таком виде, называются *двоичными файлами.*

Обычно большое число данных (или потенциально большое), которое хранится в двоичных файлах, требуется разбить на небольшие порции так же, как информация в символьных файлах разбивается на строки. Как было показано в гл. 11, эти порции данных называются *записями.* В основном имеется два способа сегментации двоичных файлов: записи фиксированной и записи переменной длины. В случае записей фиксированной длины существует соглашение, что все без исключения записи имеют некоторый фиксированный размер. Следовательно, не требуется иметь какую-либо управляющую информацию для описания длины каждой записи.

В объектных файлах, как будет показано далее, необходимо хранить данные различного вида и объема. В этом случае записи переменной длины позволяют уменьшать занимаемое пространство памяти. Существует два основных способа для определения размера записей переменной длины. Эти способы аналогичны способам, применяемым в языках высокого уровня для разделения символьных строк. Например, первый способ основан на использовании указателя длины записи, который включается в запись как часть данных. Это подобно записи холлеритовских констант в Фортране. Например:

```
17HHERE'S A MESSAGE.
```

Символы 17H в начале строки определяют длину сообщения, которое может содержать любые символы.

Другой способ определения длины записи основывается на применении управляющих символов, таких как "Возврат каретки" и "Перевод строки", используемых при вводе

текста в коде ASCII с терминала. Но в этом случае становится неудобно работать с текстом, внутри которого также содержатся управляющие символы. Эта проблема весьма существенна для двоичных файлов, в которых любая комбинация битов может являться правильной информацией. Одно решение проблемы состоит в том, чтобы использовать дублирующий управляющий символ для указания одиночного управляющего символа в тексте. Такое решение применяется в Фортране, когда желательно заключить в кавычки строку, внутри которой тоже есть кавычки. Предыдущий пример может быть переписан следующим образом:

```
'HERE''S A MESSAGE.'
```

Две стоящие рядом кавычки обрабатываются, как одна кавычка в тексте. Этот способ может быть приспособлен для работы с двоичными данными; он применяется в некоторых стандартных протоколах передачи данных, таких как BISYNC (протокол, применяемый ЭВМ фирмы IBM, а также другими ЭВМ в вычислительных сетях). Тем не менее в операционной системе VAX/VMS для разграничения записей переменной длины применяются указатели длины записи, что будет рассмотрено в следующем разделе.

ВНУТРЕННИЙ ФОРМАТ ЗАПИСЕЙ НА ЭВМ VAX

Как было описано в гл. 11, на ЭВМ VAX информация в файлах разбивается на записи, к которым может быть осуществлен последовательный или прямой доступ. Записи могут иметь переменную или фиксированную длину. Для записей фиксированной длины не требуется разграничивать их начало или конец, так как размер записей известен и фиксирован. Но для записей переменной длины должен быть способ определения местонахождения их границ.

Большинство внешних запоминающих устройств большой емкости на ЭВМ VAX устроены так, что информация записывается в физические блоки длиной 512 байтов[1]. Для того чтобы установить формат записи и методы доступа для файла, в начале каждого файла записывается дополнительный блок. Обычно пользователь "не видит" этот блок, так как он содержит только управляющую информацию. Среди прочего этот блок содержит битовые коды, идентифицирующие тип записи и метод доступа, который используется для оставшейся части файла. Если вы любознательны, то можете посмотреть, что находится в данном блоке, с помощью команды DUMP/HEADER или DUMP/HEADER/NOFORMATED, введя их с терминала или поместив в задание, выполняемое в режиме пакетной обработки.

Остальные блоки файла содержат информацию пользователя, отформатированную так, как это указано в управляющем блоке. Если заданы записи переменной длины, что является обычным, то каждой записи предшествует 16-битовый указатель длины записи. Кроме того, при необходимости, для того чтобы число байтов в записи было всегда четным, к концу записи добавляется нулевой байт. Таким образом, приведенные строки текста

```
Hello
This is
Some text
. . .
```

могут иметь следующую структуру записей:

```
69 29 73 69 68 54 00 07 00 6F 6C 6C 65 48 00 05
...    00 74 78 45 74 20 65 6D 6F 53 00 09 00 73
```

Отметим, что записи расположены здесь справа налево, так же как принято на ЭВМ VAX для дампов и листингов ассемблера. Вследствие того что каждая строка текста занимает в записи нечетное число байтов, к концу записи требовалось добавить нулевой байт 00.

[1] Некоторые диски, например гибкие диски, разбиваются на секторы с меньшей длиной блока. В этих случаях на ЭВМ VAX для обеспечения совместимости блок 512 байтов формируется из нескольких секторов.

Байты записи не обязательно должны содержать коды ASCII, они могут содержать любую комбинацию битов, поскольку указатели длины записей явно задают длину каждой записи. В некоторых других операционных системах, подобных системам для ЭВМ семейства PDP-11 фирмы DEC, для разграничения записей используются такие управляющие символы, как "Возврат каретки" (^X0D) или "Перевод строки" (^X0A). Это означает, что в середине записи эти символы не могут присутствовать в тексте. В таких операционных системах файлы часто разделяют на символьные и двоичные файлы. Как было описано выше, для разграничения записей в символьных файлах применяются управляющие символы, а в двоичных файлах используются указатели длины записей. В ОС VAX/VMS указатели длины записей применяются как для двоичных, так и для символьных файлов, тем самым минимизируется различие между ними. По сути различие заключается в том, что обычно символьные файлы имеют в управляющем блоке параметры, задающие управление кареткой. Первоначально такая информация указывается при создании файла в *блоке доступа к файлу* (см. гл. 11).

ОБЪЕКТНЫЙ МОДУЛЬ

Объектный модуль представляет собой форматированный двоичный файл, создаваемый ассемблером во время ассемблирования. Объектный модуль содержит весь двоичный машинный код, сгенерированный ассемблером. Тем не менее требуется еще дополнительная информация. Вспомним, что программы на языке ассемблера обычно ассемблируются так, что они могут загружаться в различные области памяти, а несколько модулей программ и подпрограмм могут быть скомпонованы вместе.

Поэтому на ЭВМ VAX объектные модули обычно состоят из трех секций. Первая секция содержит общую информацию о программе и используемых в ней глобальных именах; вторая секция – полученный после трансляции машинный код; в третьей секции содержится информация о том, как должен быть модифицирован машинный код при настройке программы на какую-либо область памяти и где в коде при настройке должны задаваться глобальные адреса или параметры. Эти три секции называются *словарем глобальных имен, текстом* и *таблицей настройки*.

Словарь глобальных имен. Состоит из одной или нескольких записей[1]. Записи состоят из полей, в которых указывается следующая информация:

1) длина программы и длина каждой программной секции, определенной директивой .PSECT;

2) относительное местонахождение для всех определенных в программной секции глобальных имен;

3) имена всех используемых в программе неопределенных глобальных имен;

4) прочая информация, такая как имя программной секции, заданное в директиве .TITLE.

Текст. Полученный после трансляции двоичный код называется текстом, текст состоит из ряда записей. Каждая запись, называемая блоком текста, содержит относительный адрес области памяти, куда должен быть загружен текст, а также некоторый машинный код.

Таблица настройки. За каждым блоком текста может следовать запись, содержащая закодированные указания для модификации предшествующего ей блока текста. Ниже приведены примеры различных указаний, которые можно найти в таблице настройки:

1. Модифицировать в предшествующем блоке текста заданный адрес, складывая его с действительным адресом начала программы. Вспомним, что ассемблер, как правило, присваивает программной секции начальный адрес ^X00000000, который обычно модифицируется при загрузке программы.

2. Заменить заданный адрес в блоке текста смещением, вычисленным относительно значения в программном счетчике во время выполнения инструкции.

[1] Для того чтобы предотвратить значительное увеличение длины записи, большие словари глобальных имен состоят из нескольких записей.

3. Заменить адрес назначенным глобальному имени абсолютным адресом.

Существуют и другие указания по настройке, аналогичные перечисленным выше, но здесь они рассматриваться не будут. Отметим, что в некоторых инструкциях должна осуществляться модификация либо байта, либо слова, либо длинного слова.

Все двоичные записи в объектном модуле содержат коды, определяющие тип записи. Кроме того, имеются коды, отмечающие как концы различных секций модуля, так и конец самого модуля. Это позволяет объединять несколько модулей вместе, помещая их в один файл. Если при трансляции указывался ключ /DEBUG, то к объектному модулю будут добавлены дополнительные секции. Эти секции необходимы для сохранения таблицы имен и другой информации для отладчика.

ПРОЦЕСС КОМПОНОВКИ

Компоновка нескольких объектных модулей в одну абсолютную программу в машинном коде является двухпроходным процессом. Два прохода необходимы по тем же соображениям, что и два прохода при ассемблировании. Глобальные имена могут использоваться в одном модуле, но их определение может находиться в другом модуле, расположенном далее.

Во время первого прохода компоновки производятся распределение пространства памяти и определение значений для глобальных имен. Необходимая для этого информация содержится в словарях глобальных имен модулей. Компоновщик по мере считывания словарей глобальных имен может выделять пространство в памяти ЭВМ исходя из информации о длине каждой программной секции. Затем программным секциям могут быть назначены их истинные, а глобальным именам – абсолютные адреса. После чего становится возможным провести модификацию, указанную в таблицах настройки.

Таким образом, во время второго прохода блокам текста назначаются абсолютные адреса вместо относительных. Отдельные адреса модифицируются в соответствии с тем, что указано в таблицах настройки. В результате будет получена программа в абсолютных адресах, которая может быть непосредственно загружена в указанную область виртуальной памяти ЭВМ VAX и выполнена без какой-либо дальнейшей модификации, поскольку она содержит всю необходимую для выполнения информацию; выполнение программы начинается с адреса ^X200. В других случаях выходная информация компоновщика должна быть в той или иной степени модифицирована.

ПРОГРАММНЫЕ СЕКЦИИ

Программные секции представляют собой блоки кода в программе, используемые для некоторого специального назначения и обладающие достаточной полнотой для выполнения требуемых функций. Образование программных секций может быть проиллюстрировано на примере написания исходного текста программы, когда программист пишет различные части программы на отдельных листах бумаги. Каждый лист бумаги можно сравнить с программной секцией. После написания программы программист может сам переупорядочить листы так, чтобы расположить код в соответствующем порядке. Например, определения могут находиться на одном листе, блоки данных – на втором, основная программа – на третьем, подпрограммы – на других листах. При написании программы программист время от времени может ссылаться на тот или иной лист бумаги. Завершив программу, программист может ввести исходный текст в ЭВМ с упорядоченных соответствующим образом листов.

Когда программа генерируется автоматически некоторой обрабатывающей программой, такой как макропроцессор или компиляторы Паскаля или Фортрана, часто бывает полезно разбить код программы на различные блоки. Это можно осуществить, сделав каждый блок программной секцией. В каждую программную секцию одновременно может быть введено по несколько строк исходного текста; строки текста, относящиеся к разным программным секциям, отделяются с помощью директивы .PSECT. Во время компоновки программы блоки кода упорядочиваются так, что они появляются в соответствующих программных секциях.

С точки зрения назначения адресов компоновщиком программные секции могут быть разделены на две основные категории. К первой категории относятся последовательно связываемые секции. Такие секции представляют собой локальные блоки в данном программном модуле, не используемые другими программными модулями. Эти секции имеют имена, позволяющие различать их между собой, но чаще всего имена используются только для того, чтобы связать эти секции. К другой категории программных секций относятся перекрывающиеся секции. Они подобны общим блокам в Фортране, содержат данные или инструкции, разделяемые несколькими программными модулями. Эти секции имеют глобальные имена, поэтому компоновщик будет выделять одну и ту же область памяти для одинаково поименованных секций, находящихся в других модулях.

Во время первого прохода компоновщик должен выделить память для программных секций. Вспомним, что словарь глобальных имен модуля содержит информацию о длине каждой программной секции и ее типе. Последовательно связываемые секции располагаются одна за другой, а в случае перекрывающихся секций память выделяется для каждой секции, имеющей отличающееся имя. Закончив выделение памяти, компоновщик имеет полную карту распределения памяти для всей программы, которая может быть распечатана по запросу. Теперь рассмотрим второй проход компоновки.

БИБЛИОТЕКИ ПОДПРОГРАММ

Большинство операционных систем работают с библиотеками подпрограмм, предварительно ассемблированных или полученных после компиляции. Например, в ОС VAX/VMS имеются системные библиотеки, в которых содержатся следующие типы подпрограмм:

1. Стандартные подпрограммы Паскаля, Фортрана или других языков, а также функции, подобные SIN, SQRT, EXP и т.д.

2. Внутренние подпрограммы и функции Паскаля, Фортрана и других языков. Они вызываются при появлении более сложных операторов языка, таких как READ и WRITE. На самом деле некоторые мини-ЭВМ не имеют таких сложных операций, какие есть на ЭВМ семейства VAX. Поэтому для выполнения даже таких простых операций, как умножение и деление, могут потребоваться подпрограммы. Для того чтобы не спутать подпрограммы с подпрограммами, написанными пользователями, внутренние подпрограммы имеют необычные глобальные имена, в которые входят знаки подчеркивания или знаки доллара, например имя PAS$FV_OUTPUT.

3. Системные подпрограммы. Эти подпрограммы могут быть вызваны из программ, написанных на языках высокого уровня или языке ассемблера, для выполнения специальных функций операционной системы VAX/VMS. Имеются подпрограммы, которые могут быть вызваны из программ, написанных на Паскале или Фортране, позволяющие выполнять функции, подобные большинству функций ввода-вывода, реализуемых с помощью макроинструкций, описанных в гл. 11. Например, функции ввода-вывода с очередями могут быть выполнены в программах на языках высокого уровня посредством вызова подпрограммы SYS$QIOW.

Такие подпрограммы собираются в объектный файл специального вида, особенности организации файла облегчают поиск в нем глобальных имен. Этот файл используется после завершения первого прохода компоновки. Если были обнаружены какие-либо неопределенные глобальные имена, то начинается поиск в библиотеке. Вспомним, что словари глобальных имен наряду с определениями имен содержат списки неопределенных глобальных имен.

Если в программе имеются какие-либо неопределенные глобальные имена, то в библиотеке производится поиск объектных модулей, в которых эти имена определены. При нахождении такие объектные модули добавляются к программе. Модули, не содержащие

определения пропущенных глобальных имен, к программе не добавляются, поскольку это увеличило бы длину программы больше, чем необходимо.

Тем не менее отметим, что добавляемые к программе модули могут сами иметь ссылки на неопределенные глобальные имена. Эти имена должны быть добавлены к списку поиска, и в этом случае может потребоваться включение в программу дополнительных модулей.

В операционной системе VAX/VMS существуют три способа подключения библиотечных подпрограмм. Первый способ применяется для подключения системных подпрограмм, таких как SYS$QIOW. Эти подпрограммы представляют собой постоянные резидентные части операционной системы; в системном пространстве памяти им назначены фиксированные адреса. Следовательно, компоновщик получает только ссылки на адреса подпрограмм. К программе не добавляется никакого кода.

Второй способ используется, когда в подпрограмму необходимо включать в код, генерируемый компоновщиком. Объем памяти, занимаемый программой, будет увеличен, чтобы сохранить место для библиотечной подпрограммы. Копия скомпонованной программы помещается в файл выполняемого кода.

Третий способ занимает промежуточное положение между двумя первыми. В старых вычислительных системах второй способ применялся для всех несистемных подпрограмм. Это означало, что если имелось 100 программ, в которых использовалась подпрограмма вычисления квадратного корня, то необходимо было иметь 100 копий этих подпрограмм, занимающих пространство в файлах этих программ. Еще хуже, если бы необходимы были также многочисленные копии бо́льших по объему подпрограмм ввода-вывода, автоматически вызываемых всякий раз, когда в программах на Паскале или Фортране встречались операторы WRITELN или PRINT или им подобные. Чтобы сохранить файловое пространство, для каждой из этих наиболее важных подпрограмм образуется только один выполняемый образ. Эти образы хранятся в библиотеке *разделяемых образов*. Во время подготовки программы пользователя к выполнению все необходимые разделяемые образы добавляются к выполняемому образу программы.

15.4. ФАЙЛЫ ВЫПОЛНЯЕМЫХ ОБРАЗОВ

Всякий раз, когда выполняется команда RUN, система будет загружать вашу программу, используя файл типа .EXE или иначе – файл выполняемого образа, сгенерированный компоновщиком. Поэтому файлы типа .EXE должны содержать всю необходимую информацию для загрузки и выполнения вашей программы. Формат файлов типа .EXE на самом деле достаточно сложный, так как файл включает значительный объем информации для изменения адресов, относящихся к разделяемым образам или используемых в них. Однако если это проигнорировать, то можно значительно упростить обсуждение. Чаще всего файлы типа .EXE являются образом, повторяющим байт в байт программу пользователя. Имеется два существенных исключения. Одно из них проявляется тогда, когда включаются разделяемые образы, другое – когда в программе есть большие неинициализированные массивы данных, такие как массивы, заданные директивой .BLKL, и т. п. Для сохранения файлового пространства файлы типа .EXE содержат коды для пропуска этих областей. Если не затрагивать вопросы, касающиеся разделяемых образов и больших массивов, то можно упростить формат файлов типа .EXE, сведя его к следующему. (Отметим, что данное обсуждение опирается на текущую версию ОС VAX/VMS, используемую во время написания книги. Нет гарантий, что рассматриваемый формат будет таким же, как здесь описано. Читателю рекомендуется получить дампы файлов типа .EXE и проверить формат этих файлов.)

Благодаря побайтовой организации файлов типа .EXE в них используется одна из простых структур записей. Файл типа .EXE представляет собой файл последовательного доступа с записями фиксированной длины. Длина фиксированных записей составляет 512 байтов; эти записи загружаются в 512-байтовые блоки памяти. (Отметим, что десятичное число 512 эквивалентно шестнадцатеричному числу 200, кроме того, это число соответствует

длине физических блоков на большинстве внешних запоминающих устройств большой емкости ЭВМ VAX.)

Помимо текста программы в файле выполняемого образа содержится следующая управляющая информация:

1) длина программы;
2) начальный адрес программы;
3) длина программных секций и коды защиты для этих секций;
4) место расположения стека и других системных областей;
5) информация для отладчика (если требуется).

Данная информация имеет такую же структуру записей фиксированной длины, как и **остальная информация в файле.** Полный формат файла выполняемого образа приводится ниже.

1. **Заголовок образа.** Эта запись содержит такую информацию, как старший адрес памяти, используемый для загрузки программы, начальный адрес, называемый также адресом передачи управления, и символьную строку – имя программы, определенное в директиве .TITLE. (Имя программы начинается с первого символа, следующего после директивы .TITLE. Длина имени программы, как и длина всех имен в ассемблере, ограничивается до 31 символа.) В записи заголовка содержатся также и другие данные, которые связывают эту запись с информацией, содержащейся в последующих блоках.

2. **Переменное число записей текста.** Данные записи содержат побайтовое содержимое памяти, загружаемое загрузчиком. Загрузка программы начинается с адреса ^X200 и продолжается до тех пор, пока не будет считан старший адрес, как указано в записи заголовка.

3. **Записи вспомогательной информации.** Записи содержат информацию о программных секциях и таблицу имен, **используемых отладчиком.**

Три основные порции информации, содержащиеся в записи заголовка, имеют следующий формат:

1. **Старший адрес.** Адрес находится в длинном слове, расположенном со смещением ^X2C байтов от начала записи. Длинное слово задает старший адрес, используемый в тексте программы, и в действительности сообщает загрузчику, сколько записей текста необходимо считать.

2. **Адрес передачи управления.** Адрес помещен в длинное слово, расположенное со смещением ^X34 байта от начала записи. Длинное слово задает адрес, который должен быть указан при вызове программы для того, чтобы начать ее выполнение. Напомним, что программа пользователя вызывается с помощью инструкций CALLS или CALLG. Поэтому по адресу передачи управления должна находиться маска сохранения регистров, занимающая одно слово, за которой следует первая выполняемая инструкция программы пользователя.

3. **Имя программы.** Имя программы представляет собой последовательность байтов, смещенную от начала записи на ^X4C байта. Первый байт последовательности задает число байтов имени. Этот указатель длины имени должен иметь значение меньше или равное 31. Все остальные байты являются символами в коде ASCII.

Такое упрощенное понимание структуры заголовка и записей текста достаточно, чтобы написать программу, которая может загрузить и выполнить программу пользователя (см. п. 6 упр. 15.2).

ЗАГРУЗКА В ВИРТУАЛЬНУЮ ПАМЯТЬ

В ЭВМ с традиционной архитектурой код программы из файла типа .EXE должен быть считан или загружен в память перед выполнением программы. Однако в архитектуре ЭВМ с виртуальной памятью процесс загрузки может модифицироваться, так как обычно загрузки в память программы целиком не требуется. Отметим, что в системе виртуальной памяти имеется таблица, в которой каждой странице соответствует своя запись. Каждая запись таблицы содержит следующую информацию (см. гл. 14).

1. Является ли виртуальная страница активной и,таким образом, отображается ли она в настоящий момент времени на страницу реальной памяти.

2. Если страница активна, то в какое место реальной памяти она отображается.
3. Местонахождение блока, используемого для сохранения копии страницы, на дисковой системе.

Когда пользователь вводит команду RUN, ОС VAX/VMS запускает программу, называемую *активатором образа*. Активатор образа не загружает программу, а копирует страницы программы в специальную область виртуальной памяти на диске, при необходимости выполняя соответствующую модификацию. Как уже было отмечено, в программе может потребоваться модификация адресов. Если страница находится в программной секции, предназначенной только для чтения и не требующей модификации, страница может отображаться прямо в соответствующий блок файла типа .EXE. Активатор образа делает также неактивными отображения всех страниц программы в реальную память. Поэтому в начале выполнения программы сразу возникает прерывание по отсутствию страницы, обслуживаемое процедурой обмена страниц, которая в конечном счете загружает программу пользователя.

15.5. ВЫПОЛНЕНИЕ ПРОГРАММЫ

РАБОТА ПРОЦЕССОРА

В гл. 3 цикл выполнения инструкции рассматривался в общем виде; здесь он будет рассмотрен более детально. На рис. 15.2 приведена блок-схема цикла выполнения инструкции для ЭВМ семейства VAX, на самом деле такой цикл выполняется на многих ЭВМ.

ЭВМ семейства VAX имеют инструкции, состоящие из различного числа байтов. Может показаться, что изменение программного счетчика не является вполне определенной операцией. Однако, как показано на блок-схеме, предполагается, что все инструкции имеют длину один байт, но при выполнении инструкции может потребоваться выборка дополнительных байтов из программы. Следовательно, изменение программного счетчика представляет собой следующую операцию: PC ← PC + 1.

Для декодирования инструкции по существу требуется получить однобайтовый код операции и использовать его, чтобы выбрать одну из 256 частей электронной схемы, необходимую для выполнения частных операций, определенных для данной инструкции. Однако если реализацию каждой инструкции осуществлять без учета реализации других инструкций, то построить процессор было бы очень дорого. Для снижения экономических затрат при проектировании процессора необходимо принять во внимание, что многие инструкции имеют общие функции. Например, в большинстве инструкций ЭВМ VAX используются одни и те же режимы адресации. Поэтому некоторые функции, такие как адресация, могут быть реализованы для многих различных инструкций с помощью только одной части электронной схемы процессора. Работа таких частей электронной схемы очень похожа на работу подпрограмм.

Во время работы каждая инструкция активизирует выбранную последовательность действий, выполняемых различными обрабатывающими электронными схемами процессора. Эта выбранная для инструкции последовательность операций называется *микропрограммой* или *микрокодом*. Микропрограммы для некоторых инструкций могут быть очень сложными. Например, такие инструкции, как инструкции умножения, часто реализуются с помощью повторяющейся активизации электронных схем суммирования и сдвига. Таким образом, микропрограммы могут, подобно обычным программам, включать циклы. Конечно, использование циклов будет замедлять выполнение операций. Поэтому для достижения высокого быстродействия отдельные большие ЭВМ имеют в составе процессора дорогие быстродействующие электронные схемы умножения, выполняющие умножение двоичных чисел с помощью логических операций без циклов. Тем не менее такие инструкции, как POLYF или POLYD, являются слишком сложными, чтобы их можно было эффективно реализовать без циклов. В табл. 15.1 даны сравнительные оценки реализации умножения на различных ЭВМ.

Таблица 15.1. Реализация умножения

Быстродействие	Стоимость	Реализация	Тип процессора
Высокое	Высокая	Электронная схема	VAX-11/780
Среднее	Средняя	Микропрограммная с циклами	PDP-11/23
Низкое	Низкая	Программная с циклами	Z80 или 6502

Чтобы упростить декодирование, инструкции часто разделяются на группы, а сходные числовые значения кода операции будут показывать принадлежность инструкции к группе. Например, рассмотрим следующий список кодов операций:

```
ADDF2   40      ADDF3   41
ADDD2   60      ADDD3   61
ADDB2   80      ADDB3   81
ADDW2   A0      ADDW3   A1
ADDL2   C0      ADDL3   C1
```

Применение числовых кодов операций позволяет упростить процесс декодирования, что, в свою очередь, может снизить стоимость и повысить быстродействие ЭВМ. Однако, становясь более мощными, современные ЭВМ должны иметь большое число сложных инструкций, и тогда при реализации приходится искать компромисс. В конечном счете числовые значения кодов операций могут потерять свой очевидный смысл, что часто и происходит.

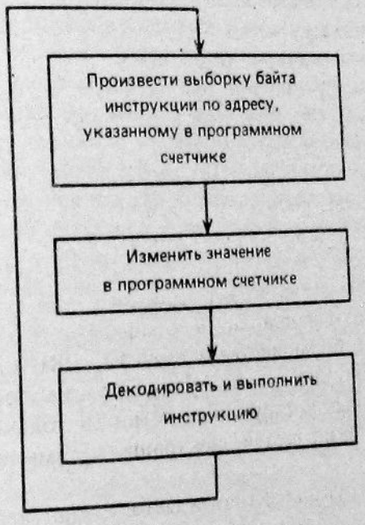


Рис. 15.2. Цикл выполнения инструкции

В этом случае процессор использует код операции как адрес постоянной памяти, в которой содержится таблица перевода, преобразующая 8-битовые коды операций в более длинные коды, а эти коды уже непосредственно устанавливают связь с микрокодом для декодируемой инструкции.

ЭВМ семейства VAX имеют более 256 инструкций. Такое число инструкций означает, что недостаточно иметь однобайтовый код операции для однозначной идентификации всех инструкций. Решение состоит в том, что коды операций ^XFD, ^XFE и ^XFF используются как переключающие коды, которые сообщают процессору, чтобы он произвел выборку второго байта для второго уровня декодирования. Данный подход эффектив-

но позволяет увеличивать число кодов операций без каких-либо особых ограничений. В настоящее время на ЭВМ семейства VAX расширение кодов операций применяется довольно редко. Все 2-байтовые коды операций используются для выполнения операций над числами с плавающей точкой в форматах тип G и H, в качестве переключающего кода служит ^XFD (см. гл. 12).

ПОЗИЦИОННО-НЕЗАВИСИМЫЙ КОД

Одним из особых преимуществ архитектуры ЭВМ семейства VAX является возможность использования программного счетчика как обычного регистра смещения. Это позволяет адресовать данные и инструкции относительно текущего места выполнения программы. Преимущество заключается в том, что если программа адресует свои собственные ячейки относительно текущего значения в программном счетчике, то относительные адреса не изменяются при перемещении программы.

Например, предположим, что при выполнении инструкции программы в программном счетчике находится значение ^X0740 и должен быть осуществлен доступ к данным, расположенным по адресу ^X0A14; смещение от текущего значения программного счетчика к этому адресу равно ^X0A14 – ^X0740 = ^X03D4. Фактически тот же самый способ адресации используется на ЭВМ семейства VAX в режиме CF (см. гл. 7). Теперь, если программа перемещается на ^X0200 байтов, то адрес в программном счетчике и адрес данных соответственно изменяется на ^X0940 и ^X0C14, но разница между адресами останется прежней – ^X03D4. В результате данная программа может быть загружена в какое-либо место памяти без требования модификации этой отдельной инструкции. Если вся программа написана таким образом, то говорят, что ее код *позиционно-независимый*. Для того чтобы код программы был позиционно-независимым, программист должен специально следить за тем, чтобы применялись только те инструкции и режимы адресации, которые являются позиционно-независимыми. Следующий набор правил отмечает наиболее важные моменты.

Правило 1. Адресация ячеек программы. Как показано выше, доступ ко всем адресам программы должен осуществляться с помощью режима относительной адресации. Однако так как данный режим адресации является обычным режимом, то не накладывается никаких специальных требований. Инструкция MOVL A,B работает правильно.

Правило 2. Адресация фиксированной области памяти или регистров устройств, имеющих фиксированные адреса. Режим относительной адресации больше не срабатывает, так как адрес, получаемый при сложении значения в регистре PC с фиксированным смещением, изменится при перемещении программы. Однако можно применять режим абсолютной адресации, поскольку он требует указания исполнительного фиксированного адреса. Режим абсолютной адресации может быть задан, если поместить знаки @# перед адресом. Например, предположим, что адреса PORG и PLOC фиксированы в памяти, а адрес A находится в самой программе. В этом случае обе инструкции TSTB @#PORG и MOVB A, @#PLOC являются позиционно-независимыми.

Правило 3. Инструкции условных и безусловных переходов. В инструкциях условных переходов адресация осуществляется относительно программного счетчика и перекрывает небольшой диапазон адресов, поэтому не возникает дополнительных проблем. В инструкциях безусловных переходов используются обычные операнды и поэтому применяются режимы либо относительной, либо абсолютной адресации, в зависимости от того, осуществляется переход внутри самой программы или в фиксированное место в памяти.

Правило 4. Адресация массивов в программе. Массивы вызывают особую сложность, поскольку в инструкциях с индексным или регистровым косвенным режимами адресации требуется исполнительный адрес. Вспомним, что в гл. 7 рассматривались несколько способов доступа к элементам массива. Индексный режим адресации, как, например, в инструкции CLRB A[R0], будет позиционно-независимым, даже если адрес A является перемещаемым, поскольку исполнительный адрес вычисляется в два приема. Сначала вычисляется адрес A как смещение относительно текущего значения регистра PC, после чего он складывается с содержимым регистра R0 для получения доступа к индексированному элементу массива. В конце гл. 7 были представлены такие режимы адресации, как косвенные, режи-

мы со смещением, а также режим с автоувеличением. В этих режимах адрес помещается в регистр общего назначения. Отметим, что обычным способом загрузки адреса в регистр является следующий:

```
MOVAB    A, R0
```

Эта инструкция позиционно-независимая, если адрес A переместимый; если адрес A – абсолютный, то инструкцию следует написать так:

```
MOVAB    @#A, R0
```

Поскольку такой способ адресации поддерживается на ЭВМ VAX, этим правилам легко следовать и трудно написать программу, которая не была бы позиционно-независимой. Тем не менее требуется тщательно следить за данными. В таких массивах данных, как дескрипторы и списки аргументов программ, часто появляются адреса. Если переместимые адреса помещаются в массивы данных, то программа не будет полностью позиционно-независимой. Например, программа не будет позиционно-независимой, если в ней содержится директива .ASCID. Следует избегать применения директивы .ASCID, употребляя взамен директиву .ASCII и формируя дескриптор во время выполнения. Например, могло быть сделано следующее:

```
MSGD:    .WORD     20$-10$,0
         .LONG     0
10$:     .ASCII    "ТЕКСТ СООБЩЕНИЯ"
20$:     .
         .
         .
         MOVAB     MSGD+8, MSGD+4
         .
         .
         .
```

Как показано в этом фрагменте, инструкция MOVAB позиционно-независимая. С помощью этой инструкции соответствующий адрес помещается в дескриптор. Чтобы избежать этого и упростить написание программ, ОС VAX/VMS обеспечивает для файлов типа .EXE последнюю перед загрузкой модификацию, позволяя использовать для разделяемых образов директивы .ASCID и .ADDRESS. В результате эти директивы могут использоваться в позиционно-независимых (PIC) программных секциях, несмотря на то, что, строго говоря, такие директивы не являются позиционно-независимыми.

Позиционно-независимый код может быть весьма полезен для программ операционной системы. Например, в ОС VAX/VMS системные программы часто обладают способностью загружаться в любое место памяти туда, где есть свободное пространство. Многие из этих программ, такие как программы обслуживания устройств ввода-вывода и программы обслуживания пользователей, при необходимости динамически вызываются в память на любое свободное место.

УПРАЖНЕНИЯ 15.2.

1. Разработайте формат файлов с записями переменной длины, в котором вместо указателей длины записей использовались бы управляющие символы. Для байтов данных должны быть разрешены любые комбинации битов.

2. Напишите программу, осуществляющую запись и чтение файлов в формате, разработанном вами в п. 1.

3. Образуйте несколько файлов с различной структурой записей. Это может быть выполнено с помощью простой программы на Фортране, где в операторе OPEN задаются различные параметры, или с помощью программы, написанной на языке ассемблера, использующей различные макровызовы $FAB и $RAB. После этого, воспользовавшись командой DUMP/HEADER/NOFORMATED, посмотрите управляющие блоки файлов. Попробуйте определить предназначение различных управляющих кодов, используемых в управляющем блоке, и письменно изложите ваши предположения. Проверьте их, получив дампы других файлов.

4. Напишите программу, которая считывает дампы, образованные по команде DUMP, и распечатывает содержимое файла в его первоначальном виде. Программа должна работать с обычными символьными файлами, какие используются для хранения исходного текста программ. Ваша программа может быть написана либо на языке ассемблера, либо на языке высокого уровня. Заметим, что в программе должна быть заложена возможность пропуска заголовков страницы и другой, не относящейся к делу информации, которая может присутствовать в дампе.

5. Напишите программу, имеющую позиционно-независимый код. Программа должна распечатывать несколько строк сообщений для того, чтобы можно было понять, что она работает. После вывода сообщений программа должна скопировать сама себя в массив, размещенный в какой-то области памяти, затем заполнить нулями область памяти, в которой она находилась первоначально, и передать управление на перемещенный адрес начала программы, чтобы вновь выполнить распечатку сообщения, позволяющего убедиться, что код программы действительно позиционно-независимый. Отметим, что области данных также следует перемещать вместе с программой. (См. замечания по использованию директивы .ASCID в конце последнего раздела данной главы.) Особо отметьте следующее: для того чтобы код, порождаемый макроинструкциями $RAB и $FAB, был позиционно-независимым, все параметры в макроинструкциях, представляющих собой адреса, должны быть помещены в блоки доступа во время выполнения программы. К таким параметрам относятся NAM = ⟨FILE.XYZ⟩ и FAB = FABLOC.

6. Напишите программу, в начале которой расположен большой пустой массив, например такой, как .BLKL 5000. После компоновки программы он будет находиться по адресу ^X200. Напишите программу, которая могла бы считать файл типа .EXE простой программы, поместить записи текста этой программы в массив, начинающийся с адреса ^X200, и выполнить считанную программу. В связи с тем, что обычно настройка программы осуществляется так, чтобы программа начиналась с адреса ^X200, загруженную в массив программу для выполнения нужно разместить правильным образом.

ПРИЛОЖЕНИЕ А. ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ, НАПИСАННЫХ НА ЯЗЫКЕ АССЕМБЛЕРА

А.1. ПРОГРАММА НА ЯЗЫКЕ АССЕМБЛЕРА

В данном приложении показано, как можно подготовить, а затем выполнить программу, написанную на языке ассемблера, на вычислительной системе VAX с ОС VAX/VMS. Используемая в качестве примера программа на языке ассемблера приведена ниже; это программа из гл. 1. При завершении выполнения программы содержимое DIF будет равно 8.

```
J:      .BLKL   1
K:      .BLKL   1
DIF:    .BLKL   1
        .ENTRY  ADDRESS,0
        MOVL    #20,J
        MOVL    #12,K
        MOVL    J,DIF
        SUBL2   K,DIF
        $EXIT_S
        .END    ADDRESS
```

А.2. ПОДГОТОВКА ПРОГРАММ НА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЕ VAX

На вычислительных системах VAX ввод программ осуществляется с помощью интерактивного терминала. Существует два основных типа терминалов – *телетайпы* и *видеотерминалы* или дисплеи. Телетайпы внешне похожи на пишущие машинки: вся информация, поступающая на телетайп, печатается на бумаге. Видеотерминал по внешнему виду напоминает небольшой телевизор с подсоединенной к нему клавиатурой. Терминалы любого типа могут быть использованы для подготовки программ на языке ассемблера.

В этом приложении будет рассмотрено, как выполняются программы с помощью систем интерактивной и пакетной обработки. Следующие несколько разделов предназначены для пользователей, работающих в интерактивном режиме, но они полезны также для пользователей, работающих в пакетном режиме, так как многое из изложенного является общим как для интерактивной, так и для пакетной обработки информации.

А.3. НАЧАЛО СЕАНСА РАБОТЫ НА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЕ VAX

Терминалы могут подключаться к вычислительной системе VAX различными способами. Например, некоторые терминалы могут быть подключены к вычислительной системе VAX постоянно. Работу за таким терминалом можно начать сразу же после его включения. Другие терминалы подключаются к вычислительной системе по телефонным каналам. При

этом может потребоваться набор правильного телефонного номера, а затем подключение телефона к акустическому устройству сопряжения или модему. В любом случае, если терминал подключен к вычислительной системе, пользователь обычно нажимает клавишу с маркировкой RETURN, CARRIAGE RETURN или CR (возврат каретки). Может потребоваться неоднократное нажатие этой клавиши.

Когда ОС VAX/VMS обнаруживает, что была нажата клавиша RETURN, она инициирует последовательность действий, необходимых для начала сеанса работы. Система запрашивает назначенные пользователю имя, выводя на терминал подсказку Username, и пароль на подсказку Password. Если пользователь зарегистрирован в системе, то на терминале появится приветствующее его сообщение, а после сообщения – знак доллара. Для пользователя, зарегистрированного в системе под именем JOHNDOE, которому известен пароль GUESS, последовательность действий для входа в систему может выглядеть следующим образом:

```
Username: JOHNDOE
Password: ——
    Welcome to VAX/VMS version Vx.x
$
```

В этом примере все, что вводится пользователем, выделено подчеркиванием. Такое соглашение об обозначении информации, вводимой пользователем, будет соблюдаться и в дальнейшем. Обратите внимание, что операционная система не печатает пароль GUESS. Такая мера предосторожности препятствует тому, чтобы кто-то подсмотрел пароль и воспользовался им для несанкционированного входа в систему. После набора слов JOHNDOE и GUESS пользователь каждый раз должен нажимать на клавишу RETURN. Обычно вычислительная система VAX не обрабатывает ничего из того, что вводится пользователем до тех пор, пока не будет нажата клавиша RETURN.

А.4. КОМАНДЫ ОПЕРАЦИОННОЙ СИСТЕМЫ VAX/VMS

Знак доллара $, появляющийся на строке после приветственного сообщения, является приглашением к вводу, выводимому операционной системой VAX/VMS. Операционная система VAX/VMS представляет собой управляющую программу, которая координирует и руководит выполнением всех других программ. Знак доллара служит признаком того, что пользователь взаимодействует с ОС VAX/VMS, а не с другой программой и ему следует вводить *команды* ОС VAX/VMS.

Команда HELP (помощь) – одна из команд операционной системы VAX/VMS. Если пользователь вводит команду HELP, то на терминал будет выведена следующая информация:

```
$ HELP
  Information available:

  ACCOUNTING ALLOCATE   ANALYZE    APPEND     ASSIGN     ATTACH     BACKUP
  BASIC      BLISS      CANCEL     CC         CLOSE      COBOL      CONTINUE
  CONVERT    COPY       CORAL      CREATE     DBO        DDL
  DEALLOCATE DEASSIGN   DEBUG      DECK       DEFINE     DELETE     DEPOSIT
  DIFFERENCES           DIRECTORY  DISMOUNT   DUMP       EDIT       EOD
  EOJ        Errors     EXAMINE    EXIT       FDL        FORTRAN    GOTO
  HELP       IF         INITIALIZE INQUIRE    JOB        Lexical    LIBRARY
  LINK       Login      LOGOUT     MACRO      MAIL       MCR        MERGE
  MESSAGE    MONITOR    MOUNT      ON         OPEN       PASCAL     PASSWORD
  PATCH      PHONE      PLI        PRINT      Procedure  PURGE      Queues
  READ       RENAME     REPLY      REQUEST    RMS        RTL        RUN
  RUNOFF     SEARCH     SET        SHOW       SORT       SPAWN      Specify
  START      STOP       SUBMIT     Symbol_Assign         SYNCHRONIZE
  System     TECO       TYPE       UNLOCK     WAIT       WRITE

Topic?
$
```


В этом и последующих примерах распечаток программ точное содержание и формат выдачи в различных системах могут отличаться.

После печати списка команд ОС VAX/VMS будет выведен запрос Topic? ("Предмет обсуждения"). Если пользователь на этот запрос введет название одной из команд, перечисленных в приведенном выше списке, то он сможет получить дополнительную информацию об интересующей его команде. В данном случае пользователь просто нажал на клавишу RETURN. Такой ответ пользователя завершает работу команды HELP, и управление возвращается обратно ОС VAX/VMS. В свою очередь, операционная система дает пользователю приглашение к вводу, печатая знак доллара.

Для подготовки программ на языке ассемблера и запуска их на выполнение требуется всего лишь несколько команд ОС VAX/VMS. Наиболее важными командами являются команды COPY, CREATE, DELETE, DIRECTORY, EDIT, LINK, LOGOUT, MACRO, PRINT, PURGE и TYPE. Дополнительные сведения об этих командах можно найти в приложении Г.

А.5. СОЗДАНИЕ ФАЙЛА

Большинство из названных выше команд предназначены для работы с *файлами*. Файл представляет собой упорядоченную совокупность записей, например строк текста. Каждый из описываемых здесь файлов является программой на некотором языке. Обычно файлы пользователя хранятся на устройстве внешней памяти, называемом диском. Команда DIRECTORY применяется для того, чтобы получить список имен файлов, находящихся на диске пользователя. Как показано в следующем примере, пользователь на текущий момент времени не имеет каких-либо файлов:

```
$ DIR
No files found
$
```

Отметим, что вместо полного названия команды DIRECTORY было введено сокращение DIR. Названия большинства команд ОС VAX/VMS могут сокращаться до трех или четырех букв.

Программисты могут создавать файлы с помощью команды CREATE. Пользователь вводит команду CREATE EXAMPLE.MAR, завершая ввод нажатием на клавишу RETURN; далее вводится текст программы, который завершается нажатием комбинации клавиш CTRL/Z. Ниже дан пример создания файла:

```
$ CREATE EXAMPLE.MAR
J:      .BLKL   1
K:      .BLKL   1
DIF:    .BLKL   1
        .ENTRY  ADDRESS,0
        MOVL    #20,J
        MOVL    #12,K
        MOVL    J,DIF
        SUBL2   K,DIF
        $EXIT_S
        .END    ADDRESS
^Z
$
```

В команде CREATE указывается имя файла, выбираемое пользователем; в рассматриваемом примере именем файла является EXAMPLE. Следующее за именем файла сочетание символов .MAR обозначает тип файла. В данном случае тип .MAR означает, что файл содержит программу на языке ассемблера (.MAR – сокращение от MAcRo). Такой тип файла следует использовать для программ, написанных на языке ассемблера ОС VAX/VMS. Символ ^Z вводится при нажатии клавиши с маркировкой CONTROL или CTRL, как если бы это была клавиша переключения регистра, и одновременном нажатии клавиши с буквой

Z. Символ, получаемый при нажатии комбинации клавиш CTRL/Z, является специальным символом, служащим признаком конца файла.

При вводе команд или текста программ часто возникают опечатки. Существуют два способа исправления опечаток. Первым является применение клавиши RUBOUT или DELETE. Нажатие на эти клавиши вызывает стирание предыдущего символа. Если для работы используется видеотерминал, можно увидеть, что удаляемый символ на самом деле исчезнет с экрана. На телетайпе печатается обратная косая черта, за которой следует удаленный символ. При желании таким способом можно удалить все символы до начала строки. Когда пользователь начинает снова вводить символы, перед ними на телетайпе печатается вторая обратная косая черта. Например, если пользователь напечатал

```
HEXY(delete)(delete)LLO
```

то в результате получилось бы то же самое, как если бы он просто напечатал HELLO. На видеотерминале было бы отображено слово HELLO, но на телетайпе это выглядело бы следующим образом:

```
HEXY\XY\LLO
```

Второй способ исправления ошибок состоит в том, что удаляется вся строка целиком и выполняется ее повторный ввод. Удаление всей строки целиком осуществляется при нажатии комбинации клавиш CTRL/U (одновременно нажать клавишу CTRL и клавишу с буквой U).

После того как с помощью команды CREATE создан файл, можно ввести команду DIRECTORY. Выводимая по этой команде информация позволит убедиться, что файл действительно был создан:

```
$ DIR

Directory DISK$USER:[JOHNDOE]

EXAMPLE.MAR;1

Total of 1 file.
```

Выведенная информация показывает, что на диске с именем DICK$USER в каталоге пользователя JOHNDOE содержится файл EXAMPLE.MAR;1. Символы ;1 в наименовании файла EXAMPLE.MAR;1 обозначают номер версии файла. Если бы содержимое этого файла модифицировалось, то на диске было бы два файла; EXAMPLE.MAR;1 и EXAMPLE.MAR;2. Когда в команде ОС VAX/VMS номер версии файла не указывается, то большинство команд выбирают последнюю версию, т. е. файл с самым большим номером версии. Например, по следующей команде TYPE на терминал будет выведено содержимое последней версии файла EXAMPLE.MAR:

```
$ TYPE EXAMPLE.MAR
J:      .BLKL   1
K:      .BLKL   1
DIF:    .BLKL   1
        .ENTRY  ADDRESS,0
        MOVL    #20,J
        MOVL    #12,K
        MOVL    J,DIF
        SUBL2   K,DIF
        $EXIT_S
        .END    ADDRESS
$
```


Команда PRINT может использоваться для вывода содержимого файлов на быстродействующее устройство печати – принтер.

Ввод текстов программ с помощью команды CREATE довольно проблематичен, так как требуется, чтобы пользователь создал файл, не делая ошибок. Практически это выполнимо только для очень небольших программ. Чтобы вводить большие программы, пользователь должен иметь возможность исправлять ошибки, не перепечатывая всю программу целиком. Такая коррекция ошибок может быть сделана с помощью программы, называемой редактором текста. В ОС VAX/VMS один из доступных редакторов текста имеет название EDT. Более подробно редактор EDT описывается в приложении Д, где рассматриваются также особые свойства редактора EDT, которые могут использоваться при работе с терминалами типа VT52 или с терминалами серий VT100/VT200.

А.6. ВЫПОЛНЕНИЕ ПРОГРАММЫ

Программа на языке ассемблера может быть подготовлена для выполнения с помощью следующих трех команд ОС VAX/VMS:

```
$ MACRO/DEBUG EXAMPLE
$ LINK/DEBUG EXAMPLE
$ RUN EXAMPLE

          VAX-11 DEBUG Version x.x

%DEBUG-I-INITIAL, language is MACRO, module set to '.MAIN.'
DBG>
```

Ключ /DEBUG в первых двух командах означает, что пользователь хочет воспользоваться программой-*отладчиком*, представляющей различный сервис при выполнении программы. Команда MACRO служит для вызова на выполнение ассемблера MACRO. Ассемблер считывает программу на языке ассемблера из файла EXAMPLE.MAR и создает вариант программы в машинном коде, называемый *объектным модулем*. По команде LINK вызывается на выполнение программа-*компоновщик* (linker). Компоновщик считывает объектный модуль и создает готовую для выполнения программу на машинном языке, называемую *выполняемым образом* (executable image). По команде RUN начинается выполнение выполняемого образа под управлением отладчика.

После вывода двух информационных строк отладчик печатает символы DBG>. Таким способом отладчик дает приглашение пользователю для ввода команд отладчика. Как можно было бы предположить в этой ситуации, по команде HELP отладчиком будет выведен на терминал список команд отладчика. Однако в данном случае пользователь вводит команду GO, по которой начинается выполнение программы в машинном коде, представляющей собой выполняемый вариант программы на языке ассемблера. Результат этого действия выглядит следующим образом:

```
DBG>GO
routine start at .MAIN.\ADDRESS
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

Сообщение "normall successful completion" (нормальное успешное завершение) означает, что программа выполнилась до оператора $EXIT_S без ошибок.

Команда отладчика EXAMINE применяется для вывода на терминал содержимого различных символических адресов (переменных) в программе. Например, команда E/DEC J дает указание отладчику напечатать в десятичном виде число, содержащееся по символическому адресу J:

```
DBG>E/DEC J
.MAIN.\J:  20
DBG>E/DEC K
.MAIN.\K:  12
DBG>E/DEC DIF
.MAIN.\DIF:  8
DBG>
```

Как показали три команды EXAMINE, содержимым символических адресов J, K и I являются соответственно числа 20, 12 и 8. Число 8 является ожидаемым результатом.

Команда EXIT используется для возврата управления программой-отладчиком DEBUG обратно ОС VAX/VMS. Если после этого пользователь вводит команду DIRECTORY, то на терминал будет выведено следующее:

```
DBG>EXIT
$ DIR

Directory DISK$USER:[JOHNDOE]

EXAMPLE.EXE;1      EXAMPLE.MAR;1      EXAMPLE.OBJ;1

Total of 3 files.
```

В приведенной распечатке каталога в файле EXAMPLE.MAR содержится программа на языке ассемблера, созданная с помощью команды CREATE. Файл EXAMPLE.OBJ содержит объектный модуль, созданный по команде MACRO, а файл EXAMPLE.EXE, созданный по команде LINK, содержит выполняемый образ.

И наконец, команда операционной системы LOGOUT служит для окончания сеанса работы с вычислительной системой:

```
$ LOGOUT
JOHNDOE   logged out at 25-JUL-1985 11:56:50.32
```

А.7. ОШИБКИ ПРОГРАММИРОВАНИЯ И ОТЛАДКА

В приведенном выше примере ошибки отсутствуют. Но обычно вновь созданные программы содержат ошибки, и программист должен их обнаружить и исправить. Ошибки в программах на языке ассемблера могут вызвать появление сообщений об ошибках, выдаваемых ассемблером, компоновщиком или ОС VAX/VMS во время выполнения программы, или же привести к неверным результатам. Последствия некоторых типов ошибок показаны ниже.

Предположим, что третья строка программы была введена неправильно:

```
DIF     .BLKL   1
```

Заметим, что здесь после символа DIF пропущено двоеточие (:). Такая ошибка обнаруживается ассемблером, который выводит следующее сообщение об ошибке:

```
$ MACRO/DEBUG EXAMPLE
                                0008    3 DIF     .BLKL   1
%MACRO-E-UNRECSTMT, Unrecognized statement
```

Подобное сообщение могло бы появиться, если в пятой строке программы в мнемоническом обозначении кода операции MOVL была бы пропущена буква и код операции был бы введен как MOV. Обычно ошибки, о которых сообщает ассемблер, легко обнаружить и устранить, так как ассемблер печатает строки программы, содержащие ошибки.

Компоновщик может также давать сообщения об ошибках. Например, предположим, что седьмая строка программы была введена так:


```
MOVL    J,DIFF
```

Обратите внимание, что символический адрес DIF был введен с опечаткой как DIFF. Ассемблер рассматривает DIFF как неопределенное имя. Однако он не может выдать сообщение об ошибке, так как предполагает, что компоновщик будет искать определение символического адреса DIFF. Если компоновщик не сможет найти определение символического адреса DIFF, то он напечатает, например, следующее сообщение об ошибке:

```
$ MACRO/DEBUG EXAMPLE
$ LINK/DEBUG EXAMPLE
%LINK-W-NUDFSYMS, 1 undefined symbol
%LINK-I-UDFSYM,     DIFF
%LINK-W-USEUNDEF, undefined symbol DIFF referenced
    B     in psect . BLANK . offset %X0000001A
        in module .MAIN. file DISK$USER:[JOHNDOE]EXAMPLE.OBJ;1
```

Обычно ошибки, вызванные появлением неопределенных имен, легко находятся, так как компоновщик сообщает обо всех неопределенных именах.

Ошибки, возникающие во время выполнения программы, находить существенно труднее. Например, предположим, что пятая строка программы была введена с ошибкой, что выглядело бы следующим образом:

```
MOVL    20,J
```

Обратите внимание, что перед числом 20 был пропущен знак номера #. Ассемблирование и компоновка программы пройдут без ошибок. Тем не менее при выполнении программы ОС VAX/VMS выведет сообщение об ошибке

```
$ MACRO/DEBUG EXAMPLE
$ LINK/DEBUG EXAMPLE
$ RUN/DEBUG EXAMPLE
                        VAX-11 DEBUG Version 3.2-1

%DEBUG-I-INITIAL, language is MACRO, module set to '.MAIN.'
DBG>GO
routine start at .MAIN.\ADDRESS
%SYSTEM-F ACCVIO, access violation, reason mask=00, virtual address=00000014,
PC=0000020E, PSL=03C00000
```

где сообщается пользователю о том, что число 20 было интерпретировано как адрес. Представленное в шестнадцатеричном виде это число можно найти в строке сообщения "virtual address = 00000014", в котором оно является виртуальным адресом (см. гл. 2, 3 и 4).

Пропущенный знак номера # может также явиться причиной совершенно неверных результатов. Подобные ошибки возникают в случае отсутствия соответствующей инициализации переменной. Иногда, чтобы найти ошибку, необходимо просмотреть символ за символом всю программу на языке ассемблера. Отладка программ, написанных на языке ассемблера, может быть значительно более трудной, чем отладка программ на языках высокого уровня.

Применение команды отладчика STEP может помочь при нахождении таких ошибок. Команда STEP вводится вместо команды GO. Команда STEP вызовет выполнение одной инструкции, а не всей программы полностью, в отличие от команды GO. Отладчик будет показывать действие выполняемых последовательно инструкций. Команда STEP может выполняться неоднократно, а между шагами выполнения можно использовать команду EXAMINE для отслеживания продвижения по программе. Если на очередном шаге выполнения встречается вызов подпрограммы, то подпрограмма выполняется как один шаг. Если желательно выполнить подпрограмму в режиме пошагового выполнения, то применяется команда STEP/INTO.

А.8. ВЫПОЛНЕНИЕ ОТЛАЖЕННОЙ ПРОГРАММЫ

Когда программа полностью отлажена, желателен завершающий прогон программы без применения отладчика. До тех пор, пока не будут использоваться программы ввода-вывода, описанные в гл. 5, для ввода и вывода должен применяться отладчик. Для прогона программы без отладчика в командах MACRO и LINK надо опустить ключ /DEBUG. Листинг может быть получен, если в команде MACRO указать ключ /LIST. Команды теперь имеют следующий вид:

```
$ MACRO/LIST EXAMPLE
$ LINK EXAMPLE
$ RUN EXAMPLE
ВВОД И ВЫВОД ПРОГРАММЫ
```

Отметим, что ключ /LIST не вызовет появления на терминале дополнительной информации. При указании этого ключа будет создан файл с именем EXAMPE.LIS. В этом файле содержится листинг ассемблера, который впоследствии может быть напечатан с помощью команд TYPE или PRINT.

А.9. ПОЛУЧЕНИЕ ДОКУМЕНТИРОВАННОЙ КОПИИ

При работе с видеотерминалом сложность заключается в том, что во время вывода информации нельзя сохранить выводимую информацию, чтобы потом показать ее кому-либо.

Существует два простых способа получения документированной копии. Первым способом является запуск программы на выполнение с телетайпа; к таким типам терминалов относятся терминалы LA36 или LA100. Как правило, в вычислительных центрах в каждой группе терминалов есть один терминал типа телетайпа. Видеотерминалы применяются при создании файлов с текстами программ и для отладки, тогда как заключительный прогон программы может быть сделан с помощью телетайпа. При использовании этого способа обычно желательно распечатать листинг программы. Для этого вводятся две команды:

```
$ MACRO/LIST EXAMPLE
$ TYPE EXAMPLE
```

Второй способ получения документированной копии состоит в том, чтобы выполнить программу пользователя в режиме пакетной обработки. Обычно вывод пакетов заданий печатается на устройстве системной печати. Пакетные задания могут быть сформированы посредством создания командного файла. Командный файл состоит из команд ОС VAX/VMS, требуемых для выполнения задания. Каждой команде ОС VAX/VMS предшествует знак доллара $. Данные программы следуют после команды $ RUN. Строки, в которых содержатся данные, не должны начинаться со знака доллара. Командный файл должен иметь тип .COM. В качестве примера можно было бы создать следующий файл с именем EXAMPLE.COM:

```
$ MACRO/LIST EXAMPLE
$ TYPE EXAMPLE
$ LINK EXAMPLE
$ RUN EXAMPLE
строка данных #1
строка данных #2
строка данных #3
```

Перед запуском командного файла в режиме пакетной обработки, чтобы убедиться в том, что он работает, командный файл может быть выполнен с помощью команды

```
$ @EXAMPLE
```


Информация, выводимая программой, появится на терминале. Если все выглядит так, как и предполагалось, то программа может быть отправлена на пакетную обработку командой

```
$ SUBMIT EXAMPLE
```

Вывод программы, который был виден на терминале при запуске на выполнение командой @EXAMPLE, будет направлен на устройство системной печати.

ПРИЛОЖЕНИЕ Б. ПРОГРАММЫ ВВОДА-ВЫВОДА

Приведенные программы могут использоваться как простое средство для выполнения функций ввода-вывода в программах, написанных на языке ассемблера. Эти программы можно применять или для ввода и печати чисел, или для ввода и печати символьных строк. Для включения в программу пользователя их можно скопировать и выполнить их ассемблирование вместе с программой пользователя. Можно поступить и иначе: выполнить ассемблирование программ ввода-вывода только один раз, а их подключение к программам пользователя осуществлять на этапе компоновки, связывая эти программы через глобальные имена.

Всего для выполнения функций ввода-вывода используется пять подпрограмм, а именно IOINIT, RNUM, PNUM, RLINE и PLINE. Все они вызываются с помощью инструкции JSB. Ниже дано описание функций, выполняемых этими подпрограммами.

IOINIT

Подпрограмма IOINIT используется для установки начальных значений в четырех других подпрограммах ввода-вывода. Она вызывается только один раз, но при этом *должна быть* вызвана перед тем, как будет использована какая-либо из этих других подпрограмм. Вызов подпрограммы осуществляется следующим образом:

```
JSB   IOINIT
```

Подпрограмма IOINIT возвращает управление, сохраняя содержимое всех регистров. Кроме того, что она позволяет выполнять другие подпрограммы из данного набора, явным результатом ее работы является печать сообщения об ее успешном завершении.

RNUM

Подпрограмма RNUM предназначается для ввода шестнадцатеричного числа из файла системного ввода. Преобразованное во внутреннее представление число помещается в регистр R0. Содержимое всех других регистров не изменяется. При работе в интерактивном режиме на терминал выдается запрос "ВВЕДИТЕ ШЕСТНАДЦАТЕРИЧНОЕ ЧИСЛО – ". Обращение к подпрограмме следующее:

```
JSB   RNUM
результат возвращается в R0
```

PNUM

Подпрограмма PNUM служит для выполнения печати содержимого регистра R0 (содержимое рассматривается как длинное слово) в шестнадцатеричном и десятичном виде. Содержимое всех регистров не изменяется. Обращение к подпрограмме осуществляется следующим образом:

```
загрузить число в R0
JSB   PNUM
```

RLINE

Подпрограмма RLINE предназначена для чтения строки символов в коде ASCII из файла системного ввода. Введенные символы помещаются в массив, находящийся в программе пользователя. При вызове подпрограммы ей необходимо передать адрес байтового массива, в который будет помещена строка символов, и длину этого массива. Адрес в формате длинного слова должен находиться в регистре R0, а длина массива в формате слова помещается в регистр R1. После того, как подпрограмма RLINE передает управление обратно, в массиве будут находиться все символы, присутствовавшие во вводимой строке, а в регистре R1 будет содержаться в формате слова число, соответствующее количеству в действительности введенных символов. Это число может изменяться от 0 до заданной длины массива, но не может превышать заданную длину массива. В регистре R0 будет находиться в формате слова код ошибки. Если ошибок не было, то регистр R0 будет содержать число 1. Наличие в регистре любого другого числа означает, что произошла ошибка. Наиболее вероятными являются ошибки, связанные с тем, что длина буфера слишком мала, чтобы вместить всю строку, или же была предпринята попытка продолжить ввод после того, как был встречен признак конца вводимых данных. Содержимое всех других регистров не изменяется. Последовательность действий при вызове подпрограммы RLINE приведена ниже.

```
загрузить адрес буфера в R0
предельную длину буфера в R1
JSB   RLINE
R0 содержит код ошибки
R1 содержит число введенных символов
```

PLINE

Подпрограмма PLINE используется для вывода на печать в файл системного вывода строки текста в коде ASCII. В регистр R0 должен быть помещен адрес массива символов, а регистр R1 должен содержать в формате слова число символов в этом массиве. Содержимое всех регистров сохраняется. Так как подпрограмма PLINE сопровождает вывод каждой строки управляющими символами "Возврат каретки" и "Перевод строки", то в выводимой строке эти символы не должны присутствовать, если только нет необходимости вывести дополнительные строчки. Последовательность вызова для подпрограммы PLINE дается ниже.

```
загрузить адрес буфера в R0
число символов в R1
JSB   PLINE
```

ПРОГРАММЫ ВВОДА-ВЫВОДА НА ЯЗЫКЕ АССЕМБЛЕРА

Для использования программ ввода-вывода в программе пользователя приведенные ниже исходные тексты программ могут быть помещены между последней строчкой текста программы пользователя и директивой .END. Программы сопровождаются комментариями, поясняющими их действие. Однако предполагается, что не все будет полностью понятно, пока не будут прочитаны гл. 1 – 11. Для ускорения ввода текста программ можно опустить комментарии. Программы могут быть ассемблированы как отдельный программный модуль, а полученный в результате ассемблирования объектный файл может быть скомпонован с основной программой.

```
RNUMBUFSIZ=80
PNUMBUFSIZ=80                          ;УСТАНОВИТЬ ДЛИНУ БУФЕРОВ 80

                .PSECT  IOROUTINE,LONG          ;FAB И RAB ВЫРОВНЕНЫ
RNUMFILE:       $FAB    FNM=<SYS$INPUT>         ;НА ГРАНИЦУ ДЛИННОГО СЛОВА
PNUMFILE:       $FAB    FNM=<SYS$OUTPUT>,RAT=<CR>
RNUMIREC:       $RAB    FAB=RNUMFILE,UBF=RNUMBUF,USZ=RNUMBUFSIZ,-
                        PBF=RNUMPRO,PSZ=RNUMPROL-RNUMPRO,ROP=<PMT>
PNUMOREC:       $RAB    FAB=PNUMFILE,RBF=RNUMOUT
RLINEREC:       $RAB    FAB=RNUMFILE,-
```



```
                        PBF=RLINEPRO,PSZ=RLINEPROL-RLINEPRO,ROP=<PMT>
PLINEREC:       $RAB    FAB=RNUMFILE

PNUMTEMP:       .BLKL   1
PNUMOUT:        .BLKB   PNUMBUFSIZ      ;ВРЕМЕННАЯ ПАМЯТЬ И БУФЕРЫ
RNUMBUF:        .BLKB   RNUMBUFSIZ

;СИМВОЛЬНЫЕ СТРОКИ И ДЕСКРИПТОРЫ

PNUMSTR:        .ASCID  "ВЫВОД !XL(ШЕСТНАДЦАТЕРИЧНОЕ) !SL(ДЕСЯТИЧНОЕ)"
IOINITSTR:      .ASCID  "ВВОД И ВЫВОД ИНИЦИАЛИЗИРОВАНЫ !%D"
RNUMPRO:        .ASCII  "ВВЕДИТЕ ШЕСТНАДЦАТЕРИЧНОЕ ЧИСЛО - "
RNUMPROL:
RLINEPRO:       .ASCII  "ВВЕДИТЕ СТРОКУ - "
RLINEPROL:
PNUMDES:        .LONG   PNUMBUFSIZ
                .LONG   PNUMOUT


;ПРОГРАММА ИНИЦИАЛИЗАЦИИ

IOINIT::
        PUSHR   #^M<R0,R1>              ;СОХРАНИТЬ СОДЕРЖИМОЕ РЕГИСТРОВ
        $OPEN   FAB=RNUMFILE            ;ОТКРЫТЬ ФАЙЛ ВВОДА
        $CREATE FAB=PNUMFILE            ;ОТКРЫТЬ ФАЙЛ ВЫВОДА
        $FAO_S  CTRSTR=IOINISTR,OUTBUF=PNUMDES,-                ;ФОРМАТ
                OUTLEN=PNUMOREC+RAB$W_RSZ,P1=$0                 ;СООБЩЕНИЯ
        $CONNECT RAB=PNUMOREC           ;ПОДКЛЮЧИТЬ RAB СООБЩЕНИЙ
        $PUT    RAB=PNUMOREC            ;ПЕЧАТЬ СООБЩЕНИЯ ОБ ИНИЦИАЛИЗАЦИИ
        POPR    #^M<R0,R1>              ;ВОССТАНОВИТЬ РЕГИСТРЫ
        RSB                             ;ВОЗВРАТ


;ПРОГРАММА ПЕЧАТИ ЧИСЛА

PNUM::
        PUSHR   #^M<R0,R1>              ;СОХРАНИТЬ СОДЕРЖИМОЕ РЕГИСТРОВ
        MOVL    R0,PNUMTEMP             ;ПОЛУЧИТЬ И ПРЕОБРАЗОВАТЬ ЧИСЛО
        $FAO_S  CTRSTR=PNUMSTR,OUTBUF=PNUMDES,OUTLEN=PNUMOREC+RAB$W_RSZ,-
                P1=PNUMTEMP,P2=PNUMTEMP
        $CONNECT        RAB=PNUMOREC    ;ПОДКЛЮЧИТЬ RAB ВЫВОДА
        $PUT    RAB=PNUMOREC            ;И НАПЕЧАТАТЬ ЧИСЛО
        POPR    #^M<R0,R1>              ;ВОССТАНОВИТЬ РЕГИСТРЫ
        RSB                             ;ВОЗВРАТ

;ПРОГРАММА ВВОДА ЧИСЛА

RNUM::
        PUSHR   #^M<R1>                 ;СОХРАНИТЬ РЕГИСТР R1
        $CONNECT        RAB=RNUMIREC    ;ПОДКЛЮЧИТЬ RAB ВВОДА
        $GET    RAB=RNUMIREC            ;ВВЕСТИ ЧИСЛО
        CLRL    R0                      ;ОЧИСТИТЬ РЕГИСТР ПРЕОБРАЗОВАНИЯ
        MOVAB   RNUMBUF,R1              ;УКАЗАТЕЛЬ НА БУФЕР ЧИСЛА
10$:    DECW    RNUMIREC+RAB$W_RSZ      ;УМЕНЬШИТЬ СЧЕТЧИК СИМВОЛОВ
        BLSS    $30                     ;ЗАКОНЧИТЬ, ЕСЛИ В СЧЕТЧИКЕ 0
        ASHL    #4,R0,R0                ;УМНОЖИТЬ НА 16
        SUBB2   #48,(R1)                ;ПРЕОБРАЗОВАТЬ В ДВОИЧНОЕ ЧИСЛО
        CMPB    (R1),#10                ;БОЛЬШЕ ЧЕМ 10?
        BLSS    $20                     ;НЕТ
        SUBB2   #65-48-10,(R1)          ;ДА, ПРЕОБРАЗОВАТЬ ASCII "A" В 10
20$:    BISB    (R1)+,R0                ;ДОБАВИТЬ К ПОЛУЧЕННОМУ ЧИСЛУ
        BRB     $10                     ;ПОВТОРИТЬ ДЛЯ ОСТАВШИХСЯ ЦИФР
30$:    POPR    #^M<R1>                 ;ВОССТАНОВИТЬ РЕГИСТР R1
        RSB                             ;ВОЗВРАТ
```

```
;ПРОГРАММА ВВОДА СТРОКИ

RLINE::
        MOVL      R0,RLINEREC+RAB$L_UBF  ;УСТАНОВИТЬ АДРЕС БУФЕРА
        MOVW      R1,RLINEREC+RAB$W_USZ  ;И ЕГО ПРЕДЕЛЬНУЮ ДЛИНУ
        $CONNECT  RAB=RLINEREC           ;ПОДКЛЮЧИТЬ RAB
        $GET      RAB=RLINEREC           ;ВВЕСТИ СТРОКУ
        MOVZWL    RLINEREC+RAB$W_RSZ,R1  ;ВЕРНУТЬ ДЛИНУ ВВЕДЕННОЙ СТРОКИ
        RSB                              ;И КОД ОШИБКИ В R0

;ПРОГРАММА ПЕЧАТИ СТРОКИ

PLINE::
        PUSHR     #^M<R0,R1>             ;СОХРАНИТЬ СОДЕРЖИМОЕ РЕГИСТРОВ
        MOVL      R0,PLINEREC+RAB$L_RBF  ;ПОЛУЧИТЬ АДРЕС БУФЕРА
        MOVW      R1,PLINEREC+RAB$W_RSZ  ;И ДЛИНУ СТРОКИ
        $CONNECT  RAB=PLINEREC           ;ПОДКЛЮЧИТЬ RAB ВЫВОДА
        $PUT      RAB=PLINEREC           ;НАПЕЧАТАТЬ СТРОКУ
        POPR      #^M<R0,R1>             ;ВОССТАНОВИТЬ РЕГИСТРЫ
        RSB                              ;ВОЗВРАТ
```

ПРИЛОЖЕНИЕ В. ПРОГРАММЫ ВВОДА-ВЫВОДА НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

На вычислительных системах семейства VAX удобным способом для выполнения ввода-вывода является использование средств ввода и вывода языков высокого уровня. Приведенные здесь программы обеспечивают выполнение таких же функций ввода-вывода, как и программы, описанные в приложении Б. Однако в них применяются подпрограммы, процедуры и функции, написанные на Фортране, Бейсике и Паскале. Чтобы сделать последовательность действий, необходимых для вызова подпрограмм, одинаковой с последовательностями вызова подпрограмм, приведенных в приложении Б, сюда включается программа на языке ассемблера, которая служит для согласования связи программ на языке ассемблера с программами на языках высокого уровня. В гл. 9 описывается соглашение о связях между программами на языке ассемблера и программами на языках высокого уровня. Единственным замечанием является то, что все системы ввода-вывода, используемые в языках высокого уровня, сами выполняют свою инициализацию. Как следствие, программа IOINIT становится ненужной. Тем не менее для совместимости была включена фиктивная программа IOINIT, не выполняющая никаких действий, но возвращающая управление сразу после ее вызова.

ПРОГРАММА НА ЯЗЫКЕ АССЕМБЛЕРА

Приведенная ниже программа используется для вызова подпрограмм, написанных на перечисленных выше трех языках высокого уровня. Она может быть включена в основную программу пользователя как ее составная часть либо может быть отдельно ассемблирована и связана на этапе компоновки с основной программой и с подпрограммами, написанными на языках высокого уровня.

```
;
; ЭТА ПРОГРАММА РАБОТАЕТ АНАЛОГИЧНО  RNUM, PNUM И Т.Д.
; ВЫЗЫВАЕМЫЕ ПРОГРАММЫ НАПИСАНЫ НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ
;
        .EXTERNAL       FRNUM,FPNUM,FRSTR,FPSTR
;
IOINIT::
        RSB                             ;IOINIT НЕ ТРЕБУЕТСЯ
;
RNUM::
        PUSHR   #^M<R1>         ;СОХРАНИТЬ R1
        CALLS   #0,FRNUM        ;ВЫЗВАТЬ RNUM (ЯЗЫК ВЫСОКОГО УРОВНЯ)
```



```
        POPR    #^M<R1>
        RSB                     ;ВОССТАНОВИТЬ R1
                                ;РЕЗУЛЬТАТ В R0
;
PNUM::
        PUSHR   #^M<R1>         ;СОХРАНИТЬ R1
        PUSHL   R0              ;ПОМЕСТИТЬ АРГУМЕНТ В СТЕК
        PUSHAL  (SP)            ;ПОМЕСТИТЬ АДРЕС В СТЕК
        CALLS   #1,FPNUM        ;ВЫЗВАТЬ PNUM (ЯЗЫК ВЫСОКОГО УРОВНЯ)
        MOVL    (SP)+,R0        ;ВОССТАНОВИТЬ R0
        POPR    #^M<R1>         ;ВОССТАНОВИТЬ R1
        RSB
;
RLINE::
        PUSHL   R0              ;ПОМЕСТИТЬ АДРЕС В ДЕСКРИПТОР
        MOVZWL  R1,-(SP)        ;И ДЛИНУ В СТЕК
        PUSHAL  (SP)            ;ДЕСКРИПТОР ПЕРЕДАЕТСЯ КАК АРГУМЕНТ
        CALLS   #1,FRSTR        ;ВВЕСТИ СТРОКУ
        ADDL2   #8,SP           ;УДАЛИТЬ ИЗ СТЕКА ДЕСКРИПТОР
        MOVZWL  R0,R1           ;ПОМЕСТИТЬ ДЛИНУ В R1
        MOVL    #1,R0           ;И ПРИЗНАК УДАЧНОГО ЗАВЕРШЕНИЯ В R0
        RSB
;
PLINE::
        PUSHR   #^M<R0,R1>      ;СОХРАНИТЬ R0 И R1
        PUSHL   R0              ;ПОМЕСТИТЬ АДРЕС В ДЕСКРИПТОР
        MOVZWL  R1,-(SP)        ;И ДЛИНУ В СТЕК
        PUSHAL  (SP)            ;ДЕСКРИПТОР ПЕРЕДАЕТСЯ КАК АРГУМЕНТ
        CALLS   #1,FPSTR        ;НАПЕЧАТАТЬ СТРОКУ
        ADDL2   #8,SP           ;УДАЛИТЬ ИЗ СТЕКА ДЕСКРИПТОР
        POPR    #^M<R0,R1>      ;ВОССТАНОВИТЬ R0 И R1
        RSB
```

ПРОГРАММЫ НА ЯЗЫКЕ ФОРТРАН

Следующие подпрограммы и функции компилируются и компонуются с основной программой, написанной на языке ассемблера, а также с приведенными выше программами на языке ассемблера, используемыми для вызова данных подпрограмм и функций. По своему действию они в основном идентичны программам ввода-вывода из приложения Б, за исключением незначительных различий в формате ввода-вывода данных.

```
INTEGER FUNCTION FRNUM()
TYPE '('' $ВВЕДИТЕ ШЕСТНАДЦАТЕРИЧНОЕ ЧИСЛО - '')'
ACCEPT '(BN,Z8)',FRNUM
RETURN
END

SUBROUTINE FPNUM(N)
INTEGER N
TYPE '('' ВЫВОД ''Z9.8''(ШЕСТНАДЦ.)''I14''(ДЕСЯТИЧ.)'')',N,N
RETURN
END

INTEGER FUNCTION FRSTR(S)
INTEGER M,MINO,LEN
CHARACTER *(*) S
TYPE '('' $ВВЕДИТЕ СТРОКУ - '')'
ACCEPT '(Q,A)',M,S
FRSTR=MINO(M,LEN(S))
RETURN
END
```

```
SUBROUTINE FPSTR(S)
CHARACTER *(*) S
TYPE '(1X,A)',S
RETURN
END
```

ПРОГРАММЫ НА ЯЗЫКЕ БЕЙСИК

Ниже следуют программы, написанные на расширенном диалекте языка Бейсик, поддерживаемого в ОС VAX/VMS. Однако так как в языке Бейсик отсутствует шестнадцатеричный формат представления данных, то ввод и вывод чисел выполняется исключительно в десятичном формате. Для преобразования данных в шестнадцатеричное представление или для обратного преобразования могли бы потребоваться довольно большие программы преобразования, что свело бы на нет преимущество применения языка высокого уровня вместо языка ассемблера.

```
100     FUNCTION INTEGER FRNUM
110     INPUT "ВВЕДИТЕ ДЕСЯТИЧНОЕ ЧИСЛО - ";X%
120     FRNUM=X%
130     FUNCTIONEND


200     SUB FPNUM(N%)
210     PRINT "ВЫВОД ";N%;"(ДЕСЯТИЧНОЕ)"
220     SUBEND


300     FUNCTION INTEGER FRSTR(S$)
310     LINPUT "ВВЕДИТЕ СТРОКУ - ";S%
320     FRSTR=LEN(S$)
330     FUNCTIONEND

400     SUB FPSTR(S)
410     PRINT S$
420     SUBEND
```

ПРОГРАММЫ НА ЯЗЫКЕ ПАСКАЛЬ

Приведенные ниже программы написаны на расширенном диалекте языка Паскаль, поддерживаемого в ОС VAX/VMS. Так как в языке Паскаль отсутствует простой способ обработки символьных строк переменной длины, то в данный пример не были включены программы RLINE и PLINE.

```
MODULE PRPNUM (INPUT, OUTPUT);
FUNCTION FRNUM : INTEGER;
VAR N : INTEGER;
  BEGIN
  WRITE ('ВВЕДИТЕ ДЕСЯТИЧНОЕ ЧИСЛО - ')
  READLN (N);
  FRNUM := N;
  END;
PROCEDURE FPNUM (N : INTEGER);
  BEGIN
  WRITELN ('ВЫВОД - ',N,'(ДЕСЯТИЧНОЕ)');
  END;
END.
```


ПРИЛОЖЕНИЕ Г. РАБОТА С ФАЙЛАМИ В ОС VAX/VMS

Г.1. ФАЙЛЫ

Вся информация, располагаемая на устройствах внешней памяти вычислительных систем VAX, хранится в виде совокупностей данных, которые называются *файлами*. Файлы могут содержать информацию различного вида, объем информации в файле также может быть различен. Файлы могут быть очень маленькими или очень большими. В операционной системе VAX/VMS единицей измерения файла является *блок* из 512 символов (байтов). Каждый пользователь вычислительной системы имеет ограничение на число блоков, которые могут занимать его файлы. Информация в файлах бывает двух видов: символьные данные и несимвольные данные. Первые могут быть распечатаны и выглядят, как обычный печатный текст. Несимвольные данные предназначены для чтения только ЭВМ, а попытка их распечатать даст абсолютно бессмысленную распечатку и может вызвать "зависание" терминала вычислительной системы.

В каталоге пользователя файл может быть найден по своему наименованию, которое состоит из трех частей: имя файла (подходящее), тип файла и номер версии. Наименование файла может выглядеть следующим образом:

```
DATA.TXT;5
```

Здесь DATA – имя файла; TXT – тип файла; 5 – номер версии файла.

Отметим, что символы . и ; используются для разделения составных частей наименования файла. Именем является некоторое сочетание символов, которое вы (или тот, кто создал файл) присваивает файлу, чтобы его можно было однозначно идентифицировать. Имена файлов представляют собой строки от 1 до 39 символов, состоящие из букв или цифр. Примерами возможных имен файлов являются

```
X
132
R2D2
XYZ123AAA
```

Тип файла обычно состоит из трехбуквенного кода, обозначающего предположительное применение файла и используемого операционной системой[1]. В случаях, когда файл используется в предполагаемом контексте, указание типа файла не требуется. Ниже даны примеры типов файла.

Тип файла	Предполагаемое использование
.MAR	Программа на языке ассемблера MACRO
.OBJ	Объектный файл, получаемый при компиляции
.EXE	Файл выполняемого образа
.LIS	Файл листинга программы
.PAS	Программа на языке Паскаль
.FOR	Программа на языке Фортран

Номер версии файла назначается таким образом, что можно создать новую версию файла, не уничтожая при этом старой. Обычно пользователь явно не указывает номер версии файла. Отсутствие в наименовании файла номера версии обозначает, что указывается последняя версия файла, и если создается новая версия файла, то ей будет присвоен номер на единицу больше, чем номер созданной до этого последней версии файла.

---

[1] В старых версиях ОС VAX/VMS длина имени файла не могла быть более 9 символов, а длина типа файла – более 3 символов. Однако, начиная с версии 4.0, в ОС VAX/VMS имя файла и тип файла могут иметь длину до 39 символов.

Г.2. КАТАЛОГИ

Чтобы избежать неприятностей, которые могли бы возникнуть, если несколько пользователей случайно дали бы одно и то же имя двум разным файлам, наименования файлов заносятся в каталоги. Каждый пользователь имеет свой собственный каталог, назначенный ему администратором системы. Каталог связывается с именем регистрации пользователя в системе. Имена каталогов могут иметь длину до 39 символов. Имя каталога заключается в квадратные скобки, например:

```
[SMITH]
```

Пользователи могут (и это поощряется) создавать подкаталоги. Подкаталоги имеют свои имена, которые в полной спецификации указываются как второе (или третье, или четвертое и т. д.) имя, отделенное точкой. Например, в спецификации каталога

```
[SMITH.HOMEWORK.CIS72]
```

указан подкаталог CIS72, находящийся в подкаталоге HOMEWORK каталога SMITH. Для того чтобы дать полную спецификацию файла, наряду с наименованием файла необходимо указать имя каталога[1]. Примером такой полной спецификации файла является

```
[SMITH.HOMEWORK.CIS72]DATA.TXT;5
```

Чтобы не вводить полностью такую длинную строку символов, в ОС VAX/VMS разрешается определение каталога по умолчанию. В этом случае до тех пор, пока не будет задано конкретное имя другого каталога, предполагается использование имени каталога по умолчанию. Таким образом, если каталогом по умолчанию является [SMITH.HOMEWORK.CIS72], то спецификацию файла в предыдущем примере можно существенно сократить:

```
DATA.TXT
```

Номер версии файла опущен, так как предполагается, что версия 5 файла является самой последней.

Г.3. ВЫВОД СОДЕРЖИМОГО ФАЙЛА

Как уже говорилось, если файл является символьным, то его содержимое может быть выведено на терминал или на печатающее устройство. Для вывода содержимого файла на терминал используется команда операционной системы

```
$ TYPE имя_файла
```

Отметим, что знак доллара печатает операционная система, все остальное вводит пользователь. Ввод всех команд должен завершаться нажатием на клавишу RETURN. Например, если надо посмотреть, что содержится в файле TEXT.TXT, то следует набрать команду

```
$ TYPE TEXT.TXT
```

После ввода команды на экране терминала начнет появляться информация. Если не уметь читать очень быстро, то весьма затруднительно прочитать информацию с той скоростью, с которой выводит ее ЭВМ. Чтобы приостановить вывод информации на терминал, необходимо нажать комбинацию клавиш CTRL/S. Для этого надо нажать клавишу с маркировкой CONTROL или CTRL (обычно на клавиатуре она расположена в левой нижней части) и одновременно с ней нажать на клавишу с маркировкой S. Вывод на терминал будет остановлен до тех пор, пока не будет нажата комбинация клавиш CTRL/Q – сигнал, разрешающий возобновить вывод.

[1] На самом деле в полную спецификацию файла входят еще имя узла в сети ЭВМ и имя устройства внешней памяти, на котором находится каталог. Но обычно непривилегированному пользователю не нужно указывать эти атрибуты, так как вместе с именем каталога пользователя они определяются по умолчанию при регистрации пользователя в системе. – *Прим. ред.*

Если желательно получить распечатку содержимого файла на бумаге, то вместо команды TYPE используется команда PRINT. По этой команде содержимое файла будет распечатано на системном устройстве печати. Распечатке будет предшествовать титульный лист, где указываются имя пользователя и спецификация распечатываемого файла.

Г.4. ВЫВОД СОДЕРЖИМОГО КАТАЛОГА

Команда DIRECTORY (или сокращенное DIR) используется для просмотра имен и версий всех файлов, находящихся в каталоге пользователя. Так же как и для команды TYPE или любой команды либо программы, осуществляющих вывод на экран терминала, для остановки и возобновления вывода можно применять управляющие сигналы CTRL/S и CTRL/Q. Содержимое каталогов может выводиться в различном виде, для чего существует множество вариантов команды DIRECTORY. В списке ниже приведены некоторые из вариантов:

Команда	Результат действия
DIRECTORY/PRINT	Содержимое каталога выводится на устройство печати
DIRECTORY/SINCE=TODAY	Выводится список созданных за текущий день файлов
DIRECTORY/SINCE=5-JUL-1985	Выводится список файлов, созданных за период с 5-Июля-1985 по текущий день. (*Примечание*: год должен указываться четырьмя цифрами)
DIRECTORY/SINCE=10:30	Выводится список файлов, созданных за текущий день, начиная с 10:30 утра. (*Примечание*: на ЭВМ VAX применяется 24-часовой формат времени.)
DIRECTORY [SMITH.HOMEWORK.CIS72]	Выводится содержимое каталога [SMITH.HOMEWORK.CIS72]
DIRECTORY TEXT	Выводится список файлов, имеющих имя TEXT
DIRECTORY .TXT	Выводится список файлов, имеющих тип .TXT

Другие возможные варианты показаны в разд. Г.8. Заметим, что мнемоника большинства команд может быть сокращена до трех или четырех букв, например DIR/PRI. Если мнемоника сокращена больше, чем это допускается, то вычислительная система VAX сообщит, что не может однозначно определить, какая команда была введена.

Г.5. КОПИРОВАНИЕ ФАЙЛОВ

Команда COPY позволяет получить копию файла. С помощью этой команды можно было бы образовать в текущем каталоге файл DATA.TXT, являющийся копией файла TEXT.TXT, находящегося в каталоге [SMITH]. Команда COPY полезна не только для получения копии файла из другого каталога. Ее можно применить и просто для создания рабочей копии файла текущего каталога. По команде

```
$ COPY файл_1 файл_2
```

выполняется копирование содержимого файла FILE1 в файл FILE2. Например:

```
$ COPY [SMITH]TEXT.TXT DATA.TXT
```

Г.6. ИСПОЛЬЗОВАНИЕ ПОДКАТАЛОГОВ

В начале сеанса работы текущим каталогом пользователя является каталог по умолчанию, устанавливаемый для пользователя администратором системы. Если пользователь имеет много файлов, то в этом каталоге лучше хранить по возможности как можно меньше файлов. Это означает, что для каждой разработки пользователя следует создать свой подкаталог, а может быть, и подкаталоги в этих подкаталогах для отдельных этапов разработки. Предположим, что в начале сеанса работы каталогом по умолчанию является [SMITH]. В каталоге [SMITH] могли бы находиться следующие подкаталоги:

```
[SMITH.HOMEWORK]
[SMITH.PROJECT]
[SMITH.LISTINGS]
[SMITH.DATA]
```

Для того чтобы сделать один из этих подкаталогов текущим, надо ввести команду

```
$ SET DEFAULT [SMITH.HOMEWORK]
```

Чтобы убедиться в том, что выбранный подкаталог стал теперь текущим, полезно воспользоваться командой DIRECTORY.

Г.7. СОЗДАНИЕ ПОДКАТАЛОГОВ

Обычно хорошим правилом является создание подкаталогов для каждой из разработок пользователя. В этом случае файлы, созданные в ходе одной разработки, не будут смешиваться с файлами другой разработки. При необходимости всегда можно использовать команду COPY для копирования файлов из одного каталога в другой. Например, для создания подкаталога с именем [SMITH.HOMEWORK] применяется команда

```
$ CREATE/DIRECTORY [SMITH.HOMEWORK]
```

Г.8. ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НАД ГРУППАМИ ФАЙЛОВ

Копирование большого числа файлов могло бы потребовать значительных усилий для ввода информации. Предположим, что надо скопировать 100 файлов. В данном случае ввод всех команд копирования занял бы продолжительное время. Для выполнения операций над группами файлов (например, копирование всех файлов по одной команде) применяются универсальные символы-заменители. Способ использования символа-заменителя при групповом копировании файлов заключается в том, что он служит для представления любой строки символов. В качестве универсального символа-заменителя применяется символ звездочка *. Так, например, сочетание символов V* обозначает любое имя, начинающееся с буквы V. Сам по себе символ * обозначает любое имя вообще. Поэтому скопировать все последние версии всех файлов из одного подкаталога в другой можно следующей командой:

```
$ COPY [SMITH.DATA]*.* [SMITH.PROJECT]*.*
```

В наименовании файла универсальный символ-заменитель может присутствовать в полях имени, типа и номера версии.

Кроме команды COPY универсальный символ-заменитель можно использовать в командах TYPE, PRINT и DIRECTORY. В качестве примера рассмотрим следующие определения для групповых операций.

Обозначение	Значение
*.OBJ	Последние версии файлов с любым именем, имеющие тип .OBJ, т. е. SAMPLE.OBJ, PROG.OBJ и т. д.
PROG.*	Последние версии всех файлов с именем PROG, т. е. PROG.MAR, PROG.OBJ и т. д.
P*.MAR	Все программы на языке ассемблера, имя которых начинается с буквы P
SAMPLE.MAR;*	Все версии файла SAMPLE.MAR

Возможно, хотя это несколько сложнее, применять универсальные символы-заменители в спецификации каталогов. В спецификации каталогов могут присутствовать следующие символы-заменители:

Символ-заменитель	Значение
*	Любая строка символов
. в начале спецификации	Текущий каталог
– в начале спецификации	Каталог на следующем, более высоком уровне
...	Все последовательности имен каталогов

Рассмотрим действие этих символов-заменителей на конкретных примерах. Приведенные ниже команды DIRECTORY выполнялись в то время, когда текущим каталогом был [SMITH.HOMEWORK].

Команда	Содержимое выведенных списков
DIRECTORY [SMITH.*]	Все файлы из всех подкаталогов каталога [SMITH]
DIRECTORY [.CIS72]	Все файлы из каталога [SMITH.HOMEWORK.CIS72]
DIRECTORY [–.PROJECT]	Все файлы из каталога [SMITH.PROJECT]
DIRECTORY [SMITH...]	Все файлы из каталога [SMITH] и всех подкаталогов каталога [SMITH], а также всех подкаталогов подкаталогов и т. д.

Г.9. УДАЛЕНИЕ ФАЙЛОВ

В результате повторяющегося редактирования файлов создаются новые версии файлов. Избыточное число версий файлов может явиться помехой для дальнейшей работы. Кроме того, некоторые файлы устаревают и становятся ненужными. Если сохранять все версии файлов и никогда не удалять ненужные файлы, то очень скоро можно будет исчерпать выделяемое пользователю дисковое пространство. Для того чтобы привести в порядок содержимое каталогов и избавиться от ненужных файлов, имеются две команды PURGE и DELETE. Команда PURGE является простейшей из этих команд и служит для уничтожения всех версий файла, за исключением последней. Ниже приводится несколько примеров использования этой команды.

Команда	Результат действия
PURGE	Удаляются старые версии всех файлов текущего каталога
PURGE NAME.TXT	Удаляются старые версии файла NAME.TXT в текущем каталоге
PURGE *.OBJ	Удаляются старые версии всех файлов текущего каталога, имеющих тип .OBJ

Иногда можно сохранять несколько версий файла на случай, если будут испорчены последние версии файла. Это может быть выполнено с помощью команды

```
$ PURGE/KEEP=5
```

По этой команде в текущем каталоге будут удаляться все версии всех файлов, кроме пяти последних версий. Администратор системы может периодически давать команду PURGE/KEEP=5 или такую же команду, но с другим числом версий для всех файлов, находящихся в системе, чтобы привести в порядок все каталоги системы.

Вторая команда – DELETE – применяется для удаления определенных файлов по их наименованию. В команде в явном виде должны присутствовать имя, тип и номер версии файла, хотя при этом допускается использование универсальных символов-заменителей. Ниже даны примеры с командой DELETE.

Команда	Результат действия
DELETE NAME.TXT; 17	Удаляется версия 17 файла NAME.TXT. Эта команда полезна при удалении последней версии файла, если версия оказалась испорчена
DELETE *.OBJ; *	Удаляются все файлы, имеющие тип .OBJ
DELETE *.*; *	Удаляются все файлы текущего каталога – будьте осторожны!!!
DELETE/CONFIRM *.*;*	Удаляются все файлы текущего каталога, но система будет запрашивать подтверждение на удаление каждого файла
DELETE NAME.TXT; 0	Удаляется последняя версия файла NAME.TXT. *Примечание*: номер версии 0 обозначает последнюю версию файла
DELETE NAME.TXT; –3	Удаляется версия файла NAME.TXT, имеющая на 3 меньший номер, чем последняя версия этого файла

Теперь мы подошли к задаче удаления каталога. Предположим, что надо удалить каталог [SMITH.PROJECT.TEXT]. Как от него избавиться раз и навсегда? Для этого должна быть введена такая последовательность команд:

Команда	Результат действия
SET DEFAULT [SMITH.PROJECT.TEXT]	Удаляемый каталог становится текущим
DELETE *.*;*	Удаляются все файлы в каталоге
SET DEFAULT [–]	Текущим каталогом становится каталог предыдущего уровня [SMITH.PROJECT]
SET PROTECT= (OWNER:D) TEXT.DIR	Снимается защита, что позволяет удалить файл каталога
DELETE TEXT.DIR; *	Удаляется каталог

ПРИЛОЖЕНИЕ Д. РЕДАКТОР ТЕКСТА EDT И ЕГО ПРИМЕНЕНИЕ

Д.1. НАЗНАЧЕНИЕ РЕДАКТОРА ТЕКСТА

Как было показано в приложении Г, если только не используется система пакетной обработки, программы должны находиться в таких файлах, как PROG.MAR (PROG.FOR для программ на Фортране или PROG.PAS для программ на Паскале). Хотя возможно создавать программы, используя команду CREATE, при этом возникают некоторые трудности. Например, можно ввести команду

```
CREATE  PROG. MAR
```

затем – целиком текст программы, ввод которого завершается нажатием комбинации клавиш CTRL/Z. Но если были допущены какие-либо ошибки во введенной программе, то пришлось бы перепечатать всю программу заново, так как нет другого способа исправить ошибки с помощью команды CREATE. Редактор текста EDT позволяет вводить и модифицировать программу таким образом, что исправление ошибок или внесение добавлений в программу делается относительно просто.

Редактор EDT применяется для создания и модификации строк текста. Этот текст рассматривается как произвольная последовательность символов. Содержание текста не имеет никакого значения, будь то правильная программа на ассемблере, программы на Фортране

или Паскале, детские стишки или что-нибудь другое. В действительности многие применяют редактор текста EDT для работы с текстами на естественном языке, например для написания деловых писем. Расширяющееся применение ЭВМ для таких целей известно под общим названием word processing – подготовка текстов. Важно запомнить то, что редактор текста не знает, что записывается программа на ассемблере, поэтому символы языка ассемблера, такие как .TITLE, .END, ; , ; и т. д., для редактора текста не имеют специального значения.

На вычислительной системе VAX есть целый ряд редакторов текста. Наиболее популярными являются редакторы EDT, SOS и TECO. Редактор EDT, возможно, самый простой из них в работе, особенно если применяется такой сложный видеотерминал, как VT52, терминалы серий VT100/VT200 или совместимые с ними терминалы.

Д.2. РАБОТА С РЕДАКТОРОМ EDT В ЭКРАННОМ РЕЖИМЕ

При работе в экранном режиме редактор EDT отображает информацию, которая вводится на видеотерминале типа VT52, VT100/VT200 или совместимых с ними тип, на экране этого видеотерминала. В то время, как осуществляется ввод текста, выполняется удаление или какое-либо изменение в тексте, все происходящие изменения можно видеть на экране видеотерминала, что позволяет преобразовывать текст к его окончательному виду[1].

Для вызова редактора EDT надо ввести команду[2]

```
EDIT/EDT  PROGRAM. MAR
```

В этой команде PROGRAM.MAR – имя редактируемого файла программы. Если файл с именем PROGRAM.MAR уже существует, на экране появится первая строка файла, после которой строкой ниже будет следовать символ *. Если этот файл является новым файлом, создаваемым пользователем, то на экране появится сообщение "Input file does not exist" (файл ввода не существует), сопровождаемое на следующей строке символом [EOB], после которого строкой ниже следует символ *. Данное сообщение не является сообщением об ошибке; эта информация предупреждает пользователя о том, что он создает новый файл. Символ [EOB] будет появляться время от времени; он помечает конец редактируемого текста, а его мнемоника означает End Of Buffer (конец буфера).

В любом случае символ * показывает, что редактор EDT находится в режиме построчного редактирования, который описывается в разд. Д.4. Чтобы перейти в экранный режим, надо ввести команду CHANGE, завершив ввод команды нажатием на клавишу RETURN. По этой команде на экране видеотерминала будут отображены первые 22 строки программы пользователя (или не более 22 строк). На данном этапе в левом верхнем углу экрана будет находиться мерцающая отметка курсора, которая показывает, что в этом месте редактор EDT готов выполнить вставку или удаление текста.

Программа пользователя, если она не очень короткая (не более 24 строк), не может быть отображена на экране видеотерминала полностью. Поэтому на экране отображаются 24 строки, окружающие то место в тексте, где будет выполняться вставка или удаление. Вставки и удаления могут быть сделаны в любом месте текста. Место в тексте, где появляется изменение, отмечается мерцающей отметкой, называемой *курсором*. Курсор можно

---

[1] Для того чтобы редактор EDT мог сохранять на экране видеотерминала текущее состояние текста при редактировании, ЭВМ постоянно перепечатывает на экране строки или части строк. Это означает, что если терминал работает с низкой скоростью передачи данных, то производительность пользователя при редактировании значительно снижается. Минимальная рекомендуемая скорость передачи данных для видеотерминала составляет 1200 Бод (120 символов в секунду). Тем не менее в экранном режиме редактора EDT можно работать и при скорости передачи данных 300 Бод, если уменьшить количество строк текста, одновременно отображаемых на экране. Это выполняется с помощью команд SET LINE 8 и SET CURSOR 4:6, которые вводятся перед командой CHANGE.

[2] Ключ /EDT не требуется, если в вычислительной системе, в которой работает пользователь, по умолчанию установлен вызов редактора EDT.

```
MSG1:    .ASCID  "THIS ARE SOME MESSAGE"
 MSG2:    .ASCID  "NOW IS TIME"
 MSG3:    .ASCID  "FOR ALL MEN TO TO COME"
 MSG4:    .ASCID  "COME TWO THE AID OF HIS"
 MSG5:    .ASCID  "PARTY"
 ;ВЫШЕ ПРИВЕДЕНЫ НЕКОТОРЫЕ ПРОИЗВОЛЬНЫЕ СООБЩЕНИЯ
          .ENTRY  START
          MOVAB   MSG1,R0
          JSB     PMESG
          MOVAB   MSG2,R0
          JSB     PMESG
          MOVAB   MSG3,R0
          JSB     PMESG
          MOVAB   MSG5,R0
          JSB     PMESG
          $EXIT_S
;НИЖЕ РАСПОЛОЖЕНА ПОДПРОГРАММА ПЕЧАТИ СООБЩЕНИЙ
PMESG:    TSTB    OPENFLAG          ←——— мерцающая отметка курсора
          BNEQ    10$
          JSB     OPENFILE
          DECB    OPENFLAG
10$:      MOVW    2(R0),FILEFAB+FAB$W_RSZ
```

Рис. Д.1. Фрагмент текста программы, отображенный на экране видеотерминала

всюду перемещать с помощью клавиш, на которых нарисованы стрелки, обозначающие направление перемещения; кроме того, курсор можно также перемещать и другими способами. Текст в окне обычно устанавливается так, что курсор находится во второй трети экрана терминала пользователя по вертикали. Конечно, если курсор находится на какой-либо из первых или последних строк программы пользователя, то на экране видеотерминала он может располагаться в его верхней или нижней части. На рис. Д.1 показано, как могло бы выглядеть изображение на экране видеотерминала при редактировании текста. В показанной на рисунке программе содержатся некоторые умышленно допущенные ошибки, которые должны быть исправлены.

ВСТАВКА И УДАЛЕНИЕ ТЕКСТА

Для того чтобы исправить ошибки во фрагменте программы, показанном на рис. Д.1, в первую очередь необходимо прочитать ее текст и обнаружить ошибки. Первая ошибка находится во второй строке:

```
MSG2:    .ASCID      "NOW IS TIME"
```

Должно быть

```
MSG2:    .ASCID      "NOW IS THE TIME"
```

Чтобы сделать исправление, надо вставить слово THE после слова IS. Для этого сначала надо переместить курсор в нужное место, что может быть выполнено с помощью клавиш управления курсором. Имеются четыре клавиши с нанесенными на них стрелками, стрелки указывают направление перемещения курсора – влево, вправо, вверх, вниз. Так как курсор расположен в нижней части экрана, то необходимо 15 раз нажать на клавишу со стрелкой, направленной вверх, чтобы переместить курсор на вторую строку (см. рис. Д.1). Отметим, что во время перемещения курсора вверх текст программы будет перемещаться на экране так, чтобы по возможности положение курсора сохранялось как можно ближе к нижней части второй трети экрана. Поэтому вверху экрана будут появляться новые строки текста программы, а строки внизу экрана будут исчезать. Это называется перемещением или "прокруткой" (scrolling) вверх по направлению к началу текста. Если на экране ви-

деотерминала появляется первая строка текста, то "прокрутка" прекращается, и курсор будет перемещаться в верхнюю часть экрана выше той части, где сохранялось его положение при "прокрутке". Отметим также, что некоторые строки текста по своей длине не достигают горизонтальной позиции курсора в строке, на которой он находился первоначально. В результате этого при перемещении курсор будет изменять свою горизонтальную позицию в строке, стараясь сохранить первоначальную горизонтальную позицию, но не выходя за пределы какой-либо строки текста.

После того как 15 раз была нажата клавиша со стрелкой, направленной вверх, курсор будет находиться в правой части второй строки текста, но не в нужном месте. Отметка курсора будет мерцать над буквой E в слове TIME, что выглядит следующим образом:

```
MSG2:    .ASCID  "NOW IS TIME"
                          ↑——— курсор
```

После трехкратного нажатия на клавишу со стрелкой, направленной влево, отметка курсора появится над буквой T в слове TIME:

```
MSG2:    .ASCID  "NOW IS TIME"
                          ↑——— курсор
```

Теперь все готово для вставки символов. Вводимые с клавиатуры терминала символы будут вставляться в текст перед тем символом, на котором находится курсор. На рис. Д.2 показано, как будет выглядеть строка на экране терминала каждый раз после ввода очередного символа, если вводятся символы T, H, E и пробел. Обратите внимание на то, как перемещается курсор при вводе каждого символа, что отображено на рис. Д.2. Отметьте также, что пробелы и табуляция занимают очень большую часть текста и должны вставляться на соответствующие места.

Следующая ошибка находится в строке

```
MSG3:    .ASCID      "FOR ALL MEN TO TO COME"
```

Во-первых, в данной строке пропущено слово GOOD; должно быть . . . ALL GOOD MEN. . . Курсор необходимо снова переместить, на этот раз с помощью клавиши со стрелкой, направленной вниз, и клавиши со стрелкой, направленной влево. После чего может быть вставлено слово GOOD таким же способом, как это делалось в предыдущем примере со словом THE.

Однако в строке осталась еще одна ошибка – слово TO появляется в ней дважды. Требуется удалить одно из этих слов. Для удаления текста сначала необходимо переместить курсор так, чтобы он оказался справа от удаляемого символа, а затем удалить расположенные от курсора слева символы, нажав на клавишу удаления. Удаление выполняется так же, как при работе без редактора текста, и с помощью той же клавиши. Основное отличие состоит в том, что при работе с редактором EDT символы можно удалять в любом месте

```
                  MSG2:    .ASCID    "NOW IS TIME"
                                                ↑——— курсор
Введено  T        MSG2:    .ASCID    "NOW IS TTIME"
                                                ↑——— курсор
Введено  H        MSG2:    .ASCID    "NOW IS THTIME"
                                                ↑——— курсор
Введено  E        MSG2:    .ASCID    "NOW IS THETIME"
                                                ↑——— курсор
Введен "пробел"   MSG2:    .ASCID    "NOW IS THE TIME"
                                                ↑——— курсор
```

Рис. Д.2. Вставка символов в текст

```
                MSG3:   .ASCID   "FOR ALL GOOD MEN TO TO COME"
                                                  ↑ курсор
Удаление        MSG3:   .ASCID   "FOR ALL GOOD MEN T TO COME"
                                                  ↑ курсор
Удаление        MSG3:   .ASCID   "FOR ALL GOOD MEN  TO COME"
                                                  ↑ курсор
Удаление        MSG3:   .ASCID   "FOR ALL GOOD MEN TO COME"
                                                  ↑ курсор
```

Рис. Д.3. Удаление текста

строки, устанавливая курсор на нужную позицию. В данном случае для перемещения курсора на следующую позицию после буквы О в первом слове ТО может использоваться клавиша со стрелкой, направленной вправо. Можно удалять любое из двух слов ТО; это не имеет никакого значения, за исключением того, что первое слово ТО расположено ближе и требуется меньше нажатий на клавишу управления курсором, чтобы переместить его на нужную позицию. На рис. Д.3 показано, как могла бы выглядеть строка на экране терминала после каждого удаления. Отметим, что клавиша удаления нажимается три раза. Дополнительное третье нажатие клавиши нужно для удаления лишнего пробела, отделяющего слово ТО.

Полученных сведений достаточно, чтобы читатель смог легко устранить оставшиеся ошибки, так как все редактирование сводится к нахождению нужного места в тексте и вставке или удалению символов. Тем не менее если бы все редактирование с помощью редактора EDT ограничивалось использованием пяти описанных выше управляющих клавиш (четыре клавиши управления курсором и клавиша удаления), то, чтобы отредактировать любой текст большого объема, очень скоро можно было бы сбить пальцы до костей, нажимая эти клавиши. С помощью названных клавиш редактирование выполняется над каждым символом в отдельности. Но имеются еще и другие управляющие клавиши, с помощью которых можно выполнять функции редактирования над группами символов, например такие, как удаление слова, удаление строки или удаление параграфа целиком. В следующем разделе будут описываться некоторые команды редактирования, позволяющие выполнять подобные функции; другие команды будут оставлены читателю для самостоятельного изучения.

Д.3. ВВОД КОМАНД С ПОМОЩЬЮ ВСПОМОГАТЕЛЬНОЙ КЛАВИАТУРЫ

Использование вспомогательной клавиатуры является еще одним средством редактирования программы. Команды редактора EDT можно вводить с помощью вспомогательной числовой клавиатуры, расположенной в правой части основной клавиатуры. Числовые клавиатуры видеотерминалов работают в одном из двух режимов. Первый – режим обычной работы, в котором клавиши числовой клавиатуры просто дублируют возможность ввода чисел с основной клавиатуры. В тех случаях, когда терминал главным образом применяется для ввода числовых данных, использование клавиш числовой клавиатуры может повысить скорость ввода данных, так как расположение клавиш в основном такое же, как на многих калькуляторах. Второй режим работы является режимом специальных символов, где при нажатии клавиши генерируется свой собственный символьный код, который не похож на код, генерируемый при нажатии клавиш основной клавиатуры[1]. Во время работы с экранным редактором EDT вспомогательная клавиатура будет находиться в режиме ввода специальных символов.

[1] В режиме специальных символов при нажатии клавиш вспомогательной клавиатуры передается последовательность символов, начинающаяся с управляющего символа ESC кода ASCII. Например, нажатие клавиши с цифрой 1 на терминале типа VT52 вызовет передачу последовательности символов ⟨ESC⟩?q, а нажатие такой же клавиши на терминале серии VT100 или VT200 – последовательность ⟨ESC⟩0q.

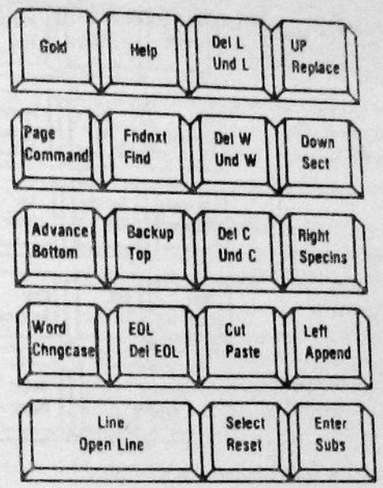


Рис. Д.4. Функциональное назначение клавиш вспомогательной клавиатуры терминала VT52

В связи с тем, что функций редактирования больше, чем клавиш на вспомогательной клавиатуре, большинство клавиш имеет две функции: основную и дополнительную. Для выполнения основной функции надо просто нажать на клавишу. Например, основной функцией клавиши с цифрой 2 является перемещение курсора в конец строки. Если пользоваться этой клавишей, то можно избежать многократного нажатия клавиши со стрелкой, направленной вправо. Дополнительная функция клавиш с цифрой 2 – удаление всех символов, начиная от позиции курсора до конца строки текста. (Действие этой функции противоположно действию функции CTRL/U, применяемой в обычном режиме работы для удаления всех символов, находящихся слева от курсора.) Для выполнения дополнительной функции клавиши с цифрой 2 надо предварительно нажать и отпустить клавишу, обозначенную на рисунках с надписью GOLD. На терминалах серий VT100/VT200 клавиша GOLD имеет маркировку PF1, а на терминале типа VT52 она окрашена в голубой цвет. В любом случае клавиша GOLD располагается в левом верхнем углу числовой клавиатуры. На рис. Д.4 и рис. Д.5 показано расположение клавиш вспомогательной клавиатуры терминалов типа VT52 и серий VT100/VT200. Возможно даже заказать клавиши с маркировкой названий функций редактора EDT. Но если не посчастливилось иметь клавиши с такой маркировкой, то можно нажать специальную клавишу с функцией HELP (помощь), которая находится справа от клавиши GOLD. На терминале типа VT52 она окрашена в красный цвет, а на терминалах серий VT100/VT200 клавиша имеет маркировку PF2. При нажатии на эту клавишу текст программы исчезнет с экрана. (Не стоит беспокоиться – текст сохраняется в памяти ЭВМ.) После этого на экране появляется точная копия рис. Д.4 или рис. Д.5, что помогает найти соответствующую функцию.

В данном разделе не ставится цель дать полное описание функций редактирования, вызываемых с помощью вспомогательной клавиатуры. Лучший способ изучить редактор текста – это попробовать выполнить различные функции и посмотреть, что при этом произойдет. На вычислительных системах VAX файлы достаточно хорошо резервируются, так что эксперименты можно проводить без особого риска. Поэтому каждая функция вспомогательной клавиатуры будет описываться несколькими предложениями. Более подробную информацию можно получить, воспользовавшись командой HELP. В связи с тем, что вспомогательная клавиатура терминалов типа VT52 отличается от вспомогательной клавиату-

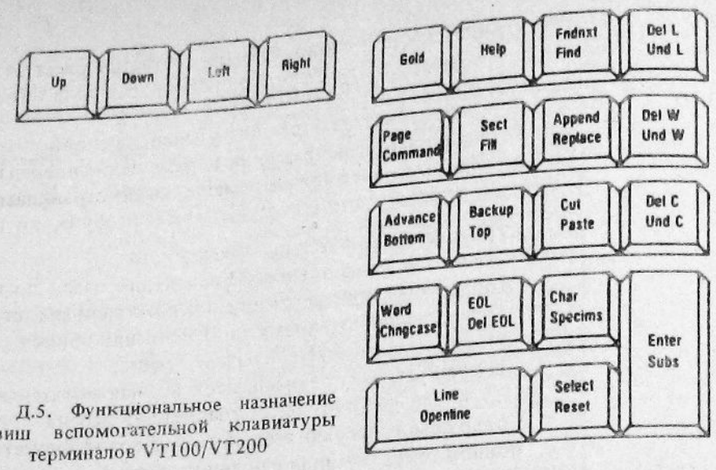


Рис. Д.5. Функциональное назначение клавиш вспомогательной клавиатуры терминалов VT100/VT200

ры терминалов серий VT100/VT200, функции будут описываться без указания стандартной маркировки клавиш. На самом деле, так как на вспомогательной клавиатуре терминала типа VT52 меньше клавиш, некоторые клавиши, соответствующие функциям на этой клавиатуре отсутствуют. Клавиши, соответствующие описываемым функциям, можно найти по рис. Д.4 или Д.5. На рисунках верхняя надпись на клавишах обозначает основную функцию, нижняя – дополнительную функцию, для вызова которой требуется применение клавиши GOLD.

Функция	Результат действия
GOLD	Устанавливается обращение к дополнительной функции
HELP	На экран терминала выводится схема вспомогательной клавиатуры, а также другие сообщения, помогающие при работе с редактором EDT
FIND NEXT	Выполняется в выбранном направлении поиск по тексту следующего появления строки, соответствующей образцу
FIND	Осуществляется ввод строки образца и выполняется поиск по тексту
DELETE LINE	Удаляется строка справа от курсора
UNDELETE LINE	Восстанавливается последняя удаленная строка. Это может использоваться в случае непредумышленного удаления строки. Кроме того, может применяться для копирования строки – удаляя, а затем несколько раз восстанавливая строку
PAGE	Курсор перемещается на одну страницу в выбранном направлении, т. е. до следующего символа перевода страницы
COMMAND	Позволяет вводить обычные символьные команды редактора текста
SECTION	Курсор перемещается на 16 строк в выбранном направлении
FILL	При подготовке текста выравнивается длина строк. Обычно не используется при программировании
APPEND	Помеченный текст добавляется к тексту, находящемуся во вспомогательном буфере
REPLACE	Выбранная часть текста замещается содержимым вспомогательного буфера

DELETE WORD	Удаляется одно слово. Словом считается текст от позиции курсора до следующего пробела, символа табуляции или конца строки
UNDELETE WORD	Восстанавливается последнее удаленное слово
ADVANCE	Устанавливается направление перемещения вперед по тексту. Используется при перемещении по страницам, строкам, словам, символам, а также для функций поиска
BOTTOM	Курсор перемещается на конец текста
BACKUP	Выбирается направление перемещения назад по тексту. Используется при перемещении по страницам, строкам, словам, символам, а также для функций поиска
TOP	Курсор перемещается на начало текста,
CUT	Отмеченный текст помещается во вспомогательный буфер, после чего этот текст удаляется с экрана
PASTE	Содержимое вспомогательного буфера копируется в основной текст, начиная с позиции курсора
DELETE CHARACTER	Удаляется символ, на котором находится отметка курсора. Действие отличается от действия клавиши удаления, когда удаляется символ слева от курсора
UNDELETE CHARACTER	Восстанавливается последний удаленный символ
WORD	Курсор перемещается на одно слово в выбранном направлении до следующего пробела, символа табуляции или конца строки
CHANGE CASE	В тексте происходит замена прописных букв строчными и наоборот
END OF LINE	Курсор перемещается в конец текущей или предыдущей строки в зависимости от выбранного режима (ADVANCE или BACKUP)
DELETE TO END OF LINE	Удаляются все символы справа, начиная от отметки курсора и до конца строки. При этом символы "Возврат каретки" и "Перевод строки", находящиеся в конце строки, не удаляются, что является единственным различием между функциями DELETE TO END OF LINE и DELETE LINE
MOVE CHARACTER	Курсор перемещается на одну позицию в заданном направлении
SPECIAL CHARACTER INSERT	Позволяет вставлять в текст управляющие символы
ENTER	Завершение ввода символьной команды
SUBSTITUTE	Найденная по образцу символьная строка заменяется содержимым вспомогательного буфера, после чего автоматически выполняется функция FIND NEXT (применяемая после функции FIND)
BEGINNING OF LINE	Курсор перемещается на начало следующей или предыдущей строки, в зависимости от выбранного направления
OPEN LINE	В текст вставляется пустая строка, а курсор помещается в ее начало. Команда облегчает работу, если надо вставить новую строку в середину текста
SELECT	Помечается начало или конец выбираемого фрагмента текста, на котором находится отметка курсора. После этого курсор надо переместить на другой конец фрагмента. Выбранный фрагмент текста может быть помещен во вспомогательный буфер. На терминалах серий

VT100/VT200 выбранный текст выделяется инверсным изображением

RESET — Отменяется выбор помеченного фрагмента текста

Двумя наиболее важными функциями являются FIND и FIND NEXT. В больших программах нахождение ошибок может стать достаточно утомительным занятием, если при поиске ошибки перемещать курсор с помощью клавиш управления курсором. Лучше осуществить с помощью редактора текста поиск некоторой строки символов, определяющей место в тексте, которое необходимо отредактировать. Например, предположим, что ассемблер выдал сообщение об использовании в программе неопределенного кода операции MOVX. Для исправления ошибки можно вызвать редактор текста и воспользоваться функцией поиска FIND. На терминалах серий VT100/VT200 функция FIND выполняется при последовательном нажатии клавиш с маркировкой PF1 и PF3; на терминале типа VT52 следует нажать клавишу голубого цвета и клавишу с цифрой 8. После нажатия клавиш внизу экрана терминала появится запрос Search for: (искать). В ответ на запрос с клавиатуры терминала вводится строка, которую желательно найти; в данном случае вводится MOVX. Затем в зависимости от того, в каком направлении желательно осуществить поиск – к концу файла или к его началу, надо нажать на одну из двух клавиш направления перемещения – BOTTOM или BACKUP (клавиши с цифрой 4 или 5 на терминале любого типа). Редактор текста будет выполнять поиск до тех пор, пока не найдет MOVX. После этого можно с помощью клавиш управления курсором установить курсор над буквой X и удалить ее, а затем напечатать на ее месте буквы L, W или B или что-нибудь подобное. Если такая ошибка появлялась несколько раз, то поиск местонахождения последующих кодов MOVX можно выполнить, используя функцию FIND NEXT (клавиша PF3 на терминалах VT100/VT200 или клавиша с цифрой 8 на терминале VT52).

Еще одну важную операцию представляет собой ввод обычных символьных команд редактирования. Хотя имеется целый ряд таких команд, только одна из них – команда EXIT является критической. Она сообщает редактору текста EDT, что сеанс редактирования закончен и новый файл должен быть сохранен. Никакие из проведенных изменений не будут запомнены до тех пор, пока не выполнится эта команда. Для перехода в командный режим надо нажать последовательно клавишу GOLD и клавишу с цифрой 7 на терминале любого типа. Внизу экрана появится запрос Command (команда). После того как пользователем будет напечатана команда EXIT и нажата клавиша ENTER, редактор EDT завершит работу и произойдет возврат к диалогу с ОС VAX/VMS. Если теперь ввести команду операционной системы DIRECTORY, можно увидеть, что образован новый файл с номером версии на единицу больше, чем последняя версия этого файла. Например, если файл редактировался 5 раз, то в каталоге находятся файлы

```
.
.
.
PROGRAM.MAR;3
PROGRAM.MAR;4
PROGRAM.MAR;5
```

Чтобы избавиться от старых версий файла, можно воспользоваться командой PURGE, описанной в приложении Г. Отредактированная информация будет содержаться в файле PROGRAM.MAR;5. Если при редактировании случайно была испорчена информация, то она может быть восстановлена, так как файл PROGRAM.MAR;4 содержит информацию в том виде, в котором она находилась до редактирования.

Д.4. РЕДАКТИРОВАНИЕ С ТЕЛЕТАЙПА

Если вам не посчастливилось приобрести терминалы типа VT52 или VT100/VT200 либо совместимые с ними терминалы, то редактирование усложняется. Предположим, что редактирование осуществляется на терминале типа телетайп или, иначе, на печатающем терминале. На телетайпе нельзя сразу увидеть происходящие изменения текста, так как что-либо однажды напечатанное печатается на бумаге и не может изменяться. Вследствие этого редактирование с помощью редактора EDT должно выполняться в построчном режиме. Хотя имеется значительное число команд редактора EDT, облегчающих построчное редактирование, тем не менее возможно отредактировать текст, используя только несколько основных команд, описанных в данном разделе.

Для того чтобы начать редактирование файла, вводится следующая команда ОС VAX/VMS[1]:

```
EDIT/EDT  PROGRAM.MAR
```

где PROGRAM.MAR – имя файла, где содержится программа. Если файл с таким именем редактируется впервые, то появится сообщение Input file does not exist (файл ввода не существует), после которого следует символ [EOB], а строкой ниже – символ *. Это сообщение похоже на сообщение об ошибке, а на самом деле – просто предупреждение о том, что создается новый файл. Строка

```
[EOB]
```

означает, что в файле содержится только признак конца буфера – End Of Buffer.

Если файл используется не в первый раз, то сообщение не появится, так как файл ввода уже *будет* существовать. Первая строка программы будет отображена на терминале как строка с номером 1. Например:

```
1       .TITLE   MY PROGRAM
*
```

Снова отметим наличие символа *. Он является приглашением, сообщающим, что редактор EDT ожидает ввод команд редактирования.

Ниже приводится основной набор команд редактора EDT, с помощью которого можно начать редактирование. Для более эффективного использования редактора текста могут быть также изучены другие команды и разновидности приведенных команд.

INSERT (вставить). Команда INSERT может использоваться при создании нового файла или при добавлении новых строк к программе. После приглашения к вводу (символа *) надо напечатать команду INSERT[2], ввод которой завершается нажатием на клавишу RETURN. В ответ на команду редактор EDT установит новую пустую строку, сделав отступ от начала этой строки. Такое действие редактора является приглашением к началу ввода текста. Все, что будет затем введено, вставится в текст в том виде, в котором вводилось. Отметим, что каждая строка будет начинаться с отступа. Отступы не являются частью текста – это просто подсказка, что ЭВМ продолжает работать. Когда все, что было желательно, вставлено в текст, надо нажать комбинацию клавиш CTRL/Z. (Одновременно нажимаются клавиша CTRL и клавиша с буквой Z.) Редактор EDT напечатает символ *, который свидетельствует о том, что редактор снова находится в командном режиме. В общем случае приглашением к вводу для командного режима является символ *, а для режима вставки – отступ от начала строки.

TYPE (напечатать). Обычно хочется увидеть, как выглядит содержимое файла после вставки текста или каких-либо других изменений. Это можно сделать с помощью команды TYPE. В команде TYPE должен быть указан *код интервала*, определяющий, какие строки

---

[1] Ключ /EDT не требуется, если в операционной системе по умолчанию вызывается редактор EDT.

[2] Все команды и ключевые параметры, описанные в данном разделе (кроме команд EXIT и RESEQUENCE), могут быть сокращены до одного символа. Таким образом, команда INSERT может вводиться как I. Аналогично команды TYPE, FIND, DELETE и SUBSTITUTE, а также ключевые параметры BEGIN и END могут быть сокращены как T, F, D, S, B и E.

надо распечатать. Если интервал не указан, то напечатается *текущая* строка. Текущей является строка, готовая для выполнения редактирования. Всякий раз при вставке текста текст будет располагаться перед текущей строкой.

Если надо напечатать всю программу, то в команде может быть указан интервал BEGIN:END (начало:конец):

```
TYPE BEGIN:END
```

Отметим, что при печати строк слева добавляется номер строки. Он не является частью текста и используется только в целях редактирования. Например, если строки вставлялись в пустой файл, как это делается при создании файла, то по команде TYPE BEGIN:END выводится следующее:

```
1       .TITLE   PROGRAM
2       .ASCII   "ЗДЕСЬ ПРОСТО"
3       .ASCII   "НЕСКОЛЬКО СТРОЧЕК"
[EOB]
*
```

Интервалы можно задать также с помощью номеров строк. Например, по команде TYPE 1:2 напечатаются строки 1 и 2, а указание единственного номера строки вызовет печать одной строки – TYPE 2.

Еще одно замечание касается того, что после выполнения команды TYPE текущей строкой становится первая из напечатанных строк. Таким образом, по команде TYPE 2:3 строка 2 будет установлена текущей. Это означает, что вставляемый по команде INSERT текст будет теперь помещен между строками 1 и 2. Если вставки делаются между такими двумя последовательно перенумерованными строками, то редактор EDT присвоит строкам дробные десятичные номера, т. е. 1.1, 1.2, 1.3 и т. д. Дробные номера строк могут также использоваться для указания интервала, как и целые номера. Например, по команде

```
TYPE 1.2:1.3
```

можно напечатать все строки, имеющие номера от 1.2 до 1.3. После этого строка с номером 1.2 может стать текущей. Поэтому вставляемым между строками с номерами 1.1 и 1.2 новым строкам могут быть присвоены номера 1.11, 1.12, 1.13 и т. д.

Символ *"точка"* добавляет еще одно полезное свойство – является ссылкой на текущую строку и может использоваться в выражениях. Так, выражение .–3 обозначает третью строку перед текущей, а выражение .+3 обозначает третью строку после текущей. По команде TYPE .–3,.+3 будут напечатаны третья строка перед и третья после текущей строки.

FIND (найти). Для того чтобы воспользоваться командой INSERT, строка, перед которой нужно вставить текст, должна быть сделана текущей строкой. Иначе говоря, если текст требуется вставить между строками 1.11 и 1.12, то текущей строкой должна быть строка с номером 1.12. Это может быть выполнено по команде FIND.

В команде FIND указывается обозначение строки, идентифицирующее искомую строку. Допустимо указывать любое из обозначений строк, описанных в данном разделе для команды TYPE. Например, приведенные ниже команды позволяют найти текущую строку по ее описанию.

Команда	Текущая строка
FIND 1.12	Строка с номером 1.12
FIND BEGIN	Первая строка в файле
FIND END	Строка, содержащая символ [EOB]
FIND .+4	Четвертая строка после текущей, установленной на момент ввода команды

Кроме того, в качестве указателя строки может использоваться заключенная в кавычки символьная строка-образец. В этом случае редактор EDT будет искать первую строку, содержащую указанную строку-образец. Например, по команде FIND ".BLKL" редактор EDT будет осуществлять поиск по направлению к концу файла (вперед) первой строки после

текущей, содержашей символьную строку .BLKL. Иногда полезно выполнять поиск в обратном направлении, т. е. искать строку, расположенную ближе к началу файла, чем текущая. При поиске в обратном направлении заключенной в кавычки строке должен предшествовать знак "минус", как, например, в команде FIND – ".BLKL".

При поиске важно подобрать достаточно длинную строку-образец, однозначно определяющую искомую строку, иначе можно найти совсем не то, что хотелось бы. В действительности строка ".BLKL" как образец не очень подходит, так как в программе может оказаться много строк, содержащих символьную строку ".BLKL". Было бы лучше задать следующий образец:

```
FIND "BLKL (tab) 23"
```

Здесь tab обозначает символ табуляции, который может присутствовать в заключенной в кавычки строке так же, как и любой другой символ, используемый в программе.

После выполнения команды FIND обычно рекомендуется напечатать текущую строку. Это может быть сделано по команде TYPE без указаний в ней интервала распечатываемых строк. Последовательность команд могла бы выглядеть так[1]:

```
FIND "BLKL (tab) 23"
TYPE
```

В результате выполнения этих команд будет напечатана найденная строка.

DELETE (удалить). Строки программы, в которых содержатся ошибки, нужно удалить из программы, что может быть сделано с помощью команды DELETE. В команде DELETE указывается интервал номеров строк так же, как и в команде TYPE. Все строки, попадающие в заданный интервал, будут удалены. Если по неосторожности будут удалены лишние строки, то их придется вводить снова. Поэтому перед удалением строк обычно рекомендуется напечатать строки, попадающие в заданный интервал. Например, можно использовать команды

```
TYPE 1.2:1.22
DELETE 1.2:1.22
```

Если в команде DELETE указать интервал строк точно так же, как и в команде TYPE, то будет точно известно, какие строки удалялись. К тому же возможно, чтобы редактор EDT запрашивал подтверждение при удалении каждой строки с помощью команды DELETE/VERIFY.

SUBSTITUTE (заменить). Если в строке имеется ошибка, то ее можно устранить обычным способом – удалить эту строку, а затем вставить исправленную. Однако, если есть длинная строка только с одной небольшой ошибкой, существует лучший способ исправления. Предположим, что строка, содержащая ошибку, имеет вид

```
MESG:   .ASCII  " ЭТО БОЛЬШОЕ СООБЩЕНИЕ"
```

В этой строке вместо директивы .ASCII должна быть директива .ASCID. Команда SUBSTITUTE сообщает редактору EDT точно то же самое. После применения для нахождения строки команды FIND вводится команда

```
SUBSTITUTE  /.ASCII/.ASCID/
```

Отметим, что строка с ошибкой указывается первой, а правильная строка – второй. Как и в случае команды FIND важно, чтобы строка-образец была выбрана достаточно длинной для ее однозначного определения. Команда SUBSTITUTE /II/ID/ привела бы к желаемому

[1] Число команд можно сократить, поместив заключенную в кавычки строку непосредственно в команду TYPE, как TYPE ".BLKL(tab) 23". В таком случае команда FIND будет не нужна.

результату, но по команде SUBSTITUTE /I/D/ могла произойти ошибочная замена не того символа I, который нужен, что дало бы строку .ASCDI.

RESEQUENCE (перенумеровать). При выполнении вставок появляется большое число строк с дробными номерами. Это может вносить путаницу при нахождении строк. По команде RECEQUENCE все строки будут перенумерованы и им будут присвоены целые номера, начиная с номера 1. Рекомендуется после перенумерации распечатать всю программу целиком, чтобы можно было найти все строки. Перенумерация выполняется автоматически в начале редактирования файла, если только не приняты специальные меры для сохранения номеров строк.

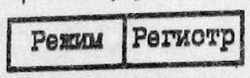
EXIT (выйти). После окончания редактирования должна быть введена команда EXIT. Если работа с редактором текста завершается другим способом, например нажатием комбинации клавиш CTRL/Y, то можно потерять все, что было отредактировано.

Хотя редактор EDT кажется сложным, работать с ним легко.

ПРИЛОЖЕНИЕ Е. ИНСТРУКЦИИ И ТИПЫ ДАННЫХ ЭВМ СЕМЕЙСТВА VAX

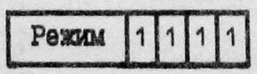
РЕЖИМЫ АДРЕСАЦИИ

АДРЕСАЦИЯ С ИСПОЛЬЗОВАНИЕМ РЕГИСТРОВ ОБЩЕГО НАЗНАЧЕНИЯ



Шест.	Дес.	Название режима	Обозначение	Тип доступа r	m	w	a	v	Использов. регистров PC	SP	AP FP	Индексирование
0-3	0-3	Литеральный	S^#литерал	y	f	f	f	f	-	-	-	f
4	4	Индексный	i[Rx]	y	y	y	y	y	f	y	y	f
5	5	Регистровый	Rn	y	y	y	f	y	u	uq	uo	f
6	6	Регистровый косвенный	(Rn)	y	y	y	y	y	u	y	y	y
7	7	С автоуменьшением	-(Rn)	y	y	y	y	y	u	y	y	ux
8	8	С автоувеличением	(Rn)+	y	y	y	y	y	p	y	y	ux
9	9	Косвенный с автоувеличением	@(Rn)+	y	y	y	y	y	p	y	y	ux
A	10	Со смещением (байт)	B^D(Rn)	y	y	y	y	y	p	y	y	y
B	11	Косвенный со смещением (байт)	@B^D(Rn)	y	y	y	y	y	p	y	y	y
C	12	Со смещением (слово)	W^D(Rn)	y	y	y	y	y	p	y	y	y
D	13	Косвенный со смещением (слово)	@W^D(Rn)	y	y	y	y	y	p	y	y	y
E	14	Со смещением (длинное слово)	L^D(Rn)	y	y	y	y	y	p	y	y	y
F	15	Косвенный со смещением (длинное слово)	@L^D(Rn)	y	y	y	y	y	p	y	y	y

АДРЕСАЦИЯ С ИСПОЛЬЗОВАНИЕМ ПРОГРАММНОГО СЧЕТЧИКА (РЕГИСТР 15)



Шест.	Дес.	Название режима	Обозначение	Тип доступа r	m	w	a	v	Индексирование
8	8	Непосредственный	I^#константа	y	u	u	y	y	u
9	9	Абсолютный	@#адрес	y	y	y	y	y	y
A	10	Относительный (смещение - байт)	B^адрес	y	y	y	y	y	y
B	11	Относительный косвенный (смещение - байт)	@B^адрес	y	y	y	y	y	y
C	12	Относительный (смещение - слово)	W^адрес	y	y	y	y	y	y
D	13	Относительный косвенный (смещение - слово)	@W^адрес	y	y	y	y	y	y
E	14	Относительный (смещение - длинное слово)	L^адрес	y	y	y	y	y	y
F	15	Относительный косвенный (смещение - длинное слово)	@L^адрес	y	y	y	y	y	y

Условные обозначения:

- D - смещение
- i - любой режим адресации, совместимый с индексным
- - - использование логически невозможно
- f - вызывает прерывание по резервному режиму адресации
- p - допустима адресация с использованием программного счетчика (PC)
- u - результат НЕПРЕДСКАЗУЕМ
- uq - результат НЕПРЕДСКАЗУЕМ для чисел в формате квадра- и октаслова и для чисел с плавающей точкой в форматах типа G и H, а также для битовых полей, если сумма значений смещения поля (начальная позиция поля) и длины битового поля превышает 32
- uo - результат НЕПРЕДСКАЗУЕМ для чисел в формате октаслова и чисел с плавающей точкой в формате H
- ux - результат НЕПРЕДСКАЗУЕМ, если один и тот же регистр используется одновременно в качестве индексного и базового
- y - да, такое использование допустимо и является корректным
- r - чтение
- m - модификация
- w - запись
- a - получение адреса
- v - обращение к битовому полю
- Rn - регистры общего назначения, n = 0-15
- Rx - регистры общего назначения, n = 0-14

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ, ПРИМЕНЯЕМЫЕ ДЛЯ СПЕЦИФИКАЦИИ ОПЕРАНДОВ

Описание операндов инструкции имеет следующий вид:

<имя операнда>.<тип доступа><тип данных>

где:

1. Имя операнда - краткое обозначение, отражающее роль операнда в контексте данной инструкции.

2. Тип доступа - однобуквенное обозначение разрешенного типа доступа к данным определяемых операндом:

   a - Вычисляется исполнительный адрес указанного операнда. Адрес возвращается в формате длинного слова, которое по существу является истинным операндом инструкции. Вычисление адреса осуществляется в контексте определяемом полем <тип данных> описания оператора, то есть значение приращения в операциях автоувеличения, автоуменьшения и индексирования выбирается в зависимости от формата данных.

   b - Ссылка на операнд отсутствует. Описание операнда задает смещение, определяющее адрес перехода. Формат смещения определяется полем <тип данных> описания оператора.

   m - Разрешено считывание определяемых операндом данных, данные могут быть модифицированы и вновь записаны. Отметьте, что если данные не были модифицированы, то перезапись данных может и не производиться. Однако, для операндов имеющих этот тип доступа (разрешающий модификацию данных), всегда осуществляется проверка на право доступа по чтению и записи.

   r - Разрешено только считывание данных.

   v - Вычисляется исполнительный адрес указанного операнда. Если этот адрес является адресом ячейки памяти, то он возвращается в формате длинного слова, которое по существу является истинным операндом инструкции. Вычисление адреса осуществляется в контексте определяемом полем <тип данных> описа-

ния оператора. Если обращение происходит к регистру Rn, то данные либо находятся в регистре Rn, либо в смежных регистрах Rn и R[n+1].

w - Разрешено только занесение (запись) данных.

3. Тип данных - буква, обозначающая данные какого типа представляет операнд:

b - целое число в формате байта
d - число с плавающей точкой в формате типа D
f - число с плавающей точкой в формате типа F
g - число с плавающей точкой в формате типа G
h - число с плавающей точкой в формате типа H
l - целое число в формате длинного слова
o - целое число в формате октаслова
q - целое число в формате квадраслова
w - целое число в формате слова
x - первый тип данных, из указанных в инструкции
y - второй тип данных, из указанных в инструкции
* - несколько смежных длинных слов
(применяется только для обозначения неявно заданных операндов)

4. Операнды, которые не должны указываться в инструкции явно, но наличие которых предполагается (регистры или ячейки памяти) заключаются в квадратные скобки [ ].

Обозначения, применяемые для описания состояния битов кодов условия:

. = установка/сброс бита зависит от выполнения условия
- = значение бита не изменяется
0 = бит сбрасывается
1 = бит устанавливается

НАБОР ИНСТРУКЦИЙ

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
9D	ACBB	Сложить, сравнить и перейти (байт)	limit.rb, add.rb, index.mb, displ.bw	. . . -
6F	ACBD	Сложить, сравнить и перейти (тип D)	limit.rd, add.rd, index.md, displ.bw	. . . -
4F	ACBF	Сложить, сравнить и перейти (тип F)	limit.rf, add.rf, index.mf, displ.bw	. . . -
4FFD	ACBG	Сложить, сравнить и перейти (тип G)	limit.rg, add.rg, index.mg, displ.bw	. . . -
6FFD	ACBH	Сложить, сравнить и перейти (тип H)	limit.rh, add.rh, index.mh, displ.bw	. . . -
F1	ACBL	Сложить, сравнить и перейти (дл.слово)	limit.rl, add.rl, index.ml, displ.bw	. . . -
3D	ACBW	Сложить, сравнить и перейти (слово)	limit.rw, add.rw, index.mw, displ.bw	. . . -
58	ADAWI	Заблокировать и сложить (слово)	add.rw, sum.mw	. . . .
80	ADDB2	Сложить (2-опер., байт)	add.rb, sum.mb	. . . .
81	ADDB3	Сложить (3-опер., байт)	add1.rb, add2.rb, sum.wb	. . . .
60	ADDD2	Сложить (2-опер., тип D)	add.rd, sum.md	. . . 0
61	ADDD3	Сложить (3-опер., тип D)	add1.rd, add2.rd, sum.wd	. . . 0
40	ADDF2	Сложить (2-опер., тип F)	add.rf, sum.mf	. . . 0
41	ADDF3	Сложить (3-опер., тип F)	add1.rf, add2.rf, sum.wf	. . . 0
40FD	ADDG2	Сложить (2-опер., тип G)	add.rg, sum.mg	. . . 0
41FD	ADDG3	Сложить (3-опер., тип G)	add1.rg, add2.rg, sum.wg	. . . 0
60FD	ADDH2	Сложить (2-опер., тип H)	add.rh, sum.mh	. . . 0
61FD	ADDH3	Сложить (3-опер., тип H)	add1.rh, add2.rh, sum.wh	. . . 0
C0	ADDL2	Сложить (2-опер., дл.слово)	add.rl, sum.ml	. . . .
C1	ADDL3	Сложить (3-опер., дл.слово)	add1.rl, add2.rl, sum.wl	. . . .
20	ADDP4	Сложить (4-опер., упаков.десятич.)	addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab, [R0-3.wl]	. . . 0
21	ADDP6	Сложить (6-опер., упаков.десятич.)	addlen.rw, addladdr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab, [R0-5.wl]	. . . 0

Продолжение таблицы

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
A0	ADDW2	Сложить (2-опер., слово)	add.rw, sum.mw	. . . .
A1	ADDW3	Сложить (3-опер., слово)	add1.rw, add2.rw, sum.ww	. . . .
D8	ADWC	Сложить с переносом	add.rl, sum.ml	. . . .
F3	AOBLEQ	Прибавить 1; перейти, если >= 0	limit.rl, index.ml, displ.bb	. . . -
F2	AOBLSS	Прибавить 1; перейти, если <= 0	limit.rl, index.ml, displ.bb	. . . -
78	ASHL	Арифметический сдвиг (дл.слово)	count.rb, src.rl, dst.wl	. . . 0
F8	ASHP	Арифметический сдвиг и округление (упаков.десятич.)	count.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab, [R0-3.wl]	. . . 0
79	ASHQ	Арифметический сдвиг (квадраслово)	count.rb, src.rq, dst.wq	. . . 0
E1	BBC	Перейти, если бит сброшен	pos.rl, base.vb, displ.bb, [field.rv]	- - - -
E5	BBCC	Перейти, если бит сброшен и сбросить его	pos.rl, base.vb, displ.bb, [field.mv]	- - - -
E7	BBCCI	Перейти, если бит сброшен и заблокировав сбросить его	pos.rl, base.vb, displ.bb, [field.mv]	- - - -
E3	BBCS	Перейти, если бит сброшен и установить его	pos.rl, base.vb, displ.bb, [field.mv]	- - - -
E0	BBS	Перейти, если бит установлен	pos.rl, base.vb, displ.bb, [field.rv]	- - - -
E4	BBSC	Перейти, если бит установлен и сбросить его	pos.rl, base.vb, displ.bb, [field.mv]	- - - -
E2	BBSS	Перейти, если бит установлен и установить его	pos.rl, base.vb, displ.bb, [field.mv]	- - - -
E6	BBSSI	Перейти, если бит установлен и заблокировав установить его	pos.rl, base.vb, displ.bb, [field.mv]	- - - -
1E	BCC	Перейти, если бит C сброшен	displ.bb	- - - -
1F	BCS	Перейти, если бит C установлен	displ.bb	- - - -
13	BEQL	Перейти, если равны	displ.bb	- - - -
13	BEQLU	Перейти, если равны (беззнаковые)	displ.bb	- - - -
18	BGEQ	Перейти, если больше или равны	displ.bb	- - - -
1E	BGEQU	Перейти, если больше или равны (беззнаковые)	displ.bb	- - - -
14	BGTR	Перейти, если больше	displ.bb	- - - -
1A	BGTRU	Перейти, если больше (беззнаковые)	displ.bb	- - - -
8A	BICB2	Сбросить биты (2-опер., байт)	mask.rb, dst.mb	. . 0 -
8B	BICB3	Сбросить биты (3-опер., байт)	mask.rb, src.rb, dst.wb	. . 0 -
CA	BICL2	Сбросить биты (2-опер., дл.слово)	mask.rl, dst.ml	. . 0 -
CB	BICL3	Сбросить биты (3-опер., дл.слово)	mask.rl, src.rl, dst.wl	. . 0 -
B9	BICPSW	Сбросить биты в PSW	mask.rw	. . . .
AA	BICW2	Сбросить биты (2-опер., слово)	mask.rw, dst.mw	. . 0 -
AB	BICW3	Сбросить биты (3-опер., слово)	mask.rw, src.rw, dst.ww	. . 0 -
88	BISB2	Установить биты (2-опер., байт)	mask.rb, dst.mb	. . 0 -
89	BISB3	Установить биты (3-опер., байт)	mask.rb, src.rb, dst.wb	. . 0 -
C8	BISL2	Установить биты (2-опер., дл.слово)	mask.rl, dst.ml	. . 0 -
C9	BISL3	Установить биты (3-опер., дл.слово)	mask.rl, src.rl, dst.wl	. . 0 -
B8	BISPSW	Установить биты в PSW	mask.rw	. . . .
AB	BISW2	Установить биты (2-опер., слово)	mask.rw, dst.mw	. . 0 -
A9	BISW3	Установить биты (3-опер., слово)	mask.rw, src.rw, dst.ww	. . 0 -
93	BITB	Проверить биты (байт)	mask.rb, src.rb	. . 0 -
D3	BITL	Проверить биты (дл.слово)	mask.rl, src.rl	. . 0 -
B3	BITW	Проверить биты (слово)	mask.rw, src.rw	. . 0 -
E9	BLBC	Перейти, если младший бит сброшен	src.rl, displ.bb	- - - -
E8	BLBS	Перейти, если младший бит установлен	src.rl, displ.bb	- - - -
15	BLEQ	Перейти, если меньше или равны	displ.bb	- - - -
1B	BLEQU	Перейти, если меньше или равны (беззнаковые)	displ.bb	- - - -
19	BLSS	Перейти, если меньше	displ.bb	- - - -
1F	BLSSU	Перейти, если меньше (беззнаковые)	displ.bb	- - - -
12	BNEQ	Перейти, если не равны	displ.bb	- - - -
12	BNEQU	Перейти, если не равны (беззнаковые)	displ.bb	0 0 0 0
03	BPT	Прерывание в контрольной точке	[-(KSP).w*]	- - - -
11	BRB	Перейти (смещение - байт)	displ.bb	- - - -
31	BRW	Перейти (смещение - слово)	displ.bw	

Продолжение таблицы

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
10	BSBB	Перейти к подпрограмме (смещение - байт)	displ.bb, [-(SP).wl]	- - - -
30	BSBW	Перейти к подпрограмме (смещение - слово)	displ.bw, [-(SP).wl]	- - - -
FDFF	BUGL	Внутренний контроль ОС VAX/VMS		0 0 0 0
FEFF	BUGW	Внутренний контроль ОС VAX/VMS		0 0 0 0
1C	BVC	Перейти, если бит V сброшен	displ.bb	- - - -
1D	BVS	Перейти, если бит V установлен	displ.bb	- - - -
FA	CALLG	Вызвать процедуру (общий список аргументов)	arglist.ab, dst.ab, [-(SP).w*]	0 0 0 0
FB	CALLS	Вызвать процедуру (список аргументов в стеке)	numarg.rl, dst.ab, [-(SP).w*]	0 0 0 0
8F	CASEB	Перейти в зависимости от выбора (селектор - байт)	selector.rb, base.rb, limit.rb, displ.bw-list	. . 0 .
CF	CASEL	Перейти в зависимости от выбора (селектор - дл. слово)	selector.rl, base.rl, limit.rl, displ.bw-list	. . 0 .
AF	CASEW	Перейти в зависимости от выбора (селектор - слово)	selector.rw, base.rw, limit.rw, displ.bw-list	. . 0 .
BD	CHME	Переключить процессор в режим "управления"	param.rw, [-(ySP).w*] y=MINU(E, PSLтекущ.режима)	0 0 0 0
BC	CHMK	Переключить процессор в режим "ядра"	param.rw, [-(KSP).w*]	0 0 0 0
BE	CHMS	Переключить процессор в режим "супервизора"	param.rw, [-(ySP).w*] y=MINU(S, PSLтекущ.режима)	0 0 0 0
BF	CHMU	Переключить процессор в режим "пользователя"	param.rw, [-(SP).w*]	0 0 0 0
94	CLRB	Очистить (байт)	dst.wb	0 1 0 -
7C	CLRD	Очистить (тип D)	dst.wd	0 1 0 -
D4	CLRF	Очистить (тип F)	dst.wf	0 1 0 -
7C	CLRG	Очистить (тип G)	dst.wg	0 1 0 -
7CFD	CLRH	Очистить (тип H)	dst.wh	0 1 0 -
D4	CLRL	Очистить (дл. слово)	dst.wl	0 1 0 -
7CFD	CLRO	Очистить (октаслово)	dst.wo	0 1 0 -
7C	CLRQ	Очистить (квадраслово)	dst.wq	0 1 0 -
B4	CLRW	Очистить (слово)	dst.ww	0 1 0 -
91	CMPB	Сравнить (байт)	src1.rb, src2.rb	. . 0 .
29	CMPC3	Сравнить (3-опер., симв.строка)	len.rw, src1addr.ab, src2addr.ab, [R0-3.wl]	. . 0 .
2D	CMPC5	Сравнить (5-опер., симв.строка)	src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab, [R0-3.wl]	. . 0 .
71	CMPD	Сравнить (тип D)	src1.rd, src2.rd	. . 0 0
51	CMPF	Сравнить (тип F)	src1.rf, src2.rf	. . 0 0
51FD	CMPG	Сравнить (тип G)	src1.rg, src2.rg	. . 0 0
71FD	CMPH	Сравнить (тип H)	src1.rh, src2.rh	. . 0 0
D1	CMPL	Сравнить (дл.слово)	src1.rl, src2.rl	. . 0 .
35	CMPP3	Сравнить (3-опер., упаков.десятич.)	len.rw, src1addr.ab, src2addr.ab, [R0-3.wl]	. . 0 0
37	CMPP4	Сравнить (4-опер., упаков.десятич.)	src1len.rw, src1addrab, src2len.rw, src2addr.ab, [R0-3.wl]	. . 0 0
EC	CMPV	Сравнить (битовое поле)	pos.rl, size.rb, base.vb, [field.rv], src.rl	. . 0 .
B1	CMPW	Сравнить (слово)	src1.rw, src2.rw	. . 0 .
ED	CMPZV	Сравнить дополненное незначащими нулями (битовое поле)	pos.rl, size.rb, base.vb, [field.rv], src.rl	. . 0 .
0B	CRC	Вычислить контрольную сумму	tbl.ab, initialcrc.rl, strlen.rw, stream.ab, [R0-3.wl]	. . 0 0
6C	CVTBD	Преобразовать (байт - тип D)	src.rb, dst.wd	. . . 0
4C	CVTBF	Преобразовать (байт - тип F)	src.rb, dst.wf	. . . 0
4CFD	CVTBG	Преобразовать (байт - тип G)	src.rb, dst.wg	. . . 0
6CFD	CVTBH	Преобразовать (байт - тип H)	src.rb, dst.wh	. . . 0
98	CVTBL	Преобразовать (байт - дл.слово)	src.rb, dst.wl	. . . 0
99	CVTBW	Преобразовать (байт - слово)	src.rb, dst.ww	. . . 0
68	CVTDB	Преобразовать (тип D - байт)	src.rd, dst.wb	. . . 0
76	CVTDF	Преобразовать (тип D - тип F)	src.rd, dst.wf	. . . 0

Продолжение таблицы

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
32FD	CVTDH	Преобразовать (тип D - тип H)	src.rd, dst.wh	. . . 0
6A	CVTDL	Преобразовать (тип D - дл.слово)	src.rd, dst.wl	. . . 0
69	CVTDW	Преобразовать (тип D - слово)	src.rd, dst.ww	. . . 0
48	CVTFB	Преобразовать (тип F - байт)	src.rf, dst.wb	. . . 0
56	CVTFD	Преобразовать (тип F - тип D)	src.rf, dst.wd	. . . 0
99FD	CVTFG	Преобразовать (тип F - тип G)	src.rf, dst.wg	. . . 0
98FD	CVTFH	Преобразовать (тип F - тип H)	src.rf, dst.wh	. . . 0
4A	CVTFL	Преобразовать (тип F - дл.слово)	src.rf, dst.wl	. . . 0
49	CVTFW	Преобразовать (тип F - слово)	src.rf, dst.ww	. . . 0
48FD	CVTGB	Преобразовать (тип G - байт)	src.rg, dst.wb	. . . 0
33FD	CVTGF	Преобразовать (тип G - тип F)	src.rg, dst.wf	. . . 0
56FD	CVTGH	Преобразовать (тип G - тип H)	src.rg, dst.wh	. . . 0
4AFD	CVTGL	Преобразовать (тип G - дл.слово)	src.rg, dst.wl	. . . 0
49FD	CVTGW	Преобразовать (тип G - слово)	src.rg, dst.ww	. . . 0
68FD	CVTHB	Преобразовать (тип H - байт)	src.rh, dst.wb	. . . 0
F7FD	CVTHD	Преобразовать (тип H - тип D)	src.rh, dst.wd	. . . 0
F6FD	CVTHF	Преобразовать (тип H - тип F)	src.rh, dst.wf	. . . 0
76FD	CVTHG	Преобразовать (тип H - тип G)	src.rh, dst.wg	. . . 0
6AFD	CVTHL	Преобразовать (тип H - дл.слово)	src.rh, dst.wl	. . . 0
69FD	CVTHW	Преобразовать (тип H - слово)	src.rh, dst.ww	. . . 0
F6	CVTLB	Преобразовать (дл.слово - байт)	src.rl, dst.wb	. . . 0
6E	CVTLD	Преобразовать (дл.слово - тип D)	src.rl, dst.wd	. . . 0
4E	CVTLF	Преобразовать (дл.слово - тип F)	src.rl, dst.wf	. . . 0
4EFD	CVTLG	Преобразовать (дл.слово - тип G)	src.rl, dst.wg	. . . 0
6EFD	CVTLH	Преобразовать (дл.слово - тип H)	src.rl, dst.wh	. . . 0
F9	CVTLP	Преобразовать (дл.слово - упаков.десятич.)	src.rl, dstlen.rw, dstaddr.ab, [R0-3.wl]	. . . 0
F7	CVTLW	Преобразовать (дл.слово - слово)	src.rl, dst.ww	. . . 0
36	CVTPL	Преобразовать (упаков.десятич. - дл.слово)	srclen.rw, srcaddr.ab, [R0-3wl], dst.wl	. . . 0
08	CVTPS	Преобразовать (упаков.десятич. - числ.строка с разд. ведущим знаком)	srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]	. . . 0
24	CVTPT	Преобразовать (упаков.десятич. - числ.строка с ведомым знаком)	srclen.rw, srcaddr.ab, rbladdr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]	. . . 0
6B	CVTRDL	Преобразовать с округлением (тип D - дл.слово)	src.rd, dst.wl	. . . 0
4B	CVTRFL	Преобразовать с округлением (тип F - дл.слово)	src.rf, dst.wl	. . . 0
4BFD	CVTRGL	Преобразовать с округлением (тип G - дл.слово)	src.rg, dst.wl	. . . 0
6BFD	CVTRHL	Преобразовать с округлением (тип H - дл.слово)	src.rh, dst.wl	. . . 0
09	CVTSP	Преобразовать (числ.строка с разд. ведущим знаком - упаков.десятич.)	srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]	. . . 0
26	CVTTP	Преобразовать (числ.строка с ведомым знаком - упаков.десятич.)	srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]	. . . 0
33	CVTWB	Преобразовать (слово - байт)	src.rw, dst.wb	. . . 0
6D	CVTWD	Преобразовать (слово - тип D)	src.rw, dst.wd	. . . 0
4D	CVTWF	Преобразовать (слово - тип F)	src.rw, dst.wf	. . . 0
4DFD	CVTWG	Преобразовать (слово - тип G)	src.rw, dst.wg	. . . 0
6DFD	CVTWH	Преобразовать (слово - тип H)	src.rw, dst.wh	. . . 0
32	CVTWL	Преобразовать (слово - дл.слово)	src.rw, dst.wl	. . . 0
97	DECB	Уменьшить на 1 (байт)	dif.mb	. . . .
D7	DECL	Уменьшить на 1 (дл.слово)	dif.ml	. . . .
B7	DECW	Уменьшить на 1 (слово)	dif.mw	. . . 0
86	DIVB2	Разделить (2-опер., байт)	divr.rb, quo.mb	. . . 0
87	DIVB3	Разделить (3-опер., байт)	divr.rb, divd.rb, quo.wb	. . . 0
66	DIVD2	Разделить (2-опер., тип D)	divr.rb, quo.md	. . . 0
67	DIVD3	Разделить (3-опер., тип D)	divr.rd, divd.rd, quo.wd	. . . 0
46	DIVF2	Разделить (2-опер., тип F)	divr.rf, quo.mf	. . . 0
47	DIVF3	Разделить (3-опер., тип F)	divr.rf, divd.rf, quo.wf	. . . 0
46FD	DIVG2	Разделить (2-опер., тип G)	divr.rg, quo.mg	. . . 0

Продолжение таблицы

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
47FD	DIVG3	Разделить (3-опер., тип G)	divr.rg, divd.rg, quo.wg	. . . 0
66FD	DIVH2	Разделить (2-опер., тип H)	divr.rh, quo.mh	. . . 0
67FD	DIVH3	Разделить (3-опер., тип H)	divr.rh, divd.rh, quo.wh	. . . 0
C6	DIVL2	Разделить (2-опер., дл. слово)	divr.rl, quo.ml	. . . 0
C7	DIVL3	Разделить (3-опер., дл. слово)	divr.rl, divd.rl, quo.wl	. . . 0
27	DIVP	Разделить (упаков. десятич.)	divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab, [R0-5.wl], -16(SP):-1(SP).wb]	. . . 0
A6	DIVW2	Разделить (2-опер., слово)	divr.rw, quo.mw	. . . 0
A7	DIVW3	Разделить (3-опер., слово)	divr.rw, divd.rw, quo.ww	. . . 0
38	EDITPC	Отредактировать (упаков. десятич. - симв. строка)	srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab, [R0-5.wl]	. . . .
7B	EDIV	Расширенное деление	divr.rl, divd.rq, quo.wl, rem.wl	. . . 0
74	EMODD	Расширенное умножение с выделением целой части числа (тип D)	mulr.rd, mulrx.rb, muld.rd, int.wl, fract.wd	. . . 0
54	EMODF	Расширенное умножение с выделением целой части числа (тип F)	mulr.rf, mulrx.rb, muld.rf, int.wl, fract.wf	. . . 0
54FD	EMODG	Расширенное умножение с выделением целой части числа (тип G)	mulr.rg, mulrx.rw, muld.rg, int.wl, fract.wg	. . . 0
74FD	EMODH	Расширенное умножение с выделением целой части числа (тип H)	mulr.rh, mulrx.rw, muld.rh, int.wl, fract.wh	. . . 0
7A	EMUL	Расширенное умножение	mulr.rl, muld.rl, add.rl, prod.wq	. . 0 0
FD	ESCD	Переключающий код D		. . . .
FE	ESCE	Переключающий код E		. . . .
FF	ESCF	Переключающий код F		. . . .
EE	EXTV	Выделить битовое поле	pos.rl, size.rb, base.vb, [field.rv], dst.wl	. . 0 -
EF	EXTZV	Выделить дополненное незначащими нулями битовое поле	pos.rl, size.rb, base.vb, [field.rv], dst.wl	. . 0 -
EB	FFC	Найти первый сброшенный бит	startpos.rl, size.rb, base.vb, [field.rv], findpos.wl	0 . 0 0
EA	FFS	Найти первый установленный бит	startpos.rl, size.rb, base.vb, [field.rv], findpos.wl	0 . 0 0
00	HALT	Останов	[-(KSP).w.*]	. . . .
96	INCB	Увеличить на 1 (байт)	sum.mb	. . . .
D6	INCL	Увеличить на 1 (дл. слово)	sum.ml	. . . .
B6	INCW	Увеличить на 1 (слово)	sum.mw	. . . .
0A	INDEX	Вычислить индекс массива	subscript.rl, low.rl, high.rl, size.rl, entry.rl, addr.wl	. . 0 0
5C	INSQHI	Включить элемент в начало очереди	entry.ab, header.aq	0 . 0 .
5D	INSQTI	Включить элемент в конец очереди	entry.ab, header.aq	0 . 0 .
0E	INSQUE	Включить элемент в очередь	entry.ab, addr.wl	. . 0 .
F0	INSV	Вставить битовое поле	src.rl, pos,rl, size.rb, base.vb,[field.wv]	- - - -
17	JMP	Безусловный переход	dst.ab	- - - -
16	JSB	Перейти к подпрограмме	dst.ab, [-(SP)+.wl]	- - - -
06	LDPCTX	Загрузить контекст процесса	[PCB.r*, -(KSP).w*]	- - - -
3A	LOCC	Найти символ	char.rb, len.rw, addr.ab, [R0-1.wl]	0 . 0 0
39	MATCHC	Найти подстроку символов	len1.rw, addr1.ab, len2.rw, addr2.ab, [R0-3.wl]	0 . 0 0
92	MCOMB	Переслать дополнение (байт)	src.rb, dst.wb	. . 0 -
D2	MCOML	Переслать дополнение (дл. слово)	src.rl, dst.wl	. . 0 -
B2	MCOMW	Переслать дополнение (слово)	src.rw, dst.ww	. . 0 -
DB	MFPR	Переслать из привилегированного регистра	procreg.rl, dst.wl	. . 0 -
8E	MNEGB	Переслать с изменением знака (байт)	src.rb, dst.wb	. . . .
72	MNEGD	Переслать с изменением знака (тип D)	src.rd, dst.wd	. . 0 0
52	MNEGF	Переслать с изменением знака (тип F)	src.rf, dst.wf	. . 0 0
52FD	MNEGG	Переслать с изменением знака (тип G)	src.rg, dst.wg	. . 0 0
72FD	MNEGH	Переслать с изменением знака (тип H)	src.rh, dst.wh	. . 0 0
CE	MNEGL	Переслать с изменением знака (дл. слово)	src.rl, dst.wl	. . . .
AE	MNEGW	Переслать с изменением знака (слово)	src.rw, dsr.ww	. . . .

Продолжение таблицы

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
9E	MOVAB	Переслать адрес (байт)	src.ab, dst.wl	. . 0 -
7E	MOVAD	Переслать адрес (тип D)	src.aq, dst.wl	. . 0 -
DE	MOVAF	Переслать адрес (тип F)	src.al, dst.wl	. . 0 -
7E	MOVAG	Переслать адрес (тип G)	src.aq, dsr.wl	. . 0 -
7EFD	MOVAH	Переслать адрес (тип H)	src.ao, dst.wl	. . 0 -
DE	MOVAL	Переслать адрес (дл. слово)	src.al, dst.wl	. . 0 -
7EFD	MOVAO	Переслать адрес (октаслово)	src.ao, dst.wl	. . 0 -
7E	MOVAQ	Переслать адрес (квадраслово)	src.aq, dst.wl	. . 0 -
3E	MOVAW	Переслать адрес (слово)	src.aw, dst.wl	. . 0 -
90	MOVB	Переслать (байт)	src.rb, dst.wb	. . 0 -
28	MOVC3	Переслать (3-опер., симв. строка)	len.rw, srcaddr.ab, dstaddr.ab, [R0-5.wl]	0 1 0 0
2C	MOVC5	Переслать (5-опер., симв. строка)	srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab, [R0-5.wl]	. . 0 .
70	MOVD	Переслать (тип D)	src.rd, dst.wd	. . 0 -
50	MOVF	Переслать (тип F)	src.rf, dsr.wf	. . 0 -
50FD	MOVG	Переслать (тип G)	src.rg, dst.wg	. . 0 -
70FD	MOVH	Переслать (тип H)	src.rh, dst.wh	. . 0 -
D0	MOVL	Переслать (дл. слово)	src.rl, dst.wl	. . 0 -
7DFD	MOVO	Переслать (октаслово)	src.ro, dst.wo	. . 0 -
34	MOVP	Переслать (упаков. десятич.)	len.rw, srcaddr.ab, dstaddr.ab, [R0-3.wl]	. . 0 -
DC	MOVPSL	Переслать PSL	dst.wl	- - - -
7D	MOVQ	Переслать (квадраслово)	src.rq, dst.wq	. . 0 -
2E	MOVTC	Переслать символы с перекодированием	srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, [R0-5.wl]	. . 0 .
2F	MOVTUC	Пересылать символы с перекодированием до обнаружения символа-переключателя	srclen.rw, srcaddr.ab, escape.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, [R0-5.wl]	. . . .
B0	MOVW	Переслать (слово)	src.rw, dst.ww	. . 0 -
9A	MOVZBL	Переслать дополненный незначащими нулями (байт - дл. слово)	src.rb, dst.wl	0 . 0 -
9B	MOVZBW	Переслать дополненный незначащими нулями (байт - слово)	src.rb, dst.ww	0 . 0 -
3C	MOVZWL	Переслать дополненное незначащими нулями (слово - дл. слово)	src.rw, dst.wl	0 . 0 -
DA	MTPR	Переслать в привилегированный регистр	src.rl, procreg.wl	. . . 0
84	MULB2	Умножить (2-опер., байт)	mulr.rb, prod.mb	. . . 0
85	MULB3	Умножить (3-опер., байт)	mulr.rb, muld.rb, prod.wb	. . . 0
64	MULD2	Умножить (2-опер., тип D)	mulr.rd, prod.md	. . . 0
65	MULD3	Умножить (3-опер., тип D)	mulr.rd, muld.rd, prod.wd	. . . 0
44	MULF2	Умножить (2-опер., тип F)	mulr.rf, prod.mf	. . . 0
45	MULF3	Умножить (3-опер., тип F)	mulr.rf, muld.rf, prod.wf	. . . 0
44FD	MULG2	Умножить (2-опер., тип G)	mulr.rg, prod.mg	. . . 0
45FD	MULG3	Умножить (3-опер., тип G)	mulr.rg, muld.rg, prod.wg	. . . 0
64FD	MULH2	Умножить (2-опер., тип H)	mulr.rh, prod.rh	. . . 0
65FD	MULH3	Умножить (3-опер., тип H)	mulr.rh, muld.rh, prod.wh	. . . 0
C4	MULL2	Умножить (2-опер., дл. слово)	mulr.rl, prod.rl	. . . 0
C5	MULL3	Умножить (3-опер., дл. слово)	mulr.rl, muld.rl, prod.wl	. . . 0
25	MULP	Умножить (упаков. десятич.)	mulrlen.rw, mulradr.ab, muldlen.rw, muldadr.ab, prodlen.rw, prodadr.ab, [R0-5.wl]	. . . 0
A4	MULW2	Умножить (2-опер., слово)	mulr.rw, prod.mw	. . . 0
A5	MULW3	Умножить (3-опер., слово)	mulr.rw, muld.rw, prod.ww	- - - -
01	NOP	Нет операции		. . . 0
75	POLYD	Вычислить полином (тип D)	arg.rd, degree.rw, tbladdr.ab, [R0-5.wl]	. . . 0
55	POLYF	Вычислить полином (тип F)	arg.rf, degree.rw, tbladdr.ab, [R0-3.wl]	

Продолжение таблицы

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
55FD	POLYG	Вычислить полином (тип G)	arg.rg, degree.rw, tbladdr.ab, [R0-5.wl]	. . . 0
75FD	POLYH	Вычислить полином (тип H)	arg.rh, degree.rw, tbladdr.ab, [R0-5.wl, -16(SP):-1(SP).wl]	. . . 0
BA	POPR	Извлечь из стека содержимое регистров	mask.rw, [(SP)+.r*]	- - - -
0C	PROBER	Проверить право на доступ по чтению	mode.rb, len.rw, base.ab	0 . 0 -
0D	PROBEW	Проверить право на доступ по записи	mode.rb, len.rw, base.ab	0 . 0 -
9F	PUSHAB	Включить в стек адрес (байт)	src.ab, [-(SP).wl]	. . 0 -
7F	PUSHAD	Включить в стек адрес (тип D)	src.aq, [-(SP).wl]	. . 0 -
DF	PUSHAF	Включить в стек адрес (тип F)	src.al, [-(SP).wl]	. . 0 -
7F	PUSHAG	Включить в стек адрес (тип G)	src.aq, [-(SP).wl]	. . 0 -
7FFD	PUSHAH	Включить в стек адрес (тип H)	src.ao, [-(SP).wl]	. . 0 -
DF	PUSHAL	Включить в стек адрес (дл.слово)	src.al, [-(SP).wl]	. . 0 -
7FFD	PUSHAO	Включить в стек адрес (октаслово)	src.ao, [-(SP).wl]	. . 0 -
7F	PUSHAQ	Включить в стек адрес (квадраслово)	src.aq, [-(SP).wl]	. . 0 -
3F	PUSHAW	Включить в стек адрес (слово)	src.aw, [-(SP).wl]	. . 0 -
DD	PUSHL	Включить в стек (дл.слово)	src.al, [-(SP).wl]	. . 0 -
BB	PUSHR	Включить в стек содержимое регистров	mask.rw,[-(SP).w*]	- - - -
02	REI	Возврат после обработки исключительной ситуации или прерывания	[(SP)+.r*]	. . . .
5E	REMQHI	Исключить элемент из начала очереди	header.aq, addr.wl	0 . . .
5F	REMQTI	Исключить элемент из конца очереди	header.aq, addr.wl	0 . . .
0F	REMQUE	Исключить элемент из очереди	entry.ab, addr.wl	. . . .
04	RET	Возврат из процедуры	[(SP)+.r]	. . . .
9C	ROTL	Циклический сдвиг (дл.слово)	count.rb, src.rl, dst.wl. . 0 -	
05	RSB	Возврат из подпрограммы	[(SP)+.rl]	- - - -
57		Зарезервировано		
5A		Зарезервировано		
5B		Зарезервировано		
77		Зарезервировано		
FE		Зарезервировано		
FF		Зарезервировано		
D9	SBWC	Вычесть с заемом (переносом)	sub.rl, dif.ml	. . . .
2A	SCANC	Искать символ принадлежащий заданному множеству	len.rw, addr.ab, tbladdr.ab, mask.rb, [R0-1.wl]	C . 0 0
3B	SKPC	Пропустить символ	char.rb,len.rw,addr.ab,[R0-1.wl]	0 . 0 0
F4	SOBGEQ	Вычесть 1; перейти, если >= 0	index.ml, displ.bb	. . . -
F5	SOBGTR	Вычесть 1; перейти, если <= 0	index.ml, displ.bb	. . . -
2B	SPANC	Пропустить символы принадлежащие заданному множеству	len.rw, addr.ab, tbladdr.ab, mask.rb, [R0-3wl]	0 . 0 0
82	SUBB2	Вычесть (2-опер., байт)	sub.rb, dif.mb	. . . .
83	SUBB3	Вычесть (3-опер., байт)	sub.rb, min.rb, dif.wb	. . . .
62	SUBD2	Вычесть (2-опер., тип D)	sub.rd, dif.md	. . . 0
63	SUBD3	Вычесть (3-опер., тип D)	sub.rd, min.rd, dif.wd	. . . 0
42	SUBF2	Вычесть (2-опер., тип F)	sub.rf, dif.mf	. . . 0
43	SUBF3	Вычесть (3-опер., тип F)	sub.rf, min.rf, dif.wf	. . . 0
42FD	SUBG2	Вычесть (2-опер., тип G)	sub.rg, dif.mg	. . . 0
43FD	SUBG3	Вычесть (3-опер., тип G)	sub.rg, min.rg, dif.wg	. . . 0
62FD	SUBH2	Вычесть (2-опер., тип H)	sub.rh, dif.mh	. . . 0
63FD	SUBH3	Вычесть (3-опер., тип H)	sub.rh, min.rh, dif.wh	. . . 0
C2	SUBL2	Вычесть (2-опер., дл.слово)	sub.rl, dif.ml	. . . .
C3	SUBL3	Вычесть (3-опер., дл.слово)	sub.rl, min.rl, dif.wl	. . . .
22	SUBP4	Вычесть (4-опер., упаков.десятич.)	sublen.rw, subaddr.ab, diflen.rw, difaddr.ab, [R0-3.wl	. . . 0
23	SUBP6	Вычесть (6-опер., упаков.десятич.)	sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, difaddr.ab, [R0-5.wl]	. . . 0
A2	SUBW2	Вычесть (2-опер., слово)	sub.rw, dif.mw	. . . .
A3	SUBW3	Вычесть (3-опер., слово)	sub.rw, min.rw , dif.ww	. . . .
07	SVPCTX	Сохранить контекст процесса	[(SP)+.r*,-(KSP).w*]	- - - -

Окончание таблицы

Код операции	Мнемоника	Операция	Операнды	Коды условий N Z V C
95	TSTB	Проверить (байт)	src.rb	. . 0 0
73	TSTD	Проверить (тип D)	src.rd	. . 0 0
53	TSTF	Проверить (тип F)	src.rf	. . 0 0
53FD	TSTG	Проверить (тип G)	src.rg	. . 0 0
73FD	TSTH	Проверить (тип H)	src.rh	. . 0 0
D5	TSTL	Проверить (дл.слово)	src.rl	. . 0 0
B5	TSTW	Проверить (слово)	src.rw	. . 0 0
FC	XFC	Вызов функции расширения	операнды определяются пользователем	0 0 0 0
8C	XORB2	Исключающее ИЛИ (2-опер., байт)	mask.rb, dst.mb	. . 0 -
8D	XORB3	Исключающее ИЛИ (3-опер., байт)	mask.rb, src.rb, dst.wb	. . 0 -
CC	XORL2	Исключающее ИЛИ (2-опер., дл.слово)	mask.rl, dst.ml	. . 0 -
CD	XORL3	Исключающее ИЛИ (3-опер., дл.слово)	mask.rl, src.rl, dst.wl	. . 0 -
AC	XORW2	Исключающее ИЛИ (2-опер., слово)	mask.rw, dst.mw	. . 0 -
AD	XORW3	Исключающее ИЛИ (3-опер., слово)	mask.rw, src.rw, dst.ww	. . 0 -

ИНСТРУКЦИИ В ПОРЯДКЕ ВОЗРАСТАНИЯ ЧИСЛОВОГО ЗНАЧЕНИЯ КОДА ОПЕРАЦИИ

00	HALT	10	BSBB	1E	BCC	2C	MOVC5	3A	LOCC
01	NOP	11	BRB	1E	BGEQU	2D	CMPC5	3B	SKPC
02	REI	12	BNEQ	1F	BCS	2E	MOVTC	3C	MOVZWL
03	BPT	12	BNEQU	1F	BLSSU	2F	MOVTUC	3D	ACBW
04	RET	13	BEQL	20	ADDP4	30	BSBW	3E	MOVAW
05	RSB	13	BEQLU	21	ADDP6	31	BRW	3F	PUSHAW
06	LDPCTX	14	BGTR	22	SUBP4	32	CVTWL	40	ADDF2
07	SVPCTX	15	BLEQ	23	SUBP6	32FD	CVTDH	40FD	ADDG2
08	CVTPS	16	JSB	24	CVTPT	33	CVTWB	41	ADDF3
09	CVTSP	17	JMP	25	MULP	33FD	CVTGF	41FD	ADDG3
0A	INDEX	18	BGEQ	26	CVTTP	34	MOVP	42	SUBF2
0B	CRC	19	BLSS	27	DIVP	35	CMPP3	42FD	SUBG2
0C	PROBER	1A	BGTRU	28	MOVC3	36	CVTPL	43	SUBF3
0D	PROBEW	1B	BLEQU	29	CMPC3	37	CMPP4	43FD	SUBG3
0E	INSQUE	1C	BVC	2A	SCANC	38	EDITPC	44	MULF2
0F	REMQUE	1D	BVS	2B	SPANC	39	MATCHC	44FD	MULG2
45	MULF3	4E	CVTLF	57	Reserved	65	MULD3	6E	CVTLD
45FD	MULG3	4EFD	CVTLG	58	ADAWI	65FD	MULH3	6EFD	CVTLH
46	DIVF2	4F	ACBF	5A	Reserved	66	DIVD2	6F	ACBD
46FD	DIVG2	4FFD	ACBG	5B	Reserved	66FD	DIVH2	6FFD	ACBH
47	DIVF3	50	MOVF	5C	INSQHI	67	DIVD3	70	MOVD
47FD	DIVG3	50FD	MOVG	5D	INSQTI	67FD	DIVH3	70FD	MOVH
48	CVTFB	51	CMPF	5E	REMQHI	68	CVTDB	71	CMPD
48FD	CVTGB	51FD	CMPG	5F	REMQTI	68FD	CVTHB	71FD	CMPH
49	CVTFW	52	MNEGF	60	ADDD2	69	CVTDW	72	MNEGD
49FD	CVTGW	52FD	MNEGG	60FD	ADDH2	69FD	CVTHW	72FD	MNEGH
4A	CVTFL	53	TSTF	61	ADDD3	6A	CVTDL	73	TSTD
4AFD	CVTGL	53FD	TSTG	61FD	ADDH3	6AFD	CVTHL	73FD	TSTH
4B	CVTRFL	54	EMODF	62	SUBD2	6B	CVTRDL	74	EMODD
4BFD	CVTRGL	54FD	EMODG	62FD	SUBH2	6BFD	CVTRHL	74FD	EMODH
4C	CVTBF	55	POLYF	63	SUBD3	6C	CVTBD	75	POLYD
4CFD	CVTBG	55FD	POLYG	63FD	SUBH3	6CFD	CVTBH	75FD	POLYH
4D	CVTWF	56	CVTFD	64	MULD2	6D	CVTWD	76	CVTDF
4DFD	CVTWG	56FD	CVTGH	64FD	MULH2	6DFD	CVTWH	76FD	CVTHG
77	Reserved	7F	PUSHAG	8E	MNEGB	9E	MOVAB	B0	MOVW
78	ASHL	7F	PUSHAQ	8F	CASEB	9F	PUSHAB	B1	CMPW
79	ASHQ	7FFD	PUSHAH	90	MOVB	A0	ADDW2	B2	MCOMW
7A	EMUL	7FFD	PUSHAO	91	CMPB	A1	ADDW3	B3	BITW
7B	EDIV	80	ADDB2	92	MCOMB	A2	SUBW2	B4	CLRW
7C	CLRD	81	ADDB3	93	BITB	A3	SUBW3	B5	TSTW
7C	CLRG	82	SUBB2	94	CLRB	A4	MULW2	B6	INCW
7C	CLRQ	83	SUBB3	95	TSTB	A5	MULW3	B7	DECW

7CFD	CLRH	84	MULB2	96	INCB	A6	DIVW2	B8	BISPSW
7CFD	CLRO	85	MULB3	97	DECB	A7	DIVW3	B9	BICPSW
7D	MOVQ	86	DIVB2	98	CVTBL	A8	BISW2	BA	POPR
7DFD	MOVO	87	DIVB3	98FD	CVTFH	A9	BISW3	BB	PUSHR
7E	MOVAD	88	BISB2	99	CVTBW	AA	BICW2	BC	CHMK
7E	MOVAG	89	BISB3	99FD	CVTFG	AB	BICW3	BD	CHME
7E	MOVAQ	8A	BICB2	9A	MOVZBL	AC	XORW2	BE	CHMS
7EFD	MOVAH	8B	BICB3	9B	MOVZBW	AD	XORW3	BF	CHMU
7EFD	MOVAO	8C	XORB2	9C	ROTL	AE	MNEGW	C0	ADDL2
7F	PUSHAD	8D	XORB3	9D	ACBB	AF	CASEW	C1	ADDL3

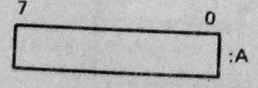
C2	SUBL2	D4	CLRF	E3	BBCS	F5	SOBGTR
C3	SUBL3	D4	CLRL	E4	BBSC	F6	CVTLB
C4	MULL2	D5	TSTL	E5	BBCC	F6FD	CVTHF
C5	MULL3	D6	INCL	E6	BBSSI	F7	CVTLW
C6	DIVL2	D7	DECL	E7	BBCCI	F7FD	CVTHD
C7	DIVL3	D8	ADWC	E8	BLBS	F8	ASHP
C8	BISL2	D9	SBWC	E9	BLBC	F9	CVTLP
C9	BISL3	DA	MTPR	EA	FFS	FA	CALLG
CA	BICL2	DB	MFPR	EB	FFC	FB	CALLS
CB	BICL3	DC	MOVPSL	EC	CMPV	FC	XFC
CC	XORL2	DD	PUSHL	ED	CMPZV	FD	ESCD
CD	XORL3	DE	MOVAF	EE	EXTV	FDFF	BUGL
CE	MNEGL	DE	MOVAL	EF	EXTZV	FE	ESCE
CF	CASEL	DF	PUSHAF	F0	INSV	FE	Reserved
D0	MOVL	DF	PUSHAL	F1	ACBL	FEFF	BUGW
D1	CMPL	E0	BBS	F2	AOBLSS	FF	ESCF
D2	MCOML	E1	BBC	F3	AOBLEQ	FF	Reserved
D3	BITL	E2	BBSS	F4	SOBGEQ		

ТАБЛИЦА ПРЕОБРАЗОВАНИЯ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

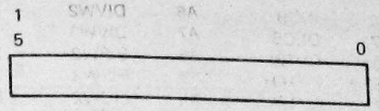
	8		7		6		5		4		3		2		1
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,929,104	9	150,994,994	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	12,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,685,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

ТИПЫ ДАННЫХ

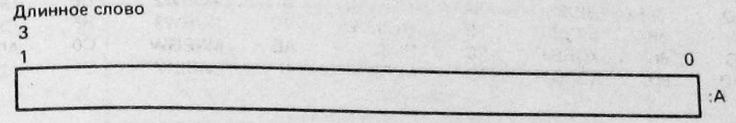
Байт



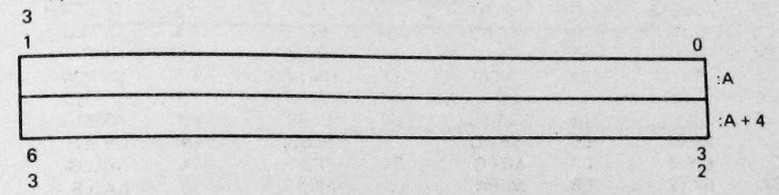
Слово



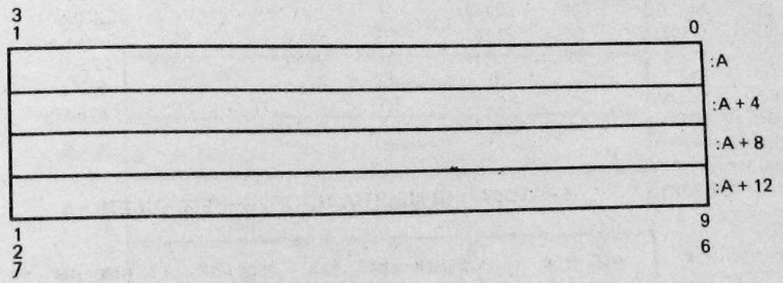
Длинное слово



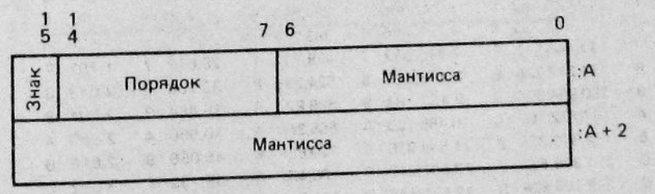
Квадраслово



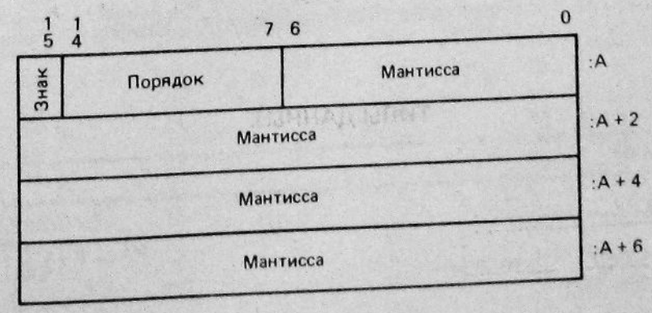
Октаслово



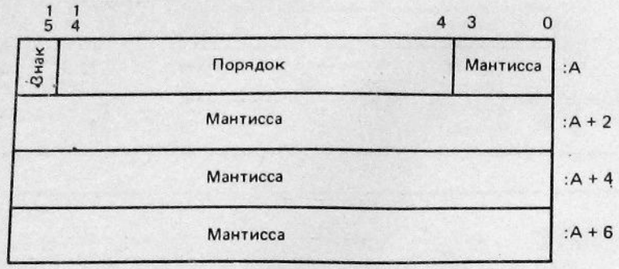
Формат типа F числа с плавающей точкой



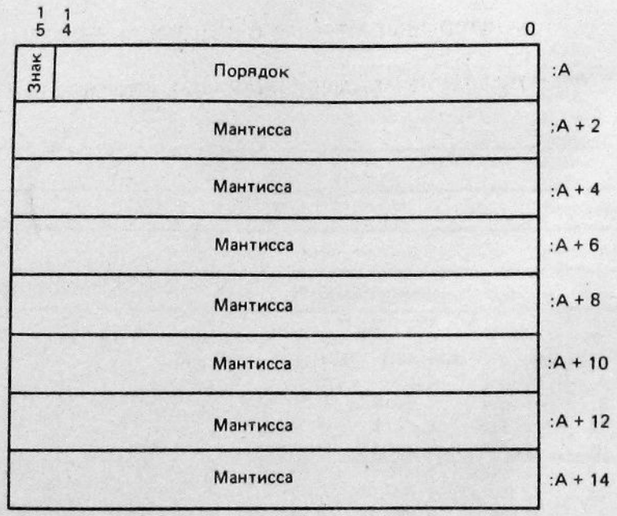
Формат типа D числа с плавающей точкой



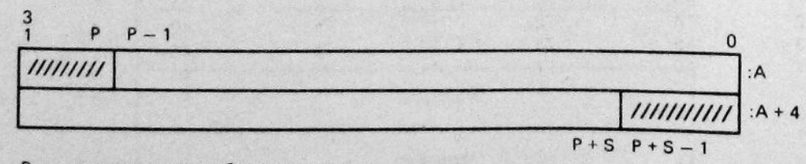
Формат типа G числа с плавающей точкой



Формат типа H числа с плавающей точкой

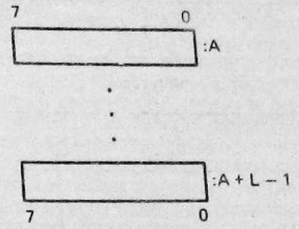


БИТОВОЕ ПОЛЕ ПЕРЕМЕННОЙ ДЛИНЫ



P – начальная позиция битового поля
S – длина поля

СИМВОЛЬНАЯ СТРОКА



ЧИСЛОВЫЕ СТРОКИ С ВЕДОМЫМ ЗНАКОМ

ПРЕДСТАВЛЕНИЕ МЛАДШИХ ЗНАЧАЩИХ ЦИФР ЧИСЛА И ЗНАКА

Цифра	Зонный формат 1: Десятичный код	Зонный формат 1: Шестнадцатеричный код	Зонный формат 1: Символы кода ASCII	Зонный формат 2: Десятичный код	Зонный формат 2: Шестнадцатеричный код	Символы кода ASCII: Нормальное представление	Символы кода ASCII: Альтернативное представление
				123	7B	{	0[?
0	48	30	0	65	41	A	1
1	49	31	1	66	42	B	2
2	50	32	2	67	43	C	3
3	51	33	3	68	44	D	4
4	52	34	4	69	45	E	5
5	53	35	5	70	46	F	6
6	54	36	6	71	47	G	7
7	55	37	7	72	48	H	8
8	56	38	8	73	49	I	9
9	57	39	9				

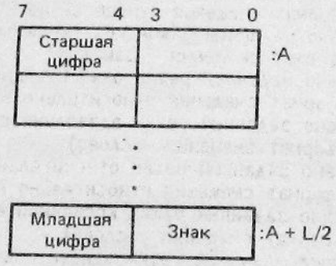
Цифра	Зонный формат 1: Десятичный код	Зонный формат 1: Шестнадцатеричный код	Зонный формат 1: Символы кода ASCII	Зонный формат 2: Десятичный код	Зонный формат 2: Шестнадцатеричный код	Символы кода ASCII: Нормальное представление	Символы кода ASCII: Альтернативное представление
				125	7D	}	]!:
– 0	112	70	p	74	4A	J	
– 1	113	71	q	75	4B	K	
– 2	114	72	r	76	4C	L	
– 3	115	73	s	77	4D	M	
– 4	116	74	t	78	4E	N	
– 5	117	75	u	79	4F	O	
– 6	118	76	v	80	50	P	
– 7	119	77	w	81	51	Q	
– 8	120	78	x	82	52	R	
– 9	121	79	y				

Представление числовой строки с раздельным ведущим знаком

Знак	Десятичный код	Шестнадцатеричный код	Символы кода ASCII
+	43	2B	+
+	32	20	⟨пробел⟩
-	45	2D	

Лучше всего для представления знака + применять символ кода ASCII +

ФОРМАТ УПАКОВАННОГО ДЕСЯТИЧНОГО ЧИСЛА



ПРЕДСТАВЛЕНИЕ ЗНАКА УПАКОВАННОГО ДЕСЯТИЧНОГО ЧИСЛА (В ПОЛУБАЙТЕ)

Знак	Десятичный код	Шестнадцатеричный код
+	10, 12, 14 или 15,	A, C, E или F
–	11 или 13	B или D

Для представления знака + лучше всего использовать код 12, для знака – используется код 13

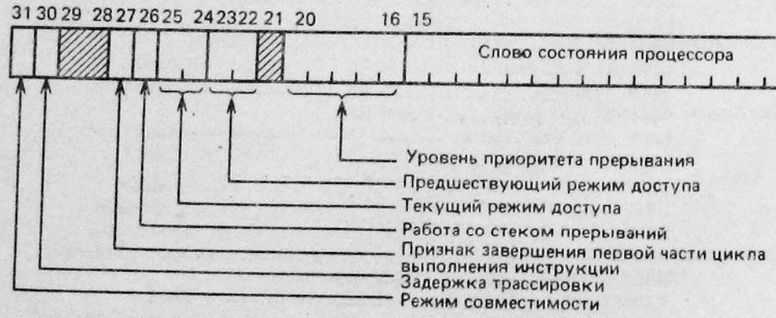
ПРИМЕРЫ ОБОЗНАЧЕНИЯ РЕЖИМОВ АДРЕСАЦИИ

S^#5	явно заданный режим литеральной адресации
#5	литеральный режим (будет выбран ассемблером при оптимизации)
R10	регистровый режим адресации
(R10)	регистровая косвенная адресация
-(R10)	режим адресации с автоуменьшением
(R10)+	режим адресации с автоувеличением
#START	режим непосредственной адресации
I^#1	явно заданный режим непосредственной адресации
@(R10)+	косвенная адресация с автоувеличением
@#START	абсолютная адресация
1(R10)	адресация со смещением заданным в формате байта (формат - байт, выбирается ассемблером при оптимизации)
0(R10)	регистровая косвенная адресация (будет выбрана при оптимизации)
@1(R10)	косвенная со смещением заданным в формате байта (формат - байт, выбирается ассемблером при оптимизации)
@(R10)	косвенная со смещением (предполагается, что для смещения ассемблер выберет формат байта)

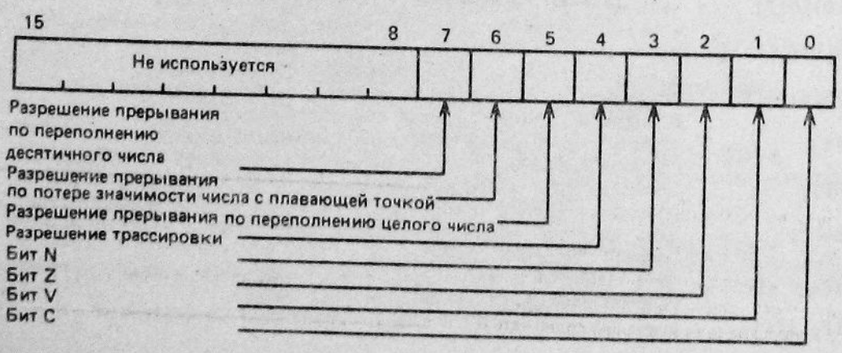
START	относительный режим адресации (формат смещения относительно PC будет выбран ассемблером)
@START	относительный косвенный режим адресации (формат смещения относительно PC будет выбран ассемблером)
1234(R10)	адресация со смещением заданным в формате слова (этот формат будет выбран ассемблером при оптимизации)
@1234(R10)	косвенная со смещением заданным в формате слова (формат - слово, будет выбран ассемблером при оптимизации)
12345678(R10)	адресация со смещением заданным в формате длинного слова (данный формат будет выбран ассемблером при оптимизации)
@12345678(R10)	косвенная со смещением заданным в формате длинного слова (данный формат будет выбран ассемблером при оптимизации)
B^12(R10)	явно заданный режим адресации со смещением (формат смещения - байт)
B^START	явно заданный режим относительной адресации (формат смещения относительно PC - байт)
@B^12(R10)	явно заданный режим косвенной адресации со смещением (формат смещения - байт)
@B^START	явно заданный режим относительной косвенной адресации (формат смещения относительно PC - байт)
W^12(R10)	явно заданный режим адресации со смещением (формат смещения - слово)
W^START	явно заданный режим относительной адресации (формат смещения относительно PC - слово)
@W^12(R10)	явно заданный режим косвенной адресации со смещением (формат смещения - слово)
@W^START	явно заданный режим относительной косвенной адресации (формат смещения относительно PC - слово)
L^12(R10)	явно заданный режим адресации со смещением (формат смещения - длинное слово)
L^START	явно заданный режим относительной адресации (формат смещения относительно PC - длинное слово)
@L^12(R10)	явно заданный режим косвенной адресации со смещением (формат смещения - длинное слово)
@L^START	явно заданный режим относительной косвенной адресации (формат смещения относительно PC - длинное слово)
(R10)[R11]	регистровая косвенная адресация с индексированием
-(R10)[R11]	режим адресации с автоуменьшением и индексированием
(R10)+[R11]	режим адресации с автоувеличением и индексированием
@(R10)+[R11]	косвенная адресация с автоувеличением и индексированием
@#START[R11]	абсолютная адресация с индексированием
1(R10)[R11]	адресация со смещением заданным в формате байта и индексированием (формат - байт, выбирается ассемблером)
0(R10)[R11]	регистровая косвенная адресация с индексированием (будет выбрана при оптимизации)
@1(R10)[R11]	косвенная со смещением заданным в формате байта и индексированием (формат - байт, выбирается ассемблером)
@(R10)[R11]	косвенная со смещением и индексированием (предполагается, что для смещения ассемблер выберет формат байта)
1234(R10)[R11]	адресация со смещением заданным в формате слова и индексированием (формат - слово, выбирается ассемблером)
@1234(R10)[R11]	косвенная со смещением заданным в формате слова и индексированием (формат - слово, выбирается ассемблером)
12345678(R10)[R11]	адресация со смещением заданным в формате длинного слова и индексированием (формат выбирается ассемблером)
@12345678(R10)[R11]	косвенная со смещением заданным в формате длинного слова и индексированием (формат выбирается ассемблером)
START[R11]	относительный режим адресации с индексированием (формат смещения относительно PC будет выбран ассемблером)
@START[R11]	относительный косвенный режим адресации с индексированием (формат смещения относительно PC будет выбран ассемблером)
B^12(R10)[R11]	явно заданный режим адресации со смещением и индексированием (формат смещения - байт)
B^START[R11]	явно заданный режим относительной адресации с индексированием (формат смещения относительно PC - байт)
@B^12(R10)[R11]	явно заданный режим косвенной адресации со смещением

	и индексированием (формат смещения - байт)
@B^START[R11]	явно заданный режим относительной косвенной адресации
	с индексированием (формат смещения относительно PC - байт)
W^12(R10)[R11]	явно заданный режим адресации со смещением и индексированием
	(формат смещения - слово)
W^START[R11]	явно заданный режим относительной адресации с индексированием
	(формат смещения относительно PC - слово)
@W^12(R10)[R11]	явно заданный режим косвенной адресации со смещением
	и индексированием (формат смещения - слово)
@W^START[R11]	явно заданный режим относительной косвенной адресации
	с индексированием (формат смещения относительно PC - слово)
L^12(R10)[R11]	явно заданный режим адресации со смещением и индексированием
	(формат смещения - длинное слово)
L^START[R11]	явно заданный режим относительной адресации с ин
	(формат смещения относительно PC - длинное слово)
@L^12(R10)[R11]	явно заданный режим косвенной адресации со смещением
	и индексированием (формат смещения - длинное слово)
@L^START[R11]	явно заданный режим относительной косвенной адресации
	с индексированием (формат смещения относительно PC - длинное слово)

Длинное слово состояния процессора



Слово состояния процессора



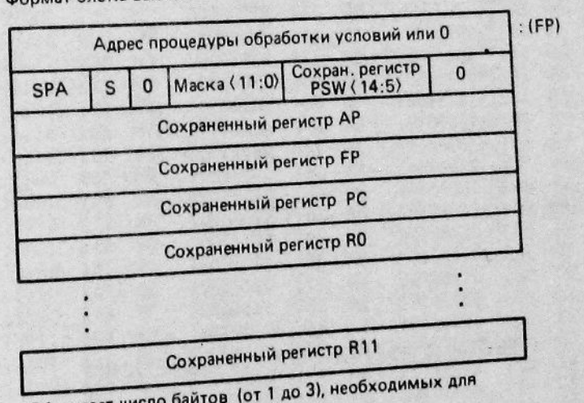
Степени числа 2

2^n	n
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20
2097152	21
4194304	22
8388608	23
16777216	24

Степени числа 16

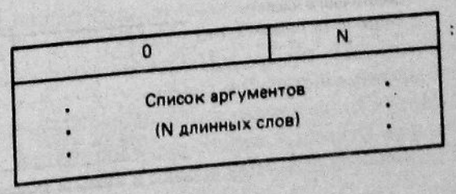
16^n	n
1	0
16	1
256	2
4096	3
65536	4
1048576	5
16777216	6
268435456	7
4294967296	8
68719476736	9
1099511627776	10
17592186044416	11
281474976710656	12
4503599627370496	13
72057594037927936	14
1152921504606846976	15

Формат блока вызова



SPA задает число байтов (от 1 до 3), необходимых для выравнивания границы стека
S – устанавливается для CALLS; сбрасывается для CALLG

Формат списка аргументов



Первый байт списка является счетчиком числа аргументов (число без знака). Старшие 24 бита первого длинного слова зарезервированы для будущего использования и должны содержать 0

ПРИЛОЖЕНИЕ Ж. ТАБЛИЦА КОДОВ ASCII

Шест. код	Символ ASCII	Шест. код	Символ ASCII	Шест. код	Символ ASCII	Шест. код	Символ ASCII
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

ОТВЕТЫ К ИЗБРАННЫМ УПРАЖНЕНИЯМ

УПРАЖНЕНИЯ 1.1

1. а) Г. Холлерит изобрел метод обработки данных, подготовленных на перфокартах. Разработанное им оборудование использовалось при переписи населения США в 1890 г.
   б) Г. Айкен завершил разработку первой большой цифровой вычислительной машины в 1944 г. Поскольку в вычислительной машине МАРК I использовались электромеханические устройства, такие как реле, она работала медленнее, чем разработанные вслед за ней электронные вычислительные машины.
   в) П. Эккерт – соразработчик (вместе с Д. Мочли) электронной цифровой вычислительной машины. Эккерт и Мочли завершили создание первой электронной вычислительной машины "Эниак" в 1946 г.
   г) Д. Мочли является соразработчиком электронной цифровой вычислительной машины.
   д) Дж. фон Нейман – легендарный математик, чье имя тесно связано с разработкой цифровых вычислительных машин в 1940 и в начале 1950-х гг.

   е) Дж. Бэкус руководил группой специалистов фирмы IBM, занимавшихся разработкой одного из первых языков высокого уровня – Фортрана, созданного в 1954 г. Язык программирования Фортран предназначался для выполнения сложных математических вычислений.
3. Ламповые вычислительные машины очень громоздкие, потребляют много энергии, кроме того, они медленно работают и очень дорогие, а также менее надежные, чем последующие модели.

УПРАЖНЕНИЯ 1.2

1.

```
J:      .BLKL   1
K:      .BLKL   1
        .ENTRY  START,0
        MOVL    #50,J
        MULL3   #32,J,K
        $EXIT_S
        .END    START
```

УПРАЖНЕНИЯ 2.1

2.

Дес	Шест	Двоичное	Дес	Шест	Двоичное	Дес	Шест	Двоичное	Дес	Шест	Двоичное
101	65	1100101	126	7E	1111110	151	97	10010111	176	B0	10110000
102	66	1100110	127	7F	1111111	152	98	10011000	177	B1	10110001
103	67	1100111	128	80	10000000	153	99	10011001	178	B2	10110010
104	68	1101000	129	81	10000001	154	9A	10011010	179	B3	10110011
105	69	1101001	130	82	10000010	155	9B	10011011	180	B4	10110100
106	6A	1101010	131	83	10000011	156	9C	10011100	181	B5	10110101
107	6B	1101011	132	84	10000100	157	9D	10011101	182	B6	10110110
108	6C	1101100	133	85	10000101	158	9E	10011110	183	B7	10110111
109	6D	1101101	134	86	10000110	159	9F	10011111	184	B8	10111000
110	6E	1101110	135	87	10000111	160	A0	10100000	185	B9	10111001
111	6F	1101111	136	88	10001000	161	A1	10100001	186	BA	10111010
112	70	1110000	137	89	10001001	162	A2	10100010	187	BB	10111011
113	71	1110001	138	8A	10001010	163	A3	10100011	188	BC	10111100
114	72	1110010	139	8B	10001011	164	A4	10100100	189	BD	10111101
115	73	1110011	140	8C	10001100	165	A5	10100101	190	BE	10111110
116	74	1110100	141	8D	10001101	166	A6	10100110	191	BF	10111111
117	75	1110101	142	8E	10001110	167	A7	10100111	192	C0	11000000
118	76	1110110	143	8F	10001111	168	A8	10101000	193	C1	11000001
119	77	1110111	144	90	10010000	169	A9	10101001	194	C2	11000010
120	78	1111000	145	91	10010001	170	AA	10101010	195	C3	11000011
121	79	1111001	146	92	10010010	171	AB	10101011	196	C4	11000100
122	7A	1111010	147	93	10010011	172	AC	10101100	197	C5	11000101
123	7B	1111011	148	94	10010100	173	AD	10101101	198	C6	11000110
124	7C	1111100	149	95	10010101	174	AE	10101110	199	C7	11000111
125	7D	1111101	150	96	10010110	175	AF	10101111	200	C8	11001000

3. а. 1000 б. 1011 в. 1010
   г. 100001 д. 100100 е. 100010
   ж. 10001000 з. 1001011111 и. 1100001111101

4. а. 10 б. 1 в. 0
   г. 1101 д. 10010 е. 11000
   ж. 101110 з. 11111011 и. 1001001001

5. а. 5 б. 26 в. 58
г. 46 д. 51 е. 93
ж. 99 з. 111 и. 229
к. 939 л. 2977 м. 12974

6. а. ^X5 б. ^X1A в. ^X3A
г. ^X2E д. ^X33 е. ^X5D
ж. ^X63 з. ^X6F и. ^XE5
к. ^X3AB л. ^XBA1 м. ^X32AE

7. а. ^XB1 б. ^XFD в. ^X18D
г. ^XAFD д. ^X981 е. ^X1B18
ж. ^XA82F з. ^X85FC и. ^X3B88

8. а. ^X4D б. ^XC9 в. ^X39
г. ^X9F9 д. ^X247 е. ^X1FE
ж. ^XA343 з. ^XBFE и. ^X1B8A

9. а. 10 б. 16 в. 256
г. 31 д. 42 е. 69
ж. 164 з. 200 и. 218
к. 4,980 л. 6,717 м. 62,430

10. а. 1010 б. 1111 в. 11000
г. 100101 д. 101010 е. 110011
ж. 1010101 з. 1100001 и. 10000000
к. 1111101000 л. 110111100011 м. 1011101101111

11. а. ^XA б. ^XF в. ^X18
г. ^X25 д. ^X2A е. ^X33
ж. ^X55 з. ^X61 и. ^X80
к. ^X3EB л. ^XDE3 м. ^X176F

УПРАЖНЕНИЯ 2.2

2. а. 00001000 нет переноса (3+5=8)
б. 00000111 перенос (11+(-4)=7)
в. 11110111 перенос ((-3)+(-6)=-9)
г. 11101011 перенос ((-20)+(-1)=-21)
д. 00000010 перенос (39+(-37)=2)
е. 00001000 нет переноса ((-99)+37=-62)

3. а. 11111110 заем (3-5=2)
б. 00001111 заем (11-(-4)=15)
в. 00000011 нет заема ((-3)-(-6)=3)
г. 11101101 заем ((-20)-(-1)=-19)
д. 01001100 заем (39-(-37)=76)
е. 01111000 нет заема ((-99)-37=120)

4. а. ^X0B нет переноса б. ^X07 перенос в. ^XF7 перенос
г. ^XEB перенос д. ^X02 перенос е. ^XC2 перенос

5. а. ^XFE нет заема б. ^X0F заем в. ^X03 нет заема
г. ^XED заем д. ^X04 заем е. ^X78 нет заема

7. а. ^X37
б. ^XA1
в. ^X7C
г. ^XEF
д. ^X39E5
е. ^XABF2
ж. ^X3BEF58D1
з. ^XDE698E5E

8. а. 1 б. -1 в. 15 г. 26
д. -6 е. -96 ж. 122 з. -125

9. а. 1 б. 255 в. 15 г. 26
д. 250 е. 160 ж. 122 з. 131

УПРАЖНЕНИЯ 3.1

1. а. ^X9F90
б. ^X349F
в. ^X0BCD
г. ^X12349F90
д. ^X00123494
е. ^XCDEF9F00

	Байт	Адрес
а.	1F	00000203
б.	24	00000201
в.	1E	00000202
г.	1A	00000200
д.	00	00000203

УПРАЖНЕНИЯ 3.2

1.

	Содержимое	Адрес
а.	0000000F	00000208
б.	00000200	00000208
в.	00000000	00000204
г.	0000020B	00000204
д.	00000002	0000020C
е.	00000110	00000204
ж.	000000E0	00000204
з.	00000000	00000208
и.	0000000F	00000208
	FFFFFFFF	0000020C

2.

Содержимое	Адрес
00000020	00000200
00000040	00000204
00000060	00000208
00000051	0000020C

УПРАЖНЕНИЯ 4.1

3. а. ^X14
   б. ^X80
   в. ^X80
   г. ^X0080
   д. ^XFF80
   е. ^X00000128
   ж. ^X80
   з. ^X00001000

УПРАЖНЕНИЯ 6.1

1. а. 253, 253 б. 65533, -3 в. 32767, 32767
   г. 63254, -2282 д. 32768, -32768, е. 32769, -32767
   ж. 33021, -32515 з. 30293, 30293 и. 28399, 28399

2.

	N	Z	V	C
а. ^XFF03	1	0	0	1
б. ^X0003	0	0	0	1
в. ^X8001	1	0	0	1
г. ^X08EA	0	0	0	1
д. ^X8000	1	0	1	1
е. ^X7FFF	0	0	0	1
ж. ^X7F03	0	0	0	1
з. ^X89AB	1	0	0	1
и. ^X9111	1	0	0	1

3.

	N	Z	V	C
а. ^X0217	0	0	0	0
б. ^XEA46	1	0	1	0
в. ^X6E48	0	0	0	1
г. ^XFFFE	1	0	0	1
д. ^X0000	0	1	0	1
е. ^X45AF	0	0	0	1

7. (устанавливаются отмеченные биты)

	INCL				DECL				INCW				DECW			
	N	Z	V	C	N	Z	V	C	N	Z	V	C	N	Z	V	C
а.	1	0	1	0	0	0	0	0	0	1	0	1	1	0	0	0
б.	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1
в.	0	1	0	1	1	0	0	0	0	1	0	1	1	0	0	0
г.	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1
д.	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
е.	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0

УПРАЖНЕНИЯ 6.2

1. а. ^X03AC б. ^X85E0 (переполнение) в. ^XFFFF
   г. ^XFFFE д. ^X0001 е. ^XFFFF (переполнение)

2. а. ^X0050 б. ^X0000 в. недопустимо (прерывание)
   г. ^X0000 д. ^X0001 е. недопустимо (установится бит V)

УПРАЖНЕНИЯ 7.1

1. а. ^X0000C4E0 <=0 б. ^X0000C4FB <=0
   в. ^X000010E4 <=0 г. ^X0000112C <=0
   д. ^X00001111 <=0 е. ^X0000040E <=0
   ж. ^X00000E24 <=0 з. ^X000002FB <=0
   и. ^X0000FFC4 <=0 к. ^X00000005C <=0

УПРАЖНЕНИЯ 7.2

1.

Значение R0	Содержимое	Адрес	Значение R0	Содержимое	Адрес
а. 00000000					
в. 00001204	00000000	00001200	б. не изменилось	00000000	00001200
д. не изменилось	00001200	00001200	г. 000011FC	00000000	000011FC
ж. 00001208	0000120C	00001204	е. 0000120C		
и. не изменилось	не изменилось	00001200	з. 00001204	0000120C	00001204
л. 000011F4			к. 000011FC	000011F4	000011F8
			м. 00001208	000011F4	00001208

УПРАЖНЕНИЯ 8.2

1.

	N	Z	V	C
а. B = ^XFFFFFFFF	1	0	0	-
б. C = ^X10305070	0	0	0	-
в. A = ^X0D0B0907	0	0	0	-
г. C = ^XF2F4F6F8	1	0	0	-
д. не изменилось	0	0	0	-
е. не изменилось	0	0	0	-

10. а. A = B(первое значение)
    B = A(первое значение)
    б. B = 0
    г. B = -A
    в. B = -A
    д. C = A + B

УПРАЖНЕНИЯ 9.1

1. ^X00000005
   ^X00004212

2. а. ^X7FFC8FF4 <= ^X00000256
   SP <= ^X7FFC8FF4
   б. R1 <= ^X00000005
   SP <= ^X7FFC8FFC
   в. SP <= ^X7FFC9000
   г. ^X7FFC8FF4 <= ^X000004E6 до ^X000004E9
   SP <= ^X7FFC8FF4
   PC <= ^X0000067F

```
д.  ^X7FFC8FF4 <= ^X000004E5
    SP <= ^X7FFC8FF4
    PC <= ^X00000256
е.  SP <= ^X7FFC8FFC
    PC <= ^X00000005
ж.  SP <= ^X7FFC8FF4
з.  ^X7FFC8FF4 <= ^X00000005
    SP <= ^X7FFC8FF4
и.  ^X7FFC8FF4 <= ^XFFFFF624
    ^X7FFC8FF0 <= ^X00000256
    SP <= ^X7FFC8FF0
к.  R1 <= ^X00000005
    R2 <= ^X00004212
    SP <= ^X7FFC9000
```

УПРАЖНЕНИЯ 10.1

3. а. 11 б. 25
   в. 3 г. 1
   д. 3 е. 22
   ж. 25 з. 13

4. а. переместимый б. переместимый
   в. абсолютный г. сложный
   д. абсолютный е. абсолютный
   ж. сложный з. переместимый

5. Пропущены следующие операторы:

```
B:      .BLKL   D-C
K = J
J = I+5
```

отмечены как ошибочные следующие операторы:

```
A:      .BLKL   K
B:      .BLKL   D-C
D:      .BLKL   K+I
K = J
```

6.
```
I = 3
J = I+5
K = J
A:      .BLKL   K
C:      .BLKL   I
D:      .BLKL   K+I
B:      .BLKL   D-C
```


УПРАЖНЕНИЯ 10.2

2. а. ничего
   в. MOVL R, R0
   д. ничего
   б. MOVL #3, W
   г. MOVL H, Q
   е. MOVL I, J
      MOVL G, F

УПРАЖНЕНИЯ 10.3

1. а. PROB EXP=A, TOTAL=B, SUM=C, MAX=D
   б. PROB EXP=A+5, TOTAL=B(R5)[R3], SUM=-(SP), MAX=#27
   в. PROB EXP=A, MAX=D
   г. PROB TOTAL=R5, SUM=<A, B>, MAX=LONGSYMBOL
   д. PROB SUM=(SP)+
   е. PROB EXP=<A, B, C, D>

2. а. SECOND A, B, C, D
   б. SECOND C, A, B, D
   в. SECOND ADS, ORK, W, CHAN
   г. SECOND ,(R5+,,-(SP) Это выглядит безобразно, но может быть правильно
   д. SECOND <A, B>
   е. SECOND ,,,<A, B>

5.
```
.MACRO      ADD     A=X, B=#100, C=(R1)+, D=#0, ANS=R0
ADDL3       A, B, ANS
ADDL2       C, ANS
ADDL2       D, ANS
.END        ADD
```

УПРАЖНЕНИЯ 12.1

1. а. 51500000 б. 53374000
   в. 51314159 г. 47500000
   д. 73803500 е. 35492300
   ж. 69849600 з. 41954200

2. а. 3.00000E 00
      3
   в. 8.94026E-01
      0.894026
   д. -2.93465E 05
      -293465
   ж. 9.50125E-13
      .000000000000950125
   б. -7.42000E 02
      -742
   г. 8.05216E-06
      .00000805216
   е. -1.00000E 06
      -1000000
   з. -7.90881E 13
      -79088100000000

УПРАЖНЕНИЯ 12.2

1. а. ^X00004080 б. ^X0000C220 в. ^X000043C8
   г. ^X00004000 д. ^XCCCD3ECC е. ^XD70ABD23

2. а. 1.0 б. 0.5
   в. 0.25 г. 1.8446744E 19 или 2.0**64
   д. -1.0 е. 0.2

3. Число с плавающей точкой (из п. 1) в формате типа D

а.	00000000	00004080	1.0
б.	00000000	0000C220	-10.0
в.	00000000	000043C8	100.0
г.	00000000	00004000	0.5
д.	CCCDCCCC	CCCC3ECC	0.1
е.	A3D73D70	D70ABD23	-0.01

Число с плавающей точкой (из п. 2) в формате типа D

а.	00000000	00004080	1.0
б.	00000000	00004000	0.5
в.	00000000	00003F80	0.25
г.	00000000	00006080	2.0**64
д.	00000000	0000C080	-1.0
е.	CCCDCCCC	CCCC3F4C	0.2

Число с плавающей точкой (из п. 1) в формате типа G

а.	00000000	00004010	1.0
б.	00000000	0000C044	-10.0
в.	00000000	00004079	100.0
г.	00000000	00004000	0.5
д.	999A9999	99993FD9	0.1
е.	147B47AE	7AE1BFA4	-0.01

Число с плавающей точкой (из п. 2) в формате типа G

а.	00000000	00004010	1.0
б.	00000000	00004000	0.5
в.	00000000	00003FF0	0.25
г.	00000000	00004410	2.0**64
д.	00000000	0000C010	-1.0
е.	999A9999	99993FE9	0.2

Число с плавающей точкой (из п. 1) в формате типа H

а.	00000000	00000000	00000000	00004001	1.0
б.	00000000	00000000	00000000	4000C004	-10.0
в.	00000000	00000000	00000000	90000407	100.0
г.	00000000	00000000	00000000	00004000	0.5
д.	999A9999	99999999	99999999	99993FFD	0.1
е.	147B47AE	7AE1AE14	E147147A	47AEBFFA	-0.01

Число с плавающей точкой (из п. 2) в формате типа H

а.	00000000	00000000	00000000	00004001	1.0
б.	00000000	00000000	00000000	00004000	0.5
в.	00000000	00000000	00000000	00003FFF	0.25
г.	00000000	00000000	00000000	00004041	2.0**64
д.	00000000	00000000	00000000	0000C001	-1.0
е.	999A9999	99999999	99999999	99993FFE	0.2

УПРАЖНЕНИЯ 14.1

1. а) **Оверлей.** Оверлей представляет собой часть кода программы, который при необходимости может быть загружен из более медленной внешней памяти в основную, что позволяет ЭВМ выполнять программы, не помещающиеся целиком в ее основной памяти.
   б) **Виртуальный адрес.** Виртуальный адрес – адрес по которому, как предполагает программа, производится выборка или запись информации. Вследствие того, что программа перемещается при настройке адресов, реальный адрес может отличаться от виртуального.
   в) **Виртуальная память.** Система виртуальной памяти – это система, реализующая процесс автоматического перекрытия с помощью смены страниц, находящихся в быстродействующей основной памяти ЭВМ, страницами, загружаемыми с внешнего запоминающего устройства большой емкости. Это существенно расширяет пространство памяти, доступное для программы.
   г) **Кэш-память.** Кэш-память представляет собой сверхоперативную память, используемую для буферизации запросов при обменах с основной памятью.
   д) **Страница.** Страница является блоком памяти, представляющим собой единицу информационного обмена в системах виртуальной памяти. На ЭВМ семейства VAX длина страницы составляет 512 байтов.
   е) **Фрейм.** Фрейм – блок памяти, буферизуемый в кэш-памяти.
   ж) **Контекст процесса.** Контекст образуют ресурсы вычислительной машины, требуемые процессу для выполнения программы.
2. Пространство P0 или область программы предназначается для программ и их данных. Диапазон виртуальных адресов, охватываемый пространством начинается с адреса ^X00000000 и кончается адресом ^X3FFFFFFF. Пространство P1 или область управления предназначено для организации стека и различных вспомогательных функций обслуживания программы пользователя, пространство P1 занимает виртуальные адреса в диапазоне ^X7FFFFFFF – ^X40000000; такое обратное указание границ диапазона показывает, что область памяти, выделяемая в этом пространстве, расширяется в сторону младших адресов. Пространство системы начинается с виртуального адреса ^X80000000 и кончается адресом ^XFFFFFFFF. Однако в настоящее время верхнюю часть области памяти, расположенную выше адреса ^XBFFFFFFF, использовать не разрешается.

УПРАЖНЕНИЯ 14.2

1. а) **Привилегия** – уровень, на котором программа имеет доступ к ресурсам вычислительной системы.

   б) **Режим доступа** – определяет уровень привилегии, на котором в текущий момент времени программа имеет доступ к памяти и инструкциям. На ЭВМ семейства VAX имеется четыре режима доступа: ядра, управления, супервизора и пользователя.

   в) **Регистры процессора** – внутренние ячейки памяти процессора. Имеется 16 регистров общего назначения, а также специальные регистры, такие как регистр PSL.

   г) **Шина** – совокупность электрических проводников, используемая для подключения к процессору памяти, и периферийных устройств.

   д) **UNIBUS** – асинхронная шина с 16-битовым трактом данных, применяемая для подключения многих устройств ввода-вывода на ЭВМ семейства VAX.

   е) **MASSBUS** – шина с 16-битовым трактом данных, применяемая для подключения к ЭВМ семейства VAX высокоскоростных устройств ввода-вывода.

   ж) **Q-BUS** – асинхронная шина с 16-битовым трактом данных, имеющая низкую стоимость, применяемая для ЭВМ серии MicroVAX.

   з) **Прямой доступ к памяти** – метод, который позволяет устройствам ввода-вывода производить выборку или запись в памяти без вмешательства процессора.

   и) **Прерывание** – передача управления программе, вызванная по сигналу от устройства ввода-вывода.

   к) **Ловушка** (trap) – передача управления, вызванная возникновением состояния ошибки в выполняющейся программе. Прерывание по исключительной ситуации такого типа возникает сразу после завершения выполнения инструкции, а сохраняемое значение программного счетчика является адресом следующей инструкции, которая должна выполняться, если бы не произошло прерывание.

   л) **Отказ** (fault) – передача управления, вызванная состоянием ошибки в выполняющейся программе. Прерывание по исключительной ситуации этого типа возникает до завершения выполнения инструкции, а сохраняемое значение программного счетчика представляет собой адрес инструкции, при выполнении которой произошло такое прерывание.

2. а) 15 возможных комбинаций.

   б) Да.

УПРАЖНЕНИЯ 15.1

4. Директива .LIST служит для управления форматом листинга, создаваемого во время второго прохода ассемблирования.

   Директива .WORD используется для резервирования памяти и инициализации одного или нескольких слов. Во время первого прохода ассемблера осуществляется резервирование памяти, для чего при обнаружении данной директивы счетчик адресов увеличивается на соответствующее значение. Во время второго прохода для списка выражений, находящегося в поле операндов директивы, генерируется объектный код. Значение счетчика адресов изменяется так же, как и на первом проходе.

   Директива .REPT дает указание ассемблеру повторить следующий после нее фрагмент текста программы столько раз, сколько задано в поле операндов директивы. Конец фрагмента текста помечает директива .ENDR. Действие директивы одинаково для двух проходов.

   Директива .ASCII служит для резервирования памяти и инициализации символов в коде ASCII. Число символов может быть от 0 и больше. Во время первого прохода осуществляется резервирование памяти, для чего счетчик адресов увеличивается на соответствующее значение. Во время второго прохода для строки символов в коде ASCII, находящейся в поле операндов директивы, генерируется объектный код. Кроме того, значение счетчика адресов изменяется так же, как и при первом проходе.

УКАЗАТЕЛЬ КОДОВ ОПЕРАЦИЙ

ADAWI,
ADDBx,
ADDfx,
ADDLx,
ADDP4,
ADDP6,
ADDWx,
ADWC,
AOBLEQ,
AOBPSS,
ASHL,
ASHP,
ASHQ,

BBCCI,
BBSSI,
BCC,
BCS,
BEQL,
BGEQ,
BGEQU,
BGTR,
BGTRU,
BICBx,
BICLx,
BICPSW,
BICWx,
BISBx,
BISLx,
BISPSW,
BISWx,
BITB,
BITL,
BITW,
BLBC,
BLBS,
BLEQ,
BLEQU,
BLSS,
BLSSU,
BNEQ,
BRB,
BRW,
BSBB,
BSBW,
BVC,
BVS,

CMPW,
CVTBf,
CVTBL,
CVTBW,
CVTfB,
CVTff,
CVTfL,
CVTfW,
CVTLB,
CVTLf,
CVTLP,
CVTLW,
CVTPL,
CVTPS,
CVTRfL,
CVTSP,
CVTWB,
CVTWf,
CVTWL,

DECB,
DECL,
DECW,
DIVBx,
DIVfx,
DIVLx,
DIVP,
DIVWx,

EDITPC,
EDIV,
EMODf,
EMUL,
EXTV,
EXTZV,

INCB,
INCL,
INCW,
INDEX,
INSQHI,
INSQTI,
INSQUE,
INSV,
IOINIT,

JMP,
JSB,

MOVAW,
MOVB,
MOVC3,
DOVC5,
MOVf,
MOVL,
MOVP,
MOVPSL,
MOVQ,
MOVTC,
MOVTUC,
MOVW,
MOVZBL,
MOVZBW,
MOVZWL,
MTPR,
MULBx,
MULfx,
MULLx,
MULP,
MULWx,

POLYf,
POPR,
PROBER,
PROBEW,
PUSHAB,
PUSHAf,
PUSHAL,
PUSHAO,
PUSHAQ,
PUSHAW,
PUSHL,
PUSHR,

REMQHI,
REMQTI,
REMQUE,
RET,
ROTL,
RSB,

SBWC,
SCANC,
SKPC,
SOBGEQ,
SOBGTR,
SPANC,

CALLG,
CALLS,
CLRB,
CLRfx,
CLRL,
CLRQ,
CLRW,
CMPB,
CMPC3,
CMPC5,
CMPf,
CMPL,
CMPP3,
CMPP4,
CMPV,
LOCC,
MATCHC,
MCOMB,
MCOML,
MCOMW,
MFPR,
MNEGB,
MNEGf,
MNEGL,
MNEGW,
MOVAB,
MOVAf,
MOVAL,
MOVAO,
MOVAQ,
SUBBx,
SUBfx,
SUBLx,
SUBP4,
SUBP6,
SUBWx,
TSTB,
TSTf,
TSTL,
TSTW,
XORBx,
XORLx,
XORWx,

Примечание: x – цифра 2 или 3, f – формат числа с плавающей точкой (буква D, F, G или H)

ОГЛАВЛЕНИЕ

Предисловие 7

Глава 1. Введение

1.1. Аннотация 9
1.2. Исторический обзор 9
1.3. Развитие программного обеспечения 11
1.4. Семейство ЭВМ VAX 14
1.5. Начальные сведения 17

Глава 2. Представление чисел, арифметические и логические операции в вычислительных машинах

2.1. Введение 21
2.2. Системы счисления 21
2.3. Двоичные числа 23
2.4. Преобразование чисел 25
2.5. Шестнадцатеричные числа 28
2.6. Двоичная арифметика в дополнительных кодах 32
2.7. Представление чисел в ЭВМ семейства VAX 34
2.8. Булева логика 38
2.9. Другие способы кодирования 39

Глава 3. Процессор и структура памяти

3.1. Введение 43
3.2. Организация ЭВМ 43
3.3. Организация памяти ЭВМ семейства VAX 44
3.4. Взаимодействие процессора с памятью 47
3.5. Программы в машинном коде 49
3.6. Программный счетчик 51
3.7. Ошибки выполнения 53
3.8. Некоторые дополнительные инструкции 55
3.9. Дополнительные спецификаторы операндов 57

Глава 4. Ассемблер и процесс ассемблирования

4.1. Введение 61
4.2. Пример использования языка ассемблера 62
4.3. Процесс ассемблирования 64
4.4. Завершенный вариант программы на языке ассемблера 68
4.5. Синтаксис языка ассемблера VAX-11 MACRO 69
4.6. Дополнительные возможности ассемблера 71
4.7. Компоновщик 77
4.8. Выполнение программы 78

Глава 5. Управление последовательностью выполнения программы

5.1. Введение 86
5.2. Операторы переходов 87
5.3. Коды условий 92

5.4. Формат инструкций переходов . . . 95
5.5. Дополнительные инструкции . . . 98
5.6. Организация циклов . . . 100
5.7. Регистры процессора . . . 101
5.8. Подпрограммы . . . 106

Глава 6. Арифметические операции на ЭВМ VAX

6.1. Введение . . . 112
6.2. Числа со знаком и без знака . . . 112
6.3. Умножение и деление . . . 120
6.4. Арифметические операции с повышенной точностью . . . 124
6.5. Алгоритмы умножения и деления . . . 126

Глава 7. Массивы

7.1. Введение . . . 132
7.2. Массивы в программах на языке ассемблера . . . 133
7.3. Другие режимы адресации . . . 139
7.4. Режимы адресации с использованием регистров общего назначения . . . 146
7.5. Режимы адресации с использованием программного счетчика . . . 149
7.6. Многомерные массивы . . . 153

Глава 8. Символьная информация

8.1. Представление символьной информации . . . 158
8.2. Обработка символьной информации . . . 164
8.3. Упрощенный ввод-вывод . . . 169
8.4. Инструкции побитовой обработки . . . 171
8.5. Другие способы представления символьной информации . . . 177

Глава 9. Подпрограммы

9.1. Введение . . . 182
9.2. Вызов подпрограммы . . . 182
9.3. Раздельное ассемблирование и глобальные имена . . . 189
9.4. Вызов процедуры . . . 192
9.5. Связь между программами на языке ассемблера и на языках высокого уровня . . . 195
9.6. Рекурсивные функции . . . 203

Глава 10. Макроинструкции и условное ассемблирование

10.1. Введение . . . 206
10.2. Повторяющиеся фрагменты программы . . . 206
10.3. Символические выражения . . . 210
10.4. Макроинструкции . . . 215
10.5. Условное ассемблирование . . . 218
10.6. Вложенность и рекурсия . . . 221
10.7. Более сложные приемы написания макроинструкции . . . 224

Глава 11. Системный ввод и вывод

11.1 Введение . . . 231
11.2. Использование системного сервиса . . . 232
11.3. Запросы ввода-вывода с очередями . . . 233
11.4. Ввод-вывод с терминала . . . 236
11.5. Дополнительные соображения по организации ввода-вывода с очередями . . . 240
11.6. Файлы и записи . . . 246
11.7. Подсистема управления записями (RMS) . . . 249
11.8. Использование подсистемы RMS . . . 253

Глава 12. Числа с плавающей точкой

12.1. Введение . . . 261
12.2. Числа с фиксированной и с плавающей точкой . . . 261
12.3. Операции над числами с плавающей точкой . . . 264
12.4. Числа с плавающей точкой в ЭВМ семейства VAX . . . 270
12.5. Операции с плавающей точкой в ЭВМ семейства VAX . . . 275
12.6. Сложные инструкции для работы с числами с плавающей точкой . . . 280

Глава 13. Инструкции обработки символьных, десятичных и других данных

13.1. Введение . . . 284
13.2. Инструкции семейства MOVC . . . 284
13.3. Инструкции обработки символьных строк . . . 287
13.4. Инструкции семейства CMPC . . . 289
13.5. Другие инструкции обработки символьных строк . . . 291
13.6. Арифметика упакованных десятичных чисел . . . 295
13.7. Числовые строки . . . 300
13.8. Битовые поля переменной длины . . . 305
13.9. Очереди и инструкции для работы с ними . . . 307

Глава 14. Элементы архитектуры ЭВМ семейства VAX

14.1. Введение . . . 310
14.2. Управление памятью и виртуальная память . . . 310
14.3. Управление памятью на ЭВМ семейства VAX . . . 315
14.4. Защита и привилегии . . . 319
14.5. Системные интерфейсы и их структура . . . 323
14.6. Физический уровень ввода-вывода на ЭВМ семейства VAX . . . 325

Глава 15. Язык ассемблера более подробно

15.1. Введение . . . 329
15.2. Формат программы . . . 329
15.3. Объектный код . . . 335
15.4. Файлы выполняемых образов . . . 340
15.5. Выполнение программы . . . 342

Приложение А. Подготовка и выполнение программ, написанных на языке ассемблера

А.1. Программа на языке ассемблера . . . 347
А.2. Подготовка программ на вычислительной системе VAX . . . 347
А.3. Начало сеанса работы на вычислительной системе VAX . . . 347
А.4. Команды операционной системы VAX/VMS . . . 348
А.5. Создание файла . . . 349
А.6. Выполнение программы . . . 351
А.7. Ошибки программирования и отладка . . . 352
А.8. Выполнение отлаженной программы . . . 354
А.9. Получение документированной копии . . . 354

Приложение Б. Программы ввода-вывода

Приложение В. Программы ввода-вывода на языках высокого уровня

Приложение Г. Работа с файлами в ОС VAX/VMS

Г.1. Файлы . . . 361
Г.2. Каталоги . . . 362
Г.3. Вывод содержимого файла . . . 362
Г.4. Вывод содержимого каталога . . . 363
Г.5. Копирование файлов . . . 363

Г.6. Использование подкаталогов 364
Г.7. Создание подкаталогов 364
Г.8. Выполнение операций над группами файлов 364
Г.9. Удаление файлов 365

Приложение Д. Редактор текста EDT и его применение

Д.1. Назначение редактора текста 366
Д.2. Работа с редактором EDT в экранном режиме 367
Д.3. Ввод команд с помощью вспомогательной клавиатуры 370
Д.4. Редактирование с телетайпа 374

Приложение Е. Инструкции и типы данных ЭВМ семейства VAX

Приложение Ж. Таблица кодов ASCII

Ответы к избранным упражнениям 396
Указатель кодов операций 407

Производственное издание

КЭПС ЧАРЛЬЗ, СТАФФОРД РИЧАРД

VAX: ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА И АРХИТЕКТУРА

Заведующий редакцией *Ю. Г. Ивашов*
Редакторы *Л. Ю. Камочкина, М. Г. Коробочкина*
Переплет художника *А. С. Дзуцева*
Художественный редактор *А. С. Широков*
Технический редактор *И. Л. Ткаченко*
Корректор *А. К. Акименкова*

ИБ № 1787

Подписано в печать 08. 04. 91 Формат 70×100/16 Бумага тип. № 2 Гарнитура Пресс-роман Печать офсетная
Усл.печ.л. 33,80 Усл.кр.-отт. 34,13 Уч.изд.л. 36,90 Тираж 30 000 экз. Изд. № 22385 Зак. № 1279 Цена 6 руб.

Издательство "Радио и связь", 101000 Москва, Почтамт, а/я 693

Московская типография № 4 Государственного комитета СССР по печати. Москва, И – 41, Б. Переяславская, 46

Книжные магазины — опорные пункты издательства "Радио и связь"

111024, **Москва**, шоссе Энтузиастов, 24/43, магазин №15

197198, **Ленинград** ПС, Большой пр., 34, магазин №55

700070, **Ташкент**, ул.Шота Руставели, 43, магазин №21

226050, **Рига**, бул. Падомью, 17, магазин "Гайсма"

634032, **Томск**, ул.Нахимова, 15/1, магазин №2

503000, **Нижний Новгород**, пр.Гагарина, 110, магазин №9

630091, **Новосибирск**, Красный пр., 60, магазин №7
"Техническая книга"

443090, **Самара**, ул.Советской Армии, 124, магазин №16
"Техническая книга"

173016, **Новгород**, Ленинградская ул., 13, магазин №2
"Прометей"

CHARACTER STRING INSTRUCTIONS (See Section 13.2)

A character string (str) is specified by a starting address (adr) and a 16-bit word that specifies the length in bytes (len)

CMPC3	len,adr1,adr2 R0 R1 R3	CoMPare Characters
CMPC5	len1,adr1,fill,len2,adr2 R0 R1 R2 R3	CoMPare Characters, (pad shorter with fill)
MOVC3	len,adr1,adr2 R0 R1 R3	MOVe Characters
MOVC5	len1,adr1,fill,len2,adr2 R0 R1 R2 R3	MOVe Characters, pad srt1 if necessary
MOVTC	len1,adr1,table,len2,adr2 R0 R1 R3 R4 R5	MOVe Translated Characters (using 256 byte table)
MOVTUC	len1,adr1,byte,table,len2,adr2 R0 R1 R3 R4 R5	MOVe Trans. Until Char. stop if table(chr) EQ byte
LOCC	byte,len,adr R0 R1	LOCate first Character in str equal to byte
SKPC	byte,len,adr R0 R1	SKiP Characters in str equal to byte
SCANC	len,adr,table,mask R0 R1 R3	SCAN Char. (find chr where mask AND table(chr) NE 0)
SPANC	len,adr,table,mask R0 R1 R3	SPAN Char. (find chr where mask AND table(chr) EQ 0)
MATCHC	len1,adr1,len2,adr2 R0 R1 R2 R3	MATCH Characters (find first occurance of str2 in str1)

PACKED DECIMAL INSTRUCTIONS (See Section 13.3)

A decimal string (str) is specified by a starting address (adr) and a length (len) which is the number of decimal digits in the string. Packed decimal instructions use R0 through R3 or R0 through R5.

ARITHMETIC INSTRUCTIONS

MOVP	len,adr1,adr2	str2 ← str1	MOVe
CMPP3	len,adr1,adr2	tmp ← str1 - str2	CoMPare
CMPP4	len1,adr1,len2,adr2	tmp ← str1 - str2	CoMPare
ADDP4	len1,adr1,len2,adr2	str2 ← str2 + str1	ADD
SUBP4	len1,adr1,len2,adr2	str2 ← str2 - str1	SUBtract
ADDP6	len1,adr1,len2,adr2,len3,adr3	str3 ← str2 + str1	
SUBP6	len1,adr1,len2,adr2,len3,adr3	str3 ← str2 - str1	
MULP	len1,adr1,len2,adr2,len3,adr3	str3 ← str2 * str1	
DIVP	len1,adr1,len2,adr2,len3,adr3	str3 ← str2 / str1	
ASHP	cnt,len1,adr1,rnd,len2,adr2	str2 ← str1 * 10**cnt	

PACKED CONVERT INSTRUCTIONS

(to and from Longwords, leading Separate numeric strings, and Trailing numeric strings)

CVTPL	len1,adr1,long	long ← str1	CVT Packed to Long
CVTLP	long,len1,adr1	str1 ← long	CVT Long to Packed
CVTPT	len1,adr1,tbl,len2,adr2	str2 ← str1	CVT Packed to T or S numeric string
CVTPS	len1,adr1,len2,adr2	str2 ← str1	
CVTTP	len1,adr1,tbl,len2,adr2	str2 ← str1	CVT T or S numeric string to Packed
CVTSP	len1,adr1,len2,adr2	str2 ← str1	

QUEUE INSTRUCTIONS (see Section 13.9)

ABSOLUTE QUEUES (pointer is 32 bit absolute address)

INSQUE	entry,head	INSert at head of absolute QUEue
INSQUE	entry,@head+4	INSert at tail of absolute QUEue
REMQUE	entry,head	REMove at head of absolute QUEue
REMQUE	entry,@head+4	REMove at tail of absolute QUEue

RELATIVE QUEUES (pointer is 32 bit displacement)

INSQHI	entry,head	INSert into rel. Queue at Head Interlocked
INSQTI	entry,head	INSert into rel. Queue at Tail Interlocked
REMQHI	entry,head	REMove from rel. Queue at Head Interlocked
REMQTI	entry,head	REMove from rel. Queue at Tail Interlocked

EDIT INSTRUCTIONS (see Section 13.4)

EDIT	len1,adr1,pattern,adr2	str2 ← EDITED(str1) str1 is packed decimal str2 is a character string

EDIT PATTERN OPERATORS

Load fill and sign buffers

EO$LOAD_FILL	char	load the fill register (usually # A" ")
EO$LOAD_SIGN	char	unconditionally load sign register
EO$LOAD_PLUS	char	load sign if str1 GTR 0 (#^A" " or #^A"+")
EO#LOAD_MINUS	char	load sign if str1 LSS 0 (#^A"-")

Output regardless of significance

EO$STORE_SIGN		output the sign character
EO$FILL	cnt	output cnt fill characters (cnt < 16)

Change significance (significance set by non-zero digit in str1)

EO$SET_SIGNIF	set significance
EO$CLEAR_SIGNIF	clear significance

Output digit or character (or fill if significance clear)

EO$INSERT	char	insert character (or fill)
EO$MOVE	cnt	copy digits from str1 to str2 (cnt < 16)

Float sign over insignificant digits (until significance set)

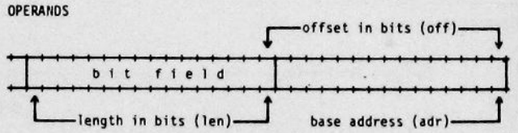
EO$FLOAT	cnt	EO$INSERT, but output sign on significance
EO$END_FLOAT		EO$STORE_SIGN if significance not yet set

Miscellaneous (see VAX Architecture Handbook)

EO$BLANK_ZERO	cnt	Store fill in prev. cnt bytes if str1 EQ 0
EO$REPLACE_SIGN	cnt	Store fill in byte (.-cnt) if str1 EQ 0.
EO$ADJUST_INPUT	cnt	Fudge str1 to length cnt (add or del zeros)
EO$END		MUST terminate string of pattern operators

BIT FIELD INSTRUCTIONS

OPERANDS



The offset (off) is a longword and the length (len) is a byte.
Z in the operation code implies Zero extend (unsigned field).

INSTRUCTIONS.

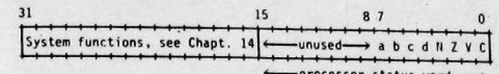
FFC	off,len,adr,long	Find First Clear bit (long gets offset)	
FFS	off,len,adr,long	Find First Set bit (long gets offset)	
EXTV	off,len,adr,long	long ← field	EXTract field
EXTZV	off,len,adr,long	long ← field	EXTract field
INSV	long,off,len,adr	field ← long	INSert field
CMPV	off,len,adr,long	tmp ← field - long	CoMPare field
CMPZV	off,len,adr,long	tmp ← field - long	CoMPare field

STACK OPERATIONS

POPR	word	POP Registers (word is a 16-bit register mask)
PUSHR	word	PUSH Registers (word is a 16 bit register mask)
PUSHL	long	PUSH Longword
PUSHAx	oper	PUSH Address of oper (x = Byte, Word, Long, Quad, Octa, F_flt, D_flt, G_flt, or H_flt)

PROCESSOR STATUS INSTRUCTIONS

PROCESSOR STATUS LONGWORD



a. decimal overflow trap bit
b. floating underflow trap bit
c. integer overflow trap bit
d. trace trap bit

INSTRUCTIONS

BICPSW	mask	BIC mask,PSW	BIt Clear, Processor Status Word
BISPSW	mask	BIS mask,PSW	BIt Set, Processor Status Word
MOVPSL	dst	dst ← PSL	MOVe Processor Status Longword

SYSTEM AND MISCELLANEOUS INSTRUCTIONS

(see Chapter 14 and VAX Architecture Handbook)

ADAWI	add,sum	ADd Aligned Word Interlocked
BBCCI	off,adr,disp	Branch - Bit Clear and Clear Interlocked
BBSSI	off,adr,disp	Branch - Bit Set and Set Interlocked
BPT		Break Point faulT (for debugging)
BUGx		BUGcheck (x = Word or Long)
CHMx	param	CHange Mode (x = Kernel, Executive, Supervisor, or User)
ESCx		ESCape (x = D, E, or F)
HALT		HALT processor
LDPCTX		LoaD Process ConTeXt
MFPR	src,reg	Move From special Processor Register
MTPR	reg,dst	Move To special Processor Register
NOP		No Operation
PROBER	mode,len,base	PROBE Read accessibility in mode
PROBEW	mode,len,base	PROBE Write accessibility in mode
REI		Return from Exception or Interrupt
SVPCTX		SaVe Process ConTeXt
XFC		Extended Function Call

POWERS OF 2 and 16

16**n	n	2**n
1	0	1
16	1	2
256	2	4
4,096	3	8
65,536	4	16
1,048,576	5	32
16,777,216	6	64
268,435,456	7	128
4,294,967,296	8	256
68,719,476,736	9	512
1,099,511,627,776	10	1,024
17,592,186,044,416	11	2,048
281,474,976,710,656	12	4,096
4,503,599,627,370,496	13	8,192
72,057,594,037,927,936	14	16,384
1,152,921,504,606,846,976	15	32,768
18,446,744,073,709,551,616	16	65,536

ADDRESSING MODES SELECTED BY THE ASSEMBLER

Name	Assembler	Description
immediate	#constant	literal or immediate
displacement	DISP(Rx)	byte, word, or long displacement
disp. deferred	@DISP(Rx)	byte, word, or long displacement deferred
relative	address	byte, word, or long relative addressing
rel. deferred	@address	byte, word, or long relative deferred

7-BIT ASCII CODE

Hex Code	ASCII Char	Hex Code	ASCII Char	Hex Code	ASCII Char	Hex Code	ASCII Char
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

GENERAL ADDRESSING MODES

(Rx is a general register R0 to R15)
(In descriptions, (Rx) means the contents of register Rx)

Hex	Dec	Name	Assembler	Description
0-3	0-3	literal	S^#literal	literal is operand
4	4	indexed	mode[Rx]	adr(mode)+n*(Rx) is adr
5	5	register	Rx	(Rx) is operand
6	6	reg deferred	(Rx)	(Rx) is address of oper
7	7	autodecrement	-(Rx)	dec Rx, Rx contains adr
8	8	autoincrement	(Rx)+	Rx contains adr, inc Rx
9	9	autoinc. def.	@(Rx)+	Rx contains adr of adr
A	10	byte disp.	B^DISP(Rx)	DISP+(Rx) is adr
B	11	byte disp. def.	@B^DISP(Rx)	DISP+(Rx) is adr of adr
C	12	word disp.	W^DISP(Rx)	DISP+(Rx) is adr
D	13	word disp. def.	@W^DISP(Rx)	DISP+(Rx) is adr of adr
E	14	long disp.	L^DISP(Rx)	DISP+(Rx) is adr
F	15	long disp. def.	@L^DISP(Rx)	DISP+(Rx) is adr of adr

PROGRAM COUNTER ADDRESSING MODES (Rx is PC or Register 15)

Hex	Dec	Name	Assembler	Description
8	8	immediate	I^#constant	constant is operand
9	9	absolute	@#address	address is adr of oper
A	10	byte relative	B^address	address is adr of oper
B	11	byte rel. def.	@B^address	address is adr of adr
C	12	word relative	W^address	address is adr of oper
D	13	word rel. def.	@W^address	address is adr of adr
E	14	long relative	L^address	address is adr of oper
F	15	long rel. def.	@L^address	address is adr of adr