



НОВЫЕ РАЗДЕЛЫ КЛАССИЧЕСКОГО ТРУДА

Искусство программирования

ТОМ 1

MMIX

RISC-компьютер
для нового
тысячелетия

ВЫПУСК

1

ДОНАЛЬД Э. КНУТ

ASCII СИМВОЛЫ

	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	
#2x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	#2x
#3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	#3x
#4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	#4x
#5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	#5x
#6x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	#6x
#7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	␣	#7x
	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	

MMIX КОДЫ ОПЕРАЦИЙ (ОПКОДЫ)

	#0	#1	#2	#3	#4	#5	#6	#7	
#0x	TRAP $5v$	FCMP v	FUN v	FEQL v	FADD $4v$	FIX $4v$	FSUB $4v$	FIXU $4v$	#0x
	FLOT[I] $4v$		FLOTU[I] $4v$		SFLOT[I] $4v$		SFLOTU[I] $4v$		
#1x	FMUL $4v$	FCMPE $4v$	FUNE v	FEQLE $4v$	FDIV $40v$	FSQRT $40v$	FREM $4v$	FINT $4v$	#1x
	MUL[I] $10v$		MULU[I] $10v$		DIV[I] $60v$		DIVU[I] $60v$		
#2x	ADD[I] v		ADDU[I] v		SUB[I] v		SUBU[I] v		#2x
	2ADDU[I] v		4ADDU[I] v		8ADDU[I] v		16ADDU[I] v		
#3x	CMP[I] v		CMPU[I] v		NEG[I] v		NEGU[I] v		#3x
	SL[I] v		SLU[I] v		SR[I] v		SRU[I] v		
#4x	BN[B] $v+\pi$		BZ[B] $v+\pi$		BP[B] $v+\pi$		BOD[B] $v+\pi$		#4x
	BNN[B] $v+\pi$		BNZ[B] $v+\pi$		BNP[B] $v+\pi$		BEV[B] $v+\pi$		
#5x	PBN[B] $3v-\pi$		PBZ[B] $3v-\pi$		PBP[B] $3v-\pi$		PBOD[B] $3v-\pi$		#5x
	PBNN[B] $3v-\pi$		PBNZ[B] $3v-\pi$		PBNP[B] $3v-\pi$		PBEV[B] $3v-\pi$		
#6x	CSN[I] v		CSZ[I] v		CSP[I] v		CSOD[I] v		#6x
	CSNN[I] v		CSNZ[I] v		CSNP[I] v		CSEV[I] v		
#7x	ZSN[I] v		ZSZ[I] v		ZSP[I] v		ZSOD[I] v		#7x
	ZSNN[I] v		ZSNZ[I] v		ZSNP[I] v		ZSEV[I] v		
#8x	LDB[I] $\mu+v$		LDBU[I] $\mu+v$		LDW[I] $\mu+v$		LDWU[I] $\mu+v$		#8x
	LDT[I] $\mu+v$		LDTU[I] $\mu+v$		LDO[I] $\mu+v$		LDOU[I] $\mu+v$		
#9x	LDSF[I] $\mu+v$		LDHT[I] $\mu+v$		CSWAP[I] $2\mu+2v$		LDUNC[I] $\mu+v$		#9x
	LDVTS[I] v		PRELD[I] v		PREGO[I] v		GO[I] $3v$		
#Ax	STB[I] $\mu+v$		STBU[I] $\mu+v$		STW[I] $\mu+v$		STWU[I] $\mu+v$		#Ax
	STT[I] $\mu+v$		STTU[I] $\mu+v$		STO[I] $\mu+v$		STOU[I] $\mu+v$		
#Bx	STSF[I] $\mu+v$		STHT[I] $\mu+v$		STCO[I] $\mu+v$		STUNC[I] $\mu+v$		#Bx
	SYNCD[I] v		PREST[I] v		SYNCID[I] v		PUSHGO[I] $3v$		
#Cx	OR[I] v		ORN[I] v		NOR[I] v		XOR[I] v		#Cx
	AND[I] v		ANDN[I] v		NAND[I] v		NXOR[I] v		
#Dx	BDIF[I] v		WDIF[I] v		TDIF[I] v		ODIF[I] v		#Dx
	MUX[I] v		SADD[I] v		MOR[I] v		MXOR[I] v		
#Ex	SETH v	SETMH v	SETML v	SETL v	INCH v	INCMH v	INCML v	INCL v	#Ex
	ORH v	ORMH v	ORML v	ORL v	ANDNH v	ANDNMH v	ANDNML v	ANDNL v	
#Fx	JMP[B] v		PUSHJ[B] v		GETA[B] v		PUT[I] v		#Fx
	POP $3v$	RESUME $5v$	[UN]SAVE $20\mu+v$		SYNC v	SWYM v	GET v	TRIP $5v$	
	#8	#9	#A	#B	#C	#D	#E	#F	

$\pi = 2v$, если условный переход выполняется; $\pi = 0$, если условный переход не выполняется.

THE ART OF COMPUTER PROGRAMMING

VOLUME 1, FASCICLE 1

MMIX A RISC Computer for the New Millennium

DONALD E. KNUTH *Stanford University*



ADDISON-WESLEY

Upper Saddle River, NJ · Boston · Indianapolis · San Francisco
New York · Toronto · Montréal · London · Munich · Paris · Madrid
Capetown · Sydney · Tokyo · Singapore · Mexico City

ИСКУССТВО ПРОГРАММИРОВАНИЯ

ТОМ 1, ВЫПУСК 1

MMIX

RISC-компьютер для нового тысячелетия

ДОНАЛЬД Э. КНУТ *Станфордский университет*



Москва · Санкт-Петербург · Киев
2007

ББК 32.973.26–018.2.75

К53

УДК 681.142.2

Издательский дом “Вильямс”

Зав. редакцией *С. Н. Тригуб*

Перевод с английского и редакция канд. физ.-мат. наук *Ю.Г. Гордиенко*

По общим вопросам обращайтесь в Издательский дом “Вильямс”
по адресу: info@williamspublishing.com, <http://www.williamspublishing.com>
115419, Москва, а/я 783; 03150, Киев, а/я 152

Кнут, Дональд, Эрвин

Искусство программирования, том 1, выпуск 1. MMIX — RISC-компьютер нового тысячелетия. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2007. — 160 с. : ил. — Парал. тит. англ.

ISBN 978–5–8459–1163–6 (рус.)

Эта книга представляет собой один из выпусков очередных томов всемирно известного труда Искусство программирования, не нуждающейся ни в представлении, ни в рекламе. В данный выпуск вошли разделы первого тома, посвященные RISC-компьютеру MMIX, который заменит прежний компьютер MIX, и языка ассемблера MMIX. Материалы этого выпуска в будущем войдут в первый том серии, посвященный базовым алгоритмам — возможно, с определенными дополнениями и исправлениями на основе отзывов читателей данного выпуска.

ББК 32.973.26–018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Rights to this book were obtained by arrangement with Addison-Wesley Longman, Inc.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2007

ISBN 978–5–8459–1163–6 (рус.)
ISBN 0–201–85392–2 (англ.)

© Издательский дом “Вильямс”, 2007
© by Pearson Education, Inc., 2005

СОДЕРЖАНИЕ

Глава 1. Основные понятия	10
1.3'. ММІХ	10
1.3.1'. Описание ММІХ	10
1.3.2'. Язык ассемблера компьютера ММІХ	39
1.3.3'. Применения к перестановкам	63
1.4'. Некоторые фундаментальные методы программирования	64
1.4.1'. Подпрограммы	64
1.4.2'. Сопрограммы	79
1.4.3'. Программы-интерпретаторы	86
Ответы к упражнениям	109
Предметно-именной указатель	143

ОТ ИЗДАТЕЛЬСТВА

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail:	info@williamspublishing.com
WWW:	http://www.williamspublishing.com

Информация для писем из:

России:	115419, Москва, а/я 783
Украины:	03150, Киев, а/я 152

ПРЕДИСЛОВИЕ

fas·ci·cle \ˈfasəˌkəl\ n ... 1: маленький пучок соцветия, состоящий из компактного зонтика и образующий маленькую головку
... 2: отдельный выпуск какого-либо издания, публикуемого по частям
— П. Б. ГОУВ (P. B. GOVE), Вебстерский словарь международного английского языка, Третье издание (1961)

ДАННАЯ КНИГА является первым из серии отдельных выпусков-дополнений, которые я планирую регулярно выпускать по мере завершения работы над окончательной редакцией книги *Искусство программирования*.

К работе над дополнениями меня вдохновил пример Чарльза Диккенса, который публиковал свои произведения в виде серий отдельных глав. Он напечатал десятки глав романа *Приключения Оливера Твиста* (*The Adventures of Oliver Twist*), даже не подозревая о том, что же случится с Биллом Сайксом! Меня также вдохновляет Джеймс Марри, который начал публикацию первых 350-страничных порций Оксфордского словаря английского языка в 1884 году, в 1888 году завершил работу над буквой В, а в 1888 году над буквой С. (Марри умер в 1915 году, работая над буквой Т. К счастью, моя задача гораздо проще.)

В отличие от Диккенса и Марри, для редактирования материала я использую компьютер, а потому могу легко вносить изменения в текст до получения его окончательного вида. Хотя я изо всех сил стараюсь писать так, чтобы потом не приходилось вносить исправления, но каждая страница содержит сотни потенциальных возможностей для совершения ошибок и упущения важных идей. Мои файлы заполнены заметками о новых прекрасных алгоритмах, но информатика достигла такого уровня, при котором я не могу претендовать на полный охват всего имеющегося материала. Именно поэтому мне необходимо получить отклики от читателей еще до завершения работы над всеми томами.

Иначе говоря, дополнения будут содержать достаточно Достойного Материала, и я с волнением предлагают все, что я хотел написать, тем, кто хотел бы это прочесть. Все же я надеюсь, что внимательные читатели помогут мне улучшить их. Как обычно, за каждую опечатку*, политически некорректное высказывание, а также ошибку, относящуюся к сути излагаемого материала или к приведенным историческим сведениям, я охотно заплачу \$2,56 тому, кто первым ее найдет.

Чарльз Диккенс обычно публиковал свои очередные главы ежемесячно, а Джеймс Марри стремился издавать 350-страничные выпуски каждые полтора года. Моя цель, будь на то воля Божья, ежегодно создавать по два 129-страничных отдельных выпуска. Большинство выпусков будет представлять новый материал для тома 4 и далее. Но иногда я буду выпускать дополнения к прежним томам. Например, в томе 4 потребуется обратиться к материалу из тома 3, который еще

* Имеется в виду оригинал настоящего издания. — *Примеч. ред.*

не существовал в момент выпуска тома 3. По мере выхода этих выпусков вся эта работа, в конечном итоге, приобретет законченный осмысленный вид.

Выпуск 1 посвящен ММIX, т.е. долгожданной замене MIX. Спустя 37 лет после создания компьютера MIX архитектура компьютеров претерпела значительные изменения и привела к появлению компьютеров нового типа. Именно поэтому я еще менее насыщенным “жирами”, чем его полиненасыщенный предшественник*.

В упражнениях 1.3.1–25 в первых трех изданиях тома 1 используется расширение MixMaster языка MIX, обратно совместимое с прежними версиями языка MIX. Но даже расширение MixMaster давно и безнадежно устарело. Оно способно работать с несколькими гигабайтами памяти, но не может обрабатывать код ASCII, например, для вывода символов в нижнем регистре. Более того, стандартные соглашения для вызова подпрограмм были основаны на самоизменяющихся инструкциях! Десятичная арифметика и самоизменяющийся код были очень популярны в 1962 году, но они быстро исчезли по мере увеличения размеров и скорости вычисления компьютеров. К счастью, современная архитектура RISC имеет более привлекательную структуру, а потому я воспользовался возможностью создать новый компьютер, который отвечает современным требованиям и с которым просто приятно работать.

Многие читатели могут спросить: “Зачем Кнут заменил MIX другим компьютером, а не перешел к языку программирования высокого уровня, ведь в наши дни мало кто использует языки ассемблера в своей работе?” Они, несомненно, имеют право на такую точку зрения и могут пропустить ту часть книги, в которой описывается машинный язык. Однако причины, по которым я создал описание машинного языка, были перечислены еще в 1960-х годах в предисловии к тому 1 и до сих пор остаются в силе.

- Одна из основных целей данной книги — показать, как конструкции высокого уровня реализуются, а не применяются в компьютерах. Таким образом, с самого начала рассматриваются связь сопрограмм, древовидные иерархические структуры, генерирование случайных чисел, арифметика высокой точности, преобразование системы счисления, упаковка данных, комбинаторный поиск, рекурсия и многие другие проблемы.
- Все нужные нам программы, написанные на машинно-ориентированном языке, за редким исключением будут иметь небольшой размер, чтобы можно было легко уловить их суть.
- Программист должен иметь, по крайней мере, представление об архитектуре компьютера, ибо в противном случае созданные им программы могут быть чрезвычайно неэффективными и запутанными.
- Некоторое знание машинного языка необходимо в любом случае, чтобы разобратся в выходных данных программ, приведенных во многих примерах.
- Выражая на машинном языке базовые алгоритмы, например для сортировки и поиска, можно изучать влияние кэш-памяти, ОЗУ и других параметров аппаратного обеспечения (скорости оперативной памяти, конвейеризации, множественной выдачи команд, буфер ассоциативной трансляции, размеров блоков в кэш-памяти и многих других) при сравнении разных схем.

* Аналогия с полиненасыщенными жирами, широко рекламируемыми сегодня по всему миру. — *Примеч. ред.*

Более того, если бы я решился использовать язык высокого уровня, то какой из них следовало бы выбрать? В 1960-х годах я, вероятно, выбрал бы Algol W, в 1970-х годах я был бы вынужден переписать мою книгу на языке Pascal, в 1980-х годах все пришлось бы изменить для языка C, а в 1990-х годах надо было бы переключиться на C++ или Java.

Вне всяких сомнений, в 2000-х годах еще какой-то язык станет *de rigueur**. Я не могу позволить себе тратить время на переписывание книги в угоду моде, поскольку не сам язык программирования является сутью книги, а что можно сделать с его помощью, например с помощью вашего любимого языка программирования. Основное внимание в книге уделяется “вечным истинам”.

Следовательно, в книге *Искусство программирования* в качестве языка высокого уровня я использую английский**, а язык низкого уровня будет использоваться для описания механизмов работы компьютеров. Читателям, которых интересуют алгоритмы, уже реализованные в виде готовых подключаемых модулей на основе ультрамодных языков программирования, рекомендуется обратиться к другим книгам.

Хотел бы успокоить читателей: программирование для MMIX представляет собой приятное и простое занятие. В этом выпуске представлены следующие темы:

- 1) вводное описание архитектуры MMIX (заменяет раздел 1.3.1 третьего издания тома 1);
- 2) описание языка ассемблера MMIX (заменяет раздел 1.3.2);
- 3) новые сведения о подпрограммах, сопрограммах и программах-интерпретаторах (заменяет разделы 1.4.1, 1.4.2 и 1.4.3).

Конечно, компьютер MIX присутствует во многих местах существующих изданий томов 1-3, и десятки программ нужно переписать для использования их вместе с компьютером MMIX в новых изданиях этих томов. Читатели, которые хотели бы принять участие в этом процессе переписывания, могут присоединиться к группе добровольцев-создателей MMIX, обращаясь к Web-узлу по адресу mmixmasters.sourceforge.net.

Четвертое издание тома 1 не будет готово до тех пор, пока не будет завершена работа над томами 4 и 5. Поэтому две разные версии разделов 1.3.1, 1.3.2, 1.4.1, 1.4.2 и 1.4.3 будут сосуществовать в течение нескольких лет. Во избежание путаницы, новые версии этих разделов будут иметь нумерацию “простых чисел” со штрихами: 1.3.1', 1.3.2', 1.4.1', 1.4.2' и 1.4.3'.

Я очень благодарен всем людям, которые помогли мне создать компьютер MMIX. Особую благодарность заслуживают Джон Хэннэси (John Hennessy) и Ричард Л. Сайтс (Richard L. Sites) за их активное участие и существенный вклад. Благодарю Владимира Ивановича (Vladimir Ivanović) за добровольную помощь в поддержке Web-узла группы добровольцев-создателей MMIX.

Станфорд, Калифорния
май 1999

Д. Э. К.

Переписывать можно вечно, если нужно.

— НЭЙЛ САЙМОН (NEIL SIMON), *Перезапись: мемуары* (1996)

* Модным. — *Примеч. ред.*

** Имеется в виду оригинал настоящего издания. — *Примеч. ред.*

1.3'. MMIX

В ЭТОЙ КНИГЕ очень часто встречаются упоминания о внутреннем машинном языке компьютера. Причем использовать мы будем гипотетический компьютер под названием “MMIX”. MMIX — произносится *эм-микс* — практически ничем не отличается от любого другого компьютера с 1985 года, за исключением того, что он, вероятно, более изящен. При разработке языка для компьютера MMIX преследовалась цель сделать его достаточно мощным, позволяющим для большинства алгоритмов писать короткие программы, и в то же время достаточно простым, чтобы его операции можно было легко запомнить.

Настоятельно рекомендую читателю внимательно изучить этот раздел, так как язык для компьютера MMIX используется в очень многих разделах книги. Отбросьте все сомнения по поводу того, стоит ли изучать машинный язык. Автор однажды понял, что нет ничего необычного в том, чтобы в течение одной недели заниматься написанием программ на нескольких различных машинных языках! Каждый, кто серьезно интересуется компьютерами, должен рано или поздно изучить по крайней мере один машинный язык. Машинный язык помогает программисту понять, что фактически происходит внутри компьютеров. Зная основы одного машинного языка, можно легко освоить характеристики любого другого. Информатика во многом связана с механизмами достижения целей высокого уровня на основе работы с компонентами низкого уровня.

Программное обеспечение для MMIX практически для любого реального компьютера можно загрузить с Web-узла этой книги (см. ii). Весь авторский исходный код для MMIX приводится в книге *MMIXware [Lecture Notes in Computer Science 1750 (1999)]*. Далее эта книга называется “документация MMIXware”.

1.3.1'. Описание MMIX

MMIX — это полиненасыщенный и на 100% натуральный органический компьютер*. Как и у большинства компьютеров, у него есть идентификационный номер — 2009. Этот номер получен следующим образом: взяли 14 очень похожих на MMIX реальных компьютеров, на которых можно легко имитировать MMIX, а затем нашли среднее значение их номеров, взятых с равными весовыми коэффициентами:

$$\begin{aligned} & (\text{Cray I} + \text{IBM 801} + \text{RISC II} + \text{Clipper C300} + \text{AMD 29K} + \text{Motorola 88K} \\ & \quad + \text{IBM 601} + \text{Intel i960} + \text{Alpha 21164} + \text{POWER 2} + \text{MIPS R4000} \\ & \quad + \text{Hitachi SuperH4} + \text{StrongARM 110} + \text{Sparc 64}) / 14 \\ & = 28126 / 14 = 2009. \end{aligned} \tag{1}$$

Это же число можно получить значительно проще — прочитать слово MMIX как римское число.

Биты и байты. MMIX работает с последовательностями нулей и единиц, которые называются двоичными цифрами, или *битами*. Обычно MMIX обрабатывает сразу по 64 бита. Вот пример типичной последовательности 64 битов.

$$10011110001101110111100110111001011111010010100111110000010110 \tag{2}$$

* Аналогия с полиненасыщенными жирами, широко рекламируемыми сегодня по всему миру. — *Примеч. ред.*

Такие длинные последовательности удобнее изображать в виде групп по 4 бита в каждой группе и использовать *шестнадцатеричные цифры* для представления каждой группы, как показано ниже.

$$\begin{array}{llll} 0 = 0000, & 4 = 0100, & 8 = 1000, & c = 1100, \\ 1 = 0001, & 5 = 0101, & 9 = 1001, & d = 1101, \\ 2 = 0010, & 6 = 0110, & a = 1010, & e = 1110, \\ 3 = 0011, & 7 = 0111, & b = 1011, & f = 1111. \end{array} \quad (3)$$

Чтобы не путать шестнадцатеричные цифры с десятичными цифрами 0–9, перед шестнадцатеричным числом всегда будет приводиться символ #. Например, число (2) в шестнадцатеричном представлении будет иметь следующий вид:

$$\#9e3779b97f4a7c16 \quad (4)$$

в шестнадцатеричной системе счисления. Цифры ABCDEF обычно приводятся в верхнем регистре, а не в нижнем abcdef, поскольку #9E3779B97F4A7C16 выглядит лучше, чем #9e3779b97f4a7c16 в определенном контексте, хотя между ними нет никакой разницы.

Последовательность из 8 бит или 2 шестнадцатеричных цифр называется *байтом*. В большинстве современных компьютеров байты используются, как отдельно адресуемые единицы данных. В программе компьютера MMIX можно адресовать до 2^{64} байт, каждый из которых имеет собственный адрес от #0000000000000000 до #ffffffffffffffff. Буквы, цифры и знаки пунктуации (например, английского языка) часто представляются одним байтом для каждого символа на основе Американского стандартного кода обмена информацией (American standard code for information interchange — ASCII). Например, ASCII-код для имени компьютера MMIX имеет вид #4d4d4958. ASCII-кодировка основана на коде из 7 битов, которые могут включать контрольные символы #00–#1f, символы печати #20–#7e и символ удаления #7f [see CACM 8 (1965), 207–214; 11 (1968), 849–852; 12 (1969), 166–178]. Эта 7-битовая кодировка в 1980-х годах была расширена до международной стандартной 8-битовой кодировки, которая называется Latin-1 или ISO 8859-1 и позволяет кодировать буквы со специальными знаками, например слову *pâté* соответствует код #70e274e9.

“Из 256-й эскадрильи?”

“Да, из 256-й боевой эскадрильи,” — ответил Йоссариан.

... “Это два в боевой восьмой степени.”

— ДЖОЗЕФ ХЕЛЛЕР (JOSEPH HELLER), *Уловка-22* (1961)

16-битовая кодировка, которая используется практически в *каждом* современном языке программирования, стала международным стандартом в 1990-х годах. Эта кодировка официально называется ISO/IEC 10646 UCS-2, а неформально — Unicode. Она позволяет кодировать буквы греческого алфавита Σ и σ (#03a3 и #03c3)), буквы русского алфавита Щ и щ (#0429 и #0449), буквы армянского алфавита Տ и տ (#0547 и #0577), буквы иврита װ (#05e9), буквы арабского алфавита ش (#0634), и буквы индийского алфавита ञ (#0936), или ण (#09b6), или ण

(#0b36), или ω (#0bb7), а также десятки тысяч иероглифов, например китайских, 算 (#7b97). В этой кодировке предусмотрены специальные коды даже для римских цифр: MMIX = #216f216f21602169. Символы кодировки ASCII или Latin-1 представлены с первым нулевым байтом, например слову *pâté* соответствует код #007000e2007400e9, à l'Unicode.

Поскольку 2-байтовые величины имеют большое практическое значение, то для обозначения таких 16-битовых величин, как, например, символы кодировки Unicode, мы будем использовать более удобный термин *вайд*. Нам также потребуются более удобные термины *тетрабайт* (или просто “тетра”) и *октабайт* (или просто “окта”) для обозначения 4-байтовых и 8-байтовых величин. Таким образом, 1 октабайт содержит 4 вайда или 8 байт или 64 бита.

2 байта = 1 вайд;
2 вайда = 1 тетра;
2 тетра = 1 окта.

Конечно, байты и другие перечисленные выше величины могут представлять не только буквы, но и цифры. Например, на основе описанной двоичной системы:

беззнаковый байт может выражать числа 0 .. 255;
беззнаковый вайд может выражать числа 0 .. 65535;
беззнаковый тетрабайт может выражать числа 0 .. 4294967295;
беззнаковый октабайт может выражать числа 0 .. 18446744073709551615.

Целые числа обычно представляются с помощью *арифметики двоичного дополнения*, в которой самый левый бит обозначает знак: если он равен 1, то для получения n -битового представления в этой арифметике нужно отнять 2^n . Например, -1 соответствует знаковому байту #ff; знаковому вайду #ffff, знаковому тетрабайту #fffffffff, знаковому октабайту #fffffffffffffffff. Таким образом:

знаковый байт может выражать числа $-128 \dots 127$;
знаковый вайд может выражать числа $-32768 \dots 32,767$;
знаковый тетрабайт может выражать числа $-2147483648 \dots 2147483647$;
знаковый октабайт может выражать числа $-9223372036854775808 \dots 9223372036854775807$.

Память и регистры. С точки зрения программиста, в компьютере MMIX имеется 2^{64} ячеек *памяти*, 2^8 *регистров* общего назначения и 2^5 специальных регистра (рис. 13). Данные передаются из памяти в регистры, преобразуются в регистрах и передаются из регистров в память. Ячейки памяти обозначаются $M[0]$, $M[1]$, ..., $M[2^{64} - 1]$. Если x — октабайт, то $M[x]$ — это байт памяти. Регистры общего назначения обозначаются $\$0$, $\$1$, ..., $\$255$. Если x — байт, то $\$x$ — это октабайт.

2^{64} байт памяти сгруппированы в 2^{63} вайда: $M_2[0] = M_2[1] = M[0]M[1]$, $M_2[2] = M_2[3] = M[2]M[3]$, ...; каждый вайд состоит из двух последовательных байтов $M[2k]M[2k + 1] = M[2k] \times 2^8 + M[2k + 1]$ и обозначается $M_2[2k]$ или $M_2[2k + 1]$. Аналогично, 2^{64} байт памяти сгруппированы в 2^{62} тетрабайта

$$M_4[4k] = M_4[4k + 1] = \dots = M_4[4k + 3] = M[4k]M[4k + 1] \dots M[4k + 3],$$

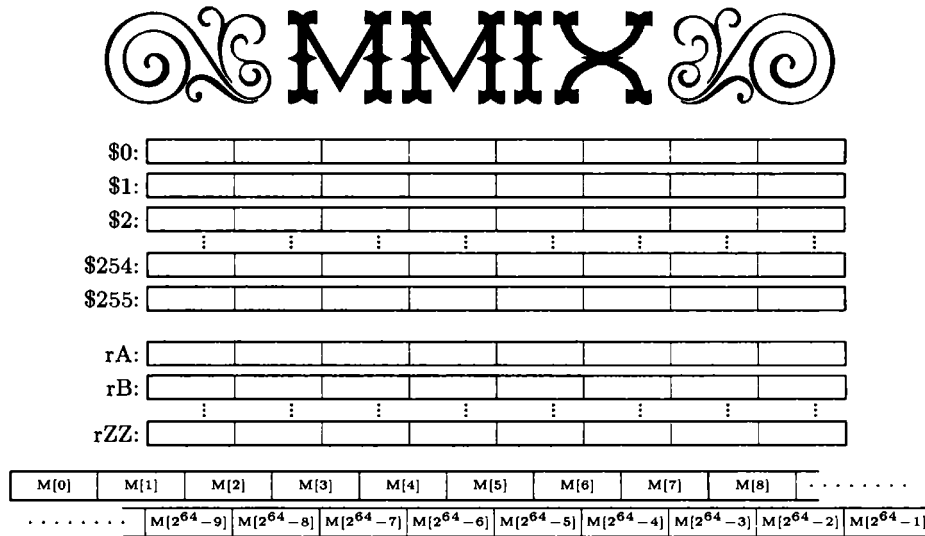


Рис. 13. Компьютер MMIX, с точки зрения программиста, имеет 256 регистров общего назначения, 32 регистра специального назначения и имеет виртуальную память 2^{64} байт. Каждый регистр содержит 64 бита данных.

и 2^{61} октабайта

$$M_8[8k] = M_8[8k + 1] = \dots = M_8[8k + 7] = M[8k]M[8k + 1] \dots M[8k + 7].$$

Вообще, если x — октабайт, то $M_2[x]$, $M_4[x]$ и $M_8[x]$ обозначают вайд, тетра и окта, которые содержат байт $M[x]$. При ссылке на $M_t[x]$ игнорируется $\lg t$ наименее значимых битов x . Для полноты запишем $M_1[x] = M[x]$ и определим $M[x] = M[x \bmod 2^{64}]$, если $x < 0$ или $x \geq 2^{64}$.

32 специальных регистра MMIX называются rA, rB, ..., rZ, rBB, rTT, rWW, rXX, rYY и rZZ. Как и регистры общего назначения, каждый из них содержит октабайт. Их применение описывается ниже. Например, регистр rA управляет арифметическими прерываниями, а регистр rR содержит остаток от деления.

Инструкции. В памяти MMIX хранятся не только данные, но и инструкции. *Инструкция*, или “команда”, — это тетрабайт, состоящий из четырех байтов OP, X, Y, и Z. Байт OP содержит *код операции* (или кратко “опкод”); а байты X, Y и Z — *операнды*. Например, #20010203 является инструкцией с байтами OP = #20, X = #01, Y = #02 и Z = #03, которая означает “Установить \$1 в значение суммы \$2 и \$3”. Байты операндов всегда рассматриваются как беззнаковые целые числа.

Каждый из 256 возможных опкодов имеет легко запоминаемое символьное представление. Например, опкод #20 записывается как ADD. Мы будем иметь дело исключительно с символьными опкодами, а их числовые эквиваленты в случае необходимости можно найти в табл. 1, а также в конце этой книги.

Байты X, Y и Z также имеют символьное представление, совместимое с языком ассемблера, который рассматривается в разделе 1.3.2'. Например, инструкция сложения #20010203 обычно записывается ‘ADD \$1,\$2,\$3’, а в более общем виде — ‘ADD \$X,\$Y,\$Z’. Большинство инструкций включает три операнда, некоторые — только два, а очень немногие — только один операнд. При наличии двух операндов первым операндом является X, вторым — двухбайтовая величина YZ, а символьное представление содержит только одну запятую. Например, инструкция

'INCL \$X,YZ' увеличивает регистр \$X на величину YZ. При наличии только одного операнда он имеет вид беззнаковой трехбайтовой величины XYZ, а символьное представление не имеет запятой. Например, инструкция в символьном представлении 'JMP @+4*XYZ' приказывает компьютеру MMIX найти следующую инструкцию, пропустив XYZ тетрабайт. Инструкция 'JMP @+1000000' имеет шестнадцатеричное представление #f003d090, поскольку $\text{JMP} = \text{\#f0}$ и $250000 = \text{\#03d090}$.

Далее для каждой инструкции MMIX дается формальное определение и неформальное описание. Например, неформальный смысл инструкции 'ADD \$X,\$Y,\$Z' заключается в следующем: "Установить \$X в значение суммы \$Y и \$Z", а ее формальное определение гласит: ' $s(\$X) \leftarrow s(\$Y) + s(\$Z)$ '. Здесь $s(x)$ обозначает *знаковое целое число*, соответствующее битовой комбинации x , согласно соглашениям двоичного дополнения. Присваивание $s(x) \leftarrow N$ означает, что x устанавливается в битовую комбинацию для $s(x) = N$. (Такое присваивание приводит к *переполнению*, если N принимает очень большое или очень малое значение, чтобы его можно было представить в x . Например, инструкция ADD приведет к переполнению, если $s(\$Y) + s(\$Z)$ меньше -2^{63} или больше $2^{63} - 1$. При неформальном обсуждении инструкций мы часто будем опускать возможность переполнения, но в формальном определении все должно быть предельно точно. Вообще, присваивание $s(x) \leftarrow N$ устанавливает x в двоичное представление $N \bmod 2^n$, где n — количество битов в x , и приводит к переполнению, если $N < -2^{n-1}$ или $N \geq 2^{n-1}$; подробнее см. упражнение 5.)

Загрузка и сохранение. Хотя MMIX имеет 256 разных опкодов, их можно классифицировать на несколько легко запоминаемых категорий. Начнем с инструкций, которые передают информацию между регистрами и памятью.

Каждая из следующих инструкций содержит *адрес ячейки памяти* A, полученный в результате сложения \$Y и \$Z. Строго говоря,

$$A = (u(\$Y) + u(\$Z)) \bmod 2^{64} \quad (5)$$

является суммой *беззнаковых целых чисел*, представленных \$Y и \$Z, приведенной к 64-битовому числу с игнорированием выхода за левую границу при сложении этих двух целых чисел. В этой формуле обозначение $u(x)$ аналогично $s(x)$, но x считается беззнаковым двоичным числом.

- LDB \$X,\$Y,\$Z (загрузить байт): $s(\$X) \leftarrow s(M_1[A])$.
- LDW \$X,\$Y,\$Z (загрузить вайд): $s(\$X) \leftarrow s(M_2[A])$.
- LDT \$X,\$Y,\$Z (загрузить тетра): $s(\$X) \leftarrow s(M_4[A])$.
- LDO \$X,\$Y,\$Z (загрузить окта): $s(\$X) \leftarrow s(M_8[A])$.

Эти инструкции заносят данные из памяти в регистр \$X, изменяя данные, в случае необходимости, от знакового байта, вайда или тетрабайта в знаковый октабайт той же величины. Предположим, что октабайт $M_8[1002] = M_8[1000]$ равен

$$M[1000]M[1001] \dots M[1007] = \text{\#0123456789abcdef}. \quad (6)$$

Затем, если $\$2 = 1000$ и $\$3 = 2$, то $A = 1002$ и

```
LDB $1,$2,$3 sets $1 ← #0000 0000 0000 0045;
LDW $1,$2,$3 sets $1 ← #0000 0000 0000 4567;
LDT $1,$2,$3 sets $1 ← #0000 0000 0123 4567;
LD0 $1,$2,$3 sets $1 ← #0123 4567 89ab cdef.
```

Но если $\$3 = 5$, то $A = 1005$,

```
LDB $1,$2,$3 sets $1 ← #ffffff ffff fab;
LDW $1,$2,$3 sets $1 ← #ffffff ffff 89ab;
LDT $1,$2,$3 sets $1 ← #ffffff 89ab cdef;
LD0 $1,$2,$3 sets $1 ← #0123 4567 89ab cdef.
```

Если знаковый байт, вайд или тетра преобразуется в знаковый окта, то знаковый бит “расширяется” на все позиции слева.

- LDBU $\$X, \$Y, \$Z$ (загрузить беззнаковый байт): $u(\$X) \leftarrow u(M_1[A])$.
- LDWU $\$X, \$Y, \$Z$ (загрузить беззнаковый вайд): $u(\$X) \leftarrow u(M_2[A])$.
- LDTU $\$X, \$Y, \$Z$ (загрузить беззнаковый тетра): $u(\$X) \leftarrow u(M_4[A])$.
- LDOU $\$X, \$Y, \$Z$ (загрузить беззнаковый окта): $u(\$X) \leftarrow u(M_8[A])$.

Эти инструкции аналогичны инструкциям LDB, LDW, LDT, и LD0, но они воспринимают данные из памяти, как *беззнаковые*, а в регистре битовые позиции слева устанавливаются в 0, если нужно “удлинить” короткое значение. Так, в приведенном выше примере, LDBU $\$1, \$2, \$3$ с $\$2 + \$3 = 1005$ примет вид $\$1 \leftarrow \#0000\ 0000\ 0000\ 00ab$.

Инструкции LD0 и LDOU выполняются абсолютно одинаково, поскольку расширение знака или дополнение нулями не нужно при загрузке октабайта в регистр. Но профессиональный программист всегда использует LD0 при работе со знаковыми и LDOU при работе с беззнаковыми значениями. Это необходимо для того, чтобы его коллеги могли сразу же понять смысл загружаемого значения.

- LDHT $\$X, \$Y, \$Z$ (загрузить тетра в старшую половину): $u(\$X) \leftarrow u(M_4[A]) \times 2^{32}$.
Здесь тетрабайт $M_4[A]$ загружается в *левую* половину $\$X$, а правая половина заполняется нулями. Например, LDHT $\$1, \$2, \$3$ приводит к $\$1 \leftarrow \#89ab\ cdef\ 0000\ 0000$, учитывая (6) с $\$2 + \$3 = 1005$.

- LDA $\$X, \$Y, \$Z$ (загрузить адрес): $u(\$X) \leftarrow A$.

Эта инструкция помещает адрес ячейки памяти в регистр и она идентична инструкции ADDU “добавить беззнаковое число”, которая описывается ниже. Дело в том, что фраза “загрузить адрес” описывает данный процесс лучше, чем фраза “добавить беззнаковое число”.

- STB $\$X, \$Y, \$Z$ (сохранить байт): $s(M_1[A]) \leftarrow s(\$X)$.
- STW $\$X, \$Y, \$Z$ (сохранить вайд): $s(M_2[A]) \leftarrow s(\$X)$.
- STT $\$X, \$Y, \$Z$ (сохранить тетра): $s(M_4[A]) \leftarrow s(\$X)$.
- STO $\$X, \$Y, \$Z$ (сохранить окта): $s(M_8[A]) \leftarrow s(\$X)$.

Эти инструкции выполняют обратное действие, помещая данные из регистра в ячейку памяти. Переполнение возможно, если (знаковое) число в регистре лежит за пределами ячейки памяти. Например, регистр $\$1$ содержит число $-65536 = \#ffffff\ ffff\ 0000$. Тогда, если $\$2 = 1000$, $\$3 = 2$ и (6) содержит

STB \$1,\$2,\$3 sets $M_8[1000] \leftarrow \#0123006789abcdef$ (переполнение);
 STW \$1,\$2,\$3 sets $M_8[1000] \leftarrow \#0123000089abcdef$ (переполнение);
 STT \$1,\$2,\$3 sets $M_8[1000] \leftarrow \#ffff000089abcdef$;
 STO \$1,\$2,\$3 sets $M_8[1000] \leftarrow \#ffffffffff0000$.

- STBU \$X,\$Y,\$Z (сохранить беззнаковый байт):

$$u(M_1[A]) \leftarrow u(\$X) \bmod 2^8.$$

- STWU \$X,\$Y,\$Z (сохранить беззнаковый вайд):

$$u(M_2[A]) \leftarrow u(\$X) \bmod 2^{16}.$$

- STTU \$X,\$Y,\$Z (сохранить беззнаковый тетра):

$$u(M_4[A]) \leftarrow u(\$X) \bmod 2^{32}.$$

- STOU \$X,\$Y,\$Z (сохранить беззнаковый окта): $u(M_8[A]) \leftarrow u(\$X)$.

Эти инструкции оказывают такое же влияние на память, как и их знаковые аналоги STB, STW, STT и STO, но без переполнения.

- STHT \$X,\$Y,\$Z (сохранить старшую половину в тетра):

$$u(M_4[A]) \leftarrow \lfloor u(\$X)/2^{32} \rfloor.$$

Левая половина регистра \$X сохраняется в тетрабайте памяти $M_4[A]$.

- STCO X,\$Y,\$Z (сохранить константу в октабайте): $u(M_8[A]) \leftarrow X$.

Константа между 0 и 255 сохраняется в октабайте памяти $M_8[A]$.

Арифметические операторы. Большинство операций MMIX выполняется с регистрами. Начнем их описание с операций сложения, вычитания, умножения и деления, поскольку компьютеры предназначены именно для вычислений.

- ADD \$X,\$Y,\$Z (сложить): $s(\$X) \leftarrow s(\$Y) + s(\$Z)$.
- SUB \$X,\$Y,\$Z (вычесть): $s(\$X) \leftarrow s(\$Y) - s(\$Z)$.
- MUL \$X,\$Y,\$Z (умножить): $s(\$X) \leftarrow s(\$Y) \times s(\$Z)$.
- DIV \$X,\$Y,\$Z (разделить): $s(\$X) \leftarrow \lfloor s(\$Y)/s(\$Z) \rfloor$ [$\$Z \neq 0$] и $s(rR) \leftarrow s(\$Y) \bmod s(\$Z)$.

Суммы, разности и произведения получаются очевидным образом, а результат команды DIV приводит к частному и остатку, согласно определению из раздела 1.2.4. Остаток помещается в специальный *регистр остатка* rR, доступ к которому можно получить с помощью описанной ниже инструкции GET \$X,rR. Если делитель \$Z равен нулю, то DIV дает $\$X \leftarrow 0$ и $rR \leftarrow \$Y$ (см. 1.2.4–(10)). Контроль “допустимости деления целых чисел” также предусмотрен.

- ADDU \$X,\$Y,\$Z (сложить беззнаковые): $u(\$X) \leftarrow (u(\$Y) + u(\$Z)) \bmod 2^{64}$.
- SUBU \$X,\$Y,\$Z (вычесть беззнаковые): $u(\$X) \leftarrow (u(\$Y) - u(\$Z)) \bmod 2^{64}$.
- MULU \$X,\$Y,\$Z (умножить беззнаковые): $u(rH \$X) \leftarrow u(\$Y) \times u(\$Z)$.
- DIVU \$X,\$Y,\$Z (разделить беззнаковые): $u(\$X) \leftarrow \lfloor u(rD \$Y)/u(\$Z) \rfloor$, $u(rR) \leftarrow u(rD \$Y) \bmod u(\$Z)$, if $u(\$Z) > u(rD)$; в противном случае $\$X \leftarrow rD$, $rR \leftarrow \$Y$.

Арифметические действия с беззнаковыми значениями никогда не приводят к переполнению. Полное 16-байтовое произведение выполняется с помощью команды MULU, а старшая половина помещается в специальный *регистр старшей половины произведения* rH. Например, если беззнаковое значение $\#9e3779b97f4a7c16$ в (2) и (4) выше умножить на самое себя, то получим

$$rH \leftarrow \#61c8864680b583ea, \quad \$X \leftarrow \#1bb32095ccdd51e4. \quad (7)$$

В данном примере значение rH оказалось равным 2^{64} минус исходное число $\#9e3779b97f4a7c16$, но это не совпадение! Причина заключается в том, что (2) фактически дает первые 64 бита двоичного представления “золотого сечения” $\phi^{-1} = \phi - 1$, если разместить точку в двоичном числе *слева*. (См. табл. 2 в приложении А.) Возведение в квадрат дает нам приближение к двоичному представлению $\phi^{-2} = 1 - \phi^{-1}$, с точкой в двоичном представлении *слева* от rH .

Деление с помощью DIVU дает 8-байтовое частное и остаток от деления 16-байтового делимого на 8-байтовый делитель. Верхняя половина делимого появляется в специальном *регистре делимого* rD , который равен нулю в начале программы.

В этом регистре может быть установлено любое значение с помощью описанной ниже команды PUT $rD, \$Z$. Если значение в rD больше или равно делителю, то DIVU $\$X, \$Y, \$Z$ просто равняется $\$X \leftarrow rD$ и $rR \leftarrow \$Y$. (Так всегда происходит, если $\$Z$ равно нулю.) Но при выполнении команды DIVU никогда не происходит проверки допустимости деления.

Инструкция ADDU вычисляет адрес памяти A , согласно определению (5). Следовательно, как уже говорилось ранее, иногда для ADDU используется альтернативное имя LDA. Перечисленные ниже команды также способствуют вычислению адреса.

- 2ADDU $\$X, \$Y, \$Z$ (умножает на 2 и прибавляет беззнаковое число):

$$u(\$X) \leftarrow (u(\$Y) \times 2 + u(\$Z)) \bmod 2^{64}.$$
- 4ADDU $\$X, \$Y, \$Z$ (умножает на 4 и прибавляет беззнаковое число):

$$u(\$X) \leftarrow (u(\$Y) \times 4 + u(\$Z)) \bmod 2^{64}.$$
- 8ADDU $\$X, \$Y, \$Z$ (умножает на 8 и прибавляет беззнаковое число):

$$u(\$X) \leftarrow (u(\$Y) \times 8 + u(\$Z)) \bmod 2^{64}.$$
- 16ADDU $\$X, \$Y, \$Z$ (умножает на 16 и прибавляет беззнаковое число):

$$u(\$X) \leftarrow (u(\$Y) \times 16 + u(\$Z)) \bmod 2^{64}.$$

Команда 2ADDU $\$X, \$Y, \$Y$ выполняется быстрее, чем умножение на 3, если не возникает переполнения.

- NEG $\$X, Y, \Z (отрицание): $s(\$X) \leftarrow Y - s(\$Z)$.
- NEGU $\$X, Y, \Z (отрицание беззнакового значения): $u(\$X) \leftarrow (Y - u(\$Z)) \bmod 2^{64}$.

В этих командах Y — это беззнаковая константа, а не номер регистра (как и значение X было беззнаковой константой в инструкции STC0). Обычно Y равняется нулю и в этом случае можно записать NEG $\$X, \Z или NEGU $\$X, \Z .

- SL $\$X, \$Y, \$Z$ (сдвиг влево): $s(\$X) \leftarrow s(\$Y) \times 2^{u(\$Z)}$.
- SLU $\$X, \$Y, \$Z$ (сдвиг влево беззнакового значения):

$$u(\$X) \leftarrow (u(\$Y) \times 2^{u(\$Z)}) \bmod 2^{64}.$$
- SR $\$X, \$Y, \$Z$ (сдвиг вправо): $s(\$X) \leftarrow \lfloor s(\$Y) / 2^{u(\$Z)} \rfloor$.
- SRU $\$X, \$Y, \$Z$ (сдвиг вправо беззнакового значения): $u(\$X) \leftarrow \lfloor u(\$Y) / 2^{u(\$Z)} \rfloor$.

Операторы SL и SLU приводят к одинаковому результату $\$X$, но оператор SL может приводить к переполнению, а оператор SLU — никогда. Оператор SR пропускает знаковый бит при сдвиге вправо, а оператор SRU вставляет нули с левого края. Следовательно, операторы SR и SRU приводят к одинаковому результату $\$X$ тогда и только тогда, когда $\$Y$ содержит неотрицательное значение или $\$Z$ содержит нуль. Инструкции SL и SR выполняются гораздо быстрее, чем MUL и DIV, причем почти в 2 раза. Инструкция SLU выполняется в 2 раза быстрее, чем MULU, хотя она не влияет на rH , в отличие от MULU. Инструкция SRU выполняется в 2 раза быстрее,

чем DIVU, хотя она не влияет на гD. Обозначение $y \ll z$ часто используется для указания сдвига двоичного значения y влево на z битов. Аналогично, обозначение $y \gg z$ используется для указания сдвига вправо.

- CMP $\$X, \$Y, \$Z$ (сравнить):

$$s(\$X) \leftarrow [s(\$Y) > s(\$Z)] - [s(\$Y) < s(\$Z)].$$
- CMPU $\$X, \$Y, \$Z$ (сравнить беззнаковые значения):

$$s(\$X) \leftarrow [u(\$Y) > u(\$Z)] - [u(\$Y) < u(\$Z)].$$

Каждая из этих инструкций устанавливает $\$X$ в -1 , либо 0 , либо 1 , в зависимости от того $\$Y$ меньше, равно или больше, чем регистр $\$Z$.

Условные инструкции. Некоторые инструкции основаны на том, содержит ли регистр положительное значение, отрицательное значение, нуль и т.д.

- CSN $\$X, \$Y, \$Z$ (условное присвоение, если значение отрицательное):
 если $s(\$Y) < 0$, установить $\$X \leftarrow \Z .
- CSZ $\$X, \$Y, \$Z$ (условное присвоение, если значение равно нулю):
 если $\$Y = 0$, установить $\$X \leftarrow \Z .
- CSP $\$X, \$Y, \$Z$ (условное присвоение, если значение положительное):
 если $s(\$Y) > 0$, установить $\$X \leftarrow \Z .
- CSOD $\$X, \$Y, \$Z$ (условное присвоение, если значение нечетное):
 если $\$Y \bmod 2 = 1$, установить $\$X \leftarrow \Z .
- CSNN $\$X, \$Y, \$Z$ (условное присвоение, если значение неотрицательное):
 если $s(\$Y) \geq 0$, установить $\$X \leftarrow \Z .
- CSNZ $\$X, \$Y, \$Z$ (условное присвоение, если значение не равно нулю):
 если $\$Y \neq 0$, установить $\$X \leftarrow \Z .
- CSNP $\$X, \$Y, \$Z$ (условное присвоение, если значение неположительное):
 если $s(\$Y) \leq 0$, установить $\$X \leftarrow \Z .
- CSEV $\$X, \$Y, \$Z$ (условное присвоение, если значение четное):
 если $\$Y \bmod 2 = 0$, установить $\$X \leftarrow \Z .

Если регистр $\$Y$ удовлетворяет заданному условию, то регистр $\$Z$ копируется в регистр $\$X$; в противном случае ничего не происходит. Считается, что регистр содержит отрицательное значение тогда и только тогда, когда ведущий (самый левый) бит содержит 1. Аналогично, считается, что регистр содержит нечетное значение тогда и только тогда, когда замыкающий (самый правый) бит содержит 1.

- ZSN $\$X, \$Y, \$Z$ (нуль или установить, если значение отрицательное):

$$\$X \leftarrow \$Z [s(\$Y) < 0].$$
- ZSZ $\$X, \$Y, \$Z$ (нуль или установить, если значение равно нулю):

$$\$X \leftarrow \$Z [\$Y = 0].$$
- ZSP $\$X, \$Y, \$Z$ (нуль или установить, если значение положительное):

$$\$X \leftarrow \$Z [s(\$Y) > 0].$$
- ZSOD $\$X, \$Y, \$Z$ (нуль или установить, если значение нечетное):

$$\$X \leftarrow \$Z [\$Y \bmod 2 = 1].$$
- ZSNN $\$X, \$Y, \$Z$ (нуль или установить, если значение неотрицательное):

$$\$X \leftarrow \$Z [s(\$Y) \geq 0].$$
- ZSNZ $\$X, \$Y, \$Z$ (нуль или установить, если значение не равно нулю):

$$\$X \leftarrow \$Z [\$Y \neq 0].$$

- ZSNP $\$X, \$Y, \$Z$ (нуль или установить, если значение положительное):
 $\$X \leftarrow \$Z [s(\$Y) \leq 0]$.
- ZSEV $\$X, \$Y, \$Z$ (нуль или установить, если значение нечетное):
 $\$X \leftarrow \$Z [\$Y \bmod 2 = 0]$.

Если регистр $\$Y$ удовлетворяет заданному условию, то регистр $\$Z$ копируется в регистр $\$X$; в противном случае регистр $\$X$ устанавливается в нуль.

Битовые операторы. Часто бывает удобно представить октабайт x в виде вектора $v(x)$ из 64 отдельных битов и выполнять операции одновременно с каждым компонентом двух таких векторов.

- AND $\$X, \$Y, \$Z$ (битовое И): $v(\$X) \leftarrow v(\$Y) \wedge v(\$Z)$.
- OR $\$X, \$Y, \$Z$ (битовое ИЛИ): $v(\$X) \leftarrow v(\$Y) \vee v(\$Z)$.
- XOR $\$X, \$Y, \$Z$ (битовое исключающее ИЛИ): $v(\$X) \leftarrow v(\$Y) \oplus v(\$Z)$.
- ANDN $\$X, \$Y, \$Z$ (битовое И-НЕ): $v(\$X) \leftarrow v(\$Y) \wedge \bar{v}(\$Z)$.
- ORN $\$X, \$Y, \$Z$ (битовое ИЛИ-НЕ): $v(\$X) \leftarrow v(\$Y) \vee \bar{v}(\$Z)$.
- NAND $\$X, \$Y, \$Z$ (битовое НЕ-И): $\bar{v}(\$X) \leftarrow v(\$Y) \wedge v(\$Z)$.
- NOR $\$X, \$Y, \$Z$ (битовое НЕ-ИЛИ): $\bar{v}(\$X) \leftarrow v(\$Y) \vee v(\$Z)$.
- NXOR $\$X, \$Y, \$Z$ (битовое НЕ-исключающее ИЛИ): $\bar{v}(\$X) \leftarrow v(\$Y) \oplus v(\$Z)$.

Здесь \bar{v} обозначает *дополнение* вектора v , полученное при изменении 0 на 1 и 1 на 0. Бинарные операторы \wedge , \vee и \oplus определяются следующими правилами:

$$\begin{array}{lll}
 0 \wedge 0 = 0, & 0 \vee 0 = 0, & 0 \oplus 0 = 0, \\
 0 \wedge 1 = 0, & 0 \vee 1 = 1, & 0 \oplus 1 = 1, \\
 1 \wedge 0 = 0, & 1 \vee 0 = 1, & 1 \oplus 0 = 1, \\
 1 \wedge 1 = 1, & 1 \vee 1 = 1, & 1 \oplus 1 = 0,
 \end{array} \tag{8}$$

и применяются независимо к каждому биту. Логическое И аналогично умножению или нахождению минимума, а логическое ИЛИ — нахождению максимума. Логическое исключающее-ИЛИ аналогично сложению по модулю 2.

- MUX $\$X, \$Y, \$Z$ (битовое умножение): $v(\$X) \leftarrow (v(\$Y) \wedge v(rM)) \vee (v(\$Z) \wedge \bar{v}(rM))$.

Оператор MUX комбинирует два битовых вектора с учетом особого *регистра мультиплексной маски* rM (multiplex mask), выбирая те биты $\$Y$, для которых rM равен 1, и те биты $\$Z$, для которых rM равен 0.

- SADD $\$X, \$Y, \$Z$ (сложение в сторону): $s(\$X) \leftarrow s(\sum (v(\$Y) \wedge \bar{v}(\$Z)))$.

Оператор SADD вычисляет количество битовых позиций, для которых регистр $\$Y$ содержит 1, а регистр $\$Z$ — 0.

Байтовые операторы. Аналогично, октабайт x можно рассматривать как вектор $b(x)$ из восьми отдельных байтов, каждый из которых содержит целое число от 0 до 255. Либо его можно рассматривать как вектор $w(x)$ из четырех отдельных вайдов, либо как вектор $t(x)$ из двух беззнаковых тетра. Перечисленные ниже операторы выполняются сразу со всеми компонентами вектора.

- BDIF $\$X, \$Y, \$Z$ (разность байтов): $b(\$X) \leftarrow b(\$Y) \div b(\$Z)$.
- WDIF $\$X, \$Y, \$Z$ (разность вайдов): $w(\$X) \leftarrow w(\$Y) \div w(\$Z)$.
- TDIF $\$X, \$Y, \$Z$ (разность тетра): $t(\$X) \leftarrow t(\$Y) \div t(\$Z)$.
- ODIF $\$X, \$Y, \$Z$ (разность окта): $u(\$X) \leftarrow u(\$Y) \div u(\$Z)$.

Здесь знак $\dot{-}$ обозначает операцию *насыщающего вычитания*, которая часто называется “точка минус” (“dot minus”), или “монус” (“monus”):

$$y \dot{-} z = \max(0, y - z). \quad (9)$$

Эти операторы имеют большое значение для обработки текста и компьютерной графики (где байты или вайды представляют значения пикселей).

В упражнениях 27–30 рассматриваются некоторые основные свойства этих операторов.

Октабайт можно также рассматривать как 8×8 *логическую матрицу*, которая является массивом 8×8 из 0 и 1. Пусть $m(x)$ является матрицей, которая в строках сверху вниз содержит байты x слева направо. Пусть $m^T(x)$ является транспонированной матрицей, в которой *столбцы* содержат байты x . Например, если $x = \#9e3779b97f4a7c16$ является октабайтом (2), то

$$m(x) = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}, \quad m^T(x) = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}. \quad (10)$$

Эта интерпретация октабайтов предполагает две хорошо известные математикам операции, но их определение будет дано немного позже.

Если A является матрицей $m \times n$ и B является матрицей $n \times s$, а \circ и \bullet — бинарные операторы, то *обобщенное матричное произведение* $A \circ B$ дает в результате матрицу C с размерами $m \times s$, в которой

$$C_{ij} = (A_{i1} \bullet B_{1j}) \circ (A_{i2} \bullet B_{2j}) \circ \cdots \circ (A_{in} \bullet B_{nj}) \quad (11)$$

для $1 \leq i \leq m$ и $1 \leq j \leq s$. [См. К. Е. Iverson, *A Programming Language* (Wiley, 1962), 23–24; здесь предполагается, что операция \circ является ассоциативной.] Обычное матричное произведение получается в случае, если операцией \circ является операция $+$, а операцией \bullet — операция \times . Кроме них имеется еще несколько важных операций с логическими матрицами, если допустить, что операцией \circ является операция \vee или \oplus :

$$(A \vee_x B)_{ij} = A_{i1} B_{1j} \vee A_{i2} B_{2j} \vee \cdots \vee A_{in} B_{nj}; \quad (12)$$

$$(A \oplus_x B)_{ij} = A_{i1} B_{1j} \oplus A_{i2} B_{2j} \oplus \cdots \oplus A_{in} B_{nj}. \quad (13)$$

Обратите внимание, что если каждая строка A содержит по крайней мере одно значение 1, то, по крайней мере, один член в (12) или (13) не равен нулю. То же верно, если каждый столбец B содержит по крайней мере одно значение 1. Следовательно, в таких случаях $A \vee_x B$ и $A \oplus_x B$ оказываются идентичными обычному матричному произведению $A \times B = AB$.

- MOR $\$X, \$Y, \$Z$ (множественное ИЛИ): $m^T(\$X) \leftarrow m^T(\$Y) \vee_x m^T(\$Z)$;
или эквивалентно, $m(\$X) \leftarrow m(\$Z) \vee_x m(\$Y)$. (См. упражнение 32.)

- **MXOR \$X, \$Y, \$Z** (множественное исключительное ИЛИ): $m^T(\$X) \leftarrow m^T(\$Y) \oplus m^T(\$Z)$; или эквивалентно $m(\$X) \leftarrow m(\$Z) \oplus m(\$Y)$.

Эти операции устанавливают каждый байт \$X с учетом соответствующего байта \$Z и используя его биты для выборки байтов \$Y; а выбранные байты затем участвуют в операции ИЛИ, либо исключительное ИЛИ. Если, например,

$$\$Z = \#0102040810204080, \quad (14)$$

то **MOR** и **MXOR** устанавливают регистр \$X в *обращение байта* регистра \$Y: k -й байт с левой части \$X будет установлен в k -й байт с правой части \$Y для $1 \leq k \leq 8$. С другой стороны, если $\$Z = \#00000000000000ff$, то **MOR** и **MXOR** установят все байты \$X в нуль, за исключением самого правого байта, который станет равным результату **OR** или **XOR** всех восьми байтов \$Y. В упражнениях 33–37 иллюстрируются некоторые из многих практических применений этих распространенных команд.

Операции с плавающей точкой. MMIX включает полную реализацию известного стандарта IEEE/ANSI Standard 754 для арифметических операций с числами с плавающей точкой. Полное описание подробностей операций с плавающей точкой приводятся в разделе 4.2 и документации *MMIXware*, а здесь достаточно привести только краткое описание.

Каждый октабайт x представляет двоичное число с плавающей точкой $f(x)$ со следующим определением: самый левый бит x определяет знак ($0 = '+'$, $1 = '-'$), следующие 11 битов — *экспоненту* E , а остальные 52 бита — *дробную часть* F . Тогда значения представляются следующим образом:

$$\begin{aligned} &\pm 0.0, \text{ если } E = F = 0 \text{ (нуль);} \\ &\pm 2^{-1074}F, \text{ если } E = 0 \text{ и } F \neq 0 \text{ (субнормальное);} \\ &\pm 2^{E-1023}(1 + F/2^{52}), \text{ если } 0 < E < 2047 \text{ (нормальное);} \\ &\pm \infty, \text{ если } E = 2047 \text{ и } F = 0 \text{ (бесконечное);} \\ &\pm \text{NaN}(F/2^{52}), \text{ если } E = 2047 \text{ и } F \neq 0 \text{ (не-является-числом).} \end{aligned}$$

“Короткое” (“short”) число с плавающей точкой $f(t)$ аналогично представляется тетрабайтом t , но его экспонента содержит только 8 битов, дробная часть только 23 бита. Например, в обычном случае $0 < E < 255$ “короткое” (“short”) число с плавающей точкой представляется как $\pm 2^{E-127}(1 + F/2^{23})$.

- **FADD \$X, \$Y, \$Z** (сложение чисел с плавающей точкой): $f(\$X) \leftarrow f(\$Y) + f(\$Z)$.
- **FSUB \$X, \$Y, \$Z** (вычитание чисел с плавающей точкой): $f(\$X) \leftarrow f(\$Y) - f(\$Z)$.
- **FMUL \$X, \$Y, \$Z** (умножение чисел с плавающей точкой): $f(\$X) \leftarrow f(\$Y) \times f(\$Z)$.
- **FDIV \$X, \$Y, \$Z** (деление чисел с плавающей точкой): $f(\$X) \leftarrow f(\$Y)/f(\$Z)$.
- **FREM \$X, \$Y, \$Z** (остаток от деления чисел с плавающей точкой):
 $f(\$X) \leftarrow f(\$Y) \bmod f(\$Z)$.
- **FSQRT \$X, \$Z** или **FSQRT \$X, Y, \$Z** (корень числа с плавающей точкой):
 $f(\$X) \leftarrow f(\$Z)^{1/2}$.
- **FINT \$X, \$Z** или **FINT \$X, Y, \$Z** (целая часть числа с плавающей точкой):
 $f(\$X) \leftarrow \text{int } f(\$Z)$.
- **FCMP \$X, \$Y, \$Z** (сравнение чисел с плавающей точкой):
 $s(\$X) \leftarrow [f(\$Y) > f(\$Z)] - [f(\$Y) < f(\$Z)]$.
- **FEQL \$X, \$Y, \$Z** (приравнивание чисел с плавающей точкой):
 $s(\$X) \leftarrow [f(\$Y) = f(\$Z)]$.

- **FUN \$X,\$Y,\$Z** (проверка упорядоченности чисел с плавающей точкой):
 $s(\$X) \leftarrow [f(\$Y) \parallel f(\$Z)] *$.
- **FCMPE \$X,\$Y,\$Z** (сравнение чисел с плавающей точкой с регистром rE):
 $s(\$X) \leftarrow [f(\$Y) \succ f(\$Z) (f(rE))] - [f(\$Y) \prec f(\$Z) (f(rE))]$, см. 4.2.2–(21).
- **FEQLE \$X,\$Y,\$Z** (приравнивание чисел с плавающей точкой с регистром rE):
 $s(\$X) \leftarrow [f(\$Y) \approx f(\$Z) (f(rE))]$, см. 4.2.2–24.
- **FUNE \$X,\$Y,\$Z** (проверка упорядоченности чисел с плавающей точкой по отношению к регистру rE): $s(\$X) \leftarrow [f(\$Y) \parallel f(\$Z) (f(rE))]$.
- **FIX \$X,\$Z** или **FIX \$X,Y,\$Z** (преобразование чисел с плавающей точкой к целому типу): $s(\$X) \leftarrow \text{int } f(\$Z)$.
- **FIXU \$X,\$Z** или **FIXU \$X,Y,\$Z** (преобразование чисел с плавающей точкой к беззнаковому типу): $u(\$X) \leftarrow (\text{int } f(\$Z)) \bmod 2^{64}$.
- **FLOT \$X,\$Z** или **FLOT \$X,Y,\$Z** (преобразование к числу с плавающей точкой):
 $f(\$X) \leftarrow s(\$Z)$.
- **FLOTU \$X,\$Z** или **FLOTU \$X,Y,\$Z** (преобразование к беззнаковому числу с плавающей точкой): $f(\$X) \leftarrow u(\$Z)$.
- **SFLOT \$X,\$Z** или **SFLOT \$X,Y,\$Z** (преобразование к краткому (short) числу с плавающей точкой): $f(\$X) \leftarrow f(T) \leftarrow s(\$Z)$.
- **SFLOTU \$X,\$Z** или **SFLOTU \$X,Y,\$Z** (преобразование к беззнаковому краткому (short) числу с плавающей точкой): $f(\$X) \leftarrow f(T) \leftarrow u(\$Z)$.
- **LDSF \$X,\$Y,\$Z** или **LDSF \$X,A** (загрузить short float): $f(\$X) \leftarrow f(M_4[A])$.
- **STSF \$X,\$Y,\$Z** или **STSF \$X,A** (сохранить short float): $f(M_4[A]) \leftarrow f(\$X)$.

Если нельзя присвоить точное значение, то при присвоении чисел с плавающей точкой для определения приближенного значения используется текущий режим округления. Существует четыре режима округления: 1 (ROUND_OFF), 2 (ROUND_UP), 3 (ROUND_DOWN) и 4 (ROUND_NEAR). В случае необходимости для указания иного режима округления, вместо текущего, можно использовать поле Y операторов FSQRT, FINT, FIX, FIXU, FLOT, FLOTU, SFLOT и SFLOTU. Например, оператор **FIX \$X, ROUND_UP, \$Z** приводит к $s(\$X) \leftarrow \lceil f(\$Z) \rceil$. Операторы SFLOT и SFLOTU сначала округляют значение, как при сохранении его в анонимный тетрабайт T, а потом преобразуют его в октабайт.

Оператор **int** приводит к округлению до целого числа. Оператор $y \text{ rem } z$ означает $y - nz$, где n является ближайшим целым числом к y/z или ближайшим *четным* целым числом в случае равенства. Если операнды имеют бесконечную величину или не-являются-числами (NaN), то применяются особые правила, а особые соглашения определяют знак нулевого результата. Значения +0.0 и -0.0 имеют разное представление в виде чисел с плавающей точкой, но оператор FEQL приравнивает их. Все такие технические детали подробно рассматриваются в документации *MMIXware*, а их важность обосновывается в разделе 4.2.

Немедленные константы. В программах часто используются малые постоянные величины. Это необходимо, например, чтобы добавить или вычесть 1 из регистра либо сдвинуть на 32 и т.п. В таких случаях излишне загружать малую константу из

* Регистр X равен 1, если \$Y и \$Z “неупорядочены” согласно соглашениям об арифметике чисел с плавающей точкой, т.е. если любое из них не-является-числом. В противном случае регистр X равен 0. — *Примеч. ред.*

памяти в другой регистр. Потому в MMIX предусмотрен общий механизм, благодаря которому такие константы можно получить “немедленно” в самой инструкции: для каждой рассмотренной выше инструкции предусмотрен вариант, в котором \$Z заменяется неотрицательным числом Z, если только инструкция не рассматривает \$Z как число с плавающей точкой.

Например, ‘ADD \$X,\$Y,\$Z’ имеет вариант ‘ADD \$X,\$Y,Z’, означающий, что $s(\$X) \leftarrow s(\$Y) + Z$; ‘NEG \$X,\$Z’ имеет вариант ‘NEG \$X,Z’, означающий, что $s(\$X) \leftarrow -Z$; ‘SRU \$X,\$Y,\$Z’ имеет вариант ‘SRU \$X,\$Y,Z’, означающий, что $u(\$X) \leftarrow \lfloor u(\$Y)/2^Z \rfloor$; ‘FLOT \$X,\$Z’ имеет вариант ‘FLOT \$X,Z’, означающий, что $f(\$X) \leftarrow Z$. Но ‘FADD \$X,\$Y,\$Z’ не имеет варианта с немедленной константой.

Опкод для ‘ADD \$X,\$Y,\$Z’ имеет вид #20, а опкод для ‘ADD \$X,\$Y,Z’ — вид #21. Для простоты мы используем тот же символ ADD в обоих случаях. Вообще, опкод для варианта с немедленной константой имеет больший размер, чем опкод для варианта с регистром.

В нескольких инструкциях так же используется *немедленная вайд-константа*, которая находится в диапазоне от #0000 = 0 до #ffff = 65535. Эти константы, которые появляются в байтах YZ, могут сдвигаться в позиции старшего, среднего старшего, среднего младшего или младшего вайда в октабайте.

- SETH \$X,YZ (установить старший вайд): $u(\$X) \leftarrow YZ \times 2^{48}$.
- SETMH \$X,YZ (установить средний старший вайд): $u(\$X) \leftarrow YZ \times 2^{32}$.
- SETML \$X,YZ (установить средний младший вайд): $u(\$X) \leftarrow YZ \times 2^{16}$.
- SETL \$X,YZ (установить младший вайд): $u(\$X) \leftarrow YZ$.
- INCH \$X,YZ (увеличить на старший вайд): $u(\$X) \leftarrow (u(\$X) + YZ \times 2^{48}) \bmod 2^{64}$.
- INCMH \$X,YZ (увеличить на средний старший вайд):
 $u(\$X) \leftarrow (u(\$X) + YZ \times 2^{32}) \bmod 2^{64}$.
- INCML \$X,YZ (увеличить на средний младший вайд):
 $u(\$X) \leftarrow (u(\$X) + YZ \times 2^{16}) \bmod 2^{64}$.
- INCL \$X,YZ (увеличить на младший вайд): $u(\$X) \leftarrow (u(\$X) + YZ) \bmod 2^{64}$.
- ORH \$X,YZ (битовое ИЛИ со старшим вайдом): $v(\$X) \leftarrow v(\$X) \vee v(YZ \ll 48)$.
- ORMH \$X,YZ (битовое ИЛИ со средним старшим вайдом):
 $v(\$X) \leftarrow v(\$X) \vee v(YZ \ll 32)$.
- ORML \$X,YZ (битовое ИЛИ со средним младшим вайдом):
 $v(\$X) \leftarrow v(\$X) \vee v(YZ \ll 16)$.
- ORL \$X,YZ (битовое ИЛИ с младшим вайдом): $v(\$X) \leftarrow v(\$X) \vee v(YZ)$.
- ANDNH \$X,YZ (битовое И-НЕ со старшим вайдом): $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ \ll 48)$.
- ANDNMH \$X,YZ (битовое И-НЕ со средним старшим вайдом):
 $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ \ll 32)$.
- ANDNML \$X,YZ (битовое И-НЕ со средним младшим вайдом):
 $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ \ll 16)$.
- ANDNL \$X,YZ (битовое И-НЕ с младшим вайдом): $v(\$X) \leftarrow v(\$X) \wedge \bar{v}(YZ)$.

Используя не более четырех из этих инструкций, можно получить произвольный октабайт в регистр без загрузки из памяти. Например, команды

SETH \$0,#0123; INCMH \$0,#4567; INCML \$0,#89ab; INCL \$0,#cdef

позволяют занести значение #0123456789abcdef в регистр \$0.

Язык ассемблера MMIX позволяет использовать сокращенную запись SET для SETL и SET \$X,\$Y, как аббревиатуру для распространенной операции OR \$X,\$Y,0.

Переходы и ветвления. Инструкции обычно выполняются в естественной последовательности. Иначе говоря, команда, которая выполняется после того, как MMIX отработает тетрабайт по адресу @, обычно является тетрабайтом по адресу @ + 4. (Символ @ обозначает место, “где” мы находимся.) Однако инструкции перехода и ветвления позволяют прервать эту последовательность.

- JMP RA (перейти): @ \leftarrow RA.

Здесь RA обозначает трехбайтовый *относительный адрес*, который более явно можно было бы записать как @ + 4 * XYZ, а именно XYZ тетрабайт вслед за текущим положением @. Например, ‘JMP @+4*2’ — это символическая форма для тетрабайта #f0000002. Если эта инструкция находится по адресу #1000, то следующей выполняемой инструкцией будет та, которая находится по адресу #1008. На самом деле можно было бы записать ‘JMP #1008’, но тогда значение XYZ зависело бы от предыдущего адреса.

Относительные отступы могут иметь отрицательное значение и в таком случае опкод возрастает на 1 и XYZ является отступом плюс 2^{24} . Например, ‘JMP @- 4*2’ является тетрабайтом #f1ffffffe. Опкод #f0 говорит компьютеру “перейти вперед”, а опкод #f1 — “перейти назад”, но оба они обозначаются JMP. Действительно, для перехода к адресу Addr обычно просто пишут ‘JMP Addr’, а программа MMIX вычисляет соответствующий опкод и соответствующее значение XYZ. Такой переход возможен на не более, чем 67 миллионов байт от текущего адреса.

- GO \$X,\$Y,\$Z (перейти по адресу): u(\$X) \leftarrow @ + 4, затем @ \leftarrow A.

Инструкция GO позволяет перейти к *абсолютному адресу*, в произвольной части памяти. Адрес A вычисляется по формуле (5), точно так же, как и при выполнении команд загрузки и сохранения. Перед переходом к указанному адресу адрес инструкции, которая изначально должна была быть следующей помещается в регистр \$X. Следовательно, можно было бы вернуться к этому адресу позднее, например, с помощью инструкции ‘GO \$X,\$X,0’ с Z = 0 в качестве немедленной константы.

- BN \$X,RA (перейти, если отрицательное): если s(\$X) < 0, то @ \leftarrow RA.
- BZ \$X,RA (перейти, если нуль): если \$X = 0, то @ \leftarrow RA.
- BP \$X,RA (перейти, если положительное): если s(\$X) > 0, то @ \leftarrow RA.
- BOD \$X,RA (перейти, если нечетное): если \$X mod 2 = 1, то @ \leftarrow RA.
- BNN \$X,RA (перейти, если неотрицательное): если s(\$X) \geq 0, то @ \leftarrow RA.
- BNZ \$X,RA (перейти, если ненулевое): если \$X \neq 0, то @ \leftarrow RA.
- BNP \$X,RA (перейти, если неположительное): если s(\$X) \leq 0, установить @ \leftarrow RA.
- BEV \$X,RA (перейти, если четное): если \$X mod 2 = 0, set @ \leftarrow RA.

Инструкция *ветвления* является условным переходом, который зависит от содержимого регистра \$X. Диапазон адресов назначения RA более ограничен, чем при работе JMP, поскольку только два байта доступны для выражения относительного отступа. Тем не менее, условный переход (ветвление) можно выполнять к любому тетрабайту между @ - 2^{18} и @ + 2^{18} - 4.

- PBN \$X,RA (вероятно перейти, если отрицательное): если s(\$X) < 0, то @ \leftarrow RA.
- PBZ \$X,RA (вероятно перейти, если нуль): если \$X = 0, то @ \leftarrow RA.

- PBP $\$X, RA$ (вероятно перейти, если положительное): если $s(\$X) > 0$, то $@ \leftarrow RA$.
- PBOD $\$X, RA$ (вероятно перейти, если нечетное): если $\$X \bmod 2 = 1$, то $@ \leftarrow RA$.
- PBNB $\$X, RA$ (вероятно перейти, если неотрицательное): если $s(\$X) \geq 0$, то $@ \leftarrow RA$.
- PBNZ $\$X, RA$ (вероятно перейти, если ненулевое): если $\$X \neq 0$, то $@ \leftarrow RA$.
- PBNP $\$X, RA$ (вероятно перейти, если неположительное): если $s(\$X) \leq 0$, то $@ \leftarrow RA$.
- PBEV $\$X, RA$ (вероятно перейти, если четное): если $\$X \bmod 2 = 0$, то $@ \leftarrow RA$.

Высокоскоростные компьютеры обычно работают быстрее, если можно предсказать момент ветвления, поскольку такая информация позволяет заглянуть вперед и подготовить к работе будущие инструкции. Следовательно, ММIX поощряет программистов создавать подсказки о вероятности ветвления. Если ветвление ожидается в более, чем половине случаев, то опытный программист будет использовать операторы-подсказки условного перехода (которые начинаются с символов PB) вместо стандартных операторов условного перехода (которые начинаются с символов B).

***Вызовы подпрограмм.** В ММIX также предусмотрено несколько инструкций, которые способствуют эффективному обмену данными между подпрограммами, с помощью *стека регистров*. Технические детали более подробно рассматриваются в разделе 1.4', а здесь приводится краткое неформальное описание. В коротких программах эти инструменты не используются.

- PUSHJ $\$X, RA$ (протолкнуть регистры и перейти): $\text{push}(X)$ и $rJ \leftarrow @ + 4$, а затем $@ \leftarrow RA$.
- PUSHGO $\$X, \$Y, \$Z$ (протолкнуть регистры и перейти к адресу): $\text{push}(X)$ и $rJ \leftarrow @ + 4$, а затем $@ \leftarrow A$.

В специальный *регистр возврата-перехода* rJ (return-jump) устанавливается адрес тетрабайта после выполнения команды PUSH. Действие “протолкнуть(X)” (“push(X)”) означает, грубо говоря, что локальные регистры от $\$0$ до $\$X$ сохраняются и временно недоступны. То, что было в $\$(X+1)$, теперь в $\$0$, а то, что было в $\$(X+2)$, теперь в $\$1$ и т.д. Но все регистры $\$k$ для $k \geq rG$ остаются в неизменном состоянии. Регистр rG является особым *глобальным пороговым регистром*, чье значение находится в диапазоне от 32 до 255, включительно.

Регистр $\$k$ называется *глобальным*, если $k \geq rG$. Регистр называется *локальным*, если $k < rL$. Здесь rL (local threshold) — это специальный *локальный пороговый регистр*, который информирует о количестве текущих активных локальных регистров. В противном случае, а именно, если $rL \leq k < rG$, то регистр $\$k$ называется *маргинальным*, а $\$k$ равняется нулю, когда он используется в команде как операнд-источник. Если маргинальный регистр $\$k$ используется в команде как операнд-цель, то rL автоматически увеличивается до $k + 1$ до выполнения команды, делая таким образом локальным регистр $\$k$.

- POP X, YZ (вытолкнуть из регистров и вернуться): $\text{pop}(X)$, then $@ \leftarrow rJ + 4 * YZ$.
Здесь “вытолкнуть(X)” (“pop(X)”) означает, грубо говоря, что все текущие локальные регистры, кроме X , становятся маргинальными. Потом локальные регистры, скрытые самой последней операцией “протолкнуть” (“push”), которая еще не была “вытолкнута” (“popped”), восстанавливаются в исходное состояние. Более подробно детали этого процесса описываются в разделе 1.4' и демонстрируются на многочисленных примерах.

- **SAVE \$X, 0** (сохранить состояние процесса): $u(\$X) \leftarrow \text{context}$.
- **UNSAVE \$Z** (восстановить состояние процесса): $\text{context} \leftarrow u(\$Z)$.

Инструкция **SAVE** сохраняет все текущие регистры в памяти в верхней части стека регистров и помещает адрес самого верхнего сохраненного октабайта в $u(\$X)$. Регистр $\$X$ должен быть глобальным, т.е. X должен быть $\geq rG$. Все текущие локальные и глобальные регистры сохраняются вместе со специальными регистрами rA , rD , rE , rG , rH , rJ , rM , rR и некоторыми другими, которые еще не рассматривались. Инструкция **UNSAVE** берет адрес самого верхнего октабайта и восстанавливает соответствующий контекст, т.е. отменяет предыдущую инструкцию **SAVE**. Инструкция **SAVE** устанавливает значение регистра rL в нуль, а инструкция **UNSAVE** восстанавливает прежнее значение. В **MMIX** имеются специальные регистры: *регистр отступа стека* (rO) (stack offset) и *регистр указателя стека* (rS) (stack pointer), которые контролируют операторы **PUSH**, **POP**, **SAVE** и **UNSAVE**. (Более подробно они описываются в разделе 1.4.)

***Системные операторы.** Несколько опкодов, предназначенных преимущественно для сверхбыстрой и/или параллельной архитектуры **MMIX**, представляют интерес только для опытных пользователей, поэтому здесь они будут только кратко упомянуты. Некоторые связанные с ними операции подобны командам “условного ветвления” в том смысле, что они подсказывают компьютеру, как спланировать будущие действия для достижения максимальной эффективности. Большинству программистов не нужны эти инструкции, возможно, за исключением **SYNCID**.

- **LDUNC \$X, \$Y, \$Z** (загрузить некешируемый октабайт): $s(\$X) \leftarrow s(M_8[A])$.
- **STUNC \$X, \$Y, \$Z** (сохранить некешируемый октабайт): $s(M_8[A]) \leftarrow s(\$X)$.

Эти команды выполняют те же операции, что и команды **LDO** и **STO**, но они также информируют компьютер о том, что загруженный или сохраненный октабайт и его ближайшие соседи, вероятно, не будут считываться или записываться в ближайшем будущем.

- **PRELD X, \$Y, \$Z** (предварительно загрузить данные).

Это значит, что некоторые байты от $M[A]$ до $M[A + X]$, вероятно, будут загружены или сохранены в ближайшем будущем.

- **PREST X, \$Y, \$Z** (предварительно сохранить данные).

Это значит, что все байты от $M[A]$ до $M[A + X]$ определено будут записаны (сохранены) до следующей операции чтения (загрузки).

- **PREGO X, \$Y, \$Z** (предварительно выбрать).

Это значит, что некоторые байты от $M[A]$ до $M[A + X]$, вероятно, будут использованы как инструкции в ближайшем будущем.

- **SYNCID X, \$Y, \$Z** (синхронизировать инструкции и данные).

Это значит, что все байты от $M[A]$ до $M[A + X]$ должны быть предварительно выбраны до их интерпретации как инструкций. В **MMIX** допускается, что инструкции программы не изменяются после запуска программы, если только эти инструкции не были подготовлены с помощью команды **SYNCID**. (См. упражнение 57.)

- **SYNCD X, \$Y, \$Z** (синхронизировать данные).

Это значит, что все байты от $M[A]$ до $M[A + X]$ должны быть перенесены в физическую память, чтобы другие компьютеры и устройства ввода-вывода могли прочесть их.

- **SYNC XYZ** (синхронизировать).

Ограничивает параллельные действия так, чтобы разные процессоры могли надежно сотрудничать (см. *MMIXware*). XYZ должны иметь значение 0, 1, 2 или 3.

- **CSWAP \$X, \$Y, \$Z** (сравнить и переставить октабайты).

Если $u(M_8[A]) = u(rP)$, где rP является специальным *регистром предсказания*, то $u(M_8[A]) \leftarrow u(\$X)$ и $u(\$X) \leftarrow 1$. В противном случае $u(rP) \leftarrow u(M_8[A])$ и $u(\$X) \leftarrow 0$. Такая атомарная (неделимая) операция полезна, когда независимые компьютеры совместно используют общую память.

- **LDVTS \$X, \$Y, \$Z** (загрузить статус виртуальной трансляции).

Эта инструкция подробно описывается в *MMIXware* и предназначена только для операционной системы.

***Прерывания.** Обычный поток инструкций от одного тетрабайта к следующему можно изменять не только с помощью переходов и ветвлений, но и с помощью менее предсказуемых событий, таких как переполнение или внешний сигнал. Реальные компьютеры должны справляться с такими проблемами, как нарушение безопасности и сбой аппаратного обеспечения. В MMIX различаются два типа программных прерываний: “перескоки” и “прерывания”. Перескок передает управление *обработчику перескоков*, который является частью пользовательской программы, а прерывание передает управление *обработчику прерываний*, который является частью операционной системы.

Восемь типов исключительных ситуаций могут возникать, когда MMIX выполняет арифметические действия, а именно проверка допустимости целочисленного деления (D), целочисленное переполнение (V), переполнение при операциях с числами с фиксированной точкой (W), недействительная операция чисел с плавающей точкой (I), переполнение при операциях с числами с плавающей точкой (O), потеря значащих разрядов при операциях с числами с плавающей точкой (U), деление на нуль при операциях с числами с плавающей точкой (Z), неточность при операциях с числами с плавающей точкой (X). Специальный *арифметический регистр состояния* гА содержит текущую информацию обо всех таких исключительных ситуациях. Восемь битов самого правого байта называются *битами событий* и обозначаются следующим образом: D_BIT (#80), V_BIT (#40), ..., X_BIT (#01), т.е. расположены в последовательности DVWIOUZX.

Восемь битов слева от битов событий в гА называются *битами разрешения*. Они располагаются в том же порядке DVWIOUZX. Если во время арифметической операции возникает исключительная ситуация, то MMIX учитывает соответствующий бит разрешения до перехода к следующей инструкции. Если бит разрешения равен 0, то соответствующий бит события равен 1. В противном случае компьютер вызывает обработчик перескоков, “перескакивая” к адресу #10 в случае исключительной ситуации D, #20 в случае исключительной ситуации V, ..., #80 в случае исключительной ситуации X. Таким образом, биты событий регистра гА записывают исключительные ситуации, которые не привели к перескокам. (Если происходит

несколько исключительных ситуаций, то самая левая обладает наибольшим приоритетом. Например, при одновременном возникновении исключительных ситуаций О и Х обрабатывается исключительная ситуация О.)

Два бита регистра rA слева от битов разрешения содержат текущий режим округления, mod 4. Другие 46 битов регистра rA должны быть равны нулю. Программа может изменить значения в регистре rA в любой момент с помощью команды PUT, которая рассматривается ниже.

- TRIP X,Y,Z или TRIP X,YZ или TRIP XYZ (перескочить).

Эта команда выполняет перескок к адресу #00.

После перескока компьютер MMIX использует пять специальных регистров для записи текущего состояния: *регистр самозагрузки* rB, *регистр места прерывания* rW, *регистр исполнения* rX, *регистр операнда Y* rY и *регистр операнда Z* rZ. Сначала для rB задается значение \$255, затем значение \$255 задается для rJ, а для rW задается значение @ + 4.

Для левой части rX задается значение #80000000, а для правой части — инструкция, через которую совершен перескок. Если прерванная инструкция не была командой сохранения, то для регистра rY задается значение \$Y, а для регистра rZ — значение \$Z (или значение Z при использовании немедленной константы). В противном случае для регистра rY задается значение A (адрес команды сохранения), а для регистра rZ — значение \$X (величина, которую нужно сохранить). В конечном итоге управление передается обработчику за счет установки для @ адреса обработчика (#00, либо #10, либо ..., либо #80).

- TRAP X,Y,Z или TRAP X,YZ или TRAP XYZ (прервать).

Эта команда аналогична команде TRIP, но она выполняет прерывание операционной системы. Специальные регистры rBB, rWW, rXX, rYY и rZZ заменяют регистры rB, rW, rX, rY и rZ. Специальный *регистр адреса прерывания* rT предоставляет адрес обработчика прерываний, который помещается в @. В разделе 1.3.2' описывается несколько команд TRAP, которые выполняют простые операции ввода-вывода. Обычный способ завершения программы заключается в том, чтобы выполнить команду 'TRAP 0'. Эта инструкция представляет собой #00000000, поэтому нужно быть осторожным, чтобы не ввести ее по ошибке.

Документация MMIXware содержит дополнительные сведения о внешних прерываниях, которые управляются специальным *регистром маски прерывания* rK и *регистром запроса прерывания* rQ. Динамическое прерывание, которое возникает, если $rK \wedge rQ \neq 0$, обрабатывается по адресу rTT вместо rT.

- RESUME 0 (продолжить после прерывания).

Если s(rX) отрицательно, то MMIX просто задает @ ← rW и берет следующую инструкцию оттуда. В противном случае, если старший байт rX равен нулю, то MMIX задает @ ← rW - 4 и выполняет инструкцию в младшей половине rX, как если бы она находилась там. (Этот прием можно использовать даже при отсутствии прерывания. Вставленная инструкция не обязательно должна быть командой RESUME.) В противном случае MMIX выполняет специальные действия, которые подробно описаны в документации MMIXware и преимущественно предназначены для операционной системы (см. упражнение 1.4.3'-14).

Полный набор инструкций. В табл. 1 приведены имена всех 256 опкодов, упорядоченных по их числовым значениям в шестнадцатеричной системе счисления. Например, команда ADD находится на пересечении верхней половины строки #2х и столбца #0, т.е. ADD является опкодом #20. Аналогично, команда ORL находится на пересечении нижней половины строки #Ех и столбца #В снизу, т.е. ORL является опкодом #ЕВ.

В табл. 1 фактически приведена команда 'ADD[I]', а не 'ADD', поскольку ADD на самом деле обозначает сразу два опкода. Опкод #20 возникает при использовании командой ADD \$X,\$Y,\$Z регистра \$Z, а опкод #21 — при использовании командой ADD \$X,\$Y,Z немедленной константы Z. Если такое различие необходимо подчеркнуть явно, то в таком случае говорят, что #20 является командой ADD, а опкод #21 — командой ADDI ("add immediate" — "сложить немедленно"). Аналогично, #F0 является командой JMP, а #F1 — командой JMPB ("jump backward" — "перейти назад"). Таким образом, каждому опкоду можно присвоить уникальное имя. Однако дополнительные символы I и B для большего удобства обычно опускаются при работе с программами для MMIX.

Итак, мы обсудили практически все опкоды MMIX. Здесь следует отметить еще две команды.

- GET \$X,Z (извлечь из специального регистра): $u(\$X) \leftarrow u(g[Z])$, где $0 \leq Z < 32$.
- PUT X,\$Z (поместить в специальный регистр): $u(g[X]) \leftarrow u(\$Z)$, где $0 \leq X < 32$.

Каждый специальный регистр имеет кодовый номер от 0 до 31. Символы гА, гВ, ..., используются для обозначения регистров исходя из особенностей человеческого восприятия информации. На самом деле регистр гА с точки зрения компьютера обозначается символами g[21], а регистр гВ — символами g[0], и т.д. Эти кодовые номера перечислены в табл. 2.

Команды GET не имеют ограничений, а команды PUT имеют. В регистр гG нельзя помещать значения больше 255, меньше 32 или меньше текущего значения регистра гL. В регистр гА нельзя помещать значения больше #3ffff. Если программа попытается увеличить значение регистра гL с помощью команды PUT, то значение регистра гL останется неизменным. Более того, программа не может с помощью команды PUT поместить значение в регистр гC, гN, гO, гS, гI, гT, гTT, гK, гQ, гU или гV. Эти "экстраспециальные" регистры имеют кодовые номера в диапазоне 8–18.

Большинство специальных регистров уже упомянуто в связи со специальными инструкциями, но MMIX также имеет "регистр времени" или *счетчик циклов* гC, который постоянно увеличивает свое значение; *счетчик сбоев* гF, для упрощения диагностики ошибок аппаратного обеспечения, *счетчик интервалов* гI, который постоянно уменьшает свое значение и приводит к прерыванию после достижения нуля; *регистр последовательных значений* гN (serial number), который присваивает каждому компьютеру MMIX уникальный номер; *счетчик использования*, гU, который увеличивается на 1 при каждом выполнении заданных опкодов; а *регистр виртуальной трансляции* гV, который устанавливает соответствие между "виртуальными" 64-битовыми адресами в программах и "фактическими" физическими адресами используемой памяти. Эти специальные регистры позволяют успешно реализовать компьютер MMIX в наиболее полной и жизнеспособной форме. Однако, они не имеют большого значения для материала этой книги. Более подробные сведения о них можно найти в документации MMIXware.

Таблица 1
ОПКОДЫ ММIX

	#0	#1	#2	#3	#4	#5	#6	#7	
#0x	TRAP 5v	FCMP v	FUN v	FEQL v	FADD 4v	FIX 4v	FSUB 4v	FIXU 4v	#0x
	FLOT[I] 4v		FLOTU[I] 4v		SFLOT[I] 4v		SFLOTU[I] 4v		
#1x	FMUL 4v	FCMPE 4v	FUNE v	FEQLE 4v	FDIV 40v	FSQRT 40v	FREM 4v	FINT 4v	#1x
	MUL[I] 10v		MULU[I] 10v		DIV[I] 60v		DIVU[I] 60v		
#2x	ADD[I] v		ADDU[I] v		SUB[I] v		SUBU[I] v		#2x
	2ADDU[I] v		4ADDU[I] v		8ADDU[I] v		16ADDU[I] v		
#3x	CMP[I] v		CMPU[I] v		NEG[I] v		NEGU[I] v		#3x
	SL[I] v		SLU[I] v		SR[I] v		SRU[I] v		
#4x	BN[B] v+π		BZ[B] v+π		BP[B] v+π		BOD[B] v+π		#4x
	BNN[B] v+π		BNZ[B] v+π		BNP[B] v+π		BEV[B] v+π		
#5x	PBN[B] 3v-π		PBZ[B] 3v-π		PBP[B] 3v-π		PBOD[B] 3v-π		#5x
	PBNN[B] 3v-π		PBNZ[B] 3v-π		PBNP[B] 3v-π		PBEV[B] 3v-π		
#6x	CSN[I] v		CSZ[I] v		CSP[I] v		CSOD[I] v		#6x
	CSNN[I] v		CSNZ[I] v		CSNP[I] v		CSEV[I] v		
#7x	ZSN[I] v		ZSZ[I] v		ZSP[I] v		ZSOD[I] v		#7x
	ZSNN[I] v		ZSNZ[I] v		ZSNP[I] v		ZSEV[I] v		
#8x	LDB[I] μ+v		LDBU[I] μ+v		LDW[I] μ+v		LDWU[I] μ+v		#8x
	LDT[I] μ+v		LDTU[I] μ+v		LDO[I] μ+v		LDOU[I] μ+v		
#9x	LDSF[I] μ+v		LDHT[I] μ+v		CSWAP[I] 2μ+2v		LDUNC[I] μ+v		#9x
	LDVTS[I] v		PRELD[I] v		PREGO[I] v		GO[I] 3v		
#Ax	STB[I] μ+v		STBU[I] μ+v		STW[I] μ+v		STWU[I] μ+v		#Ax
	STT[I] μ+v		STTU[I] μ+v		STO[I] μ+v		STOU[I] μ+v		
#Bx	STSFI[I] μ+v		STHT[I] μ+v		STCO[I] μ+v		STUNC[I] μ+v		#Bx
	SYNCD[I] v		PREST[I] v		SYNCID[I] v		PUSHGO[I] 3v		
#Cx	OR[I] v		ORN[I] v		NOR[I] v		XOR[I] v		#Cx
	AND[I] v		ANDN[I] v		NAND[I] v		NXOR[I] v		
#Dx	BDIF[I] v		WDIF[I] v		TDIF[I] v		ODIF[I] v		#Dx
	MUX[I] v		SADD[I] v		MOR[I] v		MXOR[I] v		
#Ex	SETH v	SETMH v	SETML v	SETL v	INCH v	INCMH v	INCML v	INCL v	#Ex
	ORH v	ORMH v	ORML v	ORL v	ANDNH v	ANDNMH v	ANDNML v	ANDNL v	
#Fx	JMP[B] v		PUSHJ[B] v		GETA[B] v		PUT[I] v		#Fx
	POP 3v	RESUME 5v	[UN]SAVE 20μ+v		SYNC v	SWYM v	GET v	TRIP 5v	
	#8	#9	#A	#B	#C	#D	#E	#F	

π = 2v, если условный переход выполняется; π = 0, если условный переход не выполняется.

- GETA \$X, RA (получить адрес): $u(\$X) \leftarrow RA$.

Эта инструкция загружает относительный адрес в регистр \$X, с помощью тех же соглашений, которые используются в командах ветвления. Например, GETA \$0, @ задает \$0 для адреса самой этой инструкции.

- SWYM X, Y, Z или SWYM X, YZ или SWYM XYZ (sympathize with your machinery — приносим соболезнования вашему компьютеру).

Эти последние 256 опкодов ММIX, вероятно, наиболее простые. Они часто называются холостыми, поскольку не выполняют никаких действий. Однако они позволяют компьютеру работать более слаженно, как плавание в бассейне укрепляет здоровье программиста. Байты X, Y и Z игнорируются.

Таблица 2
СПЕЦИАЛЬНЫЕ РЕГИСТРЫ ММІХ

	код	сохранить?	поместить?
rA регистр арифметического статуса	21	✓	✓
rB регистр самозагрузки (перескока)	0	✓	✓
rC регистр счетчика циклов	8		
rD регистр делимого	1	✓	✓
rE регистр эпсилон	2	✓	✓
rF регистр места сбоя	22		✓
rG регистр глобального порога	19	✓	✓
rH регистр старшей половины произведения	3	✓	✓
rI регистр счетчика интервалов	12		
rJ регистр возврата-перехода	4	✓	✓
rK регистр маски прерывания	15		
rL регистр локального порога	20	✓	✓
rM регистр мультиплексной маски	5	✓	✓
rN регистр последовательных значений	9		
rO регистр отступа стека	10		
rP регистр предсказания	23	✓	✓
rQ регистр запроса прерывания	16		
rR регистр остатка от деления	6	✓	✓
rS регистр указателя стека	11		
rT регистр адреса прерывания	13		
rU регистр счетчика использования	17		
rV регистр виртуальной трансляции	18		
rW регистр места прерывания (перескока)	24	✓	✓
rX регистр выполнения (перескока)	25	✓	✓
rY регистр операнда Y (перескока)	26	✓	✓
rZ регистр операнда Z (перескока)	27	✓	✓
rBB регистр самозагрузки (прерывания)	7		✓
rTT регистр адреса динамического прерывания	14		
rWW регистр места прерывания (прерывания)	28		✓
rXX регистр выполнения (прерывания)	29		✓
rYY регистр операнда Y (прерывания)	30		✓
rZZ регистр операнда Z (прерывания)	31		✓

Хронометраж. Далее в этой книге часто будут сравниваться разные программы для ММІХ для определения наиболее быстрой программы. Вообще говоря, такое сравнение нелегко выполнить, поскольку архитектура ММІХ может быть реализована разными способами. Хотя ММІХ представляет собой мифический компьютер, его мифическая аппаратная часть реализована как в дешевых медленных версиях, так и в дорогостоящих высокопроизводительных моделях. Время выполнения программы зависит не только от тактовой частоты, но и от количества функциональных единиц, которые могут быть активированы одновременно, и степени их конвейеризации, от используемых методов предварительной выборки инструкций до их выполнения, от размера оперативной памяти, используемой для создания иллюзии 2^{64} виртуальных байтов, от размера и стратегии выделения кэш-памяти, буферов и от многого другого.

С практической точки зрения время выполнения программы MMIX можно удовлетворительно оценить, присваивая фиксированные затраты каждой операции на основе приблизительного времени выполнения, которое определено для высокопроизводительного компьютера с большим объемом памяти. Итак, можно приступать. Предполагается, что для каждой операции требуется целое число v , где v (произносится “упс”)* — это единица измерения длительности такта в конвейерной реализации. Величина v уменьшается по мере улучшения технологии, и мы всегда будем стараться идти в ногу с самыми последними достижениями, поскольку измеряем время в единицах v , а не в наносекундах. Время выполнения, по нашим оценкам, будет зависеть от количества обращений к памяти, или *мемсов*, используемых программой, т.е. количество операций загрузки и сохранения. Например, мы будем предполагать, что для каждой инструкции LD0 (загрузить окта) требуется $\mu + v$, где μ — это средние затраты при обращении к памяти. Общее время выполнения программы может быть равно, скажем, $35\mu + 1000v$, что означает “35 мемсов плюс 1000 уцсов”.

Отношение μ/v постоянно увеличивается в течение многих лет. Никто не знает наверняка, продолжится ли эта тенденция, но опыт показывает, что μ и v могут рассматриваться как независимые величины.

В табл. 1, которая повторяется в конце этой книги, отображается время выполнения каждого опкода. Обратите внимание, что для большинства инструкций требуется $1v$, а для загрузки и сохранения $\mu + v$. Для ветвления или условного ветвления требуется $1v$, в случае верного предсказания, либо $3v$, в случае неверного предсказания. Для каждой операции с числами с плавающей точкой обычно требуется $4v$, хотя затраты на выполнение операций FDIV и FSQRT равны $40v$. Для целочисленного умножения требуется $10v$, а целочисленного деления — $60v$.

Используя предположения из табл. 1 для интуитивных оценок времени выполнения, нужно помнить, что фактическое время выполнения может очень сильно зависеть от порядка выполнения инструкций. Например, для целочисленного деления может потребоваться только один цикл, если найдется 60 других действий, которые нужно выполнить между моментом запуска команды и моментом получения результата. Несколько инструкций загрузки LDB (загрузить байт) может потребоваться для только одного обращения к памяти, если они относятся к одному октабайту. Результат команды загрузки обычно не доступен для использования в немедленно выполняемой инструкции. Как показывает опыт, некоторые алгоритмы хорошо работают с кэш-памятью, а другие — нет. Следовательно, величина μ на самом деле не является постоянной. Даже расположение инструкций в памяти может оказывать значительное влияние на производительность, поскольку некоторые инструкции могут отбираться предварительно вместе с другими. Поэтому программное обеспечение MMIXware содержит не только простой симулятор, который вычисляет время выполнения по правилам табл. 1, но и более полный *мета-симулятор*, который выполняет программы MMIX в широком диапазоне разных технологических реализаций. Пользователи мета-симулятора могут указать характеристики шины памяти и параметры кэш-памяти для инструкций и дан-

* Греческий символ эпсилон (v) шире, чем символ “ви” (v), но автор признает, что это различие весьма незначительно. Читатели, которые предпочитают говорить “ви” вместо “упс”, могут поступать по своему усмотрению. Однако здесь этот символ соответствует эпсилону.

ных, трансляции виртуальных адресов, конвейеризации и одновременной выдачи команд, предсказания ветвления и т.п. При наличии файла конфигурации и файла программы мета-симулятор точно определяет, насколько долго данное аппаратное обеспечение будет выполнять программу. Для получения надежной информации о реальном поведении программы следует использовать только мета-симулятор. Но такие результаты трудно интерпретировать, поскольку существует бесконечно большое количество конфигураций. Именно по этой причине часто используются более простые оценки из табл. 1.

Результаты любого теста не следует воспринимать буквально.

— БРАЙАН КЕРНИГАН (BRIAN KERNIGHAN)

и КРИСТОФЕР ВАН ВИК (CHRISTOPHER VAN WYK) (1998) SP+E 28 (98) 819

MMIX и реальность. Читателю, знающему основы программирования для MMIX, очевидны те задачи, которые могут решать современные компьютеры общего назначения, ведь MMIX очень похож на них. Однако компьютер MMIX в некотором смысле идеализирован, отчасти потому, что автор пытался придумать некий компьютер “будущего”, который не устарел бы чересчур быстро. Потому имеет смысл провести краткое сравнение компьютера MMIX с реально существующими компьютерами на исходе тысячелетия. Основные отличия между MMIX и этими компьютерами состоят в следующем.

- Коммерческие компьютеры не игнорируют младшие биты адресов памяти, как это делает MMIX при доступе к $M_8[A]$; они обычно настаивают на том, чтобы A было кратно 8. (Мы найдем множество применений этих ценных младших битов.)
- Коммерческие компьютеры обычно не имеют полной поддержки целочисленной арифметики. Например, они практически никогда не дают истинное частное $\lfloor x/y \rfloor$ и истинный остаток $x \bmod y$, если x отрицательно или y отрицательно; они часто отбрасывают верхнюю часть произведения. Они не рассматривают сдвиги влево и вправо, как строгие эквиваленты умножения и деления на 2. Иногда в них деление вообще не реализовано на аппаратном уровне, а при необходимости выполнить деление они обычно предполагают, что верхняя половина 128-битового делимого равна нулю. Такие ограничения существенно усложняют высокоточные вычисления.
- Коммерческие компьютеры неэффективно выполняют операции FINT и FREM.
- Коммерческие компьютеры не обладают (пока?) мощными операциями MOR и MXOR. Обычно для этого у них предусмотрено около полдюжину специализированных инструкций, которые обрабатывают только самые популярные особые случаи MOR.
- Коммерческие компьютеры редко имеют более 64 регистров общего назначения. 256 регистров MMIX позволяют значительно уменьшить длину программы, поскольку многие переменные и константы программы могут находиться в них, а не в памяти. Более того, стек регистров MMIX обладает большей гибкостью, чем аналогичные механизмы в существующих компьютерах.

Все преимущества MMIX имеют связанные с ними недостатки, поскольку проектирование компьютеров всегда сопряжено с компромиссами. Основная цель проектирования MMIX заключалась в создании максимально простого, лишенного пороков, целостного и перспективного компьютера без большого ущерба для скорости и реализма.

*Ясно слышу я отныне
Четкий пульс своей машины.*

— ВИЛЬЯМ ВОРДСВОРТ (WILLIAM WORDSWORTH),
Созданием зыбкой красоты (She Was a Phantom of Delight) (1804)

Резюме. MMIX — это удобный для программирования компьютер, который оперирует 64-битовыми величинами или октабайтами. Он обладает общими характеристиками так называемых RISC-компьютеров (“reduced instruction set computer” — “вычисления с сокращённым набором команд”). То есть, его инструкции имеют всего несколько разных форматов (OP X, Y, Z or OP X, YZ или OP XYZ), а каждая инструкция либо передает данные между памятью и регистром, либо включает только регистры. В табл. 1 приведены 256 кодов операций (или опкодов) и их принимаемые по умолчанию времена выполнения, а в табл. 2 — специальные регистры, которые порой имеют большое значение.

В следующих упражнениях приводится краткий обзор материала из данного раздела. Большая его часть очень проста и читатели легко смогут справиться с ними.

УПРАЖНЕНИЯ:

1. [00] Число 2009 имеет двоичное представление $(11111011001)_2$, а как оно будет выглядеть в шестнадцатеричном представлении?
2. [05] Какие символы {A, B, C, D, E, F, a, b, c, d, e, f} являются *четными*, если рассматриваются как (a) шестнадцатеричные числа, (b) ASCII-символы?
3. [10] Четырехбитовая величина — полубайт, или шестнадцатеричное число — часто называется *ниблом* (*nibble*). Предложите имя для *двухбитовых* величин, чтобы дополнить систему именования от битов до октабайтов.
4. [15] Килобайт (Кбайт или Кб) равен 1000 байтам, а мегабайт (Мбайт или Мб) равен 1000 Кб. Знаете ли вы официальные обозначения для более крупных величин в байтах?
5. [M13] Если α является произвольной строкой из 0 и 1, то пусть $s(\alpha)$ и $u(\alpha)$ будут целыми числами, которые представляют ее в виде знакового или беззнакового двоичного числа. Докажите, что, если x является произвольным целым числом, то

$$x = s(\alpha) \quad \text{тогда и только тогда} \quad x \equiv u(\alpha) \pmod{2^n} \text{ и } -2^{n-1} \leq x < 2^{n-1},$$

где n — это длина α .

- 6. [M20] Докажите или опровергните следующее правило для отрицания n -битового числа в дополнении до двух: “Дополнить все биты, потом добавить 1.” (Например, $\#0\dots01$ сначала становится $\#f\dots fe$, а потом $\#f\dots ff$; аналогично $\#f\dots ff$ сначала становится $\#0\dots 00$, а потом $\#0\dots 01$.)

7. [M15] Можно ли для LDHT и STHT дать следующие определения

$$s(\$X) \leftarrow s(M_4[A]) \times 2^{32} \quad \text{и} \quad s(M_4[A]) \leftarrow \lfloor s(\$X)/2^{32} \rfloor,$$

рассматривая их как знаковые, а не беззнаковые числа?

8. [10] Если регистры \$Y и \$Z представляют числа между 0 и 1, в которых предполагаемая точка в двоичном представлении числа находится в левой части каждого регистра, то (7) иллюстрирует тот факт, что MULU дает произведение, в котором предполагаемая точка в двоичном представлении числа находится в левой части регистра rH. С другой стороны, предположим, что \$Z является целым числом, в котором предполагаемая точка в двоичном представлении числа находится в правой части регистра, а \$Y является дробной величиной между 0 и 1, как и прежде. В таком случае, где располагается точка в двоичном представлении числа после выполнения операции MULU?

9. [M10] Всегда ли выполняется равенство $s(\$Y) = s(\$X) \cdot s(\$Z) + s(rR)$ после выполнения инструкций DIV \$X, \$Y, \$Z?

10. [M16] Приведите пример переполнения операции DIV.

11. [M16] Верно или нет утверждение: (а) Операция MUL \$X, \$Y, \$Z и операция MULU \$X, \$Y, \$Z дают одинаковый результат в \$X. (б) Если регистр rD содержит нуль, то DIV \$X, \$Y, \$Z и DIVU \$X, \$Y, \$Z дают одинаковый результат в \$X.

► 12. [M20] Хотя ADDU \$X, \$Y, \$Z никогда не приводит к переполнению, порой нужно знать о переносе слева при сложении \$Y и \$Z. Покажите, что перенос можно вычислить с помощью двух инструкций.

13. [M21] Допустим, что MMIX не имеет команды ADD, а имеет только ее беззнаковый вариант ADDU. Как программист узнает о переполнении при вычислении $s(\$Y) + s(\$Z)$?

14. [M21] Допустим, что MMIX не имеет команды SUB, а имеет только ее беззнаковый вариант SUBU. Как программист узнает о переполнении при вычислении $s(\$Y) - s(\$Z)$?

15. [M25] Произведение двух знаковых октабайтов всегда находится в диапазоне от -2^{126} до 2^{126} , а потому его можно представить как знаковую 16-байтовую величину. Объясните: как вычислить старшую половину такого знакового произведения.

16. [M23] Допустим, что MMIX не имеет команды, а имеет только ее беззнаковый вариант MULU. Как программист узнает о переполнении при вычислении $s(\$Y) \times s(\$Z)$?

► 17. [M22] Докажите, что беззнаковое целочисленное деление на 3 всегда можно реализовать умножением: если регистр \$Y содержит произвольное беззнаковое целое значение y и если регистр \$1 содержит константу #aaaa aaaa aaaa aaab, то последовательность команд

MULU \$0, \$Y, \$1; GET \$0, rH; SRU \$X, \$0, 1

приводит к размещению $\lfloor y/3 \rfloor$ в регистре \$X.

18. [M23] В продолжение предыдущего упражнения докажите или опровергните утверждение, что инструкции

MULU \$0, \$Y, \$1; GET \$0, rH; SRU \$X, \$0, 2

приводят к размещению $\lfloor y/5 \rfloor$ в \$X, если \$1 константа.

► 19. [M26] В продолжение упражнений 17 и 18 докажите или опровергните следующее утверждение: беззнаковое целочисленное деление на константу всегда может быть выполнено с помощью “умножения старшей половины”, за которыми следует сдвиг вправо. Точнее говоря, если $2^e < z < 2^{e+1}$, мы можем вычислить $\lfloor y/z \rfloor$, вычисляя $\lfloor ay/2^{64+e} \rfloor$, где $a = \lceil 2^{64+e}/z \rceil$, для $0 \leq y < 2^{64}$.

20. [16] Покажите, что две особым образом выбранные инструкции MMIX смогут выполнить умножение в 25 раз быстрее, чем одна инструкция $MUL \$X, \$Y, 25$, если предположить, что переполнения не происходит.

21. [15] Опишите результат выполнения SL, SLU, SR и SRU, если в регистре \$Z находится беззнаковое значение, равное 64 или больше.

- 22. [15] Г-н Б. К. Далл (B. C. Dull) написал программу, в которой задано ветвление к позиции `Case1`, если знаковое число в регистре \$1 было меньше, чем знаковое число в регистре \$2. Для этого он предложил следующий код: `'SUB $0,$1,$2; BN $0,Case1'`.

Какую ужасную ошибку он допустил? Какой код нужно было использовать вместо этого?

- 23. [10] В продолжение предыдущего упражнения: какой код нужно было бы написать Даллу для организации ветвления, если $s(\$1)$ меньше или равно $s(\$2)$?

24. [M10] Если представить подмножество S множества $\{0, 1, \dots, 63\}$ с помощью битового вектора

$$([0 \in S], [1 \in S], \dots, [63 \in S]),$$

то битовые операции \wedge и \vee correspond соответствуют пересечению множеств $(S \cap T)$ и объединению множеств $(S \cup T)$. Какие битовые операции соответствуют разности множеств $(S \setminus T)$?

25. [10] Расстоянием Хамминга между двумя битовыми векторами называется количество позиций, которыми они отличаются. Покажите, что двух инструкций MMIX достаточно для установки регистра \$X равным расстоянию Хамминга между $v(\$Y)$ и $v(\$Z)$.

26. [10] Предложите эффективный способ вычисления 64-битовых разностей $v(\$X) \leftarrow v(\$Y) \dot{-} v(\$Z)$?

- 27. [20] Покажите, как с помощью BDIF одновременно вычислить максимальное и минимальное значение среди двух восьмибитовых значений: $b(\$X) \leftarrow \max(b(\$Y), b(\$Z))$, $b(\$W) \leftarrow \min(b(\$Y), b(\$Z))$.

28. [16] Как одновременно вычислить восемь абсолютных разностей пикселей $|b(\$Y) - b(\$Z)|$?

29. [21] Операция насыщающего сложения для n -битовых пикселей определяется формулой

$$y \dot{+} z = \min(2^n - 1, y + z).$$

Покажите, что с помощью последовательности из трех инструкций MMIX можно выполнить насыщающее сложение $b(\$X) \leftarrow b(\$Y) \dot{+} b(\$Z)$.

- 30. [25] Допустим, что регистр \$0 содержит восемь ASCII-символов. Найдите последовательность из трех инструкций MMIX, которая вычисляет количество пробелов среди символов. (Предположим, что вспомогательные константы предварительно загружены в других регистрах. Пробел имеет ASCII-код #20.)

31. [22] В продолжение предыдущего упражнения покажите, как можно вычислить количество символов \$0, которые имеют отрицательную четность (т.е. нечетное количество битов 1).

32. [M20] Истинно или ложно утверждение:
если $C = A \circ B$, то $C^T = B^T \circ A^T$. (См. (11).)

33. [20] Найдите кратчайшую последовательность инструкций MMIX, которая выполняет циклический сдвиг вправо регистра из восьми бит. Например, таким образом из `#9e3779b97f4a7c16` можно получить `#169e3779b97f4a7c`.

- 34. [21] Допустим, что восемь ASCII-символов находится в регистре \$Z. Покажите, как можно было бы преобразовать их в соответствующие восемь Unicode-символов (вайдов) с помощью только двух инструкций MMIX чтобы поместить результаты в \$X, и \$Y. Как совершить обратное преобразование в ASCII-символы?
- 35. [22] Покажите, что правильно выбранные инструкции MOR обращают слева направо все 64 бита в данном регистре \$Y.
- 36. [20] С помощью только двух инструкций создайте маску, которая помещает #ff во всех байтовых позициях, если \$Y отличается от \$Z, #00 во всех байтовых позициях, если \$Y равно \$Z.
- 37. [HM30] (Конечные поля.) Как с помощью операции MXOR организовать арифметические действия в поле 256 элементов, где каждый элемент поля должен представляется октабайтом.
- 38. [20] Что делает следующая небольшая программа?

```
SETL $1,0; SR $2,$0,56; ADD $1,$1,$2; SLU $0,$0,8; PBNZ $0,@-4*3.
```

- 39. [20] Какая из следующих эквивалентных последовательностей кода быстрее, если принять во внимание информацию из табл. 1?
 - a) BN \$0,@+4*2; ADDU \$1,\$2,\$3 или ADDU \$4,\$2,\$3; CSNN \$1,\$0,\$4.
 - b) BN \$0,@+4*3; SET \$1,\$2; JMP @+4*2; SET \$1,\$3 или CSNN \$1,\$0,\$2; CSN \$1,\$0,\$3.
 - c) BN \$0,@+4*3; ADDU \$1,\$2,\$3; JMP @+4*2; ADDU \$1,\$4,\$5 или ADDU \$1,\$2,\$3; ADDU \$6,\$4,\$5; CSN \$1,\$0,\$6.
 - d, e, f) То же, что и в (a), (b), и (c), но с PBN вместо BN.
- 40. [10] Что случится, если с помощью G0 перейти к адресу, который не кратен 4?
- 41. [20] Истинны или ложны следующие утверждения:
 - a) Инструкции CSOD \$X,\$Y,0 и ZSEV \$X,\$Y,\$X приводят к одинаковым результатам.
 - b) Инструкции CMPU \$X,\$Y,0 и ZSNZ \$X,\$Y,1 приводят к одинаковым результатам.
 - c) Инструкции MOR \$X,\$Y,1 и AND \$X,\$Y,#ff приводят к одинаковым результатам.
 - d) Инструкции MXOR \$X,\$Y,#80 и SR \$X,\$Y,56 приводят к одинаковым результатам.
- 42. [20] Как эффективнее всего установить в регистре \$1 абсолютное значение величины из регистра \$0, если \$0 содержит (a) знаковое целое? (b) число с плавающей точкой?
- 43. [28] При наличии ненулевого октабайта \$Z, как эффективнее всего подсчитать количество ведущих и замыкающих нулей? (Например, #13fd8124f32434a2 имеет три ведущих нуля и один замыкающий нуль.)
- 44. [M25] Предположим, что нужно эмулировать 32-битовую арифметику MMIX. Покажите, что можно легко осуществить операции сложения, вычитания, умножения и деления знаковых тетрабайтов, с переполнением, возникающим, если результат не находится в интервале $[-2^{31} \dots 2^{31}]$.
- 45. [10] Придумайте простой способ запоминания последовательности DVWIOUZX.
- 46. [05] Тетрабайт со всеми нулями #00000000 останавливает работу программы и действует как инструкция MMIX. А что делает тетрабайт со всеми единицами #ffffffff?
- 47. [05] Приведите символьные имена опкодов #DF и #55.
- 48. [11] Выше говорится, что опкоды LD0 и LDOU выполняют одинаковые действия с одинаковой эффективностью, независимо от операндов-байтов X, Y и Z. Какие другие пары опкодов эквивалентны в том же смысле?

- 49. [22] Какие изменения в регистрах и памяти произойдут после выполнения следующей программы “номер 1”? (Например, каково окончательное состояние регистров \$1, гА и гВ?)

```

NEG      $1,1
STCO     1,$1,1
CMPU     $1,$1,1
STB      $1,$1,$1
LDOU     $1,$1,$1
INCH     $1,1
16ADDU   $1,$1,$1
MULU     $1,$1,$1
PUT      rA,1
STW      $1,$1,1
SADD     $1,$1,1
FLOT     $1,$1
PUT      rB,$1
XOR      $1,$1,1
PBOD     $1,0-4*1
NOR      $1,$1,$1
SR       $1,$1,1
SRU      $1,$1,1 ■

```

- 50. [14] Какова длительность выполнения программы из предыдущего упражнения?
51. [14] Преобразуйте программу “номер 1” из упражнения 49 в последовательность тетрабайтов в шестнадцатеричной системе обозначений.
52. [22] Для каждого опкода MMIX определите, существует ли способ установить байты X, Y и Z таким образом, чтобы результат этой инструкции был эквивалентен инструкции SWYM (за исключением того, что время выполнения будет больше). Предположим, что ничего неизвестно о содержимом любого регистра или адреса памяти. Если есть возможность создать холостую операцию, укажите, как это можно осуществить. *Примеры:* INCL является холостой операцией, если $X = 255$ и $Y = Z = 0$. BZ является холостой операцией, если $Y = 0$ и $Z = 1$. MULU никогда не может быть холостой операцией, поскольку влияет на гН.
53. [15] Перечислите все опкоды MMIX, которые могут изменить значение гН.
54. [20] Перечислите все опкоды MMIX, которые могут изменить значение гА.
55. [21] Перечислите все опкоды MMIX, которые могут изменить значение гL.
- 56. [28] По адресу #2000 0000 0000 0000 находится знаковое целое число, x . Создайте две программы, которые вычисляют x^{13} в регистре \$0. Одна программа должна использовать минимальное количество адресов памяти MMIX, а другая должна использовать минимальное время. Допустим, что x^{13} помещается в одном октабайте и все необходимые константы уже загружены в глобальные регистры.
- 57. [20] Когда программа изменяет одну или несколько своих инструкций в памяти, то такой код называется *самоизменяющимся кодом*. В MMIX необходимо выполнять команду SYNCID перед выполнением таких самоизменяющихся команд. Объясните, почему самоизменяющийся код обычно нежелательно применять в современных компьютерах.
58. [50] Напишите книгу об операционных системах, которые включают полное ядро MMIX для архитектуры MMIX.

Эти парни mtiх-шируют практически всё.

— У. РЭНДОЛЬФ (V. RANDOLPH) и ДЖ. П. УИЛСОН (G. P. WILSON),
В глубинке (Down in the Holler) (1953)

1.3.2'. Язык ассемблера компьютера ММІХ

Для создания программ для компьютера ММІХ используется символьный язык, который значительно упрощает операции чтения и записи, а также предохраняют программиста от излишних хлопот, связанных с рутинными деталями, которые часто приводят к необязательным ошибкам. Язык ассемблера ММІХАL (“ММІХ Assembly Language”) является расширением системы обозначений, которая используется для инструкций в предыдущем разделе. Ее основными компонентами являются необязательное использование символьных имен для чисел и поле **МЕТКА** для связи имен с адресами памяти и номерами регистров.

Чтобы понять логику ММІХАL, рассмотрим следующий простой пример. Следующий код является частью более крупной программы, а именно подпрограммы поиска максимального значения среди n элементов $X[1], \dots, X[n]$, согласно алгоритму 1.2.10М.

Программа М (*Найти максимальное значение*). В исходном состоянии n находится в регистре \$0, а адрес $X[0]$ — в регистре x0, а глобальный регистр определяется в другом месте.

Код ассемблера	Строка	МЕТКА	ОП	EXPR	Повтор	Примечания
	01	j	IS	\$0		j
	02	m	IS	\$1		m
	03	kk	IS	\$2		$8k$
	04	xk	IS	\$3		$X[k]$
	05	t	IS	\$255		Временное хранилище
	06		LOC	#100		
*100: #39 02 00 03	07	Maximum	SL	kk,\$0,3	1	<u>M1. Инициализировать.</u> $k \leftarrow n, j \leftarrow n$.
*104: #8c 01 fe 02	08		LDO	m,x0,kk	1	$m \leftarrow X[n]$.
*108: #f0 00 00 06	09		JMP	DecrK	1	В M2 с $k \leftarrow n - 1$.
*10c: #8c 03 fe 02	10	Loop	LDO	xk,x0,kk	$n - 1$	<u>M3. Сравнить.</u>
*110: #30 ff 03 01	11		CMP	t,xk,m	$n - 1$	$t \leftarrow [X[k] > m] - [X[k] < m]$.
*114: #5c ff 00 03	12		PBNP	t,DecrK	$n - 1$	В M5, если $X[k] \leq m$.
*118: #c1 01 03 00	13	ChangeM	SET	m,xk	A	<u>M4. Заменить m.</u> $m \leftarrow X[k]$.
*11c: #3d 00 02 03	14		SR	j,kk,3	A	$j \leftarrow k$.
*120: #25 02 02 08	15	DecrK	SUB	kk,kk,8	n	<u>M5. Уменьшить k.</u> $k \leftarrow k - 1$.
*124: #55 02 ff fa	16		PBP	kk,Loop	n	<u>M2. Все проверено?</u> В M3, если $k > 0$.
*128: #f8 02 00 00	17		POP	2,0	1	Возврат в основную программу. ■

Эта программа демонстрирует сразу несколько особенностей.

а) Столбцы с надписями “МЕТКА”, “ОП” и “EXPR” представляют особый интерес. Они содержат программу на языке ассемблера ММІХАL, которая подробно рассматривается ниже.

б) Столбец с надписью “Код ассемблера” содержит фактический цифровой машинный язык, который соответствует программе ММІХАL. ММІХАL создавался с такой целью, чтобы любая программа ММІХАL легко транслировалась в цифровой машинный язык. Трансляция обычно выполняется другой программой, которая называется *программой ассемблирования*, или *ассемблером*. Таким образом, программисты могут создавать любые программы на машинном языке с помощью ММІХАL, не беспокоясь о ручном определении соответствующих эквивалентных цифровых кодов. Практически все программы ММІХ в этой книге созданы с помощью ММІХАL.

с) Столбец с надписью “Строка” не является значительной частью программы MMIXAL. Он включен в примеры применения MMIXAL в данной книге, чтобы легко обращаться к любой части программы.

д) Столбец с надписью “Примечания” содержит пояснения к программе и ссылки на этапы алгоритма 1.2.10M. Читателю рекомендуется сравнить этот алгоритм с приведенной выше программой. Обратите внимание, что при транскрипции (преобразования) алгоритма в код MMIX были допущены некоторые “программистские вольности”: например, этап M2 пришлось расположить в самом конце.

е) Столбец с надписью “Повтор” будет использоваться во многих программах MMIX в данной книге. Он содержит *профиль*, т.е. сколько раз выполняется инструкция в данной строке во время работы программы. Например, инструкция в строке 10 выполняется $n - 1$ раз. На основании этой информации можно определить время выполнения подпрограммы, оно равно $n\mu + (5n + 4A + 5)\nu$, где A — величина, которая подробно анализировалась в разделе 1.2.10. (Инструкция PBNP выполняется за время $(n - 1 + 2A)\nu$.)

Рассмотрим теперь ассемблерную часть MMIXAL программы M. Строка 01, ‘j IS \$0’ указывает, что символ j обозначает регистр \$0, а строки 02–05 аналогичны ей. Результат выполнения строк 01 и 03 виден в строке 14, где представлен численный эквивалент инструкции ‘SR j, kk, 3’ в виде #3d 00 02 03, т.е., ‘SR \$0, \$2, 3’.

Строка 06 гласит: положения следующих строк должны быть выбраны последовательно, начиная с #100. Следовательно, символ Maximum, который находится в поле МЕТКА строки 07, становится эквивалентным значению #100, а символ Loop в строке 10 находится на три тетрабайта дальше, что эквивалентно #10с.

В строках от 07 до 17 поле ОП содержит символьные имена инструкций MMIX: SL, LD0 и т.д. Но символьные имена IS и LOC, которые находятся в столбце ОП строк 01–06, несколько отличаются. IS и LOC называются *псевдооператорами*, поскольку они являются операторами MMIXAL, но не являются операторами MMIX. Псевдооператоры дают специальную информацию о символьной программе, не являясь инструкциями самой программы. Например, строка ‘j IS \$0’ сообщает *только* о программе M и не указывает, что какая-либо переменная равняется содержимому регистра \$0 во время выполнения программы. Обратите внимание, что для строк 01–06 не определены никакие инструкции ассемблера.

Строка 07 означает “сдвиг влево”: $k \leftarrow n$ за счет $kk \leftarrow 8n$. Эта программа работает с величиной $8k$, а не с k , поскольку $8k$ требуется для адресов октабайтов в строках 08 и 10.

Строка 09 передает управление строке 15. Ассемблер, зная, что эта инструкция JMP находится по адресу #108 и что DescrK эквивалентно #120, вычисляет относительный отступ $(\#120 - \#108)/4 = 6$. Аналогично, относительные адреса вычисляются для команд ветвления (условного перехода) в строках 12 и 16.

Остальная часть символьного кода очевидна и не требует пояснений. Как уже упоминалось ранее, программа M является частью более крупной программы. Например, последовательность команд

```
SET    $2,100
PUSHJ  $1,Maximum
STO    $1,Max
```

означает передачу управления программе М с n равным 100. Программа М найдет наибольший элемент множества $X[1], \dots, X[100]$ и вернет управление инструкции 'STO \$1,Max' с максимальным значением в \$1 и его адресом j в \$2. (См. упражнение 3.)

Рассмотрим теперь *полную* программу, а не подпрограмму. Следующая программа Hello напечатает широко известное сообщение 'Hello, world' и прекратит свою работу.

Программа Н (*Приветствуем мир*).

Код ассемблера	Строка	МЕТКА	ОП	EXPR	Примечания
	01	argv	IS	\$1	Вектор аргументов.
	02		LOC	#100	
*100: #8f ff 01 00	03	Main	LD0U	\$255,argv,0	\$255 ← адрес имени программы.
*104: #00 00 07 01	04		TRAP	0,Fputs,StdOut	Печать имени.
*108: #f4 ff 00 03	05		GETA	\$255,String	\$255 ← адрес строки ", world".
*10c: #00 00 07 01	06		TRAP	0,Fputs,StdOut	Печать этой строки.
*110: #00 00 00 00	07		TRAP	0,Halt,0	Остановка.
*114: #2c 20 77 6f	08	String	BYTE	", world",#a,0	Строка символов
*118: #72 6c 64 0a	09				с новой строкой
*11c: #00	10				и конечным элементом. █

Читатели, имеющие доступ к ассемблеру и симулятору MMIX, могут создать файл со столбцами МЕТКА ОП EXPR программы Н. Сохраните его под именем 'Hello.mms' и ассемблируйте, например с помощью команды 'mmixal Hello.mms'. (Ассемблер создаст файл 'Hello.mmo', где расширение .mms означает "MMIX symbolic" ("MMIX символьный"), а расширение .mmo означает "MMIX object" ("MMIX объектный").) Теперь можно вызвать симулятор с помощью команды 'mmix Hello'.

Симулятор MMIX реализует некоторые простейшие компоненты гипотетической операционной системы NNIX. Например, операционная система NNIX с помощью командной строки

foo bar xzyzy (1)

запускает объектный файл foo.mmo. Аналогичное поведение можно имитировать с помощью симулятора и команды 'mmix <параметры> foo bar xzyzy', где <параметры> — это последовательность нулей или специальных символов. Например, параметр -P печатает профиль программы после ее остановки.

Программа MMIX всегда начинается с символьного адреса Main. В этот момент регистр \$0 содержит количество *аргументов командной строки*, т.е. количество слов в командной строке. Регистр \$1 содержит адрес памяти первого такого аргумента, который всегда является именем программы. Операционная система располагает все аргументы в последовательных октабайтах, начиная с адреса \$1 и заканчивая октабайтом из нулей. Каждый аргумент представляется в виде *строки*, которая является адресом (пустого или непустого) множества ненулевых байтов (*символов* этой строки) с конечным нулевым байтом.

Например, командная строка (1) означает наличие значения 3 в \$0, т.е.:

\$1 = #400000000000000008	Указатель на первую строку
M ₈ [#400000000000000008] = #400000000000000028	Первый аргумент, строка "foo"
M ₈ [#400000000000000010] = #400000000000000030	Второй аргумент, строка "bar"
M ₈ [#400000000000000018] = #400000000000000038	Третий аргумент, строка "xyzyy"
M ₈ [#400000000000000020] = #000000000000000000	Нулевой указатель после последнего аргумента
M ₈ [#400000000000000028] = #666f6f000000000000	'f','o','o',0,0,0,0,0
M ₈ [#400000000000000030] = #626172000000000000	'b','a','r',0,0,0,0,0
M ₈ [#400000000000000038] = #78797a7a7900000000	'x','y','z','z','y',0,0,0

NNIX задает строку каждого аргумента такой, что его символы начинаются с границы октабайта. Хотя, вообще говоря, строки могут начинаться в любом месте внутри октабайта.

Первая инструкция в программе Н, в строке 03, размещает указатель строки M₈[\$1] в регистр \$255. Эта строка содержит имя программы "Hello". Строка 04 содержит специальную инструкцию TRAP, которая предписывает операционной системе разместить \$255 в *стандартном устройстве вывода*, которым является файл. Аналогично, строки 05 и 06 предписывают NNIX разместить строку ", world" и символ новой строки в стандартном устройстве вывода. Согласно определению, символ Fputs равен 7, а символ StdOut равен 1. Строка 07, "TRAP 0,Halt,0", представляет собой обычный способ завершения работы программы. Все специальные команды TRAP подробно рассматриваются в конце этого раздела.

Символы вывода строк в 05 и 06 генерируются командой BYTE в строке 08. BYTE является псевдооператором, а не оператором MMIX. Но BYTE отличается от псевдооператоров IS и LOC, поскольку он ассемблирует данные в память. Вообще говоря, BYTE ассемблирует последовательность выражений в однобайтовые константы. Конструкция ", world" в строке 08 является сокращенной формой записи в MMIXAL для списка

‘,’,’,’,’w’,’o’,’r’,’l’,’d’

из семи односимвольных констант. Константа #a в строке 08 является ASCII-символом *новой строки*, которая создает новую строку в файле вывода. Концевой символ ‘,0’ в строке 08 завершает строку символов. Таким образом, строка 08 содержит список девяти выражений, которые ведут к девяти байтам, показанным слева в строках 08–10.

В третьем примере приводится несколько компонентов языка ассемблера. Его цель заключается в вычислении и печати таблицы, содержащей первые 500 простых чисел с 10 столбцами по 50 чисел в каждом. Таблица будет иметь следующий вид, если стандартным устройством вывода для нашей программы является текстовый файл:

Первые 500 простых чисел									
0002	0233	0547	0877	1229	1597	1993	2371	2749	3187
0003	0239	0557	0881	1231	1601	1997	2377	2753	3191
0005	0241	0563	0883	1237	1607	1999	2381	2767	3203
⋮									⋮
0229	0541	0863	1223	1583	1987	2357	2741	3181	3571

Для реализации этой программы используется следующий метод.

Алгоритм Р (*Печать таблицы 500 простых чисел*). Этот алгоритм имеет две различные части: этапы P1–P8 подготавливают внутреннюю таблицу 500 простых чисел, а этапы P9–P11 печатают ответ в представленном выше формате.

- P1.** [Начало таблицы.] Установить $\text{PRIME}[1] \leftarrow 2$, $n \leftarrow 3$, $j \leftarrow 1$. (На следующих этапах n перебирает все нечетные числа, которые являются кандидатами в простые числа; а j отслеживает количество найденных ранее простых чисел.)
- P2.** [n является простым числом.] Установить $j \leftarrow j + 1$, $\text{PRIME}[j] \leftarrow n$.
- P3.** [Найдено 500 простых чисел?] Если $j = 500$, перейти к этапу P9.
- P4.** [Увеличить n .] Установить $n \leftarrow n + 2$.
- P5.** [$k \leftarrow 2$.] Установить $k \leftarrow 2$. ($\text{PRIME}[k]$ принимает значения всех n возможных простых делителей.)
- P6.** [$\text{PRIME}[k] \setminus n$?] Поделить n на $\text{PRIME}[k]$; пусть q является частным, а r является остатком от деления. Если $r = 0$ (следовательно, n не является простым числом), то перейти к этапу P4.
- P7.** [Большое значение $\text{PRIME}[k]$?] Если $q \leq \text{PRIME}[k]$, то перейти к этапу P2. (В таком случае n должно быть простым числом. Доказательство этого факта представляет интерес и немного необычно — см. упр. 11.)
- P8.** [Увеличить k .] Увеличить k на 1, а потом перейти к этапу P6.
- P9.** [Печать заголовка.] Теперь можно напечатать заголовок таблицы и установить $m \leftarrow 1$.
- P10.** [Печать строки.] Вывести строку с $\text{PRIME}[m]$, $\text{PRIME}[50 + m]$, ..., $\text{PRIME}[450 + m]$ в соответствующем формате.
- P11.** [Напечатано 500 простых чисел?] Увеличить m на 1. Если $m \leq 50$, вернуться к P10, а в противном случае завершить работу. ■

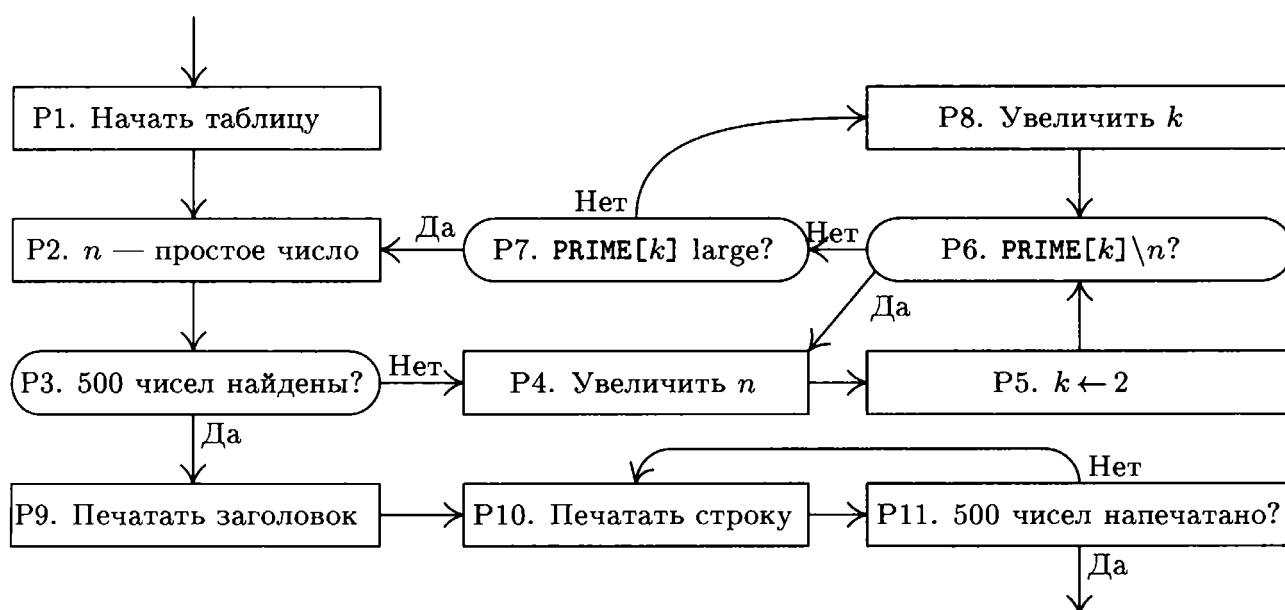


Рис. 14. Алгоритм Р.

Программа Р (*Печать таблицы 500 простых чисел*). Эта программа написана в несколько неуклюжем стиле, чтобы проиллюстрировать большую часть компонентов MMIXAL в одной программе.

01	%	Пример программы ...	Таблица простых чисел	
02	L	IS	500	Количество искомых простых чисел.
03	t	IS	\$255	Временное место хранения.
04	n	GREG	0	Кандидат в простое число.
05	q	GREG	0	Частное.
06	r	GREG	0	Остаток.
07	jj	GREG	0	Индекс PRIME[j].
08	kk	GREG	0	Индекс PRIME[k].
09	pk	GREG	0	Значение PRIME[k].
10	mm	IS	kk	Индекс выводимых строк.
11		LOC	Data_Segment	
12	PRIME1	WYDE	2	PRIME[1] = 2
13	`	LOC	PRIME1+2*L	
14	ptop	GREG	@	Адрес PRIME[501].
15	j0	GREG	PRIME1+2-@	Исходное значение jj.
16	BUF	OCTA	0	Место для строки в десятичном формате.
17				
18		LOC	#100	
19	Main	SET	n,3	<u>P1. Начало таблицы.</u> $n \leftarrow 3$.
20		SET	jj,j0	$j \leftarrow 1$.
21	2H	STWU	n,ptop,jj	<u>P2. n является простым числом.</u> $PRIME[j+1] \leftarrow n$.
22		INCL	jj,2	$j \leftarrow j + 1$.
23	3H	BZ	jj,2F	<u>P3. Найдено 500 простых чисел?</u>
24	4H	INCL	n,2	<u>P4. Увеличить n.</u>
25	5H	SET	kk,j0	<u>P5. $k \leftarrow 2$.</u>
26	6H	LDWU	pk,ptop,kk	<u>P6. PRIME[k] \n n?</u>
27		DIV	q,n,pk	$q \leftarrow \lfloor n/PRIME[k] \rfloor$.
28		GET	r,rR	$r \leftarrow n \bmod PRIME[k]$.
29		BZ	r,4B	Перейти к P4, если $r = 0$.
30	7H	CMP	t,q,pk	<u>P7. Значение PRIME[k] большое?</u>
31		BNP	t,2B	Перейти к P2, если $q \leq PRIME[k]$.
32	8H	INCL	kk,2	<u>P8. Увеличить k.</u> $k \leftarrow k + 1$.
33		JMP	6B	Перейти к P6.
34		GREG	@	Базовый адрес.
35	Title	BYTE	"Первые 500 простых чисел"	
36	NewLn	BYTE	#a,0	Новая строка и конец строки.
37	Blanks	BYTE	" ",0	Строка из трех пробелов.
38	2H	LDA	t,Title	<u>P9. Печать заголовка.</u>
39		TRAP	0,Fputs,StdOut	
40		NEG	mm,2	Инициализировать m , задавая $mm \leftarrow -2$.
41	3H	ADD	mm,mm,j0	<u>P10. Печать строки.</u>
42		LDA	t,Blanks	Вывод " ".
43		TRAP	0,Fputs,StdOut	
44	2H	LDWU	pk,ptop,mm	$pk \leftarrow$ печатаемое простое число.
45	0H	GREG	#2030303030000000	" 0000",0,0,0

46	STOU	0B,BUF	Подготовить буфер для преобразования в десятичный формат.
47	LDA	t,BUF+4	$t \leftarrow$ место цифр.
48	1H	DIV	$pk \leftarrow \lfloor pk/10 \rfloor$.
49	GET	r,rR	$r \leftarrow$ следующая цифра.
50	INCL	r,'0'	$r \leftarrow$ ASCII-символ r .
51	STBU	r,t,0	Сохранить r в буфере.
52	SUB	t,t,1	Сдвинуть один байт влево.
53	PBNZ	pk,1B	Повторить для оставшихся цифр.
54	LDA	t,BUF	Вывести " " и четыре цифры.
55	TRAP	0,Fputs,StdOut	
56	INCL	mm,2*L/10	Продвинуться на 50 вайд.
57	PBN	mm,2B	
58	LDA	t,NewLn	Вывести новую строку.
59	TRAP	0,Fputs,StdOut	
60	CMP	t,mm,2*(L/10-1)	<u>P11. Выведено 500 простых чисел?</u>
61	PBNZ	t,3B	Перейти к этапу P10, если не выведено.
62	TRAP	0,Halt,0	■

Здесь следует отметить следующие интересные особенности этой программы.

1. Строка 01 начинается со знака процентов, а строка 17 пуста. Эти строки “комментариев” предназначены для размещения пояснений и никак не влияют на ассемблированную программу.

Остальные строки содержат три поля **МЕТКА**, **ОП** и **EXPR**, разделенные пробелами. Поле **EXPR** содержит одно или более символьных выражений, разделенных запятыми. Комментарии могут располагаться после поля **EXPR**.

2. Как и в программе **M**, псевдооператор **IS** задает эквивалент символа. Например, в строке 02 эквивалентом **L** задано значение 500, которое определяет количество вычисляемых простых чисел. Обратите внимание, что в строке 03 эквивалентом **t** является \$255, т.е. *номер регистра*, а эквивалентом **L** является 500, т.е. *просто число*. Одни символы имеют в качестве эквивалентов номера регистров от \$0 до \$255, а другие — просто октабайты. Здесь символьные имена с прописным начальным символом обозначают регистры, а символьные имена с заглавным начальным символом — просто значения, хотя в языке **MMIXAL** не обязательно соблюдать это правило.

3. Псевдооператор **GREG** в строке 04 выделяет *глобальный регистр*. Регистр \$255 всегда является глобальным. Первый вызов **GREG** приводит к тому, что регистр \$254 становится глобальным, а следующий вызов **GREG** делает глобальным регистр \$253 и т.д. Следовательно, в строках 04–09 определяются шесть глобальных регистров, причем символы **n**, **q**, **r**, **jj**, **kk**, **pk** становятся эквивалентами регистров \$254, \$253, \$252, \$251, \$250, \$249. В строке 10 **mm** становится эквивалентом \$250.

Если поле **EXPR** псевдооператора **GREG** равно нулю, как в строках 04–09, то предполагается, что глобальный регистр имеет динамически изменяющееся значение во время работы программы.

Но если это поле содержит ненулевое выражение, как в строках 14, 15, 34 и 45, то предполагается, что глобальный регистр постоянен во время работы программы. В **MMIXAL** такие глобальные регистры используются как *базовые адреса*

при обращении к памяти с помощью соответствующих инструкций. Например, рассмотрим инструкцию "LDA t,BUF+4" в строке 47. MMIXAL может обнаружить, что глобальный регистр rtop содержит адрес BUF. Следовательно "LDA t,BUF+4" может быть представлена в виде "LDA t,rtop,4". Аналогично, инструкции LDA в строках 38, 42 и 58 используют безымянные базовые адреса, введенные инструкцией "GREG @" в строке 34. (Как известно из раздела 1.3.1', символ @ обозначает текущее положение.)

4. Добротный язык ассемблера должен имитировать образ *мышления* программиста. Одним из примеров такой философии является автоматическое выделение глобальных регистров и базовых адресов. Другой пример заключается в использовании *локальных символов*, например символа 2H, который появляется в поле метки строк 21, 38 и 44.

Локальные символы — это специальные символы, чьи эквиваленты могут *переопределяться* произвольное количество раз. Например, глобальный символ PRIME1 имеет лишь одно значение во всей программе, и, если он появляется в поля метки нескольких строк, то ассемблер укажет на ошибку, но локальные символы имеют другую природу. Можно, например, создать локальные символы 2H ("2 here" - "2 здесь") в поле МЕТКА, 2F ("2 forward" - "2 вперед") или 2B ("2 backward" - "2 назад") в поле EXPR строки языка MMIXAL:

2B означает ближайшую *предыдущую* метку 2H;

2F означает ближайшую *следующую* метку 2H.

Таким образом, 2F в строке 23 относится к строке 38; 2B в строке 31 относится к строке 21; а 2B в строке 57 относится к строке 44. Символы 2F и 2B никогда не ссылаются на *собственную* строку. Например, следующие инструкции языка MMIXAL

```
2H    IS    $10
2H    BZ    2B,2F
2H    IS    2B-4
```

фактически эквивалентны одной инструкции

```
BZ    $10,@-4.
```

Символы 2F и 2B никогда не следует использовать в поле МЕТКА. Символ 2H никогда не следует использовать в поле EXPR. Если 2B находится перед любым появлением символа 2H, то он обозначает нуль. Существует 10 локальных символов, которые можно получить заменой '2' в данных примерах любой цифрой от 0 до 9.

Идея локальных символов была предложена М. Э. Конвэем (М. Е. Conway) в 1958 году, в программе на языке ассемблера для компьютера UNIVAC I. Локальные символы освобождают нас от обязательства выбирать символьные имена, когда нужно лишь сослаться на инструкцию в нескольких строках далее. Часто бывает так, что нет подходящего имени для ближайших положений, поэтому программистам приходится вводить бессмысленные символы, как, например, X1, X2, X3 и т.д., с потенциальной угрозой их дублирования

5. Ссылка на Data_Segment в строке 11 представляет еще одну новую идею. В большинстве реализаций MMIX виртуальное адресное пространство из 2^{64} байтов разбито на две части, которые называются *пользовательским пространством*

(адреса `#0000000000000000 .. #7fffffffffffffff`) и *пространством ядра* (адреса `#8000000000000000 .. #fffffffffffffff`). “Отрицательные” адреса пространства ядра резервируются для операционной системы.

Пользовательское пространство далее делится на четыре сегмента по 2^{61} байтов каждый. Сначала идет *текстовый сегмент*, где обычно располагается пользовательская программа. Затем идет *сегмент данных*, который начинается с виртуального адреса `#2000000000000000` и предназначен для переменных, для которых память выделяется раз и навсегда ассемблером, а для других переменных память выделяется пользователем без помощи системной библиотеки.

Далее идет *сегмент пула*, который начинается с адреса `#4000000000000000` и предназначен для аргументов командной строки и других динамически выделяемых данных. Наконец, после них располагается *сегмент стека*, который начинается с адреса `#6000000000000000` и используется аппаратным обеспечением MMIX для стека регистров, управляемого операторами PUSH, POP, SAVE и UNSAVE. Три символа:

```
Data_Segment = #2000000000000000,
Pool_Segment = #4000000000000000,
Stack_Segment = #6000000000000000,
```

предназначены в MMIXAL для работы с сегментами. Никакой код не может ассемблироваться в сегменте пула или сегменте стека, хотя программа может ссылаться на данные в них. Ссылки на адреса в начале сегмента могут быть более эффективны, чем ссылки на адреса в конце. Например, MMIX не всегда может получить доступ к последнему байту текстового сегмента, `M[#1fffffffffffffff]`, с той же скоростью чтения, что и скорость чтения первого байта сегмента данных.

Наши программы для MMIX всегда рассматривают текстовый сегмент в режиме *только для чтения*: все адреса памяти меньше, чем `#2000000000000000` остаются постоянными сразу после ассемблирования и загрузки. Следовательно, программа P размещает таблицу простых чисел и буфер вывода в сегменте данных.

6. Текстовый сегмент и сегмент данных пусты в начале работы программы, за исключением инструкций и данных, которые загружены в соответствии со спецификацией MMIXAL. Если два или более байтов данных предназначены для одной ячейки памяти, то загрузчик заполнит ее результатом операции побитового исключительного-ИЛИ.

7. Символьное выражение ‘PRIME1+2*L’ в строке 13 означает, что MMIXAL обладает возможностями выполнять арифметические действия с октабайтами. См. также еще один пример ‘2*(L/10-1)’ в строке 60.

8. В качестве заключительного замечания в отношении программы P следует отметить, что ее инструкции организованы так, что регистры отсчитываются до нуля и проверяются на ноль тогда, когда это возможно. Например, регистр `jj` содержит значение, связанное с положительной переменной `j` алгоритма P, но `jj` обычно отрицательно. Такое изменение упрощает работу компьютера по определению момента достижения `j` значения 500 (строка 23). В этом отношении строки 40–61 представляют особый интерес, хотя, на первый взгляд, они выглядят несколько сложно. Процедура двоично-десятичного преобразования в строках 45–55 основана

на делении на 10. Она проста, но не максимально эффективна. Более эффективные методы рассматриваются в разделе 4.4.

Следует отметить некоторые статистические параметры работы программы Р. Инструкция деления в строке 27 выполняется 9538 раз. Общее время выполнения этапов Р1–Р8 (строки 19–33) равно $10036\mu + 641543\nu$; время выполнения этапов Р9–Р11 равно $2804\mu + 124559\nu$, не считая времени, которое требуется операционной системе для обработки запросов TRAP.

Резюме. Теперь, после демонстрации приведенных выше трех примеров программ на языке MMIXAL, можно приступить к более подробному обсуждению правил, наблюдая за теми действиями, которые *нельзя* выполнить. Приведенные ниже сравнительно простые правила определяют язык MMIXAL.

1. *Символ* — это строка букв и/или цифр, которая начинается с буквы.

Символ подчеркивания '_' считается буквой в данном определении. Аналогично, считаются буквами все символы Unicode, чей код больше 126. *Примеры:* PRIME1, Data_Segment, Main, __, pâté.

Специальные конструкции dH , dF и dB , где d представляет собой одну цифру, заменяются уникальными символами в соответствии с соглашением о “локальных символах”, которое упомянуто выше.

2. *Константа* — это:

- a) либо *десятичная константа*, состоящая из одной или нескольких десятичных цифр $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, представляющих беззнаковый октабайт с основанием 10;
- b) либо *шестнадцатеричная константа*, состоящая из диеза # и одной или нескольких шестнадцатеричных цифр $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F\}$, представляющих беззнаковый октабайт с основанием 16;
- c) либо *символьная константа*, состоящая из кавычки ' и любого символа (кроме символа новой строки), за которым следует другая кавычка ' ; она представляет ASCII- или Unicode-значение символа в кавычках.

Примеры: 65, #41, 'A', 39, #27, ' ', 31639, #7B97, '算'.

Строковая константа — это двойная кавычка ", за которыми следует один или несколько символов, кроме символа новой строки или двойной кавычки, за которыми следует еще одна двойная кавычка ". Эта конструкция эквивалентна последовательности символьных констант из отдельных символов, разделенных запятыми.

3. Каждое появление символа в программе MMIXAL называется либо “определенным символом”, либо “опережающей ссылкой”. *Определенный символ* — это символ, который находится в поле МЕТКА предыдущей строки данной программы MMIXAL. *Опережающей ссылкой* называется символ, который еще не был определен.

Некоторые символы, например rR, ROUND_NEAR, V_BIT, W_handler и Fputs, предварительно определены, поскольку они ссылаются на константы, связанные с аппаратным обеспечением MMIX или с операционной системой. Такие символы могут

быть переопределены, поскольку в MMIXAL не допускается, что любому программисту известны все их имена. Но ни один символ не может использоваться как метка более одного раза.

Каждый определенный символ имеет эквивалентное значение, которое является либо *чистым* (беззнаковым октабайтом), либо *номером регистра* (\$0, либо \$1, ... либо \$255).

4. *Первичным, или элементарным, выражением* называется:

- a) либо символ;
- b) либо константа;
- c) либо символ @; обозначающий текущее положение,
- d) либо выражение в скобках;
- e) унарный оператор после первичного выражения.

Унарными операторами называются + (подтверждение, которое ничего не делает), - (отрицание, которое вычитает из нуля), ~ (дополнение, которое изменяет все 64 бита) и \$ (регистризация, которая преобразует чистое значение в номер регистра).

5. *Член* — это последовательность из одного или более первичных выражений, разделенных сильными бинарными операторами, а *выражение* — это последовательность из одного или более членов, разделенных слабыми бинарными операторами. *Сильными бинарными операторами* называются * (умножение), / (деление), // (дробное деление), % (взятие остатка от деления), << (сдвиг влево), >> (сдвиг вправо) и & (битовое И). *Слабыми бинарными операторами* называются + (сложение), - (вычитание), | (битовое ИЛИ) и ^ (битовое исключительное-ИЛИ). Эти операторы действуют на беззнаковые октабайты. Оператор $x//y$ обозначает $\lfloor 2^{64}x/y \rfloor$, если $x < y$, и результат не определен, если $x \geq y$. Бинарные операторы одинаковой силы выполняются слева направо, например $a/b/c$ равносильно $(a/b)/c$, $a-b+c$ равносильно $(a-b)+c$.

Пример: $\#ab<<32+k\&\sim(k-1)$ является выражением, т.е. суммой членов $\#ab<<32$ и $k\&\sim(k-1)$. Последний член из двух является битовым И первичных выражений k и $\sim(k-1)$. Последнее первичное выражение из двух является дополнением $(k-1)$, т.е. окруженной скобками разности двух членов k и 1. Член 1 является первичным выражением, а также константой, а точнее десятичной константой. Если символ k является эквивалентом $\#cdef00$, то все выражение $\#ab<<32+k\&\sim(k-1)$ является эквивалентом $\#ab00000100$.

Бинарные операторы можно выполнять только с числами, за исключением случаев типа $\$1 + 2 = \3 и $\$3 - \$1 = 2$. Опережающие ссылки нельзя комбинировать с чем-то еще. Например, не допускается выражение $2F+1$, поскольку $2F$ никогда не относится к определенному символу.

6. *Инструкция* состоит из трех полей:

- a) поле МЕТКА, которое может быть пустым или содержать символ;
- b) поле ОП, которое содержит опкод MMIX или псевдооператор MMIXAL;
- c) поле EXPR, которое содержит список из одного или более выражений, разделенных запятыми. Поле EXPR может быть пустым и в таком случае оно эквивалентно выражению 0.

7. Ассемблирование инструкции выполняется в три этапа:

- а) Текущее положение \mathcal{Q} выравнивается, если это необходимо, за счет увеличения до следующего кратного числа,

8, если ОП является OSTA;
4, если ОП является TETRA или опкодом MMIX;
2, если ОП является WYDE.

- б) Если поле **МЕТКА** содержит символ, то его положение устанавливается в \mathcal{Q} , если только не ОП = IS или ОП = GREG.

- с) Если поле ОП содержит псевдооператор, то см. правило 8. В противном случае поле ОП содержит инструкцию MMIX, поля ОП и EXPR задают тетерабайт (см. раздел 1.3.1'), а \mathcal{Q} продвигается на 4. Опкоды MMIX могут содержать в поле EXPR три, два или только один операнд.

Если поле ОП содержит ADD, то MMIXAL ожидает три операнда и проверит, являются ли первый и второй операнды номерами регистров. Если третий операнд является чистым значением, то MMIXAL изменит опкод #20 ("сложить") на #21 ("сложить немедленно") и проверит, что немедленное значение меньше 256.

Если поле ОП содержит SETH, то MMIXAL ожидает два операнда. Первый операнд должен быть номером регистра, а второй операнд должен быть чистым значением меньше 65536.

Если поле ОП содержит BNZ, то операндов должно быть два: регистр и чистое значение. Чистое значение должно выражаться как относительный адрес, т.е. в виде $\mathcal{Q} + 4k$, где $-65536 \leq k < 65536$.

Любое поле ОП, которое ссылается на память, как LDB или G0, содержит два операнда в виде $\$X, A$ либо три операнда в виде $\$X, \$Y, \$Z$ или $\$X, \Y, Z . Формат двух операндов можно использовать в случаях, когда адрес памяти A выражается в виде суммы $\$Y + Z$ базового адреса и однобайтового значения (см. правило 8(b)).

8. В языке MMIXAL предусмотрены перечисленные ниже псевдооператоры.

- а) ОП = IS: поле EXPR должно содержать одно выражение; а если в поле **МЕТКА** находится символ, то он становится эквивалентом значения этого выражения.
- б) ОП = GREG: поле EXPR должно содержать одно выражение с чистым эквивалентом, x . Если в поле **МЕТКА** находится символ, то он становится эквивалентом номера глобального регистра, который будет содержать x в начале работы программы. Если $x \neq 0$, то значение x считается *базовым адресом*, а программа не может изменить этот глобальный регистр. Если $x = 0$, или если x является *базовым адресом*, который не встречался ранее, то выделяется новый глобальный регистр (максимально возможный).
- с) ОП = LOC: поле EXPR должно содержать одно выражение с чистым эквивалентом, x . Значение \mathcal{Q} становится равным x . Например, инструкция 'T LOC $\mathcal{Q}+1000$ ' означает, что символ T является адресом первого байта в последовательности 1000 байтов, а \mathcal{Q} продвигается к байту вслед за этой последовательностью.
- д) ОП = BYTE, WYDE, TETRA или OSTA: поле EXPR должно содержать список только тех выражений, которые помещаются в рамках 1, 2, 4 или 8 байтов, соответственно.

```

% Пример программы ... Таблица простых чисел
L IS 500           Количество искомых простых чисел
t IS $255         Временное место хранения
n GREG           ;; Кандидат в простое число
q GREG /* Частное */
r GREG // Остаток
jj GREG 0         Индекс PRIME[j]
:
PBN mm,2B
LDA t,NewLn; TRAP 0,Fputs,StdOut
CMP t,mm,2*(L/10-1) ; PBNZ t,3B;      TRAP 0,Halt,0;

```

Рис. 15. Программа P в виде файла: ассемблер допускает разные форматы

9. MMIXAL ограничивает применение опережающих ссылок так, чтобы процесс ассемблирования быстро выполнялся за один проход программы. Опережающая ссылка допускается только:

- a) либо в относительном адресе: в виде операнда JMP или второго операнда перехода, условного перехода, PUSHJ или GETA;
- b) либо в выражении, ассемблированном с ОСТА.

В MMIXAL предусмотрено несколько дополнительных компонентов, относящихся к системному программированию, но которые не рассматриваются здесь. Полное описание всех этих подробностей и логики работы ассемблера можно найти в документации *MMIXware*.

Для представления ассемблеру программы MMIXAL можно использовать произвольный формат (см. рис. 15). Поле **МЕТКА** находится в начале строки и продолжается до первого пробела. Следующий символ (не пробел) обозначает начало поля **ОП**, которое продолжается до следующего пробела и т.д. Строка является комментарием, если первый символ (не пробел) не является буквой или цифрой. В противном случае комментарий начинается после поля **EXPR**. Обратите внимание, что определения GREG для n, q и r на рис. 15 имеют пустое поле **EXPR** (которое эквивалентно одному выражению '0'). Следовательно, комментарии в этих строках нужно обозначить специальным разделителем. Но такой разделитель не нужен для GREG в строке jj, поскольку там явно присутствует поле **EXPR** в виде 0.

Заключительные строки на рис. 15 иллюстрируют тот факт, что в одной строке могут находиться две или более инструкции, если они разделены точкой с запятой. Если инструкция после точки с запятой содержит непустую метку, то эта метка должен следовать сразу после '; '.

Очевидно, что непротиворечивый формат лучше, чем мешанина разных стилей, показанных на рис. 15, поскольку упорядоченные файлы легче читаются, чем хаотические. Однако сам по себе ассемблер достаточно снисходителен, поскольку на его работу не влияет такая случайная неряшливость.

Примитивный ввод и вывод. В заключение этого раздела рассмотрим специальные операторы TRAP симулятора MMIX. Эти операторы обеспечивают базовые функции ввода и вывода, на которых основаны многие операции более высокого

уровня. Последовательность из двух инструкций в виде

SET \$255, <аргумент>; TRAP 0, <функция>, <дескриптор> (2)

обычно используется для вызова такой функции, где <аргумент> указывает на параметр и <дескриптор> идентифицирует соответствующий файл. Например, в программе Н (как и в программе М) команды

GETA \$255, String; TRAP 0, Fputs, StdOut

используются для размещения строки в стандартном устройстве вывода, т.е. файле.

После обработки команды TRAP операционной системой регистр \$255 будет содержать возвращаемое значение. В любом случае это значение отрицательно тогда и только тогда, когда произошла ошибка. В программах Н и Р нет проверки ошибок вывода в файл, поскольку в них предполагается, что правильность или неправильность вывода в стандартное устройство вывода будут сами говорить за себя. Однако обычно процедуры обнаружения и исправления ошибок имеют большое значение в профессиональных программах.

- **Fopen(дескриптор, имя, режим).** Каждая из десяти примитивных команд ввода-вывода применяется к *дескриптору*, который является однобайтовым целым числом. Fopen связывает *дескриптор* с внешним файлом, чье имя задается в строке *имя*, и готовит ввод и/или вывод в этот файл. Третий параметр, *режим*, должен иметь одно из следующих значений: TextRead, TextWrite, BinaryRead, BinaryWrite или BinaryReadWrite, которые предварительно определены в MMIXAL. В трех режимах ...Write игнорируется предыдущее содержимое файла. Если дескриптор успешно открыт, то возвращается значение 0, а в противном случае возвращается значение -1.

Последовательность вызова команды Fopen имеет вид

LDA \$255, Arg; TRAP 0, Fopen, <дескриптор> (3)

где Arg — это двухбайтовая последовательность

Arg OCTA <имя>, <режим>, (4)

которая находится в памяти. Например, для вызова функции Fopen(5, "foo", BinaryWrite) в программе MMIXAL можно поместить

Arg OCTA 1F, BinaryWrite
1H BYTE "foo", 0

в сегмент данных, а затем использовать следующую последовательность:

LDA \$255, Arg; TRAP 0, Fopen, 5.

Таким образом откроется дескриптор 5 для записи в новый файл "foo" в двоичном формате*.

* В разных типах компьютеров используются разные соглашения для текстового и двоичного файла. Каждый симулятор MMIX принимает те соглашения, которые используются текущей операционной системой.

Три дескриптора всегда открыты в начале работы любой программы: для файла стандартного ввода `StdIn` (дескриптор 0) в режиме `TextRead`; для файла стандартного вывода `StdOut` (дескриптор 1) в режиме `TextWrite`; для файла вывода ошибок `StdErr` (дескриптор 2) также в режиме `TextWrite`.

- `Fclose(дескриптор)`. Если *дескриптор* открыт, то `Fclose` закрывает его, т.е. устраняет связь с любым файлом. Здесь так же, если дескриптор успешно открыт, то возвращается значение 0, а если файл уже закрыт или не может быть закрыт, то возвращается значение -1 . Последовательность команд имеет вид

TRAP 0, Fclose, <дескриптор> (5)

поскольку нет необходимости размещать что-либо в \$255.

- `Fread(дескриптор, буфер, размер)`. Файл с указанным дескриптором должен быть открыт в режиме `TextRead`, `BinaryRead` или `BinaryReadWrite`. Байты в количестве, заданном параметром *размер*, считываются из файла в память MMIX, начиная с адреса *буфер*. Если *n* байт успешно считано и сохранено, то возвращается значение $n - \text{размер}$, а если произошла ошибка, то возвращается значение $-1 - \text{размер}$. Последовательность команд имеет вид

LDA \$255, Arg; TRAP 0, Fread, <дескриптор> (6)

с двумя октабайтами для других аргументов

Arg OCTA <буфер>, <размер> (7)

как в (3) и (4).

- `Fgets(дескриптор, буфер, размер)`. Файл с указанным дескриптором должен быть открыт в режиме `TextRead`, `BinaryRead` или `BinaryReadWrite`. Однобайтовые символы считываются из файла в память MMIX, начиная с адреса *буфер* до тех пор, пока не будет считано и сохранено *размер* $- 1$ символов либо пока не будет считан и сохранен символ новой строки. Затем следующий байт в памяти устанавливается в нуль. Если до завершения чтения произойдет ошибка или встретится конец файла, то содержимое памяти становится неопределенным и возвращается значение -1 . В противном случае успешно считывается и сохраняется заданное количество символов.

Последовательность команд та же, что и в (6) и (7), за исключением того, что `Fgets` используется вместо `Fread` в (6).

- `Fgetws(дескриптор, буфер, размер)`. Эта команда аналогична команде `Fgets`, за исключением того, что она применяется к вайдовым, а не к однобайтовым символам. Символы считываются до *размер* $- 1$, символ новой строки имеет вид `#000a`.

- `Fwrite(дескриптор, буфер, размер)`. Файл с указанным дескриптором должен быть открыт в одном из следующих режимов: `TextWrite`, `BinaryWrite` или `BinaryReadWrite`. Байты в количестве, заданном параметром *размер*, записываются из памяти MMIX в файл, начиная с адреса *буфер*. Если *n* байт успешно считано и сохранено, то возвращается значение $n - \text{размер}$. Последовательность команд имеет тот же вид, что и в (3) и (4).

- `Fputs(дескриптор, строка)`. Файл с указанным дескриптором должен быть открыт в одном из следующих режимов: `TextWrite`, `BinaryWrite` или `BinaryReadWrite`.

Однobaйтовые символы записываются из памяти MMIX в файл, начиная с адреса *строка* до первого нулевого байта. Если n байт успешно считано и сохранено, то возвращается значение n , а если произошла ошибка, то возвращается значение -1 . Последовательность команд имеет вид

$$\text{SET } \$255, \langle \text{строка} \rangle; \text{TRAP } 0, \text{Fputs}, \langle \text{дескриптор} \rangle. \quad (8)$$

• **Fputws**(*дескриптор*, *строка*). Эта команда аналогична команде **Fputs**, за исключением того, что она применяется к вайдовым, а не к однobaйтовым символам.

• **Fseek**(*дескриптор*, *отступ*). Файл с указанным дескриптором должен быть открыт в одном из следующих режимов: **BinaryRead**, **BinaryWrite** или **BinaryReadWrite**. Этот оператор указывает на то, что следующий ввод или вывод начнется с места на расстоянии *отступ* байт от начала файла, если $\text{отступ} \geq 0$, или с места на расстоянии $-\text{отступ} - 1$ байт от конца файла, если $\text{отступ} < 0$. (Например, $\text{отступ} = 0$ “перематывает” файл к самому началу, а $\text{отступ} = -1$ “перематывает” файл к самому концу.) Если команда успешно выполнена, то результат равен 0, а если нельзя достигнуть указанное положение, то результат равен -1 . Последовательность команд имеет вид

$$\text{SET } \$255, \langle \text{отступ} \rangle; \text{TRAP } 0, \text{Fseek}, \langle \text{дескриптор} \rangle. \quad (9)$$

В режиме **BinaryReadWrite** для переключения от операции ввода к операции вывода либо, наоборот, от операции вывода к операции ввода, необходимо использовать команду **Fseek**.

• **Ftell**(*дескриптор*). Файл с указанным дескриптором должен быть открыт в одном из следующих режимов: **BinaryRead**, **BinaryWrite** или **BinaryReadWrite**. Этот оператор возвращает текущее положение в файле, измеренное в байтах от начала, либо значение -1 в случае ошибки. Последовательность команд имеет вид

$$\text{TRAP } 0, \text{Ftell}, \langle \text{дескриптор} \rangle. \quad (10)$$

Полное описание всех десяти функций ввода-вывода можно найти в документации **MMIXware**. В **MMIXAL** предварительно определены символы

$$\begin{array}{lll} \text{Fopen} = 1, & \text{Fwrite} = 6, & \text{TextRead} = 0, \\ \text{Fclose} = 2, & \text{Fputs} = 7, & \text{TextWrite} = 1, \\ \text{Fread} = 3, & \text{Fputws} = 8, & \text{BinaryRead} = 2, \\ \text{Fgets} = 4, & \text{Fseek} = 9, & \text{BinaryWrite} = 3, \\ \text{Fgetws} = 5, & \text{Ftell} = 10, & \text{BinaryReadWrite} = 4 \end{array} \quad (11)$$

и символ **Halt** = 0.

УПРАЖНЕНИЯ — первая часть:

1. [05] (а) Что означает ‘4В’ в строке 29 программы Р? (б) Будет ли работать программа, если метку в строке 24 заменить на ‘2Н’, а в поле **EXPR** строки 29 использовать ‘**r**, 2В’?

2. [10] Что произойдет, если программа **MMIXAL** будет содержать несколько экземпляров строки

$$9\text{H} \quad \text{IS} \quad 9\text{B}+1$$

и ни одного упоминания **9H**.

- 3. [23] Каким будет результат следующей программы?

```

          LOC      Data_Segment
X0        IS      @
N         IS      100
x0        GREG    X0
(Здесь вставить программу M)
Main      GETA     t,9F; TRAP 0,Fread,StdIn
          SET      $0,N<<3
1H        SR       $2,$0,3; PUSHJ $1,Maximum
          LDO      $3,x0,$0
          SL       $2,$2,3
          STO      $1,x0,$0; STO $3,x0,$2
          SUB      $0,$0,1<<3; PBNZ $0,1B
          GETA     t,9F; TRAP 0,Fwrite,StdOut
          TRAP     0,Halt,0
          9H       OCTA  X0+1<<3,N<<3      █

```

4. [10] Чему равно значение константы #112233445566778899?
5. [11] Что получится в результате 'BYTE 3+"pills"+6'?
- 6. [15] Истинно или ложно утверждение: одна инструкция TETRA (выражение1), (выражение2) всегда приводит к тому же результату, что и пара инструкций TETRA (выражение1); TETRA (выражение2).
7. [05] Студент Джон Х. Квик с удивлением обнаружил, что инструкция GETA \$0,@+1 дает такой же результат, что и GETA \$0,@. Почему в этом нет ничего удивительного?
- 8. [15] Предложите эффективный способ выравнивания текущего положения @ таким образом, чтобы оно стало кратным 16, увеличиваясь на 0..15 в случае необходимости.
9. [10] Как нужно изменить программу P, чтобы она печатала таблицы 600 простых чисел?
- 10. [25] Ассемблируйте программу P вручную. (Для этого потребуется не так много усилий, как может показаться.) Каким будет фактическое численное содержание памяти, соответствующее этой символьной программе?
11. [HM20] (a) Покажите, что каждое непустое число $n > 1$ имеет делитель d с $1 < d \leq \sqrt{n}$. (b) На основе этого факта покажите, что n является простым, если оно удовлетворяет этому условию на этапе P7 алгоритма P.
12. [15] Инструкция GREG в строке 34 программа П определяет базовый адрес, который используется для строковых констант Title, NewLn и Blank в строках 38, 42, и 58. Предложите способ исключения этого дополнительного глобального регистра без замедления работы программы.
13. [20] Символы Unicode позволяют печатать первые 500 простых чисел даже в показанном ниже

أول خمس ميات الأرقام الأولية

٣١٨٧	٢٧٤٩	٢٣٧١	١٩٩٣	١٥٩٧	١٢٢٩	٠٨٧٧	٠٥٤٧	٠٢٣٣	٠٠٠٢
٣١٩١	٢٧٥٣	٢٣٧٧	١٩٩٧	١٦٠١	١٢٣١	٠٨٨١	٠٥٥٧	٠٢٣٩	٠٠٠٣
٣٢٠٣	٢٧٦٧	٢٣٨١	١٩٩٩	١٦٠٧	١٢٣٧	٠٨٨٣	٠٥٦٣	٠٢٤١	٠٠٠٥
⋮									⋮
٣٥٧١	٣١٨١	٢٧٤١	٢٣٥٧	١٩٨٧	١٥٨٣	١٢٢٣	٠٨٦٣	٠٥٤١	٠٢٢٩

“аутентичном” виде арабских цифр. Здесь используются вайдовые символы вместо байтов с преобразованием английских цифр и подстановкой арабско-индийских цифр #0660 – #0669 вместо ASCII-цифр #30 – #39. (В арабской системе письма буквы пишутся справа налево, а числа записываются так, что наименее значимые цифры располагаются справа. Правила двунаправленного представления Unicode автоматически заботятся о необходимых преобразованиях при форматировании вывода.) Как нужно изменить программу P, чтобы добиться этого?

- 14. [21] Измените программу P так, чтобы в ней использовалась арифметика чисел с плавающей запятой для теста на деление на этапе П6. (Инструкция `FREM` всегда дает точный результат.) Используйте \sqrt{n} вместо q на этапе P7. Увеличивают или уменьшают время программы эти изменения?
- 15. [22] Что делает следующая программа? (Не применяйте для этого компьютер, а сообразите самостоятельно!)

```
* Загадочная программа
a   \ GREG   '*'
b     GREG   ' '
c     GREG   Data_Segment
      LOC    #100
Main NEG     $1,1,75
      SET     $2,0
2H   ADD     $3,$1,75
3H   STB     b,c,$2
      ADD     $2,$2,1
      SUB     $3,$3,1
      PBP     $3,3B
      STB     a,c,$2
      INCL    $2,1
      INCL    $1,1
      PBN     $1,2B
      SET     $255,c; TRAP 0,Fputs,StdOut
      TRAP    0,Halt,0 █
```

16. [46] Язык `MMIXAL` задумывался настолько простым и эффективным, чтобы люди могли легко составлять относительно короткие программы на машинном языке для `MMIX`. Более длинные программы обычно создают на языках более высокого уровня, как C или Java, игнорируя подробности на машинном уровне. Иногда возникает необходимость создать крупномасштабную программу специально для конкретного типа компьютеров, причем со строгим контролем над каждой инструкцией. В таких случаях нужно использовать машиноориентированный язык с гораздо более богатой структурой, чем построчный подход традиционного ассемблера.

Спроектируйте и реализуйте язык `PL/MMIX`, аналогичный языку `PL/360` Никласа Вирта (Niklaus Wirth) [*JACM* 15 (1968), 37–74]. Желательно, чтобы ваш язык содержал идеи “грамотного программирования” (literate programming) [D. E. Knuth, *Literate Programming* (1992)].

УПРАЖНЕНИЯ — вторая часть:

Следующие упражнения представляют собой короткие задачи по программированию, представляющие типичные компьютерные приложения и охватывающие широкий диапазон методов.

Читателю предлагается выбрать и решить несколько задач, чтобы получить некоторый опыт применения MMIX, а также приобрести базовые навыки программирования. Эти упражнения можно изучать по мере чтения остальной части главы 1. Ниже перечислены несколько методов программирования, которые здесь применяются.

Таблица переключения для многовариантных решений: упражнение 17.

Вычисления с двумерными массивами: упражнения 18, 28 и 35.

Операции с текстом и строками: упражнения 24, 25, и 35.

Арифметика целых чисел и арифметика с разбиением чисел на целую и дробную части: упражнения 21, 27, 30 и 32.

Элементарная арифметика с числами с плавающей точкой : упражнения 27 и 32.

Подпрограммы: упражнения 23, 24, 32, 33, 34, и 35.

Обработка списков: упражнение 29.

Управление в текущем (реальном) времени: упражнение 34.

Имитация типографского набора: упражнение 35.

Оптимизация циклов и конвейеров: упражнения 23 и 26.

Если в упражнении в этой книге предлагается “создать программу для MMIX” или “создать подпрограмму для MMIX”, то от читателя требуется написать только символьный код MMIXAL. Сам по себе этот код не полон, а рассматривается всего лишь как фрагмент (гипотетической) полной программы. Во фрагменте кода не требуется организовывать ввод или вывод, если данные поставляются извне. Достаточно только указать поля **МЕТКА**, **ОП** и **EXPR** инструкций MMIXAL вместе с соответствующими комментариями. Численный машинный язык, номер строки и столбец “Повтор” (см. программу M) не требуются, если это не оговорено особо, как не требуется метка **Main**.

С другой стороны, если в упражнении предлагается “создать *полную* программу для MMIX”, то это значит, что выполняемая программа должна быть создана на языке MMIXAL, включая метку **Main**. Такие программы следует проверить с помощью ассемблера MMIX и симулятора.

- 17. [25] Регистр \$0 содержит адрес тетрабайта, который предположительно является непривилегированной допустимой инструкцией MMIX. (Т.е. $\$0 \geq 0$, а байты X, Y и Z $M_4[\$0]$ подчиняются всем ограничениям, накладываемым байтом OP, согласно правилам раздела 1.3.1'. Например, допустимая инструкция с опкодом **FIX** будет иметь ограничение $Y \leq \text{ROUND_NEAR}$, а допустимая инструкция с опкодом **PUT** будет иметь ограничение $Y = 0$ и либо $X < 8$, либо $18 < X < 32$. Опкод **LDVTS** всегда является привилегированным, т.е. предназначенным для использования только операционной системой. Но большинство опкодов определяют инструкции, которые являются допустимыми и непривилегированными для всех X, Y и Z.) Напишите подпрограмму MMIX, которая проверяет данный тетрабайт на допустимость в этом смысле. Постарайтесь создать эту программу максимально эффективной.

Замечание: неопытные программисты стремятся решать подобные проблемы, создавая длинную последовательность проверок байта OP, например “**SR op,tetra,24; CMP t,op,#18; BN t,1F; CMP t,op,#98; BN t,2F; ...**”. Этот способ *нельзя* назвать эффективным! Самый оптимальный способ организации многовариантного решения заключается в создании вспомогательной *таблицы*, содержащей информацию о нужной логике. Например, доступ к таблице из 256 октабайтов, по одному для каждого опкода, можно получить с помощью инструкций “**SR t,tetra,21; LD0 t,Table,t**”, за которыми может следовать инструкция **GO**, если нужно выполнять разные типы действий. Использование таблицы часто позволяет существенно ускорить и упростить работу.

- 18. [31] Предположим, что матрица 9×8 знаковых однобайтовых элементов

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{28} \\ a_{91} & a_{92} & a_{93} & a_{98} \end{pmatrix}$$

сохранена так, что a_{ij} располагается по адресу $A + 8i + j$, где A — это константа. В памяти MMIX этот массив предстанет как:

$$\begin{pmatrix} M[A + 9] & M[A + 10] & M[A + 11] & M[A + 16] \\ M[A + 17] & M[A + 18] & M[A + 19] & M[A + 24] \\ M[A + 73] & M[A + 74] & M[A + 75] & M[A + 80] \end{pmatrix}.$$

Матрица $m \times n$ имеет “седловую точку”, если значение в каком-то месте является минимальным по строке и максимальным по столбцу. Иначе говоря, a_{ij} является седловой точкой, если

$$a_{ij} = \min_{1 \leq k \leq n} a_{ik} = \max_{1 \leq k \leq m} a_{kj}.$$

Напишите программу MMIX, которая вычисляет положение седловой точки (если существует хотя бы одна такая точка) или возвращает нуль (при отсутствии седловой точки), а потом помещает это значение в регистр \$0.

19. [M29] Чему равна *вероятность* того, что матрица из предыдущего примера имеет седловую точку, предполагая, что она содержит 72 различных элемента и все 72! перестановки равновероятны? Чему равна аналогичная вероятность, если элементами матрицы являются нули и единицы, а все возможные 2^{72} вариантов такой матрицы являются равновероятными?

20. [HM42] Для упражнения 18 (см. стр. ??) дано два алгоритма решения и подсказка для получения третьего алгоритма решения, однако не ясно, какое из них является эффективным. Проанализируйте оба алгоритма и с помощью ответа к упражнению 19 найдите самый эффективный метод.

21. [25] Возрастающая последовательность всех сокращенных дробей между 0 и 1, которые имеют знаменатели $\leq n$, называется “последовательностью Фэри порядка n ”. Например, последовательностью Фэри порядка 7 является

$$\frac{0}{1}, \frac{1}{7}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{2}{7}, \frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{1}{2}, \frac{4}{7}, \frac{3}{5}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{4}{5}, \frac{6}{7}, \frac{1}{1}.$$

Если обозначить эту последовательность $x_0/y_0, x_1/y_1, x_2/y_2, \dots$, то в упражнении 22 доказывается, что

$$\begin{aligned} x_0 &= 0, & y_0 &= 1; & x_1 &= 1, & y_1 &= n; \\ x_{k+2} &= \lfloor (y_k + n)/y_{k+1} \rfloor x_{k+1} - x_k; \\ y_{k+2} &= \lfloor (y_k + n)/y_{k+1} \rfloor y_{k+1} - y_k. \end{aligned}$$

Создайте подпрограмму MMIX, которая вычисляет последовательность Фэри порядка n , сохраняя значения x_k и y_k в тетрабайтах $X + 4k$ и $Y + 4k$, соответственно. (Общее количество членов в этой последовательности приблизительно равно $3n^2/\pi^2$, т.е. можно предположить, что $n < 2^{32}$.)

22. [M30] (а) Покажите, что числа x_k и y_k определенные рекуррентным соотношением из предыдущего упражнения, удовлетворяют соотношению $x_{k+1}y_k - x_ky_{k+1} = 1$. (б) Покажите, что дроби x_k/y_k являются последовательностью Фэри порядка n , на основе доказанного утверждения из пункта (а).

23. [25] Напишите подпрограмму MMIX, которая задает значения 0 для n последовательных байт в памяти при условии, что начальный адрес находится в регистре \$0, а целое значение $n \geq 0$ в регистре \$1. Попытайтесь создать максимально быструю подпрограмму для достаточно больших значений n . Используйте симулятор конвейера MMIX для получения реалистичной статистики времени выполнения.

- 24. [30] Создайте подпрограмму MMIX, которая копирует строку, начинающуюся с адреса в регистре \$0, в место по адресу в регистре \$1. Строки содержат концевой нулевой символ (т.е. байт, равный нулю). Предполагается, что строка и ее копия в памяти не перекрываются. В подпрограмме должно содержаться минимальное количество обращений к памяти для загрузки и сохранения сразу восьми байтов, чтобы добиться максимально эффективного копирования длинных строк. Сравните эффективность работы вашей программы с приведенным ниже тривиальным кодом побайтового копирования

```
SUBU $1,$1,$0;1H LDBU $2,$0,0; STBU $2,$0,$1; INCL $0,1; PBNZ $2,1B
```

Для копирования строки длиной n потребуется $(2n + 2)\mu + (4n + 7)\nu$ времени.

25. [26] Криптоаналитик хочет подсчитать частоту появления каждого символа в длинной строке зашифрованного текста. Создайте программу MMIX для вычисления частоты появления 255 ненулевых символов. Первый нулевой байт завершает данную строку. Попробуйте найти эффективное решение на основе критерия хронометража “мемсов и упсов” из табл. 1 в разделе 1.3.1'.

- 26. [32] Попробуйте улучшить решение, предложенное в предыдущем упражнении, за счет оптимизации с учетом реальной конфигурации симулятора конвейера MMIX.

27. [26] (*Приближение Фибоначчи.*) Согласно 1.2.8–(15), формула $F_n = \text{round}(\phi^n/\sqrt{5})$ верна для всех $n \geq 0$, где ‘round’ обозначает округление до ближайшего целого числа. (а) Создайте полную программу MMIX для проверки этой формулы в отношении операций с числами с плавающей точкой: Вычислите непосредственное приближение $\phi^n/\sqrt{5}$ для $n = 0, 1, 2, \dots$, и найдите наименьшее n , для которого это приближение не округляется до F_n . (б) В упражнении 1.2.8–28 доказывалось, что $F_n = \text{round}(\phi F_{n-1})$ для всех $n \geq 3$. Найдите наименьшее $n \geq 3$, для которого это уравнение неверно, при приближенном вычислении ϕF_{n-1} на основе умножения с *фиксированной точкой* беззнаковых октабайтов. (См. упр. 1.3.1'–(7).)

28. [26] *Магический квадрат порядка n* является упорядочением чисел от 1 до n^2 в виде квадратного массива таким образом, что сумма по каждой строке и столбцу равна $n(n^2 + 1)/2$, причем такой же является сумма по двум главным диагоналям. На рис. 16 показан магически квадрат порядка 7. Для его создания используется следующий очень простой алгоритм. Сначала поместите 1 вблизи середины квадрата. Затем помещайте следующие числа, опускаясь вниз и вправо по диагонали до достижения края квадрата. Далее помещайте числа в “свернутом” квадрате так, как если бы такие квадраты покрывали всю плоскость. После достижения занятой ячейки опуститесь на две ячейки от самого последнего квадрата и продолжите процедуру. Этот метод работает для нечетных n .

Используя память, выделенную так же, как и в упр. 18, создайте полную программу MMIX для генерации магического квадрата 19×19 с помощью приведенного выше метода и выведите результат в стандартное устройство вывода (в файл в данном случае). [Этот алгоритм предложил Ибн-Эль-Хайтам, который родился в Басре в 965 году и умер в Каире около 1040 года. Многие другие способы построения магического квадрата представляют

32. [31] Приведенный ниже алгоритм предложен неаполитанским астрономом Алоизием Лилиусом (Aloysius Lilius) и немецким математиком иезуитом Кристофером Клавием (Christopher Clavius) в конце XVI века. Он используется большинством западных церквей для определения даты Пасхи для любого года после 1582.

Алгоритм Е (*Дата Пасхи*). Пусть Y обозначает год, для которого нужно определить дату Пасхи.

- Е1.** [Золотое число.] Пусть $G \leftarrow (Y \bmod 19) + 1$. (G — это так называемое “золотое число” года в 19-летнем метонимическом цикле.)
- Е2.** [Век.] Пусть $C \leftarrow \lfloor Y/100 \rfloor + 1$. (Если Y не кратно 100, то C является номером века; например, 1984 относится к XX веку.)
- Е3.** [Поправки.] Пусть $X \leftarrow \lfloor 3C/4 \rfloor - 12$, $Z \leftarrow \lfloor (8C + 5)/25 \rfloor - 5$. (Здесь X — это год, например 1900, в котором високосный год не учитывается для синхронизации с движением Солнца; Z — это специальная поправка для синхронизации Пасхи с движением Луны.)
- Е4.** [Найти воскресенье.] Пусть $D \leftarrow \lfloor 5Y/4 \rfloor - X - 10$.
(День $((-D) \bmod 7)$ марта будет воскресеньем.)
- Е5.** [Эпакта (разница лунного и солнечного календаря).] Пусть $E \leftarrow (11G + 20 + Z - X) \bmod 30$. Если $E = 25$ и золотое число G больше 11, либо если $E = 24$, то увеличить E на 1. (В этом контексте *эпакта* E обозначает момент полной Луны.)
- Е6.** [Найти дату полной Луны.] Пусть $N \leftarrow 44 - E$. Если $N < 21$, то $N \leftarrow N + 30$. (Датой Пасхи считается первое воскресенье после первой полной Луны, которая происходит 21 марта или позже. Фактически возмущения орбиты Луны не позволяют строго сформулировать это правило, но в данном случае рассматривается понятие “календарная Луна”, а не фактическая Луна. N -е марта является датой полной календарной Луны.)
- Е7.** [Переход к воскресенью.] Пусть $N \leftarrow N + 7 - ((D + N) \bmod 7)$.
- Е8.** [Получить месяц.] Если $N > 31$, то искомая дата равна $(N - 31)$ APRIL; в противном случае искомая дата равна N MARCH. ■

Создайте подпрограмму для вычисления и вывода даты Пасхи для заданного года (до 100000 года). Вывод должен иметь формат “ dd MONTH, $yyyy$ ” где dd — это день, MONTH — месяц, а $yyyy$ — год. Создайте полную программу MMIX, которая использует эту подпрограмму для подготовки таблицы дат Пасхи от 1950 до 2000.

33. [M30] Некоторые компьютеры — но только не MMIX — дают отрицательный остаток при делении отрицательного числа на положительное число. Следовательно, программа вычисления даты Пасхи в предыдущем примере может дать ошибочный результат, если величина $(11G + 20 + Z - X)$ на этапе Е5 является отрицательной. Например, в 14250 году получим $G = 1$, $X = 95$, $Z = 40$. Если $E = -24$, а не $E = +6$, то программа выдаст бессмысленный результат “42 APRIL” (42 апреля). [См. CACM 5 (1962), 556.] Создайте полную программу MMIX, способную найти *самый первый* год, в котором из-за такой ошибки происходит неверное вычисление даты Пасхи.

- **34.** [33] Предположим, что компьютер MMIX связан со светофорами на перекрестке бульвара Дель Мар (Del Mar Boulevard) и и Беркли Авеню (Berkeley Avenue) с помощью особых “файлов” `/dev/lights` и `/dev/sensor`. Компьютер переключает огни светофора, выводя

один байт в файл `/dev/lights` на основе суммы следующих четырех двухбитовых кодов:

Светофор для машин на Дель Мар: #00 выключен, #40 зеленый, #80 желтый,
: #с0 красный;
Светофор для машин на Беркли Авеню: #00 выключен, #10 зеленый, #20 желтый,
: #30 красный;
Светофор для пешеходов на Дель Мар: #00 выключен, #04 ИДИТЕ, #0с СТОЙТЕ;
Светофор для пешеходов на Беркли Авеню: #00 выключен, #01 ИДИТЕ, #03 СТОЙТЕ.

Машины или пешеходы для продолжения движения по Беркли Авеню через бульвар Дель Мар должны активировать сенсор (sensor), в противном случае свет на бульваре Дель Мар остается зеленым. Если ММІХ считывает байт из `/dev/sensor`, то вход не является нулевым тогда и только тогда, когда сенсор активирован с момента предыдущего входа.

Время циклической активности цветов имеет следующий вид:

Светофор для машин на Дель Мар зеленый ≥ 30 секунд, желтый 8 секунд;

Светофор для машин на Беркли Авеню зеленый 20 секунд, желтый 5 секунд.

Если светофор имеет зеленый или желтый цвет в одном направлении, то в другом направлении он имеет красный цвет. Если светофор имеет зеленый цвет, то горит соответствующая ему надпись ИДИТЕ, за исключением того, что надпись СТОЙТЕ начинает мигать за 12 секунд до переключения зеленого цвета на желтый в следующей последовательности:

СТОЙТЕ $\frac{1}{2}$ секунды
выключена $\frac{1}{2}$ секунды } повторяется 8 раз;

СТОЙТЕ 4 секунды (включена во время циклов желтого и красного цвета)

Если сенсор активирован во время зеленого цвета светофора Беркли Авеню, то машины или пешеходы могут идти во время этого цикла. Но, если сенсор активирован во время желтого или красного цвета светофора Беркли Авеню, потребуется другой цикл по окончании движения по бульвару Дель Мар.

Создайте программу ММІХ для управления светофором, согласно указанному протоколу. Предположим, что специальный регистр часов гС увеличивается на 1 точно ρ раз в секунду, где ρ является целочисленной константой.

35. [37] Это упражнение предназначено для тренировки методов вывода информации в графическом виде, а не в виде обычной таблицы. Цель данного упражнения заключается в том, чтобы “нарисовать” клетчатую форму для кроссворда.

На входе имеется матрица из нулей и единиц. Ноль обозначает белый, а единица — черный квадратик. На выходе должна генерироваться форма кроссворда с нумерацией для вписывания слов по вертикали и горизонтали.

Например, входной матрице

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

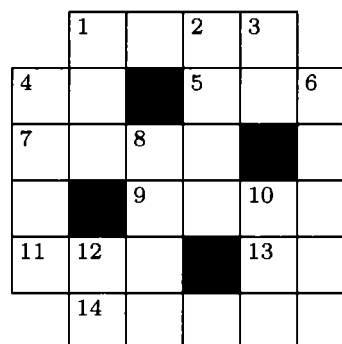


Рис. 18. Форма кроссворда, соответствующая матрице в упражнении 35.

соответствует форма кроссворда, показанная на рис. 18. Квадратик получает номер, если он белый и либо (а) под ним есть белый квадратик и над ним нет белого квадратика, либо

(b) справа есть белый квадратик и слева нет белого квадратика. Если черный квадратик возникает на краях формы, то его не нужно рисовать. На рис. 18 показано, что черные квадратик на углах формы опущены. Простейший способ достижения этой цели состоит в том, чтобы искусственно вставить строки и столбцы из отрицательных единиц -1 сверху, снизу, слева и справа от входной матрицы, а потом заменить каждую положительную единицу $+1$, которая находится в смежной позиции с -1 , на -1 до тех пор, пока совсем не останется положительной единицы $+1$, смежной с любой отрицательной единицей -1 .

Рис. 18 создан программой METAPOST, текст которой приводится на рис. 19. С помощью простых манипуляций с операторами `line` и `black`, координатами в циклах `for` можно создать практически любую форму кроссворда.

Создайте программу MMIX для считывания матрицы 25×25 нулей и единиц в стандартном входном файле и вывода соответствующей программы METAPOST в стандартном выходном файле. Вход должен состоять из 25 строк, включающих 25 цифр и символ "новая строка". Например, первая строка, соответствующая приведенной выше матрице, будет иметь вид '1000011111111111111111', где дополнительные единицы используются для расширения исходного массива 6×6 . Форма кроссворда не обязательно должна быть симметричной и может иметь длинные участки черных квадратиков, связанных с внешним миром самым причудливым образом.

```

beginfig(18)
transform t; t=identity rotated -90 scaled 17pt;
def line(expr i,j,ii,jj) =
  draw ((i,j)--(ii,jj)) transformed t;
enddef;
def black(expr i,j) =
  fill ((i,j)--(i+1,j)--(i+1,j+1)--(i,j+1)--cycle) transformed t;
enddef;

line (1,2,1,6); line (2,1,2,7); line (3,1,3,7); line (4,1,4,7);
line (5,1,5,7); line (6,1,6,7); line (7,2,7,6);
line (2,1,6,1); line (1,2,7,2); line (1,3,7,3); line (1,4,7,4);
line (1,5,7,5); line (1,6,7,6); line (2,7,6,7);


numeric n; n=0;
for p = (1,2),(1,4),(1,5), (2,1),(2,4),(2,6),
  (3,1),(3,3), (4,3),(4,5), (5,1),(5,2),(5,5), (6,2):
  n:=n+1; label.lrt(decimal n infont "cmr8", p transformed t);
endfor

black(2,3); black(3,5); black(4,2); black(5,4);
endfig;

```

Рис. 19. Программа METAPOST, которая создает форму кроссворда, показанную на рис. 18

1.3.3'. Применения к перестановкам

 Программы MIX в прежнем издании раздела 1.3.3 и в главах 2, 3, 4, 5 и 6 будут конвертированы в программы MMIX. Все желающие помочь и принять участие в этом поучительном проекте приглашаются в группу разработчиков MMIXmasters (см. стр. v).

1.4'. НЕКОТОРЫЕ ФУНДАМЕНТАЛЬНЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ

1.4.1'. Подпрограммы

Если НЕКОТОРАЯ задача выполняется в нескольких разных местах программы, то обычно не рекомендуется многократно повторять один и тот же фрагмент кода в каждом таком месте. Во избежание такого повторения, такой фрагмент кода (который называется *подпрограммой*) помещается только в одно место программы, а несколько дополнительных инструкций можно добавить для повторного запуска основной программы после завершения работы подпрограммы. Передача управления между подпрограммами и основными программами называется *связыванием подпрограмм*.

В каждом компьютере предусмотрен собственный способ эффективного связывания подпрограмм, который обычно включает несколько инструкций. Дальнейшее обсуждение касается компьютера MMIX, но аналогичные рассуждения верны и для связывания подпрограмм на других типах компьютеров.

Подпрограммы используются для экономии места в программе, а не времени, если не считать неявно сэкономленного времени в связи с ее уменьшенным размером, — например, на загрузку, либо более эффективное использование высокоскоростной памяти на компьютерах с разными типами памяти. Дополнительное время на вызов и выход подпрограммы обычно пренебрежимо мало, за исключением случаев, когда они находятся в глубоко вложенных циклах.

Подпрограммы обладают несколькими преимуществами. Они упрощают внешний вид большой и сложной программы, логически разбивают задачу на части, что обычно существенно упрощает отладку программы. Многие подпрограммы ценны еще и тем, что их могут использовать даже те пользователи, которые не участвовали в их создании.

Для большинства компьютеров предусмотрены большие библиотеки полезных подпрограмм, которые значительно упрощают программирование стандартных компьютерных приложений. Однако не следует считать это *единственным* предназначением подпрограмм. Подпрограммы не нужно рассматривать всего лишь как программы общего назначения. Специализированные подпрограммы имеют не меньшую важность, даже если предназначены только для использования в одной программе. В разд. 1.4.3' содержится несколько типичных примеров таких ситуаций.

Простейшие подпрограммы имеют только один вход и один выход, как подпрограмма MAXIMUM, уже рассмотренная ранее (см. программу M в разд. 1.3.2' и упр. 1.3.2'-3). Рассмотрим еще раз эту программу, слегка переделав ее так, чтобы максимальное значение находилось среди фиксированного числа ячеек, например среди 100:

```

* Поиск максимума среди X[1..100]
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
Max100 SETL   kk,100*8   M1. Инициализировать.
        LD0    m,x0,kk
        JMP    1F
3H      LD0    xk,x0,kk   M3. Сравнить.
        CMP    t,xk,m
        PBNP   t,5F
4H      SET    m,xk       M4. Изменить m.
1H      SR     j,kk,3
5H      SUB    kk,kk,8    M5. Уменьшить k.
        PBP    kk,3B      M2. Все проверены?
6H      POP    2,0        Возврат в основную программу.  █

```

(1)

Предполагается, что эта подпрограмма является частью более крупной программы, в которой символ t обозначает регистр \$255, а символ $x0$ обозначает глобальный регистр так, что $X[k]$ находится в позиции $x0 + 8k$. В этой более крупной программе инструкция “PUSHJ \$1,Max100” устанавливает для регистра \$1 текущее максимальное значение среди $\{X[1], \dots, X[100]\}$, а позиция максимума заносится в регистр \$2. Связывание в этом случае достигается с помощью инструкции PUSHJ, которая вызывает подпрограмму, и с помощью инструкции “POP 2,0” в конце подпрограммы. Как будет показано ниже, эти инструкции MMIX вызывают перенумерацию локальных регистров во время работы подпрограммы. Более того, инструкция PUSHJ вставляет адрес возврата в специальный регистр \mathbf{rJ} , а инструкция POP переходит в эту позицию.

Связывание подпрограмм можно выполнить более простым способом с использованием инструкции GO вместо инструкций проталкивания и выталкивания. Например, для этого можно использовать следующий код вместо (1):

```

* Поиск максимума среди X[1..100]
j GREG ;m GREG ;kk GREG ;xk GREG
        GREG    @          Базовый адрес.
GoMax100 SETL   kk,100*8   M1. Инициализировать.
        LD0    m,x0,kk
        JMP    1F
3H      ...
        PBP    kk,3B      M2. Все проверены?
6H      GO     kk,$0,0     Возврат в основную программу.  █

```

(2)

Теперь инструкция “GO \$0,GoMax100” передает управление подпрограмме, заменяя адрес следующей инструкции в регистре \$0; последующая операция “GO kk,\$0,0” в конце подпрограммы вернется к этому адресу. В этом случае максимальное значение появится в глобальном регистре m , а его позиция будет в глобальном регистре j . Два дополнительных глобальных регистра, kk и xk , также используются данной подпрограммой. Более того, инструкция “GREG @” предоставляет базовый адрес так, что можно перейти к GoMax100 с помощью одной инструкции. В противном случае потребуется двухстадийная последовательность “GETA \$0,GoMax100; GO \$0,\$0,0”. Связывание подпрограмм типа (2) обычно используется на компьютерах без встроенного механизма стека регистров.

Не трудно сделать *количественную* оценку сэкономленного кода и затраченного времени при использовании подпрограмм. Предположим, что для фрагмента кода требуется k тетрабайт и он используется в m местах программы. Для преобразования этого фрагмента в подпрограмму нужно вставить инструкцию PUSHJ или GO в каждое из m мест вызова подпрограммы, а также использовать инструкцию POP или GO для возврата управления.

В целом это дает $m + k + 1$ тетрабайт, вместо mk , а потому сэкономленный код равен

$$(m - 1)(k - 1) - 2. \quad (3)$$

Если k равно 1 или m равно 1, то очевидно никакой экономии вообще не будет. Если k равно 2, то для получения выгоды m должно быть больше 3.

Затраченное время равняется времени выполнения инструкций PUSHJ, POP и/или GO в процессе связывания. Если подпрограмма вызывается t раз во время выполнения программы, а время выполнения программы определяется приближениями в табл. 1.3.1'-1, то дополнительные затраты равны $4tv$ в случае (1) или $6tv$ в случае (2).

Эти оценки нужно использовать осторожно, потому что они получены для идеализированной ситуации. Многие подпрограммы нельзя вызвать с помощью одной инструкции PUSHJ или GO. С одной стороны, если фрагмент кода реплицируется во многих местах программы без применения подпрограмм, то в каждом таком случае его можно настроить так, чтобы воспользоваться преимуществами особенностей именно этого места программы, в которой находится данный фрагмент кода. С другой стороны, при использовании подпрограммы фрагмент кода должен иметь довольно общий вид, а потому в него часто включают несколько дополнительных инструкций.

Если подпрограмма создается для обработки общего случая, то она выражается с помощью *параметров*. Параметрами называются величины, которые управляют действиями подпрограммы. Их значения могут меняться при разных вызовах подпрограммы.

Код во внешней программе, который передает управление подпрограмме и запускает ее, называется *вызывающей последовательностью*. Значения параметров, которые передаются при вызове подпрограммы, называются *аргументами*. В подпрограмме GoMax100 вызывающей последовательностью является "GO \$0, GoMax100", но при использовании аргументов вызывающая последовательность обычно длиннее.

Например, попробуем обобщить подпрограмму (2), чтобы она находила максимум среди первых n элементов массива для *произвольной* константы n , помещая n в поток инструкций с двухстадийной вызывающей последовательностью

$$GO \$0, GoMax; \quad TETRA \ n. \quad (4)$$

Подпрограмма GoMax в этом случае будет иметь вид:

```

* Поиск максимума среди X[1..100]
j GREG ;m GREG ;kk GREG ;xk GREG
      GREG  @      Базовый адрес.
GoMax LDT    kk,$0,0 Извлечь аргумент.
      SL     kk,kk,3
      LDO    m,x0,kk
      JMP    1F
3H     ...      (Продолжить, как в (1))
      PBP    kk,3B
6H     GO     kk,$0,4 Возврат в вызывающую программу. █

```

(5)

Параметр n рекомендуется поместить в регистр. Тогда двухстадийная вызывающая последовательность будет иметь вид:

$$\text{SET } \$1, n; \quad \text{GO } \$0, \text{GoMax} \quad (6)$$

и соответствующая подпрограмма будет выглядеть так:

```

GoMax SL     kk,$1,3 Извлечь аргумент.
      LDO    m,x0,kk
      ...
6H     GO     kk,$0,0 Возврат в вызывающую программу. █

```

(7)

Этот вариант быстрее, чем (5), и позволяет динамически изменять n без изменения потока инструкций.

Обратите внимание, что адрес элемента массива $X[0]$ также является параметром подпрограмм (1), (2), (5) и (7). Операция помещения этого адреса в регистр $x0$ может рассматриваться как часть вызывающей последовательности в тех случаях, когда массив меняется при каждом вызове.

Если вызывающая последовательность занимает s тетрабайт памяти, то формула (3) для сэкономленного пространства принимает вид:

$$(m - 1)(k - c) - \text{константа} \quad (8)$$

и немного увеличивается время, затраченное на связывание подпрограммы.

Дополнительное уточнение приведенных выше формул потребуется, поскольку нужно сохранять и восстанавливать содержимое некоторых регистров. Например, в подпрограмме GoMax нужно помнить, что инструкция “SET \$1, n ; GO \$0, GoMax” не только вычисляет максимальное значение в регистре m и его положение в регистре j , но также изменяет значения глобальных регистров kk и xk . Подпрограммы (2), (5) и (7) созданы с неявным предположением, что регистры kk и xk используются только для данной подпрограммы, но в большинстве компьютеров не так много регистров.

Даже в MMIX регистры быстро закончатся, если запустить большое количество подпрограмм. Поэтому подпрограмму (7) нужно подкорректировать так, чтобы она работала с $kk \equiv \$2$ и $xk \equiv \$3$ без затирания полезного содержимого этих регистров.

Это достигается следующей корректировкой подпрограммы

j	GREG	;m GREG ;kk IS \$2 ;xk IS \$3	
	GREG	@	Базовый адрес.
GoMax	ST0	kk,Tempkk	Сохранить предыдущее содержимое регистра.
	ST0	xk,Tempxk	
	SL	kk,\$1,3	Извлечь аргумент.
	LD0	m,x0,kk	
	...		
	LD0	kk,Tempkk	Восстановить предыдущее содержимое регистра.
	LD0	xk,Tempxk	
6H	GO	\$0,\$0,0	Возврат в вызывающую программу. █

(9)

и установкой двух октабайтов Tempkk и Tempxk в сегменте данных. Конечно, это изменение увеличивает накладные расходы при каждом использовании подпрограммы.

Подпрограмму можно рассматривать как *расширение* машинного языка данного компьютера. Например, если подпрограмма GoMax находится в памяти, то таким образом в нашем распоряжении оказывается машинная инструкция (а именно, "GO \$0,GoMax"), которая способна отыскать максимум. Здесь важно так же тщательно определить каждую подпрограмму, как определяются операторы машинного языка. Программисту всегда настоятельно рекомендуется указывать все характеристики подпрограммы, даже если никто кроме него не будет пользоваться этой подпрограммой или ее спецификацией. В случае подпрограммы GoMax с определением (7) или (9) ее характеристики имеют следующий вид:

Вызывающая последовательность:	GO \$0,GoMax.	
Условия входа:	$\$1 = n \geq 1$; $x0 = \text{address of } X[0]$.	(10)
Условия выхода:	$m = \max_{1 \leq k \leq n} X[k] = X[j]$.	

В спецификации следует упомянуть все изменения тех величин, которые являются внешними для подпрограммы. Если регистры kk и xk не считаются "частными" для варианта (7) подпрограммы GoMax, следует учесть, что эти регистры вовлекаются в процесс выполнения подпрограммы, как часть условий входа. Подпрограмма также изменяет регистр t, а именно регистр \$255, но этот регистр обычно используется для временных значений, поэтому не нужно указывать его явно.

Рассмотрим теперь *многократные входы* в подпрограммы. Предположим, имеется программа, в которой используется общая подпрограмма GoMax, но требуется применить специализированную подпрограмму GoMax100, в которой $n = 100$. Их можно скомбинировать таким образом:97.02.24

GoMax100	SET	\$1,100	Первый вход.	
GoMax	...		Второй вход; продолжить, как в (7) или (9).	█

(11)

Можно также добавить *третий* вход, например GoMax50, используя код

GoMax50 SET \$1,50; JMP GoMax

в некотором месте.

Подпрограмма может иметь *многократные выходы*, т.е. возвращать значения в одно из нескольких разных мест, в зависимости от заданных условий.

Например, можно расширить подпрограмму (11) с помощью предположения, что параметр верхней границы задан в глобальном регистре *b*. Теперь предполагается, что выход подпрограммы произойдет в одном и двух тетрабайтов, которые располагаются вслед за вызывающей инструкцией *G0*:

Для произвольного <i>n</i>	Для <i>n</i> = 100
SET \$1, <i>n</i> ; G0 \$0, GoMax	G0 \$0, GoMax100
Выход здесь, если $m \leq 0$ или $m \geq b$.	Выход здесь, если $m \leq 0$ или $m \geq b$.
Выход здесь, если $0 < m < b$.	Выход здесь, если $0 < m < b$.

(Иначе говоря, тетрабайт после *G0* пропускается, когда максимальное значение положительно и меньше верхней границы. Такая подпрограмма полезна в программах, где часто нужно принять определенное решение после вычисления максимального значения.) Ее реализация выглядит достаточно просто:

```

* Поиск максимума X[1..n] с проверкой границ
j GREG ;m GREG ;kk GREG ;xk GREG
      GREG      @      Базовый адрес.
GoMax100 SET      $1,100  Вход для n = 100.
GoMax    SL       kk,$1,3  Вход для произвольного n.
      LDO        m,x0,kk
      JMP        1F
3H      ...      (Продолжить, как в (1)).
      PBP        kk,3B
      BNP        m,1F      Переход, если  $m \leq 0$ .
      CMP        kk,m,b
      BN         kk,2F      Переход, если  $m < b$ .
1H      G0        kk,$0,0  Первый выход, если  $m \leq 0$  или  $m \geq b$ .
2H      G0        kk,$0,4  В противном случае второй выход. █

```

Обратите внимание, что в этой программе комбинируется технология связывания потока инструкций (5) с технологией установки регистра (7). Строго говоря, положением выхода подпрограммы является параметр. Следовательно, положения нескольких выходов могут задаваться в виде аргументов. Если подпрограмма постоянно осуществляет доступ к своим параметрам, то соответствующий аргумент лучше всего передавать в регистре, но если аргумент является константой и используется редко, то лучше хранить его в потоке инструкций.

Подпрограммы могут вызывать другие подпрограммы. Действительно, в очень сложных программах часто вызовы подпрограмм могут быть вложены на глубину более пяти уровней. Единственным ограничением в случаях связывания подпрограмм на основе *G0* является то, что все адреса и регистры временного хранения должны быть разными, таким образом, не допускается вызов подпрограммой любой другой подпрограммы, которая (прямо или косвенно) вызывает первую подпрограмму. Рассмотрим в качестве примера следующий сценарий.

[Основная программа]	[Подпрограмма А]	[Подпрограмма В]	[Подпрограмма С]	
⋮	А ⋮	В ⋮	С ⋮	(13)
G0 \$0, А	G0 \$1, В	G0 \$2, С	G0 \$0, А	
⋮	⋮	⋮	⋮	
	G0 \$0, \$0, 0	G0 \$1, \$1, 0	G0 \$2, \$2, 0	

Если основная программы вызывает подпрограмму А, которая вызывает подпрограмму В, которая вызывает подпрограмму С, а потом С вызывает подпрограмму А, то адрес в \$0, относящийся к основной программе, уничтожается, а потому обрывается путь возврата в основную программу.

Использование стековой памяти. Хотя рекурсивное использование подпрограмм, как в примере (13), редко возникает в простых программах, оно гораздо чаще встречается в исходной структуре важных приложений. К счастью, существует простой и прямолинейный способ исключения взаимного вредного влияния между вызовами подпрограмм за счет сохранения их локальных переменных в *стеке*.

Например, можно задать глобальный регистр *sp* ("stack pointer" — указатель стека) и применить инструкцию G0 \$0, Sub для вызова каждой программы. Если код подпрограммы имеет вид

```

Sub  ST0  $0, sp, 0
      ADD  sp, sp, 8
      ...
      SUB  sp, sp, 8
      LD0  $0, sp, 0
      G0   $0, $0, 0

```

(14)

то регистр \$0 всегда будет содержать соответствующий адрес возврата, а проблема (13) будет исключена. (Сначала в *sp* заносится адрес в сегменте данных, а затем все другие необходимые адреса памяти.) Более того, инструкции ST0/ADD и SUB/LD0 в (14) можно опустить, если Sub является так называемой *концевой подпрограммой*, — которая не вызывает других подпрограмм.

Стек можно использовать для хранения параметров и других локальных переменных, кроме адресов возврата в (14). Предположим, что в подпрограмме Sub, кроме адреса возврата, используется 20 октабайт локальных данных. В этой ситуации можно использовать следующую схему:

```

Sub  ST0  fp, sp, 0   Сохранить прежний указатель стекового фрейма.
      SET  fp, sp     Установить новый указатель стекового фрейма.
      INCL sp, 8*22   Передвинуть указатель стека.
      ST0  $0, fp, 8   Сохранить адрес возврата.
      ...
      LD0  $0, fp, 8   Восстановить адрес возврата.
      SET  sp, fp     Восстановить указатель стека.
      LD0  fp, sp, 0   Восстановить указатель стекового фрейма.
      G0   $0, $0, 0   Вернуться в вызывающую программу. █

```

(15)

Здесь *fp* является глобальным регистром, который называется *указателем стекового фрейма*.

В отмеченной многоточием части подпрограммы локальное значение k эквивалентно октабайту по адресу $fp + 8k + 8$ для $1 \leq k \leq 20$. Инструкции в начале “проталкивают” локальные значения в “верх” стека, а инструкции в конце “вытаскивают” их из стека, оставляя стек в том же состоянии, которое он имел во время входа подпрограммы.

Использование стека регистров. Связывание подпрограмм на основе GO столь подробно рассматривалось здесь, поскольку многие компьютеры не имеют более удачных альтернативных вариантов связывания подпрограмм.

Но в MMIX встроены инструкции PUSHJ и POP, которые гораздо более эффективно управляют связыванием подпрограмм и позволяют сократить многие накладные расходы, которые присущи схемам (9) и (15). Эти инструкции позволяют сохранять большинство параметров и локальных переменных в регистрах вместо хранения их в стековой памяти и повторной загрузки. С помощью инструкций PUSHJ и POP большая часть рутинной работы со стеком автоматически выполняется компьютером.

Основная идея стека регистров чрезвычайно проста, если вспомнить принцип работы стека, как такового. MMIX имеет *стек регистров* из октабайтов $S[0]$, $S[1]$, ..., $S[\tau - 1]$ для некоторого числа $\tau \geq 0$. Самые верхние L октабайтов этого стека (а именно $S[\tau - L]$, $S[\tau - L + 1]$, ..., $S[\tau - 1]$) являются *локальными регистрами* $\$0$, $\$1$, ..., $\$(L - 1)$, а другие $\tau - L$ октабайт стека пока недоступны программе и будут называться “протолкнутыми вниз”. Текущее количество локальных регистров, L , хранится в специальном регистре rL компьютера MMIX, хотя программисту совсем не обязательно это знать. В исходном состоянии $L = 2$, $\tau = 2$, а локальные регистры $\$0$ и $\$1$ представляют командную строку, как в программе 1.3.2'Н.

MMIX также имеет *глобальные регистры*, а именно $\$G$, $\$(G + 1)$, ..., $\$255$. Значение G хранится в специальном регистре rG , причем всегда $0 \leq L \leq G \leq 255$. (На самом деле также всегда выполняется условие $G \geq 32$.) Глобальные регистры *не* являются частью стека регистров.

Регистры, которые не являются ни локальными, ни глобальными, называются *маргинальными*. Эти регистры, а именно $\$L$, $\$(L + 1)$, ..., $\$(G - 1)$, имеют нулевое значение всякий раз, когда они используются как входные операнды инструкции MMIX.

Стек регистров возрастает, если маргинальному регистру присваивается значение. Этот маргинальный регистр становится локальным, как и все маргинальные регистры с меньшими номерами. Например, если в данный момент используется восемь локальных регистров, то инструкция ADD $\$10, \$20, 5$ приводит к тому, что регистры $\$8$, $\$9$, и $\$10$ становятся локальными. Точнее говоря, если $rL = 8$, то инструкция ADD $\$10, \$20, 5$ задает $\$8 \leftarrow 0$, $\$9 \leftarrow 0$, $\$10 \leftarrow 5$, и $rL \leftarrow 11$. (А регистр $\$20$ остается маргинальным.)

Если $\$X$ является локальным регистром, то инструкция PUSHJ $\$X, Sub$ уменьшает количество локальных регистров и изменяет их порядковые номера: прежние локальные регистры $\$(X + 1)$, $\$(X + 2)$, ..., $\$(L - 1)$ теперь называются $\$0$, $\$1$, ..., $\$(L - X - 2)$ внутри подпрограммы, а значение L уменьшается на $X + 1$. Таким образом, стек регистра остается неизменным, но $X + 1$ его элементов становятся недоступными. Подпрограмма не может повредить эти элементы и получает $X + 1$ новых маргинальных регистров.

Если $X \geq G$, а $\$X$ является глобальным регистром, то результат выполнения инструкции `PUSHJ $\$X$,Sub` аналогичен, но новый элемент помещается в стек регистров и потом $L + 1$ регистров проталкиваются вниз вместо $X + 1$. В данном случае значение L равно нулю в начале подпрограммы, все прежние локальные регистры проталкиваются вниз, а подпрограмма начинается с чистого листа.

Стек регистров сжимается, только если задается инструкция `ROR` или программа явно уменьшает количество локальных регистров с помощью инструкции `PUT rL,5`. Цель `ROR X,YZ` состоит в том, чтобы сделать снова доступными элементы, проталкиваемые вниз самой последней инструкцией `PUSHJ`, как прежде, и удалить из стека регистров более неиспользуемые элементы. Вообще, поле X инструкции `ROR` содержит количество значений, “возвращаемых” подпрограммой, если $X \leq L$. Если $X > 0$, то возвращаемое значение равно $\$(X-1)$. Из регистра удаляется этот элемент и все элементы выше него, а возвращаемое значение помещается в адрес, указанный командой `PUSHJ`, которая вызывала данную подпрограмму. Поведение инструкции `ROR` аналогично, если $X > L$, но в данном случае стек регистров остается неизменным и нулевое значение помещается по адресу, указанному командой `PUSHJ`.

Описанные выше правила могут показаться сложными, поскольку на практике могут иметь место разные ситуации. Однако несколько примеров могут прояснить их. Предположим, что есть подпрограмма A и в ней нужно вызвать подпрограмму B . Допустим, что подпрограмма A имеет 5 локальных регистров, которые не должны быть доступны подпрограмме B . Этими регистрами являются $\$0$, $\$1$, $\$2$, $\$3$ и $\$4$. Зарезервируем следующий регистр $\$5$ для основного результата подпрограммы B . Если B имеет три параметра, то $\$6 \leftarrow \text{arg0}$, $\$7 \leftarrow \text{arg1}$ и $\$8 \leftarrow \text{arg2}$, и после команды `PUSHJ $\$5$,B` вызывается подпрограмма B и аргументы помещаются в $\$0$, $\$1$ и $\$2$.

Если B не возвращает результат, то ее выполнение завершается командой `ROR 0,YZ`, которая восстанавливает $\$0$, $\$1$, $\$2$, $\$3$ и $\$4$ к исходным значениям и задает $L \leftarrow 5$.

Если B возвращает единственный результат x , то x помещается в $\$0$ и ее выполнение завершается командой `ROR 1,YZ`, которая восстанавливает $\$0$, $\$1$, $\$2$, $\$3$, и $\$4$ к прежним значениям и задает $\$5 \leftarrow x$ и $L \leftarrow 6$.

Если B возвращает два результата x и a , то основной результат x помещается в $\$1$ и вспомогательный результат a помещается в $\$0$. Затем `ROR 2,YZ` восстанавливает регистры от $\$0$ до $\$4$ и задает $\$5 \leftarrow x$, $\$6 \leftarrow a$, $L \leftarrow 7$. Аналогично, если B возвращает десять результатов (x, a_0, \dots, a_8) , то основной результат x помещается в $\$9$ и остальные результаты в первые девять регистров: $\$0 \leftarrow a_0$, $\$1 \leftarrow a_1, \dots, \$8 \leftarrow a_8$. Затем `ROR 10,YZ` восстанавливает регистры от $\$0$ до $\$4$ и задает $\$5 \leftarrow x$, $\$6 \leftarrow a_0, \dots, \$14 \leftarrow a_8$. (Забавная перестановка регистров, которая возникает, если возвращается два или более результата, может показаться довольно странной на первый взгляд. Однако она имеет смысл, поскольку оставляет стек регистров без изменений, за исключением основного результата. Например, если подпрограмма B имеет аргументы arg0 , arg1 и arg2 в регистрах $\$6$, $\$7$ и $\$8$ по окончании своей работы, то она может сохранить их в виде вспомогательных результатов в регистрах $\$0$, $\$1$ и $\$2$, а потом выполнить инструкцию `ROR 4,YZ`.)

Поле YZ инструкции `ROR` обычно нулевое, но, вообще говоря, инструкция `ROR X, YZ` возвращается к инструкции, которая располагается через $YZ + 1$ тетрабайт после `PUSHJ`, которая вызвала текущую подпрограмму. Это обобщение полезно для

подпрограмм с несколькими выходами. Точнее говоря, PUSHJ по адресу @ задает специальный регистр rJ для @ + 4 до перехода к подпрограмме, а инструкция POP затем возвращается к адресу rJ + 4YZ.

Теперь можно переписать созданные ранее программы со связыванием подпрограмм на основе инструкции GO так, чтобы в них использовалось связывание на основе инструкций PUSH/POP. Например, подпрограмма поиска максимума (12) с двумя входами и двумя выходами примет следующий вид, если в ней используется механизм стека регистров MMIX:

```

* Поиск максимума среди X[1..n] с проверкой границ.
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
Max100 SET    $0,100    Вход для n = 100.
Max      SL    kk,$0,3   Вход для произвольного n.
          LD0   m,x0,kk
          JMP   1F
          ...
          BN    kk,2F
1H      POP    2,0        Первый выход, если max ≤ 0 или max ≥ b.
2H      POP    2,1        В противном случае второй выход. ■

```

(16)

Для произвольного n

Для n = 100

```

SET $A,n; PUSHJ $R,Max (A = R+1)    PUSHJ $R,Max100
Выход здесь, если $R ≤ 0 или $R ≥ b.  Выход здесь, если $R ≤ 0 или $R ≥ b.
Выход здесь, если 0 < $R < b.        Выход здесь, если 0 < $R < b.

```

Локальный регистр результата \$R в PUSHJ в этой вызывающей последовательности является произвольным, в зависимости от количества локальных переменных, которые вызывающая программа хочет сохранить. Локальный регистр аргумента \$A переходит в \$(R + 1). После этого вызова \$R будет содержать основной результат (максимальное значение) и \$A будет содержать вспомогательный результат (индекс элемента массива с этим максимумом). При наличии нескольких аргументов и/или вспомогательных результатов они именуются A0, A1, ..., причем предполагается, что A0 = R+1, A1 = R+2, ..., если используются вызывающие последовательности с PUSH/POP.

Сравнение (12) и (16) показывает, что у (16) есть небольшие преимущества: в новом варианте необязательно выделять глобальные регистры для j, m, kk и xk, а также необязательно использовать глобальный базовый регистр для адреса команды GO. (Напомним из раздела 1.3.1', что GO имеет абсолютный, а PUSHJ — относительный адрес.) Инструкция GO немного медленнее, чем PUSHJ и POP, согласно табл. 1.3.1'–1, хотя высокоскоростные реализации MMIX могут реализовать POP более эффективно. Программы (12) и (16) имеют одинаковый размер.

Преимущества связывания на основе PUSH/POP по сравнению со связыванием на основе GO начинают проявляться при работе с *неконцевыми* подпрограммами (а именно, подпрограммами, которые вызывают другие подпрограммы, возможно, даже самих себя). Код (14) на основе GO можно заменить так:

```

Sub   GET   retadd,rJ
      ...
      PUT   rJ,retadd
      POP   X,0

```

(17)

где **retadd** является локальным регистром. (Например, **retadd** может быть равным \$5; его номер регистра обычно больше или равен количеству возвращаемых результатов *X*, потому инструкция **POP** автоматически удалит его из стека регистров.) Таким образом исключаются затратные ссылки на память в (14).

Неконцевая подпрограмма со многими локальными переменными и/или параметрами гораздо эффективнее со стеком регистров, чем со стековой памятью (15), поскольку часто вычисления непосредственно выполняются в регистрах. Однако следует отметить, что стек регистров **MMIX** применяется только к локальным переменным, которые являются *скалярами*, а не к локальным переменным-массивам, для доступа к элементам которых требуется вычислять адреса. Для подпрограмм с нескаллярными локальными переменными следует использовать схему (15) для всех таких переменных, а для скаляров применять стек регистров.

Оба подхода можно использовать одновременно, обновляя **fp** и **sp** только подпрограммами, в которых нужно применить стек регистров.

Если стек регистров становится чрезвычайно большим, то **MMIX** автоматически сохранит самые нижние элементы в сегменте памяти стека, используя скрытую процедуру, которая описывается в разделе 1.4.3'. (Напомним из раздела 1.3.2', что этот сегмент памяти стека начинается с адреса #6000 0000 0000 0000.) **MMIX** сохраняет элементы стека регистра в памяти также, когда команда **SAVE** сохраняет весь текущий контекст программы. Сохраненные элементы стека автоматически восстанавливаются из памяти при выполнении команды **POP** или команды **UNSAVE**, которая восстанавливает сохраненный контекст. Но в большинстве случаев **MMIX** способен проталкивать и выталкивать локальные регистры без фактического доступа к памяти и без фактического изменения содержимого очень многих внутренних регистров компьютера.

Стеки имеют много других способов применения в компьютерных программах. Их основные свойства рассматриваются в разделе 2.2.1. Более подробно вложенные подпрограммы и рекурсивные процедуры рассматриваются в разделе 2.3 при обсуждении операций с деревьями. В главе 8 подробно рассматривается рекурсия.

***Компоненты языка ассемблера.** Язык ассемблера **MMIX** поддерживает создание подпрограмм тремя способами, которые не упоминались в разделе 1.3.2'. Наиболее важной является операция **PREFIX**, которая упрощает определение “частных” символов, не взаимодействующих с символами, определенными в других местах большой программы. Основная идея состоит в том, что символ может иметь структурированный вид, например **Sub:X** (что значит символ *X* подпрограммы *Sub*), возможно, передаваемый на несколько уровней в виде **Lib:Sub:X** (что значит символ *X* подпрограммы *Sub* в библиотеке *Lib*).

Структурированные символы используют слегка расширенное правило 1 языка **MMIXAL** из раздела 1.3.2', которое позволяет рассматривать двоеточие ':' как “букву” для конструирования символов. Каждый символ, который не начинается с двоеточия, неявно расширяется размещением перед ним *текущего префикса*. Текущий префикс имеет вид ':', но пользователь может изменить его с помощью команды

PREFIX. Например:

ADD	x,y,z	означает	ADD :x,:y,:z
PREFIX	Foo:	текущим префиксом является	:Foo:
ADD	x,y,z	означает	ADD :Foo:x,:Foo:y,:Foo:z
PREFIX	Bar:	текущим префиксом является	:Foo:Bar:
ADD	:x,y,:z	означает	ADD :x,:Foo:Bar:y,:z
PREFIX	:	текущим префиксом является	:
ADD	x,Foo:Bar:y,Foo:z	означает	ADD :x,:Foo:Bar:y,:Foo:z

Один их способов использования этой идеи является замена первых строк в (16) на

```

PREFIX Max:
j IS $0 ;m IS $1 ;kk IS $2 ;xk IS $3
x0 IS :x0 ;b IS :b ;t IS :t      Внешние символы.
:Max100 SET      $0,100   Вход для n = 100.
:Max      SL      kk,$0,3  Вход для произвольного n.
          LD0      m,x0,kk
          JMP      1F
          ...
(Продолжить, как в (16)).

```

(18)

и включение "PREFIX :" в конце. Далее символы j, m, kk и xk можно использовать в остальной части программы или в определении других подпрограмм. Другие примеры использования префиксов рассматриваются в разделе 1.4.3'.

В MMIXAL также предусмотрен псевдооператор LOCAL. Команда ассемблера "LOCAL \$40" означает, например, что сообщение об ошибке появится в конце сборки, если команды GREG выделяют так много регистров, что \$40 будет глобальным. (Этот компонент необходим только, если подпрограмма использует более 32 локальных регистров, поскольку выражение "LOCAL \$31" всегда неявно истинно.)

Предусмотрен также третий компонент поддержки подпрограммы, BSPEC ... ESPEC. Это позволяет передавать информацию объектному файлу так, чтобы процедуры отладки и другие системные программы знали о типах связывания, которые используются каждой подпрограммой. Этот компонент обсуждается в документации MMIXware и представляет особый интерес при компилировании программ.

Стратегические соображения. В специализированных подпрограммах инструкции GREG можно использовать более свободно, чтобы заполнить глобальные регистры основными константами для ускорения работы программы. В таких случаях требуется сравнительно небольшое количество локальных регистров, если только подпрограммы не используются рекурсивно.

Но если десятки или сотни подпрограмм общего назначения создаются для включения в крупную библиотеку, которая может использоваться произвольной пользовательской программой, то, очевидно, нельзя для каждой такой подпрограммы выделять большое количество глобальных регистров. Даже по одной глобальной переменной на одну подпрограмму может оказаться чересчур много.

Таким образом, инструкцию GREG следует применять часто только при малом и очень экономно при большом количестве подпрограмм. В последнем случае без значительной утраты эффективности можно использовать локальные переменные.

В завершение раздела кратко рассмотрим процесс создания сложной и длинной программы. Какой тип подпрограмм нужно использовать в ней? Какие вызывающие последовательности использовать? Успешный способ решения этих задач заключается в использовании следующей итеративной процедуры:

Этап 0 (Исходная идея). Сначала нужно выбрать общий план программы.

Этап 1 (Грубый эскиз программы). Начните с создания “внешних уровней” программы на любом удобном языке. Один из систематических способов создания эскиза прекрасно описан в книге E. W. Dijkstra, *Structured Programming* (Academic Press, 1972), Chapter 1, и в статье N. Wirth, *CACM* 14 (1971), 221–227. Сначала вся программа разбивается на небольшое количество частей, которые временно можно рассматривать как подпрограммы, хотя они вызываются всего по одному разу. Эти части успешно разбиваются на все меньшие и меньшие части, соответственно выполняющие все более простую работу. Всякий раз, когда появляется некое похожее или повторяющееся вычисление, следует определить подпрограмму (теперь уже настоящую), которая выполнит это вычисление. На этом этапе код подпрограммы еще не создается, а продолжается работа с основной программой в предположении, что подпрограмма уже выполнила свою задачу. Наконец, после создания эскиза основной программы можно приступить к созданию подпрограмм, начиная с самых сложных и деля их на меньшие подпрограммы, и т.д. Продолжая в этой манере получаем список подпрограмм. Фактическая функция каждой подпрограммы уже, вероятно, могла измениться несколько раз так, что первые наброски эскиза могут оказаться неверными. Однако это не представляет большой проблемы, поскольку идет работа над общим эскизом. В результате этой работы создается представление о каждой вызываемой подпрограмме и ее общем предназначении. Следует учесть возможность даже минимального обобщения каждой подпрограммы.

Этап 2 (Первая работающая программа). На следующем этапе следует двигаться в противоположном направлении от этапа 1. Теперь можно приступить к созданию программы на компьютерном языке, например на языке MMIXAL или PL/MMIX, или — что более вероятно — языке более высокого уровня. Начнем с программирования подпрограмм самого низкого уровня, а код основной программы создадим в самую последнюю очередь. По мере возможности не рекомендуется создавать любые инструкции, которые вызывают подпрограмму до ее кодирования. (На этапе 1 мы поступали наоборот, т.е. не рассматривали подпрограмму до тех пор, пока не будут созданы все ее вызовы.)

По мере создания все большего числа подпрограмм уверенность программиста растет, поскольку по мере программирования растет вычислительная мощь компьютера. После создания кода отдельной подпрограммы следует немедленно подготовить полное описание выполняемых ею действий, а также ее вызывающих последовательностей, как в (10). Важно убедиться, что глобальные переменные не используются одновременно для двух конфликтующих целей. При подготовке эскиза на этапе 1 об этих проблемах не стоит беспокоиться.

Этап 3 (Повторная проверка). Результатом этапа 2 будет практически готовая работающая программа, но ее все-таки нужно попробовать улучшить. Для этого рекомендуется пойти в обратном направлении, изучая все места вызова каждой подпрограммы. Возможно подпрограмму нужно увеличить для выполнения более

общих действий, которые выполняются участками кода до и после вызова подпрограммы. Может быть, несколько подпрограмм лучше было бы объединить в одной подпрограмме. Иногда подпрограмма вызывается всего один раз и ее вовсе не следует делать подпрограммой. В результате анализа может оказаться, что некая подпрограмма никогда не вызывается и ее можно удалить.

На этом этапе рекомендуется тщательно все проверить и снова начать с этапа 1 или даже с этапа 0! Это никакая не шутка, поскольку время, затраченное на эти действия, не будет потрачено зря, ведь таким образом можно досконально изучить данную проблему. Оглядываясь на пройденный путь, вам, вероятно, удастся найти несколько улучшений в общей организации программы. Нет никаких оснований для опасений при возврате к этапу 1, и еще меньше их будет при повторном проходе от этапа 2 к этапу 3, ведь с подобной программой уже приходилось иметь дело. Более того, вполне вероятно эта работа позднее позволит сэкономить время отладки, когда снова придется переписывать код программы. Некоторые наиболее удачные компьютерные программы стали успешными только благодаря случайной утрате всего кода приблизительно на этом этапе работы, и их авторам приходилось начинать все сначала.

С другой стороны, вероятно, не существует такой сложной компьютерной программы, которую нельзя было бы улучшить, а потому не следует до бесконечности повторять этапы 1 и 2. После начальных значительных улучшений, в конечном итоге, всегда наступает момент, когда получаемая выгода не оправдывает затраченного времени.

Этап 4 (Отладка). После окончательной полировки программы, которая вероятно включает выделение памяти и другие детали, пора обратить внимание на другие вопросы, которые не учитывались на этапах 1, 2 и 3. Теперь нужно проанализировать программу в том порядке, в котором она *выполняется* компьютером. Это можно сделать вручную или, конечно же, с помощью компьютера. Автор считает весьма полезным на этом этапе использовать системные процедуры для трассировки каждой инструкции во время нескольких первых попыток их выполнения. Здесь важно еще раз представить идеи, лежащие в основе программы и проверить как они соответствуют выполняемым действиям.

Отладка — это искусство, которое требует отдельного изучения. Сам процесс отладки очень сильно зависит от имеющихся на данном компьютере соответствующих инструментов. Для эффективной отладки рекомендуется использовать подходящие проверочные данные. Наиболее успешные методы отладки обычно продумываются и встраиваются в саму программу. Многие наиболее успешные современные программисты посвящают почти половину своей программы процедурам отладки другой половины программы. Первая часть программы, которая обычно состоит из очень простых частей, отображающих информацию в читабельном формате, казалось бы не имеет большого значения, но в общем итоге это дает значительное увеличение производительности.

Еще один эффективный метод отладки заключается в том, чтобы учитывать все замеченные ошибки. Даже если эта информация порой выглядит очень неприглядно, она имеет бесценное значение для отладки программы и, несомненно, поможет справиться с подобными ошибками в будущем.

Примечание: автор впервые сформулировал эти соображения в 1964 году, после успешного завершения нескольких проектов среднего размера по созданию программного обеспечения, но еще до создания зрелого стиля программирования. Позднее, в 1980-х годах, автор узнал о существовании другой технологии, *структурного документирования*, или *грамотного программирования*, которая, вероятно, имеет большее значение. Итоговая сводка современных представлений автора о наилучших способах создания программ разных видов представлена в его книге *Literate Programming* (Cambridge University Press, впервые опубликована в 1992). Совершенно случайно в главе 11 этой книги приводится подробный перечень всех ошибок, которые удалось найти и устранить в программе TeX в 1978–1991 годы.

Уж лучше программировать с ошибками (до некоторой степени),
чем тратить столько времени на проектирование безошибочных программ
(сколько еще десятилетий потребуется для этого?).

— А. М. ТЬЮРИНГ (A. M. TURING), Предложения для ACE (Proposals for ACE) (1945)

УПРАЖНЕНИЯ:

1. [20] Создайте подпрограмму **GoMaxR**, которая обобщала бы алгоритм 1.2.10M поиска максимального значения среди $\{X[a], X[a + r], X[a + 2r], \dots, X[n]\}$, где r и n являются положительными параметрами и a является наименьшим положительным числом $a \equiv n$ (по модулю r), а именно $a = 1 + (n - 1) \bmod r$. Создайте специальный вход **GoMax** для случая $r = 1$, с помощью вызывающей последовательности на основе **G0**, чтобы полученная подпрограмма была обобщением (7).
2. [20] В подпрограмме из упр. 1 замените связывание на основе инструкции **G0** на другой тип связывания на основе инструкций **PUSHJ/POP**.
3. [15] Как можно было бы упростить схему (15), если известно, что **Sub** является концевой подпрограммой?
4. [15] В этом разделе в основном говорится о **PUSHJ**, а в разделе 1.3.1' упоминается только команда **PUSHG0**. Чем отличаются **PUSHJ** и **PUSHG0**?
5. [0] Истинно или ложно следующее утверждение: количество маргинальных регистров равно $G - L$.
6. [10] Каким будет результат выполнения инструкции **DIVU \$5,\$5,\$5**, если \$5 является маргинальным регистром?
7. [10] Каким будет результат выполнения инструкции **INCML \$5,#abcd**, если \$5 является маргинальным регистром?
8. [15] Предположим, что инструкция **SET \$15,0** выполняется при наличии 10 локальных регистров. Это приводит к увеличению числа локальных регистров до 16, но новые локальные регистры (включая \$15) равны нулю, потому что они ведут себя так, как если бы были маргинальными. Является ли инструкция **SET \$15,0** избыточной в этом случае?
9. [28] Если прерывание обхода используется для такой исключительной ситуации, как арифметическое переполнение, то обработчик обхода может вызываться в непредсказуемые моменты времени. В таких случаях нежелательно засорять регистры прерванной программы. Обработчик обхода не может сильно навредить, если только не давать ему "где развернуться". Объясните, как можно было бы использовать инструкции **PUSHJ** и **POP**, чтобы достаточное количество локальных регистров было доступно для обработчика.

- 10. [20] Истинно или ложно следующее утверждение: если программа MMIX никогда не использует инструкции PUSHJ, PUSHGO, POP, SAVE или UNSAVE, то все 256 регистров \$0, \$1, ..., \$255 эквивалентны в том смысле, что нет различий между локальными, глобальными и маргинальными регистрами.
- 11. [20] Что случится, если программа попытается выполнить больше инструкций POP, чем инструкций PUSH.
- 12. [10] Истинно или ложно следующее утверждение:
 - а) Текущий префикс в программе MMIXAL всегда начинается с двоеточия.
 - б) Текущий префикс в программе MMIXAL всегда заканчивается двоеточием.
 - с) Символы : и :: эквивалентны в программе MMIXAL.
- 13. [21] Создайте две подпрограммы MMIX для вычисления чисел Фибоначчи $F_n \bmod 2^{64}$ для заданного n .

Первая подпрограмма должна рекурсивно вызывать сама себя, согласно определению:

$$F_n = n \quad \text{если } n \leq 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{если } n > 1.$$

Вторая подпрограмма *не* должна быть рекурсивной. Обе подпрограммы должны использовать связывание на основе инструкций PUSH/POP и не использовать глобальных переменных.

- 14. [M21] Каково время выполнения подпрограмм в упражнении 13?
- 15. [21] Преобразуйте рекурсивную подпрограмму из упражнения 13 для создания подпрограммы со связыванием на основе инструкций GO, используя стековую память, как в (15) вместо стека регистров MMIX. Сравните эффективность двух версий.
- 16. [25] (*Нелокальный оператор goto.*) Иногда требуется перейти из подпрограммы в место, расположенное вне вызывающей процедуры. Предположим, что подпрограмма А вызывает подпрограмму В, которая вызывает подпрограмму С, которая рекурсивно вызывает сама себя несколько раз перед тем, как решить выйти непосредственно в подпрограмму А. Объясните, как можно было бы обработать такую ситуацию с помощью стека регистров MMIX (Нельзя просто так перейти с помощью инструкции JMP из С в А; стек должен быть обработан соответствующим образом.)

1.4.2'. Сопрограммы

Подпрограммы являются особыми случаями более общих программных компонентов, *сопрограмм*. В отличие от несимметричной связи между основной программой и подпрограммой, между сопрограммами имеется полная симметрия, которая выражается в том, что они могут *вызывать друг друга*.

Для понимания концепции сопрограммы, рассмотрим следующую ситуацию. В предыдущем разделе предлагалась точка зрения, что подпрограмма является всего лишь своеобразным расширением аппаратного обеспечения компьютера, предназначенного для экономии размера кода программы. Это верно, но существует и другая точка зрения: основную программу и подпрограмму можно считать *командой* программ, где каждый член команды имеет определенный участок работы. Основная программа на своем участке работы, активизирует подпрограмму, а подпрограмма выполняет свои функции и активизирует основную программу. Расширим рамки своего воображения и представим эту ситуацию с точки зрения подпрограммы: при выходе подпрограммы *она* вызывает *основную* программу, а основная программа продолжает свою работу и затем “выходит” в подпрограмму.

В свою очередь, подпрограмма по окончании своей работы снова вызывает основную программу.

Такой равноправный философский подход может показаться притянутым за уши, но он справедлив по отношению к сопрограммам. Дело в том, что при работе с сопрограммами нельзя сказать, какая из них подчиняется другой. Допустим, что программа состоит из сопрограмм А и В, тогда при программировании А сопрограмма В считается подпрограммой, а при программировании В сопрограмма А считается подпрограммой. Всякий раз при активизации “подчиненной” сопрограммы, выполнение “основной” сопрограммы продолжается с места ее приостановки.

Сопрограммы А и В могут, например, быть двумя программами-игроками в шахматы, т.е. их можно скомбинировать так, чтобы они играли в шахматы друг с другом.

Такое связывание сопрограмм легко осуществляется в MMIX, если применить два глобальных регистра, а и b. В сопрограмме А инструкция “GO a,b,0” используется для активизации сопрограммы В, а в сопрограмме В инструкция “GO b,a,0” используется для активизации сопрограммы А. В этой схеме для передачи управления в любом направлении требуется всего 3u единиц времени.

Существенное отличие между связыванием программы-подпрограммы и сопрограммы-сопрограммы можно обнаружить, сравнивая связывание на основе GO из предыдущего раздела со следующей схемой. Подпрограмма всегда запускается с ее начала, которое располагается в заданном месте, а основная программа или сопрограмма всегда запускается с места после ее последнего прерывания.

Сопрограммы наиболее естественным образом возникают на практике при работе с алгоритмами ввода и вывода. Например, допустим, что сопрограмма А должна считать файл и выполнить некоторые преобразования входных данных, сводя их к последовательности элементов. Другая сопрограмма, назовем ее В, выполняет дальнейшую обработку этих элементов и выводит полученный результат. Сопрограмма В периодически вызывает последовательные элементы входных данных с помощью А. Таким образом, сопрограмма В переходит к сопрограмме А всякий раз, когда нужен следующий элемент входных данных, а сопрограмма А переходит к сопрограмме В всякий раз, когда найден следующий элемент входных данных. Читатель может заметить: “Хорошо, В является основной программой и А является всего лишь *подпрограммой* для предоставления входных данных”. Это замечание, однако, становится менее справедливым, если процесс А заметно усложнится. Действительно, наоборот можно представить, что А является основной программой, В является подпрограммой для предоставления выходных данных, а приведенное выше описание остается справедливым. Польза от введения понятия “сопрограмма” проявляется на стыке между этими двумя крайними ситуациями, когда А и В имеют сложную структуру и каждая из них многократно вызывает другую. Не легко найти короткий и простой пример использования сопрограмм, которые иллюстрировали бы значение этой идеи, поскольку наиболее полезные сопрограммы обычно имеют довольно большой размер.

Для более тщательного изучения сопрограмм рассмотрим следующий притянутый за уши пример. Предположим, что нужно создать программу, которая транслирует один код в другой. Входной код, который нужно транслировать в последо-

вательность 8-битовых символов, заканчивается точкой, как показано ниже

a2b5e3426fg0zyw3210pq89r. (1)

Этот код находится в стандартном входном файле, который перемежается “пробельными символами” произвольного вида. В данном случае “пробельным символом” будем считать любой байт, значение которого меньше или равно #20, которое является символьной константой ' ' в программах MMIXAL. При считывании входных данных все символьные пробелы игнорируются, а другие символы интерпретируются следующим образом: (1) если следующий символ является одной из десятичных цифр 0 или 1 или ... или 9, например n , то нужно $(n + 1)$ раз повторить следующий символ, не важно, является он цифрой или нет. (2) нецифровой символ просто воспроизводится. Выход программы должен состоять из последовательности разделенной на группы по три символа до тех пор, пока не появится точка. Последняя группа может иметь меньше трех символов. Например, (1) транслируется в

" abb bee eee e44 446 66f gzy w22 220 0pq 999 999 999 r. (2)

Обратите внимание, что код 3426f означает не 3427 букв f, а 4 четверки, 3 шестерки и одну букву f. Если входной код имеет вид '1.', то на выходе будет '.', а не '1.', поскольку точка прекращает вывод. Цель нашей программы состоит в создании последовательности строк в стандартном выходном файле с 16 трехсимвольными группами в строке (за исключением, конечно, последней строки, которая может быть короче). Трехсимвольные группы нужно разделять пробелами, а каждую строку завершать символом ASCII новой строки #a.

Для выполнения этой трансляции создадим две сопрограммы и одну подпрограмму. Программа начинается с присвоения символьных имен трем глобальным регистрам: одно для временного хранения и остальные для связывания сопрограмм.

01 * Пример сопрограмм.

02	t	IS	\$255	Временные данные кратковременного хранения.
03	in	GREG	0	Адрес для продолжения первой сопрограммы.
04	out	GREG	0	Адрес для продолжения второй сопрограммы. █

Следующий шаг заключается в определении участков памяти для хранения рабочих данных.

05 * Входной и выходной буферы.

06		LOC	Data_Segment	
07		GREG	@	Базовый адрес.
08	OutBuf	TETRA	"	", #a, 0 (см. упражнение 3)
09	Period	BYTE	'.'	
10	InArgs	OCTA	InBuf, 1000	
11	InBuf	LOC	#100	█

Теперь вернемся к самой программе. Подпрограмма NextChar предназначена для поиска непробельных символов входа и возвращения следующего такого символа:

12 * Подпрограмма символьного входа

13	inptr	GREG	0	Текущая входная позиция.
14	1H	LDA	t, InArgs	Заполнить входной буфер.

15		TRAP	0,Fgets,StdIn	
16		LDA	inptr,InBuf	Стартовать в начале буфера.
17	OH	GREG	Period	
18		CSN	inptr,t,0B	Если произошла ошибка, считать '.'
19	NextChar	LDBU	\$0,inptr,0	Извлечь следующий символ.
20		INCL	inptr,1	
21		BZ	\$0,1B	Условный переход, если конец буфера.
22		CMPU	t,\$0,' '	
23		BNP	t,NextChar	Условный переход, если пробельный символ.
24		POP	1,0	Вернуться в вызывающую программу. █

Эта подпрограмма обладает следующими характеристиками:

Вызывающая последовательность: `PUSHJ $R,NextChar`.

Условия входа: `inptr` указывает на первый несчитанный символ.

Условия выхода: `$R` = следующий непобельный символ;
`inptr` готов для следующего элемента `NextChar`.

Подпрограмма также изменяет регистр `t`, а именно регистр `$255`, но мы обычно опускаем этот регистр в таких спецификациях, как это было в 1.4.1'-(10).

Наша первая сопрограмма, `In`, находит символы во входном коде и определяет количество повторов. Она начинает работу с позиции `In1`:

25	* Первая сопрограмма			
26	count	GREG	0	Счетчик повторов.
27	1H	GO	in,out,0	Послать символ сопрограмме <code>Out</code> .
28	In1	PUSHJ	\$0,NextChar	Получить новый символ.
29		CMPU	t,\$0,'9'	
30		PBP	t,1B	Условный переход, если превышает '9'.
31		SUB	count,\$0,'0'	
32		BN	count,1B	Условный переход, если меньше '0'.
33		PUSHJ	\$0,NextChar	Получить новый символ.
34	1H	GO	in,out,0	Передать его в <code>Out</code> .
35		SUB	count,count,1	Уменьшить счетчик повторов.
36		PBNN	count,1B	Повторить, если необходимо.
37		JMP	In1	В противном случае начать новый цикл. █

Эта сопрограмма обладает следующими характеристиками.

Вызывающая последовательность `GO out,in,0`.

(из `Out`):

Условия выхода (в `Out`): `$0` = следующий входной символ
с определенным числом повторов.

Условия входа

(после возвращения): `$0` не отличается от значения на выходе.

Регистр `count` является частным для `In` и нуждается в упоминании.

Другая сопрограмма, `Out`, размещает код в трехсимвольных группах и посылает их в стандартный выходной файл. Она начинает работу с позиции `Out1`:

38	* Вторая сопрограмма			
39	outptr	GREG	0	Текущая позиция выхода.
40	1H	LDA	t, OutBuf	Опустошить выходной буфер.
41		TRAP	0, Fputs, StdOut	
42	Out1	LDA	outptr, OutBuf	Стартовать с начала буфера.
43	2H	GO	out, in, 0	Получить новый символ из In.
44		STBU	\$0, outptr, 0	Сохранить его как первый из трех.
45		CMP	t, \$0, '.'	
46		BZ	t, 1F	Условный переход, если '.'
47		GO	out, in, 0	В противном случае получить другой символ.
48		STBU	\$0, outptr, 1	Сохранить его как второй из трех.
49		CMP	t, \$0, '.'	
50		BZ	t, 2F	Условный переход, если '.'
51		GO	out, in, 0	В противном случае получить другой символ.
52		STBU	\$0, outptr, 2	Сохранить его как третий из трех.
53		CMP	t, \$0, '.'	
54		BZ	t, 3F	Условный переход, если '.'
55		INCL	outptr, 4	В противном случае перейти к следующей группе.
56	0H	GREG	OutBuf+4*16	
57		CMP	t, outptr, 0B	
58		PBNZ	t, 2B	Условный переход, если меньше 16 групп.
59		JMP	1B	В противном случае завершить строку.
60	3H	INCL	outptr, 1	Пройти мимо сохраненного символа.
61	2H	INCL	outptr, 1	Пройти мимо сохраненного символа.
62	0H	GREG	#a	(символ новой строки)
63	1H	STBU	0B, outptr, 1	Сохранить символ новой строки после точки.
64	0H	GREG	0	(пустой символ)
65		STBU	0B, outptr, 2	Сохранить пустой символ после новой строки.
66		LDA	t, OutBuf	
67		TRAP	0, Fputs, StdOut	Вывести последнюю строку.
68		TRAP	0, Halt, 0	Завершить программу. █

Характеристики Out задуманы такими, чтобы они дополняли характеристики In.

Вызывающая последовательность (из In): GO in, out, 0.

Условия выхода (в In): \$0 не отличается от значения на входе.

Условия входа

(upon return):

\$0 = следующий входной символ с определенным числом повторов.

Для завершения программы нужно задать условия запуска. Инициализация сопрограммы имеет замысловатый вид, хотя, на самом деле, она не так сложна, как кажется.

69 * Инициализация

70	Main	LDA	inptr, InBuf	Инициализировать NextChar.
71		GETA	in, In1	Инициализировать In.
72		JMP	Out1	Стартовать с Out (см. упражнение 2). █

Работа над программой завершена. Читателю рекомендуется тщательно изучить ее, обращая особое внимание на то, как каждая сопрограмма может считы-

ваться и создаваться независимо, как если бы другая сопрограмма была ее подпрограммой.

Из раздела 1.4.1' известно, что инструкции PUSHJ и POP компьютера MMIX предпочтительнее, чем команда GO, при организации связывания подпрограмм. Но для сопрограмм верно противоположное утверждение: инструкции PUSHJ и POP очень несимметричны, а стек регистра компьютера MMIX может быть полностью запутан, если две или более сопрограмм попытаются одновременно использовать его. (См. упражнение 6.)

Между сопрограммами и *многопроходными алгоритмами* есть важная связь. Например, описанный выше процесс трансляции можно выполнить за два прохода. Сначала можно было бы выполнить сопрограмму In, применяя ее ко всему входу и записывая каждый символ с определенным количеством повторов в промежуточный файл. После этого можно было бы считать файл и выполнить сопрограмму Out, принимая символы в группах по три элемента. В этом состоит суть “двухпроходного” процесса. (Интуитивно понятно, что “проход” обозначает полное сканирование входа. Это определение не совсем точно и во многих алгоритмах не ясно, какое количество проходов используется, но интуитивная концепция “прохода” все же полезна, несмотря на ее нечеткое определение.)

На рис. 22а показан четырехпроходный процесс. Довольно часто можно обнаружить, что тот же процесс можно выполнить всего за один проход, как показано на рис. 22б, если подставить четыре сопрограммы A, B, C, D для соответствующих проходов A, B, C, D. Сопрограмма A перейдет к сопрограмме B, вместо записи результата прохода A в файл 1. Сопрограмма B перейдет к сопрограмме A, вместо чтения входа из Файла 1, потом B перейдет к сопрограмме C, вместо записи результата прохода в Файл 2; и т.д. Пользователям UNIX® эта конструкция известна под названием “канал” или “конвейер” и обозначается “Проход A | Проход B | Проход C | Проход D”. Программы для проходов B, C и D иногда называются “фильтрами”.

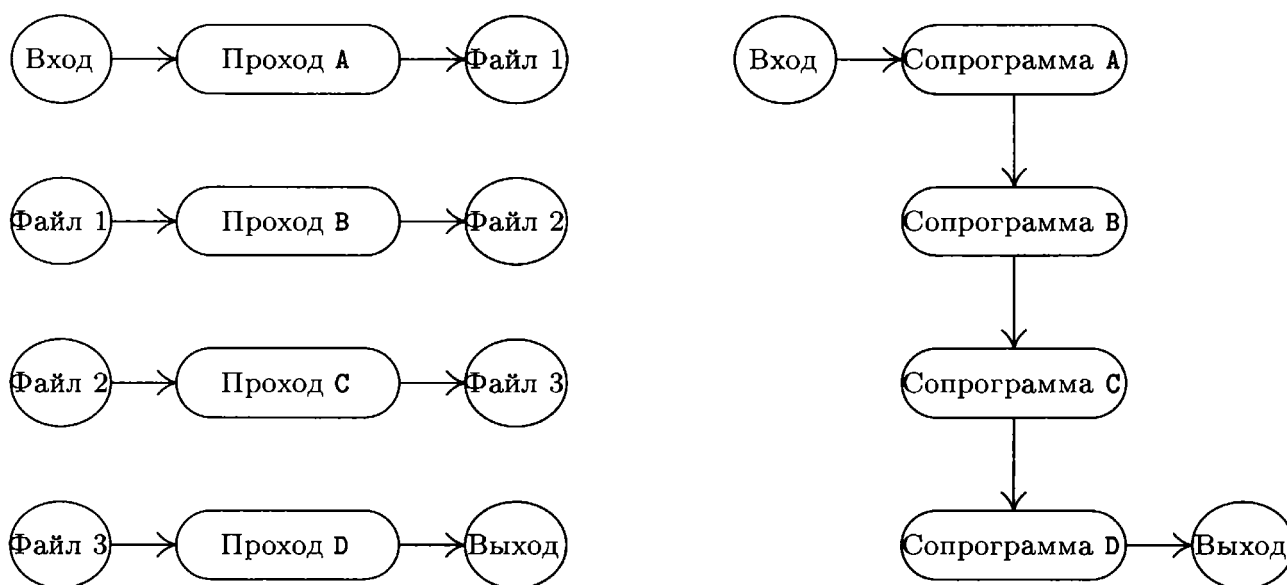


Рис. 22. Проходы: а — четырехпроходной алгоритм; б — однопроходной алгоритм

И наоборот, процесс, выполняемый n соппрограммами, часто трансформируется в n -проходный процесс. В связи с этим, имеет смысл сравнить многопроходные и однопроходные алгоритмы.

а) *Психологическое отличие.* Многопроходный алгоритм обычно проще создать и понять, чем однопроходный алгоритм для той же проблемы. Процесс, разделенный на несколько малых этапов, которые идут друг за другом, легче для понимания, чем единый сложный процесс, в котором одновременно происходит несколько преобразований.

Кроме того, если имеется крупная задача и для создания программы для ее решения предполагается коллективная работа многих людей, то многопроходной алгоритм представляет собой естественный способ разделения участков работы между ними.

Преимущества многопроходного алгоритма выражаются также в соппрограммах, поскольку каждая соппрограмма может создаваться отдельно от других соппрограмм. Их связывание превращает кажущийся многопроходный алгоритм в однопроходный процесс.

б) *Временное отличие.* Время, необходимое на упаковку, создание, чтение и распаковку промежуточных данных между потоками (например, в виде файлов на рис. 22), исключается в однопроходном алгоритме. По этой причине, однопроходный алгоритм будет выполняться быстрее.

с) *Пространственное отличие.* При использовании однопроходного алгоритма требуется пространство в памяти для одновременного хранения всех программ, а при использовании многопроходного алгоритма требуется пространство в памяти для одновременного хранения только одной соппрограммы. Эта особенность может повлиять на скорость выполнения программы еще в большей степени, чем указано в пункте (б). Например, многие компьютеры обладают ограниченным объемом "быстрой памяти" и большим объемом более медленной памяти. Если каждый проход едва помещается в быстрой памяти, то время выполнения многопроходного алгоритма будет гораздо меньше. (Дело в том, что использование соппрограмм может привести к тому, что большая часть программы окажется в медленной памяти или будет часто сбрасываться в/из быстрой памяти).

Иногда возникает необходимость спроектировать разные алгоритмы для нескольких компьютерных конфигураций с разным объемом памяти. В таких случаях можно написать программу в виде нескольких соппрограмм и, в зависимости от размера памяти, управлять количеством проходов: загружать приемлемое количество соппрограмм, а для организации отсутствующих связей применить подпрограммы ввода или вывода.

Хотя взаимосвязь между соппрограммами и проходами имеет большое значение, следует иметь в виду, что для приложений с соппрограммами не всегда можно реализовать многопроходной алгоритм. Если соппрограмма В получает входные данные от соппрограммы А и отправляет назад важную информацию соппрограмме А, как в примере с шахматами, то эту последовательность действий нельзя представить в виде прохода А, за которым следует проход В.

Наоборот, ясно, что некоторые многопроходные алгоритмы нельзя преобразовать в соппрограммы. Некоторые алгоритмы по своей природе являются многопро-

ходными. Например, для второго прохода может потребоваться такая обобщенная информация из первого прохода, как общее количество появления определенного слова во входном потоке. Вот старый анекдот, который стоит упомянуть в этом контексте.

Старушка в автобусе: “Малыш, на какой остановке нужно выходить, чтобы попасть на улицу Пасадена-Стрит?”

Малыш: “Просто следите за мной и выходите на две остановки раньше той, на которой выйду я.”

(Суть анекдота в том, что малыш предлагает двухпроходной алгоритм для решения данной проблемы.)

Вот и все о многопроходных алгоритмах. Сопрограммы также играют важную роль в симуляции дискретных систем (см. раздел 2.2.5). Если несколько более или менее независимых сопрограмм управляются основным процессом, то их часто называют *потоками* вычислений. Другие примеры сопрограмм часто рассматриваются в других частях данной книги. В главе 8 рассматривается очень важная идея *реплицируемых сопрограмм*, а некоторые интересные приложения этой идеи представлены в главе 10.

УПРАЖНЕНИЯ:

1. [10] Объясните, почему трудно найти простые примеры сопрограмм.
- ▶ 2. [20] Программа в этом разделе начинается с сопрограммы `Out`. Что произойдет, если она начнется с сопрограммы `In`, т.е., если строки 71 и 72 будут иметь вид `“GETA out,Out1; JMP In1”`?
3. [15] Объясните предназначение инструкции `TETRA` в строке 08 программы в этом разделе. (Обратите внимание, что между кавычками находится 15 пробелов.)
4. [20] Допустим, что две сопрограммы `A` и `B` рассматривают регистр остатка `rR` компьютера `MMIX` так, как если бы он был бы их закрытым элементом, хотя обе они вовлечены в процесс деления. (Иначе говоря, если одна сопрограмма передает управление другой, то содержимое `rR` не должно меняться, если управление возвращается другой сопрограмме.) Придумайте такое связывание сопрограмм, при котором соблюдалось бы это условие.
5. [20] Можно ли для `MMIX` создать достаточно эффективное связывание сопрограмм с помощью инструкций `PUSH` и `POP` без использования команд `GO`?
6. [20] В программе для `MMIX` в данном разделе стек регистров используется только очень ограниченным способом, а именно, когда `In` вызывает `NextChar`. В какой степени две взаимодействующие сопрограммы могут совместно использовать стек регистров?
- ▶ 7. [30] Создайте программу `MMIX`, которая *обращает* трансляцию, выполненную программой в данном разделе. Иначе говоря, эта программа должна преобразовывать файл с трехсимвольными группами (2) в файл с кодом (1). Выход должен иметь вид максимально короткой строки символов, например, ноль перед `z` в (1) не должен входить в (2).

1.4.3'. Программы-интерпретаторы

В этом разделе рассматриваются довольно распространенные программы, которые часто называются *программами-интерпретаторами* или, короче, *интерпретаторами*. Интерпретатор — это компьютерная программа, которая выполняет инструкции другой программы, созданной на машинном языке. Под машинным

языком подразумевается способ представления инструкций на основе кодов операций (опкодов), адресов и т.д. (Это определение, как и большинство определений современных компьютерных терминов, неточно, но оно и не должно быть таким. Нельзя точно провести линию и сказать, что какие-то программы являются интерпретаторами, а какие-то — нет.)

Исторически первые интерпретаторы создавались для машинных языков, предназначенных специально для упрощения программирования. Такой интерпретируемый язык проще в употреблении, чем машинный язык. Вскоре появление символьных языков программирования исключило необходимость использования интерпретаторов такого рода, но сами интерпретаторы не исчезли. Наоборот, их использование продолжает расти, причем в такой степени, что эффективное использование интерпретаторов может рассматриваться, как одна из существенных характеристик современного программирования. Новые области приложения интерпретаторов образовались, в основном по следующим причинам:

- а) машинный язык способен представить сложную последовательность решений и действий в компактной и эффективной форме;
- б) такое представление предоставляет прекрасный способ сообщения между проходами в многопроходном процессе.

В таких случаях создаются специальные языки, похожие на машинные, которые используются для отдельной программы, а программы на этих языках часто генерируются только компьютерами. (Современные профессиональные программисты являются прекрасными дизайнерами компьютеров: они не только создают интерпретаторы, но и определяют *виртуальный компьютер*, язык которого нужно интерпретировать.)

Технология интерпретирования имеет еще одно дополнительное преимущество относительной независимости от компьютера. Дело в том, что только интерпретатор нужно изменить при переходе на другой тип компьютера. Более того, полезные средства отладки можно дополнительно встроить в интерпретируемые системы.

Примеры интерпретаторов типа (а) представлены в нескольких местах далее в этой серии книг. Например, рекурсивный интерпретатор в главе 8 и синтаксический анализатор (“Parsing Machine”) в главе 10. Обычно возникают ситуации, когда имеется большое количество особых случаев, все похожие, но не имеющие простой структуры.

Например, рассмотрим задачу создания алгебраического компилятора, в котором нужно эффективно генерировать инструкции машинного языка. Они складываются из двух величин, которые принадлежат одному из 10 классов (константы, простые переменные, индексированные переменные, переменные с фиксированной или плавающей точкой, знаковые или беззнаковые переменные и т.д.). Комбинации всех этих пар образуют 100 разных случаев. Для соответствующей обработки всех этих случаев потребуется создать очень сложную программу. Интерпретируемое решение этой задачи заключается в создании специального языка, чьи “инструкции” помещались бы в одном байте. Затем нужно подготовить таблицу 100 “программ” на этом языке, причем каждая программа умещается в одном слове. Идея заключается в том, чтобы выбрать соответствующий элемент таблицы и выполнить найденную там программу. Эта технология очень проста и эффективна.

Пример интерпретатора типа (b) представлен в статье “Computer-Drawn Flowcharts”, D. E. Knuth, *CACM* **6** (1963), 555–563. В многопроходной программе ранние проходы должны передавать информацию более поздним проходам. Эта информация часто более эффективно передается с помощью машинного языка, как набор инструкций для более позднего прохода. Более поздний проход часто представляет собой не более, чем интерпретатор особого назначения, а более ранний интерпретатор представляет собой “компилятор” особого назначения. Философия многопроходной операции может быть охарактеризована, как *пересказ* более позднему проходу последовательности действий при любой возможности, а не просто представление множества фактов и просьбу *сообразить* что сделать.

Другим примером интерпретатора типа (b) являются компиляторы особых языков программирования. Если язык включает много компонентов, которые не так легко выполнить на компьютере без использования подпрограмм, то результирующая объектная программа будет содержать длительную последовательность вызовов подпрограмм. Это может произойти, например, если язык имеет дело преимущественно с высокоточными арифметическими операциями. В таком случае объектная программа будет значительно короче, если выражается на интерпретируемом языке. Например, в книге *ALGOL 60 Implementation*, by B. Randell and L. J. Russell (New York: Academic Press, 1964) описывается компилятор для трансляции ALGOL 60 в интерпретируемый язык, а также описывается интерпретатор этого языка. В книге “An ALGOL 60 Compiler,” by Arthur Evans, Jr., *Ann. Rev. Auto. Programming* **4** (1964), 87–124, приводятся примеры интерпретаторов, используемых в компиляторе. Появление микропрограммируемых компьютеров и интегральных микросхем специального назначения еще более повысило значение интерпретируемых подходов.

Программа \TeX , с помощью которой создана эта книга, преобразует текст данного раздела в интерпретируемый язык, т.е. формат DVI, созданный Д. Р. Фуксом в 1979. [См. D. E. Knuth, *\TeX: The Program* (Reading, Mass.: Addison–Wesley, 1986), Part 31.] Полученный файл формата DVI, который создан с помощью \TeX , потом обрабатывается интерпретатором dvips, созданный Т. Г. Рокицки, и преобразуется в файл инструкций на другом интерпретируемом языке PostScript® [Adobe Systems Inc., *PostScript Language Reference*, 3rd edition (Reading, Mass.: Addison–Wesley, 1999)]. Файл формата PostScript передается в издательство для печати на принтере, который использует интерпретатор PostScript для создания печатных форм. Этот трехпроходной пример иллюстрирует интерпретаторы типа (b); \TeX также включает малый интерпретатор типа (a) для обработки так называемой лигатуры и кернинга печатаемых символов [*\TeX: The Program*, §545].

Программу на интерпретируемом языке можно представить следующим образом: ее можно рассматривать как набор вызовов подпрограмм, один вызов за другим. Такую программу можно расширить фактически в виде длинной последовательности вызовов подпрограмм. И наоборот, такую последовательность можно упаковать в кодированной и готовой для интерпретирования форме. Преимуществами технологии интерпретирования являются компактность представления, независимость от типа компьютеров и улучшенные возможности диагностики. Интерпретатор часто можно создать таким образом, чтобы время, потраченное на

интерпретацию кода и ветвление к соответствующей процедуре, было достаточно малым.

***Симулятор MMIX.** Если язык интерпретируемой процедуры является машинным языком другого компьютера, то интерпретатор часто называется *симулятором* (или иногда *эмулятором*).

По мнению автора, чересчур много времени программистов было потрачено зря на создание таких симуляторов и чересчур много компьютерного времени было потрачено зря на их использование. Мотивация такой деятельности очень проста: при покупке нового компьютера пользователь хотел бы использовать программы, созданные для прежнего устаревшего компьютера (а не переписывать программы). Однако это обычно дороже и менее эффективно, чем перепрограммирование нужных программ с помощью временной команды программистов. Например, автору ранее приходилось участвовать в подобном проекте перепрограммирования и в ходе работы была обнаружена серьезная ошибка в исходной программе, которая использовалась в течение нескольких лет. Новая программа работала в пять раз быстрее прежней и давала правильные результаты! (Не все симуляторы плохи. Например, производители компьютеров обычно стремятся симулировать поведение нового компьютера до его постройки, чтобы как можно быстрее создать программное обеспечение для него. Однако это очень специализированный способ применения симуляторов.) Вот реальный экстремальный пример неэффективного использования компьютерного симулятора: компьютер *A* симулирует компьютер *B*, который выполняет программу, симулирующую поведение компьютера *C*. В этом случае большой и дорогой компьютер будет менее эффективен, чем его более дешевый собрат.

В свете сказанного выше, почему симулятору уделяется внимание в данной книге? Для этого есть следующие три причины.

а) Описываемый ниже симулятор является прекрасным примером типичной интерпретируемой процедуры. На его примере иллюстрируются базовые технологии создания интерпретаторов и применение подпрограмм в сравнительно длинной программе.

б) Здесь описывается симулятор компьютера MMIX, созданный (представьте себе) на языке MMIX. Это улучшит наши знания о данном компьютере и поможет в создании симуляторов MMIX для других компьютеров, хотя мы не будем погружаться в подробности 64-битовой целочисленной арифметики или вычислений с плавающей точкой.

с) Симуляция MMIX объясняет способы эффективной организации стека регистров на аппаратном уровне так, чтобы проталкивание и выталкивание достигалось с наименьшими затратами. Аналогично, представленный здесь симулятор проясняет поведение операторов *SAVE* и *UNSAVE*, а также дает более подробное представление о прерываниях обхода. Такие конструкции становятся более понятными при изучении способов реализации ссылок и работы компьютера.

Описываемые здесь компьютерные симуляторы следует отличать от *дискретных системных симуляторов*. Дискретные системные симуляторы имеют большое значение и рассматриваются в разделе 2.2.5.

Вернемся к задаче создания симулятора MMIX. Начнем со значительного упрощения: вместо симуляции всех одновременно происходящих процессов в конвейерном компьютере мы будем интерпретировать одновременно только по одной инструкции. Конвейерная обработка имеет чрезвычайно большое значение, но ее изучение выходит за рамки этой книги. Заинтересованные читатели могут найти полный код программы для конвейерного “мета-симулятора” в документации *MMIXware*. Здесь мы ограничимся симулятором, которому не известно о кэш-памяти, трансляции виртуальных адресов, динамическом планировании инструкций, буферах восстановления последовательности и многом другом. Более того, здесь симулируются только те инструкции, которые используются в обычных пользовательских программах MMIX. Такие привилегированные инструкции, как LDVTS, которые зарезервированы для операционной системы, будут рассматриваться, как ошибочные, если они появятся. Прерывания обхода не будут симулироваться нашей программой, если только они не выполняют рудиментарные операции ввода или вывода, как описано в разделе 1.3.2'.

Входом нашей программы будет двоичный файл, в котором указано исходное состояние памяти, как если бы память была задана операционной системой при выполнении пользовательской программы (включая данные командной строки). Попробуем имитировать поведение аппаратной части MMIX, считая, что MMIX сам интерпретирует инструкции, которые начинаются с символьного обозначения Main. Таким образом, попробуем реализовать спецификации, представленные в разделе 1.3.1', в среде выполнения, которая описывается в разделе 1.3.2'. Например, в программе используется массив из 256 октабайт $g[0], g[1], \dots, g[255]$ для симулированных глобальных регистров. Первые 32 элемента этого массива будут специальными регистрами, перечисленными в табл. 1.3.1'–2; один из этих специальных регистров будет симулировать часы; гС. Предполагается, что для выполнения каждой инструкции требуется определенное время, согласно таблице 1.3.1'–1. Симулированные часы гС увеличивают свои показания на 2^{32} для каждого μ и на 1 для каждого v . Таким образом, например, после симулирования программы 1.3.2'Р симулированные часы гС будут иметь значение #00003228000b091, которое представляет $12840\mu + 766097v$.

Программа очень длинна, но она содержит несколько интересных мест, и мы кратко рассмотрим наиболее простые фрагменты. Как обычно, она начинается с определения новых символов и указания содержимого сегмента данных. В этом сегменте сначала располагается массив из 256 симулированных глобальных регистров. Например, симулированный глобальный регистр \$255 будет октабайтом $g[255]$ по адресу $\text{Global} + 8 * 255$. За этим глобальным массивом располагается аналогичный массив, который называется *кольцом локальных регистров*, где будут храниться самые верхние элементы симулированного стека регистров. Размер кольца равен 256, хотя 512 или еще большее значение в степени 2 также сгодится. (Большое кольцо локальных регистров потребует больше затрат, но оно дает выигрыш в скорости, если в программе часто используется стек регистров. Одно из предназначений симулятора состоит в определении целесообразности использования дополнительного аппаратного обеспечения.) Основная часть сегмента данных, начиная с *Chunk0*, посвящается симулированной памяти.

001	* Симулятор MMIX (упрощенный)			
002	t	IS	\$255	Регистр временной информации.
003	lring_size	IS	256	Размер кольца локальных регистров.
004		LOC	Data_Segment	Начать с адреса #2000 0000 0000 0000.
005	Global	LOC	@+8*256	256 октабайт для глобальных регистров.
006	g	GREG	Global	Базовый адрес глобальных регистров.
007	Local	LOC	@+8*lring_size	lring_size октабайт для локальных регистров.
008	l	GREG	Local	Базовый адрес для локальных регистров.
009		GREG	@	Базовый адрес для IOArgs и Chunk0.
010	IOArgs	OCTA	0,BinaryRead	(См. упражнение 20.)
011	Chunk0	IS	@	Начало области симулированной памяти.
012		LOC	#100	Остальное в текстовый сегмент. █

Одна из наиболее важных подпрограмм называется MemFind. Имея 64-битовый адрес A , эта подпрограмма возвращает результирующий результат R , по которому находится симулированное содержание $M_8[A]$. Конечно, 2^{64} байт симулированной памяти нельзя втиснуть в 2^{61} -байтовый сегмент данных. Но симулятор запоминает все встречавшиеся прежде адреса и предполагает, что все не встречавшиеся адреса равны нулю.

Память делится на “порции” по 2^{12} байт каждая. MemFind анализирует первые $64 - 12 = 52$ бит A , чтобы определить их принадлежность к порции, и расширяет список известных порций в случае необходимости. Затем вычисляется R за счет сложения 12 последних бит A со стартовым адресом соответствующей симулированной порции. (Размер порции должен быть степенью 2, если каждая порция содержит по крайней мере октабайт. При работе с малыми порциями MemFind приходится вести поиск в длинных списках подручных порций, а при работе с большими порциями MemFind приходится расходовать место для байтов, к которым никогда не будет осуществляться доступ.)

Каждая симулированная порция заключена в “узел”, который занимает $2^{12} + 24$ байт памяти. Первый октабайт такого узла, допустим KEY, идентифицирует симулированный адрес первого байта в порции. Второй октабайт, допустим LINK, указывает на следующий узел в списке MemFind, и он равен нулю для последнего узла в списке. За LINK следует 2^{12} байт симулированной памяти, DATA. Наконец, каждый узел завершается восемью нулевыми байтами, которые используются для дополнения при реализации ввода-вывода (см. упражнения 15–17).

MemFind поддерживает список узлов порций в порядке использования: первый узел, указанный head, является тем, который MemFind нашел в предыдущем вызове, а он связывается со следующей наиболее недавно использованной порцией и т.д. Если будущее подобно прошлому, то MemFind не должен искать далеко в списке. (В разделе 6.1 подробно обсуждается такая “самоорганизация” поиска в списках.) Сначала head указывает на порцию Chunk0, где KEY, LINK и DATA являются нулями. Указатель выделения alloc устанавливается в адрес, где появится узел следующей порции в случае необходимости, а именно $\text{Chunk0} + \text{nodesize}$.

MemFind реализуется с помощью операции PREFIX языка MMIXAL, описанной в разделе 1.4.1', чтобы зарезервированные (приватные) символы head, key, addr и т.д. не конфликтовали с любыми символами в остальной части программы.

Вызывающая последовательность будет иметь вид

SET arg,A; PUSHJ res,MemFind (1)

после которой результирующий адрес R появится в регистре res.

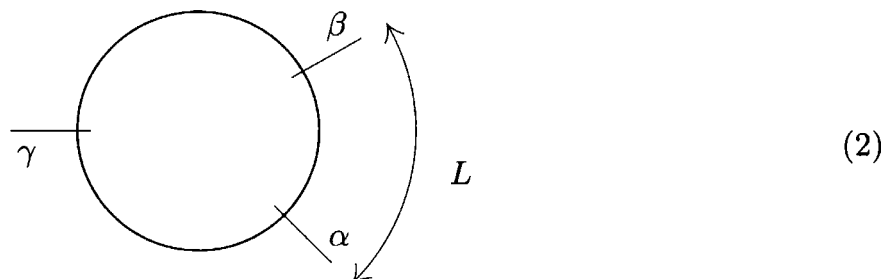
013		PREFIX	:Mem:	(Зарезервированные символы для MemFind.)
014	head	GREG	0	Адрес первой порции.
015	curkey	GREG	0	KEY(head)
016	alloc	GREG	0	Адрес следующей выделяемой порции.
017	Chunk	IS	#1000	Байты для порции, должны быть степенью 2.
018	addr	IS	\$0	Заданный адрес A.
019	key	IS	\$1	Адрес порции.
020	test	IS	\$2	Временный регистр для поиска ключа.
021	newlink	IS	\$3	Второй наиболее недавно использованный узел.
022	p	IS	\$4	Временный регистр указателя.
023	t	IS	:t	Внешний временный регистр.
024	KEY	IS	0	
025	LINK	IS	8	
026	DATA	IS	16	
027	nodesize	GREG	Chunk+3*8	
028	mask	GREG	Chunk-1	
029	:MemFind	ANDN	key,addr,mask	
030		CMPU	t,key,curkey	
031		PBZ	t,4F	Ветвление, если head является правой порцией.
032		BN	addr,:Error	Не допустить отрицательные адреса A.
033		SET	newlink,head	Подготовиться для петли поиска.
034	1H	SET	p,head	$p \leftarrow head$
035		LDQU	head,p,LINK	$head \leftarrow LINK(p)$
036		PBNZ	head,2F	Ветвление, если $head \neq 0$.
037		SET	head,alloc	В противном случае выделить новый узел.
038		STOU	key,head,KEY	
039		ADDU	alloc,alloc,nodesize	
040		JMP	3F	
041	2H	LDQU	test,head,KEY	
042		CMPU	t,test,key	
043		BNZ	t,1B	Возврат к началу цикла, если $KEY(head) \neq key$.
044	3H	LDQU	t,head,LINK	Настроить указатели: $t \leftarrow LINK(head)$,
045		STOU	newlink,head,LINK	$LINK(head) \leftarrow newlink$,
046		SET	curkey,key	$curkey \leftarrow key$,
047		STOU	t,p,LINK	$LINK(p) \leftarrow t$.
048	4H	SUBU	t,addr,key	$t \leftarrow \text{отступ порции}$.
049		LDA	\$0,head,DATA	$\$0 \leftarrow \text{адрес DATA(head)}$.
050		ADDU	\$0,t,\$0	
051		POP	1,0	Вернуть R.
052		PREFIX	:	(Конец префикса ':Mem:'.)
053	res	IS	\$2	Регистр результата PUSHJ.
054	arg	IS	res+1	Регистр аргумента PUSHJ. ■

Далее рассмотрим наиболее интересный аспект симулятора, а именно: реализацию стека регистров MMIX. Как известно из раздела 1.4.1', стек регистров

концептуально является списком τ элементов $S[0], S[1], \dots, S[\tau - 1]$. Заключительный член $S[\tau - 1]$ является “вершиной” стека, а локальные регистры MMIX $\$0, \$1, \dots, \$(L - 1)$ являются самыми верхними L элементами $S[\tau - L], S[\tau - L + 1], \dots, S[\tau - 1]$, где L является значением специального регистра rL . Стек можно было бы симулировать, просто полностью сохраняя его в симулированной памяти, но для эффективной работы компьютер должен иметь возможность мгновенного доступа к регистрам, а не к относительно медленному модулю памяти. Следовательно, будет симулироваться эффективная схема, которая хранит самые верхние элементы стека в массиве внутренних регистров, которые называются *кольцом локальных регистров*.

Основная идея чрезвычайно проста. Предположим, что кольцо локальных регистров имеет ρ элементов, $l[0], l[1], \dots, l[\rho - 1]$. Тогда локальный регистр $\$k$ будет храниться в $l[(\alpha + k) \bmod \rho]$, где α является соответствующим отступом. (Значение ρ выбрано так, чтобы быть степенью 2, чтобы для вычисления остатка от деления $\bmod \rho$ не требовалось прилагать больших усилий. Более того, ρ равно 256, по крайней мере, чтобы хватило места для всех локальных регистров.) Операция PUSH, которая перенумеровывает локальные регистры так, чтобы, например, регистр $\$3$ теперь назывался $\$0$, просто увеличивает значение α на 3. Операция POP восстанавливает предыдущее состояние за счет уменьшения α . Хотя регистры изменяют свое значение, на самом деле никакие данные не проталкиваются вниз и не выталкиваются вверх.

Конечно, потребуется использовать память для резервного копирования данных при возрастании размеров стека. Состояние кольца в любой момент времени лучше всего представить с помощью трех переменных, α , β и γ :



Элементы кольца $l[\alpha], l[\alpha + 1], \dots, l[\beta - 1]$ являются текущими локальными регистрами $\$0, \$1, \dots, \$(L - 1)$; элементы кольца $l[\beta], l[\beta + 1], \dots, l[\gamma - 1]$ не используются в данный момент; а элементы $l[\gamma], l[\gamma + 1], \dots, l[\alpha - 1]$ содержат протолкнутые вниз элементы стека регистров. Если $\gamma \neq \alpha$, то можно увеличить γ на 1, если сначала сохранить $l[\gamma]$ в памяти. Если $\gamma \neq \beta$, то можно уменьшить γ на 1, если затем загрузить $l[\gamma]$. В MMIX имеется два специальных регистра, которые называются *указателем стека* rS и *отступом стека* rO , хранящие адреса расположения $l[\gamma]$ and $l[\alpha]$, в случае необходимости. Значения α , β и γ связаны с rL , rS и rO формулами

$$\alpha = (rO/8) \bmod \rho, \quad \beta = (\alpha + rL) \bmod \rho, \quad \gamma = (rS/8) \bmod \rho. \quad (3)$$

Симулятор хранит большинство специальных регистров MMIX в первых 32 позициях массива глобальных регистров. Например, симулированный регистр остатка от деления rR является октабайтом, который находится по адресу $Global + 8 * rR$. Но восемь специальных регистров, включая rS , rO , rL и rG , потенциально относятся

к каждой симулированной инструкции, потому симулятор хранит их отдельно в собственных глобальных регистрах. Таким образом, например, регистр *ss* содержит симулированное значение *rS*, а регистр *ll* хранит восьмикратное симулированное значение *rL*:

```

055 ss GREG 0 Симулированный указатель стека, rS.
056 oo GREG 0 Симулированный отступ стека, rO.
057 ll GREG 0 Симулированный локальный регистр порога, rL, умноженный на 8.
058 gg GREG 0 Симулированный глобальный регистр порога, rG, умноженный на 8.
059 aa GREG 0 Симулированный регистр арифметического состояния, rA.
060 ii GREG 0 Симулированный счетчик интервала, rI.
061 uu GREG 0 Симулированный счетчик использования, rU.
062 cc GREG 0 Симулированный счетчик циклов, rC. █

```

Вот подпрограмма, которая получает текущее значение симулированного регистра $\$k$, given k . Вызывающая последовательность имеет вид

SLU arg,k,3; PUSHJ res,GetReg (4)

и искомое значение будет находиться в *res*.

```

063 lring_mask GREG 8*lring_size-1
064 :GetReg      CMPU  t,$0,gg      Подпрограмма для получения $k.
065             BN    t,1F          Условный переход, если k < G.
066             LDOU  $0,g,$0       В противном случае $k является
                                   глобальным; загрузить g[k].
067             POP   1,0           Вернуть результат.
068 1H          CMPU  t,$0,ll       t ← [$k is является локальным].
069             ADDU  $0,$0,oo
070             AND   $0,$0,lring_mask
071             LDOU  $0,1,$0       Загрузить l[(α + k) mod ρ].
072             CSNN  $0,t,0        Обнулить, если $k является
                                   маргинальным.
073             POP   1,0           Вернуть результат. █

```

Обратите внимание на двоеточие в поле метки строки 064. Оно избыточно, поскольку текущим префиксом является ':' (см. строку 052). Однако, двоеточие в строке 029 необходимо для внешнего символа *MemFind*, поскольку в тот момент текущим префиксом был '*Mem:*'. Двоеточия в поле метки, избыточны они или нет, Представляют собой удобный способ для оповещения того факта, что определяется подпрограмма.

Следующие подпрограммы, *StackStore* и *StackLoad*, симулируют операции увеличения γ на 1 и уменьшения γ на 1 в диаграмме (2). Они не возвращают результат. Подпрограмма *StackStore* вызывается, только если $\gamma \neq \alpha$, а подпрограмма *StackLoad* вызывается, только если $\gamma \neq \beta$. В обеих подпрограммах необходимо сохранять и восстанавливать регистр *rJ*, поскольку данные подпрограммы не являются концевыми.

```

074 :StackStore GET   $0,rJ        Сохранить адрес возврата.
075             AND   t,ss,lring_mask
076             LDOU  $1,1,t        $1 ← l[γ].
077             SET   arg,ss

```

078		PUSHJ	res,MemFind	
079		STOU	\$1,res,0	$M_8[rS] \leftarrow \$1$.
080		ADDU	ss,ss,8	Увеличить rS на 8.
081		PUT	rJ,\$0	Восстановить адрес возврата.
082		POP	0	Вернуться к вызывающей программе.
083	:StackLoad	GET	\$0,rJ	Сохранить адрес возврата.
084		SUBU	ss,ss,8	Уменьшить rS на 8.
085		SET	arg,ss	
086		PUSHJ	res,MemFind	
087		LDOU	\$1,res,0	$\$1 \leftarrow M_8[rS]$.
088		AND	t,ss,lring_mask	
089		STOU	\$1,l,t	$l[\gamma] \leftarrow \$1$.
090		PUT	rJ,\$0	Восстановить адрес возврата.
091		POP	0	Вернуться к вызывающей программе. █

(Регистр rJ в строках 074, 081, 083 и 090, конечно, является *реальным* rJ, а не симулированным регистром rJ. При симулировании самого компьютера нельзя забывать об этом!)

Подпрограмма StackRoom вызывается сразу после увеличения β . Она проверяет справедливость равенства $\beta = \gamma$ и, если оно справедливо, то увеличивает γ .

092	:StackRoom	SUBU	t,ss,oo	
093		SUBU	t,t,ll	
094		AND	t,t,lring_mask	
095		PBNZ	t,1F	Условный переход, если $(rS-rO)/8 \neq rL$ (по модулю ρ).
096		GET	\$0,rJ	Обана, это не концевая подпрограмма.
097		PUSHJ	res,StackStore	Продвинуть rS.
098		PUT	rJ,\$0	Восстановить обратный адрес.
099	1H	POP	0	Вернуться к вызывающей программе. █

Теперь перейдем к сердцевине симулятора, т.е. его основному циклу. Интерпретатор управления обычно содержит центральный раздел, который приводится в действие между интерпретируемыми инструкциями. В нашем случае, программа переходит к месту Fetch, когда она готова симулировать новую команду. Адрес @ новой симулируемой инструкции хранится в глобальном регистре inst_ptr. Fetch обычно задает $loc \leftarrow inst_ptr$ и продвигает inst_ptr на 4. Но если нужно симулировать команду RESUME, которая вставляет симулированный регистр rX в поток инструкций, то Fetch задает $loc \leftarrow inst_ptr - 4$ и оставляет inst_ptr без изменений. Этот симулятор считает инструкцию подлежащей выполнению, только если ее адрес loc находится в текстовом сегменте (т.е., если $loc < \#2000000000000000$).

100	* Основной цикл			
101	loc	GREG	0	Текущее местонахождение симулятора.
102	inst_ptr	GREG	0	Следующее местонахождение симулятора.
103	inst	GREG	0	Текущая симулируемая инструкция.
104	resuming	GREG	0	Продолжить симулирование инструкции в rX?
105	Fetch	PBZ	resuming,1F	Условный переход, если нет продолжения.
106		SUBU	loc,inst_ptr,4	$loc \leftarrow inst_ptr - 4$.
107		LDTU	inst,g,8*rX+4	$inst \leftarrow \text{right half of } rX$.

```

108      JMP    2F
109 1H      SET    loc,inst_ptr    loc ← inst_ptr.
110      SET    arg,loc
111      PUSHJ   res,MemFind
112      LDTU   inst,res,0        inst ← M4[loc].
113      ADDU   inst_ptr,loc,4    inst_ptr ← loc + 4.
114 2H      CMPU   t,loc,g
115      BNN    t,Error          Условный переход, если loc ≥ Data_Segment. █

```

Основной интерпретатор управления организует общий доступ к данным для всех инструкций. Он разбивает текущую инструкцию на разные части и помещает их в удобные регистры для последующего использования. Что еще более важно, он помещает 64 бита информации (“info”), соответствующей текущему опкоду, в глобальный регистр *f*. Основная таблица, которая начинается с адреса *Info*, содержит такую информацию для каждого из 256 опкодов компьютера MMIX. (См. табл. 1 на стр. 102.) Например, для *f* задается нечетное число, тогда и только тогда, когда поле *Z* текущего опкода является “немедленным” операндом или опкод является *JMP*. Аналогично, $f \wedge \#40$ будет ненулевым, тогда и только тогда, когда эта инструкция имеет относительный адрес. На более поздних этапах симулятор сможет быстро решить, что нужно сделать с текущей инструкцией, поскольку большая часть необходимой информации появится в регистре *f*.

```

116 op      GREG    0   Опкод текущей инструкции.
117 xx      GREG    0   Поле X текущей инструкции.
118 yy      GREG    0   Поле Y текущей инструкции.
119 zz      GREG    0   Поле Z текущей инструкции.
120 yz      GREG    0   Поле YZ текущей инструкции.
121 f       GREG    0   Упакованная информация о текущем опкоде.
122 xxx     GREG    0   Умножение поля X на 8.
123 x       GREG    0   Операнд X и/или результат.
124 y       GREG    0   Операнд Y.
125 z       GREG    0   Операнд Z.
126 xptr    GREG    0   Адрес, по которому нужно сохранить x.
127 exc     GREG    0   Арифметические исключительные ситуации.
128 Z_is_immed_bit IS #1   Флаговые биты, возможно в f.
129 Z_is_source_bit IS #2
130 Y_is_immed_bit IS #4
131 Y_is_source_bit IS #8
132 X_is_source_bit IS #10
133 X_is_dest_bit  IS #20
134 Rel_addr_bit   IS #40
135 Mem_bit        IS #80
136 Info IS      #1000
137 Done IS      Info+8*256
138 info GREG    Info      (Базовый адрес для основной информационной
                           таблицы.)
139 c255 GREG    8*255      (Удобная константа.)
140 c256 GREG    8*256      (Еще одна удобная константа.)
141      MOR     op,inst,#8  op ← inst ≫ 24.
142      MOR     xx,inst,#4  xx ← (inst ≫ 16) ∧ #ff.

```

```

143      MOR    yy,inst,#2  yy ← (inst >> 8) ∧ #ff.
144      MOR    zz,inst,#1  zz ← inst ∧ #ff.
145  OH GREG -#10000
146      ANDN   yz,inst,0B
147      SLU    xxx,xx,3
148      SLU    t,op,3
149      LDOU   f,info,t    f ← Info[op].
150      SET    x,0          x ← 0 (значение по умолчанию).
151      SET    y,0          y ← 0 (значение по умолчанию).
152      SET    z,0          z ← 0 (значение по умолчанию).
153      SET    exc,0        exc ← 0 (значение по умолчанию). █

```

После распаковки (т.е. разбиения) инструкции в разные поля нужно прежде всего преобразовать относительный адрес в абсолютный, если это необходимо.

```

154      AND    t,f,Rel_addr_bit
155      PBZ    t,1F          Условный переход, если это не относительный адрес.
156      PBEV   f,2F          Условный переход, если op не является JMP или JMPB.
157  9H GREG -#1000000
158      ANDN   yz,inst,9B    yz ← inst ∧ #ffffff (а именно XYZ).
159      ADDU   t,yz,9B        t ← XYZ - 224.
160      JMP    3F
161  2H ADDU   t,yz,0B        t ← YZ - 216.
162  3H CSOD   yz,op,t        Установить yz ← t, если op является нечетным
                             ("назад").
163      SL     t,yz,2
164      ADDU   yz,loc,t       yz ← loc + yz << 2. █

```

Следующая задача очень критична для большинства инструкций: нужно установить операнды, указанные в полях Y и Z, в глобальные регистры y и z. Иногда нужно также установить третий операнд в глобальный регистр x, указанный в поле X или в специальном регистре, например в симулированном регистре rD или rM.

```

165  1H      PBNB   resuming,Install_X  Условный переход, если только
                                         не resuming < 0.
                                         (См. упражнение 14.)
...
174  Install_X AND   t,f,X_is_source_bit
175          PBZ    t,1F          Условный переход, если только $X
                                         не источник.
176          SET    arg,xxx
177          PUSHJ  res,GetReg
178          SET    x,res          x ← $X.
179  1H      SRU    t,f,5
180          AND    t,t,#f8        t ← номер специального регистра,
                                         умноженный на 8.
181          PBZ    t,Install_Z
182          LDOU   x,g,t          Если t ≠ 0, то x ← g[t].
183  Install_Z AND   t,f,Z_is_source_bit
184          PBZ    t,1F          Условный переход, если только $Z
                                         не источник.
185          SLU    arg,zz,3
186          PUSHJ  res,GetReg

```

187		SET	z,res	$z \leftarrow \$Z$.
188		JMP	Install_Y	
189	1H	CSOD	z,f,zz	Если Z является немедленной константой, то $z \leftarrow Z$.
190		AND	t,op,#f0	
191		CMPU	t,t,#e0	
192		PBNZ	t,Install_Y	Условный переход, только если неверно $\#e0 \leq op < \#f0$.
193		AND	t,op,#3	
194		NEG	t,3,t	
195		SLU	t,t,4	
196		SLU	z,yz,t	$z \leftarrow yz \ll (48, 32, 16, \text{ or } 0)$.
197		SET	y,x	$y \leftarrow x$.
198	Install_Y	AND	t,f,Y_is_immed_bit	
199		PBZ	t,1F	Условный переход, если только Y не является немедленной константой.
200		SET	y,yy	$y \leftarrow Y$.
201		SLU	t,yy,40	
202		ADDU	f,f,t	Вставить Y в левую половину f.
203	1H	AND	t,f,Y_is_source_bit	
204		BZ	t,1F	Условный переход, если только \$Y не источник.
205		SLU	arg,yy,3	
206		PUSHJ	res,GetReg	
207		SET	y,res	$y \leftarrow \$Y$. ■

Если поле X указывает выходной регистр, то для xptr задается адрес, по которому будет в конечном итоге храниться симулированный результат. Этот адрес будет либо в массиве Global, либо в кольце Local. Симулированный стек регистров увеличивается в этот момент, если выходной регистр меняется от маргинального до локального.

208	1H	AND	t,f,X_is_dest_bit	
209		BZ	t,1F	Условный переход, если только \$X не является выходным регистром.
210	XDest	CMPU	t,xxx,gg	
211		BN	t,3F	Условный переход, если \$X не является глобальным регистром.
212		LDA	xptr,g,xxx	$xptr \leftarrow \text{адрес } g[X]$.
213		JMP	1F	
214	2H	ADDU	t,oo,ll	
215		AND	t,t,lring_mask	
216		STCO	0,1,t	$l[(\alpha + L) \bmod \rho] \leftarrow 0$.
217		INCL	ll,8	$L \leftarrow L + 1$. (\$L становится локальным.)
218		PUSHJ	res,StackRoom	Убедитесь, что $\beta \neq \gamma$.
219	3H	CMPU	t,xxx,ll	
220		BNN	t,2B	Условный переход, если \$X не является локальным регистром.
221		ADD	t,xxx,oo	
222		AND	t,t,lring_mask	
223		LDA	xptr,l,t	$xptr \leftarrow \text{адрес } l[(\alpha + X) \bmod \rho]$. ■

Итак, мы достигли кульминационного пункта основного цикла управления: симуляции текущей инструкции за счет 256-вариантного ветвления на основе текущего опкода. Левая половина регистра *f*, фактически, является инструкцией MMIX, которая *выполняется* в этот момент, за счет вставки ее в поток инструкций с помощью команды RESUME. Например, при симулировании команды ADD инструкция “ADD *x, y, z*” помещается в правую часть *rX* и очищает исключительные биты *rA*. Команда RESUME приводит к тому, что сумма регистров *y* и *z* помещается в регистр *x*, а *rA* зафиксировывает переполнение. После выполнения команды RESUME управление передается к месту Done, если только вставленная инструкция не была условным переходом или безусловным переходом.

224	1H	AND	<i>t, f, Mem_bit</i>	
225		PBZ	<i>t, 1F</i>	Условный переход, если только <i>inst</i> не осуществляет доступ к памяти.
226		ADDU	<i>arg, y, z</i>	
227		CMPU	<i>t, op, #A0</i>	<i>t</i> ← [оп является инструкцией загрузки].
228		BN	<i>t, 2F</i>	
229		CMPU	<i>t, arg, g</i>	
230		BN	<i>t, Error</i>	Ошибка, если сохранение в текстовый сегмент.
231	2H	PUSHJ	<i>res, MemFind</i>	<i>res</i> ← адрес <i>M[y + z]</i> .
232	1H	SRU	<i>t, f, 32</i>	
233		PUT	<i>rX, t</i>	<i>rX</i> ← левая половина <i>f</i> .
234		PUT	<i>rM, x</i>	<i>rM</i> ← <i>x</i> (подготовиться для MUX).
235		PUT	<i>rE, x</i>	<i>rE</i> ← <i>x</i> (подготовиться для FCMPE, FUNE, FEQLE).
236	0H	GREG	#30000	
237		AND	<i>t, aa, 0B</i>	<i>t</i> ← текущий режим округления.
238		ORL	<i>t, U_BIT << 8</i>	Разрешить обход антипереполнения (см. ниже).
239		PUT	<i>rA, t</i>	Подготовить <i>rA</i> для арифметических действий.
240	0H	GREG	Done	
241		PUT	<i>rW, 0B</i>	<i>rW</i> ← Done.
242		RESUME	0	Выполнить инструкцию в <i>rX</i> . █

Некоторые инструкции нельзя симулировать просто “выполняя их”, как команду ADD и переход к Done. Например, команда MULU должна вставить старшую половину вычисленного произведения в симулированный регистр *rH*. Команда условного перехода должна изменить *inst_ptr*, если имеет место условный переход. Команда PUSHJ должна протолкнуть симулированный стек регистров, а команда POP должна вытолкнуть его. Команды SAVE, UNSAVE, RESUME, TRAP и т.д. нужно симулировать особенно осторожно. Потому следующая часть симулятора посвящена обработке особых случаев, которые не укладываются в простую схему “*x* равно *y* операция *z*”.

Начнем с достаточно простых операций умножения и деления:

243	MulU	MULU	<i>x, y, z</i>	Умножить <i>y</i> на <i>z</i> , беззнаково.
244		GET	<i>t, rH</i>	Установить <i>t</i> ← старшая половина произведения.
245		STOU	<i>t, g, 8*rH</i>	<i>g[rH]</i> ← старшая половина произведения.
246		JMP	XDone	Завершить сохраняя <i>x</i> .
247	Div	DIV	<i>x, y, z</i>	
	...			(Для деления, см. упражнение 6.) █

Если симулированная команда является условным переходом, например “BZ \$X, RA”, то основной интерпретатор управления преобразует относительный ад-

рес RA в абсолютный адрес в регистре yz (строка 164). Он также поместит содержимое симулируемого регистра \$X в регистр x (строка 178). Команда RESUME затем выполнит инструкцию "BZ x,BTaken" (строка 242) и контроль будет передан BTaken вместо Done, если симулируется условный переход. BTaken прибавляет 2v к времени симулируемого выполнения, изменяет inst_ptr и переходит к Update.

254	BTaken	ADDU	cc,cc,4	Увеличивает rC на 4v.
255	PBTaken	SUBU	cc,cc,2	Увеличивает rC на 2v.
256		SET	inst_ptr,yz	inst_ptr \leftarrow условный переход.
257		JMP	Update	Завершить выполнение команды.
258	Go	SET	x,inst_ptr	Перейти к инструкции: задать $x \leftarrow loc + 4$.
259		ADDU	inst_ptr,y,z	inst_ptr $\leftarrow (y + z) \bmod 2^{64}$.
260		JMP	XDone	Завершить сохраняя x. █

(В строке 257 мог быть переход к Done, но это было бы медленнее. Переход к Update оправдан, поскольку команда условного перехода не сохраняет x и не может привести к возникновению арифметической исключительной ситуации. См. ниже строки 500–541.)

Команда PUSHJ или PUSHGO проталкивает симулированный стек регистров за счет увеличения указателя α из (2). Это означает увеличение симулированного регистра rO, а именно регистра oo. Если команда имеет вид "PUSHJ \$X,RA" и \$X является локальным регистром, то $X + 1$ октабайтов проталкиваются вниз за счет установки $\$X \leftarrow X$ и увеличения oo на $8(X + 1)$. (Значение в \$X будет использоваться позже командой POP для восстановления oo к исходному значению. Симулированный регистр \$X будет содержать результат подпрограммы, как объясняется в разделе 1.4.1'.) Если \$X является глобальным регистром, rL + 1 октабайт проталкиваются вниз аналогично.

261	PushGo	ADDU	yz,y,z	$yz \leftarrow (y + z) \bmod 2^{64}$.
262	PushJ	SET	inst_ptr,yz	inst_ptr $\leftarrow yz$.
263		CMPU	t,xxx,gg	
264		PBN	t,1F	Условный переход, если \$X является локальным регистром.
265		SET	xxx,ll	Предполагается, что $X = rL$.
266		SRU	xx,xxx,3	
267		INCL	ll,8	Увеличить rL на 1.
268		PUSHJ	0,StackRoom	Убедиться, что $\beta \neq \gamma$ в (2).
269	1H	ADDU	t,xxx,oo	
270		AND	t,t,lring_mask	
271		STOU	xx,l,t	$l[(\alpha + X) \bmod \rho] \leftarrow X$.
272		ADDU	t,loc,4	
273		STOU	t,g,8*rJ	$g[rJ] \leftarrow loc + 4$.
274		INCL	xxx,8	
275		SUBU	ll,ll,xxx	Уменьшить rL на $X + 1$.
276		ADDU	oo,oo,xxx	Увеличить rO на $8(X + 1)$.
277		JMP	Update	Завершить выполнение команды. █

Специальные процедуры также нужны для симулирования POP, SAVE, UNSAVE, а также нескольких других опкодов, включая RESUME. Эти процедуры связаны с интересными особенностями MMIX, которые рассматриваются в упражнениях. А сейчас

мы их пропустим, поскольку они не связаны методами использования интерпретаторов, которые не рассматривались выше.

Здесь можно было бы рассмотреть код для SYNC и TRIP, но эти процедуры чересчур просты. (Действительно, для "SYNC XYZ" не нужно делать ничего, за исключением проверки $XYZ \leq 3$, поскольку работа кэш-памяти не симулируется.) Вместо этого рассмотрим код для TRAP, который интересен тем, что иллюстрирует важную технологию использования таблицы переходов для многоканального (многопроходного) переключения:

278	Sync	BNZ	xx,Error	Условный переход, если $X \neq 0$.
279		CMPU	t,yz,4	
280		BNN	t,Error	Условный переход, если $YZ \geq 4$.
281		JMP	Update	Завершить выполнение команды.
282	Trip	SET	xx,0	Инициировать обход к адресу 0.
283		JMP	TakeTrip	(См. упражнение 13.)
284	Trap	STOU	inst_ptr,g,8*rWW	$g[rWW] \leftarrow inst_ptr$.
285	OH GREG	#8000000000000000		
286		ADDU	t,inst,0B	
287		STOU	t,g,8*rXX	$g[rXX] \leftarrow inst + 2^{63}$.
288		STOU	y,g,8*rYY	$g[rYY] \leftarrow y$.
289		STOU	z,g,8*rZZ	$g[rZZ] \leftarrow z$.
290		SRU	y,inst,6	
291		CMPU	t,y,4*11	
292		BNN	t,Error	Условный переход, если $X \neq 0$ или $Y > Ftell$.
293		LDOU	t,g,c255	$t \leftarrow g[255]$.
294	OH GREG	@+4		
295		GO	y,0B,y	Переход к $@ + 4 + 4Y$.
296		JMP	SimHalt	$Y = Halt$: Переход к SimHalt.
297		JMP	SimFopen	$Y = Fopen$: Переход к SimFopen.
298		JMP	SimFclose	$Y = Fclose$: Переход к SimFclose.
299		JMP	SimFread	$Y = Fread$: Переход к SimFread.
300		JMP	SimFgets	$Y = Fgets$: Переход к SimFgets.
301		JMP	SimFgetws	$Y = Fgetws$: Переход к SimFgetws.
302		JMP	SimFwrite	$Y = Fwrite$: Переход к SimFwrite.
303		JMP	SimFputs	$Y = Fputs$: Переход к SimFputs.
304		JMP	SimFputws	$Y = Fputws$: Переход к SimFputws.
305		JMP	SimFseek	$Y = Fseek$: Переход к SimFseek.
306		JMP	SimFtell	$Y = Ftell$: Переход к SimFtell.
307	TrapDone	STO	t,g,8*rBB	Установить $g[rBB] \leftarrow t$.
308		STO	t,g,c255	Обход заканчивается с $g[255] \leftarrow g[rBB]$.
309		JMP	Update	Завершить выполнение команды. █

(См. упражнения 15–17 для процедур SimFopen, SimFclose, SimFread, и т.д.)

Рассмотрим теперь основную таблицу Info (см. таблицу 1), которая позволяет симулятору эффективно работать с 256 разными опкодами. Каждый элемент таблицы является октабайтом, состоящим из (i) четырехбайтовой инструкции MMIX, которая вызывается инструкцией RESUME в строке 242; (ii) двух байтов, которые определяют время симулированного выполнения, один байт для μ и один байт для ν ;

Таблица 1

ОСНОВНАЯ ИНФОРМАЦИОННАЯ ТАБЛИЦА ДЛЯ КОНТРОЛЯ ЗА СИМУЛЯТОРОМ

0 IS Done-4		LDB x,res,0; BYTE 1,1,0,#aa	(LDB)
LOC Info		LDB x,res,0; BYTE 1,1,0,#a9	(LDBI)
JMP Trap+@-0; BYTE 0,5,0,#0a	(TRAP)	...	
FCMP x,y,z; BYTE 0,1,0,#2a	(FCMP)	JMP Cswap+@-0; BYTE 2,2,0,#ba	(CSWAP)
FUN x,y,z; BYTE 0,1,0,#2a	(FUN)	JMP Cswap+@-0; BYTE 2,2,0,#b9	(CSWAPI)
FEQL x,y,z; BYTE 0,1,0,#2a	(FEQL)	LDUNC x,res,0; BYTE 1,1,0,#aa	(LDUNC)
FADD x,y,z; BYTE 0,4,0,#2a	(FADD)	LDUNC x,res,0; BYTE 1,1,0,#a9	(LDUNCI)
FIX x,0,z; BYTE 0,4,0,#26	(FIX)	JMP Error+@-0; BYTE 0,1,0,#2a	(LDVTS)
FSUB x,y,z; BYTE 0,4,0,#2a	(FSUB)	JMP Error+@-0; BYTE 0,1,0,#29	(LDVTSI)
FIXU x,0,z; BYTE 0,4,0,#26	(FIXU)	SWYM 0; BYTE 0,1,0,#0a	(PRELD)
FLOT x,0,z; BYTE 0,4,0,#26	(FLOT)	SWYM 0; BYTE 0,1,0,#09	(PRELDI)
FLOTU x,0,z; BYTE 0,4,0,#25	(FLOTI)	SWYM 0; BYTE 0,1,0,#0a	(PREGO)
FLOTU x,0,z; BYTE 0,4,0,#26	(FLOTU)	SWYM 0; BYTE 0,1,0,#09	(PREGOI)
...		JMP Go+@-0; BYTE 0,3,0,#2a	(GO)
FMUL x,y,z; BYTE 0,4,0,#2a	(FMUL)	JMP Go+@-0; BYTE 0,3,0,#29	(GOI)
FCMPE x,y,z; BYTE 0,4,rE,#2a	(FCMPE)	STB x,res,0; BYTE 1,1,0,#9a	(STB)
FUNE x,y,z; BYTE 0,1,rE,#2a	(FUNE)	STB x,res,0; BYTE 1,1,0,#99	(STBI)
FEQLE x,y,z; BYTE 0,4,rE,#2a	(FEQLE)	...	
FDIV x,y,z; BYTE 0,40,0,#2a	(FDIV)	STO xx,res,0; BYTE 1,1,0,#8a	(STCO)
FSQRT x,0,z; BYTE 0,40,0,#26	(FSQRT)	STO xx,res,0; BYTE 1,1,0,#89	(STCOI)
FREM x,y,z; BYTE 0,4,0,#2a	(FREM)	STUNC x,res,0; BYTE 1,1,0,#9a	(STUNC)
FINT x,0,z; BYTE 0,4,0,#26	(FINT)	STUNC x,res,0; BYTE 1,1,0,#99	(STUNCI)
MUL x,y,z; BYTE 0,10,0,#2a	(MUL)	SWYM 0; BYTE 0,1,0,#0a	(SYNCD)
MUL x,y,z; BYTE 0,10,0,#29	(MULI)	SWYM 0; BYTE 0,1,0,#09	(SYNCIDI)
JMP MulU+@-0; BYTE 0,10,0,#2a	(MULU)	SWYM 0; BYTE 0,1,0,#0a	(PREST)
JMP MulU+@-0; BYTE 0,10,0,#29	(MULUI)	SWYM 0; BYTE 0,1,0,#09	(PRESTI)
JMP Div+@-0; BYTE 0,60,0,#2a	(DIV)	SWYM 0; BYTE 0,1,0,#0a	(SYNCID)
JMP Div+@-0; BYTE 0,60,0,#29	(DIVI)	SWYM 0; BYTE 0,1,0,#09	(SYNCIDI)
JMP DivU+@-0; BYTE 0,60,rD,#2a	(DIVU)	JMP PushGo+@-0; BYTE 0,3,0,#2a	(PUSHGO)
JMP DivU+@-0; BYTE 0,60,rD,#29	(DIVUI)	JMP PushGo+@-0; BYTE 0,3,0,#29	(PUSHGOI)
ADD x,y,z; BYTE 0,1,0,#2a	(ADD)	OR x,y,z; BYTE 0,1,0,#2a	(OR)
ADD x,y,z; BYTE 0,1,0,#29	(ADDI)	OR x,y,z; BYTE 0,1,0,#29	(ORI)
ADDU x,y,z; BYTE 0,1,0,#2a	(ADDU)	...	
...		SET x,z; BYTE 0,1,0,#20	(SETH)
CMPU x,y,z; BYTE 0,1,0,#29	(CMPUI)	SET x,z; BYTE 0,1,0,#20	(SETMH)
NEG x,0,z; BYTE 0,1,0,#26	(NEG)	...	
NEG x,0,z; BYTE 0,1,0,#25	(NEGI)	ANDN x,x,z; BYTE 0,1,0,#30	(ANDNL)
NEGU x,0,z; BYTE 0,1,0,#26	(NEGU)	SET inst_ptr,yz; BYTE 0,1,0,#41	(JMP)
NEGU x,0,z; BYTE 0,1,0,#25	(NEGUI)	SET inst_ptr,yz; BYTE 0,1,0,#41	(JMPB)
SL x,y,z; BYTE 0,1,0,#2a	(SL)	JMP PushJ+@-0; BYTE 0,1,0,#60	(PUSHJ)
...		JMP PushJ+@-0; BYTE 0,1,0,#60	(PUSHJB)
BN x,BTaken+@-0; BYTE 0,1,0,#50	(BN)	SET x,yz; BYTE 0,1,0,#60	(GETA)
BN x,BTaken+@-0; BYTE 0,1,0,#50	(BNB)	SET x,yz; BYTE 0,1,0,#60	(GETAB)
BZ x,BTaken+@-0; BYTE 0,1,0,#50	(BZ)	JMP Put+@-0; BYTE 0,1,0,#02	(PUT)
...		JMP Put+@-0; BYTE 0,1,0,#01	(PUTI)
PBNP x,PBTaken+@-0; BYTE 0,3,0,#50	(PBNPB)	JMP Pop+@-0; BYTE 0,3,rJ,#00	(POP)
PBEV x,PBTaken+@-0; BYTE 0,3,0,#50	(PBEV)	JMP Resume+@-0; BYTE 0,5,0,#00	(RESUME)
PBEV x,PBTaken+@-0; BYTE 0,3,0,#50	(PBEVB)	JMP Save+@-0; BYTE 20,1,0,#20	(SAVE)
CSN x,y,z; BYTE 0,1,0,#3a	(CSN)	JMP Unsave+@-0; BYTE 20,1,0,#02	(UNSAVE)
CSN x,y,z; BYTE 0,1,0,#39	(CSNI)	JMP Sync+@-0; BYTE 0,1,0,#01	(SYNC)
...		SWYM x,y,z; BYTE 0,1,0,#00	(SWYM)
ZSEV x,y,z; BYTE 0,1,0,#2a	(ZSEV)	JMP Get+@-0; BYTE 0,1,0,#20	(GET)
ZSEV x,y,z; BYTE 0,1,0,#29	(ZSEVI)	JMP Trip+@-0; BYTE 0,5,0,#0a	(TRIP)

Не показанные здесь элементы имеют аналогичную структуру, которую легко воспроизвести на основе представленных упражнений. (См., например, упражнение 1.)

(iii) байта-имени специального регистра, если такой регистр должен быть загружен в x в строке 182; а также (iv) байта, который является суммой восьми битовых флагов для специальных свойств опкода. Например, информация для опкода **FIX** имеет вид

FIX $x, 0, z$; **BYTE** 0, 4, 0, #26;

означающая, что (i) должна быть выполнена инструкция **FIX** $x, 0, z$, т.е. округление числа с плавающей точкой до числа с фиксированной точкой; (ii) время симулированного выполнения должно быть увеличено на $0\mu + 4v$; (iii) не нужен специальный регистр для входного операнда; (iv) флаговый байт имеет приведенный ниже вид, определяющий обработку регистров x , y и z :

#26 = $X_is_dest_bit + Y_is_immed_bit + Z_is_source_bit$

($Y_is_immed_bit$ означает вставку поля Y симулированной инструкции в поле Y инструкции "**FIX** $x, 0, z$ "; см. строку 202.)

* Интересная особенность таблицы **Info** состоит в том, что команда **RESUME** в строке 242 выполняет инструкцию так, как если бы она была по адресу **Done-4**, поскольку $rW = Done$. Следовательно, если данной инструкцией является **JMP**, то адрес должен быть относительным к **Done-4**, но **MMIXAL** всегда ассемблирует команды **JMP** с адресом относительно ассемблированного положения **@**. Чтобы обойти ассемблер используется следующая уловка "**JMP** **Trap+@-0**", где **0** определяется равным **Done-4**. Тогда команда **RESUME** совершает переход к **Trap** в случае необходимости.

После выполнения специальной инструкции, вставленной командой **RESUME**, обычно происходит переход к **Done**. С этого места все остальное проще: инструкция успешно симулирована и текущий цикл практически завершен. Осталось оформить несколько деталей: нужно сохранить результат x в соответствующем месте, если имеется флаг $X_is_dest_bit$ и проверить, не является арифметическая исключительная ситуация причиной прерывания обхода:

500	Done	AND	$t, f, X_is_dest_bit$	
501		BZ	$t, 1F$	Условный переход, если только $\$X$ не является местом назначения.
502	XDone	STOU	$x, xptr, 0$	Сохранить x в симулированном регистре $\$X$.
503	1H	GET	t, rA	
504		AND	$t, t, \#ff$	$t \leftarrow$ новая арифметическая исключительная ситуация.
505		OR	exc, exc, t	$exc \leftarrow exc \vee t$.
506		AND	$t, exc, U_BIT + X_BIT$	
507		CMPU	t, t, U_BIT	
508		PBNZ	$t, 1F$	Условный переход, если только нет антипереполнения.
509	OH GREG		$U_BIT < 8$	
510		AND	$t, aa, 0B$	
511		BNZ	$t, 1F$	Условный переход, если антипереполнение.
512		ANDNL	exc, U_BIT	Игнорировать антипереполнение, если все точно и ситуация не активирована.
513	1H	PBZ	$exc, Update$	
514		SRU	$t, aa, 8$	
515		AND	t, t, exc	

516		PBZ	t,4F	Условный переход, если не требуется прерывание обхода.
	...			(См. упражнение 13.)
539	4H	OR	aa,aa,exc	Записать новые исключительные ситуации в гА. █

Строка 500 используется для удобства, хотя несколько сотен инструкций и вся таблица Info фактически присутствуют между строкой 309 и этой части программы. Метка Done в строке 500 не конфликтует с меткой Done в строке 137, поскольку обе они определяют эквивалентное значение для этого символа.

После строки 505 регистр exc содержит битовые коды для всех арифметических исключительных ситуаций, активированных только что симулированной инструкцией. В этом случае приходится иметь дело с забавной асимметрией в правилах выполнения арифметических действий с плавающей точкой согласно стандарта IEEE. Исключительная ситуация антипереполнения (U) подавляется, только если не активируется обход антипереполнения в гА или не возникла исключительная ситуация неточности (X). (Именно по этой причине активируется обход антипереполнения в строке 238. Симулятор завершает работу командами

LOC U_Handler; ORL exc,U_BIT; JMP Done (5)

так, что exc записывает исключительные ситуации антипереполнения в тех случаях, когда вычисления с плавающей точкой точны, но дают субнормальный результат.)

Наконец — Уррра! — можно завершить цикл операций, начатых с метки Fetch. Здесь обновляются показания часов и счетчиков, а потом происходит возврат к метке Fetch:

540	OH GREG	#0000000800000004	
541	Update	MOR	t,f,0B $2^{32} \text{ mems} + \text{oops}$
542		ADDU	cc,cc,t Увеличить показания часов, гС.
543		ADDU	uu,uu,1 Увеличить значение счетчика использования, гU.
544		SUBU	ii,ii,1 Уменьшить значение счетчика интервала, гI.
545	AllDone	PBZ	resuming,Fetch Перейти к Fetch, если resuming = 0.
546		CMPU	t,op,#F9 В противном случае, установить $t \leftarrow [op = \text{RESUME}]$.
547		CSNZ	resuming,t,0 Очистить resuming, если нет продолжения,
548		JMP	Fetch и перейти к Fetch. █

Программа симулятора завершена, за исключением того, что нужно организовать правильную инициализацию. Предполагается, что симулятор запускается с помощью командной строки, в которой указан двоичный файл. В упражнении 20 описывается простой формат такого файла, в котором указывается что именно должно быть загружено в симулированную память до начала симуляции. Сразу после загрузки программы происходит следующее: в показанной ниже строке 576 регистр loc будет содержать адрес, в котором команда UNSAVE даст программе старт. (На самом деле, команда UNSAVE симулируется симулируемой командой RESUME. Для этого используется хитроумный, но работоспособный, код.)

549	Infile	IS	3	(Обработка двоичного входного файла.)
550	Main	LDA	Mem:head,Chunk0	Инициализировать MemFind.
551		ADDU	Mem:alloc,Mem:head,Mem:nodesize	

552	GET	t,rN	
553	INCL	t,1	
554	STOU	t,g,8*rN	$g[rN] \leftarrow (\text{наш } rN) + 1.$
555	LDOU	t,\$1,8	$t \leftarrow \text{имя двоичного файла } (argv[1]).$
556	STOU	t,IOArgs	
557	LDA	t,IOArgs	(См. строку 010)
558	TRAP	0,Fopen,Infile	Открыть двоичный файл.
559	BN	t,Error	
			Теперь загрузить двоичный файл (см. упражнение 20).
576	STOU	loc,g,c255	$g[255] \leftarrow \text{поместить в UNSAVE.}$
577	SUBU	arg,loc,8*13	$arg \leftarrow \text{поместить туда, где появляется } \$255.$
578	PUSHJ	res,MemFind	
579	LDOU	inst_ptr,res,0	$inst_ptr \leftarrow \text{Main.}$
580	SET	arg,#90	
581	PUSHJ	res,MemFind	
582	LDTU	x,res,0	$x \leftarrow M_4[\#90].$
583	SET	resuming,1	$resuming \leftarrow 1.$
584	CSNZ	inst_ptr,x,#90	Если $x \neq 0$, установить $inst_ptr \leftarrow \#90.$
585	OH	GREG	$\#FB < 24 + 255$
586	STOU	OB,g,8*rX	$g[rX] \leftarrow \text{"UNSAVE } \$255".$
587	SET	gg,c255	$G \leftarrow 255.$
588	JMP	Fetch	Запустить симулятор.
589	Error	NEG	t,22
			$t \leftarrow -22$ для выхода в случае ошибки.
590	Exit	TRAP	0,Halt,0
			Завершить симуляцию.
591	LOC	Global+8*rK; OCTA	-1
592	LOC	Global+8*rT; OCTA	#80000000500000000
593	LOC	Global+8*rTT; OCTA	#80000000600000000
594	LOC	Global+8*rV; OCTA	#369c200400000000

Стартовый адрес Main симулируемой программы находится в симулированном регистре \$255 после выполнения симулированной команды UNSAVE. В строках 580–584 реализуется компонент, который не упоминался в разделе 1.3.2': если инструкция загружается по адресу #90, то программа запускается с того места, а не с метки Main. (Это позволяет инициализировать библиотечную подпрограмму до запуска пользовательской программы с метки Main.)

В строках 591–594 симулированные регистры rK, rT, rTT и rV инициализируются соответствующими постоянными значениями. Затем программа завершается инструкциями обработчика обхода (5).

Уф! Программа симулятора получилась довольно большой — фактически, больше любой другой программы в этой книге. Но несмотря на большой размер, она не является полной в некоторых аспектах, поскольку автору не хотелось увеличивать ее размер.

- Несколько частей кода вынесено в упражнения.
- Эта программа совершает условный переход к Error и выходит при обнаружении любой проблемы. Качественный симулятор должен различать разные типы ошибок и находить соответствующий способ их нейтрализации и восстановления работоспособности программы.

- с) Эта программа не собирает статистические данные, за исключением общего времени выполнения (сс) и общего количества симулированных инструкций (uu). Более полная программа могла бы, например, запоминать насколько часто совершались переходы по сравнению с вероятностями переходов, записывать количество обращений подпрограмм **StackLoad** и **StackStore** к симулированной памяти, анализировать собственные алгоритмы, например изучая эффективность самоорганизующейся технологии поиска на основе **MemFind**.
- д) Эта программа не имеет никаких диагностических процедур. Например качественный симулятор, должен иметь средства интерактивной отладки и выводить избранные данные о выполнении симулированной программы. Такие средства не так уж и трудно создать, а способность отслеживать работу программы является одним из основных достоинств интерпретаторов.

УПРАЖНЕНИЯ:

1. [20] В табл. 1 приведены элементы только для избранных опкодов. Как будут выглядеть аналогичные элементы для опкодов (а) **#3F (SRUI)**? (б) **#55 (PBPB)**? (с) **#D9 (MUXI)**? (д) **#E6 (INCML)**?
- ▶ 2. [26] Сколько времени потребуется симулятору для симулирования инструкций (а) **ADDU \$255,\$Y,\$Z**; (б) **STHT \$X,\$Y,0**; (с) **PBNZ \$X,0-4**?
3. [23] Объясните, почему $\gamma \neq \alpha$, когда **StackRoom** вызывает **StackStore** в строке 097.
- ▶ 4. [20] Оцените критически то, что **MemFind** никогда не проверяет, имеет ли **alloc** очень большое значение. Насколько серьезно это упущение?
- ▶ 5. [20] Если подпрограмма **MemFind** совершает условный переход к **Error**, то стек регистров не выталкивается. Сколько элементов может быть в стеке регистров в этот момент?
6. [20] Создайте код для симулирования инструкций **DIV** и **DIVU**, пропущенный в строках 248–253.
7. [21] Создайте код для симулирования инструкции **CSWAP**.
8. [22] Создайте код для симулирования инструкции **GET**.
9. [23] Создайте код для симулирования инструкции **PUT**.
10. [24] Создайте код для симулирования инструкции **POP**. *Замечание:* если нормальное выполнение **POP**, как описано в разделе 1.4.1', приводит к $rL > rG$, то **MMIX** вытолкнет элементы из верхней части стека регистров так, что $rL = rG$. Например, если пользователь проталкивает 250 регистров вниз с помощью **PUSHJ** и выполняет "**PUT rG,32; POP**", то останутся только 32 из протолкнутых вниз регистров.
11. [25] Создайте код для симулирования инструкции **SAVE**. *Замечание:* **SAVE** проталкивает локальные регистры вниз и сохраняет в памяти весь стек регистров, за которым следуют $\$G, \$(G+1), \dots, \$255$, потом $rB, rD, rE, rH, rJ, rM, rR, rP, rW, rX, rY$ и rZ (именно в таком порядке), а потом октабайт $2^{56}rG + rA$.
12. [26] Создайте код для симулирования инструкции **UNSAVE**. *Замечание:* самая первая симулируемая инструкция **UNSAVE** является частью исходного процесса загрузки (см. строки 583–588), поэтому она не должна обновлять показания часов.
13. [27] Создайте код для симулирования прерывания обхода, пропущенный в строках 517–538.
14. [28] Создайте код для симулирования инструкции **RESUME**. *Замечание:* если rX неотрицательно, то наиболее значимый байт называется "ропкод" ("opcode"). Ропкоды 0,

1, 2 доступны для пользовательских программ. В строке 242 симулятора используется ропкод 0, который просто вставляет младшую половину rX в поток инструкций. Ропкод 1 действует аналогично, но инструкция в rX выполняется с $y \leftarrow rY$ и $z \leftarrow rZ$ вместо обычных операндов. Этот вариант допускается только, если первой шестнадцатеричной цифрой вставленного опкода является #0, #1, #2, #3, #6, #7, #C, #D или #E. Ропкод 2 задает $\$X \leftarrow rZ$ и $exc \leftarrow Q$, где X является третьим байтом справа в rX и Q является третьим байтом слева. Это позволяет задать значение регистра и одновременно охватить любое подмножество арифметических исключительных ситуаций DVWIOUZX. Ропкоды 1 и 2 можно использовать только, если $\$X$ не является маргинальным. В решении упражнения следует использовать RESUME для установки $resuming \leftarrow 0$, если симулированный регистр rX отрицателен, либо для установки $resuming \leftarrow (1, -1, -2)$ для ропкодов (0, 1, 2). Следует также создать отсутствующий код для строк 166–173.

- 15. [25] Создайте код процедуры SimFputs, которая симулирует вывод строки в файл с соответствующим идентификатором.
- 16. [25] Создайте код процедуры SimFopen, которая открывает файл с соответствующим идентификатором. (Симулятор может использовать тот же идентификатор, что и пользовательская программа.)
- 17. [25] В продолжение предыдущих упражнений создайте код процедуры SimFread, которая считывает заданное количество байтов из файла с соответствующим идентификатором.
- 18. [21] Будет ли полезен этот симулятор, если размер кольца локальных регистров lring_size был бы меньше 256, например lring_size = 32?
- 19. [14] Изучите все способы применения процедуры StackRoom (а именно, в строке 218, строке 268 и ответе к упражнению 11). Можете ли вы предложить более удачный способ организации кода? (См. п. 3 обсуждения в конце раздела 1.4.1'.)
- 20. [20] Входные двоичные файлы симулятора состоят из одной или более групп октабайтов, каждый из которых имеет простой вид

$$\lambda, x_0, x_1, \dots, x_{l-1}, 0$$

для некоторого $l \geq 0$, где x_0, x_1, \dots, x_{l-1} ненулевые; что означает

$$M_8[\lambda + 8k] \leftarrow x_k, \quad \text{for } 0 \leq k < l.$$

Файл заканчивается после последней группы. Завершите работу над симулятором, создав код MMIX для загрузки таких входных данных (строки 560–575 программы). Последним значением регистра loc должен быть адрес последнего загруженного октабайта, а именно $\lambda + 8(l - 1)$.

- 21. [20] Способна ли программа-симулятор из этого раздела симулировать саму себя? Если да, то способна ли она симулировать процесс собственной симуляции? Если да, то способна ли она симулировать ...?
- 22. [40] Создайте эффективную процедуру *трассировочного перехода* для MMIX. Эта программа должна записывать все передачи управления в процессе выполнения другой заданной программы в виде последовательности пар $(x_1, y_1), (x_2, y_2), \dots$, которые означают переход программы из x_1 в y_1 , а затем (после выполнения инструкций в местах $y_1, y_1 + 1, \dots, x_2$) переход из x_2 в y_2 , и т.д. [На основе этой информации данный интерпретатор способен реконструировать поток выполнения заданной программы и определить частоту выполнения каждой инструкции.]

Интерпретатор трассировки отличается от симулятора тем, что позволяет трассируемой программе занимать ее обычные места в памяти. Трассировочный переход изменяет

поток инструкций в памяти, но делает это только до той степени, которая позволяет сохранять управление. В противном случае компьютер может с максимальной скоростью выполнять арифметические инструкции и операции с памятью. Однако нужно учитывать некоторые ограничения: например трассируемая программа не должна сама себя изменять. Нужно стараться сводить к минимуму такие ограничения.

ОТВЕТЫ К УПРАЖНЕНИЯМ

РАЗДЕЛ 1.3.1'

1. #7d9 или #7D9.

2. (a) {B, D, F, b, d, f}. (b) {A, C, E, a, c, e}. Довольно необычный факт.

3. (Решение предложено Грегором Н. Перди (Gregor N. Purdy).) 2 бита = 1 нип (nup); 2 нипа = 1 пуббл; 2 нибла = 1 байт. Термин “байт” был случайно предложен в 1956 году членами проекта Stretch computer компании IBM; см. W. Buchholz, BYTE 2,2 (February 1977), 144.

4. 1000 Мб = 1 гигабайт — Гб, 1000 Гб = 1 терабайт — Тб, 1000 Тб = 1 петабайт — Пб, 1000 Пб = 1 эксабайт — Эб, 1000 Эб = 1 зекстибайт — Зб, 1000 Зб = 1 септибайт — Сб, согласно соглашениям 19-й Общей конференции мер и весов (19th Conférence Générale des Poids et Mesures) (1990).

(Однако некоторые считают, что вместо 1000 следует использовать 2^{10} в этих формулах, заявляя, что килобайт равен 1024 байтам. Для исключения двусмысленности такую единицу измерения следует называть *большим килобайтом*, *большим мегабайтом* и т.д., а обозначаются ККБ (ККВ), ММБ (ММВ), ... для обозначения их двоичной природы.)

Число 1024 можно считать “компьютерной тысячей” — как число 13 считается (или считалось) “чертовой дюжиной”.

— Т. Х. О’БЭЙЕРН (Т. Н. О’BEIRNE) (1962)

5. Если $-2^{n-1} \leq x < 2^{n-1}$, то $-2^n < x - s(\alpha) < 2^n$; следовательно, $x \neq s(\alpha)$ предполагает, что $x \not\equiv s(\alpha)$ (по модулю 2^n). Но $s(\alpha) = u(\alpha) - 2^n[\alpha \text{ начинается с } 1] \equiv u(\alpha)$ (по модулю 2^n).

6. Используя обозначения из предыдущего упражнения, получим $u(\bar{\alpha}) = 2^n - 1 - u(\alpha)$; следовательно, $u(\bar{\alpha}) + 1 \equiv -u(\alpha)$ (по модулю 2^n), и отсюда следует, что $s(\bar{\alpha}) + 1 = -s(\alpha)$. Однако при добавлении 1 может произойти переполнение. В таком случае $\alpha = 10 \dots 0$, $s(\alpha) = -2^{n-1}$, а $-s(\alpha)$ непредставимо.

7. Да. (См. раздел об операторах сдвига.)

8. Точка в двоичном представлении числа находится между гН и \$X. (Вообще, если точка в двоичном представлении числа находится на m позиций от конца \$Y и на n позиций от конца \$Z, то в произведении она будет на $m + n$ позиций от конца.)

9. Да, за исключением случая, когда $X = Y$, либо $X = Z$, либо в случае переполнения.

10. \$Y = #8000 0000 0000 0000, \$Z = #ffff ffff ffff ffff — это единственный пример!

11. (a) Верно, поскольку $s(\$Y) \equiv u(\$Y)$ и $s(\$Z) \equiv u(\$Z)$ (по модулю 2^{64}) согласно упражнению 5. (b) Верно, если $s(\$Y) \geq 0$ и $s(\$Z) \geq 0$, поскольку в этом случае $s(\$Y) = u(\$Y)$ и $s(\$Z) = u(\$Z)$. Также верно, если $\$Z = 0$, либо $\$Z = 1$, либо $\$Z = \Y , либо $\$Y = 0$. В противном случае неверно.

12. Если $X \neq Y$, то ‘ADDU \$X,\$Y,\$Z; CMPU carry,\$X,\$Y; ZSN carry,carry,1’. А если $X = Y = Z$, то ‘ZSN carry,\$X,1; ADDU \$X,\$X,\$X’.

13. Переполнение возникает при знаковом сложении тогда и только тогда, когда \$Y и \$Z имеют одинаковый знак, но их беззнаковая сумма имеет противоположный знак. Тогда

XOR \$0,\$Y,\$Z; ADDU \$X,\$Y,\$Z; XOR \$1,\$X,\$Y; ANDN \$1,\$1,\$0; ZSN ovfl,\$1,1

определяют наличие или отсутствие переполнения в случае $X \neq Y$.

14. Поменяйте X и Y в ответе из предыдущего упражнения. (Переполнение возникает при вычислении $x = y - z$ тогда и только тогда, если оно происходит при вычислении $y = x + z$.)

15. Пусть \dot{y} и \dot{z} являются знаковыми битами y и z , так что $s(y) = y - 2^{64}\dot{y}$ и $s(z) = z - 2^{64}\dot{z}$. Нам нужно вычислить $s(y)s(z) \bmod 2^{128} = (yz - 2^{64}(\dot{y}z + y\dot{z})) \bmod 2^{128}$. Таким образом, программа MULU \$X,\$Y,\$Z; GET \$0,rh; ZSN \$1,\$Y,\$Z; SUBU \$0,\$0,\$1; ZSN \$1,\$Z,\$Y; SUBU \$0,\$0,\$1 помещает искомым октабайт в \$0.

16. После выполнения инструкций в ответе из предыдущего упражнения нужно проверить факт, что старшая половина является расширением знака младшей половины с помощью инструкций 'SR \$1,\$X,63; CMP \$1,\$0,\$1; ZSNZ ovfl,\$1,1'.

17. Пусть a является константой, которая равна $(2^{65} + 1)/3$. Тогда $ay/2^{65} = y/3 + y/(3 \cdot 2^{65})$, где $\lfloor ay/2^{65} \rfloor = \lfloor y/3 \rfloor$ для $0 \leq y < 2^{65}$.

18. Используя аналогичные аргументы, получим $\lfloor ay/2^{66} \rfloor = \lfloor y/5 \rfloor$ для $0 \leq y < 2^{66}$, где $a = (2^{66} + 1)/5 = \#cccccccccccccccc$.

19. Это утверждение широко распространено и было воплощено создателями компиляторов, которые не проверяли математику. Но оно ложно при $z = 7, 21, 23, 25, 29, 31, 39, 47, 49, 53, 55, 61, 63, 71, 81, 89, \dots$, и для 189 нечетные делители z меньше, чем 1000!

Пусть $\epsilon = ay/2^{64+e} - y/z = (z - r)y/(2^{64+e}z)$, где $r = 2^{64+e} \bmod z$. Тогда $0 \leq \epsilon < 2/z$, следовательно опасения могут возникнуть только, если $y \equiv -1 \pmod{z}$ и $\epsilon \geq 1/z$. Отсюда следует, что формула $\lfloor ay/2^{64+e} \rfloor = \lfloor y/z \rfloor$ верна для всех беззнаковых октабайтов y , $0 \leq y < 2^{64}$, тогда и только тогда, когда оно выполняется для одного значения $y = 2^{64} - 1 - (2^{64} \bmod z)$.

(Однако это формула всегда выполняется в ограниченном диапазоне $0 \leq y < 2^{63}$. Майкл Йодер (Michael Yoder) заметил, что "умножение старшей половины" $\lfloor 2^{64+e+1}/z \rfloor - 2^{64}$, за которым следует сложение y и сдвиг вправо на $e + 1$, работает в общем случае.)

20. 4ADDU \$X,\$Y,\$Y; 4ADDU \$X,\$X,\$X.

21. SL устанавливает \$X в нуль, приводя к переполнению, если \$Y содержал ненулевое значение. SLU и SRU устанавливают \$X в нуль. SR устанавливает \$X в 64 копии знакового бита \$Y, а именно в $-\lfloor \$Y < 0 \rfloor$. (Учтите, что сдвиг влево на -1 не приводит к сдвигу вправо.)

22. Программа г-на Далла выполняет неверное ветвление, если инструкция SUB вызывает переполнение. Например, она считает, что *каждое* неотрицательное число меньше, чем -2^{63} . Кроме того, она считает, что $2^{63} - 1$ такое же малое число, как и любое отрицательное число. Хотя никакой ошибки не возникает, если \$1 и \$2 имеют одинаковый знак или значения в \$1 и \$2 меньше 2^{62} по абсолютной величине, но гораздо лучше использовать правильную формулировку 'CMP \$0,\$1,\$2; BN \$0,Case1'. (Аналогичные ошибки совершаются программистами и создателями компиляторов еще с 1950-х годов, которые часто приводили к серьезным и загадочным сбоям.)

23. CMP \$0,\$1,\$2; BNP \$0,Case1.

24. ANDN.

25. XOR \$X,\$Y,\$Z; SADD \$X,\$X,0.

26. ANDN \$X,\$Y,\$Z.

27. BDIF \$W,\$Y,\$Z; ADDU \$X,\$Z,\$W; SUBU \$W,\$Y,\$W.

28. BDIF \$0,\$Y,\$Z; BDIF \$X,\$Z,\$Y; OR \$X,\$0,\$X.

29. NOR \$0,\$Y,0; BDIF \$0,\$0,\$Z; NOR \$X,\$0,0. (Эта последовательность вычисляет $2^n - 1 - \max(0, (2^n - 1 - y) - z)$ для каждой байтовой позиции.)

30. XOR \$1,\$0,\$2; BDIF \$1,\$3,\$1; SADD \$1,\$1,0, если \$2 = #2020202020202020 и \$3 = #0101010101010101.

31. MXOR \$1,\$4,\$0; SADD \$1,\$1,0, если \$4 = #0101010101010101.

32. $C_{ji}^T = C_{ij} = (A_{1i}^T \bullet B_{j1}^T) \circ \dots \circ (A_{ni}^T \bullet B_{jn}^T) = (B^T \circ A^T)_{ji}$, если операция \bullet коммутативна.

33. MOR (или MXOR) с константой #0180402010080402.

34. MOR \$X,\$Z,[#0080004000200010]; MOR \$Y,\$Z,[#0008000400020001]. (Здесь квадратные скобки используются для обозначения регистров со вспомогательными константами.)

Для обратного преобразования достаточно применить 8-битовый код:

```
PUT    rM,[#00ff00ff00ff00ff]
MOR    $0,$X,[#4020100804020180]
MUX    $1,$0,$Y
BNZ    $1,BadCase
MUX    $1,$Y,$0
MOR    $Z,$1,[#8020080240100401]  █
```

35. MOR \$X,\$Y,\$Z; MOR \$X,\$Z,\$X; здесь \$Z является константой (14).

36. XOR \$0,\$Y,\$Z; MOR \$0,[-1],\$0. *Замечание:* замена XOR на BDIF дает маску для байтов, где \$Y превышает \$Z. С помощью такой маски операции AND с #8040201008040201 и операции MOR с #ff можно получить однобайтовую кодировку соответствующих байтовых позиций.

37. Пусть элементы поля являются многочленами в булевой матрице

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Например, этой матрицей является $m(\#402010080402018e)$, и если применить MXOR, то получим матрицу $m(\#2010080402018e47)$. Сумма и произведение таких элементов поля получаются с помощью операций XOR и MXOR, соответственно. Эта конструкция работает, поскольку $x^8 + x^6 + x^5 + x^4 + 1$ является примитивным многочленом по модулю 2 (см. раздел 3.2.2).

(Поле с 2^k элементами для $2 \leq k \leq 7$ получается аналогично на основе многочленов в матрице #0103, #020105, #04020109, #0804020112, #100804020121, #20100804020141. Матрицы с размерами до 16×16 можно представить в виде четырех октабайтов. Для организации умножения потребуется восемь операций MXOR и четыре операции XOR. Однако умножение можно выполнить в поле 2^{16} элементов, с помощью всего пяти операций MXOR и трех операций XOR, если представить большое поле как квадратичное расширение поля 2^8 элементов.)

38. Устанавливает в регистр \$1 сумму восьми знаковых байтов из \$0; кроме того, устанавливает в регистр \$2 самый правый ненулевой такой байт или нуль; и устанавливает в регистр \$0 нуль. (Замена SR на SRU приводит к тому, что байты рассматриваются как беззнаковые, а замена SLU на SL часто приводит к переполнению.)

39. Предполагаемое время выполнения (a) ($3v$ или $2v$) по сравнению с $2v$; (b) ($4v$ или $3v$) по сравнению с $2v$; (c) ($4v$ или $3v$) по сравнению с $3v$; (d) (v или $4v$) по сравнению с $2v$; (e) ($2v$ или $5v$) по сравнению с $2v$; (f) ($2v$ или $5v$) по сравнению с $3v$. Потому следует использовать условные инструкции в случаях (a, d) и (c, f), если только \$0 не является отрицательным с вероятностью $> 2/3$; в последнем случае нужно использовать варианты с PBN, (d) и (f). Условные инструкции всегда эффективнее в случаях (b, e).

Замена команды ADDU на команду ADD неэквивалентна из-за возможного переполнения.

40. Допустим, что с помощью G0 произошел переход к адресу #101; т.е. @ \leftarrow #101. Тетрабайт $M_4[\#101]$ точно такой же, что и тетрабайт $M_4[\#100]$. Если опкод этой инструкции имеет вид, например, PUSHJ, то регистр rJ будет содержать #105. Аналогично, если опкод этой инструкции имеет вид GETA \$0, @, то регистр \$0 будет содержать #101. В таких ситуациях значение @ в языке ассемблера MMIX немного отличается от фактического значения во время выполнения программы.

Программисты могут использовать эти принципы для передачи сигнала подпрограмме на основе двух замыкающих битов @. (Сложно, но почему бы не использовать уже имеющиеся биты?)

41. (a) Истинно. (b) Истинно. (c) Истинно. (d) Ложно, но становится истинным если использовать SRU вместо SR.

42. (a) NEG \$1,\$0; CSNN \$1,\$0,\$0. avoids overflow (b) ANDN \$1,\$0,[#8000000000000000].

43. Замыкающие нули (решение Дж. Даллоса): SUBU \$0,\$Z,1; SADD \$0,\$0,\$Z.

Ведущие нули: FLOTU \$0,1,\$Z; SRU \$0,\$0,52; SUB \$0,[1086],\$0. (В случае если \$Z может быть нулем, то следует добавить команду CSZ \$0,\$Z,64.) Это кратчайшая программа, но не самая быстрая. Можно было бы сэкономить $2v$, если обратить все биты (см. упр. 35) и подсчитать замыкающие биты.

44. Используйте “арифметику со старшей половиной тетрабайта”, в которой каждое 32-битовое число располагается в левой половине регистра. LDHT и STHT загружают и сохраняют такие величины (см. упр. 7); SETMH загружает немедленную константу. Для сложения, вычитания, умножения или деления старших половин тетрабайтов \$Y и \$Z с получением старшей половины тетрабайта \$X с учетом переполнения и контролем деления, следует использовать следующие команды: (a) ADD \$X,\$Y,\$Z. (b) SUB \$X,\$Y,\$Z. (c) SR \$X,\$Z,32; MUL \$X,\$Y,\$X (при условии, что $X \neq Y$). (d) DIV \$X,\$Y,\$Z; SL \$X,\$X,32; здесь rR — это остаток от деления старшей половины тетрабайта.

46. Он вызывает переход к позиции 0.

47. #DF имеет символьное имя MXORI (“multiple exclusive-or immediate” — “немедленное множественное исключительное или”); #55 имеет символьное имя PBPB (“probable branch positive backward” — “вероятное ветвление обратно положительное”). Но в программе используются имена MXOR и PBP; ассемблер самостоятельно добавляет I и B в случае необходимости.

48. STO и STOU; кроме того “немедленные” варианты LDOI и LDOUI, STOI и STOUI; NEGI и NEGUI, хотя опкод NEG не эквивалентен опкоду NEGU; а также любые два из четырех опкодов FLOTI, FLOTUI, SFLOTI и SFLOTUI.

(Каждая операция MMIX со знаковыми величинами имеет соответствующую ей операцию с беззнаковыми числами, получаемую добавлением 2 к опкоду. Это соответствие упрощает проектирование и создание компьютеров, а также создание компиляторов. Конечно, это уменьшает универсальность, поскольку ограничивает возможности других операторов.)

49. Октабайт $M_8[0]$ принимает вид #0000010000000001; rH — #0000012343210000; $M_2[\#0244420000000122]$ — #0121; rA — #00041 (поскольку в STW происходит перепол-

нение); $rB \leftarrow f(7) = \#401c000000000000$; $a\$1 \leftarrow \#6ff8ffffffffff$. (Кроме того, $rL \leftarrow 2$, если rL в исходном состоянии содержал 0 или 1.) Предполагается, что программа располагается так, что инструкции STC0, STB или STW не затирают полезные данные.

50. $4\mu + 34v = v + (\mu + v) + v + (\mu + v) + (\mu + v) + v + v + 10v + v + (\mu + v) + v + 4v + v + v + v + v + 3v + v + v + v$.

51. 35010001 a0010101 2e010101 a5010101 f6000001 c4010101
 b5010101 8e010101 1a010101 db010101 c7010101 3d010101
 33010101 e4010001 f7150001 08010001 5701ffff 3f010101

52. ADDI, ADDUI, SUBI, SUBUI, SLI, SLUI, SRI, SRUI, ORI, XORI, ANDNI, BDIFI, WDIFI, TDIFI, ODIFI: $X = Y = 255$, $Z = 0$. MULI: $X = Y = 255$, $Z = 1$. INCH, INCMH, INCL, INCL, ORH, ORMH, ORML, ORL, ANDNH, ANDNMH, ANDNML, ANDNL: $X = 255$, $Y = Z = 0$. OR, AND, MUX: $X = Y = Z = 255$. CSN, CSZ, ..., CSEV: $X = Z = 255$, Y произвольное. BN, BZ, ..., PBEV: X произвольное, $Y = 0$, $Z = 1$. JMP: $X = Y = 0$, $Z = 1$. PRELD, PRELDI, PREGO, PREGOI, SWYM: X, Y, Z произвольное. (Тонкий вопрос: инструкция, которая задает регистр $\$X$, не является холостой, если X является маргинальным, поскольку он приводит к увеличению rL , а все регистры, за исключением $\$255$, являются маргинальными, если $rL = 0$ и $rG = 255$.)

53. MULU, MULUI, PUT, PUTI, UNSAVE.

54. FCMP, FADD, FIX, FSUB, ..., FCMPE, FEQL, ..., FINT, MUL, MULI, DIV, DIVI, ADD, ADDI, SUB, SUBI, NEG, SL, SLI, STB, STBI, STW, STWI, STT, STTI, STSF, STSFI, PUT, PUTI, UNSAVE. (Этот не совсем корректный вопрос, поскольку полный набор правил для чисел с плавающей точкой здесь не представлен. Тонкий вопрос заключается в том, что FCMP может изменить I_BIT регистра rA , если $\$Y$ или $\$Z$ являются нечисловыми значениями, но FEQL и FUN никогда не приводят к исключительным ситуациям.)

55. FCMP, FUN, ..., SRUI, CSN, CSNI, ..., LDUNCI, GO, GOI, PUSHGO, PUSHGOI, OR, ORI, ..., ANDNL, PUSHJ, PUSHJB, GETA, GETAB, PUT, PUTI, POP, SAVE, UNSAVE, GET.

56. Для минимального объема памяти:

LDO	$\$1, x$	MUL	$\$0, \$0, \$1$
SET	$\$0, \1	SUB	$\$2, \$2, 1$
SETL	$\$2, 12$	PBP	$\$2, 0-4*2$ ■

Объем памяти = $6 \times 4 = 24$ байта, время = $\mu + 149v$. Можно получить более быстрые решения.

Для минимального времени: из допущения $|x^{13}| \leq 2^{63}$ следует, что $|x| < 2^5$ и $x^8 < 2^{39}$. Следующее решение основано на идее Й. Н. Патта (Y. N. Patt), в которой используется этот факт.

LDO	$\$0, x$	$\$0 = x$
MUL	$\$1, \$0, \$0$	$\$1 = x^2$
MUL	$\$1, \$1, \$1$	$\$1 = x^4$
SL	$\$2, \$1, 25$	$\$2 = 2^{25}x^4$
SL	$\$3, \$0, 39$	$\$3 = 2^{39}x$
ADD	$\$3, \$3, \$1$	$\$3 = 2^{39}x + x^4$
MULU	$\$1, \$3, \$2$	$u(\$1) = 2^{25}x^8, rH = x^5 + 2^{25}x^4 [x < 0]$
GET	$\$2, rH$	$\$2 \equiv x^5 \pmod{2^{25}}$
PUT	$rM, [\#1ffffff]$	
MUX	$\$2, \$2, \$0$	$\$2 = x^5$
SRU	$\$1, \$1, 25$	$\$1 = x^8$
MUL	$\$0, \$1, \$2$	$\$0 = x^{13}$ ■

Объем памяти = $12 \times 4 = 48$ байтов, время = $\mu + 48v$. Согласно теории из раздела 4.6.3, для этого "необходимо" выполнить, по крайней мере, пять операций умножения. Однако,

данная программа использует только четыре! На самом деле существует способ полного исключения операций умножения.

Для поистине минимального времени: как указал Флойд Р. В. (Floyd R. W.), $|x| \leq 28$, а потому минимальное время выполнения на основе таблицы (если только, не $\mu > 45\nu$):

LDO	\$0,x	$\$0 = x$
8ADDU	\$0,\$0,[Table]	
LDO	\$0,\$0,8*28	$\$0 = x^{13}$
...		
Таблица	OCTA	-28*28*28*28*28*28*28*28*28*28*28*28*28
	OCTA	-27*27*27*27*27*27*27*27*27*27*27*27*27
...		
	OCTA	28*28*28*28*28*28*28*28*28*28*28*28*28

Объем памяти = $3 \times 4 + 57 \times 8 = 468$ байтов, время = $2\mu + 3\nu$.

57. (1) Операционная система может выделять высокоскоростную память более эффективно, если известно, что программные блоки предназначены “только для чтения”. (2) Инструкция кэширования на аппаратном уровне выполняется быстрее и с меньшими затратами, если инструкции программы не изменяются. (3) Тот же ответ, что и (2), но вместо “кэширования” следует использовать “конвейеризация”. Если инструкция изменяется после конвейеризации, то конвейер нужно очистить; схема проверки этого условия очень сложна и требует больших затрат времени. (4) Самоизменяющийся код не может использоваться одновременно более, чем одним процессом. (5) Самоизменяющийся код может негативно повлиять на “профилирование” (т.е. вычисление количества выполняемых операций).

РАЗДЕЛ 1.3.2'

1. (a) Ссылка на метку строки 24. (b) Нет. В таком случае строка 23 будет ссылаться на строку 24 вместо строки 38, а строка 31 будет ссылаться на строку 24 вместо строки 21.

2. Текущее значение 9B будет текущим количеством таких строк, которые уже появились ранее.

3. Считывание 100 октабайтов из стандартного устройства ввода-вывода; перестановка их максимума и последнего значения; для остальных 99 значений — снова перестановка максимума и последнего значения и т.д. В конечном итоге 100 октабайтов будут отсортированы в порядке неубывания. Полученный результат записывается в стандартное устройство вывода. (Сравните с алгоритмом 5.2.3S.)

4. #2233445566778899. (Большие значения сокращаются по модулю 2^{64} .)

5. BYTE "silly"; но эту уловку не рекомендуется использовать.

6. Ложно; TETRA @,@ не то же самое, что TETRA @; TETRA @.

7. Он забыл, что относительные адреса относятся к положениям тетрабайтов, а два концевых бита игнорируются.

8. LOC 16*((@+15)/16), либо LOC -@/16*-16, либо LOC (@+15)&-16 и т.д.

9. Замените 500 на 600 в строке 02; замените Five на Six в строке 35. (Пятизначные числа не понадобятся до тех пор, пока не потребуется найти и напечатать 1230 или более простых чисел. Каждое из 6542 простых чисел помещается в рамках одного вайда.)

10. $M_2[\#2000000000000000] = \#0002$, следующие ненулевые данные находятся в текстовом сегменте:

#100: #e3 fe 00 03	#15c: #23 ff f6 00
#104: #c1 fb f7 00	#160: #00 00 07 01
#108: #a6 fe f8 fb	#164: #35 fa 00 02
#10c: #e7 fb 00 02	#168: #20 fa fa f7
#110: #42 fb 00 13	#16c: #23 ff f6 1b
#114: #e7 fe 00 02	#170: #00 00 07 01
#118: #c1 fa f7 00	#174: #86 f9 f8 fa
#11c: #86 f9 f8 fa	#178: #af f5 f8 00
#120: #1c fd fe f9	#17c: #23 ff f8 04
#124: #fe fc 00 06	#180: #1d f9 f9 0a
#128: #43 fc ff fb	#184: #fe fc 00 06
#12c: #30 ff fd f9	#188: #e7 fc 00 30
#130: #4d ff ff f6	#18c: #a3 fc ff 00
#134: #e7 fa 00 02	#190: #25 ff ff 01
#138: #f1 ff ff f9	#194: #5b f9 ff fb
#13c: #46 69 72 73	#198: #23 ff f8 00
#140: #74 20 46 69	#19c: #00 00 07 01
#144: #76 65 20 48	#1a0: #e7 fa 00 64
#148: #75 6e 64 72	#1a4: #51 fa ff f4
#14c: #65 64 20 50	#1a8: #23 ff f6 19
#150: #72 69 6d 65	#1ac: #00 00 07 01
#154: #73 0a 00 20	#1b0: #31 ff fa 62
#158: #20 20 00 00	#1b4: #5b ff ff ed

(Обратите внимание, что SET становится SETL в #100, но ORI в #104. Текущее положение @ выравнивается до #15c в строке 38, согласно правилу 7(a).) В начале программы rG содержит #f5 и далее \$248 = #200000000000003e8, \$247 = #ffffffffffffc1a, \$246 = #13c, \$245 = #2030303030000000.

11. (a) Если n не является простым, то по определению n имеет делитель d с $1 < d < n$. Если $d > \sqrt{n}$, то n/d является делителем с $1 < n/d < \sqrt{n}$. (b) Если n не является простым, то n имеет *простой* делитель d с $1 < d \leq \sqrt{n}$. Этот алгоритм проверяет, что n не имеет простых делителей $\leq p = \text{PRIME}[k]$. Кроме того, $n = pq + r < pq + p \leq p^2 + p < (p + 1)^2$. Следовательно, любой простой делитель n больше, чем $p + 1 > \sqrt{n}$.

Мы также должны доказать, что существует достаточно большое простое число меньше n , если n простое число такое, что $(k + 1)$ -е простое p_{k+1} меньше $p_k^2 + p_k$, а в противном случае k превышает j и $\text{PRIME}[k]$ будет равно нулю, если нужно чтобы оно было очень большим. Доказательство следует из "постулата Бертрана": если p является простым, то существует более крупное простое число меньше $2p$.

12. Переместить Title, NewLn и Blank в сегмент данных за BUF, где они могут использовать ртор в качестве базового адреса. Либо заменить инструкции LDA в строках 38, 42 и 58 на SETL, зная, что адреса этих строк помещаются в рамках двух байт, поскольку эта программа очень коротка. Либо заменить инструкции LDA на GETA, но в таком случае потребуется выровнять каждую строку по модулю 4, например

```
Title  BYTE  "Первые 500 простых чисел",#a,0
        LOC   (@+3)&-4
NewLn  BYTE  #a,0
        LOC   (@+3)&-4
Blanks BYTE  " ",0
```

(См. упражнения 7 и 8.)

13. Нужно вставить новый заголовок в строке 35, заменить BYTE на WYDE в строках 35–37, заменить Fputs на Fputws в строках 39, 43, 55, 59, заменить константу в строке 45 на #0020066006600660, заменить BUF+4 на BUF+2*4 в строке 47 и в строках 50–52 нужно использовать

```
INCL r,'.'; STWU r,t,0; SUB t,t,2.
```

Например, строка с новым заголовком может иметь вид

```
Title WYDE "أول خمس ميات الأرقام الأولية"
```

Она пишется в двух направлениях, но в компьютерном файле отдельные символы появляются в “логическом” порядке без лигатур. Потому последовательность

```
Title WYDE 'أ','و','ل',' ','خ','م','س',' ','...','ل','ي','ة'
```

даст эквивалентный результат, согласно правилу 2 работы со строками.

14. Например, можно заменить строки 26–30 в программе P следующими строками

```
fn   ^GREG   0
sqrtn GREG   0
      FLOT   fn,n
      FSQRT  sqrtn,fn
6H   LDWU    pk,ptop,kk
      FLOT   t,pk
      FREM   r,fn,t
      BZ     r,4B
7H   FCMP    t,sqrtn,t   █
```

Новая инструкция FREM выполняется 9597 раз, а не 9538, поскольку новая проверка на этапе P7 не так эффективна, как прежняя. Несмотря на это, вычисления с плавающей запятой сокращают время вычислений на $426192v - 59\mu$. Это заметное улучшение (если только не выполняется условие $\mu/v > 7000$). Дополнительной экономии $38169v$ можно добиться, если простые числа сохраняются в виде коротких чисел с плавающей запятой (short floats) вместо беззнаковых вайдов.

Количество проверок делимости можно сократить до 9357, если заменить q на $\sqrt{n} - 1.9999$ на этапе P7 (см. ответ к упр. 11). Однако дополнительные вычитания приводят к большим затратам времени, чем к их экономии, если только не выполняется условие $\mu/v > 15$.

15. Она выводит строку, состоящую из пробела, звездочки, двух пробелов, звездочки, ..., k пробелов, звездочки, ..., 74 пробелов, звездочки. Всего $2 + 3 + \dots + 75 = \binom{76}{2} - 1 = 2849$ символов.

17. Следующая подпрограмма возвращает нуль, тогда и только тогда, когда успешно завершается выполнение инструкции.

a	IS	#ffffffff	Элемент таблицы для любого действия
b	IS	#ffff04ff	Элемент таблицы для $Y \leq \text{ROUND_NEAR}$
c	IS	#001f00ff	Элемент таблицы для PUT и PUTI
d	IS	#ff000000	Элемент таблицы для RESUME
e	IS	#ffff0000	Элемент таблицы для SAVE
f	IS	#ff0000ff	Элемент таблицы для UNSAVE
g	IS	#ff000003	Элемент таблицы для SYNC
h	IS	#ffff001f	Элемент таблицы для GET
table	GREG	@	
	TETRA	a,a,a,a,a,b,a,b,b,b,b,b,b,b,b,b	0x

TETRA	a,a,a,a,a,b,a,b,a,a,a,a,a,a,a	1x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	2x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	3x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	4x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	5x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	6x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	7x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	8x	
TETRA	a,a,a,a,a,a,a,a,0,0,a,a,a,a,a,a	9x	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Ax	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Bx	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Cx	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Dx	
TETRA	a,a,a,a,a,a,a,a,a,a,a,a,a,a,a	Ex	
TETRA	a,a,a,a,a,a,c,c,a,d,e,f,g,a,h,a	Fx	
tetra	IS	\$1	
maxXYZ	IS	\$2	
InstTest	BN	\$0,9F	Неверно, если адрес отрицательный.
	LDTU	tetra,\$0,0	Выбрать тетрабайт.
	SR	\$0,tetra,22	Извлечь его опкод (умножить на 4).
	LDT	maxXYZ,table,\$0	Получить X_{\max} , Y_{\max} , Z_{\max} .
	BDIF	\$0,tetra,maxXYZ	Проверить, превышен ли максимум.
	PBNP	maxXYZ,9F	Если не PUT, завершить.
	ANDNML	\$0,#ff00	Обнулить байта OP.
	BNZ	\$0,9F	Ветвление, если превышен максимум.
	MOR	tetra,tetra,#4	Извлечь байт X.
	CMP	\$0,tetra,18	
	CSP	tetra,\$0,0	Установить $X \leftarrow 0$, если $18 < X < 32$.
	ODIF	\$0,tetra,7	Установить $\$0 \leftarrow X \div 7$.
9H	POP	1,0	Вернуть \$0 в качестве ответа. █

В этом решении тетрабайт не считается недопустимым, если совершается переход к отрицательному адресу, а также не считается недопустимой инструкция 'SAVE \$0,0' (хотя регистр \$0 никогда не может глобальным).

18. Особенность этой задачи в том, что в строке или столбце может быть несколько мест, где находится минимальное или максимальное значение, и каждое из них может быть седловой точкой.

Решение 1. В этом решении нужно обойти все строки, составляя список всех столбцов, в которых найдено минимальное значение, а затем проверить каждый столбец в списке, чтобы проверить совпадение минимума по строкам и максимума по столбцам. Обратите внимание, что во всех случаях для завершения цикла принято условие ≤ 0 .

*** Решение 1**

t	IS	\$255	
a00	GREG	Data_Segment	Адрес "a ₀₀ ".
a10	GREG	Data_Segment+8	Адрес "a ₁₀ ".
ij	IS	\$0	Индекс элемента и регистр возвращаемого значения.
j	GREG	0	Индекс столбца.
k	GREG	0	Размер списка минимальных индексов.
x	GREG	0	Текущее минимальное значение.
y	GREG	0	Текущий элемент.

Saddle	SET	ij,9*8	
RowMin	SET	j,8	
	LDB	x,a10,ij	Кандидат на минимальное значение строки.
2H	SET	k,0	Задать пустой список.
4H	INCL	k,1	
	STB	j,a00,k	Поместить индекс столбца в список.
1H	SUB	ij,ij,1	Переместиться на единицу влево.
	SUB	j,j,1	
	BZ	j,ColMax	Работа со строкой завершена?
3H	LDB	y,a10,ij	
	SUB	t,x,y	
	PBN	t,1B	Является ли x минимумом?
	SET	x,y	
	PBP	t,2B	Новый минимум?
	JMP	4B	Запомнить новый минимум.
ColMax	LDB	\$1,a00,k	Извлечь столбец из списка.
	ADD	j,\$1,9*8-8	
1H	LDB	y,a10,j	
	CMP	t,x,y	
	PBN	t,No	Верно ли, что минимум < элемента столбца?
	SUB	j,j,8	
	PBP	j,1B	Работа со столбцом завершена?
Yes	ADD	ij,ij,\$1	Да; $ij \leftarrow$ индекс седловой точки.
	LDA	ij,a10,ij	
	POP	1,0	
No	SUB	k,k,1	Пуст ли список?
	BP	k,ColMax	Если нет, попробовать снова.
	PBP	ij,RowMin	Все ли строки проверены?
	POP	1,0	Да; $\$0 = 0$, нет седловой точки. ■

Решение 2. Применение математики дает другой алгоритм.

Теорема. Пусть $R(i) = \min_j a_{ij}$, $C(j) = \max_i a_{ij}$. Элемент $a_{i_0 j_0}$ является седловой точкой тогда и только тогда, когда $R(i_0) = \max_i R(i) = C(j_0) = \min_j C(j)$.

Доказательство. Если $a_{i_0 j_0}$ является седловой точкой, тогда для любого фиксированного i , $R(i_0) = C(j_0) \geq a_{ij_0} \geq R(i)$; причем $R(i_0) = \max_i R(i)$. Аналогично, $C(j_0) = \min_j C(j)$. И наоборот, имеем $R(i) \leq a_{ij} \leq C(j)$ для всех i и j ; следовательно, $R(i_0) = C(j_0)$ предполагает, что $a_{i_0 j_0}$ является седловой точкой. ■

(Это доказательство показывает, что всегда $\max_i R(i) \leq \min_j C(j)$. Итак седловой точки нет, тогда и только тогда, когда все R меньше, чем все C .)

Согласно этой теореме, достаточно найти наименьший максимум по столбцам, а затем найти равный ему минимум по строкам.

* Решение 2

t	IS	\$255	
a00	GREG	Data_Segment	Адрес "a00".
a10	GREG	Data_Segment+8	Адрес "a10".
a20	GREG	Data_Segment+8*2	Адрес "a20".
ij	GREG	0	Индекс элемента.
ii	GREG	0	Индекс строки, умноженный на 8.
j	GREG	0	Индекс столбца.

x	GREG	0	Текущий максимум.
y	GREG	0	Текущий элемент.
z	GREG	0	Текущее наименьший максимум.
ans	IS	\$0	Регистр возвращаемого значения.
Phase1	SET	j,8	Начать со столбца 8.
	SET	z,1000	$z \leftarrow \infty$ (более или менее).
3H	ADD	ij,j,9*8-2*8	
	LDB	x,a20,ij	
1H	LDB	y,a10,ij	
	CMP	t,x,y	Верно ли, что $x < y$?
	CSN	x,t,y	Если да, то обновить максимум.
2H	SUB	ij,ij,8	Переместиться вверх на один элемент.
	PBP	ij,1B	
	STB	x,a10,ij	Сохранить значение максимума по столбцу.
	CMP	t,x,z	Верно ли, что $x < z$?
	CSN	z,t,x	Если да, то обновить наименьший максимум.
	SUB	j,j,1	Переместиться к столбцу слева.
	PBP	j,3B	
Phase2	SET	ii,9*8-8	(В этом месте $z = \min_j C(j)$.)
3H	ADD	ij,ii,8	Приготовиться к поиску по строке.
	SET	j,8	
1H	LDB	x,a10,ij	
	SUB	t,z,x	Верно ли, что $z > a_{ij}$?
	PBP	t,No	Седловой точки нет в данной строке.
	PBN	t,2F	
	LDB	x,a00,j	Верно ли, что $a_{ij} = C(j)$?
	CMP	t,x,z	
	CSZ	ans,t,ij	Если да, то запомнить возможную седловую точку.
2H	SUB	j,j,1	Переместиться в строке влево.
	SUB	ij,ij,1	
	PBP	j,1B	
	LDA	ans,a10,ans	Седловая точка найдена здесь.
	POP	1,0	
No	SUB	ii,ii,8	
	PBP	ii,3B	Проверить другую строку.
	SET	ans,0	
	POP	1,0	$\text{ans} = 0$; нет седловой точки. ■

Читателю предлагается найти еще более эффективное решение с записью на этапе Phase1 всех возможных строк, которые являются кандидатами для поиска на этапе Phase2. Необязательно проводить поиск по всем строкам. Достаточно проверить только те i_0 , для которых $C(j_0) = \min_j C(j)$ предполагает $a_{i_0 j_0} = C(j_0)$. Обычно существует не более одной такой строки.

В некоторых пробных запусках программы со случайным подбором элементов из множества $\{-2, -1, 0, 1, 2\}$, для Решения 1 требовалось приблизительно $147\mu + 863\nu$ времени, а для Решения 2 — $95\mu + 510\nu$. В случае матрицы с нулями для поиска седловой точки с помощью Решения 1 потребовалось $26\mu + 188\nu$, а с помощью Решения 2 — $96\mu + 517\nu$.

Если матрица $m \times n$ содержит *разные* элементы, причем $m \geq n$, то задачу можно решить, просматривая только $O(m+n)$ элементов и выполняя $O(m \log n)$ вспомогательных операций. См. Bienstock, Chung, Fredman, Schäffer, Shor, и Suri, АММ 98 (1991), 418-419.


19. Рассмотрим матрицу $m \times n$. (а) Согласно теореме из упражнения 18, все седловые точки матрицы имеют одинаковое значение. Поэтому (и на основании предположения о том, что матрица содержит различные элементы) существует, по крайней мере, одна седловая точка. Из соображений симметрии искомая вероятность равна произведению mn и вероятности того, что a_{11} является седловой точкой. Последняя равна произведению $1/(mn)!$ и количества перестановок с $a_{12} > a_{11}, \dots, a_{1n} > a_{11}, a_{11} > a_{21}, \dots, a_{11} > a_{m1}$; а оно равно произведению $1/(m+n-1)!$ и количества перестановок $m+n-1$ элементов, среди которых первый больше следующих $(m-1)$ и меньше остальных $(n-1)$, а именно $(m-1)!(n-1)!$. Следовательно, ответ будет иметь следующий вид:

$$mn(m-1)!(n-1)!/(m+n-1)! = (m+n) / \binom{m+n}{n}.$$

В нашем случае это равно $17/\binom{17}{8}$, т.е. только $1/1430$. (б) Во втором случае нужно применить совершенно другой метод, поскольку в данном случае возможно существование нескольких седловых точек. Действительно, целая строка или целый столбец должны состоять из седловых точек. Эта вероятность равна сумме вероятности того, что существует седловая точка с величиной 0, и вероятности того, что существует седловая точка с величиной 1. Первая вероятность равна вероятности того, что существует по крайней мере один столбец из нулей, а последняя вероятность равна вероятности того, что существует по крайней мере одна строка из единиц. Ответ имеет вид $(1 - (1 - 2^{-m})^n) + (1 - (1 - 2^{-n})^m)$. В данном случае $924744796234036231/18446744073709551616$, около $1/19,9$. Приблизительный ответ имеет вид $n2^{-m} + m2^{-n}$.

20. М. (Хоффри) (M. Hofri) и П. Джакет (P. Jacquet) [*Algorithmica* **22** (1998), 516–528] проанализировали случай, когда матрица $m \times n$ содержит различные и случайно упорядоченные элементы. Время выполнения двух упомянутых выше программ **MMIX** равно $(mn + mH_n + 2m + 1 + (m+1)/(n-1))\mu + (6mn + 7mH_n + 5m + 11 + 7(m+1)/(n-1))v + O((m+n)^2/\binom{m+n}{m})$ и $(m+1)n\mu + (5mn + 6m + 4n + 7H_n + 8)v + O(1/n) + O((\log n)^2/m)$, соответственно, если $m \rightarrow \infty$ и $n \rightarrow \infty$, при условии, что $(\log n)/m \rightarrow 0$.

21. Farey SET $y, 1; \dots$ POP.

 Этот ответ является первым ответом для тех многих задач в томах 1–3, для которых членам группы разработчиков **MMIXmasters** предлагается найти элегантные решения. (Более подробная информация предлагается по адресу, указанному на стр. ii.) В четвертом издании этой книги будут представлены самые интересные фрагменты наилучших программ. Замечание: пожалуйста, вместе со своим вариантом конкурсной программы приводите полное имя, чтобы правильно упомянуть его в списке разработчиков!

22. (а) Доказывается по индукции. (б) Пусть $k \geq 0$ и $X = ax_{k+1} - x_k$, $Y = ay_{k+1} - y_k$, где $a = \lfloor (y_k + n)/y_{k+1} \rfloor$. На основании (а) и того факта, что $0 < Y \leq n$, имеем $X \perp Y$ и $X/Y > x_{k+1}/y_{k+1}$. Потому, если $X/Y \neq x_{k+2}/y_{k+2}$, то по определению имеем $X/Y > x_{k+2}/y_{k+2}$. Отсюда следует, что

$$\begin{aligned} \frac{1}{Yy_{k+1}} &= \frac{Xy_{k+1} - Yx_{k+1}}{Yy_{k+1}} = \frac{X}{Y} - \frac{x_{k+1}}{y_{k+1}} \\ &= \left(\frac{X}{Y} - \frac{x_{k+2}}{y_{k+2}} \right) + \left(\frac{x_{k+2}}{y_{k+2}} - \frac{x_{k+1}}{y_{k+1}} \right) \\ &\geq \frac{1}{Yy_{k+2}} + \frac{1}{y_{k+1}y_{k+2}} = \frac{y_{k+1} + Y}{Yy_{k+1}y_{k+2}} \\ &> \frac{n}{Yy_{k+1}y_{k+2}} \geq \frac{1}{Yy_{k+1}}. \end{aligned}$$

Исторические замечания. К. Харос (C. Haros) предложил (более сложное) правило для построения таких последовательностей в *J. de l'École Polytechnique* 4, 11 (1802), 364–368. Хотя этот метод верен, но его доказательство было не совсем строгим. Несколько лет спустя геолог Джон Фэри (John Farey) независимо предположил, что x_k/y_k всегда равно $(x_{k-1} + x_{k+1})/(y_{k-1} + y_{k+1})$ [*Philos. Magazine и Journal* 47 (1816), 385–386]. Доказательство вскоре было предложено О. Коши А. Cauchy [*Bull. Société Philomathique de Paris* (3) 3 (1816), 133–135], который предложил имя Фэри для данной последовательности. Более подробно эта последовательность и ее свойства описываются в книге G. H. Hardy и E. M. Wright, *An Introduction to the Theory of Numbers*, Chapter 3.

23. Приведенная ниже подпрограмма подходит для большинства типов конвейеров и кэш-памяти.

a	IS	\$0		SUB	n,n,8		STCO	0,a,56
n	IS	\$1		ADD	a,a,8		ADD	a,a,64
z	IS	\$2	3H	И	t,a,63		PBNN	t,4B
t	IS	\$255		PBNZ	t,2B	5H	CMP	t,n,8
				CMP	t,n,64		BN	t,7F
1H	STB	z,a,0		BN	t,5F	6H	STCO	0,a,0
	SUB	n,n,1	4H	PREST	63,a,0		SUB	n,n,8
	ADD	a,a,1		SUB	n,n,64		ADD	a,a,8
Zero	BZ	n,9F		CMP	t,n,64		CMP	t,n,8
	SET	z,0		STCO	0,a,0		PBNN	t,6B
	И	t,a,7		STCO	0,a,8	7H	BZ	n,9F
	BNZ	t,1B		STCO	0,a,16	8H	STB	z,a,0
	CMP	t,n,64		STCO	0,a,24		SUB	n,n,1
	PBNN	t,3F		STCO	0,a,32		ADD	a,a,1
	JMP	5F		STCO	0,a,40		PBNZ	n,8B
2H	STCO	0,a,0		STCO	0,a,48	9H	POP	█

24. Читателям рекомендуется внимательно изучить приведенную ниже подпрограмму. Более быструю подпрограмму можно было бы создать в особом случае, когда $\$0 \equiv \1 (по модулю 8).

in	IS	\$2		LDOU	in,\$0,0
out	IS	\$3		PUT	rM,m
r	IS	\$4		NEG	mm,0,1
l	IS	\$5		BN	r,1F
m	IS	\$6		NEG	l,64,r
t	IS	\$7		SLU	tt,out,r
mm	IS	\$8		MUX	in,in,tt
tt	IS	\$9		BDIF	t,ones,in
flip	GREG	#0102040810204080		И	t,t,m
ones	GREG	#0101010101010101		SRU	mm,mm,r
	LOC	#100		PUT	rM,mm
StrCpy	И	in,\$0,#7		JMP	4F
	SLU	in,in,3	1H	NEG	l,0,r
	И	out,\$1,#7		INCL	r,64
	SLU	out,out,3		SUB	\$1,\$1,8
	SUB	r,out,in		SRU	out,out,1
	LDOU	out,\$1,0		MUX	in,in,out
	SUB	\$1,\$1,\$0		BDIF	t,ones,in
	NEG	m,0,1		И	t,t,m
	SRU	m,m,in		SRU	mm,mm,r

	PUT	rM,mm		STOU	out,\$0,\$1
	PBZ	t,2F	5H	INCL	\$0,8
	JMP	5F		SLU	out,in,1
3H	MUX	out,tt,out		SLU	mm,t,1
	STOU	out,\$0,\$1	1H	LDOU	in,\$0,\$1
2H	SLU	out,in,1		MOR	mm,mm,flip
	LDOU	in,\$0,8		SUBU	t,mm,1
	INCL	\$0,8		ANDN	mm,mm,t
	BDIF	t,ones,in		MOR	mm,mm,flip
4H	SRU	tt,in,r		SUBU	mm,mm,1
	PBZ	t,3B		PUT	rM,mm
	SRU	mm,t,r		MUX	in,in,out
	MUX	out,tt,out		STOU	in,\$0,\$1
	BNZ	mm,1F		POP	0

Время выполнения приблизительно равно $(n/4 + 4)\mu + (n + 40)v$ плюс время выполнения операции POP. Оно меньше времени выполнения тривиального кода, если $n \geq 8$ и $\mu \geq v$.

25. Допустим, что регистр p в исходном состоянии содержит адрес первого байта, а этот адрес кратен 8. Кроме него, объявлены другие локальные или глобальные регистры a , b , Приведенное ниже решение начинается с подсчета частоты появления вайдов, поскольку для этого требуется вдвое меньше операций по сравнению с подсчетом частоты появления байтов. Затем частоты появления байтов получаются как суммы строк и столбцов матрицы 256×256 .

* Задача криптоанализа (СЕКРЕТНО)

	LOC	Data_Segment		
count	GREG	@	Базовый адрес для отсчетов вайдов.	
	LOC	@+8*(1<<16)	Место для частот появления вайдов.	
freq	GREG	@	Базовый адрес для отсчетов байтов.	
	LOC	@+8*(1<<8)	Место для частот появления байтов.	
p	GREG	@		
	BYTE	"abracadabraa",0,"abc"	Тривиальные тестовые данные.	
ones	GREG	#0101010101010101		
	LOC	#100		
2H	SRU	b,a,45	Изолировать следующий вайд.	} Основной цикл, который выполняется как можно быстрее.
	LDO	c,count,b	Загрузить прежний счетчик.	
	INCL	c,1		
	STO	c,count,b	Сохранить новый счетчик.	
	SLU	a,a,16	Удалить один вайд.	
	PBNZ	a,2B	Завершить работу с октабайтом?	
Phase1	LDOU	a,p,0	Начало: следующие восемь байтов.	
	INCL	p,8		
	BDIF	t,ones,a	Проверить наличие нулевого байта.	
	PBZ	t,2B	Выполнить основной цикл до достижения конца.	
2H	SRU	b,a,45	Изолировать следующий вайд.	
	LDO	c,count,b	Загрузить прежнее количество отсчетов.	
	INCL	c,1		
	STO	c,count,b	Сохранить новое количество отсчетов.	
	SRU	b,t,48		
	SLU	a,a,16		
	BDIF	t,ones,a		

	PBZ	b, 2B	Продолжить до завершения.
Phase2	SET	p, 8*255	Подготовка суммирования по строкам и столбцам.
1H	SL	a, p, 8	
	LDA	a, count, a	a ← адрес строки p.
	SET	b, 8*255	
	LDO	c, a, 0	
	SET	t, p	
2H	INCL	t, #800	
	LDO	x, count, t	Элемент столбца p
	LDO	y, a, b	Элемент строки p
	ADD	c, c, x	
	ADD	c, c, y	
	SUB	b, b, 8	
	PBP	b, 2B	
	STO	c, freq, p	
	SUB	p, p, 8	
	PBP	p, 1B	
	POP	█	

Этот двухэтапный (*Phase1* и *Phase2*) метод хуже простого одноэтапного подхода, если длина строки n меньше 2^{17} . Но длительность его выполнения равна $10/17$ от времени одноэтапного подхода, если $n \approx 10^6$. Немного более быструю программу можно получить за счет “развертки” внутреннего цикла, как это показано в ответе для следующего упражнения.

Еще один подход с использованием таблицы переходов и сохранением количества отсчетов в 128 регистрах имеет смысл применять, если величина μ/ν достаточно велика.

[Эта задача имеет длинную историю. См., например, Charles P. Bourne и Donald F. Ford, “A study of the statistics of letters in English words,” *Information и Control* 4 (1961), 48–67.]

26. Трюк с подсчетом частоты появления вайдов в решении из предыдущего упражнения станет препятствием, если первичный кэш компьютера содержит менее 2^{19} байтов, если только нет сравнительно малого количества отсчетов появления ненулевых вайдов. В приведенной ниже программе подсчитывается частота появления байтов. Этот код позволяет избежать останова в обычном конвейере, не используя результат операции LDO в следующей за ней инструкции.

Start	LDOU a, p, 0	INCL c, 1	INCL c, 1
	INCL p, 8	SRU b, b, 53	BDIF t, ones, a
	BDIF t, ones, a	STO c, freq, bb	STO c, freq, bb
	BNZ t, 3F	LDO c, freq, b	PBZ t, 2B
2H	SRU b, a, 53	...	3H SRU b, a, 53
	LDO c, freq, b	SLU bb, a, 56	LDO c, freq, b
	SLU bb, a, 8	INCL c, 1	INCL c, 1
	INCL c, 1	SRU bb, bb, 53	STO c, freq, b
	SRU bb, bb, 53	STO c, freq, b	SRU b, b, 3
	STO c, freq, b	LDO c, freq, bb	SLU a, a, 8
	LDO c, freq, bb	LDOU a, p, 0	PBNZ b, 3B
	SLU b, a, 16	INCL p, 8	POP █

Еще одно приведенное ниже решение выполняется еще быстрее на суперскалярном компьютере, который одновременно выдает две инструкции.

Start	LDOU a,p,0	SLU bbb,a,48
	INCL p,8	SLU bbbb,a,56
	BDIF t,ones,a	INCL c,1
	SLU bb,a,8	INCL cc,1
	BNZ t,3F	SRU bbb,bbb,53
2H	SRU b,a,53	SRU bbbb,bbbb,53
	SRU bb,bb,53	STO c,freq,b
	LDO c,freq,b	STO cc,freqq,bb
	LDO cc,freqq,bb	LDO c,freq,bbb
	SLU bbb,a,16	LDO cc,freqq,bbbb
	SLU bbbb,a,24	LDOU a,p,0
	INCL c,1	INCL p,8
	INCL cc,1	INCL c,1
	SRU bbb,bbb,53	INCL cc,1
	SRU bbbb,bbbb,53	BDIF t,ones,a
	STO c,freq,b	SLU bb,a,8
	STO cc,freqq,bb	STO c,freq,bbb
	LDO c,freq,bbb	STO cc,freqq,bbbb
	LDO cc,freqq,bbbb	PBZ t,2B
	SLU b,a,32	3H SRU b,a,53
	SLU bb,a,40	...
	...	█

В этом случае нужно использовать две отдельные таблицы частот (и комбинировать их в конце). В противном случае возникает проблема “совмещения имен”, которая приводит к неверным результатам в случаях, когда *b* и *bb* представляют одинаковый символ.

27. (a)

t	IS	\$255	FDIV	acc,phi,rt5
n	IS	\$0	SET	n,1
new	GREG		SET	new,1
old	GREG		1H ADDU	new,new,old
phi	GREG		INCL	n,1
rt5	GREG		CMPU	t,new,old
acc	GREG		BN	t,9F
f	GREG		SUBU	old,new,old
	LOC	#100	FMUL	acc,acc,phi
Main	FLOT	t,5	FIXU	f,acc
	FSQRT	rt5,t	CMP	t,f,new
	FLOT	t,1	PBZ	t,1B
	FADD	phi,t,rt5	SET	t,1
	INCH	phi,#fff0	9H TRAP	0,Halt,0 █

(b)

t	IS	\$255		LOC	#100
n	IS	\$0		Main	SET n,2
new	GREG				SET old,1
old	GREG				SET new,1
phii	GREG	#9e3779b97f4a7c16	1H	ADDU	new,new,old
lo	GREG			INCL	n,1
hi	GREG			CMPU	t,new,old
hihi	GREG			BN	t,9F


```

SUBU    old,new,old
MULU    lo,old,phii
GET      hi,rH
ADDU    hi,hi,old
ADDU    hihi,hi,1
CSN      hi,lo,hihi
CMP      t,hi,new
PBZ      t,1B
SET      t,1
9H TRAP  0,Halt,0   █

```

Программа (а) прекращает работу при $t = 1$ и $n = 71$; для представления с плавающей точкой ϕ ошибки в конечном итоге накапливаются до тех пор, пока $\phi^{71}/\sqrt{5}$ не аппроксимируется $F_{71} + .7$, с округлением до $F_{71} + 1$. Программа (b) прекращает работу при $t = -1$ и $n = 94$; беззнаковое переполнение возникает до нарушения приближения. (Действительно, $F_{93} < 2^{64} < F_{94}$.)

29.* Последним будет исключен человек в позиции 15. Общее время равно ...

 Члены группы MMIXmasters, пожалуйста, помогите! Предложите наиболее оптимальную программу, аналогичную решениям для упражнений 1.3.2–22 в третьем издании. Кроме того, чтобы смог бы сделать Д. Ингаллс (D. Ingalls) в этой новой ситуации? (Найдите трюк, аналогичный предыдущей схеме, не используя самоизменяющийся код.)

Асимптотически более быстрый метод представлен в упражнениях 5.1.1–5.

30. Будем работать с перенормированными значениями, $R_n = 10^n r_n$. $R_n(1/m) = R$ тогда и только тогда, когда $10^n/(R + \frac{1}{2}) \leq m < 10^n/(R - \frac{1}{2})$; таким образом $m_{k+1} = \lfloor (2 \cdot 10^n - 1)/(2R - 1) \rfloor$.

* Сумма округленного гармонического ряда

MaxN	IS	10	
a	GREG	0	Накопитель.
c	GREG	0	$2 \cdot 10^n$
d	GREG	0	Делитель или число.
r	GREG	0	Перенормированное значение.
s	GREG	0	Перенормированная сумма.
m	GREG	0	m_k
mm	GREG	0	m_{k+1}
nn	GREG	0	$n - \text{MaxN}$
	LOC	Data_Segment	
dec	GREG	@+3	Место десятичной запятой.
	BYTE	" . "	
	LOC	#100	
Main	NEG	nn,MaxN-1	$n \leftarrow 1$.
	SET	c,20	
1H	SET	m,1	
	SR	s,c,1	$S \leftarrow 10^n$.
	JMP	2F	
3H	SUB	a,c,1	
	SL	d,r,1	
	SUB	d,d,1	
	DIV	mm,a,d	
4H	SUB	a,mm,m	


	MUL	a, r, a	
	ADD	s, s, a	
	SET	m, mm	$k \leftarrow k + 1.$
2H	ADD	a, c, m	
	2ADDU	d, m, 2	
	DIV	r, a, d	
	PBNZ	r, 3B	
5H	ADD	a, nn, MaxN+1	
	SET	d, #a	Новая строка.
	JMP	7F	
6H	DIV	s, s, 10	Преобразование цифр.
	GET	d, rR	
	INCL	d, '0'	
7H	STB	d, dec, a	
	SUB	a, a, 1	
	BZ	a, @-4	
	PBNZ	s, 6B	
8H	SUB	\$255, dec, 3	
	TRAP	0, Fputs, StdOut	
9H	INCL	nn, 1	$n \leftarrow n + 1.$
	MUL	c, c, 10	
	PBNP	nn, 1B	
	TRAP	0, Halt, 0	■

Результаты 3.7, 6.13, 8.445, 10.7504, 13.05357, 15.356255, 17.6588268, 19.96140681, 22.263991769, 24.5665766342 будут получены за $82\mu + 40659359v$. Эти вычисления не будут вызывать переполнение для значений n не более 17, а время выполнения будет порядка $10^{n/2}$. (Это время можно сократить вдвое, вычисляя $R_n(1/m)$ непосредственно, если $m < 10^{n/2}$, и с помощью того факта, что $R_n(m_{k+1}) = R_n(m_k - 1)$ для более крупных значений m .)

31. Пусть $N = \lfloor 2 \cdot 10^n / (2m + 1) \rfloor$. Тогда $S_n = H_N + O(N/10^n) + \sum_{k=1}^m ([2 \cdot 10^n / (2k - 1)] - [2 \cdot 10^n / (2k + 1)])k/10^n = H_N + O(m^{-1}) + O(m/10^n) - 1 + 2H_{2m} - H_m = n \ln 10 + 2\gamma - 1 + 2 \ln 2 + O(10^{-n/2})$, если суммировать по частям и считать $m \approx 10^{n/2}$.

Наше приближение для S_{10} дает результат ≈ 24.5665766209 , что ближе к истине, чем предсказывалось.

32. Чтобы усложнить задачу, предложите более эффективное решение, чем решение — в котором используется несколько *хитростей* для сокращения времени выполнения. Сможет ли читатель предложить более быстрое решение?

 **MMIXmasters:** Члены группы MMIXmasters, пожалуйста, помогите заполнить приведенный выше пробел! Обратите внимание, что остатки от деления на 7, 19 и 30 быстрее всего вычисляются *FREM*; а деление на 100 можно заменить умножением на $1//100+1$ (см. упражнение 1.3.1'-19) и т.д.

[Для вычисления даты Пасхи для периода ≤ 1582 , см. *CACM* 5 (1962), 209–210. Первый систематический алгоритм вычисления даты Пасхи, *Пасхальный канон* (*canon paschalis*), был предложен Викториусом Аквитанским (Victorius of Aquitania) в 457 году. Существует множество свидетельств о том, что единственным нетривиальным применением арифметики в Европе в Средние века было вычисление даты Пасхи, а потому такие алгоритмы имеют историческое значение. Более подробные сведения приводятся в *Puzzles and Paradoxes* by Т. Н. О'Beirne (London: Oxford University Press, 1965), Chapter 10; а также

в книге *Calendrical Calculations* by E. M. Reingold and N. Dershowitz (Cambridge Univ. Press, 2001) for date-oriented algorithms of all kinds.]

33. Первым таким годом является 10317 год, хотя данная ошибка *почти* приводит к получению неверного результата в годах $10108 + 19k$ для $0 \leq k \leq 10$.

Т. Х. О'Бэен (Т. Н. О'Beirne) обнаружил, что дата Пасхи повторяется с периодом 5700000 лет. Вычисления Роберта Хилла показывают, что наиболее частой датой является 19 апреля (220400 раз за один период), самой ранней и редкой является 22 марта (27550 раз), а самой поздней и второй по редкости является 25 апреля (42000 раз). Хилл нашел интересное объяснение этого забавного факта: количество повторов любой даты Пасхи за один период всегда кратно 25.

34. Приведенная ниже программа соответствует заданному протоколу в пределах десяти ν ; этой точности более, чем достаточно, поскольку ρ обычно более 10^8 , а $\rho\nu = 1$ сек. Все вычисления происходят в регистрах, за исключением случая, когда используется входной байт.

* Задача со светофором

rho	GREG	2500000000	Предположим, что тактовая частота равна 250 МГц.
t	IS	\$255	
Sensor_Buf	IS	Data_Segment	
	GREG	Sensor_Buf	
	LOC	#100	
Lights	IS	3	Дескриптор for /dev/lights.
Sensor	IS	4	Дескриптор for /dev/sensor.
Lights_Name	BYTE	"/dev/lights",0	
Sensor_Name	BYTE	"/dev/sensor",0	
Lights_Args	OCTA	Lights_Name,BinaryWrite	
Sensor_Args	OCTA	Sensor_Name,BinaryRead	
Read_Sensor	OCTA	Sensor_Buf,1	
Boulevard	BYTE	#77,0	Зеленый/красный, ИДИТЕ/СТОЙТЕ.
	BYTE	#7f,0	Зеленый/красный, СТОЙТЕ/СТОЙТЕ.
	BYTE	#73,0	Зеленый/красный, выключен/СТОЙТЕ.
	BYTE	#bf,0	Желтый/красный, СТОЙТЕ/СТОЙТЕ.
Avenue	BYTE	#dd,0	Красный/зеленый, СТОЙТЕ/ИДИТЕ.
	BYTE	#df,0	Красный/зеленый, СТОЙТЕ/СТОЙТЕ.
	BYTE	#dc,0	Красный/зеленый, СТОЙТЕ/выключен.
	BYTE	#ef,0	Красный/желтый, СТОЙТЕ/СТОЙТЕ.
goal	GREG	0	Время смены цветов.
Main	GETA	t,Lights_Args	Открыть файлы: Fopen(Lights,
	TRAP	0,Fopen,Lights	"/dev/lights",BinaryWrite)
	GETA	t,Sensor_Args	Fopen(Sensor,
	TRAP	0,Fopen,Sensor	"/dev/sensor",BinaryRead)
	GET	goal,rC	
	JMP	2F	
	GREG	@	
delay_go	GREG		
Delay	GET	t,rC	Подпрограмма ожидания:
	SUBU	t,t,goal	(N.B. Не CMPU; см. ниже.)
	PBN	t,Delay	Повторить пока rC не пройдет goal.
	GO	delay_go,delay_go,0	Вернуться к вызывающему оператору.

flash_go	GREG		
n	GREG	0	Счетчик итераций.
green	GREG	0	Boulevard (Бульвар Дель Мар) или Avenue (Беркли Авеню).
temp	GREG		
Flash	SET	n,8	Подпрограмма зажигания огней.
1H	ADD	t,green,2*1	
	TRAP	0,Fputs,Lights	СТОЙТЕ.
	ADD	temp,goal,rho	
	SR	t,rho,1	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	
	ADD	t,green,2*2	
	TRAP	0,Fputs,Lights	(выключен)
	SET	goal,temp	
	GO	delay_go,Delay	
	SUB	n,n,1	
	PBP	n,1B	Повторить восемь раз.
	ADD	t,green,2*1	
	TRAP	0,Fputs,Lights	СТОЙТЕ.
	MUL	t,rho,4	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	Задержка 4 секунды.
	ADD	t,green,2*3	
	TRAP	0,Fputs,Lights	СТОЙТЕ, желтый.
	GO	flash_go,flash_go,0	Вернуться к вызывающему оператору.
Wait	GET	goal,rC	Зеленый цвет продолжительностью 18 секунд.
1H	GETA	t,Read_Sensor	
	TRAP	0,Fread,Sensor	
	LDB	t,Sensor_Buf	
	BZ	t,Wait	Повторять пока сенсор не равен нулю.
	GETA	green,Boulevard	
	GO	flash_go,Flash	Завершить цикл для бульвара Дель Мар.
	MUL	t,rho,8	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	Желтый цвет продолжительностью 8 секунд.
	GETA	t,Avenue	
	TRAP	0,Fputs,Lights	Зеленый цвет для Беркли Авеню.
	MUL	t,rho,8	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	
	GETA	green,Avenue	
	GO	flash_go,Flash	Завершить цикл для Беркли Авеню.
	GETA	t,Read_Sensor	
	TRAP	0,Fread,Sensor	Игнорировать сенсор во время зеленого цвета.
	MUL	t,rho,5	
	ADDU	goal,goal,t	

	GO	delay_go,Delay	Желтый цвет продолжительностью 5 секунд.
2H	GETA	t,Boulevard	
	TRAP	0,Fputs,Lights	Зеленый цвет для бульвара Дель Мар.
	MUL	t,rho,18	
	ADDU	goal,goal,t	
	GO	delay_go,Delay	По крайней мере 18 секунд для сигнала ИДИТЕ.
	JMP	1B	■

Инструкция SUBU в подпрограмме Delay представляет собой интересный пример случая, когда сравнение выполняется с помощью оператора SUBU, а не с помощью оператора CMPU, вопреки комментариям к упражнениям 1.3.1'–22. Дело в том, что две сравниваемые величины, rC и $goal$, “обернуты” по модулю 2^{64} .

РАЗДЕЛ 1.4.1'

1. j GREG ;m GREG ;kk GREG ;xk GREG ;rr GREG

	GREG	@	Базовый адрес.
GoMax	SET	\$2,1	Специальный вход для $r = 1$.
GoMaxR	SL	rr,\$2,3	Умножить аргументы на 8.
	SL	kk,\$1,3	
	LDO	m,x0,kk	
	...		(Продолжить, как в (1)).
5H	SUB	kk,kk,rr	$k \leftarrow k - r$.
	PBP	kk,3B	Повторить, если $k > 0$.
6H	GO	kk,\$0,0	Вернуться в вызывающую программу. ■

Вызывающая последовательность для общего случая имеет вид SET \$2,r; SET \$1,n; GO \$0,GoMaxR.

2. j IS \$0 ;m IS \$1 ;kk IS \$2 ;xk IS \$3 ;rr IS \$4

Max100	SET	\$0,100	Специальный вход для $n = 100$ и $r = 1$.
Max	SET	\$1,1	Специальный вход для $r = 1$.
MaxR	SL	rr,\$1,3	Умножить аргументы на 8.
	SL	kk,\$0,3	
	LDO	m,x0,kk	
	...		(Продолжить, как в (1)).
5H	SUB	kk,kk,rr	$k \leftarrow k - r$.
	PBP	kk,3B	Повторить, если $k > 0$.
6H	POP	2,0	Вернуться в вызывающую программу. ■

В этом случае вызывающая последовательность имеет вид SET \$A1,r; SET \$A0,n; PUSHJ \$R,MaxR, где $A0 = R + 1$ и $A1 = R + 2$.

3. Достаточно внести изменение Sub ...; GO \$0,\$0,0. Локальные переменные могут полностью храниться в регистрах.

4. PUSHJ \$X,RA имеет относительный адрес, позволяющий перейти к любой подпрограмме в пределах $\pm 2^{18}$ байт от текущего положения. PUSHGO \$X,\$Y,\$Z или PUSHGO \$X,A имеет абсолютный адрес, позволяющий перейти к любой подпрограмме в любом месте.

5. Истинно, поскольку существует 256 – G глобальных и L локальных регистров.

6. $\$5 \leftarrow rD$ и $rR \leftarrow 0$ и $rL \leftarrow 6$. Все другие новые локальные регистры устанавливаются в нуль; например, если rL был равен 3, то инструкция DIVU задает $\$3 \leftarrow 0$ и $\$4 \leftarrow 0$.

7. $\$L \leftarrow 0, \dots, \$4 \leftarrow 0, \$5 \leftarrow \text{\#abcd0000}, rL \leftarrow 6$.

8. Обычно такая инструкция не имеет заметного влияния, за исключением того, что в контексте команд **SAVE** и **UNSAVE** требуется больше времени, если имеется меньше маргинальных регистров. Однако в некоторых случаях она может иметь заметное влияние. Например, последующая инструкция **PUSHJ \$255,Sub**, а за ней инструкция **POP 1,0** может расположить результат в регистре \$16, а не в \$10.

9. **PUSHJ \$255,Handler** дает, по крайней мере, 32 маргинальных регистра (поскольку $G \geq 32$); затем **POP 0** восстанавливает прежние локальные регистры, а "**PUT rJ,\$255; GET \$255,rB; RESUME**" после **PUSHJ** перезапускают программу, если ничего не происходит.

10. В основном истинно. **MMIX** запустит программу, причем в регистре **rG** будет значение 255 минус количество ассемблированных операций **GREG**, а в регистре **rL** — значение 2. Тогда, в отсутствие инструкций **PUSHJ**, **PUSHGO**, **POP**, **SAVE**, **UNSAVE**, **GET** и **PUT** значение **rG** никогда не изменится. Значение регистра **rL** увеличится, если программа поместит нечто в \$2, \$3, ... или \$(**rG** - 1), но результат будет таким же, как если бы все регистры были эквивалентны. Единственным регистром с немного другим поведением является регистр \$255, на который оказывают влияние прерывания обхода и который используется в ловушках ввода-вывода. Можно было бы переставить номера регистров \$2, \$3, ..., \$254 произвольным образом в любой программе без инструкций **PUSH/POP/SAVE/UNSAVE/RESUME**, в которой не извлекается (**GET**) ничего из регистра **rL** или не кладется (**PUT**) ничего в регистр **rL** или регистр **rG**. Программа с такими перестановками даст идентичный результат.

Различие между локальными, глобальными и маргинальными регистрами несущественны по отношению к **SAVE**, **UNSAVE** и **RESUME**, в отсутствие **PUSH** и **POP**, за исключением того, что регистр назначения команды **SAVE** должен быть глобальным и регистр назначения некоторых инструкций, вставленных инструкцией **RESUME** не должен быть маргинальным (см. упражнение 1.4.3'-14).

11. Компьютер попытается получить доступ к виртуальному адресу **#5ffffffffff8**, который находится ниже сегмента стека. По этому адресу ничего нет, потому произойдет "ошибка чтения страницы", и операционная система прекратит выполнение программы.

(Поведение, однако, может оказаться гораздо более странным, если инструкция **POP** будет выполнена после инструкции **SAVE**, поскольку инструкция **SAVE** начинает новый стек регистров сразу же после сохраненного контекста. Всякий, кто попытается выполнить такие действия, навлечет на себя серьезные неприятности.)

12. (a) Истинно. (Аналогично, имя текущего "рабочего каталога" в оболочке **UNIX** всегда начинается со слэша, т.е. косой черты.) (b) Ложно. Но если определены такие префиксы, может возникнуть путаница, и поэтому их не рекомендуется использовать. (c) Ложно. (В этом отношении структурированные символы **MMIXAL** не похожи на имена каталогов **UNIX**.)

13. Fib	CMP	\$1,\$0,2	Fib1	CMP	\$1,\$0,2	Fib2	CMP	\$1,\$0,1
	PBN	\$1,1F		BN	\$1,1F		BNP	\$1,1F
	GET	\$1,rJ		SUB	\$2,\$0,1		SUB	\$2,\$0,1
	SUB	\$3,\$0,1		SET	\$0,1		SET	\$0,0
	PUSHJ	\$2,Fib		SET	\$1,0	2H	ADDU	\$0,\$0,\$1
	SUB	\$4,\$0,2	2H	ADDU	\$0,\$0,\$1		ADDU	\$1,\$0,\$1
	PUSHJ	\$3,Fib		SUBU	\$1,\$0,\$1		SUB	\$2,\$2,2
	ADDU	\$0,\$2,\$3		SUB	\$2,\$2,1		PBP	\$2,2B
	PUT	rJ,\$1		PBNZ	\$2,2B		CSZ	\$0,\$2,\$1
1H	POP	1,0	1H	POP	1,0	1H	POP	1,0

Здесь **Fib2** является более быстрой альтернативой **Fib1**. В каждом случае вызывающая последовательность имеет вид "**SET \$A,n; PUSHJ \$R,Fib...**", где $A = R + 1$.

14. Согласно математической индукции, инструкция POP в подпрограмме Fib выполняется точно $2F_{n+1} - 1$ раз, и инструкция ADDU выполняется $F_{n+1} - 1$ раз. Инструкция у 2H выполняется $n - [n \neq 0]$ раз в подпрограмме Fib1 и $[n/2]$ раз в подпрограмме Fib2. Таким образом, общие затраты, включая две инструкции в вызывающей последовательности, равны $(19F_{n+1} - 12)v$ для подпрограммы Fib, $(4n + 8)v$ для подпрограммы Fib1 и $(4[n/2] + 12)v$ для подпрограммы Fib2, при условии, что $n > 1$.

(Рекурсивная подпрограмма Fib представляет собой ужасный способ вычисления чисел Фибоначчи, поскольку она не запоминает уже вычисленные значения. Для вычисления F_{100} требуется $10^{22}v$ единиц времени.)

15. n	GREG		G0	\$0,Fib
fn	IS	n	ST0	fn,fp,24
	GREG	@	LD0	n,fp,16
Fib	CMP	\$1,n,2	SUB	n,n,2
	PBN	\$1,1F	G0	\$0,Fib
	ST0	fp,sp,0	LD0	\$0,fp,24
	SET	fp,sp	ADDU	fn,fn,\$0
	INCL	sp,8*4	LD0	\$0,fp,8
	ST0	\$0,fp,8	SET	sp,fp
	ST0	n,fp,16	LD0	fp,sp,0
	SUB	n,n,1	1H G0	\$0,\$0,0

Вызывающая последовательность имеет вид SET n,n; G0 \$0,Fib; результат возвращается в глобальном регистре fn. Время выполнения равно $(8F_{n+1} - 8)\mu + (32F_{n+1} - 23)v$, поэтому отношение между этой версией и подпрограммой на основе стека регистров из упражнения 13 приблизительно равно $(8\mu/v + 32)/19$. (Хотя в упражнении 14 указывается, что не следует рекурсивно вычислять числа Фибоначчи, этот анализ демонстрирует преимущество схемы на основе стека регистров. Даже если быть щедрым и предположить, что $\mu = v$, то затраты вдвое больше в этом примере. Аналогичное поведение наблюдается в других подпрограммах, но анализ для Fib особенно прост.)

В особом случае с подпрограммой Fib можно обойтись без указателя стекового фрейма, поскольку fp всегда находится на фиксированном расстоянии от sp. Подпрограмма на основе стековой памяти выполняется приблизительно на $(6\mu/v + 29)/19$ медленнее, чем подпрограмма на основе стека регистров. Она лучше, чем версия с обычными стеками, но все еще не идеальна.

16. Это идеальная ситуация для подпрограммы с двумя выходами. Предположим для удобства, что B и C не возвращают никаких результатов и сохраняют rJ в \$1 (поскольку они не являются концевыми подпрограммами). Тогда можно поступить следующим образом: A вызывает B, как обычно, с помощью PUSHJ \$R,B. B вызывает C с помощью PUSHJ \$R,C; PUT rJ,\$1; POP 0,0 (возможно, с другим значением R, чем в подпрограмме A). C вызывает себя с помощью PUSHJ \$R,C; PUT rJ,\$1; POP 0,0 (возможно, с другим значением R, чем в подпрограмме B). C переходит в A с помощью PUT rJ,\$1; POP 0,0. C выходит с помощью PUT rJ,\$1; POP 0,2.

Очевидно, что можно расширить эту идею вариантами с возвращением значений и произвольного адреса перехода, как части возвращаемой информации. Аналогичные схемы применяются в (15) для протокола стековой памяти, ориентированного на инструкцию G0.

РАЗДЕЛ 1.4.2'

1. Если одна сопрограмма вызывает другую только один раз, то это всего лишь подпрограмма. Потому для примера нужно приложение, в котором каждая сопрограмма

вызывает другую, по крайней мере в двух разных местах. Даже в таком случае часто проще создать переключатель или использовать какое-то свойство данных, чтобы можно было в фиксированном месте сопрогаммы создать условный переход в одно из двух нужных мест. Однако, для этого опять достаточно было бы всего лишь подпрограммы. Сопрограммы становятся соответственно более полезными по мере возрастания количества ссылок между ними.

2. Первый символ, найденный In, будет утрачен.

3. Это уловка языка MMIXAL для размещения в OutBuf 15 тетрабайт TETRA ' ', за которым следует TETRA #a, далее нуль. TETRA ' ' эквивалентно BYTE 0,0,0,' '. Выходной буфер получает строку из 16 трехсимвольных групп, разделенных пробелами.

4. Если создать код

```
rR_A GREG
rR_B GREG
GREG @
A GET rR_B,rR
  PUT rR,rR_A
  GO t,a,0
B GET rR_A,rR
  PUT rR,rR_B
  GO t,b,0
```

то A может вызывать B с помощью "GO a,B" и B может вызывать A с помощью "GO b,A".

5. Если создать код

```
a GREG
b GREG
GREG @
A GET b,rJ
  PUT rJ,a
  POP 0
B GET a,rJ
  PUT rJ,b
  POP 0
```

то A может вызывать B с помощью "PUSHJ \$255,B" и B может вызывать A с помощью "PUSHJ \$255,A". Обратите внимание на сходство между эти ответом и ответом к предыдущему упражнению. Сопрограммам не следует использовать стек регистров для каких-либо других целей, кроме тех, которые представлены в следующем упражнении.

6. Допустим, что сопрограмма A содержит нечто в стеке регистров в момент вызова сопрогаммы B. Тогда сопрограмма B перед возвратом управления сопрограмме A должна вернуть стек в прежнее состояние, несмотря на то, что до этого B может произвольное количество раз протолкнуть и вытолкнуть элементы стека.

Сопрограммы могут, конечно, гораздо более сложным образом использовать стек регистров для своих целей. В таких случаях в MMIX рекомендуется использовать команды SAVE и UNSAVE для сохранения и восстановления контекста каждой сопрогаммы.

РАЗДЕЛ 1.4.3'

1. (a) SRU x,y,z; BYTE 0,1,0,#29. (b) PBP x,PBTaken+@-0; BYTE 0,3,0,#50.
(c) MUX x,y,z; BYTE 0,1,rM,#29. (d) ADDU x,x,z; BYTE 0,1,0,#30.

2. Время выполнения MemFind равно $9v + (2\mu + 8v)C + (3\mu + 6v)U + (2\mu + 11v)A$, где C является количеством сравнений в строке 042, $U = [\text{key} \neq \text{curkey}]$ и $A = [\text{требуется новый}]$

узел]. Время выполнения **GetReg** равно $\mu + 6v + 6vL$, где $L = [\$k$ является локальным]. Если предположить, что $C = U = A = L = 0$ при каждом вызове, то время симулирования можно систематизировать следующим образом:

	(a)	(b)	(c)
выборка (строки 105–115)	$\mu + 17v$	$\mu + 17v$	$\mu + 17v$
распаковка (строки 141–153)	$\mu + 12v$	$\mu + 12v$	$\mu + 12v$
связывание (строки 154–164)	$2v$	$2v$	$9v$
установка X (строки 174–182)	$7v$	$\mu + 17v$	$\mu + 17v$
установка Z (строки 183–197)	$\mu + 13v$	$6v$	$6v$
установка Y (строки 198–207)	$\mu + 13v$	$\mu + 13v$	$6v$
назначение (строки 208–231)	$8v$	$23v$	$6v$
продолжение (строки 232–242)	$14v$	$\mu + 14v$	$16v - \pi$
постобработка (строки 243–539)	$\mu + 10v$	$11v$	$11v - 4\pi$
обновление (строки 540–548)	$5v$	$5v$	$5v$
всего	$5\mu + 101v$	$5\mu + 120v$	$3\mu + 105v - 5\pi$

К этому времени нужно добавить $6v$ для каждого использования локального регистра в качестве источника, плюс накладные расходы, если **MemFind** не сможет найти нужный фрагмент. В случае (b), **MemFind** *должен* перейти к строке 231 и снова к строке 111 при выборке соответствующей инструкции. (Лучше всего было бы иметь две процедуры **MemFind**: одну для данных и одну для инструкций.) Наиболее оптимистичная оценка всех затрат (b) получается при допущении $C = A = 2$ и общее время выполнения равно $13\mu + 158v$. (При длительных испытаниях симулятора, симулирующего самого себя, эмпирические средние значения для одного вызова **MemFind** равнялись $C \approx .29$, $U \approx .00001$, $A \approx .16$.)

3. В строке 097 имеем $\beta = \gamma$ и $L > 0$. Тогда $\alpha = \gamma$ *может* возрасти, но только в экстремальных обстоятельствах, когда $L = 256$ (см. строку 268 и упражнение 11). К счастью, L вскоре становится равным 0 в этом случае.

4. Проблем не будет до тех пор, пока узел не вторгнется в сегмент пула, который начинается по адресу $\#4000000000000000$. Тогда остальная часть содержимого командной строки может конфликтовать с предположением программы, что вновь выделенный узел инициализирован нулем. Но сегмент данных способен вместить $\lfloor (2^{61} - 2^{12} - 2^4) / (2^{12} + 24) \rfloor = 559670633304293$ узлов так, что довольно скоро эта “особенность” даст о себе знать.

5. Строка 218 вызывает **StackRoom**, которая вызывает **StackStore**, которая вызывает **MemFind** и далее настолько глубоко, насколько получится. Строка 218 проталкивает 3 регистра вниз, подпрограмма **StackRoom** — только 2 (поскольку $rL = 1$ в строке 097), а подпрограмма **StackStore** — 3. Значение rL в строке 032 равно 2 (хотя rL увеличивается до 5 в строке 034). Следовательно, в самом худшем случае стек регистров содержит $3 + 2 + 3 + 2 = 10$ невытолкнутых элементов.

Программа завершает работу вскоре после условного перехода к **Error**. Даже если она продолжает работу, дополнительный мусор в нижней части стека не причинит никакого вреда — потому можно было игнорировать эту ситуацию. Однако можно было бы очистить стек, предоставляя дополнительный выход, как в упражнении 1.4.1'–16. Более простой способ опустошения всего стека заключается в повторном выталкивании до тех пор, пока rO не будет равен исходному значению, **Stack_Segment**.

6. 247 Div DIV x,y,z Поделить y на z с учетом знака.
 248 JMP 1F
 249 DivU PUT rD,x Поместить симулированный rD в реальный rD.
 250 DIVU x,y,z Поделить y на z без учета знака.

```

251 1H GET  t,rR
252      STO  t,g,8*rR  g[rR] ← остаток от деления.
253      JMP  XDone     Завершить, сохраняя x.  █

```

7. (Перечисленные ниже инструкции нужно вставить между строкой 309 и таблицей Info, причем вместе с ответами из нескольких следующих аналогичных упражнений.)

```

Cswap LDOU  z,g,8*rP
      LDOU  y,res,0
      CMPU  t,y,z
      BNZ   t,1F      Условный переход, если M8[A] ≠ g[rP].
      STOU  x,res,0   В противном случае установить M8[A] ← $X.
      JMP   2F
1H     STOU  y,g,8*rP  Установить g[rP] ← M8[A].
2H     ZSZ   x,t,1     x ← результат проверки равенства.
      JMP   XDone     Завершить, сохраняя x.  █

```

8. Здесь нужно сохранять симулированные регистры, которые хранятся в фактических регистрах. (Этот подход лучше, чем 32-вариантный условный переход для проверки используемого регистра, и лучше, чем альтернативный вариант сохранения регистров при каждом их изменении.)

```

Get  CMPU  t,yz,32
      BNN   t>Error    Убедиться, что YZ < 32.
      STOU  ii,g,8*rI   Поместить правильное значение в g[rI].
      STOU  cc,g,8*rC   Поместить правильное значение в g[rC].
      STOU  oo,g,8*r0   Поместить правильное значение в g[r0].
      STOU  ss,g,8*rS   Поместить правильное значение в g[rS].
      STOU  uu,g,8*rU   Поместить правильное значение в g[rU].
      STOU  aa,g,8*rA   Поместить правильное значение в g[rA].
      SR    t,ll,3
      STOU  t,g,8*rL   Поместить правильное значение в g[rL].
      SR    t,gg,3
      STOU  t,g,8*rG   Поместить правильное значение в g[rG].
      SLU   t,zz,3
      LDOU  x,g,t       Установить x ← g[Z].
      JMP   XDone     Завершить, сохраняя x.  █

```

```

9. Put  BNZ   yy>Error  Убедиться, что Y = 0.
      CMPU  t,xx,32
      BNN   t>Error    Убедиться, что X < 32.
      CMPU  t,xx,rC
      BN    t,PutOK    Условный переход, если X < 8.
      CMPU  t,xx,rF
      BN    t,1F       Условный переход, если X < 22.
PutOK  STOU  z,g,xxx    Установить g[X] ← z.
      JMP   Update     Завершить команду.
1H     CMPU  t,xx,rG
      BN    t>Error    Условный переход, если X < 19.
      SUB   t,xx,rL
      PBP   t,PutA     Условный переход, если X = rA.
      BN    t,PutG     Условный переход, если X = rG.
PutL   SLU   z,z,3     В противном случае X = rL.
      CMPU  t,z,ll

```

	CSN	11,t,z	Установить $rL \leftarrow \min(z, rL)$.
	JMP	Update	Завершить команду.
OH GREG #40000			
PutA	CMPU	t,z,0B	
	BNN	t,Error	Убедиться, что $z \leq \#3ffff$.
	SET	aa,z	Установить $rA \leftarrow z$.
	JMP	Update	Завершить команду.
PutG	SRU	t,z,8	
	BNZ	t,Error	Убедиться, что $z < 256$.
	CMPU	t,z,32	
	BN	t,Error	Убедиться, что $z \geq 32$.
	SLU	z,z,3	
	CMPU	t,z,11	
	BN	t,Error	Убедиться, что $z \geq rL$.
	JMP	2F	
1H	SUBU	gg,gg,8	$G \leftarrow G - 1$. (\$G становится глобальным.)
	STCO	0,g,gg	$g[G] \leftarrow 0$. (Сравнить со строкой 216.)
2H	CMPU	t,z,gg	
	PBN	t,1B	Условный переход, если $G < z$.
	SET	gg,z	Установить $rG \leftarrow z$.
	JMP	Update	Завершить команду. ■

В этом случае лучше использовать условный переход к командам PutOK, PutA, PutG, PutL или Error, чем 32-вариантную таблицу переключения.

10. Pop	SUBU	oo,oo,8	
	BZ	xx,1F	Условный переход, если $X = 0$.
	CMPU	t,11,xxx	
	BN	t,1F	Условный переход, если $X > L$.
	ADDU	t,xxx,oo	
	AND	t,t,1ring_mask	
	LDU	y,1,t	$y \leftarrow$ возвращаемый результат.
1H	CMPU	t,oo,ss	
	PBNN	t,1F	Условный переход, если не выполняется $\alpha = \gamma$.
	PUSHJ	0,StackLoad	
1H	AND	t,oo,1ring_mask	
	LDU	z,1,t	$z \leftarrow$ количество дополнительных выталкиваемых регистров.
	AND	z,z,#ff	Убедиться, что $z \leq 255$ (на случай ошибки).
	SLU	z,z,3	
1H	SUBU	t,oo,ss	
	CMPU	t,t,z	
	PBNN	t,1F	Условный переход, если только z регистров нет в кольце.
	PUSHJ	0,StackLoad	(См. замечание ниже.)
	JMP	1B	Повторять, пока не будут загружены все необходимые регистры.
1H	ADDU	11,11,8	
	CMPU	t,xxx,11	
	CSN	11,t,xxx	Установить $L \leftarrow \min(X, L + 1)$.
	ADDU	11,11,z	Затем увеличить L на z .
	CMPU	t,gg,11	

	CSN	11,t,gg	Установить $L \leftarrow \min(L, G)$.
	CMPU	t,z,11	
	BNN	t,1F	Условный переход, если возвращаемый результат нужно отбросить.
	AND	t,oo,lring_mask	
	STOU	y,1,t	В противном случае установить $l[(\alpha - 1) \bmod \rho] \leftarrow y$.
1H	LDQU	y,g,8*rJ	
	SUBU	oo,oo,z	Уменьшить α на $1 + z$.
	4ADDU	inst_ptr,yz,y	Установить $inst_ptr \leftarrow g[rJ] + 4YZ$.
	JMP	Update	Завершить команду. ■

Здесь удобно уменьшить oo в два этапа: сначала на 8 и потом на 8 умноженное на z . В общем программа немного сложна, но в большинстве случаев требуется выполнить сравнительно мало вычислений. Если $\beta = \gamma$ при втором вызове `StackLoad`, то неявно уменьшаем β на 1 (таким образом отбрасывая самый верхний элемент стека регистров). Этот элемент не нужен, если он не является возвращаемым значением, но оно уже помещено в y .

11. Save	BNZ	yz>Error	Убедиться, что $YZ = 0$.
	CMPU	t,xxx,gg	
	BN	t>Error	Убедиться, что $\$X$ является глобальным.
	ADDU	t,oo,11	
	AND	t,t,lring_mask	
	SRU	y,11,3	
	STOU	y,1,t	Установить $\$L \leftarrow L$, считая $\$L$ локальным.
	INCL	11,8	
	PUSHJ	0,StackRoom	Убедиться, что $\beta \neq \gamma$.
	ADDU	oo,oo,11	
	SET	11,0	Протолкнуть вниз все локальные регистры и установить $rL \leftarrow 0$.
1H	PUSHJ	0,StackStore	
	CMPU	t,ss,oo	
	PBNZ	t,1B	Сохранить в памяти все проталкиваемые вниз регистры.
	SUBU	y,gg,8	Установить $k \leftarrow G - 1$. (Здесь $y \equiv 8k$.)
4H	ADDU	y,y,8	Увеличить k на 1.
1H	SET	arg,ss	
	PUSHJ	res,MemFind	
	CMPU	t,y,8*(rZ+1)	
	LDQU	z,g,y	Установить $z \leftarrow g[k]$.
	PBNZ	t,2F	
	SLU	z,gg,56-3	
	ADDU	z,z,aa	If $k = rZ + 1$, set $z \leftarrow 2^{56}rG + rA$.
2H	STOU	z,res,0	Сохранить z в $M_8[rS]$.
	INCL	ss,8	Увеличить rS на 8.
	BNZ	t,1F	Условный переход, если сохраняются только rG и rA .
	CMPU	t,y,c255	
	BZ	t,2F	Условный переход, если сохраняется только $\$255$.
	CMPU	t,y,8*rR	
	PBNZ	t,4B	Условный переход, если не сохраняется только rR .
	SET	y,8*rP	Установить $k \leftarrow rP$.
	JMP	1B	

2H	SET	y,8*rB	Установить $k \leftarrow rB$.
	JMP	1B	
1H	SET	oo,ss	$rO \leftarrow rS$.
	SUBU	x,oo,8	$x \leftarrow rO - 8$.
	JMP	XDone	Завершить, сохраняя x. ■

(Сохраняются специальные регистры с кодами 0–6 и 23–27, а также (rG, rA).)

12.	Unsave	BNZ	xx,Error	Убедиться, что $X = 0$.
		BNZ	yy,Error	Убедиться, что $Y = 0$.
		ANDNL	z,#7	Убедиться, что z кратно 8.
		ADDU	ss,z,8	Установить $rS \leftarrow z + 8$.
		SET	y,8*(rZ+2)	Установить $k \leftarrow rZ + 2$. ($y \equiv 8k$)
1H		SUBU	y,y,8	Уменьшить k на 1.
4H		SUBU	ss,ss,8	Уменьшить rS на 8.
		SET	arg,ss	
		PUSHJ	res,MemFind	
		LDOU	x,res,0	Установить $x \leftarrow M_8[rS]$.
		CMPU	t,y,8*(rZ+1)	
		PBNZ	t,2F	
		SRU	gg,x,56-3	Если $k = rZ + 1$, инициализировать rG и rA.
		SLU	aa,x,64-18	
		SRU	aa,aa,64-18	
		JMP	1B	
2H		STOU	x,g,y	В противном случае установить $g[k] \leftarrow x$.
3H		CMPU	t,y,8*rP	
		CSZ	y,t,8*(rR+1)	Если $k = rP$, то установить $k \leftarrow rR + 1$.
		CSZ	y,y,c256	Если $k = rB$, то установить $k \leftarrow 256$.
		CMPU	t,y,gg	
		PBNZ	t,1B	Повторять цикл, пока не соблюдается условие $k = G$.
		PUSHJ	0,StackLoad	
		AND	t,ss,lring_mask	
		LDOU	x,l,t	$x \leftarrow$ количество локальных регистров.
		AND	x,x,#ff	Убедиться, что $x \leq 255$ (на случай ошибки).
		BZ	x,1F	
		SET	y,x	Теперь загрузить x локальных регистров в кольцо.
2H		PUSHJ	0,StackLoad	
		SUBU	y,y,1	
		PBNZ	y,2B	
		SLU	x,x,3	
1H		SET	ll,x	
		CMPU	t,gg,x	Установить $rL \leftarrow \min(x, rG)$.
		CSN	ll,t,gg	Установить $rO \leftarrow rS$.
		SET	oo,ss	
		PBNZ	uu,Update	Условный переход, если это не первый раз.
		BZ	resuming,Update	Условный переход, если первой командой является UNSAVE.
		JMP	AllDone	В противном случае очистить resuming и завершить. ■

*Прямой ответ также хорош,
как дружеский поцелуй.*

13.	517	SET	xx,0	
	518	SLU	t,t,55	Цикл для поиска старшего бита обхода.
	519	2H INCL	xx,1	
	520	SLU	t,t,1	
	521	PBNN	t,2B	
	522	SET	t,#100	Теперь xx = индекс бита обхода.
	523	SRU	t,t,xx	$t \leftarrow$ соответствующий бит события.
	524	ANDN	exc,exc,t	Удалить t из exc.
	525	TakeTrip STOU	inst_ptr,g,8*rW	$g[rW] \leftarrow inst_ptr$.
	526	SLU	inst_ptr,xx,4	$inst_ptr \leftarrow xx \ll 4$.
	527	INCH	inst,#8000	
	528	STOU	inst,g,8*rX	$g[rX] \leftarrow inst + 2^{63}$.
	529	AND	t,f,Mem_bit	
	530	PBZ	t,1F	Условный переход, если op не осуществляет доступ к памяти.
	531	ADDU	y,y,z	В противном случае установить $y \leftarrow (y + z) \bmod 2^{64}$,
	532	SET	z,x	$z \leftarrow x$.
	533	1H STOU	y,g,8*rY	$g[rY] \leftarrow y$.
	534	STOU	z,g,8*rZ	$g[rZ] \leftarrow z$.
	535	LDOU	t,g,c255	
	536	STOU	t,g,8*rB	$g[rB] \leftarrow g[255]$.
	537	LDOU	t,g,8*rJ	
	538	STOU	t,g,c255	$g[255] \leftarrow g[rJ]$. ■
14.	Resume	SLU	t,inst,40	
		BNZ	t,Error	Убедиться, что XYZ = 0.
		LDOU	inst_ptr,g,8*rW	$inst_ptr \leftarrow g[rW]$.
		LDOU	x,g,8*rX	
		BN	x,Update	Завершить команду, если rX отрицательно.
		SRU	xx,x,56	В противном случае пусть xx будет ропкодом.
		SUBU	t,xx,2	
		BNN	t,1F	Условный переход, если этот ропкод ≥ 2 .
		PBZ	xx,2F	Условный переход, если этот ропкод равен 0.
		SRU	y,x,28	В противном случае этот ропкод равен 1:
		AND	y,y,#f	$y \leftarrow k$, первая половина опкода.
		SET	z,1	
		SLU	z,z,y	$z \leftarrow 2^k$.
		ANDNL	z,#70cf	Обнулить допустимые значения z.
		BNZ	z,Error	Убедиться, что опкод "нормален".
	1H	BP	t,Error	Убедиться, что опкод ≤ 2 .
		SRU	t,x,13	
		AND	t,t,c255	
		CMPU	y,t,11	
		BN	y,2F	Условный переход, если \$X является локальным.
		CMPU	y,t,gg	
		BN	y,Error	В противном случае убедиться, что \$X является глобальным.
	2H	MOR	t,x,#8	
		CMPU	t,t,#F9	Убедиться, что опкод не является RESUME.

```

        BZ      t,Error
        NEG     resuming,xx
        CSNN    resuming,resuming,1   Установить resuming как указано.
        JMP     Update                 Завершить команду. █

166  LDOU     y,g,8*rY                y ← g[rY].
167  LDOU     z,g,8*rZ                z ← g[rZ].
168  BOD      resuming,Install_Y      Условный переход, если ропкод равен 1.
169  OH GREG  #C1<<56+(x-$0)<<48+(z-$0)<<40+1<<16+X_is_dest_bit
170  SET      f,OB                    В противном случае заменить f на инструкцию ORI.
171  LDOU     exc,g,8*rX
172  MOR      exc,exc,#20              exc ← третий слева байт rX.
173  JMP      XDest                   Продолжить как для ORI. █

```

15. Следует иметь в виду, что строка вывода может быть расщеплена между двумя или более участками симулированной памяти. Одно решение заключается в том, чтобы выводить по восемь байт с помощью Fwrite до последнего октабайта строки. Однако этот способ усложняется тем, что строка может начинаться в середине октабайта. Наоборот, можно было бы использовать Fwrite для вывода только по одному байту, но это было бы чрезвычайно медленно. Гораздо лучше использовать следующий метод:

```

SimFputs  SET      xx,0                (xx — это количество записываемых байтов)
          SET      z,t                Установить z ← виртуальный адрес строки.
1H        SET      arg,z
          PUSHJ    res,MemFind
          SET      t,res              Установить t ← фактический адрес строки.
          GO       $0,DoInst          (См. ниже.)
          BN       t,TrapDone         Если произошла ошибка, передать ее пользователю.
          BZ       t,1F               Условный переход, если строка пуста.
          ADD      xx,xx,t            В противном случае подсчитать количество байт.
          ADDU     z,z,t              Найти адрес после выводимой строки.
          AND      t,z,Mem:mask
          BZ       t,1B               Продолжить, если строка заканчивается на
                                     границе участка памяти.
1H        SET      t,xx              t ← количество успешно выведенных байт.
          JMP      TrapDone          Завершить операцию. █

```

Здесь DoInst — это маленькая подпрограмма, которая вставляет inst в поток инструкций. Вот дополнительные входы в нее, которые могут пригодиться при выполнении следующих упражнений:

```

          GREG     @                  Базовый адрес
:SimInst  LDA      t,I0Args           Выполнить DoInst для I0Args и вернуться.
          JMP      DoInst
SimFinish LDA      t,I0Args           Выполнить DoInst для I0Args и завершить.
SimFclose GETA     $0,TrapDone        Выполнить DoInst и завершить.
:DoInst  PUT      rW,$0              Поместить обратный адрес в rW.
          PUT      rX,inst            Поместить inst в rX.
          RESUME   0                  И выполнить ее. █

```

16. Здесь снова нужно учитывать границы участков памяти (см. ответ к предыдущему упражнению), но побайтовый метод вывода более приемлем, поскольку файлы обычно имеют короткие имена.

```

SimFopen  PUSHJ    0,GetArgs          (См. ниже.)
          ADDU     xx,Mem:alloc,Mem:nodeSize

```

	STOU	xx,IOArgs	
	SET	x,xx	(Будем копировать имя файла в это открытое пространство.)
1H	SET	arg,z	
	PUSHJ	res,MemFind	
	LDBU	t,res,0	
	STBU	t,x,0	Копировать байт M[z].
	INCL	x,1	
	INCL	z,1	
	PBNZ	t,1B	Повторять, пока строка не закончится.
	GO	\$0,SimInst	Теперь открыть файл.
3H	STCO	0,x,0	Обнулить копируемую строку.
	CMPU	z,xx,x	
	SUB	x,x,8	
	PBN	z,3B	Повторять до полного окончания.
	JMP	TrapDone	Передать результат t пользователю. █

Здесь `GetArgs` — это подпрограмма, которая будет полезна при реализации других команд ввода-вывода. Она устанавливает `IOArgs` и вычисляет несколько других полезных результатов в глобальных регистрах.

:GetArgs	GET	\$0,rJ	Сохранить обратный адрес.
	SET	y,t	$y \leftarrow g[255]$.
	SET	arg,t	
	PUSHJ	res,MemFind	
	LDOU	z,res,0	$z \leftarrow$ виртуальный адрес первого аргумента.
	SET	arg,z	
	PUSHJ	res,MemFind	
	SET	x,res	$x \leftarrow$ внутренний адрес первого аргумента.
	STO	x,IOArgs	
	SET	xx,Mem:Chunk	
	AND	zz,x,Mem:mask	
	SUB	xx,xx,zz	$xx \leftarrow$ байты от x до конца участка памяти.
	ADDU	arg,y,8	
	PUSHJ	res,MemFind	
	LDOU	zz,res,0	$zz \leftarrow$ второй аргумент.
	STOU	zz,IOArgs+8	Преобразовать IOArgs во внутреннюю форму.
	PUT	rJ,\$0	Восстановить обратный адрес.
	POP	0	█

17. Это решение, в котором используются приведенные выше процедуры, годится и для `SimFwrite(!)`.

SimFread	PUSHJ	0,GetArgs	Настройка входных аргументов.
	SET	y,zz	$y \leftarrow$ количество считываемых байтов.
1H	CMP	t,xx,y	
	PBNN	t,SimFinish	Условный переход, если все удастся сделать в одном участке памяти.
	STO	xx,IOArgs+8	Увы, придется делать работу по частям.
	SUB	y,y,xx	
	GO	\$0,SimInst	
	BN	t,1F	Условный переход, если возникает ошибка.
	ADD	z,z,xx	

```

        SET      arg,z
        PUSHJ    res,MemFind
        STOU     res,I0Args    Сводится к предыдущей задаче.
        STO      y,I0Args+8
        ADD      xx,Mem:mask,1
        JMP      1B
1H      SUB      t,t,y          Вычислить правильное число отсутствующих байтов.
        JMP      TrapDone
SimFwrite IS SimFread ;SimFseek IS SimFclose ;SimFtell IS SimFclose █

```

(В программе предполагается, что если первая операция `Fread` была успешной, то ошибки чтения файла больше не возникают.) Аналогичные процедуры для `SimFgets`, `SimFgetws` и `SimFputws` можно найти в файле `sim.mms`, которые входят в состав демонстрационных файлов программного обеспечения `MMIXware`.

18. Указанные алгоритмы будут работать с любой программой `MMIX`, в которой количество локальных регистров, L , никогда не превышает $\rho - 1$, где ρ — это `lring_size`.

19. Во всех трех случаях предыдущей инструкцией является `INCL 11,8`, а значение сохраняется по адресу $1 + ((00 + 11) \wedge \text{lring_mask})$. Поэтому программу можно было бы немного сократить.

```

20. 560 1H GETA  t,OctaArgs
      561 TRAP  0,Fread,Infile  Введите  $\lambda$  в  $g[255]$ .
      562 BN   t,9F            Условный переход, если достигнут конец файла.
      563 LDOU loc,g,c255       $\text{loc} \leftarrow \lambda$ .
      564 2H GETA  t,OctaArgs
      565 TRAP  0,Fread,Infile  Ввести октабайт  $x$  в  $g[255]$ .
      566 LDOU  x,g,c255
      567 BN   t>Error         Условный переход, если неожиданно достигнут
                                конец файла.

      568 SET   arg,loc
      569 BZ    x,1B            Начать новую последовательность, если  $x = 0$ .
      570 PUSHJ res,MemFind
      571 STOU  x,res,0          В противном случае сохранить  $x$  in  $M_8[\text{loc}]$ .
      572 INCL  loc,8           Увеличить  $\text{loc}$  на 8.
      573 JMP   2B             Повторять до тех пор, пока встретится ноль.
      574 9H TRAP  0,Fclose,Infile  Закрыть входной файл.
      575 SUBU  loc,loc,8        Уменьшить  $\text{loc}$  на 8. █

```

Также поместить "`OctaArgs OCTA Global+8*255,8`" в какое-то удобное место.

21. Да, но до некоторой степени. Этот вопрос совсем не тривиален и представляет большой интерес.

Для количественного анализа допустим, что `sim.mms` является симулятором на языке `MMIXAL`, `sim.mmo` является соответствующим объектным файлом, созданным ассемблером. Пусть `Hello.mmo` является объектным файлом, соответствующим программе 1.3.2'Н. Тогда командная строка '`Hello`' в операционной системе `MMIX` приведет к выводу '`Hello, world`' и остановке выполнения программы спустя $\mu + 17\nu$, не считая времени, необходимого операционной системе для ее загрузки и выполнения операций ввода-вывода.

Пусть `Hello0.mmb` является двоичным файлом, который соответствует командной строке '`Hello`' в формате упражнения 20. (Этот файл имеет длину 176 байтов.) Тогда командная строка '`sim Hello0.mmb`' выведет '`Hello, world`', и программа остановится спустя $168\mu + 1699\nu$.

Пусть `Hello1.mmb` является двоичным файлом, который соответствует командной строке `'sim Hello0.mmb'`. (Этот файл имеет длину 5768 байтов.) Тогда командная строка `'sim Hello1.mmb'` выведет `'Hello, world'`, и программа остановится спустя $10549\mu + 169505\nu$.

Пусть `Hello2.mmb` является двоичным файлом, который соответствует командной строке `'sim Hello1.mmb'`. (Этот файл также имеет длину 5768 байтов.) Тогда командная строка `'sim Hello2.mmb'` выведет `'Hello, world'`, и программа остановится спустя $789739\mu + 15117686\nu$.

Пусть `Hello3.mmb` является двоичным файлом, который соответствует командной строке `'sim Hello2.mmb'`. (Этот файл также имеет длину 5768 байтов.) Тогда командная строка `'sim Hello3.mmb'` выведет `'Hello, world'`, и программа остановится спустя достаточно долгое время.

Теперь допустим, что `recurse.mmb` является двоичным файлом, который соответствует командной строке `'sim recurse.mmb'`. Тогда командная строка `'sim recurse.mmb'` запустит симулятор, который будет симулировать себя, симулирующего себя, симулирующего себя, ... и так до бесконечности. Файл с идентификатором `Infile` сначала открывается в момент времени $3\mu + 13\nu$, когда `recurse.mmb` начинает считываться симулятором на уровне 1. Этот файл закрывается в момент времени $1464\mu + 16438\nu$, когда завершается загрузка. А симулированный симулятор на уровне 2 открывает его в момент времени $1800\mu + 19689\nu$ и начинает загрузку `recurse.mmb` в симулированную симулированную память. Этот файл снова закрывается в момент времени $99650\mu + 1484347\nu$, затем снова открывается симулированным симулированным симулятором в момент времени $116999\mu + 1794455\nu$. На третьем уровне загрузка завершается в момент времени $6827574\mu + 131658624\nu$, а на четвертом уровне начинается в момент времени $8216888\mu + 159327275\nu$.

Но рекурсия не может продолжаться бесконечно. Действительно, симулирующий себя симулятор является системой с конечным числом состояний, а такая система не может создавать события `Open-Fclose` с экспоненциально растущими временами выполнения. В конечном итоге память переполнится (см. упражнение 4) и симуляция прекратится. Когда это произойдет? Точный момент определить трудно, но его можно оценить. Если симулятору уровня k нужно n_k участков памяти для загрузки симулятора уровня $(k+1)$, то n_{k+1} по меньшей мере равно $4 + \lceil (2^{12} + 16 + (2^{12} + 24)n_k)/2^{12} \rceil$, где $n_0 = 0$. Имеем $n_k = 6k$ для $k < 30$, но эта последовательность растет экспоненциально, например превышает 2^{61} при $k = 6066$. Таким образом, можно симулировать по крайней мере 100^{6065} инструкций до возникновения каких-либо проблем, если предположить, что каждый уровень симуляции вводит множитель 100 (см. упражнение 2).

22. Пары (x_k, y_k) можно сохранять в памяти после самой программы трассировки, которую следует располагать после всех инструкций в текстовом сегменте трассируемой программы. (Операционная система даст процедуре трассировки разрешение на изменение текстового сегмента.) Основная идея состоит в сканировании от текущего положения в трассируемой программе до следующего условного перехода, либо инструкции `GO`, либо инструкции `PUSH`, либо инструкции `POP`, либо инструкции `JMP`, либо инструкции `RESUME`, либо инструкции `TRIP` с последующей временной заменой этой инструкции в памяти на команду `TRIP`. Тетрабайты по адресам `#0`, `#10`, `#20`, ..., `#80` трассируемой программы изменяются таким образом, чтобы совершить переход к соответствующим адресам в процедуре трассировки. Затем все переносы управления трассируются, включая переносы вследствие арифметических прерываний. Исходные инструкции по этим адресам могут трассироваться с помощью `RESUME`, если они сами не являются командами `RESUME`.

ПРЕДМЕТНО-ИМЕННОЙ УКАЗАТЕЛЬ

Если какой-то элемент этого указателя указывает на страницу с упражнением, то также рекомендуется взглянуть на *ответ* к этому упражнению, Поскольку там может содержаться полезная информация, связанная с этим элементом указателя. Страница ответа указывается только в том случае, если в ответе есть ссылка на тему, не включенную в формулировку упражнения.

@ (at sign), 24, 46, 49, 96.

\$0, 41, 71.

\$1, 41, 71.

2ADDU (times 2 and add unsigned), 17.

4ADDU (times 4 and add unsigned), 17.

8ADDU (times 8 and add unsigned), 17.

16ADDU (times 16 and add unsigned), 17.

\$255, 45, 52–54, 68, 82, 130.

ACE, 78.

ADD, 16.

ADDU, 16.

Adobe Systems, 89.

Ahrens, Wilhelm Ernst Martin Georg, 60.

ALGOL, 88.

AMD 29000, 10.

AND, 19.

ANDN, 19.

ANDNH, 23.

ANDNL, 23.

ANDNMH, 23.

ANDNML, 23.

Algol W, 9.

Alpha 21164, 10.

ANSI: The American National Standards
Institute, 21.

ASCII, 8, 11, 36, 37, 42, 45, 48, 56.

Ball, Walter William Rouse, 60.

BDIF, 19, 36, 117.

BEV, 24.

Bienstock, Daniel, 119.

Big-endian convention, 12–15.

BinaryRead, 54.

BinaryReadWrite, 54.

BinaryWrite, 54.

BN, 24.

BNN, 24.

BNP, 24.

BNZ, 24.

BOD, 24.

Bourne, Charles Percy, 123.

BP, 24.

BSPEC, 75.

Buchholz, Werner, 109.

BYTE, 42, 51.

BZ, 24.

C, 9, 56.

C++, 9.

Chung, Fan Rong King (鍾金芳蓉), 121.

Clipper C300, 10.

CMP, 18.

CMPU, 18, 129.

Coxeter, Harold Scott Macdonald, 60.

CRAY I, 10.

CSEV, 18.

CSN, 18.

CSNN, 18.

CSNP, 18.

CSNZ, 18.

CSOD, 18.

CSP, 18.

CSWAP, 27, 107.

CSZ, 18.

Dershowitz, Nachum (נחום דרשוביץ), 127.

Dickens, Charles John Huffam, 7.

Dijkstra, Edsger Wijbe, 76.

DIV, 16, 35.

Division, 17.

DIVU, 16.

DVWIOUZX, 27, 37, 104, 107.

ESPEC, 75.

Evans, Arthur, Jr., 88.

EXPR, 50.

FADD, 21.

Fclose, 53, 54.

FCMP, 21, 113.

FCMPE, 22.

FDIV, 21.

FEQL, 22, 113.

FEQLE, 22.

Fgets, 53, 54.

Fgetws, 53, 54.

FINT, 21, 33.

FIX, 22.

FIXU, 22.

FL0T, 22.

FL0TU, 22, 112.

FMUL, 21.

Fopen, 52, 54, 107.

Ford, Donald Floyd, 123.

Fputs, 54, 107.

Fputws, 54.

Fread, 53–55, 107.

Fredman, Michael Lawrence, 119.

FREM, 21, 33, 56, 126.

Fseek, 54.

FSQRT, 21.

FSUB, 21.

Ftell, 54.

Fuchs, David Raymond, 38.

FUN, 22, 113.

- FUNE, 22.
 Fwrite, 53–55, 140.

 GET, 29, 107.
 GETA, 30, 115.
 GO, 24, 37, 65–71.
 GREG, 45–46, 50, 75.

 Halt, 54.
 Hardy, Godfrey Harold, 121.
 Hello, world, 141.
 Hitachi SuperH4, 10.

 IBM 601, 10.
 IBM 801, 10.
 IEC: The International Electrotechnical Commission, 12.
 IEEE: The Institute of Electrical and Electronics Engineers, 21.
 INCH, 23.
 INCL, 23.
 INCMH, 23.
 INCML, 23.
 Intel i960, 10.
 Internet, 9.
 IS, 40, 45, 50.
 ISO: The International Organization for Standardization, 11.
 Iverson, Kenneth Eugene, 20.
 I_BIT, 113.

 Java, 9, 56.
 JMP, 24.
 Kernighan, Brian Wilson, 33.
 KKB, 109.

 LDA, 15, 115.
 LDB, 14.
 LDBU, 15.
 LDHT, 15, 35, 112.
 LDO, 14.
 LDOU, 15.
 LDSF, 22.
 LDT, 14.
 LDTU, 15.
 LDUNC, 26.
 LDVTS, 27.
 LDW, 14.
 LDWU, 15.
 LOC, 40, 50.
 LOCAL, 75.

 Main, 41, 106.
 MemFind, 106, 132–133.
 METAPOST, 63.
 MIPS R4000, 10.
 MIX, 8.
 MMIX, 10–39.
 MixMaster, 8.
 MMB, 109.
 .mmB, 141.

 MMIX, 8.
 MMIX, группа добровольцев-создателей, 9.
 MMIXmasters, 63, 120.
 MMIXAL, 39–56, 74–75.
 .mmo, 41, 141.
 .mms, 41, 141.
 MOR, 20, 33, 37.
 Motorola 88000, 10.
 MUL, 16.
 MULU, 16, 35.
 MUX, 19.
 MXOR, 21, 33, 37.

 NaN — Not-a-Number (не-является-числом), 21.
 NAND, 19.
 NEG, 17.
 NEGU, 17.
 NNIX, 39, 41.
 NOR, 19.
 NXOR, 19.

 O’Beirne, Thomas Hay, 109, 126.
 OCTA, 51.
 ODIF, 19, 117.
 OP, 50.
 OR, 19.
 ORH, 23.
 ORL, 23.
 ORMH, 23.
 ORML, 23.
 ORN, 19.

 Pascal, 9.
 PBEV, 25.
 PBN, 25.
 PBNN, 25.
 PBNP, 25.
 PBNZ, 25.
 PBOD, 25.
 PBP, 25.
 PBZ, 25.
 Phi (ϕ), 17, 59.
 PL/360, 57.
 PL/MMIX, 57, 76.
 POP, 25, 65, 72, 87, 107.
 PostScript, 89.
 POWER 2, 10.
 Prediction register, 27.
 PREFIX, 74–75, 79, 92.
 PREGO, 26.
 PRELD, 26.
 PREST, 26.
 Probable branch, 100.
 Purdy, Gregor Neal, 109.
 PUSH, 87.
 PUSHGO, 25, 78, 100.
 PUSHJ, 25, 65, 71, 100.
 PUT, 29, 107.

- RA, 24.
- rA, 27, 38.
- Randell, Brian, 88.
- rB, 28.
- rBB, 28.
- rC, 29, 127.
- rD, 17.
- rE, 22.
- Register stack, 26, 100.
- Reingold, Edward Martin (ריינגולד, יצחק משה בן חיים), 127.
- RESUME, 28, 99, 107, 130, 142.
- Rewrites, 9.
- rF, 29.
- rG, 25, 71, 107.
- rH, 16, 38, 100, 109.
- rI, 29.
- RISC, 8, 34.
- RISC II, 10.
- rJ, 25, 73, 95, 95.
- rK, 28, 106.
- rL, 25, 38, 71, 94, 107, 113, 133.
- rM, 19.
- rN, 29.
- rO, 26, 94.
- ROUND_DOWN, 22.
- ROUND_NEAR, 22.
- ROUND_OFF, 22.
- ROUND_UP, 22.
- rP, 27.
- rQ, 28.
- rR, 16.
- rS, 26, 94.
- rT, 28, 106.
- rTT, 28, 106.
- rU, 29.
- Russell, Lawford John, 88.
- rV, 29, 106.
- rW, 28.
- rWW, 28.
- rX, 28.
- rXX, 28.
- rY, 28.
- rYY, 28.
- rZ, 28.
- rZZ, 28.
- $s(x)$, 14, 34.
- SADD, 19.
- SAVE, 26, 74, 107, 130, 132.
- Schäffer, Alejandro Alberto, 119.
- SET, 24, 115.
- SETH, 23.
- SETL, 23, 115.
- SETMH, 23, 112.
- SETML, 23.
- SFLOT, 22.
- SFLOTU, 22.
- Shor, Peter Williston, 119.
- Sikes, Bill, 7.
- SL, 17, 36.
- SLU, 17, 36.
- Sparc 64, 10.
- SR, 17, 36.
- SRU, 17, 36.
- STB, 15.
- STBU, 16.
- STCO, 16.
- STHT, 16, 35, 112.
- STO, 15.
- STOU, 16.
- Stretch computer, 109.
- StrongArm 110, 10.
- STSF, 22.
- STT, 15.
- STTU, 16.
- STUNC, 26.
- STW, 15.
- STWU, 16.
- SUB, 16.
- SUBU, 16.
- Suri, Subhash (सुभाष सूरी), 119.
- SWYM, 30.
- SYNC, 27, 101.
- SYNCD, 27.
- SYNCID, 26, 38.
- System/360, 57.
- TDIF, 19.
- TETRA, 51, 86.
- TEX, 78, 88–89.
- TextRead, 54.
- TextWrite, 54.
- TRAP, 28, 52, 101.
- TRIP, 28.
- $u(x)$, 14, 34.
- UCS: Universal Multiple-Octet Coded Character Set, 12.
- Unicode, 11, 37, 48, 56.
- UNIVAC I, 46.
- UNIX, 84, 130.
- UNSAVE, 26, 74, 105, 107, 132.
- UTF: UCS Transformation Format, 12.
- Van Wyk, 33.
- WDIF, 19.
- Wordsworth, William, 34.
- WYDE, 51.
- XOR, 19.
- ZSEV, 19.
- ZSN, 18.
- ZSNN, 18.
- ZSNP, 19.
- ZSNZ, 18.
- ZSOD, 18.
- ZSP, 18.
- ZSZ, 18.

- Адрес
абсолютный, 24.
базовый, 46, 50.
относительный, 24, 25, 30, 40, 97, 114.
- Айверсон (Iverson), 20.
- Алгоритм
компилятора, 88.
многопроходный, 84–86.
нахождения максимума, 64–69.
поиска максимума+, 39.
- Анекдот, 86.
- Аргумент, 66.
командной строки, 41.
- Арифметика
двоичного дополнения, 12.
десятичная, 8.
чисел с плавающей запятой, 56, 57, 104.
- Арифметические операторы MMIX, 16.
- Ассемблер, 40.
- Ассоциативность, 20.
- Атомарная, 27.
- Байт, 11, 34
обратный порядок, 132.
- Беззнаковые целые числа, 12, 14.
- Библиотека, 64, 74, 75, 106.
- Бит, 10.
обращение, 37, 112.
разность, 36.
разрешающий, 27, 99.
события, 27, 99.
флаговый, 97.
- Битовая разность, 23.
- Битовое И-НЕ, 23.
- Битовое ИЛИ, 23.
- Битовый вектор, 19.
- Вайд, 12.
- Вайд-константа, 23.
- Ввод/вывод, 42, 52–54.
- Вебстерский словарь международного
английского языка, 3-е издание (Webster's
Third New International Dictionary of
the English Language), 7.
- Вектор, 19.
- Ветвление, условное, 25, 32, 37.
- Викториус Аквитанский (Victorius of
Aquitania), 126.
- Вирт, Никлас (Wirth, Niklaus), 57, 76.
- Виртуальная трансляция адреса, 27.
- Время выполнения, 30–33.
- Вход, 139.
- Выборка, предварительная, 26.
- Вывод, стандартное устройство, 42.
- Выделение, связанное, 91.
- Выравнивание, 50, 55.
- Выражение, 49.
первичное, 49.
элементарное, 49.
- Выход, 131.
из программы, 28.
многократный, 131.
- Вычисление
на основе таблицы, 58.
с табличным управлением, 102.
с фиксированной точкой, 57.
- Вычитание, 16, 21, 35.
насыщающее, 20.
- Гигабайт, 109.
- Гоув, П. Б. (Gove, P. B.), 7.
- Графика, 20.
- Далл, Б. К. (Dull, B. C.), 36.
- Даллос Дж.
(Dallos J.), 112.
- Данные упакованные, 96, 102.
- Двоеточие, 95.
- Двоичная система исчисления, 12.
- Двоичное число, с плавающей точкой, 21.
- Двунаправленного, 56.
- Деление, 16, 21, 35, 62, 107.
замена умножением, 126.
на малые константы, 36.
- Дескриптор, 52.
- Джакет, П. (Jacquet P.), 120.
- Диккенс, Ч. (Dickens, Charles John Huffam), 7.
- Дополнение, 7, 19.
- Достижимость, 63.
- Дробная часть, 21.
- Единицы измерений, 109.
- Заблуждение, 110.
- Завершение, 28.
- Загрузчик, 47.
- Задача, Джозефуса, 60.
- Зекстибайт, 109.
- Знак, 21.
- Знаковые целые числа, 12, 14.
- Значение
абсолютное, 36, 37.
нечисловое, 113.
- Значения пикселей, 20.
- Ибн-Эль-Хайтам (Ibn al-Haytham, Abū
'Alī al-Ḥasan = Alhazen, بن الهيثم
(أبو علي الحسن), 60.
- Иванович, Владимир (Ivanović, Vladimir), 9.
- Ингаллс, Д. (Ingalls, D.), 125.
- Инициализация, 84, 106.
- Инструкция MMIX, 13–39.
символьная, 39.
- Инструкция
в численном виде, 38.
привилегированная, 90.
- Интерпретатор, 87–89.
трассировки, 108.
- Исключительная ситуация, 27.
антипереполнения, 104.
арифметическая, 104.
неточности, 104.

- Йодер, Майкл (Yoder, Michael), 110.
Йоссариан, 11.
- Кавычка, двойная, 42, 48, 55, 86, 116.
Календарь, 61.
Канал, 84.
Квадрат, магический, 59.
Квик, Джон Х. (Quick, John H.), 55.
Килобайт, 34, 109.
 большой (large kilobyte), 109.
Клавиус, Кристофер (Clavius, Christopher), 61.
Кнут Д. Э. (Knuth, Donald Ervin, 高德纳),
 2-4, 9, 57, 78, 88.
Код, самоизменяющийся, 8, 38.
Кольцо, локальных регистров, 91, 93, 107.
Комментарии, 40.
Компилятор, 75.
Компьютер
 виртуальный, 87.
 суперскалярный, 123.
Конвейер, 59, 84.
 останов, 123.
Конвейеризация, 31, 114.
Конвейерная обработка, 90.
Конвэй, М. Э. (Conway, M. E.), 46.
Конец программы, 28.
Константа
 десятичная, 48.
 малая, 22.
 немедленная, 22, 23, 29.
 символьная, 48.
 строковая, 42, 48.
 шестнадцатеричная, 48.
Контроль деления, 16.
Копирование, строк, 59.
Корень, 21.
Коши О. (Cauchy A.), 121.
Криптоанализ, 59.
Кэш-память, 26, 32, 121, 123.
Кэширование, 114.
- Лилиус, Алоизиус (Lilius, Aloysius), 61.
- Максимум, 36.
Малые константы, 17.
Марри, Джеймс Аугустус Генри (Murray, James Augustus Henry), 7.
Массив
 последовательный, 58.
 развёртывание по строкам, 58.
Матрица, 122.
 логическая, 20.
 седловая точка, 58.
Машинный язык, 10.
Мегабайт, 34, 109.
 большой (large megabyte), 109.
Мемс, 32.
Мета-симулятор, 32, 59, 90.
Метка, 50, 104.
Минимум, 36.
Минус ноль, 22.
- Множество
 объединение, 36.
 пересечение, 36.
 разность, 36.
- Наименьшее целое число, превосходящее
 заданное, 22.
Немедленная вайд-константа, 23.
Нибл, 34.
Нип, 109.
- Обозначение
 $b(x)$, 20.
 $f(x)$, 21.
 $m(x)$, 20.
 $t(x)$, 20.
 $v(x)$, 19.
 $\bar{v}(x)$, 19.
 $w(x)$, 20.
 μ , 32.
 ν , 32.
int, 22.
 \div , 20.
 \gg , 18.
 \ll , 18.
 \wedge , 19.
 \vee , 19.
 \oplus , 19.
gem, 22.
дополнения до двух, 35.
- Обработчик, 79.
 обхода, 104.
 перескоков, 27.
 прерываний, 27.
- Обращение, 21, 37, 112.
 байта, 21.
- Округление, 59, 60.
- Оксфордский словарь английского языка (The Oxford English Dictionary), 7.
- Октабайт, 12.
- Операнд, 13, 98.
- Оператор MMIX
 битовый, 19, 23.
 ветвления, 24.
 загрузки MMIX, 14.
 нужно или установить, 18.
 нелокальный goto, 79.
 перехода, 24.
 преобразования, 22.
 с плавающей точкой, 21.
 сдвига, 17.
 сильный бинарный, 49.
 системный, 26.
 слабый бинарный, 49.
 сохранения MMIX, 15.
 сравнения, 18, 36, 129.
 стека, 25.
 условного перехода, 24.
 условный, 18, 37.

Операция

- атомарная, 27.
- битовая, 36.
- ввода-вывода, 28.
- коммутативная, 111.
- со строками, 36.
- с целыми числами, переполнение, 27.
- с числами с плавающей точкой
 - деление на нуль, 27.
 - недействительная, 27.
 - неточность, 27.
 - переполнение, 27.
 - потеря значащих разрядов, 27.
- с числами с фиксированной точкой, переполнение, 27.
- холостая, 38.
- Опкод, 13, 29.
 - диаграмма, 30.
 - допустимый, 57.
 - правый, 28.
 - привилегированный, 57.
 - холостой, 30.
- Оптимизация, цикла, 59.
- Останов, конвейера, 123.
- Остановка, 28.
- Остаток от деления, 21, 62.
- Отладка, 77–78, 87, 106.
- Отрицание, 17, 35.
- Отступ, стека, 94.
- Ошибка
 - нейтрализация, 106.
 - чтения страницы, 130.
- Память, 12.
 - иерархия, 85.
 - кэш, 121, 123.
 - стековая, 70, 131.
- Параметры, 66.
- Пасха, 61.
- Патт, Й. Н. (Patt, Y. N.), 113.
- Переключение, многоканальное+, 101.
- Перемотка, 54.
- Перенос, 35.
- Перепись населения, 19.
- Переполнение, 14, 15, 35, 37, 79, 99, 110.
 - беззнаковое, 125.
- Перепрограммирование, 89.
- Переход
 - нелокальный, 107, 133.
 - трассировочный, 108.
- Петабайт (petabyte), 109.
- Пиксель, 36.
- Подмножество, представление, 36.
- Подпрограмма, 41, 57, 64, 82–84, 89, 91–95, 108.
 - MemFind, 91.
 - вход, 64–70.
 - выход, 64–70.
 - концевая, 70, 78, 95.
 - многократные входы, 68.
 - многократные выходы, 68, 73.
 - неконцевая, 73.

рекурсивная, 79.

рекурсивное использование, 70.

связывание, 64–74.

Подсказка, 25.

Подсчет битов, 19.

Поле

EXPR строки MMIXAL, 50.

МЕТКА строки MMIXAL, 50.

ОП строки MMIXAL, 50.

адреса строки MMIXAL, 39.

конечное, 37.

метки строки MMIXAL, 39.

опкода строки MMIXAL, 39.

Полубайт, 34.

Порция, 91.

Последовательность

вызывающая, 66–69, 73–82, 84.

Фэри, 58.

Постулат Бертрана, 115.

Поток, 86.

Предварительная выборка, 31.

Представление

двунаправленное, 56.

матричное, 58.

Преобразование, двоично-десятичное, 48.

Прерывание, 27.

динамическое, 28.

обхода, 107.

обхода, 79.

Префикс, 95.

единиц измерений, 109.

текущий, 75, 79.

Приближение, Фибоначчи, 59.

Пробел, 36, 51.

Проблема, совмещения имен, 124.

Проверка допустимости деления, 27.

Программа

MMIX, полная, 41.

Hello world, 41.

ассемблирования, 40.

большая, 76–78.

завершение работы, 42.

запуск, 106.

многопроходная, 88.

нахождения максимума, 64–69.

переписывание, 77.

поиска максимума, 39.

полная для MMIX, 57.

с табличным управлением, 96.

создание, 76–78.

условия запуска, 84.

элементарная, 48.

Программирование, грамотное, 57, 78.

Произведение матричное обобщенное, 20, 37.

Пропускание

знака, 17.

знакового бита, 17.

Пространство

пользовательское, 47.

ядра, 47.

Простые числа, программа вычисления, 43–45.

- Протоколирование, 108.
 Профилирование, 108, 114.
 Профиль, 40, 41.
 Проход, 84–86.
 Процедура, отладки, 77.
 Псевдооператор, 40, 42.
- Развертка, цикла, 123.
- Разность
 - байтов, 19.
 - байдов, 19.
 - окта, 19.
 - тетра, 19.
- Райт, Э. М. (Wright, Edward Maitland), 121.
- Распаковка, 96.
- Расширение
 - знака 15, 110.
 - знакового бита, 15.
- Регистр, 12.
 - MMIX, 33.
 - MMIX, специальный, 29.
 - состояния, арифметический, 27.
 - адреса прерывания, 28.
 - виртуальной трансляции, 29.
 - возврата-перехода, 25.
 - времени, 29.
 - глобальный, 25, 45, 71, 79, 94, 99, 107.
 - делимого, 17.
 - запроса прерывания, 28.
 - исполнения, 28.
 - локальный, 25, 71, 79, 95, 99, 107.
 - локальный пороговый, 25.
 - маргинальный, 25, 71, 78, 95, 99, 113.
 - места прерывания, 28.
 - мультиплексной маски, 19.
 - номер, 45.
 - операнда Y, 28.
 - операнда Z, 28.
 - остатка, 16.
 - отступа стека, 26.
 - последовательных значений, 29.
 - самозагрузки, 28.
 - сохранение и восстановление, 67.
 - специальный, 31.
 - старшей половины произведения, 16.
 - указателя стека, 26.
 - эпсилон, 22.
- Режим
 - округления, 28.
 - только для чтения, 47.
- Результат, субнормальный, 104.
- Рекурсия, 141.
- Решение, многовариантное, 57, 58.
- Римские цифры, 10, 12.
- Рокички Т. Г. (Rokicki, Tomas Gerhard), 88.
- Ропкод, 107.
- Рэндольф У. (Randolph V.), 39.
- Ряд, гармонический, 60–61.
- Самовывоз, 142.
- Самоизменение, 108.
- Самоорганизация, 92.
- Светофор, 62.
- Сдвиг, циклический, 37.
- Сегмент
 - данных, 47, 70, 91, 96, 133.
 - памяти, стека, 74, 130.
 - пула, 47, 133.
 - стека, 47, 133.
 - текстовый, 47, 91, 96.
- Септибайт, 109.
- Сильными бинарными операторами, 49.
- Символ
 - локальный, 46, 55.
 - новой строки, 53.
 - определение, 48.
 - подчеркивания, 48.
 - предварительно определенный, 49, 54.
 - пробельный, 81.
 - структурированный, 74.
 - эквивалент в MMIXAL, 49.
- Симулятор
 - MMIX, 32, 41.
 - MMIX на языке MMIX, 90–108.
 - дискретный системный, 90.
 - компьютера, 89–90.
 - конвейера, 59.
 - мета, 59, 90.
- Система
 - обозначений $s(x)$, 14.
 - обозначений $u(x)$, 14.
 - операционная, 47.
 - письма, арабская, 56.
 - счисления, шестнадцатеричная, 29.
- Скаляр, 74.
- Слабый бинарный оператор, 49.
- Сложение, 16, 21, 35.
 - в сторону, 19.
 - насыщающее, 36.
 - цепочки, 114.
- Сопрограмма, 79–87.
 - реплицируемая, 86.
 - связывание, 80, 87.
- Специальные регистры, 13.
- Сравнение, 21.
- Ссылка
 - вперед; см.
- Ссылка, опережающая, 48.
 - опережающая, 48, 51.
- Стандарт, IEEE, 104.
- Стек, 70.
 - отступ, 94.
 - регистров, 25, 71–74, 79, 84, 87, 93–95, 99, 131.
 - указатель, 94.
- Степень
 - возведение, 38.
 - оценка, 38, 114.
- Саймон Н. (Simon, Neil), 9.
- Сайтс, Ричард Л. (Sites, Richard L.), 9.

- Строка, 42.
 вайдовая, 53, 54.
 командная, 105, 141.
 копирование, 59.
 манипуляции, 59.
 новая, 42.
- Сходимость, гармоническая, 60.
- Счетчик
 интервалов, 29.
 использования, 29.
 сбоев, 29.
 циклов, 29.
- Таблица
 переключения, 57, 58, 101, 135.
 переходов, 101.
- Твист, Оливер (Twist, Oliver), 7.
- Терабайт, 109.
- Тетрабайт, 12.
 арифметика со старшей половиной, 112.
 арифметические действия, 37.
- Точка
 в двоичном представлении числа, 17, 35.
 седловая, 58.
- Трассировщик, 108.
- Тьюринг, А. М. (Turing, Alan Mathison), 78.
- Уилсон Дж. П. (Wilson G. P.), 39.
- Указатель
 стека, 70, 94.
 стекового фрейма, 70, 131.
- Умножение, 16, 21, 35, 100.
 на малую константу, 17, 36.
- Упс, 32.
- Участок, 139.
- Файл
 вывода ошибок, 53.
 двоичный, 53, 105, 108, 141.
 объектный, 41, 141.
 стандартного ввода, 53.
 стандартного вывода, 53.
 текстовый, 53.
- Фильтр, 84.
- Флаг, 102.
- Флойд Р. В. (Floyd R. W.), 114.
- Фукс, Д. Р. (Fuchs, D. R.), 88.
- Фэри, Джон (Farey, John), 121.
- Харди, Дж. Х. (Hardy, G. H.), 121.
- Харос, К. (Haros, C.), 121.
- Хеллер, Джозеф (Heller, Joseph), 11.
- Хилл, Роберт (Hill, Robert), 127.
- Хофри, М. (Hofri M.), 120.
- Хэннэси, Джон (Hennessey, John), 9.
- Целая часть, 21.
- Целое число, знаковое, 35.
- Цикл
 оптимизация, 59, 130.
 развертка, 123.
- Цифра, арабская, 56.
- Часы, 90, 127.
- Четность, 36.
- Числа, рациональные, 58.
- Число
 с плавающей точкой, короткое, 21.
 Фибоначчи, 79.
 шестнадцатеричное, 34.
- Член, 49.
- Шахматы, 80.
- Шестнадцатеричные цифры, 11.
- Экзабайт, 109.
- Экспонента, 21.
- Эмулятор, 89.
- Эпсилон, 32.
- Язык программирования, 76.
 ассемблер, 8.
 высокого уровня, 8.
- Язык, ассемблера компьютера ММIX, 39–56.

Искусство программирования

ДОНАЛЬД Э. КНУТ

Эта многотомная работа по анализу алгоритмов давно считается исчерпывающим описанием классической информатики. Три завершенных тома, вышедших в свет к этому времени, представляют собой неоценимый источник информации для теории и практики программирования. Многочисленные читатели этого труда говорят о его огромном влиянии на них и их профессионализм. Ученых потрясает красота и элегантность анализа Кнута, программисты-практики используют эти книги как справочник для решения возникающих перед ними проблем. И все восхищены обширностью, ясностью, точностью и юмором *Искусства программирования*.

В настоящее время, работая над четвертым и последующими томами и над обновлением уже вышедших, Кнут создал серию небольших брошюр, называемых *выпусками*, которые планирует регулярно издавать. Каждый выпуск содержит один или несколько разделов с новым или обновленным материалом. В конечном счете материалы выпусков войдут в окончательные версии каждого тома, и начатая в 1962 году гигантская работа будет завершена.

Том 1, выпуск 1

Данный выпуск представляет собой обновление главы третьего издания первого тома *Искусства программирования*, посвященного основным алгоритмам, и в конечном итоге станет частью четвертого издания этой книги. Здесь читатели найдут введение для программистов с описанием долгожданного компьютера MMIX, RISC-компьютера, который заменит прежний компьютер MIX, и языка ассемблера MMIX.

Этот выпуск также представляет новый материал о подпрограммах, сопрограммах и интерпретируемых подпрограммах.

Дональд Э. Кнут известен всему миру своей пионерской работой по алгоритмам и методам программирования, разработками издательских систем T_EX и METAFONT, а также научными работами. Заслуженный профессор Искусства программирования Станфордского университета в настоящее время посвящает все свое время работе над завершением данных выпусков и всех семи томов своей великой работы.



Издательский дом "ВИЛЬЯМС"
www.williamspublishing.com



ADDISON-WESLEY
Pearson Education

www.awprofessional.com

ISBN 978-5-8459-1163-6



9 785845 911636



0 6 2 4 9