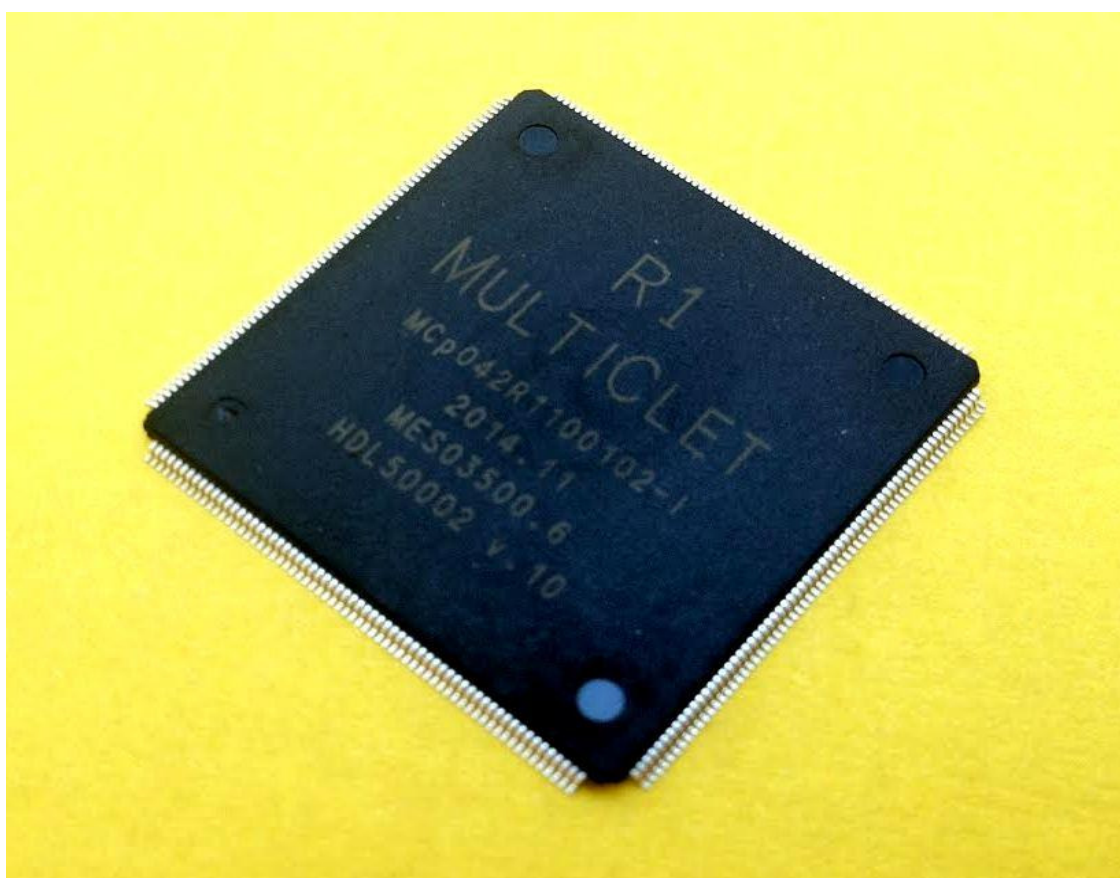




РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ.
Программное обеспечение мультিকлеточного
процессора MultiClet R1



Оглавление

1	Руководство пользователя по ассемблеру	11
1.1	Общие сведения об ассемблере	11
1.1.1	Запуск ассемблера и опции командной строки	11
1.2	Общие сведения о мультиклеточном процессоре	12
1.2.1	Организация памяти	12
1.2.1.1	Память на кристалле.	12
1.2.1.2	Внешнее ОЗУ, внешнее ПЗУ, область устройств ввода/-вывода.	13
1.2.1.3	Область отображения адресов.	14
1.2.1.4	Шина периферийных устройств.	14
1.2.2	Регистры	14
1.2.2.1	Регистры общего назначения	15
1.2.2.2	Регистры индексные	15
1.2.2.3	Регистры управляющие	16
1.2.2.3.1	Регистр PSW управления процессором	16
1.2.2.3.2	Регистр INTR прерываний	18
1.2.2.3.3	Регистр MSKR маски прерываний	20
1.2.2.3.4	Регистр ER ошибок	20
1.2.2.3.5	Регистр IRETADDR адреса возврата из прерывания	21
1.2.2.3.6	Регистр FORCE принудительного запуска прерываний	22
1.2.2.3.7	Регистр IHOOKADDR адреса первичного обработчика прерываний	22
1.2.2.3.8	Регистр INTNUMR номера прерывания	23
1.2.2.3.9	Регистр CURRENTADDR адреса текущего параграфа	23

1.2.2.3.10	Регистр NEWADDR адреса следующего параграфа	24
1.2.2.3.11	Регистр LASTPSW предыдущего значения PSW	24
1.2.2.3.12	Регистр DFADDR адреса следующего параграфа, используемый по умолчанию	25
1.2.2.3.13	Регистр ICR информационной связности	25
1.2.2.4	Регистры системные	26
1.2.2.4.1	Регистр PLLCR управления генератором	28
1.2.2.4.2	Регистр PLLSTR состояния генератора	29
1.2.2.4.3	Регистр PLLTMCR управления модулем коммутации системы тактирования	29
1.2.2.4.4	Регистр SWCHx_CTRL управления каналом x коммутатора памяти	31
1.2.2.4.5	Регистр DTC_CTRL управления контроллером транзакций данных DTC	31
1.2.2.4.6	Регистр DTC_ST состояния DTC	32
1.2.2.4.7	Регистр DTC_IMASKx маски x DTC	33
1.2.2.4.8	Регистр DTC_DATA приёма данных DTC	33
1.2.2.4.9	Регистр DTC_S_ADDR адреса источника данных для DTC	34
1.2.2.4.10	Регистр DTC_D_ADDR адреса приёмника данных для DTC	34
1.2.3	Коммутатор	34
1.3	Основные понятия языка	36
1.3.1	Комментарии	36
1.3.2	Константы	36
1.3.2.1	Числовые константы	36
1.3.2.2	Символьные (литеральные) константы	37
1.3.3	Секции	38
1.3.3.1	Подсекции	40
1.3.4	Символы	40
1.3.4.1	Система имён символов	41
1.3.4.2	Метки	41
1.3.4.3	Символы с абсолютным значением	41
1.3.4.4	Атрибуты символов	41
1.3.5	Выражения	42
1.3.5.1	Пустые выражения	42
1.3.5.2	Целочисленные выражения	42

1.3.5.3	Аргументы	42
1.3.5.4	Операторы	43
1.4	Система команд ассемблера	45
1.4.1	Условные обозначения	45
1.4.2	Типы операций	46
1.4.3	Общий принцип построения команд в ассемблере	48
1.4.3.1	Общие правила формирования аргументов команд и их интерпретация	48
1.4.3.2	Правила формирования результатов команд	51
1.4.4	Описание команд	53
1.4.4.1	abs	53
1.4.4.2	adc	56
1.4.4.3	add	60
1.4.4.4	and	64
1.4.4.5	bc	66
1.4.4.6	bsf	68
1.4.4.7	bsr	70
1.4.4.8	bswap	72
1.4.4.9	cb	75
1.4.4.10	cbst	77
1.4.4.11	cfsi	80
1.4.4.12	cfi	83
1.4.4.13	cdi	85
1.4.4.14	clfi	87
1.4.4.15	csfi	90
1.4.4.16	div	93
1.4.4.17	exa	96
1.4.4.18	fcpi	102
1.4.4.19	forsi	104
1.4.4.20	forsi / fornc	106
1.4.4.21	forsi / fornc	108
1.4.4.22	forsi	110
1.4.4.23	forsi	112
1.4.4.24	forsi	114
1.4.4.25	forsi	116
1.4.4.26	forsi	118
1.4.4.27	forsi	120

1.4.4.28	forsne	122
1.4.4.29	forsno	124
1.4.4.30	forsns	126
1.4.4.31	forso	128
1.4.4.32	forss	130
1.4.4.33	ge	132
1.4.4.34	get	134
1.4.4.35	ib	138
1.4.4.36	insub	140
1.4.4.37	irm	145
1.4.4.38	irma	147
1.4.4.39	irmae / irmnc	149
1.4.4.40	irmb / irmc	151
1.4.4.41	irmbe	153
1.4.4.42	irme	155
1.4.4.43	irmg	157
1.4.4.44	irmge	159
1.4.4.45	irml	161
1.4.4.46	irmle	163
1.4.4.47	irmne	165
1.4.4.48	irmno	167
1.4.4.49	irmns	169
1.4.4.50	irmo	171
1.4.4.51	irms	173
1.4.4.52	ja	175
1.4.4.53	jae / jnc	176
1.4.4.54	jb / jc	177
1.4.4.55	jbe	178
1.4.4.56	je	179
1.4.4.57	jg	180
1.4.4.58	jge	181
1.4.4.59	jl	182
1.4.4.60	jle	183
1.4.4.61	jmp	184
1.4.4.62	jne	186
1.4.4.63	jno	187
1.4.4.64	jns	188

1.4.4.65	jo	189
1.4.4.66	js	190
1.4.4.67	lt	191
1.4.4.68	madd	193
1.4.4.69	max	195
1.4.4.70	min	197
1.4.4.71	move	199
1.4.4.72	mul	200
1.4.4.73	norm	204
1.4.4.74	not	206
1.4.4.75	or	208
1.4.4.76	pack	210
1.4.4.77	patch	212
1.4.4.78	rd	214
1.4.4.79	rdd	219
1.4.4.80	rol	222
1.4.4.81	ror	224
1.4.4.82	sa	226
1.4.4.83	sae / snc	228
1.4.4.84	sar	230
1.4.4.85	sb / sc	232
1.4.4.86	sbb	234
1.4.4.87	sbe	236
1.4.4.88	se	238
1.4.4.89	set	240
1.4.4.90	sg	245
1.4.4.91	sge	247
1.4.4.92	sl	249
1.4.4.93	sle	251
1.4.4.94	sll / sal	253
1.4.4.95	slr	255
1.4.4.96	sne	257
1.4.4.97	sno	259
1.4.4.98	sns	261
1.4.4.99	so	263
1.4.4.100	ss	265
1.4.4.101	sqrt	267

1.4.4.102 sub	269
1.4.4.103 wr	273
1.4.4.104 wrd	276
1.4.4.105 xor	278
1.5 Система директив ассемблера	280
1.5.1 .alias	280
1.5.2 .align	280
1.5.3 .ascii	281
1.5.4 .asciiz	281
1.5.5 .bss	281
1.5.6 .byte	282
1.5.7 .comm	282
1.5.8 .data	282
1.5.9 .double	282
1.5.10 .else	283
1.5.11 .elseif	283
1.5.12 .end	283
1.5.13 .endif	283
1.5.14 .endr	283
1.5.15 .equ	284
1.5.16 .equiv	284
1.5.17 .eqv	284
1.5.18 .err	284
1.5.19 .error	284
1.5.20 .fail	285
1.5.21 .file	285
1.5.22 .fill	285
1.5.23 .float	285
1.5.24 .global, .globl	286
1.5.25 .if	286
1.5.26 .include	287
1.5.27 .lcomm	287
1.5.28 .loc	288
1.5.29 .local	288
1.5.30 .long	289
1.5.31 .p2align	289
1.5.32 .print	290

1.5.33	.quad	290
1.5.34	.rept	290
1.5.35	.section	290
1.5.36	.set	291
1.5.37	.short	291
1.5.38	.single	292
1.5.39	.size	292
1.5.40	.skip	292
1.5.41	.sleb128	292
1.5.42	.space	293
1.5.43	.string	293
1.5.44	.subsection	294
1.5.45	.syntax	294
1.5.46	.text	295
1.5.47	.type	295
1.5.48	.uleb128	295
1.5.49	.warning	295
1.5.50	.weak	296
1.6	Принципы программирования на ассемблере для мультиклеточного процессора	297
1.7	Прерывания и их обработка	300
1.8	Динамическая реконфигурация	303
1.8.1	Декомпозиция	304
1.8.2	Композиция	305
2	Руководство пользователя по редактору связей	307
2.1	Общие сведения о редакторе связей	307
2.2	Использование редактора связей	308
2.3	Скрипты	309
2.3.1	Выражения	309
2.3.1.1	Константы	310
2.3.1.2	Имена символов	310
2.3.1.3	Счётчик текущего местоположения	310
2.3.1.4	Операторы	312
2.3.1.5	Оценка и вычисление выражения	312
2.3.1.6	Секция результата вычисления выражения	313
2.3.1.7	Встроенные функции	315

2.3.1.7.1	ABSOLUTE	315
2.3.1.7.2	ADDR	315
2.3.1.7.3	ALIGN	316
2.3.1.7.4	ALIGNOF	317
2.3.1.7.5	ASSERT	317
2.3.1.7.6	DEFINED	317
2.3.1.7.7	LENGTH	318
2.3.1.7.8	LOADADDR	318
2.3.1.7.9	MAX	318
2.3.1.7.10	MIN	318
2.3.1.7.11	ORIGIN	318
2.3.1.7.12	SEGMENT_START	318
2.3.1.7.13	SIZEOF	319
2.3.2	Операции присваивания	319
2.3.2.1	Простые присваивания	319
2.3.2.2	HIDDEN символы	321
2.3.2.3	PROVIDE символы	321
2.3.2.4	PROVIDE_HIDDEN символы	321
2.3.3	Команды	321
2.3.3.1	ASSERT	322
2.3.3.2	DEBUG	322
2.3.3.3	ENTRY	323
2.3.3.4	GROUP	323
2.3.3.5	INCLUDE	323
2.3.3.6	INPUT	324
2.3.3.7	MEMORY	324
2.3.3.8	OUTPUT	326
2.3.3.9	OUTPUT_ARCH	326
2.3.3.10	PHDRS	327
2.3.3.11	REGION_ALIAS	329
2.3.3.12	SEARCH_DIR	329
2.3.3.13	SECTIONS	329
2.3.3.13.1	Аттрибуты выходной секции	331
2.3.3.13.1.1	Имя выходной секции.	331
2.3.3.13.1.2	Виртуальный адрес (VMA) выходной секции.	331
2.3.3.13.1.3	Тип выходной секции.	332

2.3.3.13.1.4	Адрес загрузки (LMA) выходной секции.	332
2.3.3.13.1.5	Принудительное выравнивание выходной секции.	333
2.3.3.13.1.6	Принудительное выравнивание входной секции.	333
2.3.3.13.1.7	Регион памяти выходной секции.	333
2.3.3.13.1.8	Программный заголовок (сегмент) выходной секции.	333
2.3.3.13.1.9	Заполнитель выходной секции.	334
2.3.3.13.2	Описание входных секций	334
2.3.3.13.2.1	Базовое описание входной секции.	335
2.3.3.13.2.2	Шаблоны подстановки для входных секций	336
2.3.3.13.2.3	Входная секция для общих символов.	337
2.3.3.13.2.4	Пример описания входной секции.	338
2.3.3.13.3	Данные выходной секции, непосредственно включаемые с помощью специальных команд	338
2.3.3.13.4	Отбрасывание выходной секции	340
2.3.3.14	STARTUP	340
3	Руководство пользователя по функциональной модели	341
3.1	Общие сведения о функциональной модели процессора	341
3.2	Использование функциональной модели процессора	342
4	Руководство пользователя по отладчику	345
4.1	Общие сведения об отладчике mc-dbg	345
4.2	Процесс использования отладчика mc-dbg	346
4.3	Набор команд	347
4.3.1	Команды общего назначения	347
4.3.1.1	help	347
4.3.1.2	quit	347
4.3.2	Команды управления процессом выполнения	348
4.3.2.1	run	348
4.3.2.2	next	348
4.3.2.3	continue	348
4.3.2.4	stop	349
4.3.2.5	kill	349
4.3.2.6	until	349
4.3.2.7	where	349

4.3.2.8	jump	350
4.3.3	Команды управления секцией памяти данных	350
4.3.3.1	print	350
4.3.3.2	display	352
4.3.3.3	set	354
4.3.4	Команды управления точками останова	354
4.3.4.1	break	354
4.3.4.2	watch	355
4.3.5	Команды просмотра доступных меток	356
4.3.5.1	file	356
4.3.5.2	list	356
4.3.6	Другие команды	357
4.3.6.1	info	357
4.3.6.2	enable	358
4.3.6.3	disable	359
4.3.6.4	delete	359
4.3.6.5	model	360
4.3.6.6	stat	362
4.4	Процесс использования отладчика mc-dbg в графическом режиме	363
4.4.1	Установка	363
4.4.2	Менеджер проектов	365
4.4.3	Инструменты сборки	369
4.4.4	Инструменты отладки	371
4.4.5	Настройка горячих клавиш	374
4.4.6	Пример	375

1. Руководство пользователя по ассемблеру

1.1 Общие сведения об ассемблере

Представленный ассемблер используется для написания программ для мультиклеточных процессоров.

1.1.1 Запуск ассемблера и опции командной строки

Ассемблер запускается из командной строки командой *mc-as*, аргументом которой является файл с исходным кодом. Поддерживаются также следующие опции:

- I, --include-path=DIR — добавить директорию DIR в список директорий, используемых для поиска файла, подключаемого директивой ассемблера «.include»
- o, --output=objfile — поместить вывод в объектный файл objfile
- g — добавить в выходной файл необходимую для отладки информацию
- arch=CPU — сгенерировать код для CPU, где CPU имеет одно из следующих значений: MCp0411100101, MCp042R100102
- syntax=[V1|V2] — использовать синтаксис команд версии V1/V2
- V — напечатать номер версии ассемблера
- h, --help — показать это сообщение и выйти

1.2 Общие сведения о мультиклеточном процессоре

Процессор МСр042R100102 имеет в своем составе мультиклеточное процессорное ядро — первое процессорное ядро с принципиально новой мультиклеточной архитектурой российской разработки. Ядро МП содержит 4 клетки — когерентных процессорных блоков, объединенных интеллектуальной коммутационной средой. МСр042R100102 имеет динамически-реконфигурируемое ядро, распределение вычислительных ресурсов (клеток) по задачам производится согласно алгоритму заложенному в исполняемой программе. При этом один и тот же код может выполняться любым количеством и комбинацией клеток. Мультиклеточный процессор предназначен для решения широкого круга задач управления и цифровой обработки сигналов в приложениях, требующих минимального энергопотребления и высокой производительности.

1.2.1 Организация памяти

Память для процессора МСр042R100102 с программной точки зрения представляет собой массив с байтовой адресацией, диапазон адресов $[0, 2^{32})$. Память не разделена на страницы и доступна вся сразу, но есть ограничения по размещению данных и программ, которые описаны ниже.

На рис. 1.1 приведена карта памяти процессора.

1.2.1.1 Память на кристалле.

Представляет собой статическое оперативное запоминающее устройство.

Блок памяти в диапазоне адресов от $0x00000000$ до $0x00040000$ специализирован для размещения в нем исполняемого кода. В нем можно разместить любые программы с любым сочетанием типов команд, а также данные. Также только в этом диапазоне адресов можно поставить аппаратные точки останова для инструкций.

Блок памяти в диапазоне адресов от $0x00040000$ до $0x00080000$ специализирован для размещения в нем данных. В нем также можно хранить исполняемый код, однако не рекомендуется передавать управление на адреса в данном блоке, т. к. клетки возможно будут обращаться к одному и тому же физическому блоку, что будет причиной простоя, и, как следствие, возможно существенное снижение производительности.

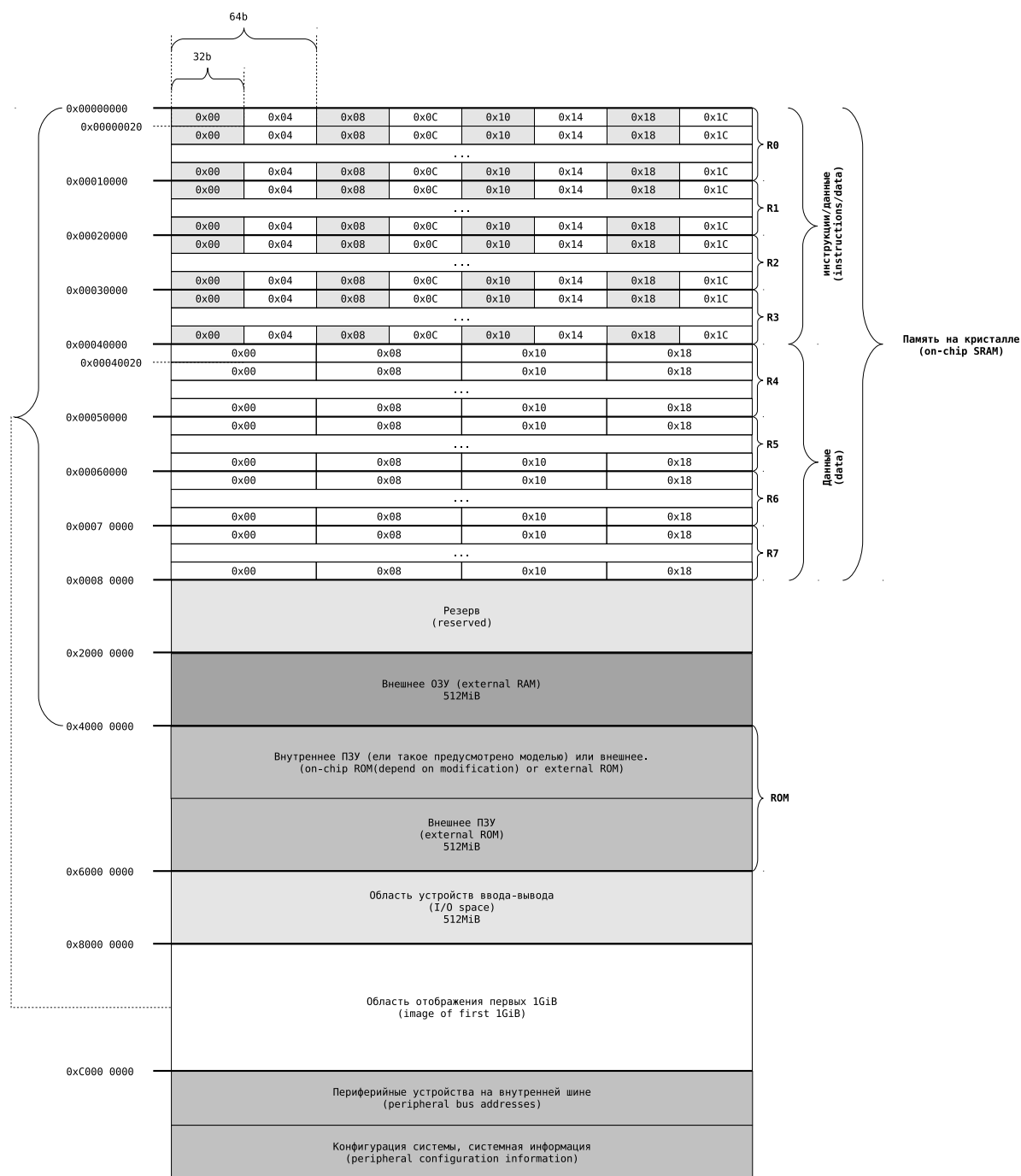


Рис. 1.1: Карта памяти

1.2.1.2 Внешнее ОЗУ, внешнее ПЗУ, область устройств ввода/вывода.

Для доступа к данным блокам памяти используется специализированная шина внешней памяти. Возможно выполнять, хранить данные в данных блоках, но следует учитывать, что это существенно снизит производительность процессора, т. к. 4 клетки за один такт могут запрашивать в памяти до 32 байт информации (максимальная ширина шины

внешней памяти — 32 бита (4 байта), скорость работы зависит от внешних компонентов и тактовой частоты процессора).

1.2.1.3 Область отображения адресов.

Данная зона используется контроллером прямого доступа к памяти на периферийной шине. Центральное процессорное устройство так же может обратиться по адресам в этой зоне.

1.2.1.4 Шина периферийных устройств.

В данной зоне расположены регистры периферийных устройств. В последних адресах расположена информация о конфигурации периферии процессора.

1.2.2 Регистры

Мультиклеточный процессор МСр042R100102 имеет в своем составе следующие регистры:

- регистры общего назначения (РОН [GPR — General Purpose Register]);
- регистры индексные (ПИ [IR — Index Register]);
- регистры управляющие (ПУ [CR — Control Register]);
- регистры системные (РС [SR — System Register]).

Обращение к регистру осуществляется по его номеру или имени, которому предшествует символ диеза '#', с помощью специализированных команд. Общее количество регистров — 64. Все команды всех процессорных устройств при декодировании имеют одновременный доступ на чтение ко всем регистрам. Запись в регистры осуществляется также одновременно по завершению текущего параграфа. Нумерация регистров является сквозной и начинается с нуля.

РОНы, индексные и управляющие регистры имеют копии в каждой клетке. Эта особенность является необходимостью процессора МСр042R100102. т. к. при динамической реконфигурации необходимо иметь набор регистров в каждой клетке.

Системные регистры находятся в устройствах и являются разделяемыми ресурсами микропроцессора.

1.2.2.1 Регистры общего назначения

Используются в качестве сверхбыстрой памяти (Scratchpad memory). Для обращения к какому-либо регистру данного типа используются номера от 0 до 7. Имен у данного типа регистров нет. Интерпретация значения регистра зависит от типа команды. Начальное значение регистров данного типа не определено.

1.2.2.2 Регистры индексные

Используются для косвенной адресации. Логическая структура индексного регистра показана на рис. 1.2.

Номера битов	63...48	47...32	31...0
	Индекс (<i>Index</i>)	Маска (<i>Mask</i>)	База (<i>Base</i>)

Рис. 1.2: Структура индексного регистра

Для обращения к какому-либо регистру данного типа используются номера от 8 до 15. Имен у данного типа регистров нет. Начальное значение регистров данного типа не определено.

В общем случае (см. исключения в описании конкретной команды в разделе «Система команд ассемблера») при использовании регистра данного типа в качестве аргумента операции значение этого аргумента формируется согласно следующей формуле:

$$Address = Index + Base$$

Модификация значения индексного регистра осуществляется аппаратно по завершению параграфа в том случае, если это явно указано в параграфе при помощи команды «itm» и она была выполнена, согласно следующей формуле:

$$Index = ((Index \mid \sim Mask) + 1) \& Mask,$$

где $\mid$ — операция побитового «ИЛИ», $\&$ — операция побитового «И», $\sim$ — операция побитового инвертирования.

В двух выше приведённых формулах используется целочисленная 32-х разрядная арифметика. Значения старших 16 разрядов (с 16 по 31) полей Индекс (*Index*) и Маска (*Mask*) заполняются нулями.

1.2.2.3 Регистры управляющие

Процессор имеет в своем составе следующие управляющие регистры:

Номер регистра	Имя регистра	Права доступа	Описание
48	PSW	R/W	Регистр управления процессором
49	INTR	R/W	Регистр прерываний
50	MSKR	R/W	Регистр маски прерываний
51	ER	R	Регистр ошибок
52	IRETADDR	R	Регистр адреса возврата из прерывания
53	FORCE	W	Регистр принудительного запуска прерываний
55	IHOOKADDR	R/W	Регистр адреса первичного обработчика прерываний
56	INTNUMR	R	Регистр номера прерывания
59	CURRENTADDR	R	Регистр адреса текущего параграфа
60	NEWADDR	W	Регистр адреса следующего параграфа
61	LASTPSW	R	Регистр предыдущего значения PSW
62	DFADDR	W	Регистр адреса по умолчанию следующего параграфа
63	ICR	R/W	Регистр информационной связности

1.2.2.3.1 Регистр PSW управления процессором

Регистр управления процессором предназначен для управления вычислительным процессом и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра PSW

№№ битов	Обозначение	Описание
0	ONIRQS	Разрешение обработки немаскируемых прерываний (аппаратные прерывания, внешние прерывания, программные глобальные и локальные прерывания). Признак не блокирует прием прерываний регистром INTR, а блокирует только их дальнейшую обработку. Устанавливается и снимается всегда только программно (путем установки нового значения PSW). Если $ONIRQS = 1$, то разрешена обработка нефатальных исключений. Если $ONIRQS = 0$, то обработка нефатальных исключений запрещена.
1 — 4		Зарезервировано
5	SWR	Программный сброс. При установке в 1, по завершению параграфа формируется аппаратный сигнал «reset» и осуществляется начальный старт с адреса 0x00000000.
6	RW	Очередность исполнения команд чтения/записи: 0 — команды записи выполняются только после выполнения всех команд чтения выполняемого параграфа, а выбранные команды чтения следующего параграфа только после выполнения всех команд записи выполняемого параграфа 1 — контроль очередности не выполняется
7	RCFG	Признак динамической реконфигурации. Формируется программно. Разрешает изменение регистра информационной связности ICR (формирование новой task-группы) и установку адреса первого исполняемого параграфа этой новой task-группы в параграфе, следующем за параграфом, в котором устанавливается $RCFG = 1$. Снимается аппаратно, после получения новых значений регистра информационной связности и адреса следующего параграфа.

8	CLNBUF	Признак очистки буфера. Формируется программно. При $CLNBUF = 1$ переход к выборке и исполнению команд следующего параграфа осуществляется только после выполнения всех команд текущего параграфа. При этом признак снимается аппаратно после передачи на исполнение последней команды в буфере.
9 — 63		Зарезервировано

1.2.2.3.2 Регистр INTR прерываний

Регистр прерываний предназначен для сигнализации наличия прерываний, а также инициации программных прерываний и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Данный регистр является регистром защёлкой и он захватывает пришедшие на его вход данные до тех пор, пока не будет сброшен записью 1 в соответствующие биты, т. е. запись 1 в какие-либо биты данного регистра произведёт сброс соответствующих бит в 0.

Структура регистра INTR

№№ битов	Обозначение	Описание
0	ENMI	Немаскируемое внешнее прерывание
1		Зарезервировано
2	PERE	Немаскируемое исключение в аппаратной части
3	PRGE	Немаскируемое программное исключение
4		Зарезервировано
5		Зарезервировано
6	WKUP	Маскируемое прерывание: внешний сигнал wake_up
7	DTC0	Маскируемое прерывание от контроллера транзакций

8	ST0	Маскируемое прерывание от системного таймера
9	ST1	Маскируемое прерывание от системного таймера
10	ST2	Маскируемое прерывание от системного таймера
11	ST3	Маскируемое прерывание от системного таймера
12	SW0	Маскируемое программное прерывание
13	SW1	Маскируемое программное прерывание
14	SW2	Маскируемое программное прерывание
15	SW3	Маскируемое программное прерывание
16	ADC0_IRQ	Маскируемое прерывание от ADC0
17	DAC0_IRQ	Маскируемое прерывание от DAC0
18	ADC1_IRQ	Маскируемое прерывание от ADC1
19	DAC1_IRQ	Маскируемое прерывание от DAC1
20	UART0_IRQ	Маскируемое прерывание от UART0
21	UART1_IRQ	Маскируемое прерывание от UART1
22	UART2_IRQ	Маскируемое прерывание от UART2
23	UART3_IRQ	Маскируемое прерывание от UART3
24	I2C0_IRQ	Маскируемое прерывание от I2C0
25	I2C1_IRQ	Маскируемое прерывание от I2C1
26	SPI0_IRQ	Маскируемое прерывание от SPI0
27	SPI1_IRQ	Маскируемое прерывание от SPI1
28	I2S0_IRQ	Маскируемое прерывание от I2S0
29	GPTIM0_IRQ	Маскируемое прерывание от GPTIM0
30	GPTIM1_IRQ	Маскируемое прерывание от GPTIM1
31	GPTIM2_IRQ	Маскируемое прерывание от GPTIM2
32	GPTIM3_IRQ	Маскируемое прерывание от GPTIM3
33	WDT_IRQ	Маскируемое прерывание от WDT
34	RTC_IRQ	Маскируемое прерывание от RTC
35	GPIOA_IRQ	Маскируемое прерывание от GPIOA
36	GPIOB_IRQ	Маскируемое прерывание от GPIOB
37	GPIOC_IRQ	Маскируемое прерывание от GPIOC
38	GPIOD_IRQ	Маскируемое прерывание от GPIOD

39	GPIOE_IRQ	Маскируемое прерывание от GPIOE
40	GPIOF_IRQ	Маскируемое прерывание от GPIOF
41	ETHERNET0_IRQ	Маскируемое прерывание от ETHERNET0
42	USB0_IRQ	Маскируемое прерывание от USB0
43	USB0_EPI_IRQ	Маскируемое прерывание от USB0_EPI
44	USB0_EPO_IRQ	Маскируемое прерывание от USB0_EPO
45	PWM0_IRQ	Маскируемое прерывание от PWM0
46	STAT_IRQ	Маскируемое прерывание от STAT
47 — 63		Зарезервировано

1.2.2.3.3 Регистр MSKR маски прерываний

Регистр маски прерываний предназначен для маскирования прерываний и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Для разрешения прерывания с определёнными номерами необходимо выставить в 1 соответствующие биты данного регистра. Биты с 0 — 3 являются немаскируемыми.

Структура регистра MSKR

№№ битов	Обозначение	Описание
0 — 3		Зарезервировано
4 — 46		Номер маскируемого прерывания
47 — 63		Зарезервировано

1.2.2.3.4 Регистр ER ошибок

Регистр ошибок предназначен для сохранения информации о программных или аппаратных сбоях и доступен только для чтения. Начальное значение регистра равно нулю.

Во всех битах регистра 1 является признаком возникновения события.

Структура регистра ER

№№ битов	Обозначение	Описание
0	FNAN	В ходе операции с числами с плавающей точкой или на входе блока операций с числами с плавающей точкой возникло значение NAN
1	ISQRT	Попытка взять квадратный корень из отрицательного числа в блоке операций с числами с плавающей точкой
2		Зарезервировано
3	OV	Возникло переполнение при попытке разделить на 0 в блоке операций с числами с плавающей точкой
4	II	Выбрана несуществующая инструкция
5 — 6	HRESP	Ошибка при обращении по шине периферийных устройств (значения отличные от нуля являются признаком возникновения ошибки)
7 — 63		Зарезервировано

1.2.2.3.5 Регистр IRETADDR адреса возврата из прерывания

Регистр адреса возврата из прерывания предназначен для сохранения адреса возврата памяти, формируемого только при прерывании, и доступен только для чтения. Начальное значение регистра равно нулю.

Следует заметить, что фактический уход на первичный обработчик прерываний (см. 1.2.2.3.7) осуществляется только по окончании выполнения всех операций текущего параграфа и известности адреса следующего параграфа, который формируется одной из операций установки адреса следующего параграфа. В регистре адреса возврата из прерывания сохраняется адрес следующего параграфа. Если адрес перехода на следующий параграф не известен (не был сформирован), тогда процессор перейдёт в состояние ожидания сигнала «start» (ожидание внешнего перезапуска).

Структура регистра ER

№№ битов	Обозначение	Описание
0 — 31		Адрес возврата из прерывания, формируемый только при прерывании
32 — 64		Зарезервировано

1.2.2.3.6 Регистр FORCE принудительного запуска прерываний

Регистр принудительного запуска прерываний предназначен для принудительного запуска прерываний и доступен только для записи. Начальное значение регистра равно нулю.

Запись 1 в какие-либо биты данного регистра сформирует единичный импульс на вход регистра прерываний, что в свою очередь вызовет установку соответствующих бит в регистре INTR. На следующем такте после записи 1 в какие-либо биты данного регистра все биты данного регистра будут сброшены в 0.

Структура регистра FORCE

№№ битов	Обозначение	Описание
0 — 46		Номер принудительного прерывания
47 — 63		Зарезервировано

1.2.2.3.7 Регистр INHOOKADDR адреса первичного обработчика прерываний

Регистр адреса первичного обработчика прерываний предназначен для сохранения адреса памяти первичного обработчика прерывания и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра IHOOKADDR

№№ битов	Обозначение	Описание
0 — 31		Значение адреса первичного обработчика прерываний
32 — 63		Зарезервировано

1.2.2.3.8 Регистр INTNUMR номера прерывания

Регистр номера прерывания предназначен для сохранения номера самого приоритетного разрешенного прерывания на данный момент и доступен только для чтения. Начальное значение регистра равно нулю.

Структура регистра INTNUMR

№№ битов	Обозначение	Описание
0 — 5		Номер самого приоритетного разрешенного прерывания на момент чтения регистра
6 — 63		Зарезервировано

1.2.2.3.9 Регистр CURRENTADDR адреса текущего параграфа

Регистр адреса текущего параграфа предназначен для сохранения адреса начала текущего выполняемого параграфа и доступен только для чтения. Значение данного регистра изменяется аппаратно по завершению параграфа (перед выборкой первой команды следующего параграфа, который будет выполняться после текущего). Начальное значение регистра равно нулю.

Структура регистра CURRENTADDR

№№ битов	Обозначение	Описание
0 — 31		Значение адреса текущего параграфа
32 — 63		Зарезервировано

1.2.2.3.10 Регистр NEWADDR адреса следующего параграфа

Регистр адреса следующего параграфа предназначен для задания адреса следующего параграфа и доступен только для записи. Начальное значение регистра равно нулю.

Структура регистра NEWADDR

№№ битов	Обозначение	Описание
0 — 31		Значение адреса следующего параграфа
32 — 63		Зарезервировано

1.2.2.3.11 Регистр LASTPSW предыдущего значения PSW

Регистр предыдущего значения PSW предназначен для сохранения предыдущего значения регистра PSW и доступен только для чтения. Начальное значение регистра равно нулю.

Структура регистра LASTPSW

№№ битов	Обозначение	Описание
0 — 8		Предыдущее значение регистра PSW
9 — 63		Зарезервировано

1.2.2.3.12 Регистр DFADDR адреса следующего параграфа, используемый по умолчанию

Регистр DFADDR предназначен для установки адреса следующего параграфа в текущем выполняемом параграфе, который будет использован в случае неопределённости адреса следующего параграфа (в выполняемом параграфе не выполнена ни одна из команд установки адреса следующего параграфа) и доступен только для записи. Начальное значение регистра равно нулю. Значение данного регистра может быть использовано только в текущем выполняемом параграфе, при переходе к следующему параграфу оно не сохраняется.

Структура регистра DFADDR

№№ битов	Обозначение	Описание
0 — 31		Значение адреса следующего параграфа, используемого по умолчанию
32 — 63		Зарезервировано

1.2.2.3.13 Регистр ICR информационной связности

Регистр информационной связности предназначен для определения подмножества клеток образующих task-группу, т. е. группу клеток, согласованно выполняющих одну программу независимо от других клеток, и доступен как для чтения, так и для записи. Начальное значение регистра равно $0x0F$.

Если i -я и j -я клетки входят в одну task-группу, то содержимое $ICR(i) = ICR(j) = 1$ для обеих клеток. Если они в разных task-группах, то $ICR(i) = ICR(j) = 0$.

Регистр ICR позволяет каждой клетке выбрать из всей совокупности управляющих сигналов и результатов команд мультিকлеточного процессора только те, которые относятся к ее task-группе.

В исходном состоянии все клетки образуют одну task-группу и, следовательно, ICR во всех битах содержит 1. Новое содержимое ICR формируется только программно при динамической реконфигурации мультиклеточного процессора (см. 1.8).

Структура регистра ICR

№№ битов	Обозначение	Описание
0	CELL3	Принадлежность клетки с физическим номером 3 к task-группе: 1 — принадлежит, 0 — не принадлежит
1	CELL2	Принадлежность клетки с физическим номером 2 к task-группе: 1 — принадлежит, 0 — не принадлежит
2	CELL1	Принадлежность клетки с физическим номером 1 к task-группе: 1 — принадлежит, 0 — не принадлежит
3	CELL0	Принадлежность клетки с физическим номером 0 к task-группе: 1 — принадлежит, 0 — не принадлежит
4 — 63		Зарезервировано

1.2.2.4 Регистры системные

Процессор имеет в своем составе следующие системные регистры:

Номер регистра	Имя регистра	Права доступа	Описание
16	PLLCR	R/W	Регистр управления генератором
17	PLLSTR	R/W	Регистр состояния генератора
18	PLLTMCr	R/W	Регистр управления модулем коммутации системы тактирования
19	SWCH0_CTRL	R/W	Регистр управления каналом 0 коммутатора памяти
20	SWCH1_CTRL	R/W	Регистр управления каналом 1 коммутатора памяти
21	SWCH2_CTRL	R/W	Регистр управления каналом 2 коммутатора памяти
22	SWCH3_CTRL	R/W	Регистр управления каналом 3 коммутатора памяти
23	ST0PRDR	R/W	Регистр периода системного таймера 0

24	ST1PRDR	R/W	Регистр периода системного таймера 1
25	ST2PRDR	R/W	Регистр периода системного таймера 2
26	ST3PRDR	R/W	Регистр периода системного таймера 3
27	ST0CR	R/W	Регистр управления таймером 0
28	ST1CR	R/W	Регистр управления таймером 1
29	ST2CR	R/W	Регистр управления таймером 2
30	ST3CR	R/W	Регистр управления таймером 3
32	ST0VAL	R/W	Регистр текущего значения таймера 0
33	ST1VAL	R/W	Регистр текущего значения таймера 1
34	ST2VAL	R/W	Регистр текущего значения таймера 2
35	ST3VAL	R/W	Регистр текущего значения таймера 3
40	DTC_CTRL	R/W	Регистр управления контроллером транзакций данных DTC
41	DTC_ST	R/W	Регистр состояния DTC
42	DTC_IMASK0	R/W	Регистр маски 0 DTC
43	DTC_IMASK1	R/W	Регистр маски 1 DTC
44	DTC_DATA	R/W	Регистр приёма данных DTC
45	DTC_S_ADDR	R/W	Регистр адреса источника данных для DTC
46	DTC_D_ADDR	R/W	Регистр адреса приёмника данных для DTC
47	ROM_CTRL	R/W	Регистр управления и статуса внешнего ПЗУ

1.2.2.4.1 Регистр PLLCR управления генератором

Регистр управления генератором предназначен для управления генератором DTC и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра PLLCR

№№ битов	Обозначение	Описание
0 — 6	NCNT	Коэффициент деления внутреннего генератора
8 — 11	MXVCO	Коэффициент деления выходного делителя
12 — 14	RCNT	Коэффициент деления опорной частоты
15	EN	Включение блока
16 — 17	LOCKDLY	Настройка детектора захвата опорной частоты, задержка в тактах опорной частоты до установления сигнала lock: 00 — 8; 01 — 16; 10 — 32; 11 — 64.
18 — 19	CPCTR	Выбор величины тока в блоке ГПЗ, мкА: 00 — 4; 01 — 2; 10 — 8; 11 — 6.
20	IRMODE	Настройка тока: 0 — выходной ток температурно-стабилизированный, с низкой зависимостью от напряжения питания vdda; 1 — выходной ток температурно-стабилизированный, с высокой зависимостью от напряжения питания vdda.
21	PDDL	Изменение длительности сигнала сброса фазочастотного детектора, нс: 0 — 1 нс; 1 — 0.5 нс;
22	LOCK_RANGE	Настройка блока детектора захвата

23	LOCK_LS	Настройки блока детектора захвата: 0 — при частоте после делителя опорного сигнала [4, 20] МГц; 1 — при частоте после делителя опорного сигнала [1, 8] МГц;
24 — 63		Зарезервировано

1.2.2.4.2 Регистр PLLSTR состояния генератора

Регистр состояния генератора предназначен для мониторинга состояния генератора и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра DFADDR

№№ битов	Обозначение	Описание
0	LOCK	Частота на выходе генератора стабильна (1 — частота стабильна)
1	FREFLOW	Частота опорного тактового сигнала: 0 — частота больше 190 кГц, 1 — частота ниже 150 кГц
2	REFACTIVE	Наличие опорного тактового сигнала на выходе генератора (1 — подано)
3 — 63		Зарезервировано

1.2.2.4.3 Регистр PLLTMCR управления модулем коммутации системы тактирования

Регистр управления модулем коммутации системы тактирования предназначен для управления модулем коммутации и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра PLLTMCR

№№ битов	Обозначение	Описание
0	INVERTB	Инвертирование сигнала канала В (1 — инвертирован)
1	INVERTA	Инвертирование сигнала канала А (1 — инвертирован)
2 — 4	SELOCUR	Управление током выходных буферов: 000 — 2,0 мА, 001 — 4,8 мА, 010 — 7,7 мА, 011 — 10,6 мА, 100 — 4,0 мА, 101 — 9,6 мА, 110 — 15,4 мА, 111 — 21,2 мА.
5	CFGOB	Управление состоянием выходного буфера В (1 — значение тока выходного буфера зависит от значения PLLTMCR(SELOCUR); 0 — значении выходного тока — 240мкА)
6	CFGOA	Управление состоянием выходного буфера А (1 — значение тока выходного буфера зависит от значения PLLTMCR(SELOCUR); 0 — значении выходного тока — 240мкА)
7 — 10	SELB	Управление мультиплексором канала В: 0100 — F_{ref} , 1001 — F_{sys} , остальные значения зарезервированы.
11 — 14	SELA	Управление мультиплексором канала А: 0100 — F_{ref} , 1001 — F_{sys} , остальные значения зарезервированы.
15 — 22		Зарезервировано
23	EN	Разрешение работы (1 — включен, 0 — выключен)
24 — 63		Зарезервировано

1.2.2.4.4 Регистр SWCHx_CTRL управления каналом x коммутатора памяти

Регистр управления каналом коммутатора предназначен для определения приоритетов доступа к областям памяти и доступен как для чтения, так и для записи. Начальное значение регистра равно 0x00550000.

Структура регистра SWCHx_CTRL

№№ битов	Обозначение	Описание
0 — 1	R0_0	Для всех полей: 0b00 — приоритет каналов запроса инструкций; 0b01 — приоритет каналов запроса данных; 0b10 — приоритет канала DTC; 0b11 — приоритет канала AMBA.
4 — 5	R1_0	
6 — 7	R1_1	
8 — 9	R2_0	
10 — 11	R2_1	
12 — 13	R3_0	
14 — 15	R3_1	
16 — 17	R4	
18 — 19	R5	
20 — 21	R6	
22 — 23	R7	
24 — 25	AMBA	
26 — 63		Зарезервировано

1.2.2.4.5 Регистр DTC_CTRL управления контроллером транзакций данных DTC

Регистр управления контроллером транзакций данных DTC доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра DTC_CTRL

№№ битов	Обозначение	Описание
0 — 9	STEP	Шаг адреса для перемещения блоков (в байтах)
10 — 15	Noblk	Количество перемещаемых блоков
16 — 25	Nob	Размер перемещаемого блока
26		Зарезервировано
27 — 28	El_size	Размер элемента в перемещаемом блоке (0b00 — 1 байт, 0b01 — 2 байта, 0b10 — 4 байта)
29	AM	Режим доступа: 0 — по программному запросу, 1 — по прерыванию
30	DT	Получатель данных: 0 — память и внутренний регистр, 1 — внутренний регистр (регистр обновляется каждую транзакцию)
31	RUN	Начать транзакцию, если DTC_CTRL(AM) = 0, то автоматически очищается после начала транзакции и становится неактивным до завершения
32 — 63		Зарезервировано

1.2.2.4.6 Регистр DTC_ST состояния DTC

Регистр состояния DTC предназначен для мониторинга состояния DTC и доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Прерывания от DTC происходят при выставлении битов EM_ERR, TEC, TC в 1.

Структура регистра DTC_ST

№№ битов	Обозначение	Описание
0	EM_ERR	Ошибка контроллера внешней памяти при транзакции

1	ТЕС	Транзакция элемента завершена (зависит от установок в DTC_CTRL). Если получатель данных — память, то ТЕС не выставляется. Если получатель регистр — то необходимо очищать данный бит
2	ТС	Все транзакции завершены
3	ТИР	Идёт выполнение транзакции
4 — 63		Зарезервировано

1.2.2.4.7 Регистр DTC_IMASKx маски x DTC

Регистр DTC_IMASKx доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Для разрешения работы DTC по прерыванию с определёнными номерами необходимо выставить в 1 соответствующие биты данного регистра.

Структура регистра DTC_IMASKx

№№ битов	Обозначение	Описание
0 — 46		Номера прерываний
47 — 63		Зарезервировано

1.2.2.4.8 Регистр DTC_DATA приёма данных DTC

Регистр приёма данных DTC доступен как для чтения, так и для записи.

Структура регистра DTC_DATA

№№ битов	Обозначение	Описание
0 — 63		Принятые данные

1.2.2.4.9 Регистр DTC_S_ADDR адреса источника данных для DTC

Регистр DTC_S_ADDR доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра DTC_S_ADDR

№№ битов	Обозначение	Описание
0 — 31		Адрес источника данных
32 — 63		Зарезервировано

1.2.2.4.10 Регистр DTC_D_ADDR адреса приёмника данных для DTC

Регистр DTC_D_ADDR доступен как для чтения, так и для записи. Начальное значение регистра равно нулю.

Структура регистра DTC_DS_ADDR

№№ битов	Обозначение	Описание
0 — 31		Адрес приёмника данных
32 — 63		Зарезервировано

1.2.3 Коммутатор

Коммутатор используется для обмена результатами команд между командами внутри одного параграфа.

Параграф — группа предложений, которая записана последовательно, имеет один вход и один выход.

Предложение — группа информационно связанных операций.

Все команды процессора внутри одного параграфа условно нумеруются, начиная с нуля.

Ссылка на результат текстуально предшествующей команды записывается как @ N . N вычисляется согласно следующей формуле

$$N = N_{req} - N_{res},$$

где N_{req} — номер команды, запрашивающей результат текстуально предшествующей команды, N_{res} — номер текстуально предшествующей команды, результат которой запрашивается.

Максимальное количество результатов текстуально предшествующих команд, которое может быть сохранено в коммутаторе, — 63.

Ссылка на результат текстуально предшествующей команды, которая по определению НЕ возвращает результат, приведет к ошибке на этапе компиляции.

1.3 Основные понятия языка

1.3.1 Комментарии

В ассемблере поддерживаются следующие типы комментариев:

- однострочный комментарий

данный тип комментария начинается с ';' или '//' и заканчивается концом строки

- многострочный комментарий

данный тип комментария начинается с '/*' и заканчивается '*/', т. е. все, что находится между этими ограничителями игнорируется; вложенные комментарии не допускаются.

1.3.2 Константы

В ассемблере поддерживаются числовые и символьные (литеральные) константы.

1.3.2.1 Числовые константы

В ассемблере возможны следующие варианты представления числовых констант:

1. В виде шестнадцатеричного числа

Такое число начинается с префикса '0x' или '0X', за которым следует одна или более шестнадцатеричных цифр '0123456789abcdefABCDEF'. Для изменения знака используется префиксный оператор '-'.

2. В виде восьмеричного числа

Такое число начинается с нулевой цифры, за которой следует одна или более восьмеричных цифр '01234567'. Для изменения знака используется префиксный оператор '-'.

3. В виде двоичного числа

Такое число начинается с префикса '0b' или '0B', за которым следует одна или более двоичных цифр '01'. Для изменения знака используется префиксный оператор '-'.

4. В виде целого десятичного числа

Такое число начинается с ненулевой цифры, за которой следует одна или более десятичных цифр '0123456789'. Для изменения знака используется префиксный оператор '-'.

5. В виде вещественного десятичного числа с плавающей точкой, записанного в следующем формате:

- а) начинается с префикса '0f' или '0F',
- б) далее опционально следует знак числа '+' или '-',
- в) далее опционально следует целая часть числа, состоящая из нуля или более десятичных цифр,
- г) далее опционально следует дробная часть числа, начинающаяся с символа точки '.' и состоящая из нуля или более десятичных цифр,
- д) далее опционально следует экспоненциальная часть числа, состоящая из:
 - 'e' или 'E'
 - знака '+' или '-' экспоненциальной части (опционально)
 - одной или более десятичных цифр.

По крайней мере одна из целой или дробной частей должна быть задана.

1.3.2.2 Символьные (литеральные) константы

В ассемблере возможны следующие варианты представления символьных (литеральных) констант:

1. В виде строки (последовательности литералов)

Строковые константы (последовательность литералов) записываются в двойных кавычках. Они могут содержать любые возможные символы (литералы), а также следующие escape-последовательности:

- \b — забой (backspace); ASCII код в восьмеричной системе счисления 010.
- \f — новая страница (FormFeed); ASCII код в восьмеричной системе счисления 014.
- \n — перевод строки (newline); ASCII код в восьмеричной системе счисления 012.

- `\r` — возврат каретки (carriage-Return); ASCII код в восьмеричной системе счисления 015.
- `\t` — горизонтальная табуляция (horizontal Tab); ASCII код в восьмеричной системе счисления 011.
- `\ oct-digit oct-digit oct-digit` — код символа в восьмеричной системе счисления. Код символа состоит из 3-х восьмеричных цифр. Если заданное число превышает максимально возможное восьми разрядное значение, будут использованы только младшие восемь разрядов.
- `\x hex-digit hex-digit` — код символа в шестнадцатеричной системе счисления. Код символа состоит из 2-х шестнадцатеричных цифр. Регистр литерала 'x' не имеет значения. Если ни одна из двух шестнадцатеричных цифр не задана, используется значение ноль.
- `\\` — соответствует литералу `'\'`.
- `\"` — соответствует литералу `'\"'`.
- `\anything-else` — соответствует любому символу, за исключением выше перечисленных.

2. В виде одиночного символа (литерала)

Одиночный символ (литерал) может быть представлен в виде одинарной кавычки «`'`», непосредственно за которой следует необходимый символ (литерал) или escape-последовательность. Поэтому для представления символа (литерала) «`\`», необходимо написать «`'\'`», где первый символ (литерал) «`\`» экранирует второй «`\`». Символ перевода строки, непосредственно следующий за «`'`», интерпретируется как символ (литерал) «`\n`» и не является окончанием выражения. Значением символьной (литеральной) константы в целочисленном выражении является машинный код, размером в один байт, символа (литерала). `as` использует ASCII кодировку символов (литералов): `'A` имеет целочисленное значение 65, `'B` имеет целочисленное значение 66, и так далее.

1.3.3 Секции

Не вдаваясь в подробности, секция представляет собой непрерывный диапазон адресов, все данные которого трактуются одинаково для некоторых определенных действий.

В результате компиляции исходного кода части программы, ассемблер создает объектный файл, предполагая, что данная часть программы располагается с нулевого адреса. Сборка самой исполняемой программы осуществляется компоновщиком из одного или нескольких объектных файлов, созданных ассемблером, в результате чего каждому объектному файлу присваивается конечный адрес таким образом, что ни один объектный файл не перекрывается другим.

Во время компоновки программы блоки байтов, как единое целое, перемещаются на те адреса, которые они будут иметь во время выполнения программы; их длина и порядок байтов в них не изменяются. Именно такой блок байтов называется секцией, а процедура назначения адресов этим секциям — перемещением (relocation). Помимо назначения секциям адресов времени выполнения и их перемещения, на этапе компоновки приводятся в соответствие значения символов объектного файла, так чтобы все ссылки на эти символы имели верные адреса времени выполнения.

Объектный файл, созданный ассемблером, включает в себя, по крайней мере, три секции, каждая из которых может быть пустой:

1. секция `.text`. В этой секции располагаются исполняемые инструкции программы.
2. секция `.data`. В этой секции располагаются начальные данные программы.
3. секция `.bss`. Данная секция содержит байты с нулевыми значениями перед началом выполнения программы. Она используется для хранения неинициализированных переменных или общего блока памяти данных.

Секции `.text` и `.data` присутствуют в объектном файле в не зависимости от того содержат они какие-либо директивы или нет.

Для того, чтобы сообщить компоновщику какие данные изменяются во время перераспределения памяти (перемещения секций), а также согласно каким правилам они изменяются, ассемблер записывает в объектный файл всю необходимую информацию в отдельные секции (как правило для секции `.text` в секцию `.rel.text`, для секции `.data` в секцию `.rel.data`).

Кроме секций `.text`, `.data`, `.bss` возможно использование абсолютной секции. При компоновке программы адреса в абсолютной секции не изменяются.

Помимо выше перечисленных секций существует также неопределённая секция. Любой символ, на который имеется ссылка и, который не был определен, на этапе ассемблирования относится к неопределенной секции. Общий (совместно используемый) символ, который адресует именованный общий блок, также на этапе ассемблирования относит-

ся к неопределенной секции. Значение атрибута связывания любого неопределённого символа по умолчанию равно «GLOBAL».

1.3.3.1 Подсекции

Ассемблированные байты по умолчанию размещаются в двух секциях: `.text` и `.data`. Для упорядочения различных групп данных внутри секций `.text` или `.data` в генерируемом объектном файле используются подсекции. Внутри каждой секции могут находиться пронумерованные подсекции, начиная с нуля. Объекты, ассемблированные в одну и ту же подсекцию, в объектном файле располагаются вместе с другими объектами той же подсекции.

Подсекции располагаются в объектном файле в порядке возрастания их номеров. Информация о подсекциях в объектном файле не сохраняется.

Для того, чтобы указать в какую подсекцию ассемблировать ниже следующие инструкции, необходимо указать номер подсекции в директиве `.text/.data`. Если в исходном коде программы подсекции не используются, то все инструкции ассемблируются в подсекцию с номером 0.

Каждая секция имеет «счетчик текущего места», увеличивающийся на один при ассемблировании каждого нового байта в эту секцию. Поскольку подсекции являются просто удобством и ограничены использованием только внутри ассемблера, не существует «счетчиков текущего места» подсекций. «Счетчик текущего места» секции, в которую в данный момент ассемблируются инструкции, называется активным счетчиком места.

Секция `bss` используется как место для хранения глобальных переменных. Для этой секции, используя директиву `.lcomm`, можно выделить адресное пространство без указания какие данные будут загружены в нее до исполнения программы. Во время начала выполнения программы содержимое секции `.bss` заполняется нулями.

1.3.4 Символы

Символы (идентификаторы, переменные и т. п.) используются в ассемблере для именования различных сущностей.

1.3.4.1 Система имён символов

Система имён в ассемблере построена по следующему принципу — имена могут состоять только из прописных и печатных букв латинского алфавита, цифр, символа подчёркивания («_») и символа точки («.»), при этом имя не может начинаться с цифры. Учитывается регистр букв в имени.

1.3.4.2 Метки

Метка определяется как символ, за которым следует двоеточие «:». Этот символ представляет текущее значение активного счетчика места в зависимости от текущей секции (.text, .data, .bss). Переопределение меток в ассемблере не допускается.

1.3.4.3 Символы с абсолютным значением

Данные символы могут быть определены при помощи следующих директив ассемблера: .set, .eqv, .equ, .equiv. Значения данных символов никогда не изменяются компоновщиком. В общем случае, символ, определённый при помощи перечисленных директив ассемблера, может не иметь абсолютного значения.

1.3.4.4 Атрибуты символов

Каждый символ помимо имени имеет атрибуты «Значение», «Тип/Связывание», «Размер», а также атрибут принадлежности символа к какой-либо секции. При использовании символа без его определения все его атрибуты имеют нулевые значения, а сам символ относится к неопределённой секции.

Значение символа является 32-х разрядным. Для символов, адресующих местоположение в секциях .text, .data, .bss, а также абсолютной, значением является смещение в адресах относительно начального положения секции до метки. В процессе компоновки программы значения таких символов для секций .text, .data, .bss изменяются, поскольку изменяются начальные положения данных секций. Значения абсолютных символов в процессе компоновки не изменяются.

Атрибут символа «Тип/Связывание» определяет тип и видимость символа компоновщиком, а также поведение компоновщика в процессе изменения значения символа при перемещении секций. Для установки значения типа данного атрибута используется ди-

ректива ассемблера `.type`, а для значения связывания — директивы ассемблера `.local`, `.global`, `.weak`.

Атрибут символа «Размер» на этапе ассемблирования по умолчанию всегда устанавливается равным нулю. Значение данного атрибута может быть изменено при помощи директивы ассемблера `.size`.

Атрибут принадлежности символа к какой-либо секции устанавливается ассемблером автоматически, в зависимости от текущей секции ассемблирования. Другими словами данный атрибут указывает в какой секции определён символ.

1.3.5 Выражения

Выражение определяет адрес или числовое значение. Результат выражения должен быть абсолютным числом или смещением в определенной секции.

1.3.5.1 Пустые выражения

Пустое выражение не имеет значения: это просто пропуск. В случае отсутствия выражения в том месте исходного кода, где оно необходимо, ассемблер использует значение ноль.

1.3.5.2 Целочисленные выражения

Целочисленное выражение — это один или более аргументов, разделенных операторами.

1.3.5.3 Аргументы

В качестве аргументов выражения могут выступать символы (идентификаторы), числа или подвыражения.

Значением символа в какой-либо секции `secName` является смещение относительно начала этой секции. В качестве секции `secName` могут выступать секции `.text`, `.data`, `.bss`, а также абсолютная (`*ABS*`) и неопределённая (`*UND*`). Значение представляет собой 32-х разрядное целое число со знаком в двоичном дополнительном коде.

Числа, как правило, являются целыми. Если выражение состоит из одного аргумента, который представляет из себя число с плавающей точкой, то результатом вычисления выражения будет именно это число без каких-либо преобразований. Если выражение

состоит из нескольких аргументов, разделенных операторами, либо из префиксного оператора, за которым следует один единственный аргумент, то при вычислении выражения значение аргумента, которое представляет из себя число с плавающей точкой, будет заменено на нулевое значение, а также выведено соответствующее предупреждение.

Подвыражения представляют из себя такие же выражения, заключенные в круглые скобки «()», либо префиксный оператор, за которым следует аргумент.

1.3.5.4 Операторы

Операторы представляют из себя арифметические функции и делятся на префиксные и инфиксные.

Префиксные операторы являются одноаргументными. Аргумент должен быть абсолютным. Доступны следующие префиксные операторы:

«-» Отрицание. Двоичное дополнительное отрицание.

«~» Дополнение. Побитовое отрицание.

«!» Логическое НЕ. Возвращается 1, если аргумент не нулевой, и 0, в противном случае.

Инфиксные операторы являются двухаргументными. Данные операторы имеют приоритет, который определяет очередность их выполнения. Операции с равным приоритетом выполняются слева на право. Аргументы всех инфиксных операторов, за исключением операторов «+» и «-» должны быть абсолютными. Доступны следующие инфиксные операторы в порядке снижения приоритета:

1. «*» Умножение.

«/» Целочисленное деление.

«%» Остаток (взятие остатка от целочисленного деления).

«<<» Сдвиг влево.

«>>» Сдвиг вправо.

2. «|» Побитовое Или.

«&» Побитовое И.

«^» Побитовое Исключающее Или

«!» Побитовое Или-Не

3. «+» Сложение. Если один из аргументов абсолютный, то результат относится к секции другого аргумента. Сложение двух аргументов, определенных относительно различных секций недопустимо.

«-» Вычитание. Если правый (второй) аргумент абсолютен, то результат относится к секции левого (первого) аргумента. Если оба аргумента определены относительно одной и той же секции, то результат абсолютен. Вычитание двух аргументов, определенных относительно различных секций недопустимо.

«==» Сравнение на равенство.

«!=» или «<>» Сравнение на неравенство.

«<» Сравнение на меньше, чем.

«>» Сравнение на больше, чем.

«>=» Сравнение на больше, чем, либо равно.

«<=» Сравнение на меньше, чем, либо равно.

В результате выполнения какой-либо операции сравнения, возвращается -1, в случае если результат истинный, и 0, если ложный. Операции сравнения интерпретируют аргументы как знаковые.

4. «&&» Логическое И.

«||» Логическое ИЛИ.

Данные логические операции могут быть использованы для объединения результатов двух подвыражений. В результате выполнения какой-либо логической операции, возвращается 1, в случае если результат истинный, и 0, если ложный.

1.4 Система команд ассемблера

Команда мультиклеточного процессора, как и любого другого, в общем случае, представляет собой закодированную по некоторому набору правил инструкцию процессора на выполнение какой-либо операции над некоторым набором операндов.

В мультиклеточном процессоре существуют короткие команды, размерностью 32 бита, и длинные, размерностью 64 бита. В команде мультиклеточного процессора закодированы:

- код операции, определяющий действие, которое необходимо выполнить процессору;
- суффикс операции, определяющий правила формирования операндов операции;
- тип операции, определяющий размер операндов и интерпретацию их значений;
- набор данных (значение ссылки на результат команды, значение ссылки на регистр, непосредственное значение) необходимый для формирования операндов;
- прочие данные, необходимые для выполнения операции или действий, связанных с данной операцией.

1.4.1 Условные обозначения

$@S, @S1, @S2$ — обозначает ссылку на результат команды, который сохранен в коммутаторе, относительно текущей команды (см. раздел «Коммутатор»).

$\#R$ — обозначает ссылку на регистр (см. раздел «Регистры»).

$\#GPR$ — обозначает ссылку на регистр общего назначения (см. раздел «Регистры общего назначения»).

$\#IR$ — обозначает ссылку на индексный регистр (см. раздел «Регистры индексные»).

$\#IR_{index}, \#IR_{mask}, \#IR_{base}$ — обозначает индекс, маску и базу соответственно индексного регистра IR .

$\#CR$ — обозначает ссылку на управляющий регистр (см. раздел «Регистры управляющие»).

$\#SR$ — обозначает ссылку на системный регистр (см. раздел «Регистры системные»).

$V32$ — обозначает непосредственное значение размером слово (32 бита)

ARG , $ARG1$, $ARG2$ — общее обозначение аргументов команды

$DM(ADDR)$ — общее обозначение обращения к памяти данных по адресу $ADDR$

$EXPR$ — общее обозначение выражения

$RES(EXPR)$ — общее обозначение результата вычисления выражения

$(|b|l|q)$ — общее обозначение выбора одного из возможных значений, т. е. либо ничего, либо b , либо l , либо q

1.4.2 Типы операций

Мультиклеточный процессор, в общем случае, может выполнять операции над следующими типами операндов:

- знаковый / беззнаковый целый, размерностью один байт (8 бит);
- знаковый / беззнаковый целый, размерностью два байта (16 бит);
- знаковый / беззнаковый целый, размерностью четыре байта (32 бита);
- знаковый / беззнаковый целый, размерностью восемь байтов (64 бита);
- знаковый вещественный одинарной точности, размерностью четыре байта (32 бита);
- знаковый вещественный двойной точности, размерностью восемь байтов (64 бита);
- знаковый / беззнаковый упакованный, размерностью восемь байтов (64 бита), каждые два байта которого (с 0 по 15 бит, с 16 по 31 бит, 32 по 47 бит, с 48 по 63 бит) представляют собой знаковое / беззнаковое целое число
- знаковый / беззнаковый упакованный, размерностью восемь байтов (64 бита), старшие четыре байта (с 32 по 63 биты) представляют собой старшую часть, а младшие четыре байта (с 0 по 31 биты) представляют собой младшую часть (тип старшей и младшей частей — знаковый / беззнаковый целый 32-х разрядный);
- знаковый упакованный, размерностью восемь байтов (64 бита), старшие четыре байта (с 32 по 63 биты) представляют собой старшую часть, а младшие четы-

ре байта (с 0 по 31 биты) представляют собой младшую часть (тип старшей и младшей частей — вещественный одинарной точности);

- знаковый комплексный, размерностью восемь байтов (64 бита), старшие четыре байта (с 32 по 63 биты) представляют собой реальную часть, а младшие четыре байта (с 0 по 31 биты) представляют собой мнимую часть (тип реальной и мнимой частей — вещественный одинарной точности);

В ассемблере зависимость от кода типа данных проявляется в мнемонике команды. Мнемоника команды состоит из двух частей корня, соответствующего коду операции, и суффикса, соответствующего типу операции.

Следует заметить, что не каждая команда ядра поддерживает все перечисленные типы данных.

В таблице 1.26 показаны мнемоники суффиксов типов операций.

Таблица 1.26: Мнемоника типов операций

Тип операции	Беззнаковый	Знаковый
Целый, размерностью один байт	b	sb
Целый, размерностью два байта	s	ss
Целый, размерностью четыре байта	l	sl
Целый, размерностью восемь байтов	q	sq
Вещественный одинарной точности, размерностью четыре байта	-	f
Вещественный двойной точности, размерностью восемь байтов	-	d
16-ти разрядный целый упакованный, размерностью восемь байтов	ps	pss
32-х разрядный целый упакованный, размерностью восемь байтов	pl	psl
Вещественный одинарной точности упакованный, размерностью восемь байтов	-	p
Вещественный одинарной точности комплексный, размерностью восемь байтов	-	c

1.4.3 Общий принцип построения команд в ассемблере

Ассемблером поддерживается два варианта синтаксиса команд мультиклеточного процессора:

- синтаксис команд версии 1, который изображён на рисунке 1.3
- синтаксис команд версии 2, который изображён на рисунке 1.4

Для процессора МСр042R100102 по умолчанию используется синтаксис команд версии 2. Синтаксис команд версии 1 используется по умолчанию для процессора МСр0411100101. Используемый в единице трансляции синтаксис команд может быть определён при помощи опции ассемблера **--syntax** или директивы ассемблера **.syntax**.

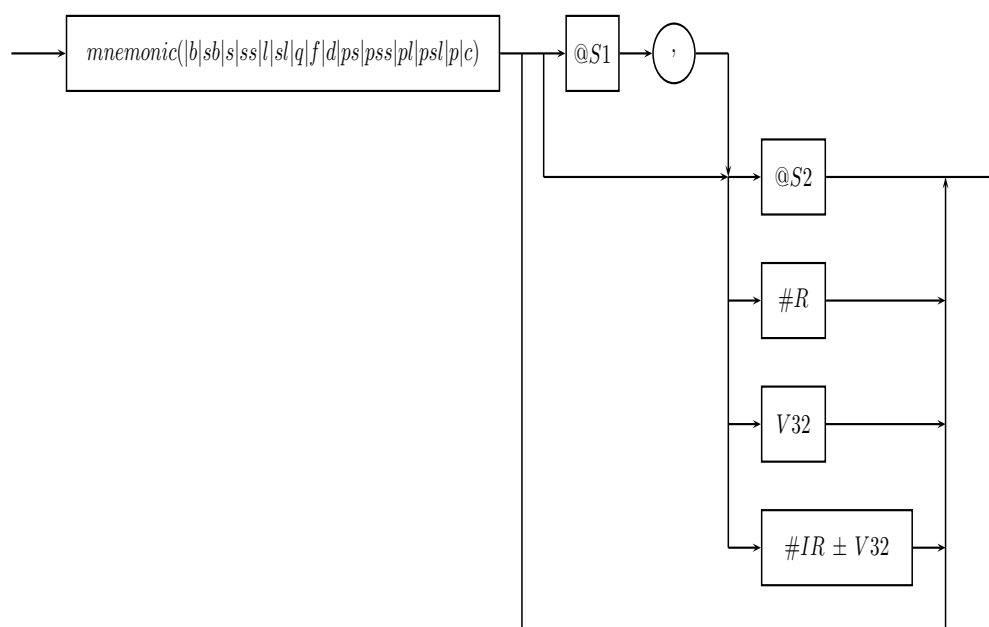


Рис. 1.3: Синтаксис команд версии 1 мультиклеточного процессора

1.4.3.1 Общие правила формирования аргументов команд и их интерпретация

В общем случае команды могут быть двухаргументными, одноаргументными и без аргументов.

Согласно синтаксическому описанию (рис. 1.3 и 1.4), в качестве первого аргумента двухаргументной команды всегда используется результат выполнения одной из 63-х предыдущих команд, содержащийся в коммутаторе и заданный ссылкой на это значение.

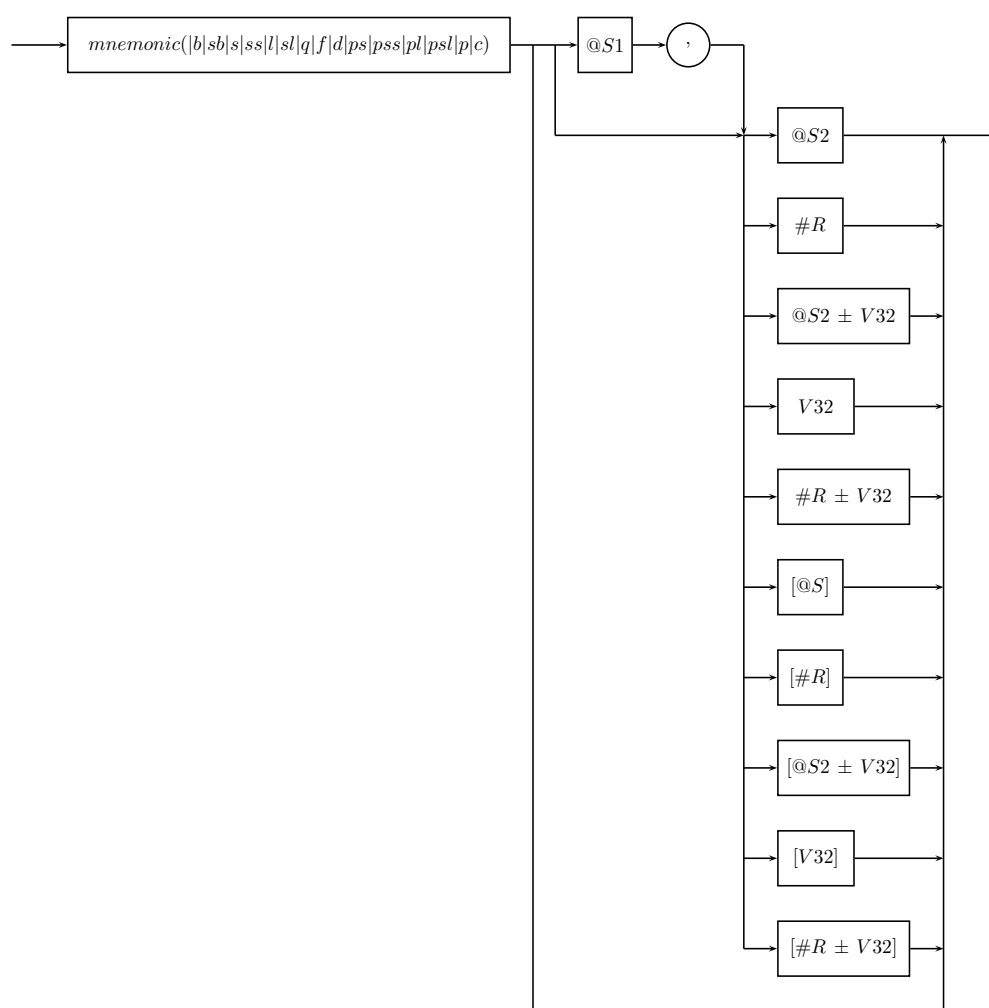


Рис. 1.4: Синтаксис команд версии 2 мультиклеточного процессора

Второй аргумент двухаргументной команды или аргумент одноаргументной команды может быть задан с использованием одного из следующих вариантов:

- для синтаксиса версии 1
 - используется результат выполнения одной из 63-х предыдущих команд, содержащийся в коммутаторе и заданный ссылкой на это значение
 - используется значение регистра общего назначения ($\#GPR$), значение управляющего регистра ($\#CR$) или значение системного регистра ($\#SR$)
 - используется значение, считанное из памяти по адресу, вычисляемому на основании значения индексного регистра ($\#IR$), согласно формуле

$$\#IR_{base} + \#IR_{index}$$

- используется непосредственное значение, заданное в командном слове ($V32$)

в зависимости от типа команды

- используется значение, считанное из памяти по адресу, вычисляемому на основании значения индексного регистра ($\#IR$) и непосредственного значения ($V32$), заданного в командном слове, согласно формуле

$$\#IR_{base} + \#IR_{index} \pm V32$$

- для синтаксиса версии 2

- используется результат выполнения одной из 63-х предыдущих команд, содержащийся в коммутаторе и заданный ссылкой на это значение
- используется значение регистра любого типа ($\#GPR$, $\#CR$, $\#IR$, $\#SR$) в зависимости от типа команды
- используется значение, вычисляемое на основании результата выполнения одной из 63-х предыдущих команд, содержащегося в коммутаторе и заданного ссылкой на это значение, и непосредственного значения ($V32$), заданного в командном слове, согласно формуле

$$@S \pm V32$$

- используется непосредственное значение ($V32$), заданное в командном слове
- используется значение, вычисляемое на основании значения регистра любого типа ($\#GPR$, $\#CR$, $\#IR$, $\#SR$) и непосредственного значения ($V32$), заданного в командном слове, согласно формуле

$$\#R \pm V32$$

- используется значение, считанное из памяти по адресу равному результату выполнения одной из 63-х предыдущих команд, содержащегося в коммутаторе и заданного ссылкой на это значение
- используется значение, считанное из памяти по адресу, вычисляемому на основании значения регистра отличного от индексного ($\#GPR$, $\#CR$, $\#SR$)
- используется значение, считанное из памяти данных по адресу, вычисляемому на основании значения индексного регистра ($\#IR$) согласно формуле

$$\#IR_{base} + \#IR_{index}$$

- используется значение, считанное из памяти по адресу, вычисляемому на основании значения результата выполнения одной из 63-х предыдущих команд, содержащегося в коммутаторе и заданного ссылкой на это значение, и непосредственного значения ($V32$), заданного в командном слове, согласно формуле

$$@S \pm V32$$

- используется значение, считанное из памяти по адресу равному непосредственному значению ($V32$), заданного в командном слове
- используется значение, считанное из памяти по адресу, вычисляемому на основании значения регистра отличного от индексного ($\#GPR$, $\#CR$, $\#SR$) и непосредственного значения ($V32$), заданного в командном слове, согласно формуле

$$\#R \pm V32$$

- используется значение, считанное из памяти данных по адресу, вычисляемому на основании значения индексного регистра ($\#IR$) и непосредственного значения ($V32$), заданного в командном слове, согласно формуле

$$\#IR_{base} + \#IR_{index} \pm V32$$

Исключения из правил формирования аргументов команд и их интерпретации смотри в описании конкретной команды.

1.4.3.2 Правила формирования результатов команд

В общем случае команды формируют результат, сохраняемый в коммутаторе.

Результат нагружен значениями флагов:

Таблица 1.27: Флаги результата.

Флаг	Перевод	Описание
ZF (Zero Flag)	Флаг нуля	Устанавливается в случае равенства нулю всех разрядов результата команды
OF (Overflow Flag)	Флаг переполнения	Устанавливается в случае потери значащего бита

<i>CF</i> (Carry Flag)	Флаг переноса	Устанавливается в случае выхода за разрядную сетку результата целочисленной команды
<i>IF</i> (Inexact Flag)	Флаг потери точности	Устанавливается в случае потери точности результата вещественной команды
<i>SF</i> (Sign Flag)	Флаг знака	Устанавливается равным значению старшего (знакового) разряда результата команды

Флаги результата устанавливаются в зависимости от типа операции и могут иметь значение не для каждой команды. Более подробную информацию можно найти в описании конкретной команды.

1.4.4 Описание команд

1.4.4.1 abs (ABSolute value)

Абсолютное значение

abs ARG

Назначение: операция вычисления абсолютного значения аргумента

Синтаксис:

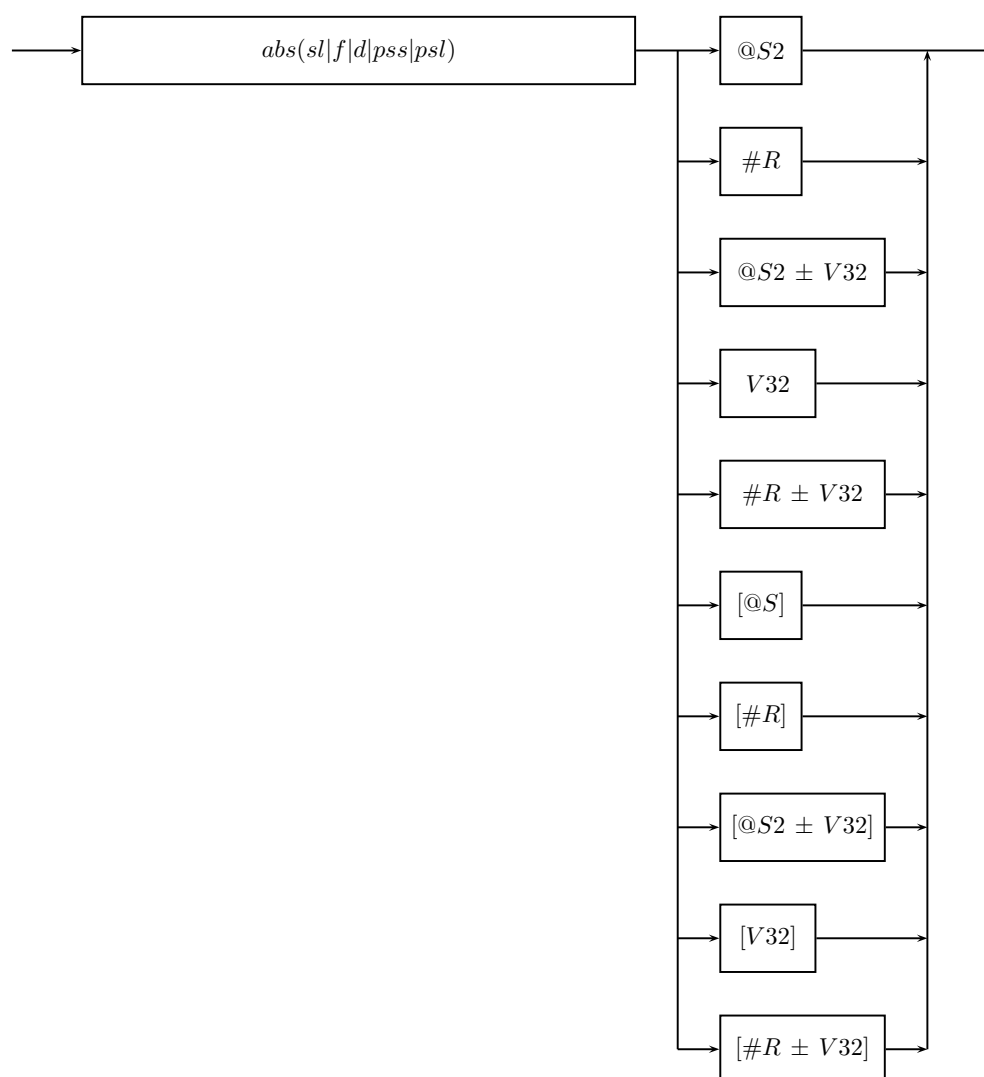


Рис. 1.5: Синтаксическое описание команды *abs*

Наличие результата: да

Алгоритм работы:

- вычислить абсолютное значение аргумента ARG ;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операции **signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типа операции **signed long**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;
 - флаг переполнения $OF = 1$, если аргумент равен минимальному значению ($ARG = 0x80000000$), иначе – $OF = 0$;
 - флаг переноса $CF = 0$;
 - флаг знака $SF = 0$.
- для типа операции **float, double**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг потери точности $IF = 0$;
 - флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.
- для типа операции **packed signed short, packed signed long**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг переноса $CF = 0$;
 - флаг знака $SF = 0$.

Применение: команда `abs` используется для вычисления абсолютного значения числа или чисел упакованного числа. Результат вычисления абсолютного значения минимального возможного целого числа, в зависимости от типа операции, выходит за границы разрядной сетки, о чём сигнализирует флаг переполнения (*OF*).

Пример:

```
1  .text
2
3  A:
4      getsl 0x80000000
5      abssl @1
6      abssl 0xF0010203
7      absf 0f-12.85
8  complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №3 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №4 командой `getsl` помещается в коммутатор 32-х разрядное целое число (константа `0x80000000`);
- в строке №5 командой `abssl` вычисляется абсолютное значение результата выполнения предшествующей команды: `@1` — результат выполнения команды извлечения в строке №4; аргумент команды `abssl` согласно суффиксу `sl` интерпретируются как целое знаковое размерностью 4 байта; результат выполнения команды интерпретируются также как целое знаковое размерностью 4 байта и помещается в коммутатор с установленным флагом переполнения (*OF*), так как значение `0x80000000` выходит за границы разрядной сетки знакового целого числа размерностью 4 байта;
- в строках №№6,7 показаны другие варианты использования команды `abs`;
- в строке №8 командой `complete` завершается текущий параграф.

1.4.4.2 adc (ADdition with Carry)

Сложение с переносом

adc ARG1, ARG2

Назначение: операция целочисленного сложения с учётом флага переноса (*CF*) результата ранее вычисленной команды

Синтаксис:

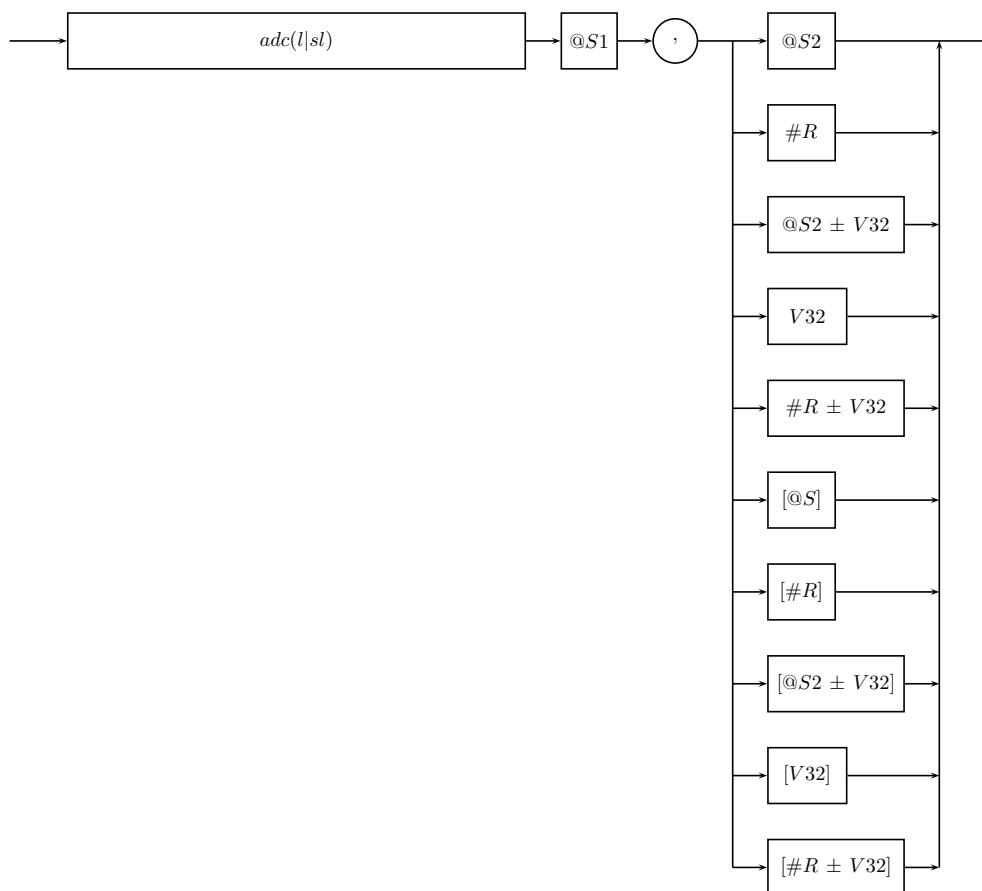


Рис. 1.6: Синтаксическое описание команды *adc*

Наличие результата: да

Алгоритм работы:

- выполнить сложение значения флага переноса (*CF*) аргумента *ARG1* со значением аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

- для типа операции **long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = CF$;

флаг переноса $CF = 1$, если расширенный бит результата равен единице, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типа операции **signed long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `adc` используется при сложении длинных целых чисел. В отличие от одноимённой команды большинства других процессоров, команда `adc` мультиклеточного процессора складывает значение второго аргумента с единицей, если флаг переноса (CF) первого аргумента установлен, в противном случае в качестве результата возвращается просто значение второго аргумента.

Пример:

```
1  /*
2   * Программа сложения двух 64-х разрядных чисел
3   */
4
5  .data
6
7  A:
8      .long 0x00010203 , 0xAABBCCDD
```

```
9  B:
10      .long 0x04050607 , 0xEEFF0908
11  C:
12      .long 0x00000000 , 0x00000000
13
14  .text
15
16  D:
17      rdl A
18      rdl A + 4
19      rdl B
20      rdl B + 4
21      addl @3, @1; результат 0x99BAD5E5, CF == 1
22      addl @5, @3; результат 0x0406080A
23      adcl @2, @1; результат 0x0406080B
24      wrd @1, C
25      wrd @4, C + 4
26  complete
```

Пояснения к примеру

- в строке №5 директивой ассемблера .data устанавливается текущая секция ассемблирования — секция инициализированных данных data;
- в строках №№7,9,11 объявляются символы (идентификаторы) *A*, *B*, *C*, которые являются метками в текущей секции ассемблирования (data), и инициализируется текущими значениями адреса ассемблирования;
- в строках №№8,10,12 директивой ассемблера .long в текущую секцию ассемблирования по текущему адресу ассемблирования записываются 32-х разрядные числа: 0x00010203, 0xAABBCCDD, 0x04050607, 0xEEFF0908, 0x00000000, 0x00000000;
- в строке №14 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;
- в строке №16 объявляется символ (идентификатор) *D*, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строках №№17,18,19,20 командами rdl читаются из памяти данных четыре

- 32-х разрядных целых беззнаковых числа по адресам A , $A + 4$, B , $B + 4$ соответственно и помещаются в коммутатор;
- в строке №21 командой `addl` складываются результаты выполнения двух предшествующих команд: `@3` — результат выполнения команды чтения в строке №18, `@1` — результат выполнения команды чтения в строке №20; оба аргумента команды `addl` согласно суффиксу `l` интерпретируются как 32-х разрядные целые беззнаковые числа; результат выполнения команды также интерпретируется как 32-х разрядное целое беззнаковое число и помещается в коммутатор;
 - в строке №22 командой `addl` складываются результаты выполнения двух предшествующих команд: `@5` — результат выполнения команды чтения в строке №17, `@3` — результат выполнения команды чтения в строке №19; оба аргумента команды `addl` согласно суффиксу `l` интерпретируются как 32-х разрядные целые беззнаковые числа; результат выполнения команды также интерпретируется как 32-х разрядное целое беззнаковое число и помещается в коммутатор;
 - в строке №23 командой `addl` складываются результаты выполнения двух предшествующих команд: `@2` — результат выполнения команды сложения в строке №20 (фактически используется только значение флага переполнения *OF*), `@1` — результат выполнения команды сложения в строке №21; оба аргумента команды `addl` согласно суффиксу `l` интерпретируются как 32-х разрядные целые беззнаковые числа; результат выполнения команды также интерпретируется как 32-х разрядное целое беззнаковое число и помещается в коммутатор;
 - в строке №24 командой `wrl` осуществляется запись в память данных по адресу C результата выполнения предшествующей команды: `@1` — результат выполнения команды сложения с учётом флага переноса в строке №24;
 - в строке №25 командой `wrl` осуществляется запись в память данных по адресу $C + 4$ результата выполнения предшествующей команды: `@4` — результат выполнения команды сложения в строке №21;
 - в строке №26 командой `complete` завершается текущий параграф.

1.4.4.3 add (ADDITION)

Сложение

add ARG1, ARG2

Назначение: операция сложения двух аргументов

Синтаксис:

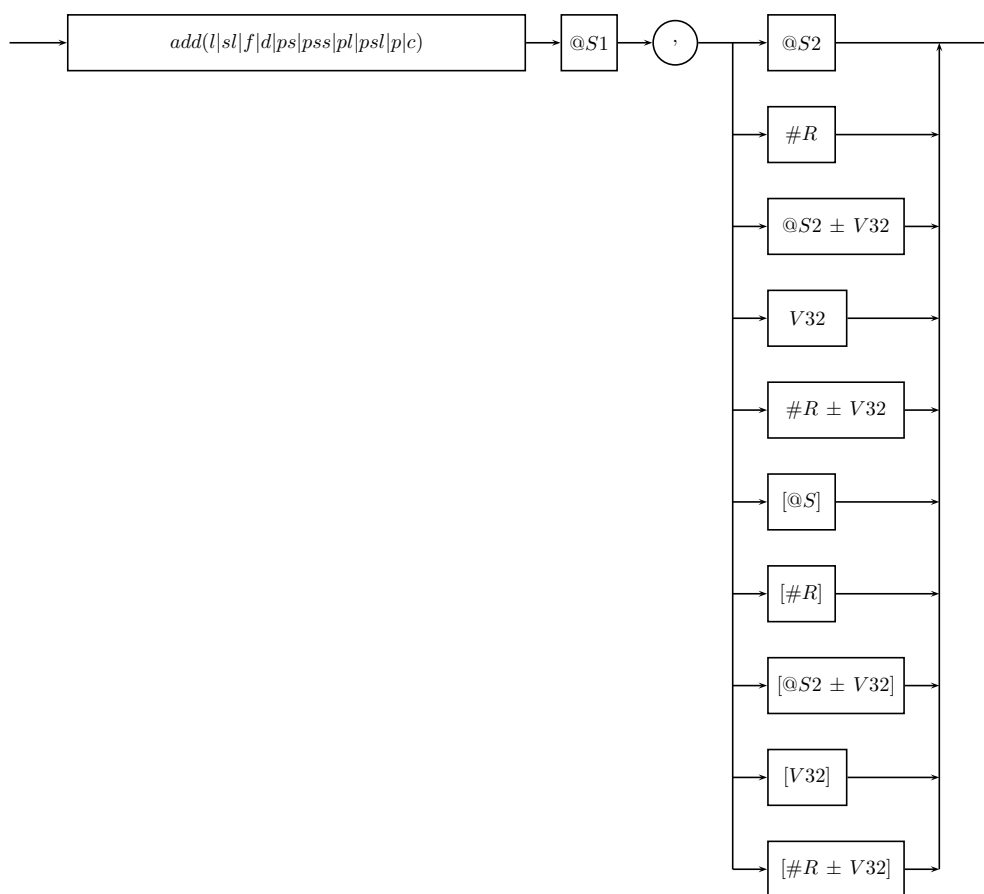


Рис. 1.7: Синтаксическое описание команды *add*

Наличие результата: да

Алгоритм работы:

- выполнить сложение $ARG1 + ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операции **long, signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типа операции **long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = CF$;

флаг переноса $CF = 1$, если расширенный бит результата равен единице, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типа операции **signed long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если флаг переноса CF не равен старшему биту результата и совпадают старшие биты аргументов, иначе – $OF = 0$;

флаг переноса $CF = 1$, если расширенный бит результата равен единице, а также старшие биты аргументов совпадают и равны единице, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

- для типов операций **float, double**:

флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе $-ZF = 0$;

флаг переполнения $OF = 1$, если все биты экспоненты результата равны единице или расширенный бит экспоненты результата равен единице, либо хотя бы один из аргументов равен бесконечности, иначе $-OF = 0$;

флаг потери точности $IF = 1$, если в результате произошла потеря точности вследствие выравнивания мантисс или округления до ближайшего чётного, либо результатом является NaN иначе $-IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе $-SF = 0$.

– для типов операций **packed float, complex float**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе $-ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе $-SF = 0$.

Применение: команда `add` используется для сложения двух операндов, значение которых интерпретируется согласно типу операции.

Пример:

```
1  .data
2
3  B:
4      .float \
5          0f12.8 , 0f-5.6 , \
6          0f-1.78 , 0f0.19
7
8  .text
9
10 A:
11     rdq B
12     rdq B + 8
13     addc @1, @2
14     wrq @1, B + 16
```

15 complete

Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) `B`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера `.float` в текущую секцию ассемблирования записывается четыре 32-х разрядных вещественных числа, начиная с текущего адреса ассемблирования (символ обратной косой черты `\` в конце строки используется для продолжения строки, т. е. строки №№4,5,6 логически являются одной строкой);
- в строке №8 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №10 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строках №№11,12 командами `rdq`, читаются из памяти данных два 64-х разрядных целых беззнаковых числа по адресам `B` и `B + 8` соответственно и помещаются в коммутатор;
- в строке №13 командой `addc` выполняется операция сложения результатов выполнения двух предшествующих команд: `@1` — результат выполнения команды чтения в строке №11, `@2` — результат выполнения команды чтения в строке №12; оба аргумента команды `addc` согласно суффиксу `s` интерпретируются как комплексные числа размерностью 64 бита, старшие 32 разряда которых представляют действительные части чисел, а младшие 32 разряда — мнимые; результат выполнения команды также интерпретируется как 64-х разрядное комплексное число и помещается в коммутатор;
- в строке №14 командой `wrg` осуществляется запись в память данных по адресу `B + 16` результата выполнения предшествующей команды: `@1` — результат выполнения команды сложения в строке №13;
- в строке №15 командой `complete` завершается текущий параграф.

1.4.4.4 and (AND)

Логического умножение

and ARG1 , ARG2

Назначение: операция логического умножения двух аргументов

Синтаксис:

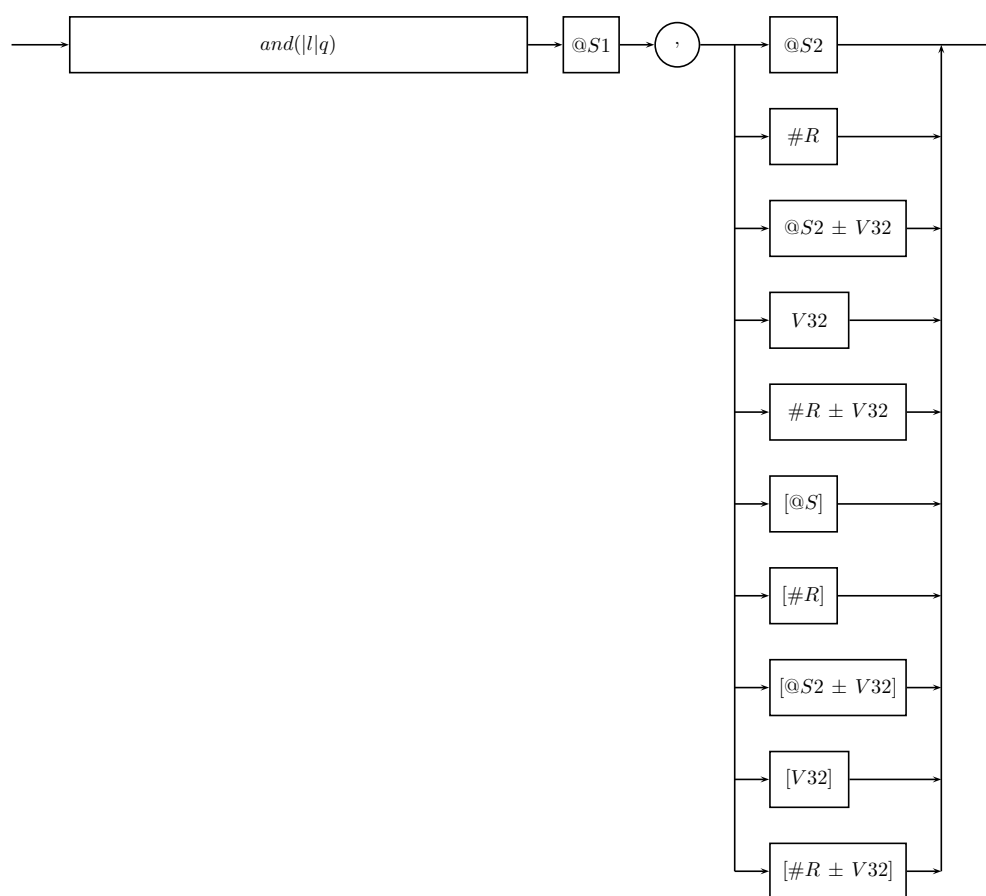


Рис. 1.8: Синтаксическое описание команды *and*

Наличие результата: да

Алгоритм работы:

- выполнить логическое умножение *ARG1 & ARG2*;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.5 bc (Bit Count)

Вычисление числа битов, равных единице

bc ARG

Назначение: команда вычисления числа битов, равных единице

Синтаксис:

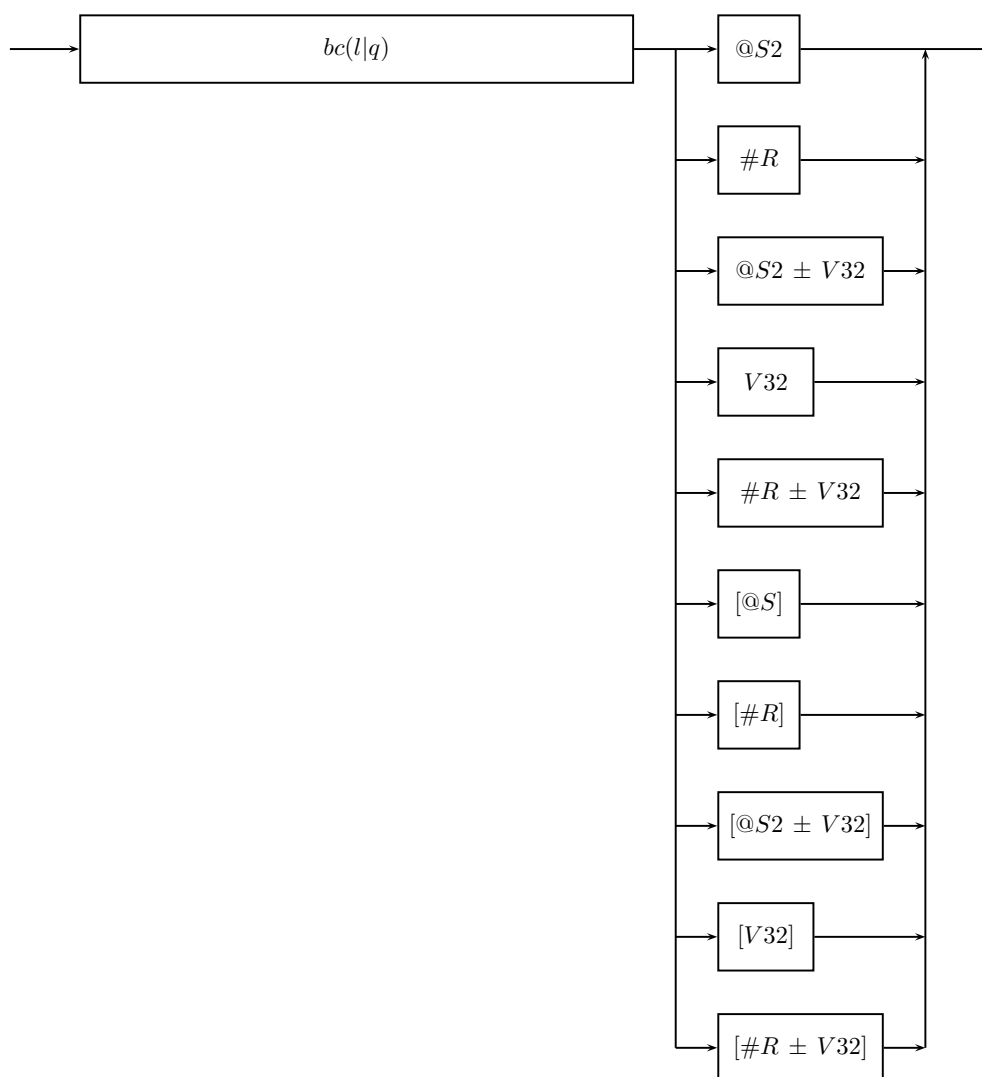


Рис. 1.9: Синтаксическое описание команды *bc*

Наличие результата: да

Алгоритм работы:

- вычислить число бит, равных единице;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 6-ти разрядный, старшие 58 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **quad** результат операции 7-ми разрядный, старшие 57 разрядов ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

- флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;
- флаг переполнения $OF = 0$;
- флаг переноса $CF = 0$;
- флаг знака $SF = 0$.

1.4.4.6 bsf (Bit Scanning Forward)

Сканирование битов вперёд

bsf ARG

Назначение: команда вычисления номера первого бита, равного единице, обнаруженного при сканировании аргумента в направлении от младшего бита к старшему (нумерация битов начинается с нуля)

Синтаксис:

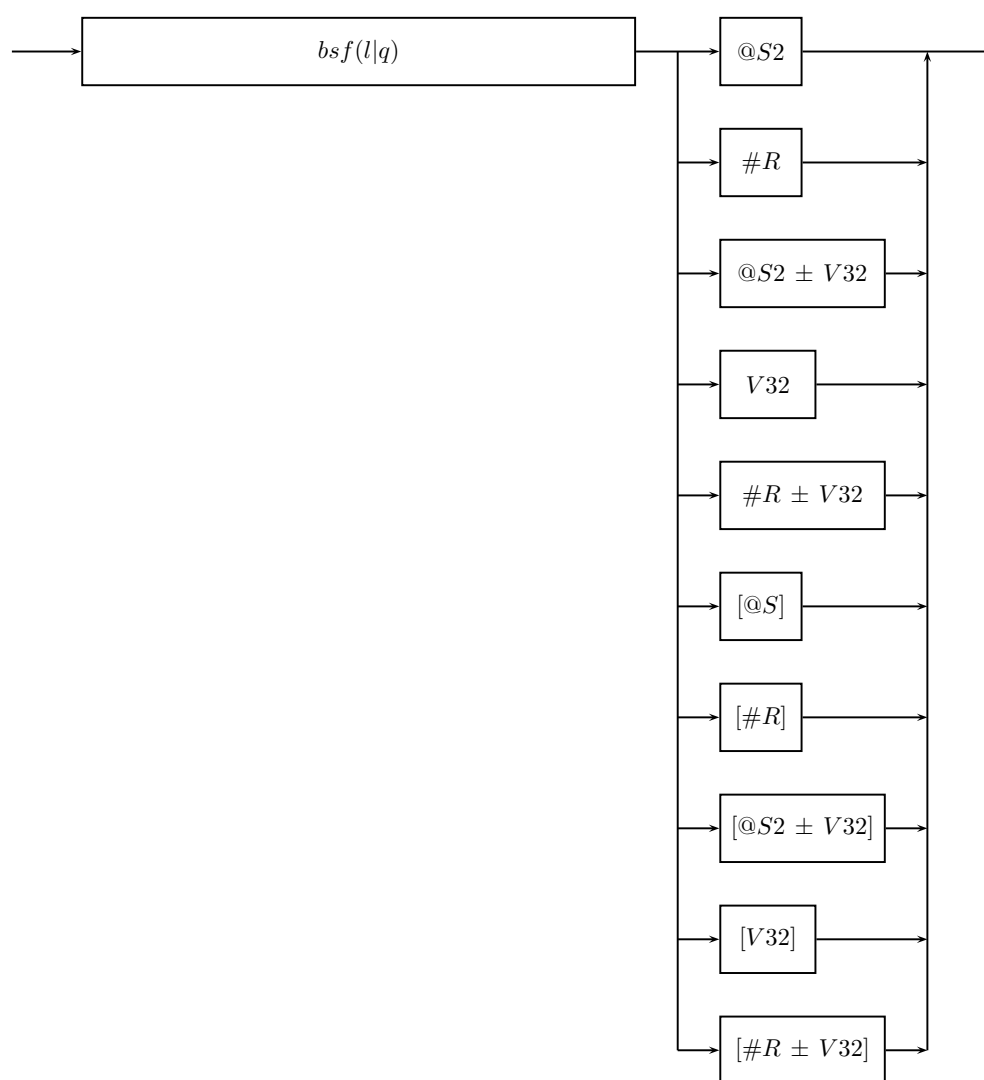


Рис. 1.10: Синтаксическое описание команды *bsf*

Наличие результата: да

Алгоритм работы:

- вычислить номер первого бита, равного единице, обнаруженного при сканировании аргумента в направлении от младшего бита к старшему;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 5-ти разрядный, старшие 59 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **quad** результат операции 6-ти разрядный, старшие 58 разрядов ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.7 bsr (Bit Scanning Reverse)

Сканирование битов назад

bsr ARG

Назначение: команда вычисления номера первого бита, равного единице, обнаруженного при сканировании аргумента в направлении от старшего бита к младшему (нумерация битов начинается с нуля)

Синтаксис:

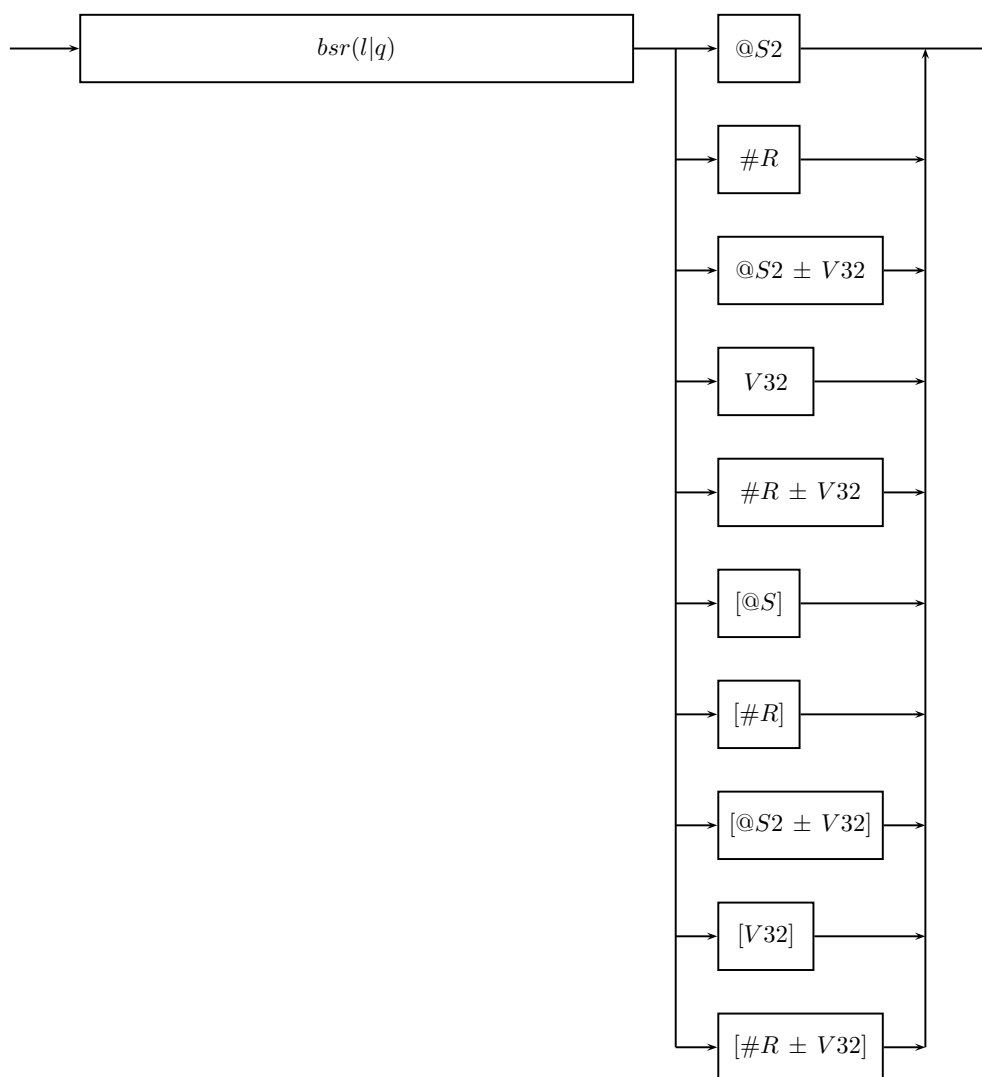


Рис. 1.11: Синтаксическое описание команды *bsr*

Наличие результата: да

Алгоритм работы:

- вычислить номер первого бита, равного единице, обнаруженного при сканировании аргумента в направлении от старшего бита к младшему;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 5-ти разрядный, старшие 59 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **quad** результат операции 6-ти разрядный, старшие 58 разрядов ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.8 bswap (Byte SWAP)

Перестановка байтов

bswap ARG

Назначение: команда перестановки байтов

Синтаксис:

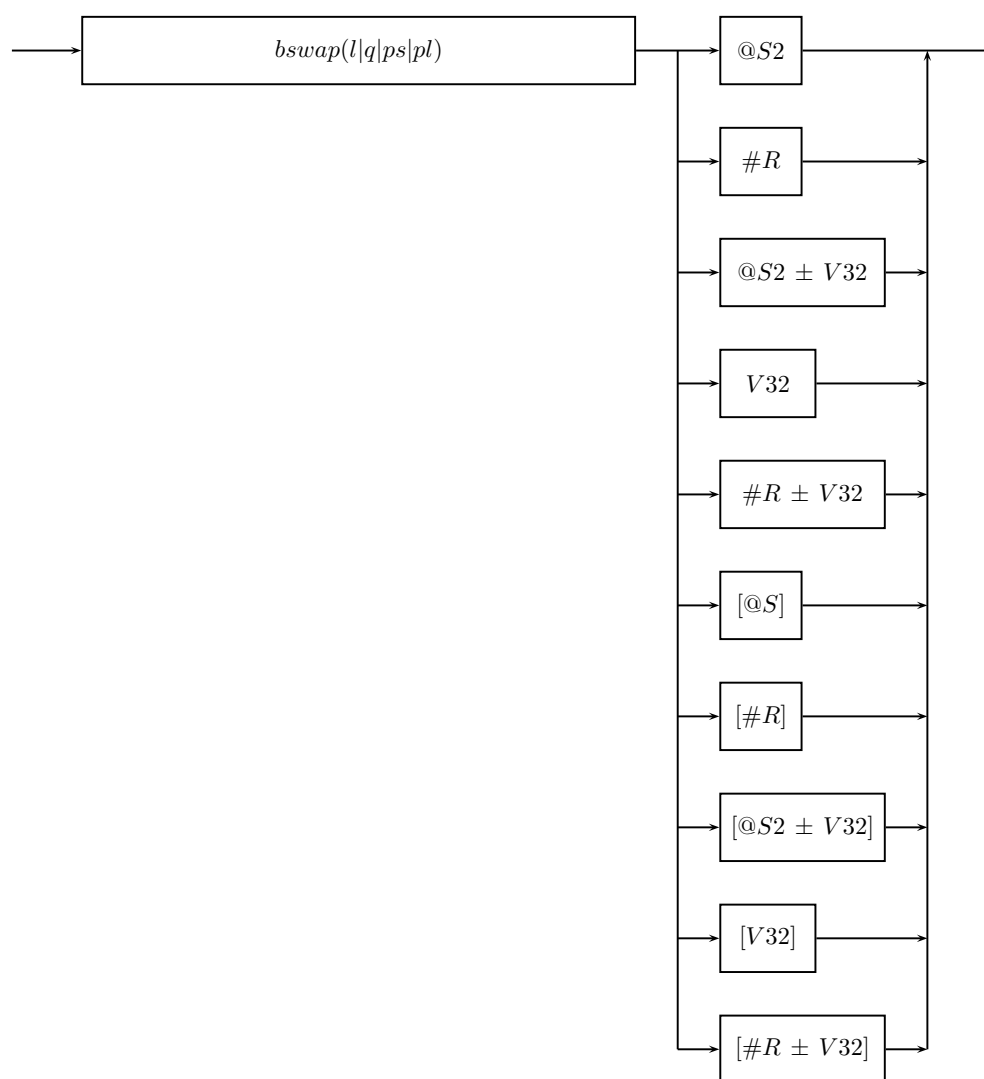


Рис. 1.12: Синтаксическое описание команды *bswap*

Наличие результата: да

Алгоритм работы:

- выполнить перестановку байтов аргумента согласно типу операции:

для типа операции **long**:

$$ARG[0] \leftrightarrow ARG[3],$$

$$ARG[1] \leftrightarrow ARG[2];$$

для типа операции **quad**:

$$ARG[0] \leftrightarrow ARG[7],$$

$$ARG[1] \leftrightarrow ARG[6],$$

$$ARG[2] \leftrightarrow ARG[5],$$

$$ARG[3] \leftrightarrow ARG[4];$$

для типа операции **packed short**:

$$ARG[0] \leftrightarrow ARG[1],$$

$$ARG[2] \leftrightarrow ARG[3],$$

$$ARG[4] \leftrightarrow ARG[5],$$

$$ARG[6] \leftrightarrow ARG[7];$$

для типа операции **packed long**:

$$ARG[0] \leftrightarrow ARG[3],$$

$$ARG[1] \leftrightarrow ARG[2],$$

$$ARG[4] \leftrightarrow ARG[7],$$

$$ARG[5] \leftrightarrow ARG[6];$$

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типа операции **long, quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

– для типа операции **packed short, packed long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.9 *cb* (Convolute Byte)

Свёртка байтов

cb ARG

Назначение: команда свёртки байтов

Синтаксис:

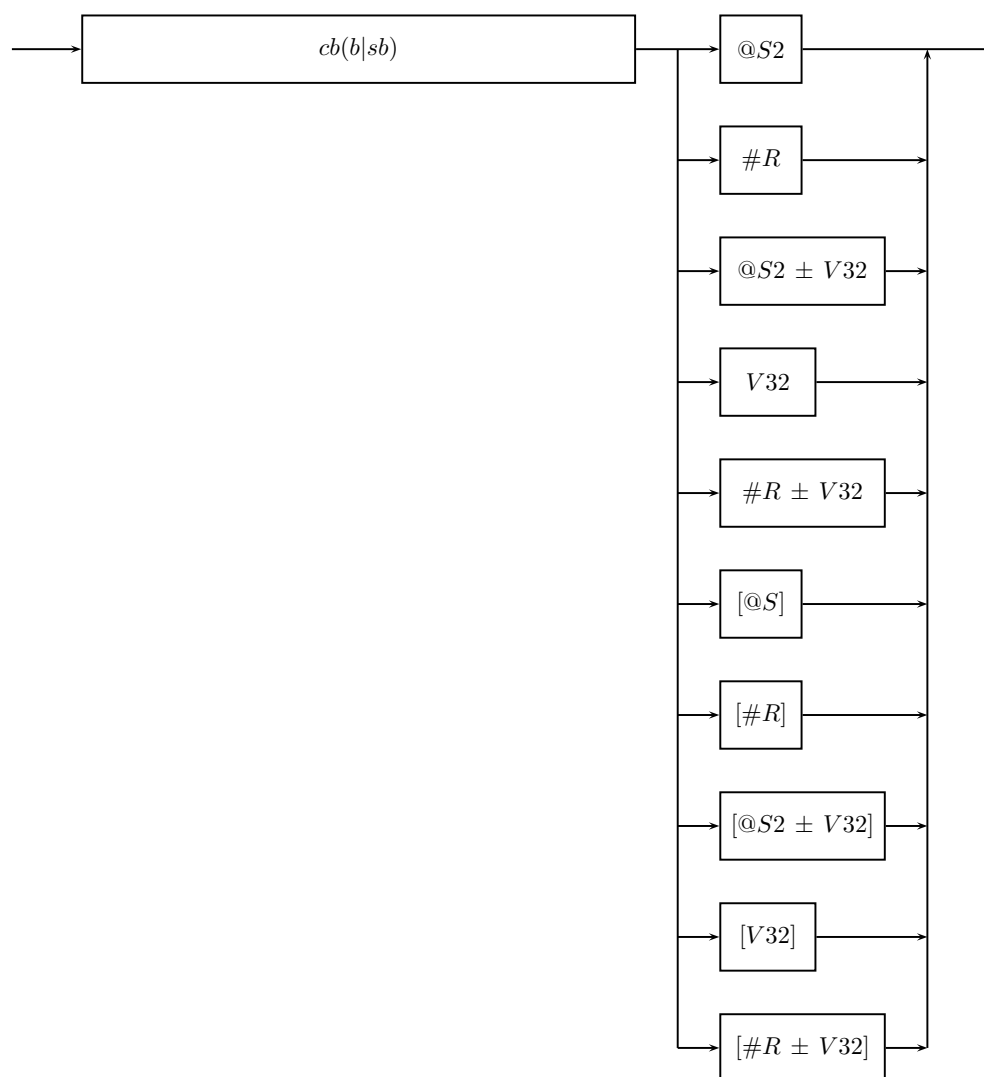


Рис. 1.13: Синтаксическое описание команды *cb*

Наличие результата: да

Алгоритм работы:

- выполнить свёртку байтов 64-х разрядного аргумента согласно типу операции (в квадратных скобках указаны номера битов аргумента и результата, нумерация битов начинается с нуля, & — операция конкатенации):

для типа операции **byte**:

$$RES[31 : 24] = ARG[63] \& ARG[54 : 48];$$

$$RES[23 : 16] = ARG[47] \& ARG[38 : 32];$$

$$RES[15 : 8] = ARG[31] \& ARG[22 : 16];$$

$$RES[7 : 0] = ARG[15] \& ARG[6 : 0];$$

для типа операции **signed byte**:

$$RES[31 : 24] = ARG[55 : 48];$$

$$RES[23 : 16] = ARG[39 : 32];$$

$$RES[15 : 8] = ARG[23 : 16];$$

$$RES[7 : 0] = ARG[7 : 0];$$

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.10 cbst (Convolute Byte with SaTuration)

Свёртка байтов с насыщением

cbst ARG

Назначение: команда свёртки байтов с насыщением

Синтаксис:

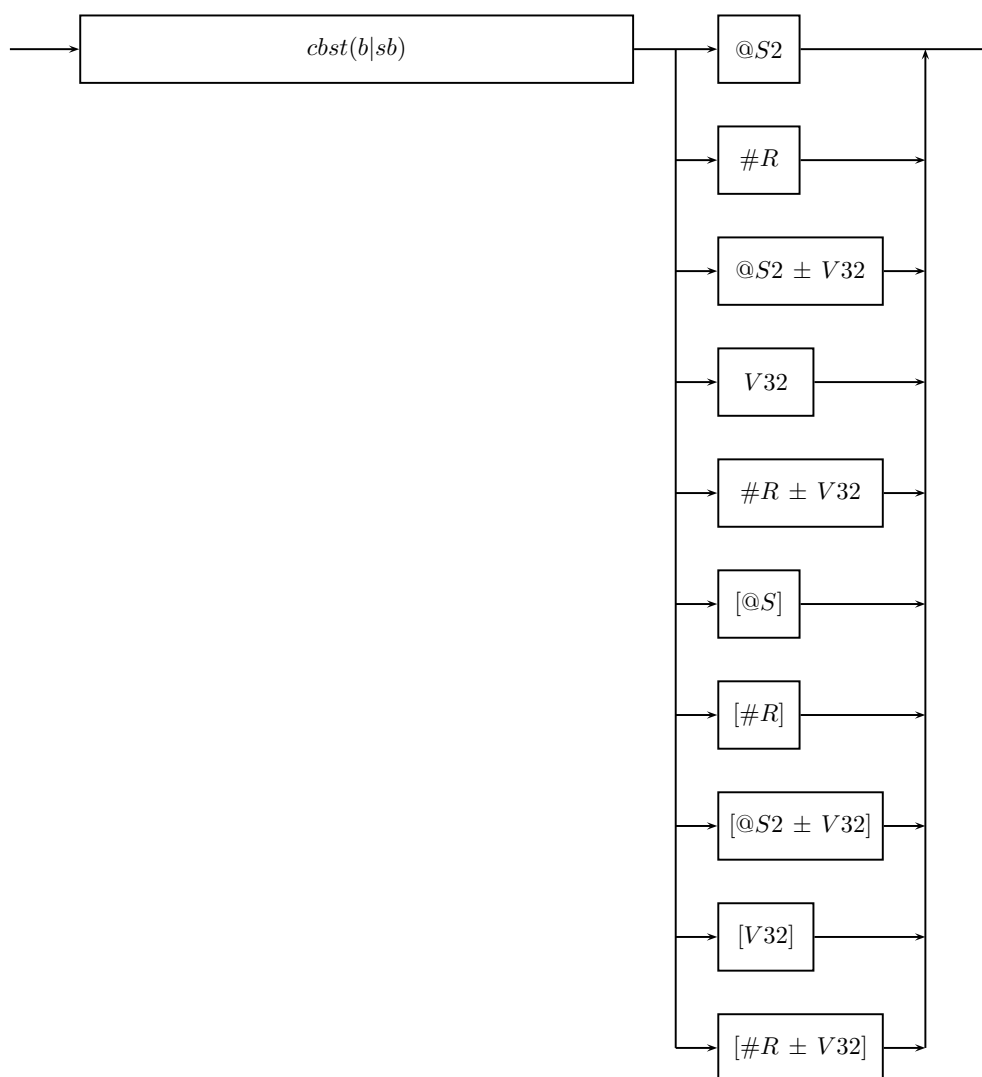


Рис. 1.14: Синтаксическое описание команды *cbst*

Наличие результата: да

Алгоритм работы:

- выполнить свёртку байтов 64-х разрядного аргумента согласно типу операции (в квадратных скобках указаны номера битов аргумента и результата, нумерация битов начинается с нуля):

для типа операции **byte**:

$$\begin{aligned}
 RES[31 : 24] &= \begin{cases} ARG[55 : 48], & \text{если } ARG[63 : 48] \leq 255 \\ 255, & \text{если } ARG[63 : 48] > 255 \end{cases} \\
 RES[23 : 16] &= \begin{cases} ARG[39 : 32], & \text{если } ARG[47 : 32] \leq 255 \\ 255, & \text{если } ARG[47 : 32] > 255 \end{cases} \\
 RES[15 : 08] &= \begin{cases} ARG[23 : 16], & \text{если } ARG[31 : 16] \leq 255 \\ 255, & \text{если } ARG[31 : 16] > 255 \end{cases} \\
 RES[07 : 00] &= \begin{cases} ARG[07 : 00], & \text{если } ARG[15 : 00] \leq 255 \\ 255, & \text{если } ARG[15 : 00] > 255 \end{cases}
 \end{aligned}$$

для типа операции **signed byte**:

$$\begin{aligned}
 RES[31 : 24] &= \begin{cases} ARG[55 : 48], & \text{если } -128 \leq ARG[63 : 48] \leq 127 \\ -128, & \text{если } ARG[63 : 48] < -128 \\ 127, & \text{если } ARG[63 : 48] > 127 \end{cases} \\
 RES[23 : 16] &= \begin{cases} ARG[39 : 32], & \text{если } -128 \leq ARG[47 : 32] \leq 127 \\ -128, & \text{если } ARG[47 : 32] < -128 \\ 127, & \text{если } ARG[47 : 32] > 127 \end{cases} \\
 RES[15 : 08] &= \begin{cases} ARG[23 : 16], & \text{если } -128 \leq ARG[31 : 16] \leq 127 \\ -128, & \text{если } ARG[31 : 16] < -128 \\ 127, & \text{если } ARG[31 : 16] > 127 \end{cases} \\
 RES[07 : 00] &= \begin{cases} ARG[07 : 00], & \text{если } -128 \leq ARG[15 : 00] \leq 127 \\ -128, & \text{если } ARG[15 : 00] < -128 \\ 127, & \text{если } ARG[15 : 00] > 127 \end{cases}
 \end{aligned}$$

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.11 *cfsl* (Convert Float to Signed Long)

Преобразование типа

cfsl *ARG*

Назначение: команда преобразование числа с плавающей точкой одинарной точности в знаковое целое

Синтаксис:

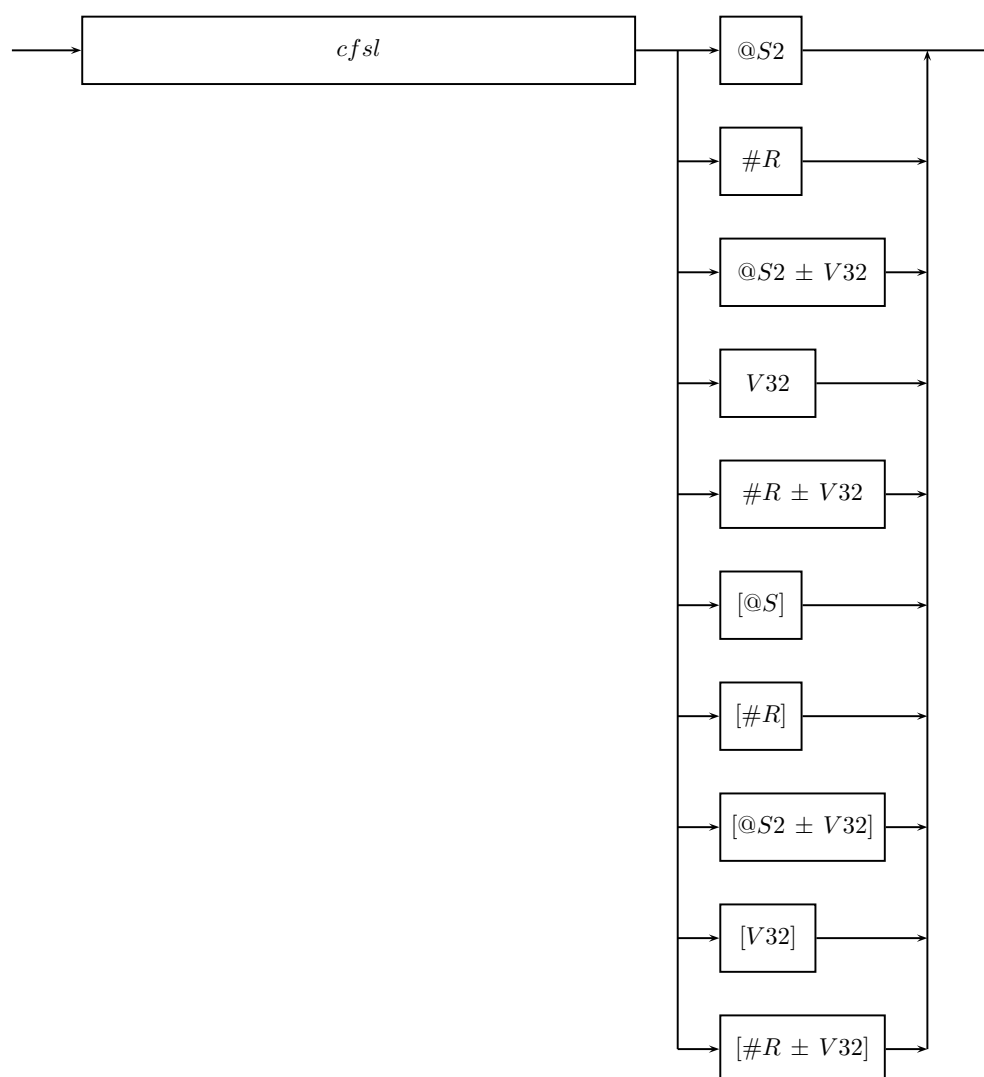


Рис. 1.15: Синтаксическое описание команды *cfsl*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование числа с плавающей точкой одинарной точности в знакове целое, если результат выходит за пределы знаковой 32-битной разрядной сетки, то в качестве результата вернуть максимально/минимально возможное число в формате знакового 32-битного числа;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если при выполнении операции произошло переполнение (экспонента аргумента больше 158), иначе – $OF = 0$;

флаг потери точности $IF = 1$, если $OF = 1$, либо аргумент является денормализованным числом, либо $ARG = 0x4F000000$, иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `cfs1` используется для преобразования аргумента, интерпретируемого как вещественное число одинарной точности, к знаковому целому 32-х разрядному числу.

Пример:

```
1 .text
2
3 A:
4     cfs1 0f1.394E1
5     wr1 @1, 0
6 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №3 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим

значением адреса ассемблирования, а также начинает новый параграф;

- в строке №4 командой `cfs1` выполняется операция преобразования с предварительным округлением до ближайшего целого вещественного числа одинарной точности 13.94 к целому знаковому 32-х разрядному числу (результат выполнения операция — 14);
- в строке №5 командой `wr1` в память данных по адресу 0 записывается значение результата выполнения предшествующей команды: @1 — результат выполнения команды преобразования типа в строке №4;
- в строке №6 командой `complete` завершается текущий параграф.

1.4.4.12 cfd (Convert Float to Double)

Преобразование типа

cfd ARG

Назначение: команда преобразования числа с плавающей точкой одинарной точности в число с плавающей точкой двойной точности

Синтаксис:

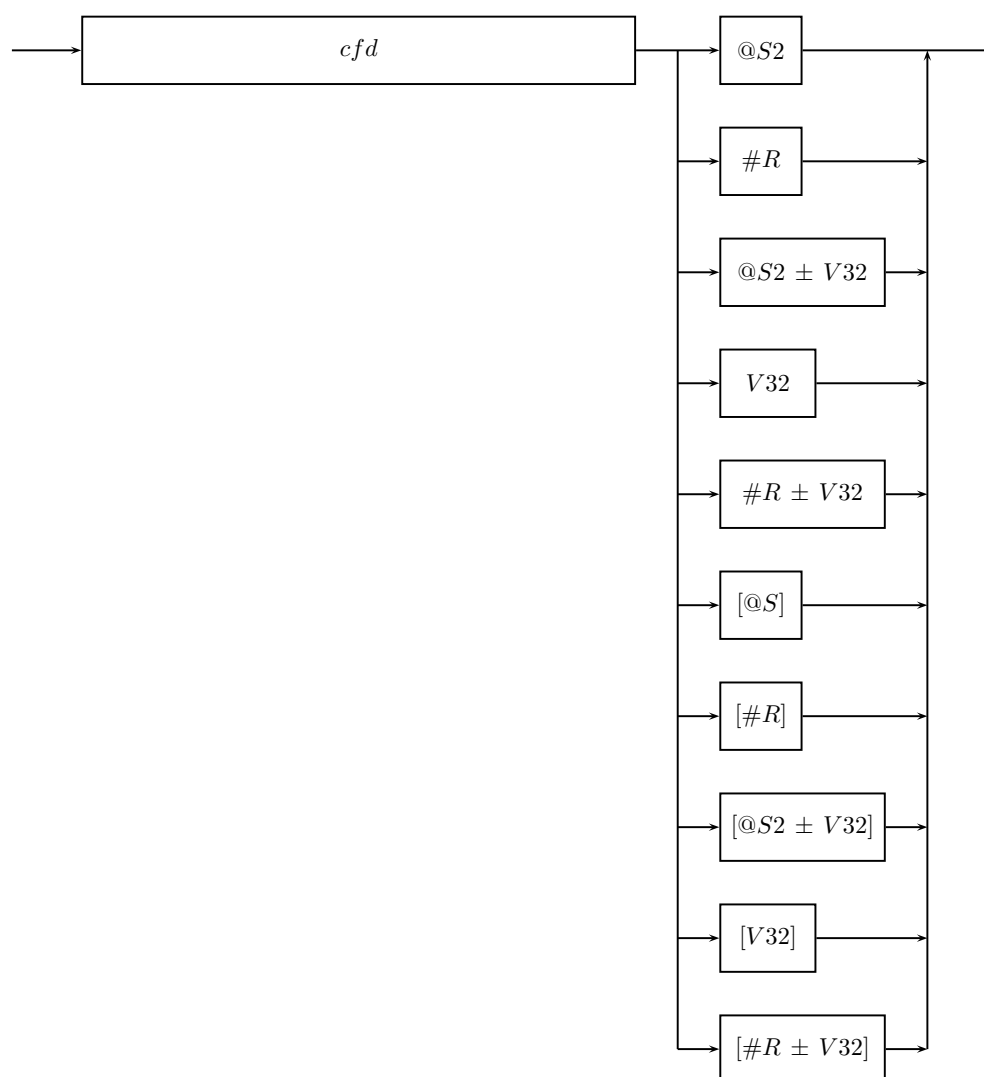


Рис. 1.16: Синтаксическое описание команды *cfd*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование числа с плавающей точкой одинарной точности в число с плавающей точкой двойной точности;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.13 *cdf* (Convert Double to Float)

Преобразование типа

cdf *ARG*

Назначение: команда преобразования числа с плавающей точкой двойной точности в число с плавающей точкой одинарной точности

Синтаксис:

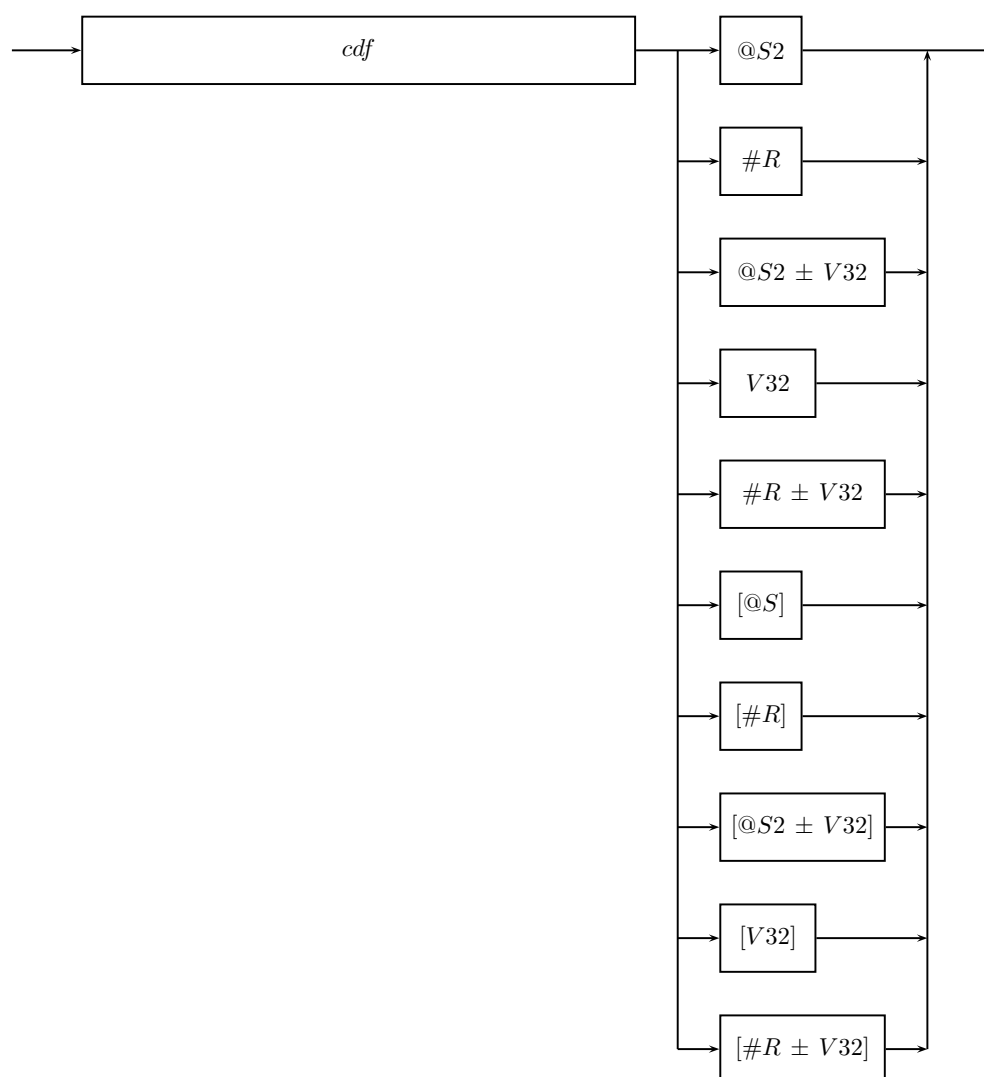


Рис. 1.17: Синтаксическое описание команды *cdf*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование числа с плавающей точкой двойной точности в число с плавающей точкой одинарной точности, если результат выходит за пределы формата числа с плавающей точкой одинарной точности, то в качестве результата возвращается значение «бесконечность» в формате числа с плавающей точкой одинарной точности;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если при выполнении операции произошло переполнение, иначе – $OF = 0$;

флаг потери точности $IF = 1$, если $OF = 1$ или аргумент является денормализованным числом, иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.14 *clf* (Convert Long to Float)

Преобразование типа

clf ARG

Назначение: команда преобразования беззнакового целого в число с плавающей точкой одинарной точности

Синтаксис:

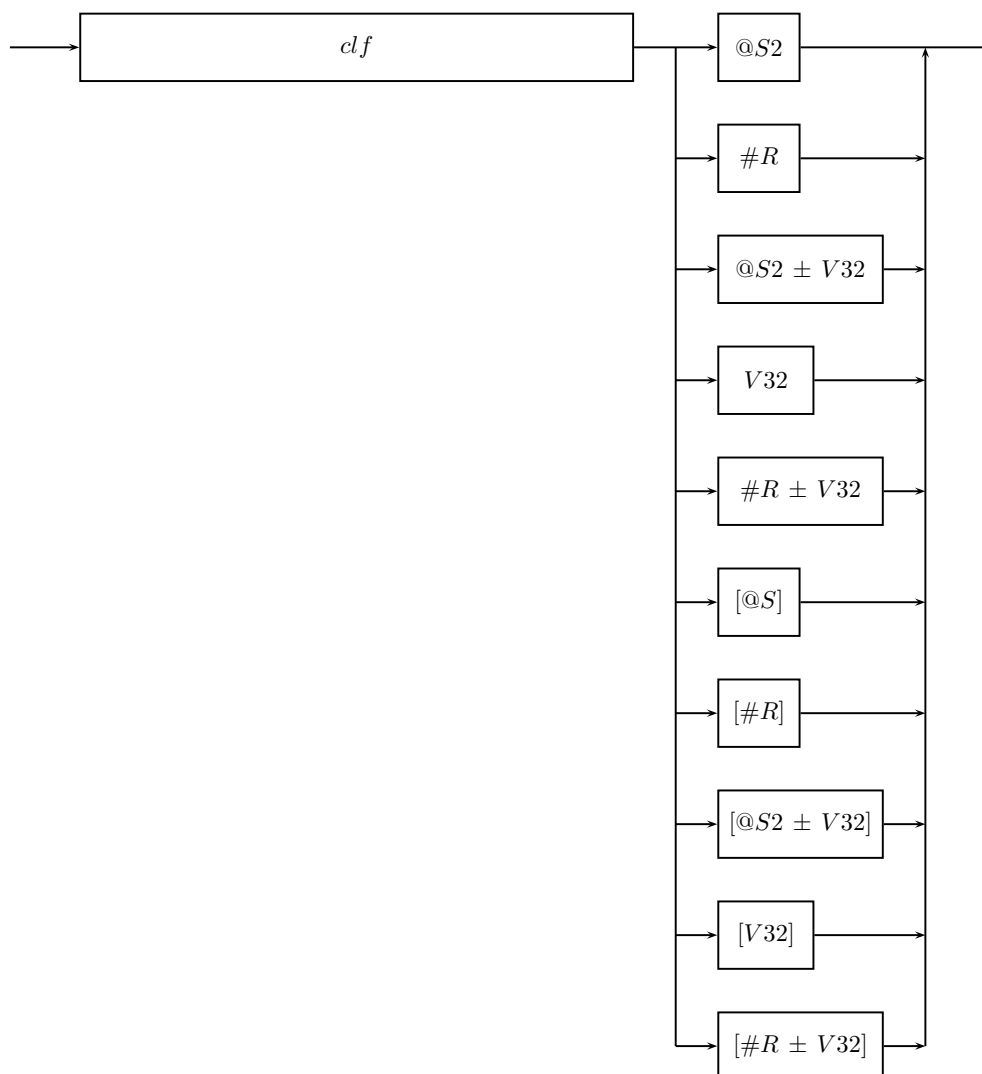


Рис. 1.18: Синтаксическое описание команды *clf*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование беззнакового целого в число с плавающей точкой одинарной точности;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `clf` используется для преобразования значения аргумента, интерпретируемого как беззнаковое целое 32-х разрядное число, к вещественному числу одинарной точности.

Пример:

```
1 .text
2
3 A:
4     clf 1394
5     wrl @1, 0
6 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №3 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №4 командой `clf` выполняется операция преобразования целого беззнакового 32-х разрядного числа 1394 к вещественному числу одинарной точности;

- в строке №5 командой `wrl` в память данных по адресу 0 записывается значение результата выполнения предшествующей команды: @1 — результат выполнения команды преобразования типа в строке №4;
- в строке №6 командой `complete` завершается текущий параграф.

1.4.4.15 csf (Convert Signed Long to Float)

Преобразование типа

csf ARG

Назначение: команда преобразования знакового целого в число с плавающей точкой

Синтаксис:

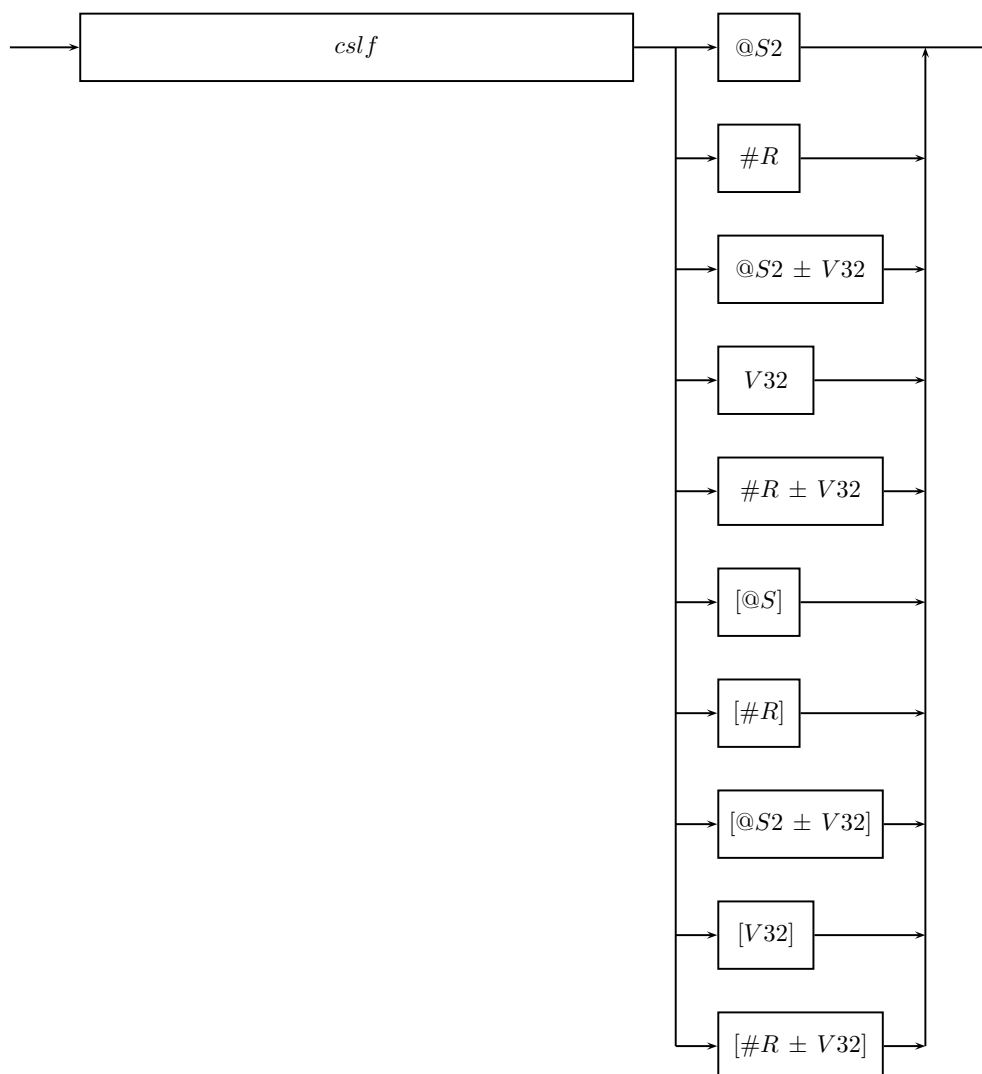


Рис. 1.19: Синтаксическое описание команды *csf*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование знакового целого в число с плавающей точкой одинарной точности;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `cslf` используется для преобразования значения аргумента, интерпретируемого как знаковое целое 32-х разрядное число, к вещественному числу одинарной точности.

Пример:

```
1 .text
2
3 A:
4     csf  -1394
5     wr  @1, 0
6 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №3 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №4 командой `csf` выполняется операция преобразования целого знакового 32-х разрядного числа `-1394` к вещественному числу одинарной точности;

- в строке №5 командой `wrl` в память данных по адресу 0 записывается значение результата выполнения предшествующей команды: @1 — результат выполнения команды преобразования типа в строке №4;
- в строке №6 командой `complete` завершается текущий параграф.

1.4.4.16 div (DIVide)

Деление

div ARG1, ARG2

Назначение: операция деления двух аргументов

Синтаксис:

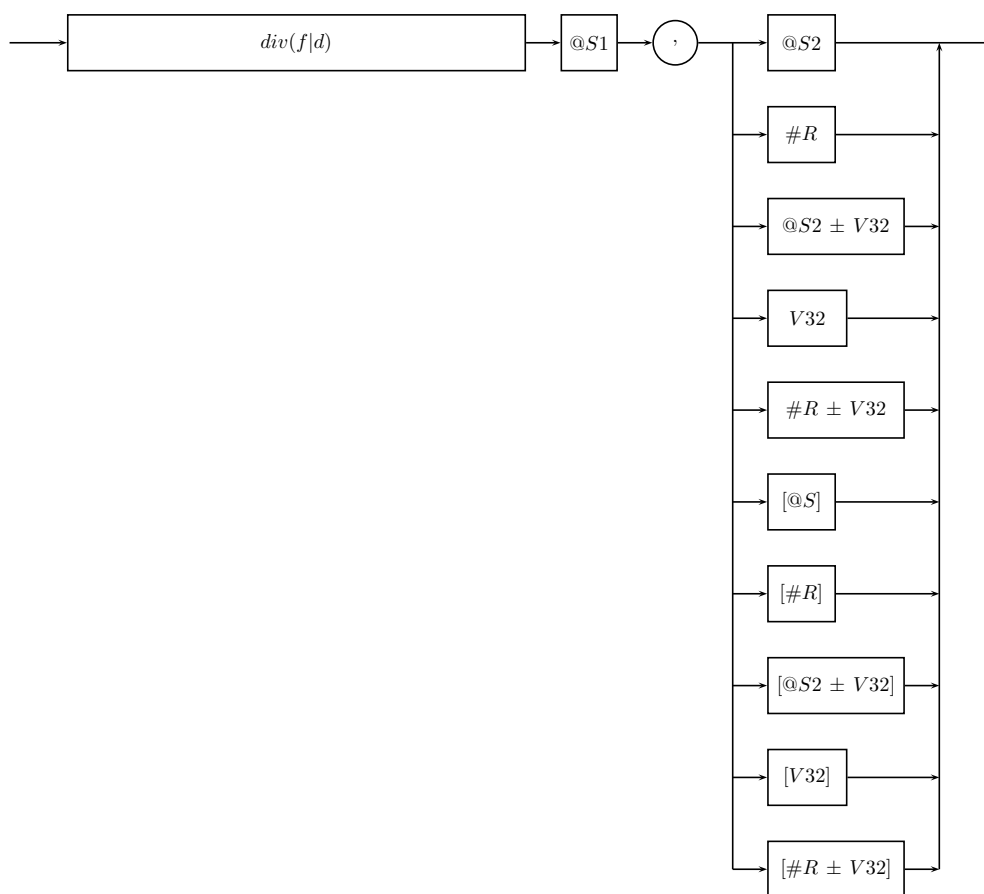


Рис. 1.20: Синтаксическое описание команды *div*

Наличие результата: да

Алгоритм работы:

- вычислить частное аргументов $ARG1/ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **double** результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 1$, если в результате произошла потеря точности вследствие постнормализации (округления до ближайшего чётного), либо остаток от деления не равен 0, либо в результате все биты экспоненты равны 1, либо при расчётах расширенный бит экспоненты оказался равен 1, иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `div` используется для деления двух операндов, значение которых интерпретируется согласно типу операции. При выполнении операции деления возможно возникновение исключительной ситуации — деление на 0. В этом случае формируется запрос на прерывание, т. е. выставляются соответствующие биты регистров прерываний *INTR* и исключений *ER*.

Пример:

```
1  .data
2
3  B:
4      .float 0f12.8e12 , 0f-5.6e-4
5
6  .text
7
8  A:
9      rdl B
10     rdl B + 4
11     divf @1, @2
12     wrf @1, B + 8
13 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) *B*, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера `.float` в текущую секцию ассемблирования записывается два 32-х разрядных вещественных числа, начиная с текущего адреса ассемблирования;
- в строке №6 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №8 объявляется символ (идентификатор) *A*, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строках №№9,10 командами `rddl`, читаются из памяти данных два 32-х разрядных целых беззнаковых числа по адресам *B* и *B + 4* соответственно и помещаются в коммутатор;
- в строке №11 командой `divf` выполняется операция деления результатов выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №19, @2 — результат выполнения команды чтения в строке №10; оба аргумента команды `divf` согласно суффиксу `f` интерпретируются как вещественные числа одинарной точности; результат выполнения команды также интерпретируется как вещественное число одинарной точности и помещается в коммутатор;
- в строке №12 командой `wrl` осуществляется запись в память данных по адресу *B + 8* результата выполнения предшествующей команды: @1 — результат выполнения команды деления в строке №11;
- в строке №13 командой `complete` завершается текущий параграф.

1.4.4.17 eхa (EXacutive Address)

Исполнительный адрес

exa ARG

Назначение: операция формирования исполнительного адреса памяти

Синтаксис:

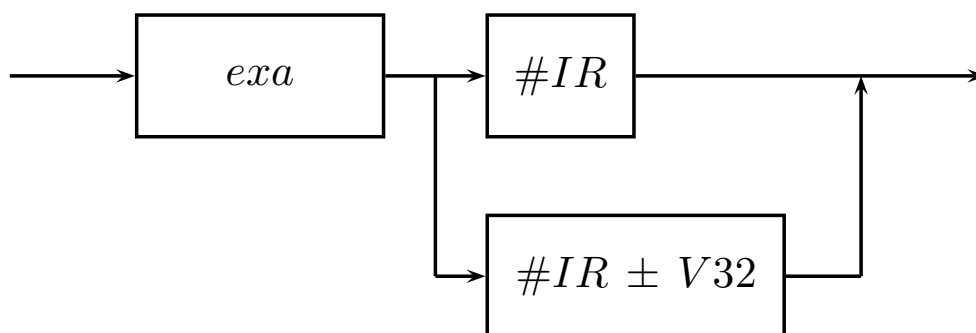


Рис. 1.21: Синтаксическое описание команды *exa*

Наличие результата: да

Алгоритм работы:

- сформировать исполнительный адрес памяти, используя значения индексного регистра и смещения, заданных аргументом *ARG*;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

Применение: команда *exa* используется для формирования исполнительного адреса памяти и сохранения его в коммутаторе. Другими словами, данная команда,

в отличие от большинства других команд процессора, не осуществляет чтение значения из памяти данных по сформированному адресу. Правила формирования исполнительного адреса описаны в разделе «Регистры индексные» (1.2.2.2). В общем случае, интерпретация значения, сохранённого данной командой в коммутаторе, зависит от алгоритма программы.

Пример:

```
1  /*
2  * Пусть имеется два массива A и B целых
3  * беззнаковых чисел из пяти элементов с
4  * исходными данными. Необходимо i-ое
5  * значение массива A умножить на i-ое
6  * значение массива B и результат умножения
7  * сохранить в i-ом значении массива C.
8  */
9
10 .data
11
12 A:
13     .long 5, 4, 3, 2, 1
14 B:
15     .long 10, 9, 8, 7, 6
16 C:
17
18 .text
19
20 INIT:
21     jmp MUL
22
23     getl 0x00040007
24     patch @1, 0
25     setq #8, @1
26     getl 0x0000001C
27     patch @1, A
28     setq #9, @1
29 complete
30
```

```
31 MUL:
32     irm 3
33     exa #8
34     je @1, FINISH
35     jne @2, MUL
36
37     rdl #9
38     rdl #9, B - A
39     mull @1, @2
40     wrl @1, #9, C - A
41 complete
42
43 FINISH:
44     jmp FINISH
45 complete
```

Пояснения к примеру

- в строке №10 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №12 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №13 директивой ассемблера `.long` в текущую секцию ассемблирования записывается пять 32-х разрядных целых числа, начиная с текущего адреса ассемблирования;
- в строке №14 объявляется символ (идентификатор) `B`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №15 директивой ассемблера `.long` в текущую секцию ассемблирования записывается пять 32-х разрядных целых числа, начиная с текущего адреса ассемблирования;
- в строке №16 объявляется символ (идентификатор) `C`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №18 директивой ассемблера `.text` устанавливается текущая секция

ассемблирования — секция исполняемых инструкций *text*;

- в строке №20 объявляется символ (идентификатор) *INIT*, который является меткой в текущей секции ассемблирования (*text*), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №21 командой *jmp* безусловно устанавливается адрес следующего параграфа равным *MUL*;
- в строке №23 командой *getl* помещается в коммутатор 32-х разрядное целое беззнаковое число (константа *0x00040007*);
- в строке №24 командой *patch* формируется 64-х разрядное число, старшие 32 разряда которого инициализируются результатом выполнения предшествующей команды: @1 — результат выполнения команды извлечения в строке №23, а младшие 32 разряда — значением 0;
- в строке №25 командой *setq* в индексный регистр №8 помещается результат выполнения предшествующей команды: @1 — результат выполнения команды склейки в строке №24;
- в строке №26 командой *getl* помещается в коммутатор 32-х разрядное целое беззнаковое число (константа *0x0000001C*);
- в строке №27 командой *patch* формируется 64-х разрядное число, старшие 32 разряда которого инициализируются результатом выполнения предшествующей команды: @1 — результат выполнения команды извлечения в строке №23, а младшие 32 разряда — значением метки *A*;
- в строке №28 командой *setq* в индексный регистр №9 помещается результат выполнения предшествующей команды: @1 — результат выполнения команды склейки в строке №24;
- в строке №29 командой *complete* завершается текущий параграф; происходит фактическая установка значений индексных регистров №№8,9.
- в строке №31 объявляется символ (идентификатор) *MUL*, который является меткой в текущей секции ассемблирования (*text*), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №32 командой *irm*, аргументом которой является значение 3, сообщается ядру о необходимости пересчёта значений индексных регистров №№8,9 по завершении текущего параграфа;

- в строке №33 командой *еха* помещается в коммутатор 32-х разрядное целое беззнаковое число, сформированное на основании значения индексного регистра №8;
- в строке №34 командой *je* устанавливается адрес следующего параграфа равным *FINISH*, если установлен флаг нуля (*ZF*) результата выполнения предшествующей команды: @1 — результат выполнения команды формирования исполнительного адреса в строке №33;
- в строке №35 командой *jne* устанавливается адрес следующего параграфа равным *MUL*, если сброшен флаг нуля (*ZF*) результата выполнения предшествующей команды: @1 — результат выполнения команды формирования исполнительного адреса в строке №33;
- в строке №37 командой *rdd*, читается из памяти данных 32-х разрядное целое беззнаковое число по адресу, сформированному на основании значения индексного регистра №9 и помещается в коммутатор;
- в строке №38 командой *rdd*, читается из памяти данных 32-х разрядное целое беззнаковое число по адресу, сформированному на основании значения индексного регистра №9 и смещения $B - A$ и помещается в коммутатор;
- в строке №39 командой *mull* выполняется операция умножения результатов выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №38, @2 — результат выполнения команды чтения в строке №37; результат выполнения команды также интерпретируются как 32-х разрядное целое знаковое число и помещается в коммутатор;
- в строке №40 командой *wrl* осуществляется запись в память данных по адресу, сформированному на основании значения индексного регистра №9 и смещения $C - A$, результата выполнения предшествующей команды: @1 — результат выполнения команды умножения в строке №39;
- в строке №41 командой *complete* завершается текущий параграф; происходит пересчёт значений индексных регистров №№8,9.
- в строке №43 объявляется символ (идентификатор) *FINISH*, который является меткой в текущей секции ассемблирования (*text*), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №44 командой *jmp* безусловно устанавливается адрес следующего

параграфа равным *FINISH*;

- в строке №45 командой *complete* завершается текущий параграф.

В приведённом примере индексный регистр №8 используется для организации циклического выполнения параграфа *MUL*; индексный регистр №9 используется для формирования исполнительного адреса памяти в командах чтения и записи. Исходя из начальных значений индексных регистров №№8,9, параграф *MUL* будет выполнен пять раз, а значение исполнительного адреса памяти в командах чтения и записи будет увеличиваться на четыре при каждом завершении выполнения параграфа *MUL*.

1.4.4.18 fcpy (Flag CoPY)

Копирование флагов

fcpy ARG1, ARG2

Назначение: команда копирования флагов

Синтаксис:

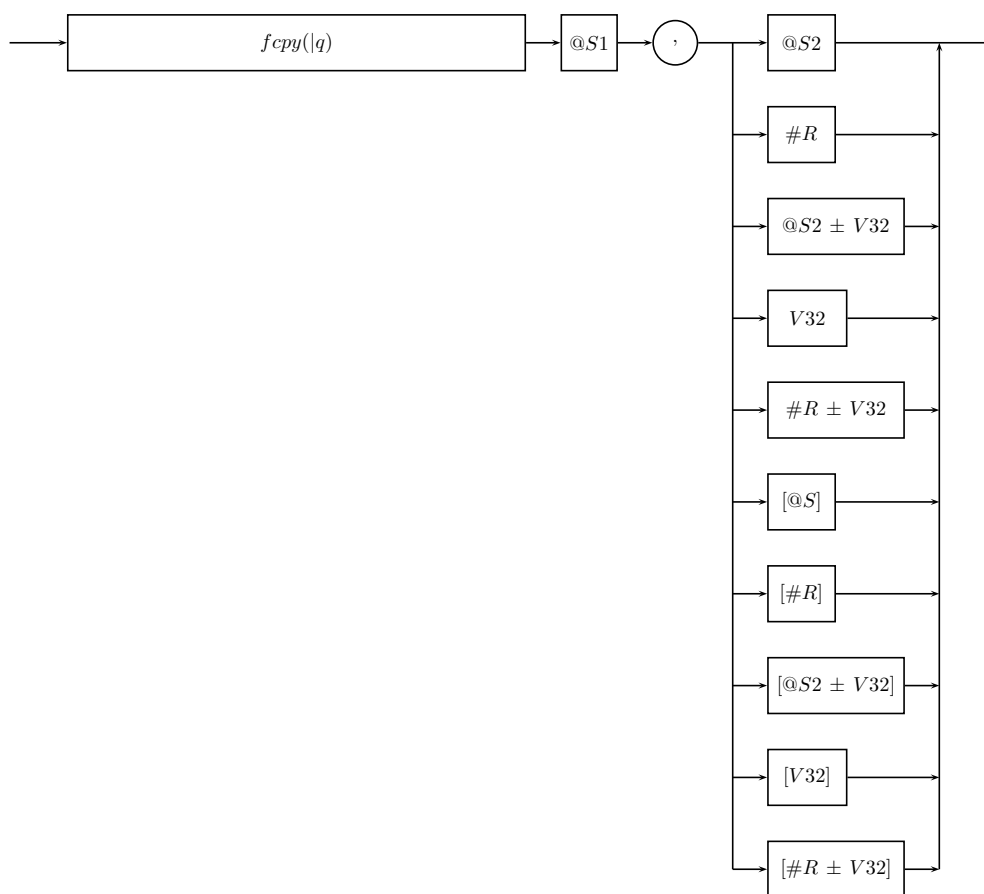


Рис. 1.22: Синтаксическое описание команды *fcpy*

Наличие результата: да

Алгоритм работы:

- установить результат, равным аргументу *ARG2*;
- установить флаги, равными флагам аргумента *ARG1*;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля равен значению флага нуля аргумента $ARG1$;

флаг переполнения равен значению флага переполнения аргумента $ARG1$;

флаг переноса равен значению флага переноса аргумента $ARG1$;

флаг знака равен значению флага знака аргумента $ARG1$.

1.4.4.19 forsa (FORced Select if Above)

Выбор первого или второго аргумента по условию

forsa ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

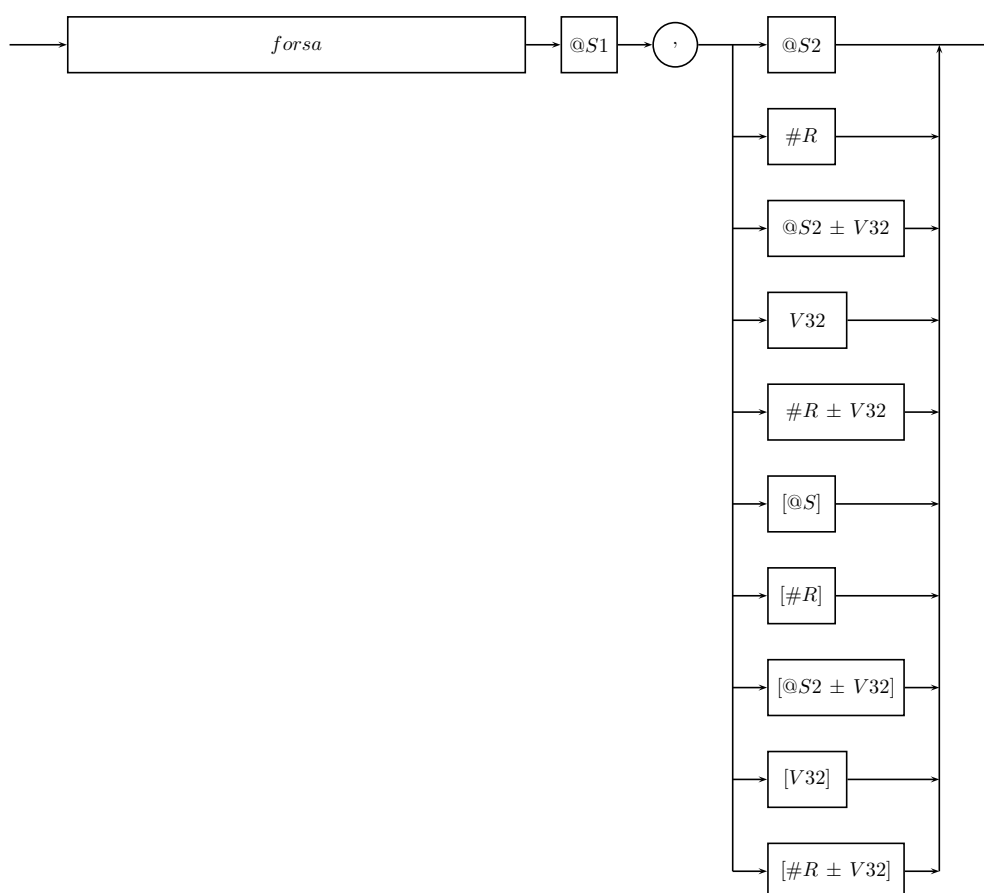


Рис. 1.23: Синтаксическое описание команды *forsa*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаги переноса (*CF*) и нуля (*ZF*) аргумента *ARG1* равны нулю, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.20 forsae (FORced Select if Above and Equal) / forsnc (FORced Select if Carry flag unset)

Выбор первого или второго аргумента по условию

$(forsae/forsnc) ARG1, ARG2$

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

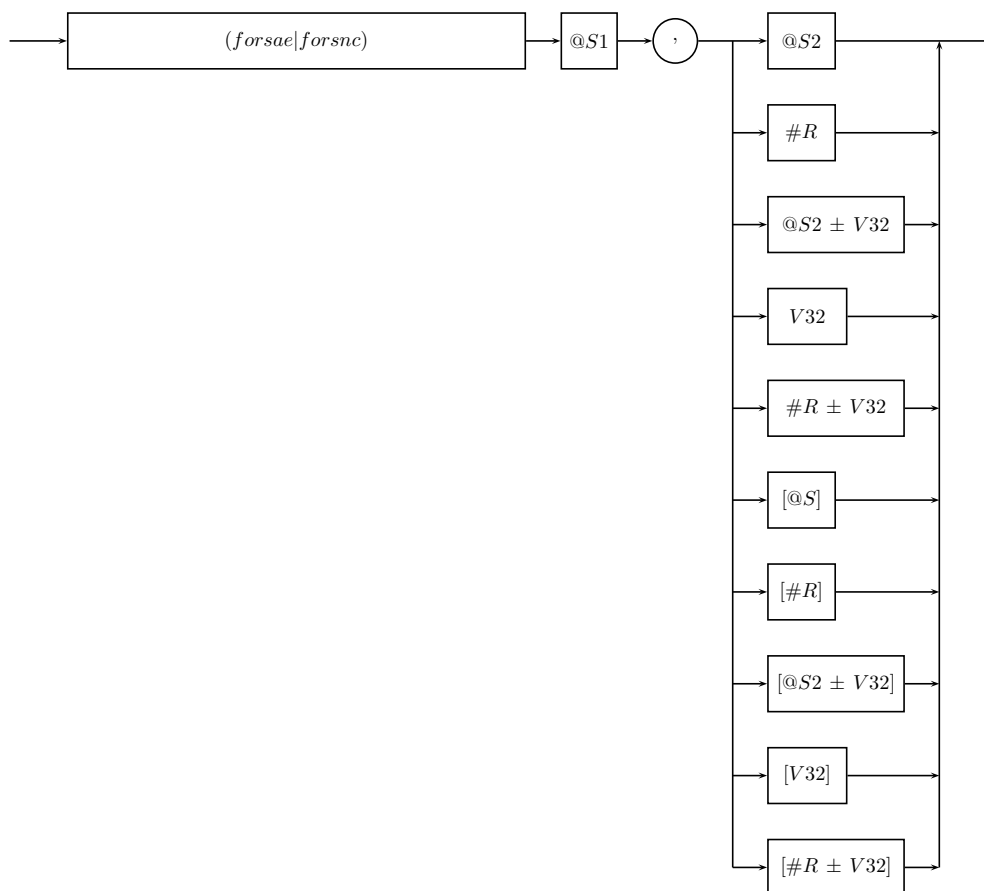


Рис. 1.24: Синтаксическое описание команды *forsae/forsnc*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг переноса (*CF*) аргумента *ARG1* равен нулю, иначе – равным значению аргумента *ARG2*;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.21 forsb (FORced Select if Below) / forsc (FORced Select if Carry flag set)

Выбор первого или второго аргумента по условию

$(forsb/forsc) ARG1, ARG2$

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

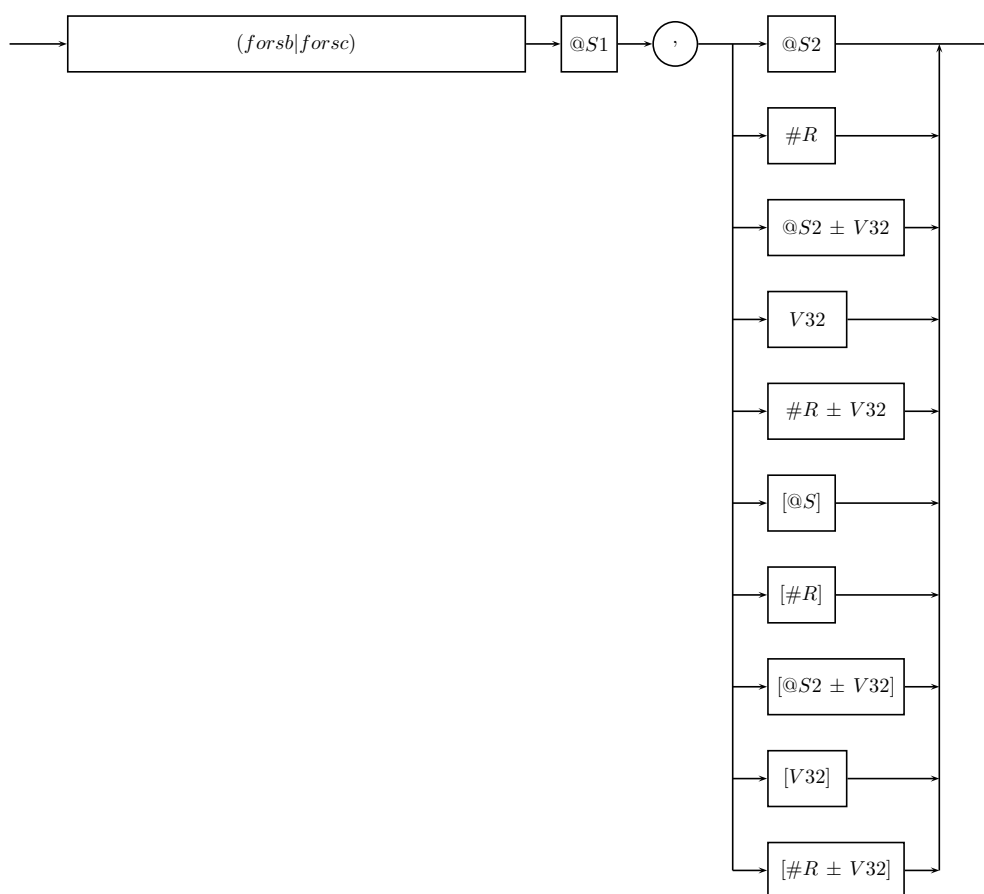


Рис. 1.25: Синтаксическое описание команды *forsb/forsc*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг переноса (*CF*) аргумента *ARG1* равен единице, иначе – равным значению аргумента *ARG2*;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.22 forsbe (FORced Select if Below and Equal)

Выбор первого или второго аргумента по условию

forsbe ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

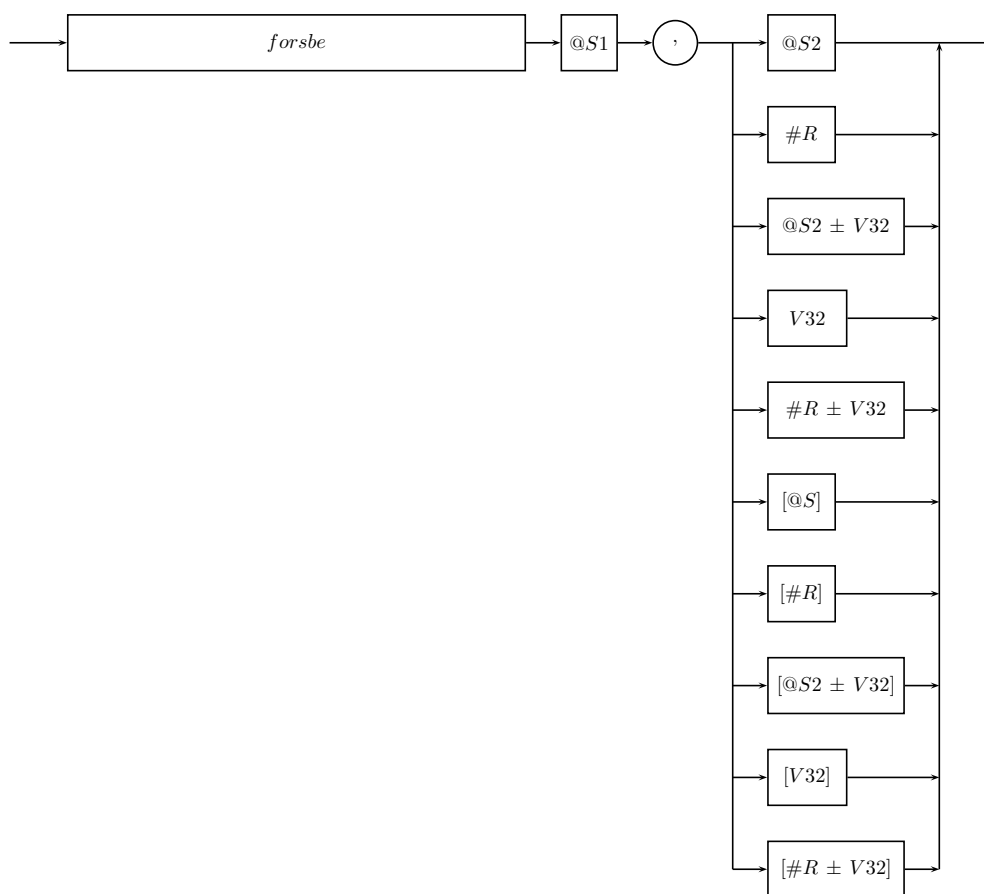


Рис. 1.26: Синтаксическое описание команды *forsbe*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаги переноса (*CF*) и нуля (*ZF*) аргумента *ARG1* равны единице, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.23 forse (FORced Select if Equal)

Выбор первого или второго аргумента по условию

forse ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

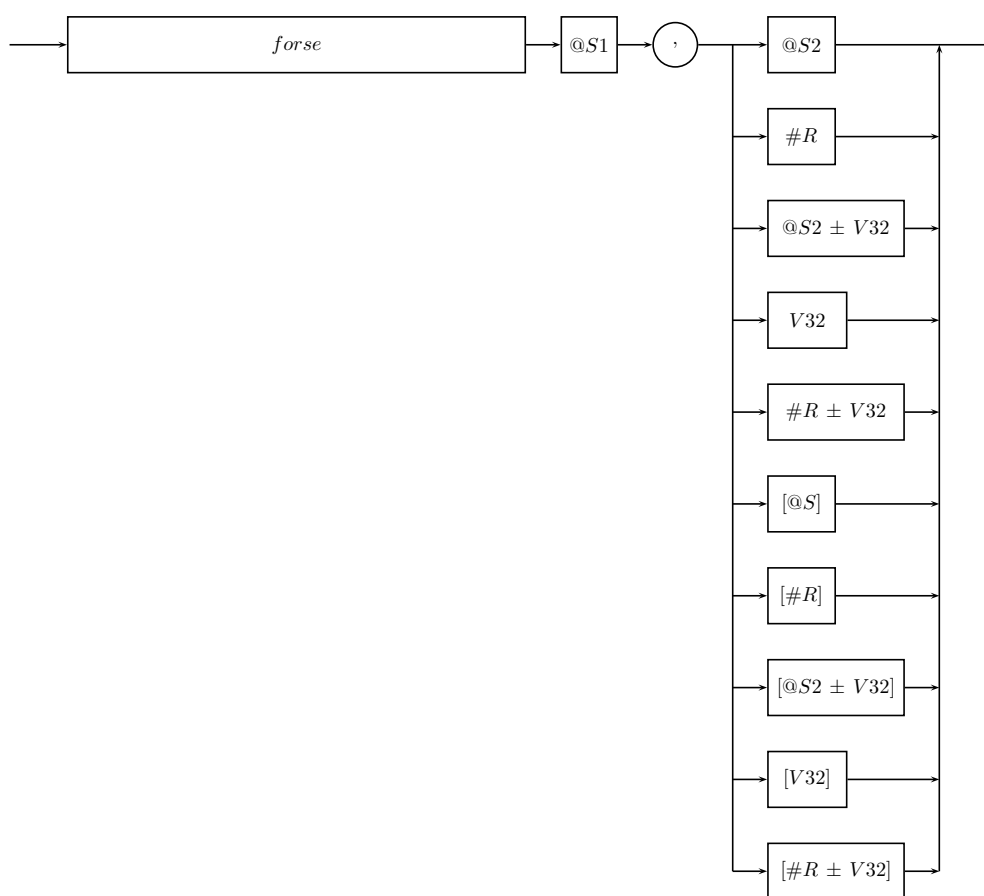


Рис. 1.27: Синтаксическое описание команды *forse*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг нуля (*ZF*) аргумента *ARG1* равен единице, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.24 forsg (FORced Select if Greater)

Выбор первого или второго аргумента по условию

*for*sg ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

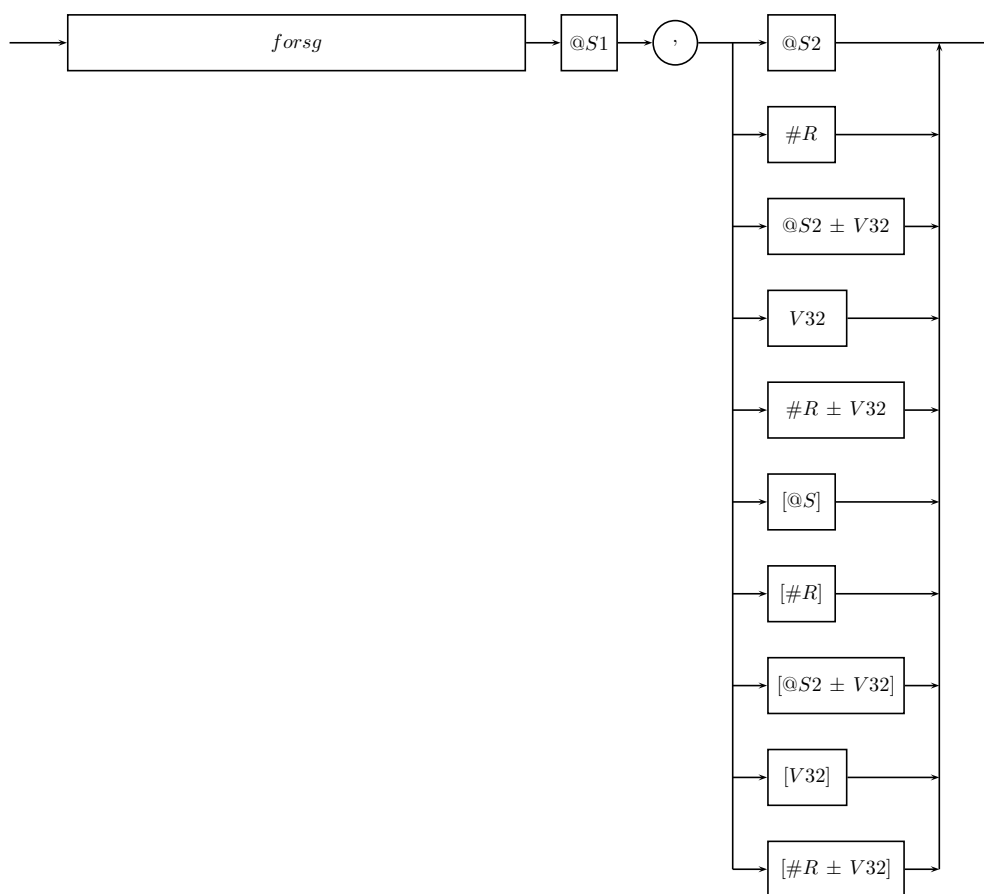


Рис. 1.28: Синтаксическое описание команды *for*sg

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента ARG1, если флаги знака (*SF*) и переполнения (*OF*) аргумента ARG1 равны и флаг нуля (*ZF*) аргумента ARG1 равен нулю, иначе – равным значению аргумента ARG2;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.25 forsgе (FORced Select if Greater and Equal)

Выбор первого или второго аргумента по условию

forsgе ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

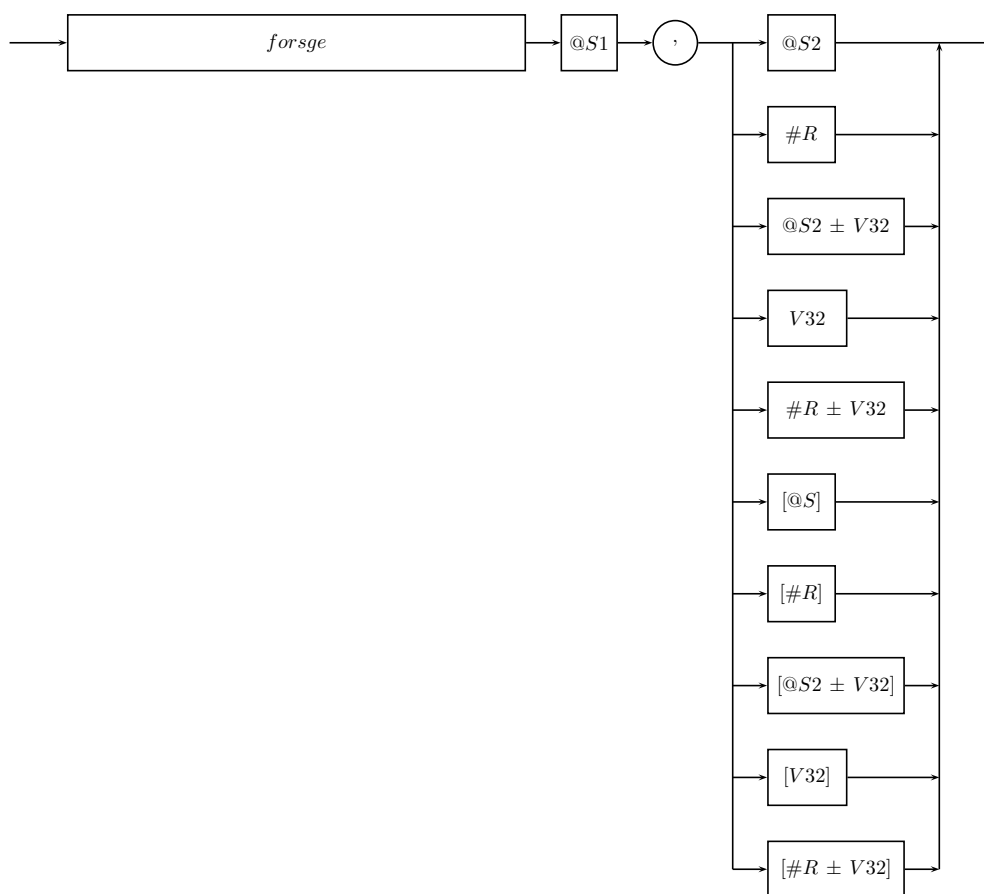


Рис. 1.29: Синтаксическое описание команды *forsgе*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* равны, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.26 forsl (FORced Select if Less)

Выбор первого или второго аргумента по условию

forsl ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

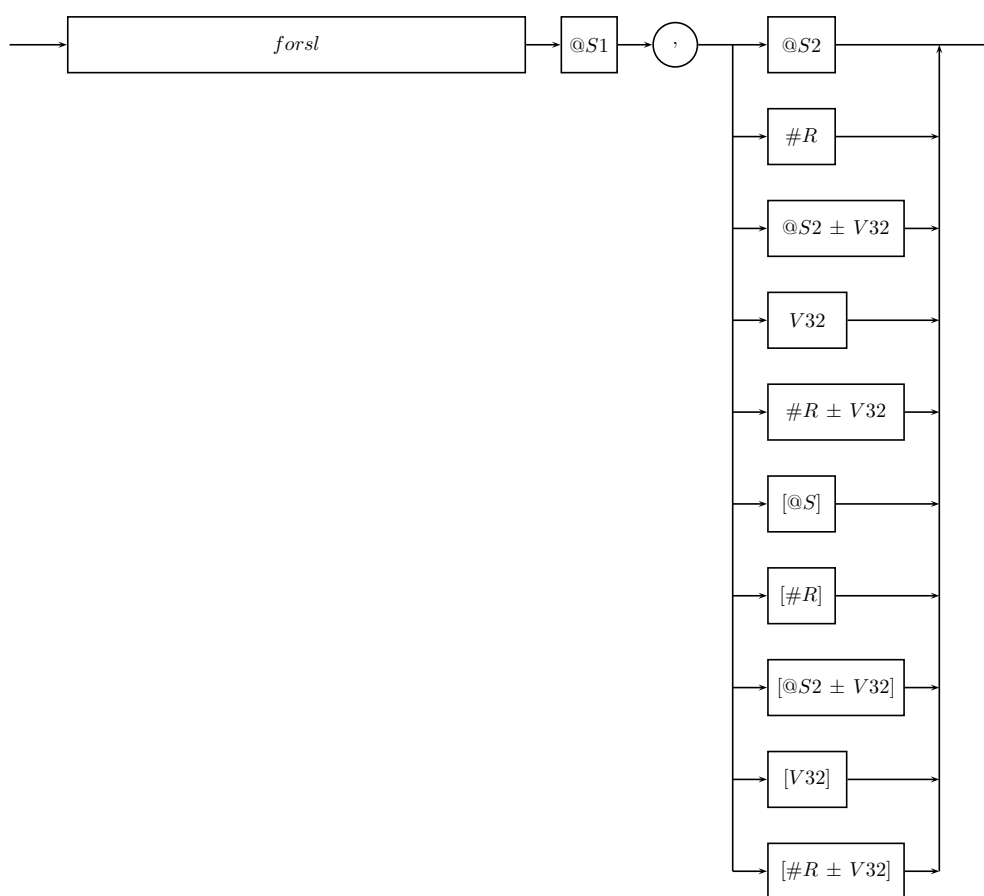


Рис. 1.30: Синтаксическое описание команды *forsl*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* не равны, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.27 forsle (FORced Select if Less and Equal)

Выбор первого или второго аргумента по условию

forsle ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

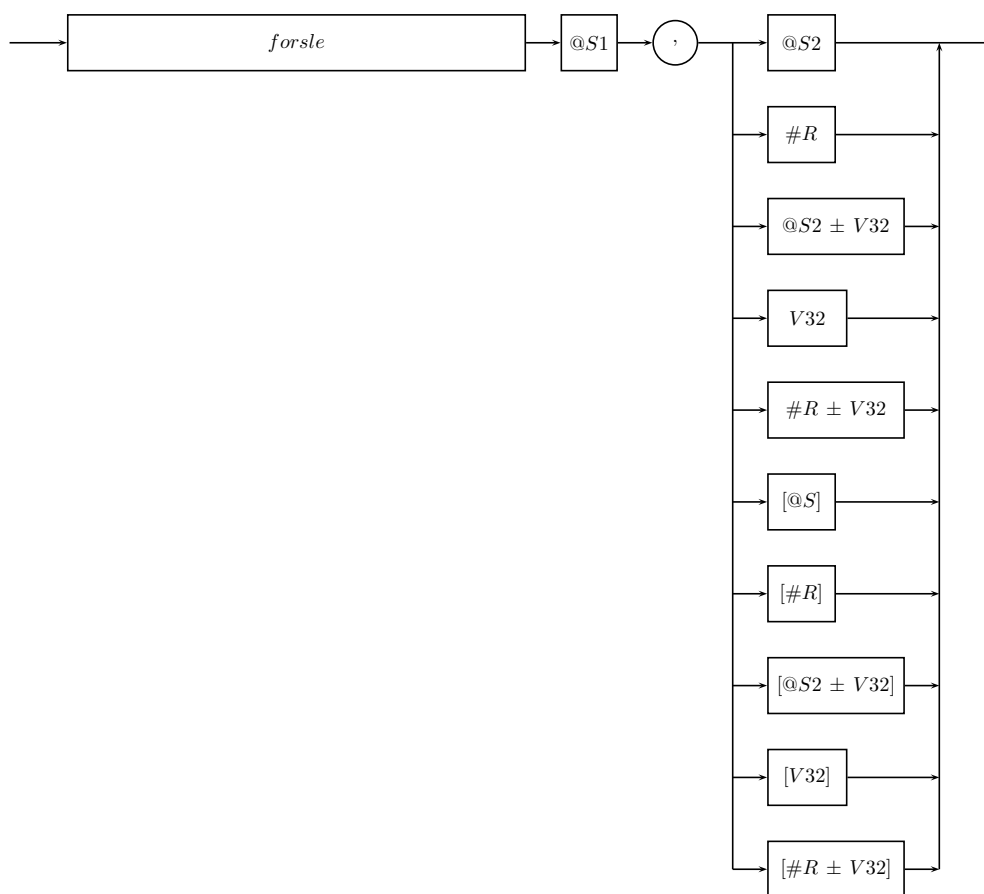


Рис. 1.31: Синтаксическое описание команды *forsle*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* не равны или флаг нуля (*ZF*) аргумента *ARG1* равен единице, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.28 forsne (FORced Select if Not Equal)

Выбор первого или второго аргумента по условию

forsne ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

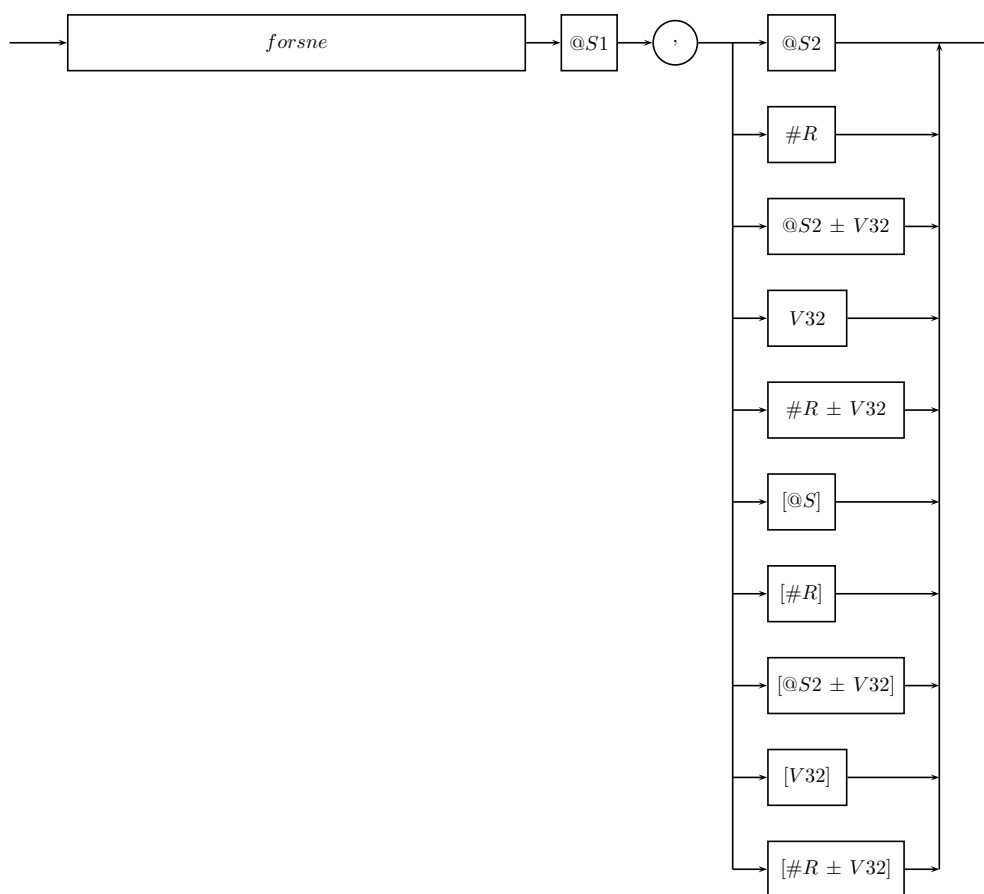


Рис. 1.32: Синтаксическое описание команды *forsne*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг нуля (*ZF*) аргумента *ARG1* равен нулю, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.29 forsno (FORced Select if Overflow flag unset)

Выбор первого или второго аргумента по условию

forsno ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

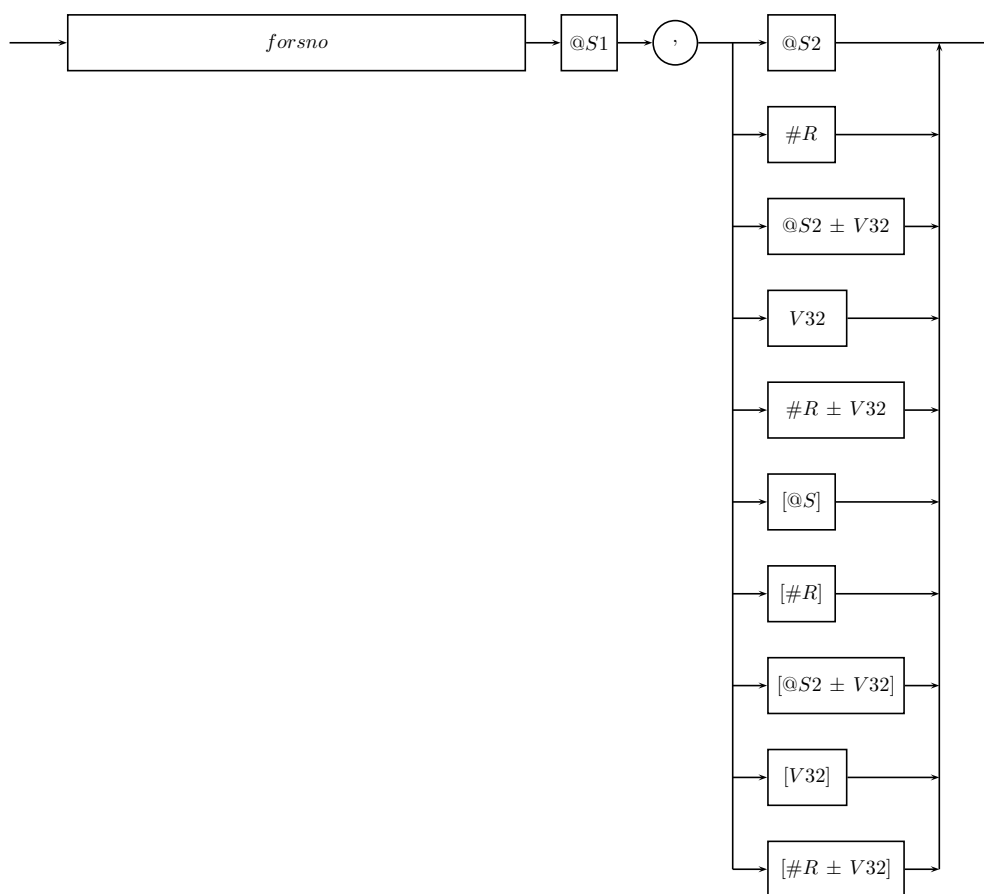


Рис. 1.33: Синтаксическое описание команды *forsno*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг переполнения (*OF*) аргумента *ARG1* равен нулю, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.30 forsns (FORced Select if Sign falg unset)

Выбор первого или второго аргумента по условию

forsns ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

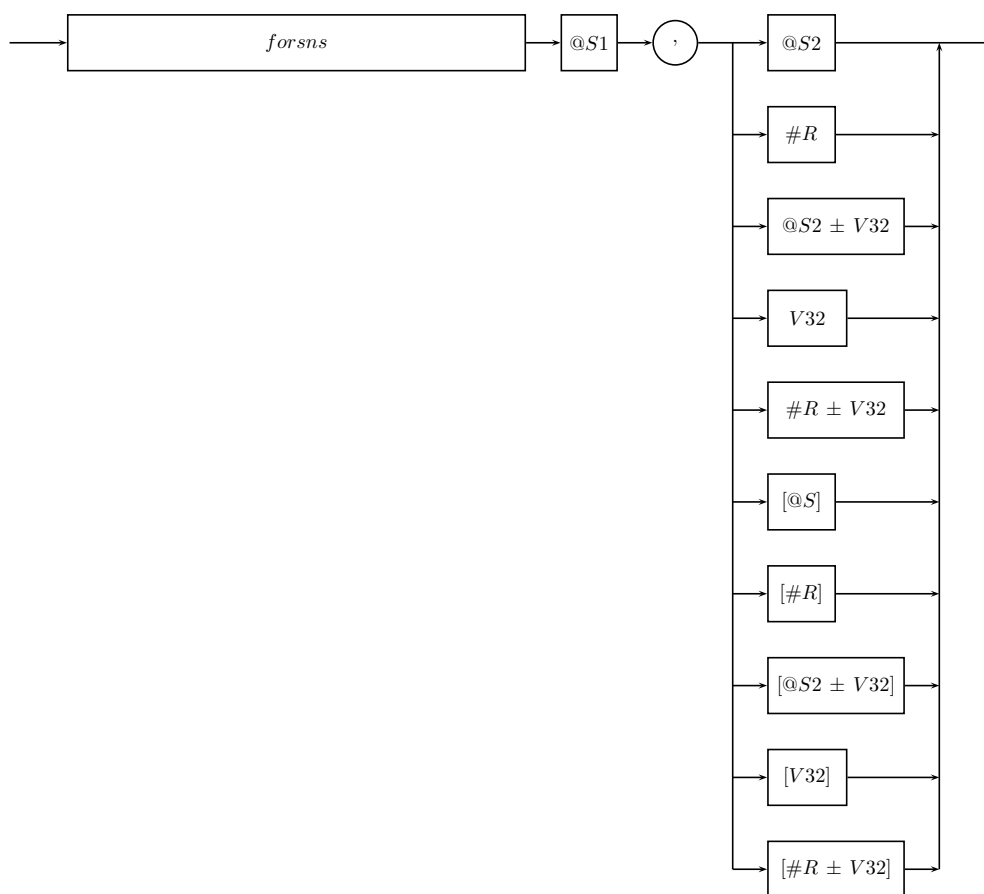


Рис. 1.34: Синтаксическое описание команды *forsns*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг знака (*ZF*) аргумента *ARG1* равен нулю, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.31 *forso* (FORced Select if Overflow flag set)

Выбор первого или второго аргумента по условию

forso ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

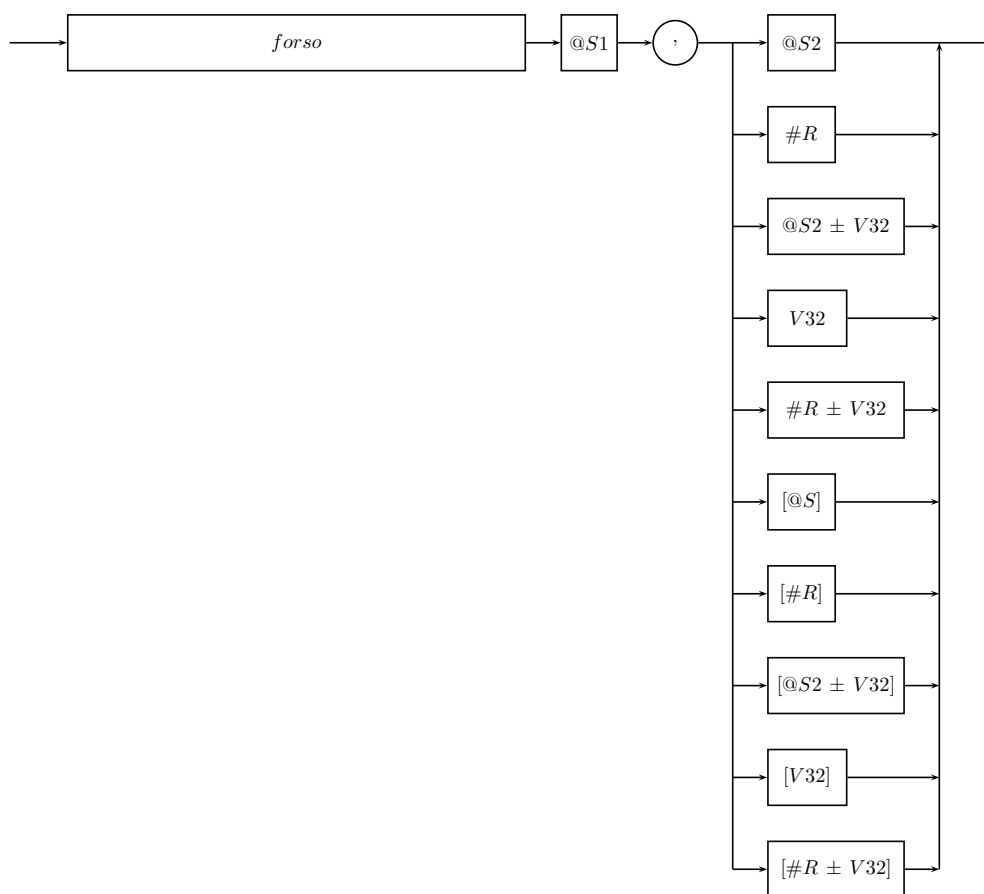


Рис. 1.35: Синтаксическое описание команды *forso*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента ARG1, если флаг переполнения (*OF*) аргумента ARG1 равен единице, иначе – равным значению аргумента ARG2;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.32 forss (FORced Select if Sign flag set)

Выбор первого или второго аргумента по условию

forss ARG1, ARG2

Назначение: команда условного выбора первого или второго аргумента

Синтаксис:

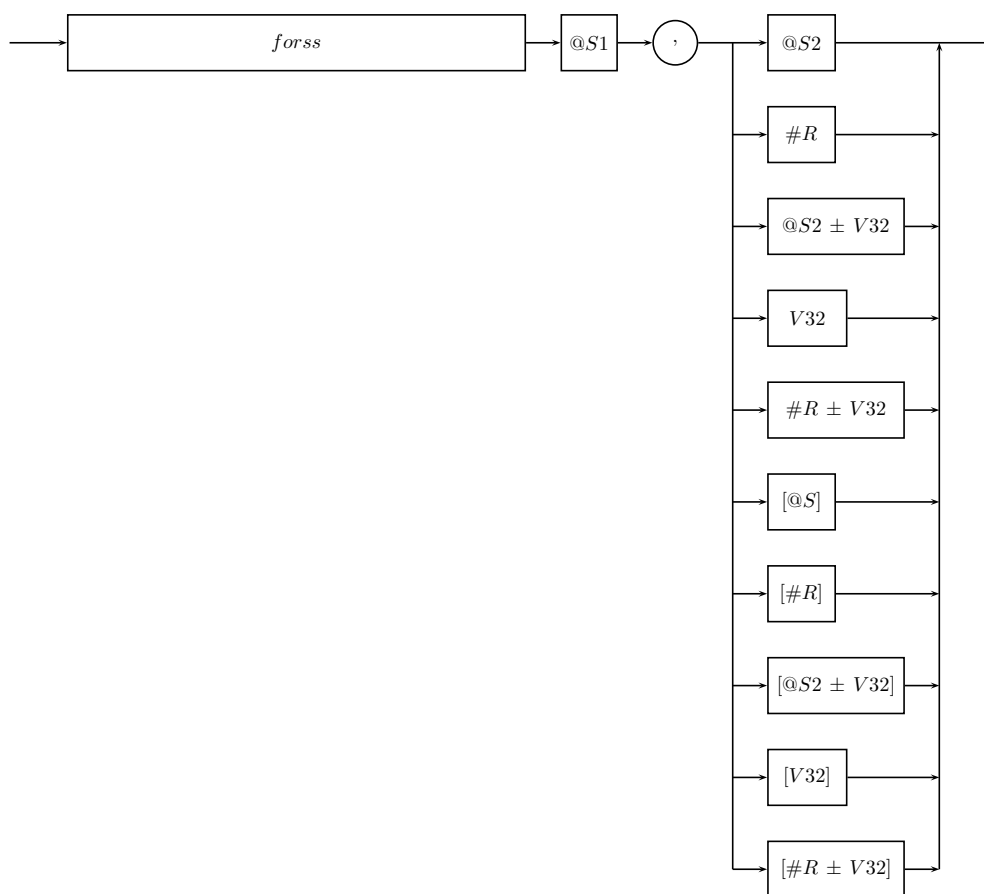


Рис. 1.36: Синтаксическое описание команды *forss*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг знака (*SF*) аргумента *ARG1* равен единице, иначе – равным значению аргумента *ARG2*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.33 ge (Greater or Equal)

Проверка условия «больше или равно».

ge ARG1, ARG2

Назначение: команда сравнения двух аргументов по критерию большего или равного

Синтаксис:

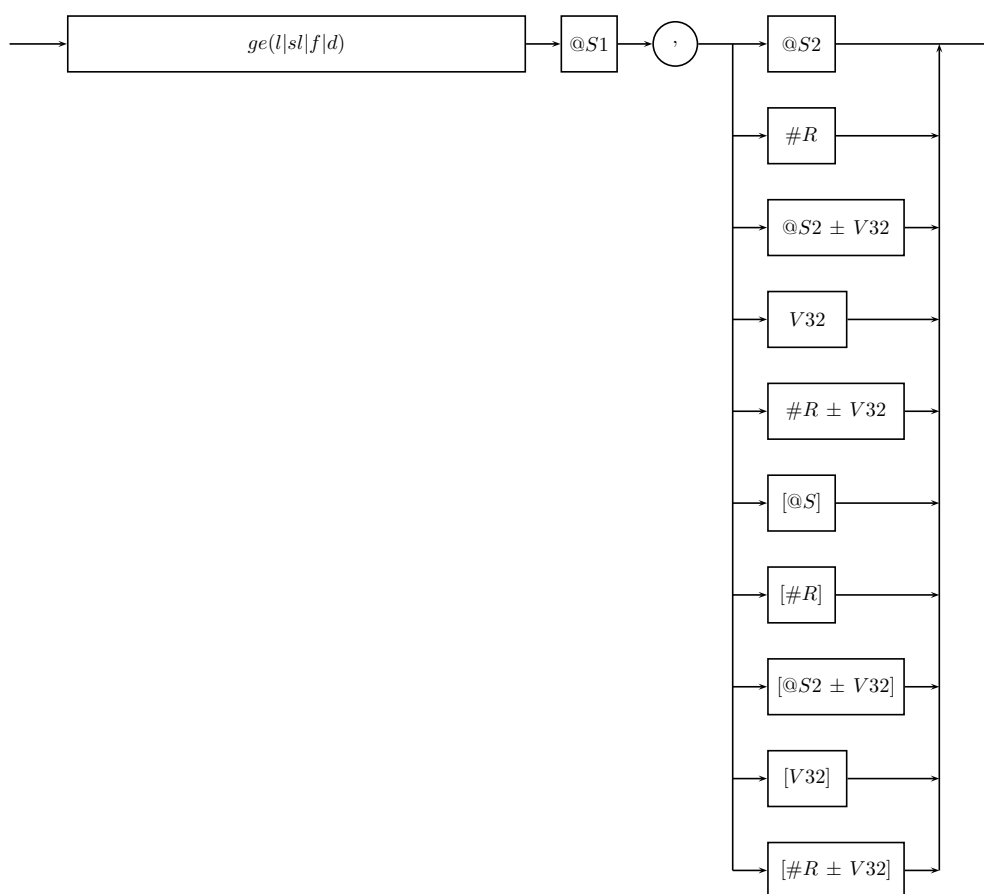


Рис. 1.37: Синтаксическое описание команды *ge*

Наличие результата: да

Алгоритм работы:

- выполнить команду сравнения двух аргументов, в качестве результата выдать «1», если *ARG1* больше или равен *ARG2*, иначе выдать «0»;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции одноразрядный, старшие 63 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.34 get (GET value)

Извлечение значения

get ARG

Назначение: операция извлечения значения

Синтаксис:

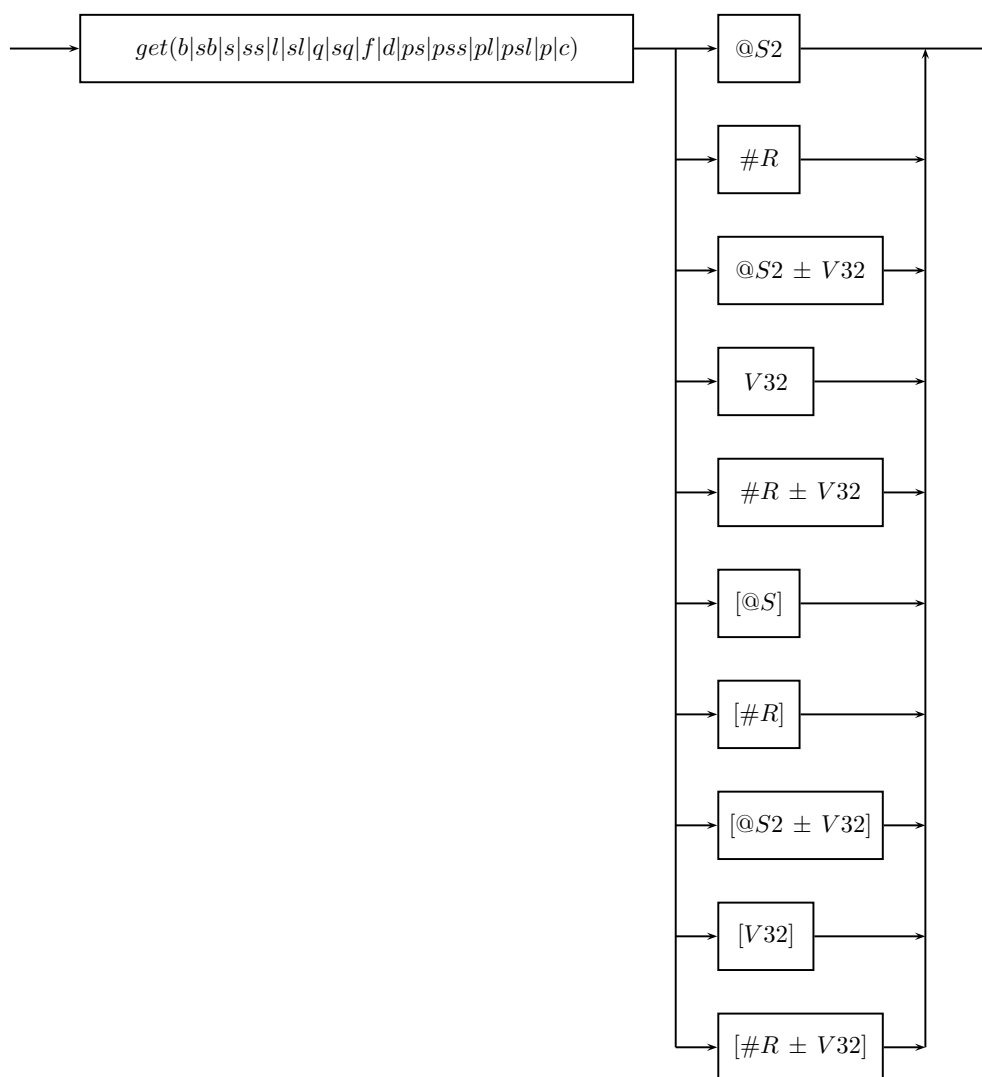


Рис. 1.38: Синтаксическое описание команды *get*

Наличие результата: да

Алгоритм работы:

- извлечь значение, заданное аргументом *ARG*;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **byte** результат операции 32-х разрядный, разряды с 8 по 31 заполняются нулём, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **signed byte** результат операции 32-х разрядный, разряды с 8 по 31 заполняются значением 7 разряда (размножение знака), старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 32-х разрядный, разряды с 16 по 31 заполняются нулём, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **signed short** результат операции 32-х разрядный, разряды с 16 по 31 заполняются значением 15 разряда (размножение знака), старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типов операций **long, signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **byte, signed byte, short, signed short, long, signed long, quad, signed quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **float, double**:

флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long, packed, complex**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

Применение: команда `get` используется

- для проброса значения, ранее сохранённого в коммутаторе (без сохранения флагов, формируются заново);
- для извлечения значения из регистра любого типа и сохранения его в коммутаторе, т. е. вне зависимости от типа регистра извлекается именно значение этого регистра;
- для сохранения константы в коммутаторе.

Пример:

```
1  .text
2
3  A:
4      getsb  -128
5      getq  #0
6      getl  #8
7      getl  #PSW
8      getq  @4
9  complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;

- в строке №3 объявляется символ (идентификатор) *A*, который является меткой в текущей секции ассемблирования (*text*), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №4 командой *getsb* помещается в коммутатор 8-ми разрядное целое знаковое число (константа -128);
- в строке №5 командой *getq* извлекается значение регистра общего назначения №0 и сохраняется в коммутаторе;
- в строке №6 командой *getl* извлекается значение базы индексного регистра №8 и сохраняется в коммутаторе (не смотря на то, что команде *getl* указан в качестве аргумента индексный регистр, формирование исполнительного адреса памяти, а также обращение к памяти, не происходит, просто извлекаются младшие 32 разряда индексного регистра и сохраняются в коммутаторе);
- в строке №7 командой *getl* извлекается значение управляющего регистра *PSW* и сохраняется в коммутаторе;
- в строке №8 командой *getq* сохраняется в коммутаторе ранее сохранённое в коммутаторе значение, т. е. сохраняется результат выполнения предшествующей команды с обновлением значений флагов результата: @4 — результат выполнения команды извлечения в строке №4;
- в строке №9 командой *complete* завершается текущий параграф.

1.4.4.35 *ib* (Involute Bytes)

Развёртка байтов

ib *ARG*

Назначение: команда развёртки байтов

Синтаксис:

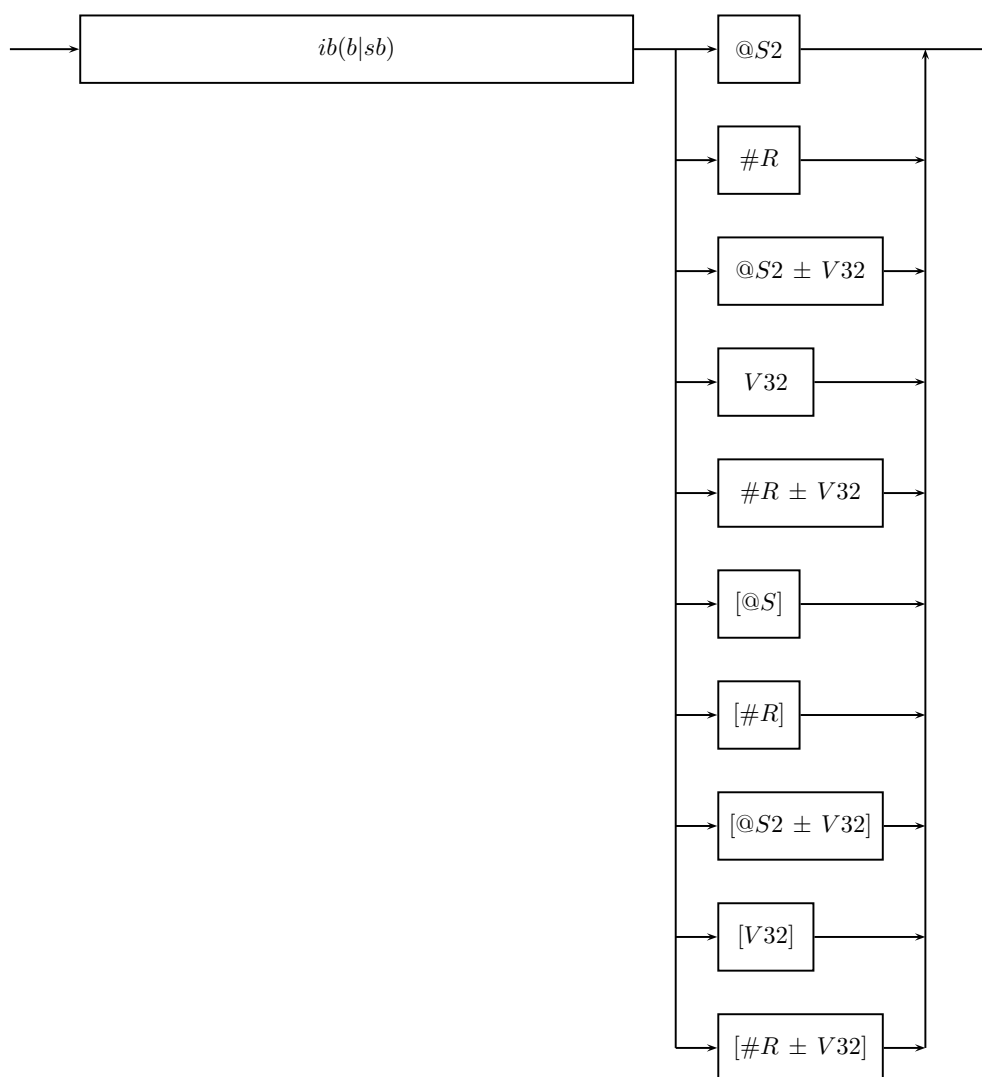


Рис. 1.39: Синтаксическое описание команды *ib*

Наличие результата: да

Алгоритм работы:

- выполнить развёртку байтов 32-х разрядного аргумента согласно типу операции (в квадратных скобках указаны номера битов аргумента и результата, нумерация битов начинается с нуля, & — операция конкатенации):

для типа операции **byte**:

$$RES[63 : 48] = 0b00000000 \& ARG[31 : 24]$$

$$RES[47 : 32] = 0b00000000 \& ARG[23 : 16]$$

$$RES[31 : 16] = 0b00000000 \& ARG[15 : 08]$$

$$RES[15 : 00] = 0b00000000 \& ARG[07 : 00]$$

для типа операции **signed byte**:

$$RES[63 : 48] = \begin{cases} 0b11111111 \& ARG[31 : 24], & \text{если } ARG[31] = 1 \\ 0b00000000 \& ARG[31 : 24], & \text{если } ARG[31] = 0 \end{cases}$$

$$RES[47 : 32] = \begin{cases} 0b11111111 \& ARG[23 : 16], & \text{если } ARG[23] = 1 \\ 0b00000000 \& ARG[23 : 16], & \text{если } ARG[23] = 0 \end{cases}$$

$$RES[31 : 16] = \begin{cases} 0b11111111 \& ARG[15 : 08], & \text{если } ARG[15] = 1 \\ 0b00000000 \& ARG[15 : 08], & \text{если } ARG[15] = 0 \end{cases}$$

$$RES[15 : 00] = \begin{cases} 0b11111111 \& ARG[07 : 00], & \text{если } ARG[07] = 1 \\ 0b00000000 \& ARG[07 : 00], & \text{если } ARG[07] = 0 \end{cases}$$

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.36 insub (INversion SUBtract)

Обратное вычитание

insub ARG1, ARG2

Назначение: операция обратного вычитания двух аргументов

Синтаксис:

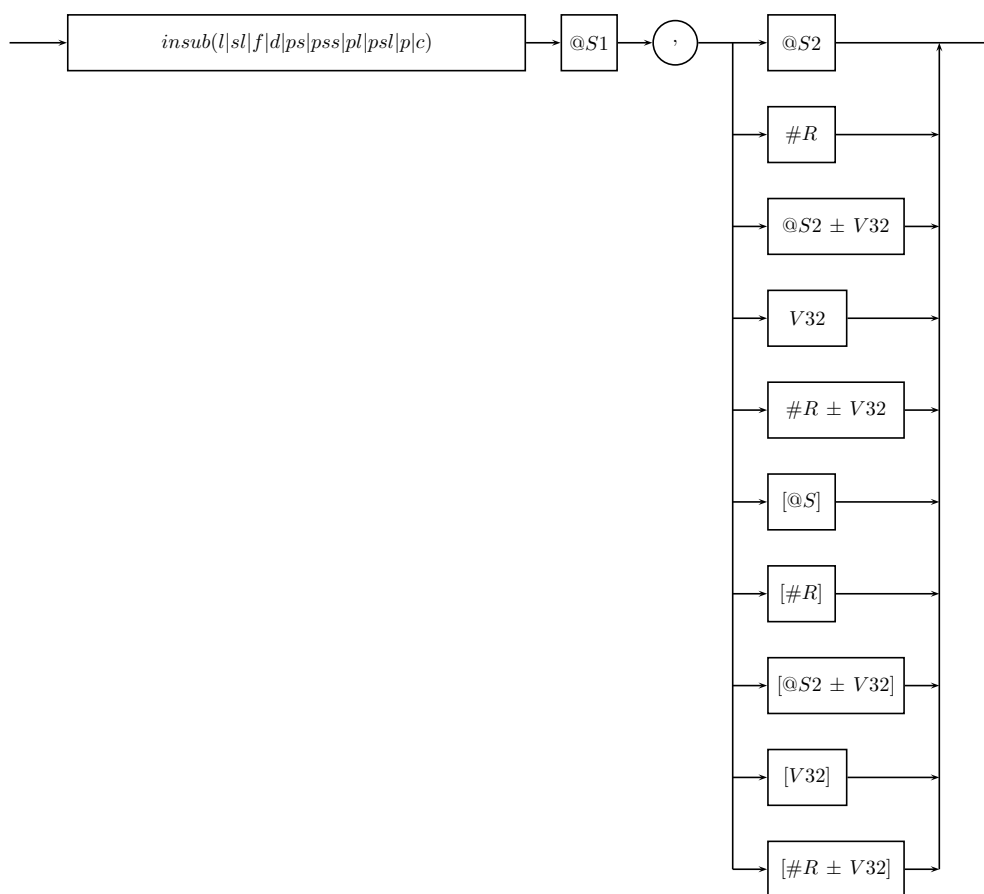


Рис. 1.40: Синтаксическое описание команды *insub*

Наличие результата: да

Алгоритм работы:

- выполнить вычитание $ARG2 - ARG1$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операции **long**, **signed long**, **float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типа операции **long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = CF$;

флаг переноса $CF = 1$, если $ARG2 < ARG1$, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типа операции **signed long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если флаг переноса CF не равен старшему биту результата и $ARG2[31] = sARG1[31]$, где число в квадратных скобках есть номер бита аргумента, а $sARG1 = (not ARG1) + 1$ (*not* — операция побитового отрицания), иначе – $OF = 0$;

флаг переноса $CF = 1$, если расширенный бит результата равен единице, а также старший бит аргумента $ARG2$ равен единице и старший бит аргумента $ARG1$ равен нулю, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **packed short**, **packed signed short**, **packed long**, **packed signed long**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

– для типов операций **float, double**:

флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если все биты экспоненты результата равны единице или расширенный бит экспоненты результата равен единице, либо хотя бы один из аргументов равен бесконечности, иначе – $OF = 0$;

флаг потери точности $IF = 1$, если в результате произошла потеря точности вследствие выравнивания мантисс или округления до ближайшего чётного, либо результатом является NaN иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

– для типов операций **packed float, complex float**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `insub` используется для обратного вычитания двух операндов, значение которых интерпретируется согласно типу операции.

Пример:

```
1  .data
2
3  B:
4      .float \
5          0f12.8 , 0f-5.6 , \
6          0f-1.78 , 0f0.19
7
8  .text
9
10 A:
11     rdq B
12     rdq B + 8
```

```
13      insubp @1, @2
14      wrq @1, B + 16
15 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) `B`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера `.float` в текущую секцию ассемблирования записывается четыре 32-х разрядных вещественных числа, начиная с текущего адреса ассемблирования (символ обратной косой черты `\` в конце строки используется для продолжения строки, т. е. строки №№4,5,6 логически являются одной строкой);
- в строке №8 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №10 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строках №№11,12 командами `rdq`, читаются из памяти данных два 64-х разрядных целых беззнаковых числа по адресам `B` и `B + 8` соответственно и помещаются в коммутатор;
- в строке №13 командой `insubp` выполняется операция обратного вычитания результатов выполнения двух предшествующих команд: `@1` — результат выполнения команды чтения в строке №11, `@2` — результат выполнения команды чтения в строке №12; оба аргумента команды `insubp` согласно суффиксу `r` интерпретируются как упакованные числа размерностью 64 бита, поэтому операция обратного вычитания выполняется независимо для старшей и младшей частей чисел; результат выполнения команды также интерпретируются как 64-х разрядное упакованное число и помещается в коммутатор;
- в строке №14 командой `wrq` осуществляется запись в память данных по адресу `B + 16` результата выполнения предшествующей команды: `@1` — результат выполнения команды обратного вычитания в строке №13;

– в строке №15 командой complete завершается текущий параграф.

1.4.4.37 irm (Index Register Modification)

Установка маски модификации значения индексных регистров, безусловная

irm ARG

Назначение: команда безусловной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG*

Синтаксис:

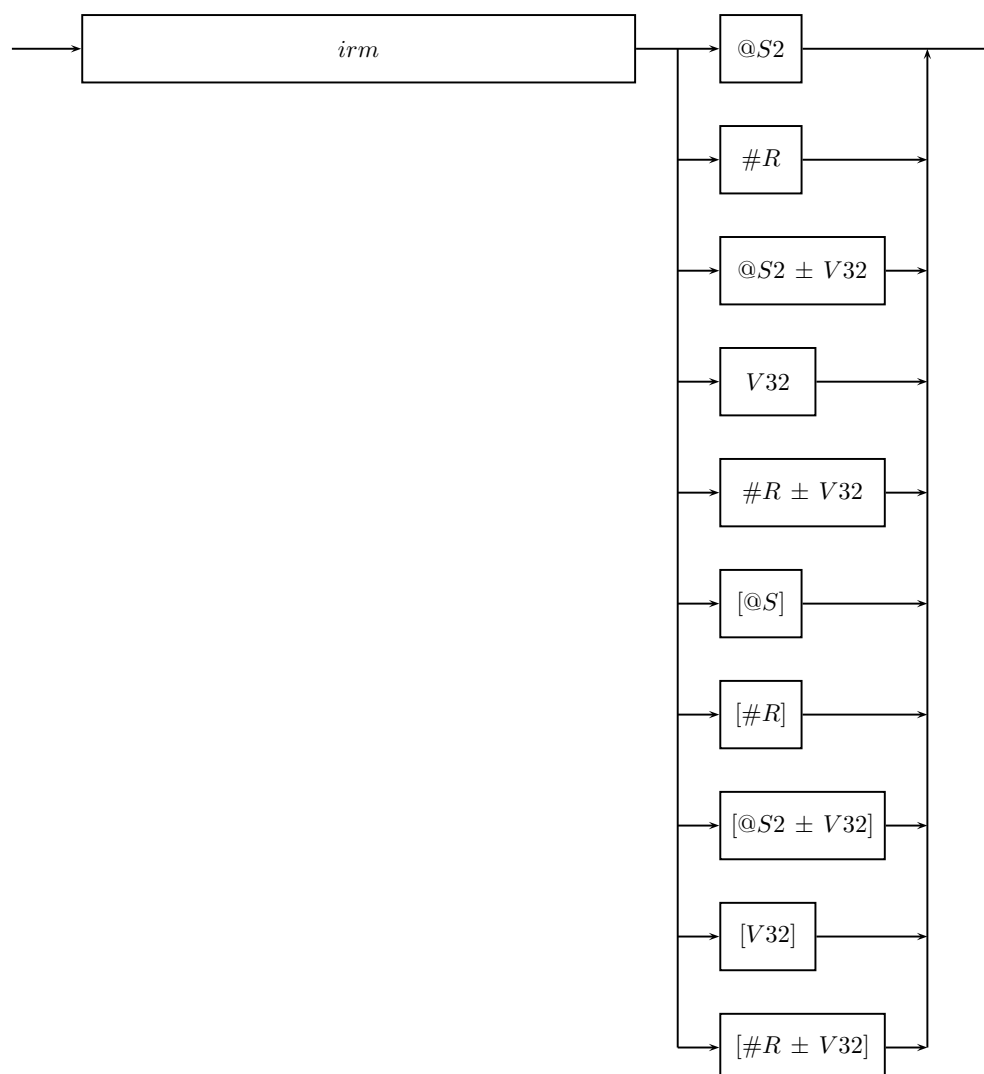


Рис. 1.41: Синтаксическое описание команды *irm*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG*.

Применение: команда *itm* используется для безусловной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента *ARG*: если *i*-ый разряд аргумента *ARG* равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра №(*i* + 8).

1.4.4.38 *irma* (Index Register Modification if Above)

Установка маски модификации значения индексных регистров, условная

irma ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

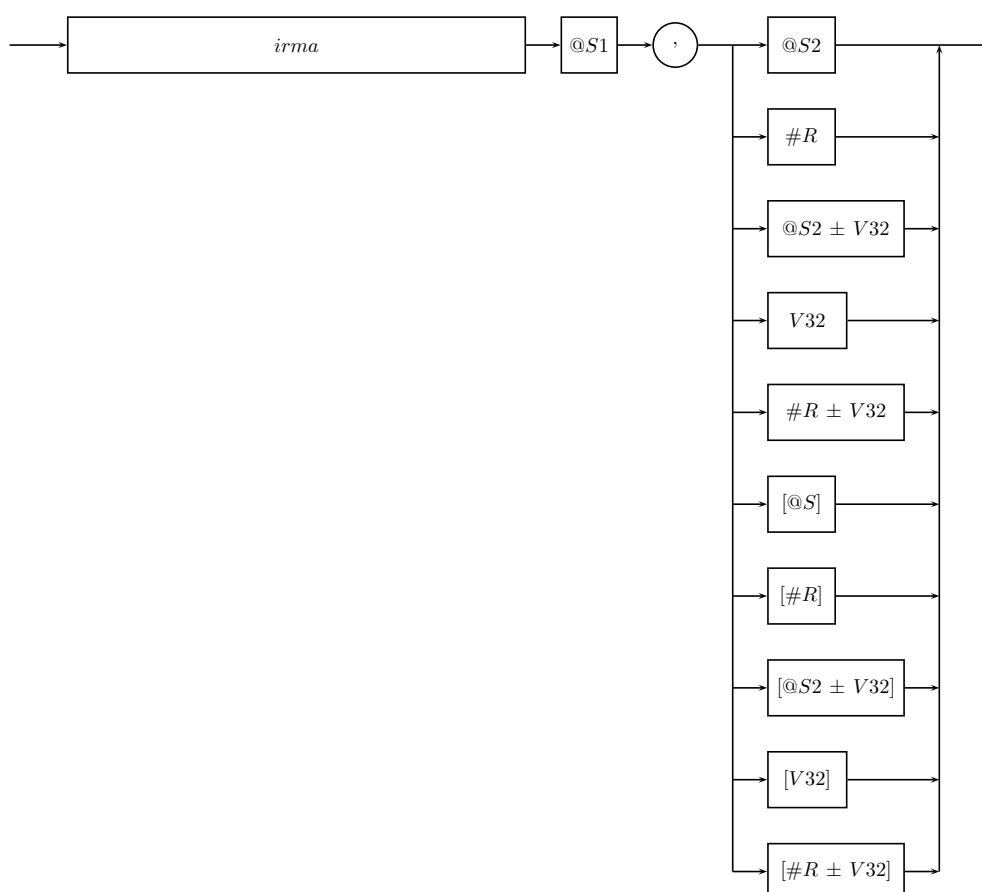


Рис. 1.42: Синтаксическое описание команды *irma*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаги переноса (*CF*) и нуля (*ZF*) аргумента

$ARG1$ равны нулю.

Применение: команда `igta` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра №($i + 8$).

1.4.4.39 *irmae* (Index Register Modification if Above and Equal) / *irmnc* (Index Register Modification if Carry flag unset)

Установка маски модификации значения индексных регистров, условная

(irmae/irmnc) *ARG1*, *ARG2*

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

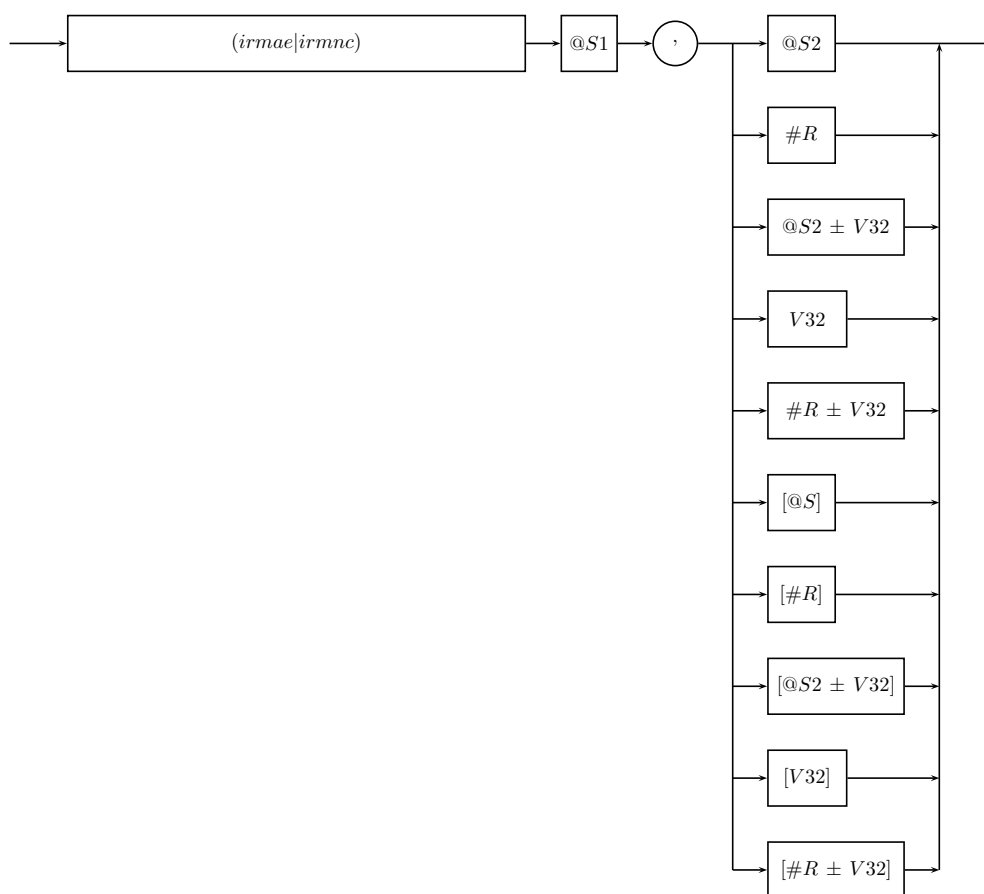


Рис. 1.43: Синтаксическое описание команды *irmae/irmnc*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента $ARG2$, если флаг переноса (CF) аргумента $ARG1$ равен нулю.

Применение: команда `irmas/irmns` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $\text{№}(i + 8)$.

1.4.4.40 *irmb* (Index Register Modification if Below) / *irmc* (Index Register Modification if Carry flag set)

Установка маски модификации значения индексных регистров, условная

(irmb/irmc) ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

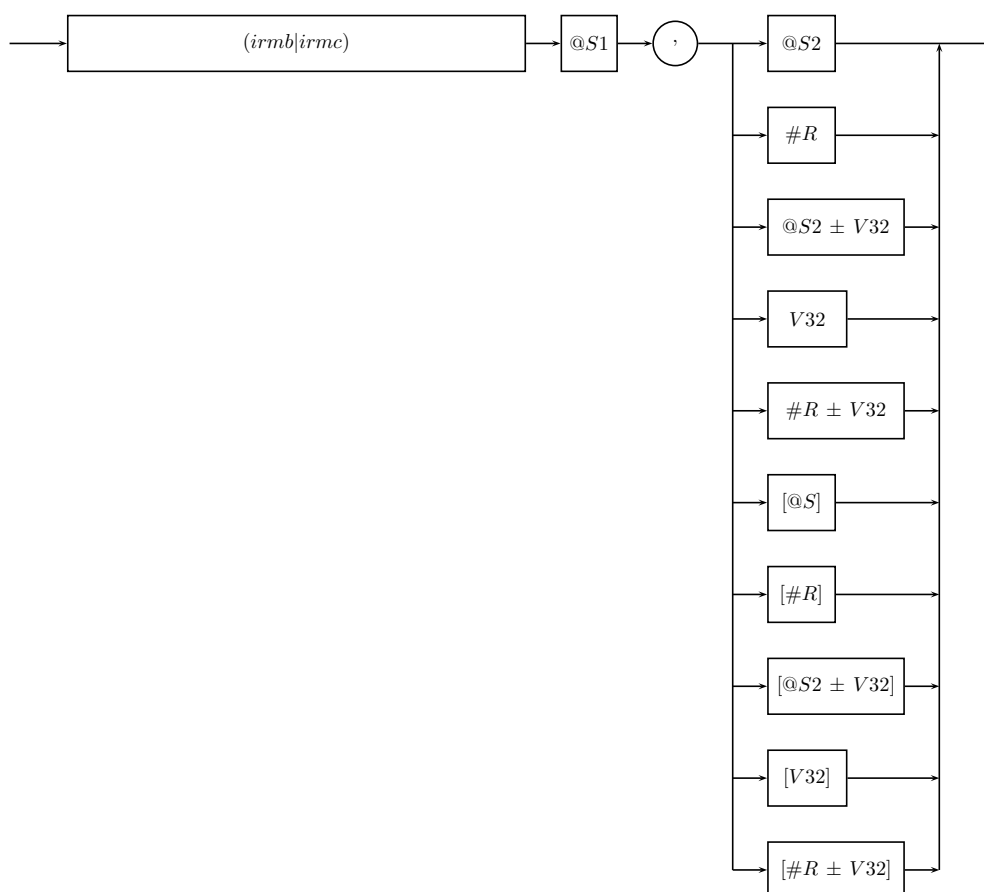


Рис. 1.44: Синтаксическое описание команды *irmb/irmc*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента $ARG2$, если флаг переноса (CF) аргумента $ARG1$ равен единице.

Применение: команда `immb/imms` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $\text{№}(i + 8)$.

1.4.4.41 irmbe (Index Register Modification if Below and Equal)

Установка маски модификации значения индексных регистров, условная

irmbe ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

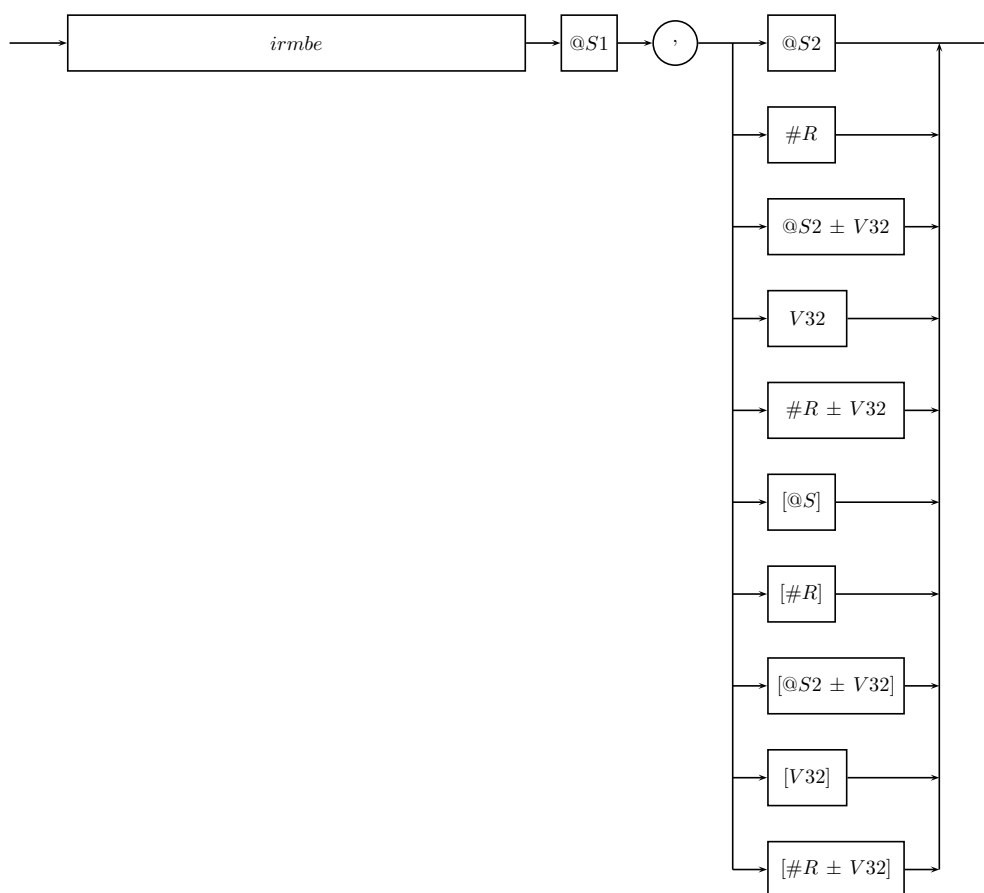


Рис. 1.45: Синтаксическое описание команды *irmbe*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаги переноса (*CF*) и нуля (*ZF*) аргумента

$ARG1$ равны единице.

Применение: команда `imbe` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $\mathbb{N}(i + 8)$.

1.4.4.42 irme (Index Register Modification if Equal)

Установка маски модификации значения индексных регистров, условная

irme ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

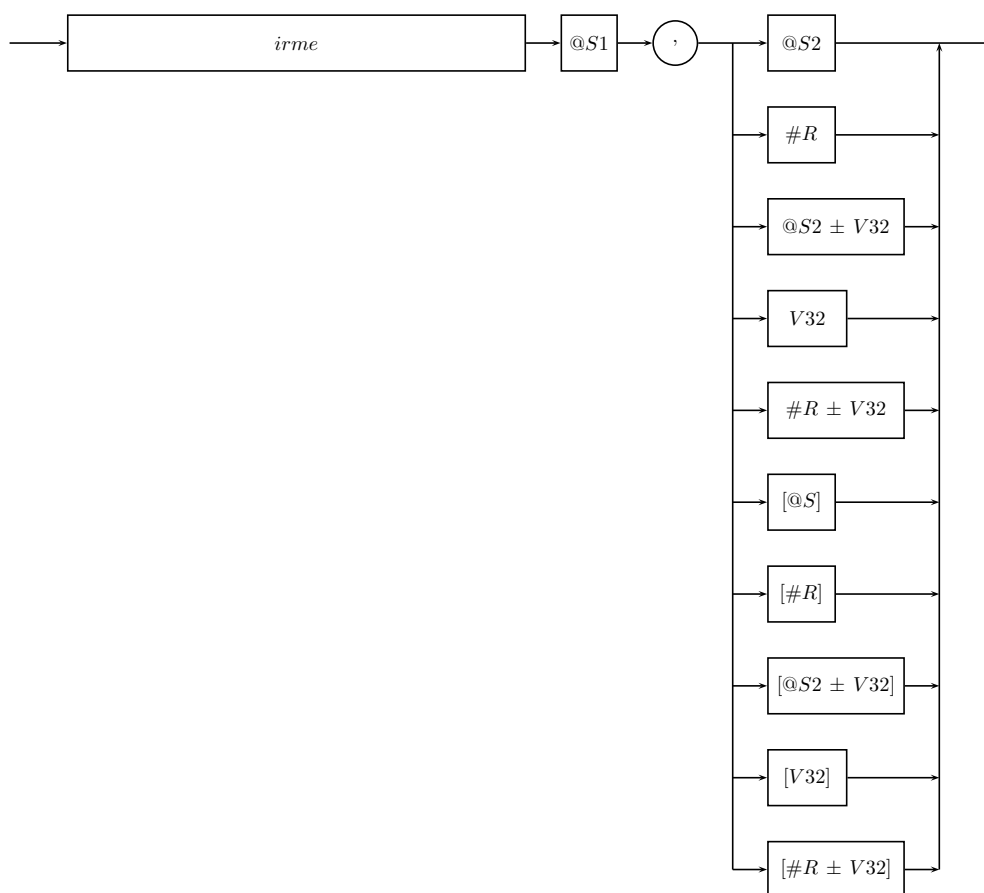


Рис. 1.46: Синтаксическое описание команды *irme*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаг нуля (*ZF*) аргумента *ARG1* равен едини-

це.

Применение: команда `irm` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента *ARG2*: если *i*-ый разряд аргумента *ARG2* равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра №(*i* + 8).

1.4.4.43 irmg (Index Register Modification if Greater)

Установка маски модификации значения индексных регистров, условная

irmg *ARG1*, *ARG2*

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

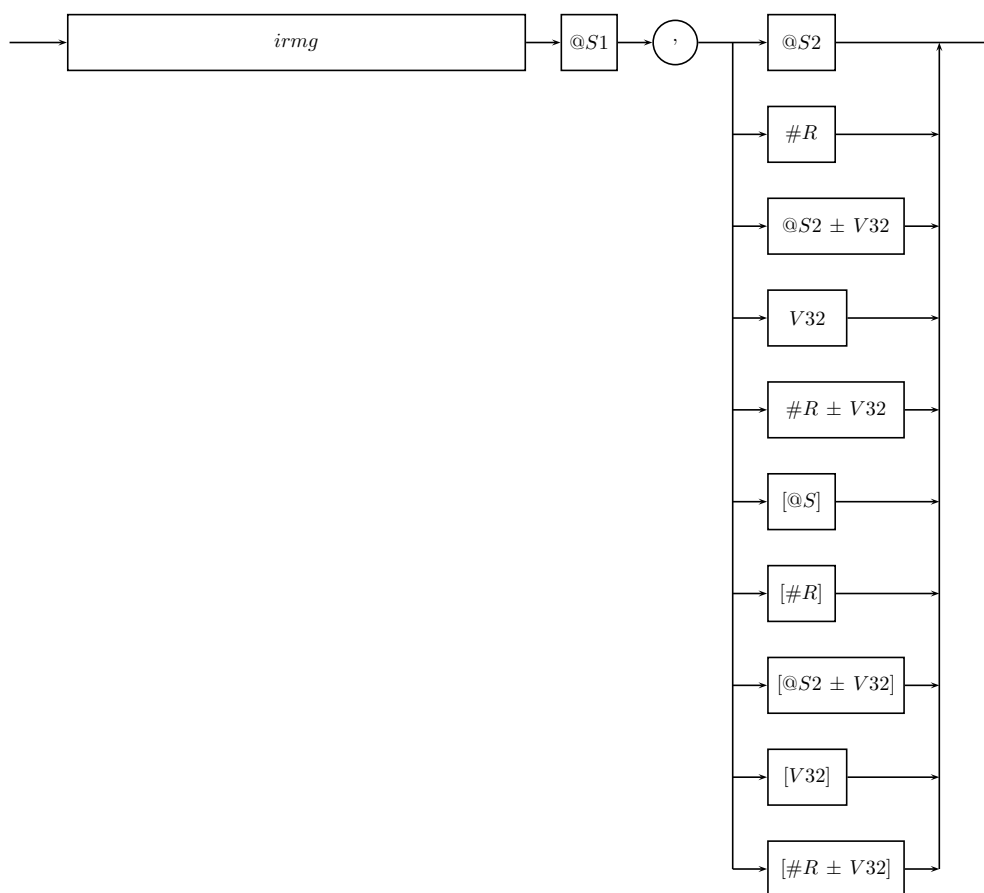


Рис. 1.47: Синтаксическое описание команды *irmg*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаги знака (*SF*) и переполнения (*OF*) аргу-

мента $ARG1$ равны и флаг нуля (ZF) аргумента $ARG1$ равен нулю.

Применение: команда `irmg` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $N^{\circ}(i + 8)$.

1.4.4.44 irmge (Index Register Modification if Greater and Equal)

Установка маски модификации значения индексных регистров, условная

irmge ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента ARG2

Синтаксис:

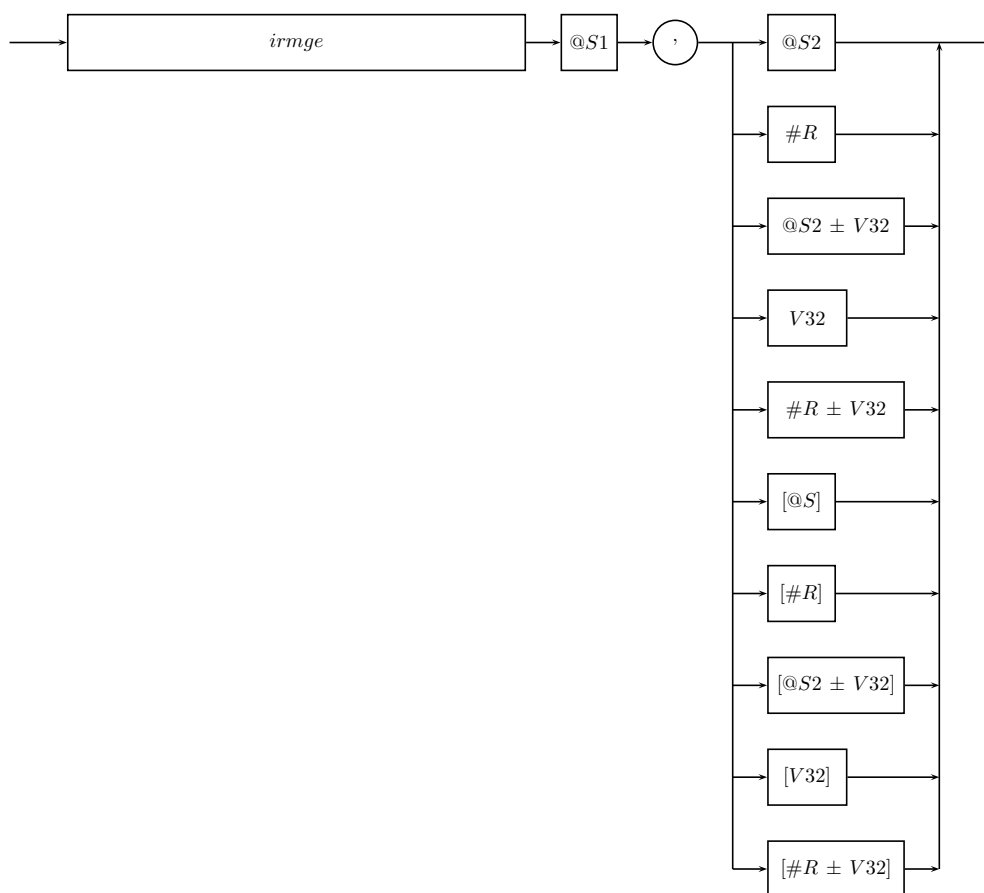


Рис. 1.48: Синтаксическое описание команды *irmge*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента ARG2, если флаги знака (*SF*) и переполнения (*OF*) аргу-

мента $ARG1$ равны.

Применение: команда `itnge` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $\mathbb{N}(i + 8)$.

1.4.4.45 *irml* (Index Register Modification if Less)

Установка маски модификации значения индексных регистров, условная

irml *ARG1*, *ARG2*

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

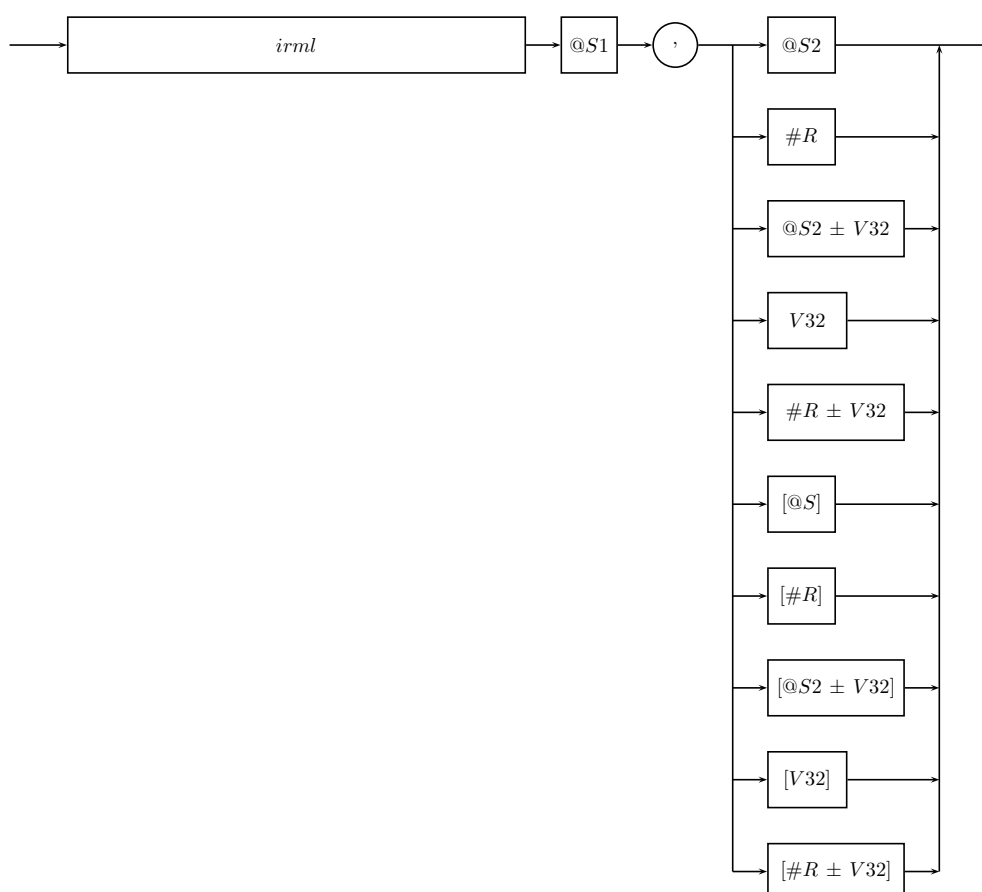


Рис. 1.49: Синтаксическое описание команды *irml*

Наличие результата: нет

Алгоритм работы:

- уустановить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаги знака (*SF*) и переполнения (*OF*) аргу-

мента $ARG1$ не равны.

Применение: команда `irml` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $N^{\circ}(i + 8)$.

1.4.4.46 irmle (Index Register Modification if Less and Equal)

Установка маски модификации значения индексных регистров, условная

irmle ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

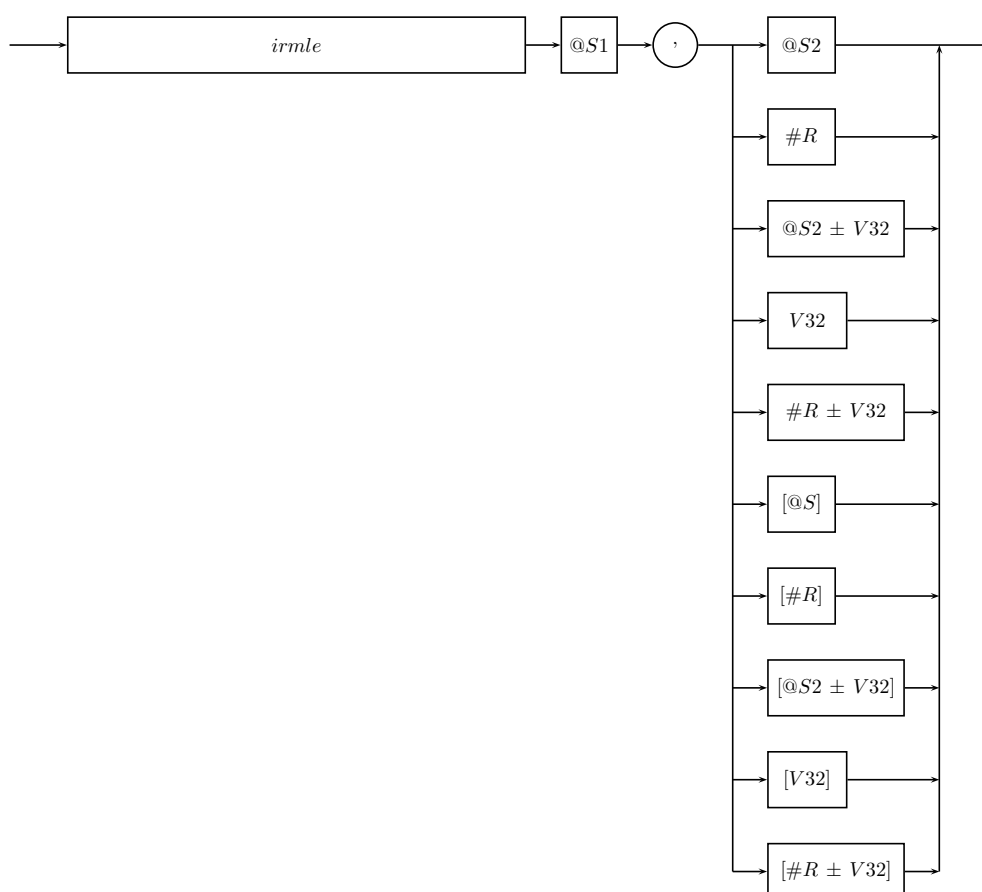


Рис. 1.50: Синтаксическое описание команды *irmle*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаги знака (*SF*) и переполнения (*OF*) аргу-

мента $ARG1$ не равны или флаг нуля (ZF) аргумента $ARG1$ равен единице.

Применение: команда `irmle` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента $ARG2$: если i -ый разряд аргумента $ARG2$ равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $\text{№}(i + 8)$.

1.4.4.47 irmne (Index Register Modification if Not Equal)

Установка маски модификации значения индексных регистров, условная

irmne ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента ARG2

Синтаксис:

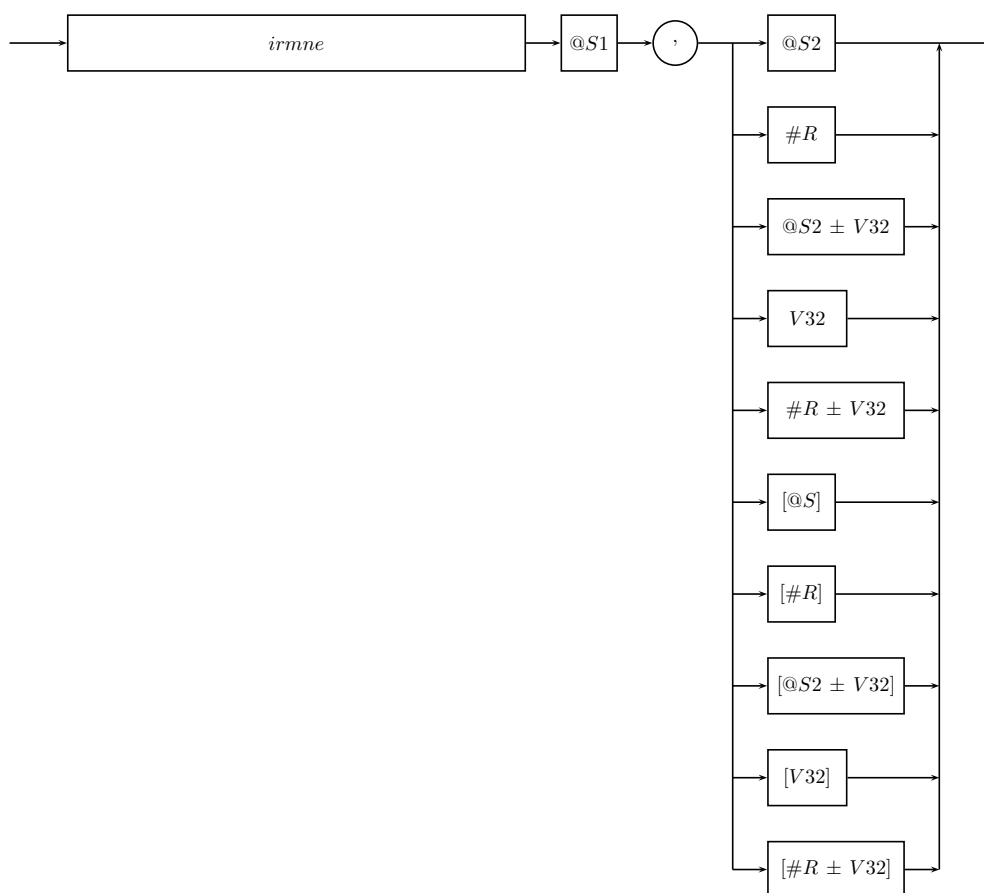


Рис. 1.51: Синтаксическое описание команды *irmne*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента ARG2, если флаг нуля (ZF) аргумента ARG1 равен нулю.

Применение: команда `igmne` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента *ARG2*: если *i*-ый разряд аргумента *ARG2* равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $N^{\circ}(i + 8)$.

1.4.4.48 irmno (Index Register Modification if Overflow flag unset)

Выбор первого или второго аргумента по условию

irmno ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента ARG2

Синтаксис:

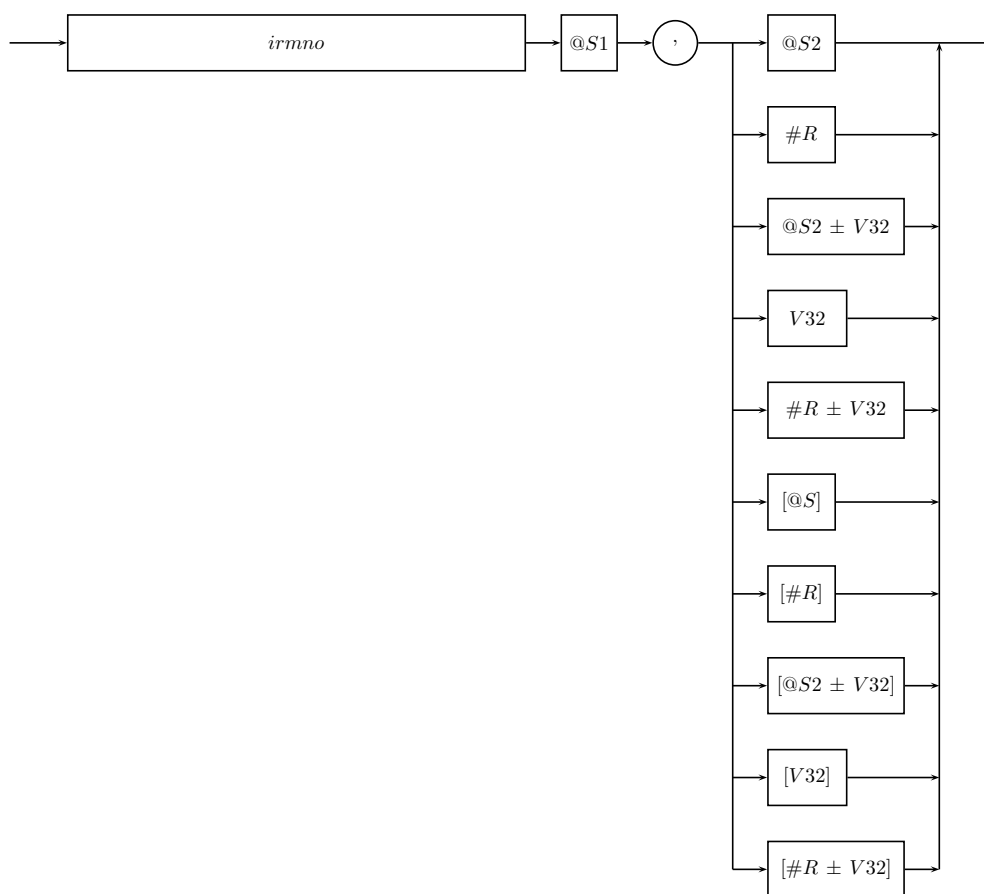


Рис. 1.52: Синтаксическое описание команды *irmno*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента ARG2, если флаг переполнения (*OF*) аргумента ARG1 равен нулю.

Применение: команда `igmno` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента *ARG2*: если *i*-ый разряд аргумента *ARG2* равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $N^{\circ}(i + 8)$.

1.4.4.49 irmns (Index Register Modification if Sign flag unset)

Установка маски модификации значения индексных регистров, условная

irmns *ARG1*, *ARG2*

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

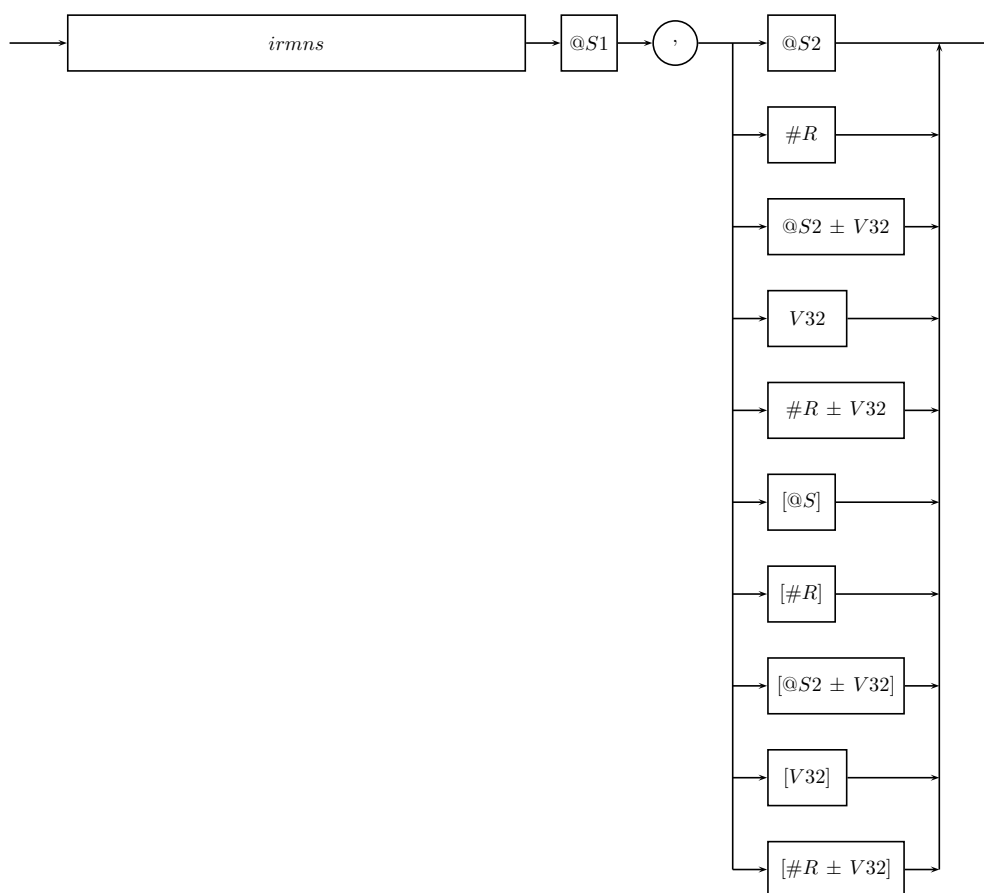


Рис. 1.53: Синтаксическое описание команды *irmns*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаг знака (*ZF*) аргумента *ARG1* равен нулю.

Применение: команда `irmns` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента *ARG2*: если *i*-ый разряд аргумента *ARG2* равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра $N^{\circ}(i + 8)$.

1.4.4.50 irmo (Index Register Modification if Overflow flag set)

Установка маски модификации значения индексных регистров, условная

irmo ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

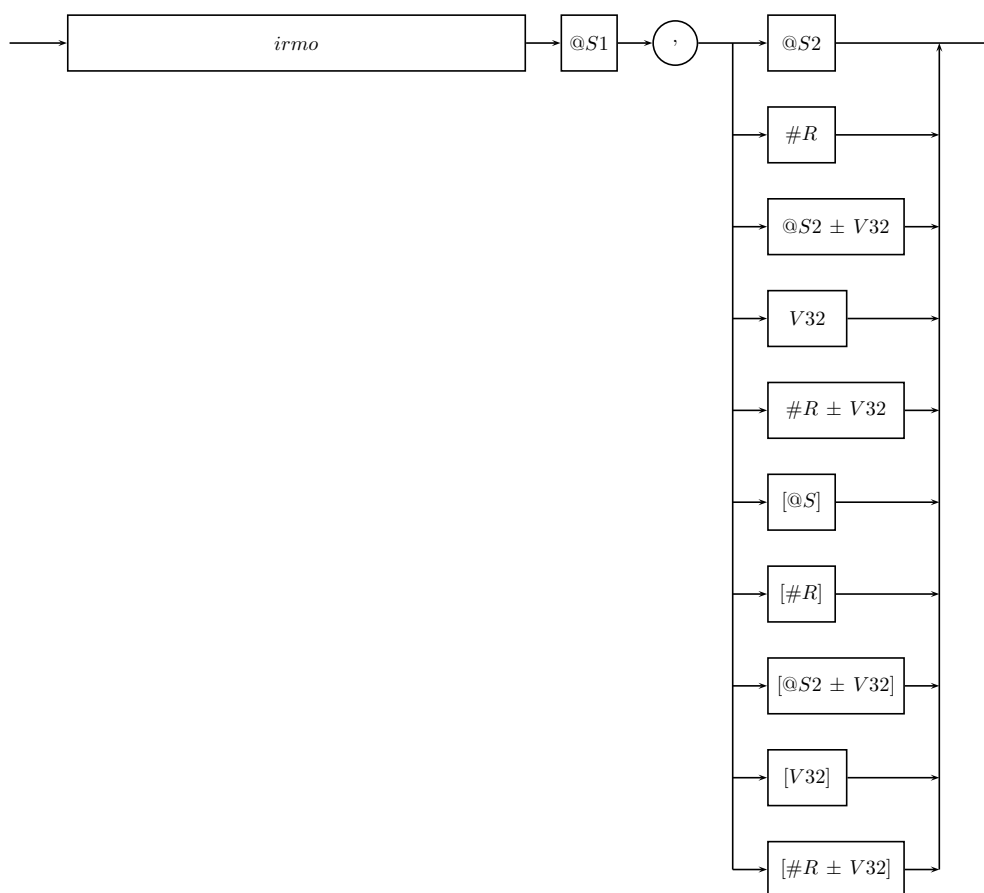


Рис. 1.54: Синтаксическое описание команды *irmo*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаг переполнения (*OF*) аргумента *ARG1* ра-

вен единице.

Применение: команда `itmo` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента *ARG2*: если *i*-ый разряд аргумента *ARG2* равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра №(*i* + 8).

1.4.4.51 irms (Index Register Modification if Sign flag set)

Установка маски модификации значения индексных регистров, условная

irms ARG1, ARG2

Назначение: команда условной установки маски модификации значения индексных регистров по завершению текущего параграфа, определяемой значением аргумента *ARG2*

Синтаксис:

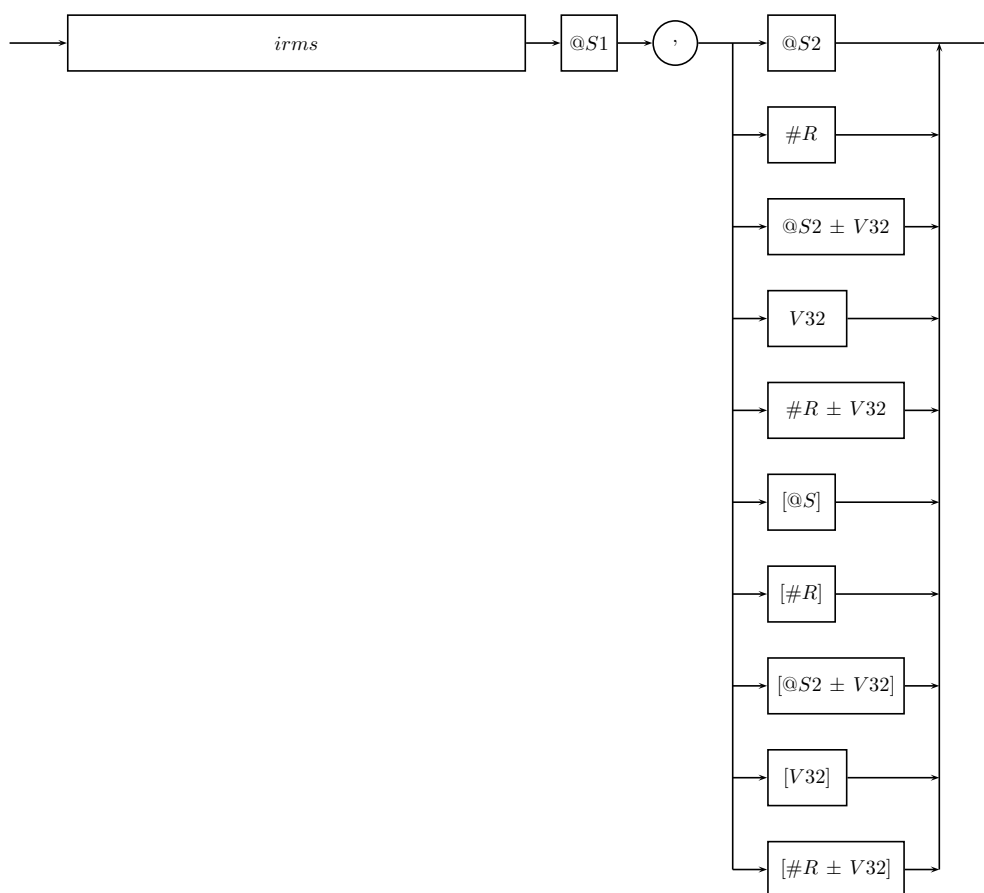


Рис. 1.55: Синтаксическое описание команды *irms*

Наличие результата: нет

Алгоритм работы:

- установить маску модификации значения индексных регистров, равным значению аргумента *ARG2*, если флаг знака (*SF*) аргумента *ARG1* равен еди-

нице.

Применение: команда `irms` используется для условной установки маски модификации значения индексных регистров по завершению текущего параграфа. Номера индексных регистров, значение которых необходимо модифицировать по окончании текущего параграфа, определяются значением младших восьми (по количеству индексных регистров) разрядов аргумента *ARG2*: если *i*-ый разряд аргумента *ARG2* равен единице, то необходимо модифицировать по окончании параграфа значение индексного регистра №($i + 8$).

1.4.4.52 ja (Jump if Above)

Установка адреса следующего параграфа, условная

ja ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

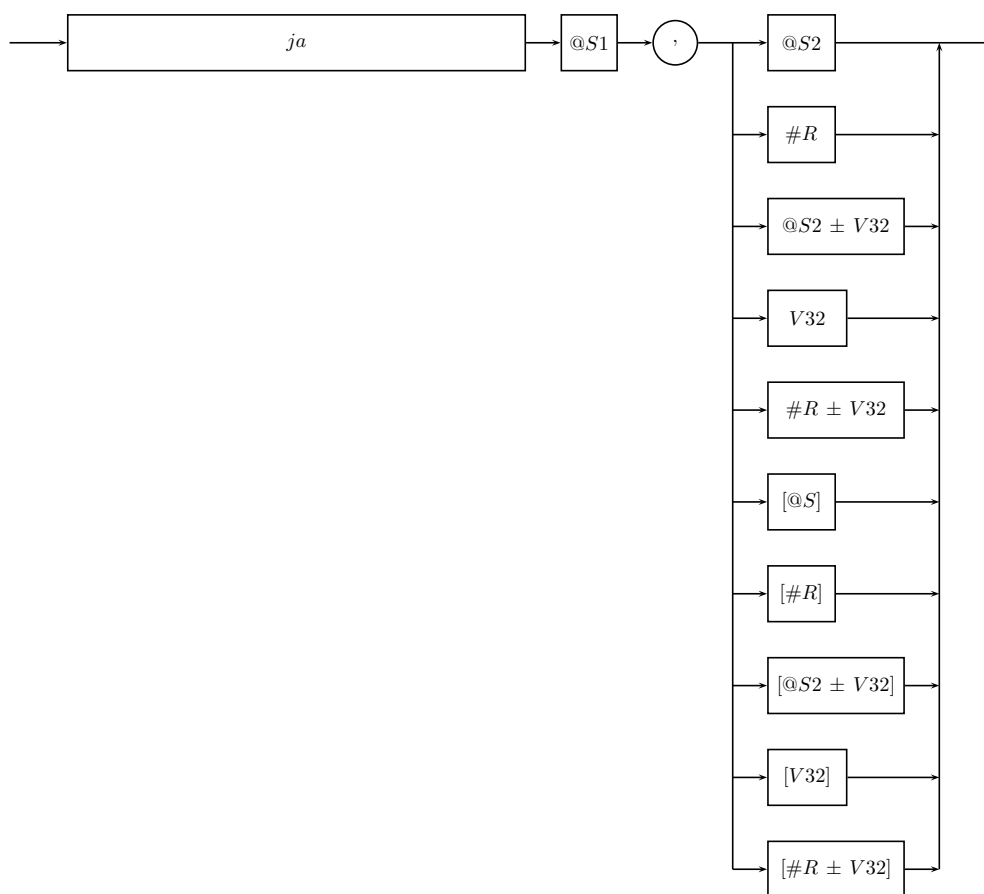


Рис. 1.56: Синтаксическое описание команды *ja*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг переноса (*CF*) аргумента *ARG1* равен нулю и флаг нуля (*ZF*) аргумента *ARG1* равен нулю;

1.4.4.53 jae (Jump if Above and Equal) / jnc (Jump if Carry flag unset)

Установка адреса следующего параграфа, условная

$(jae|jnc) ARG1, ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

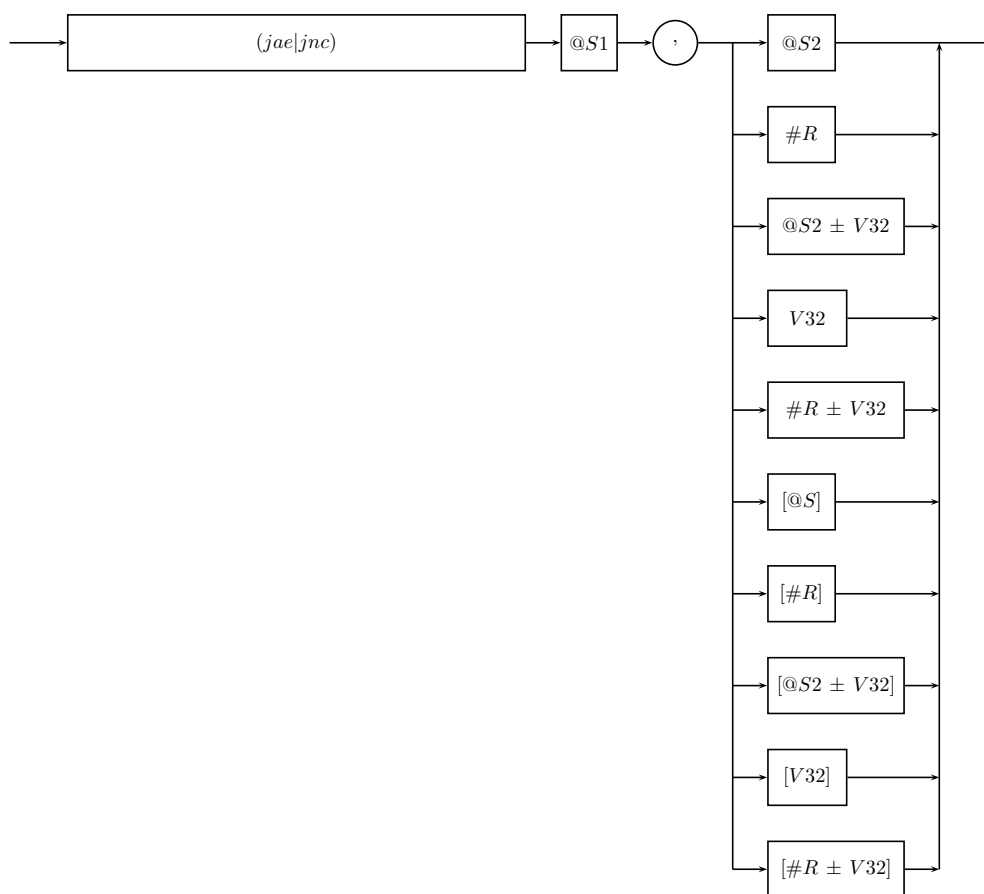


Рис. 1.57: Синтаксическое описание команды jae/jnc

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента $ARG2$ (фактический переход будет осуществлён по окончании текущего параграфа), если флаг переноса (CF) аргумента $ARG1$ равен нулю;

1.4.4.54 *jb* (Jump if Below) / *jc* (Jump if Carry flag set)

Установка адреса следующего параграфа, условная

$(jb|jc) ARG1, ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

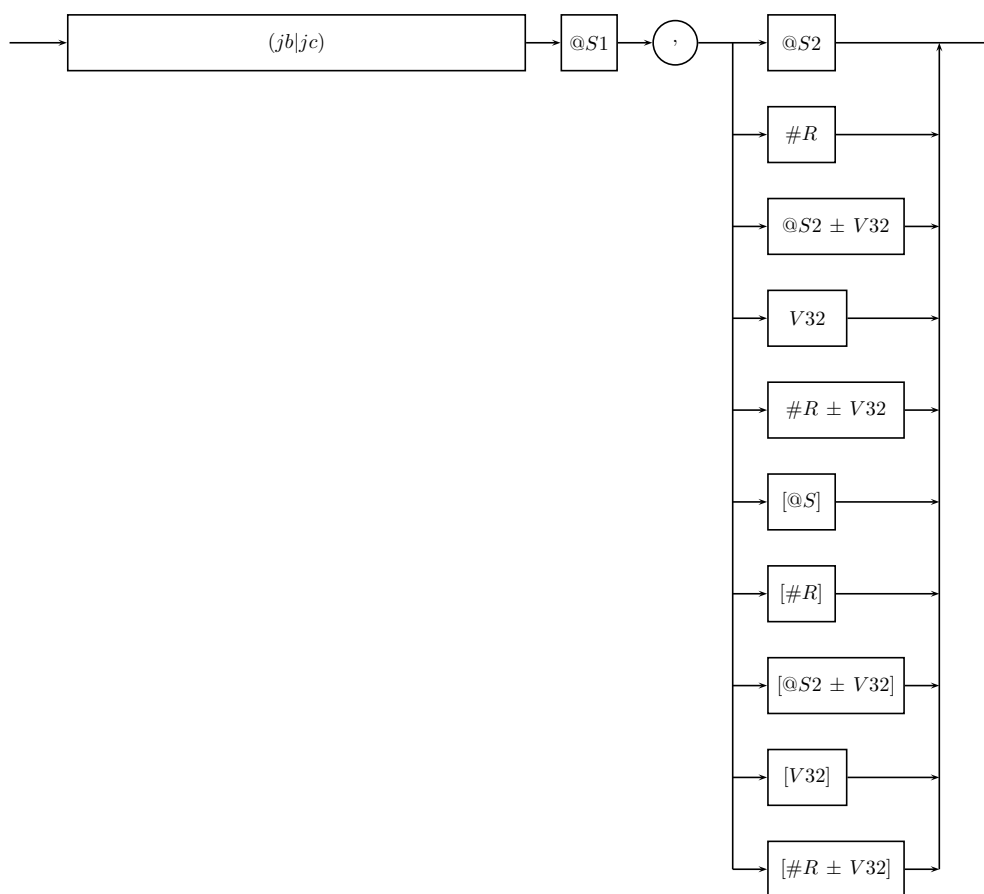


Рис. 1.58: Синтаксическое описание команды *jb/jc*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг переноса (*CF*) аргумента *ARG1* равен единице;

1.4.4.55 jbe (Jump if Below and Equal)

Установка адреса следующего параграфа, условная

jbe ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

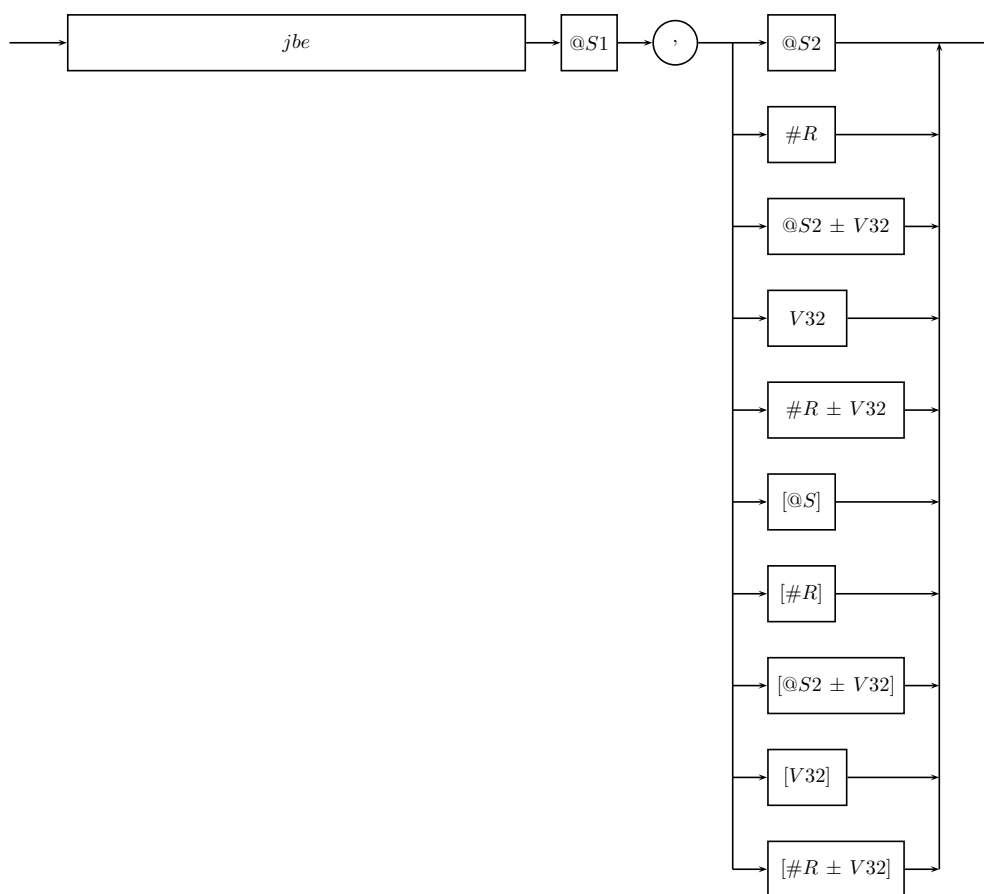


Рис. 1.59: Синтаксическое описание команды *jbe*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг переноса (*CF*) аргумента *ARG1* равен единице и флаг нуля (*ZF*) аргумента *ARG1* равен единице;

1.4.4.56 je (Jump if Equal)

Установка адреса следующего параграфа, условная

je ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

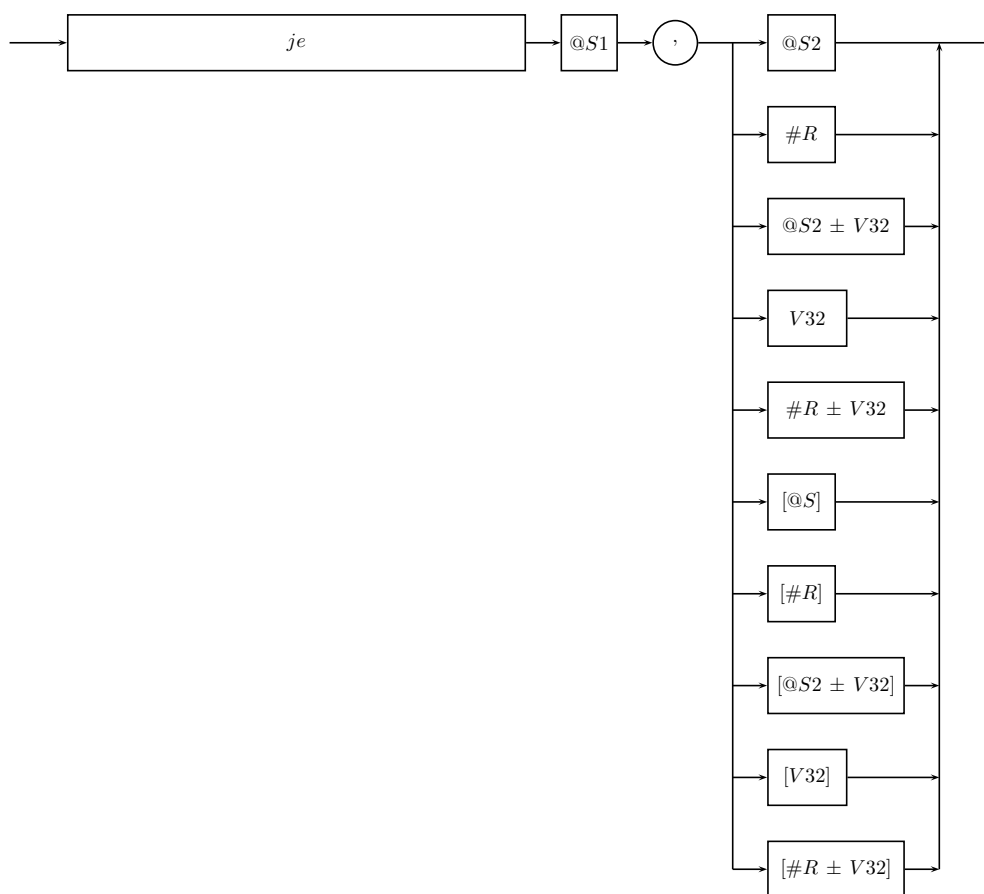


Рис. 1.60: Синтаксическое описание команды *je*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг нуля (*ZF*) аргумента *ARG1* равен единице, т. е. аргумент *ARG1* равен нулю;

1.4.4.57 jg (Jump if Greater)

Установка адреса следующего параграфа, условная

jg ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

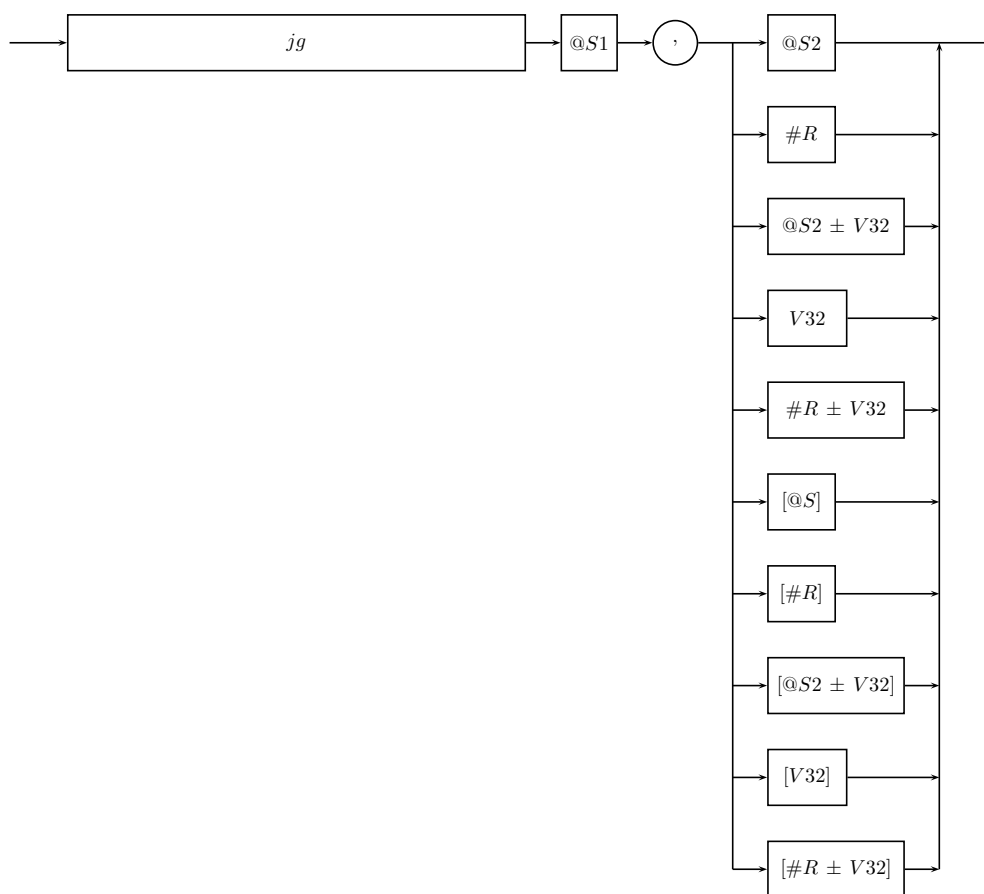


Рис. 1.61: Синтаксическое описание команды *jg*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* равны и флаг нуля (*ZF*) аргумента *ARG1* равен нулю;

1.4.4.58 jge (Jump if Greater and Equal)

Установка адреса следующего параграфа, условная

jge ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

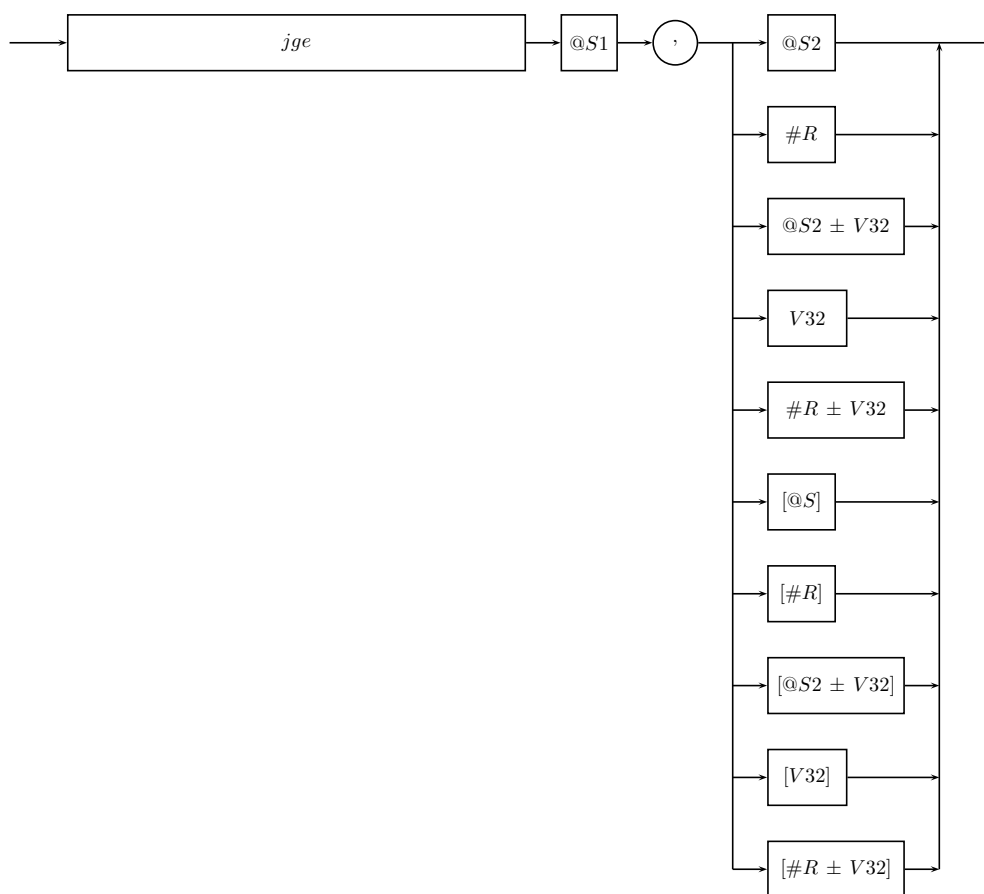


Рис. 1.62: Синтаксическое описание команды *jge*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* равны;

1.4.4.59 jl (Jump if Less)

Установка адреса следующего параграфа, условная

jl ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

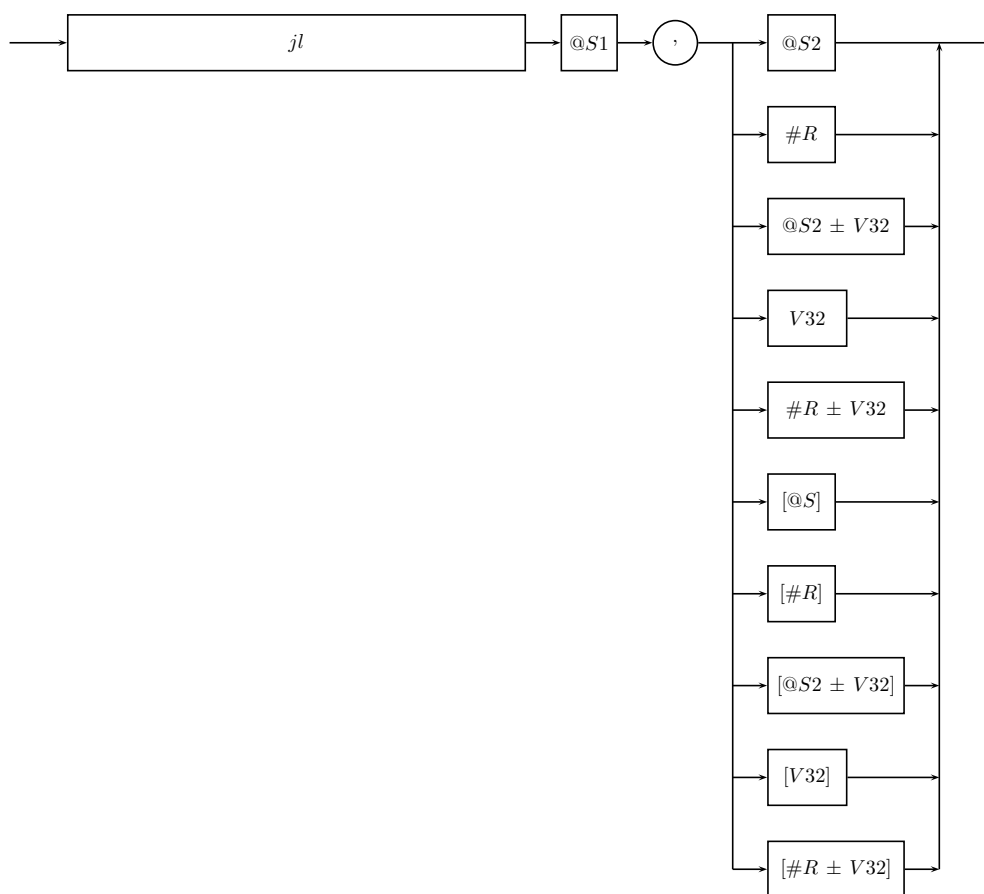


Рис. 1.63: Синтаксическое описание команды *jl*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* не равны;

1.4.4.60 jle (Jump if Less and Equal)

Установка адреса следующего параграфа, условная

jle ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

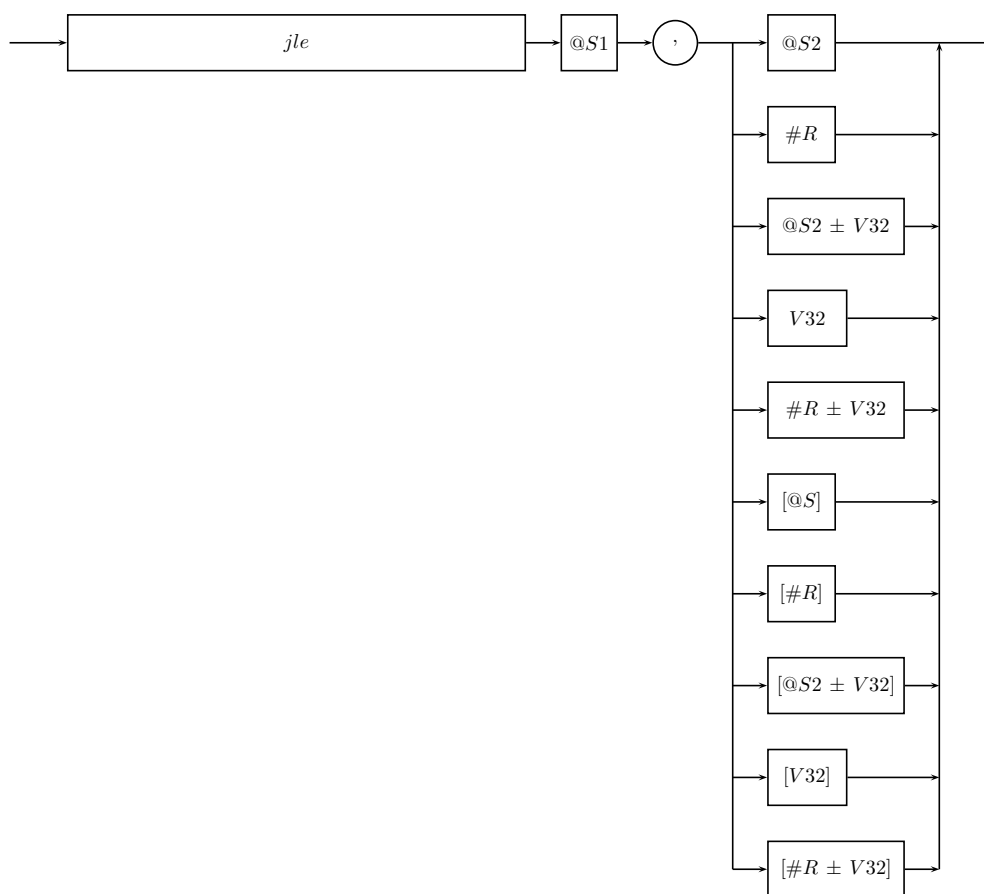


Рис. 1.64: Синтаксическое описание команды *jle*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* не равны или флаг нуля (*ZF*) аргумента *ARG1* равен единице;

1.4.4.61 jmp (unconditional JuMP)

Установка адреса следующего параграфа, безусловная

jmp ARG

Назначение: операция безусловной установки адреса следующего параграфа

Синтаксис:

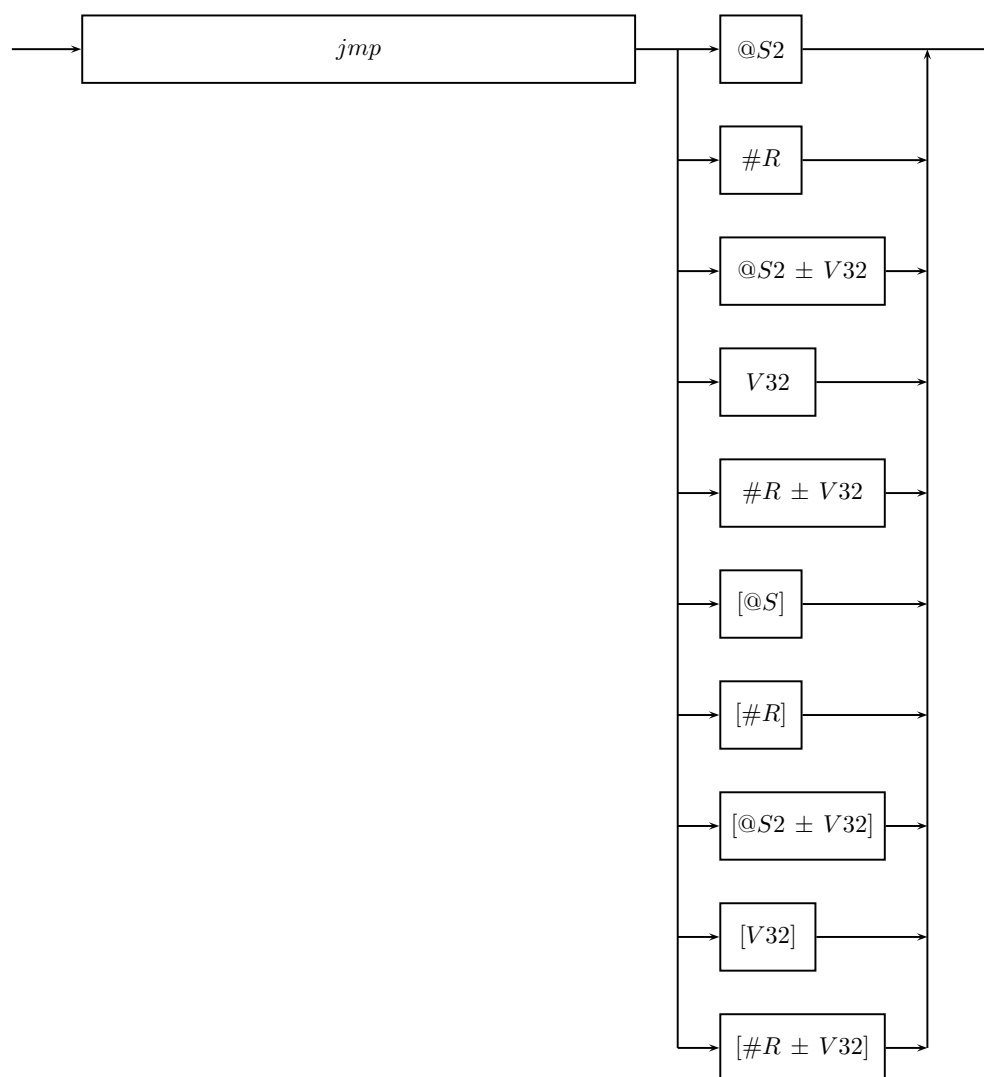


Рис. 1.65: Синтаксическое описание команды *jmp*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG* (фактический переход будет осуществлён по окончании текущего параграфа);

1.4.4.62 jne (Jump if Not Equal)

Установка адреса следующего параграфа, условная

jne ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

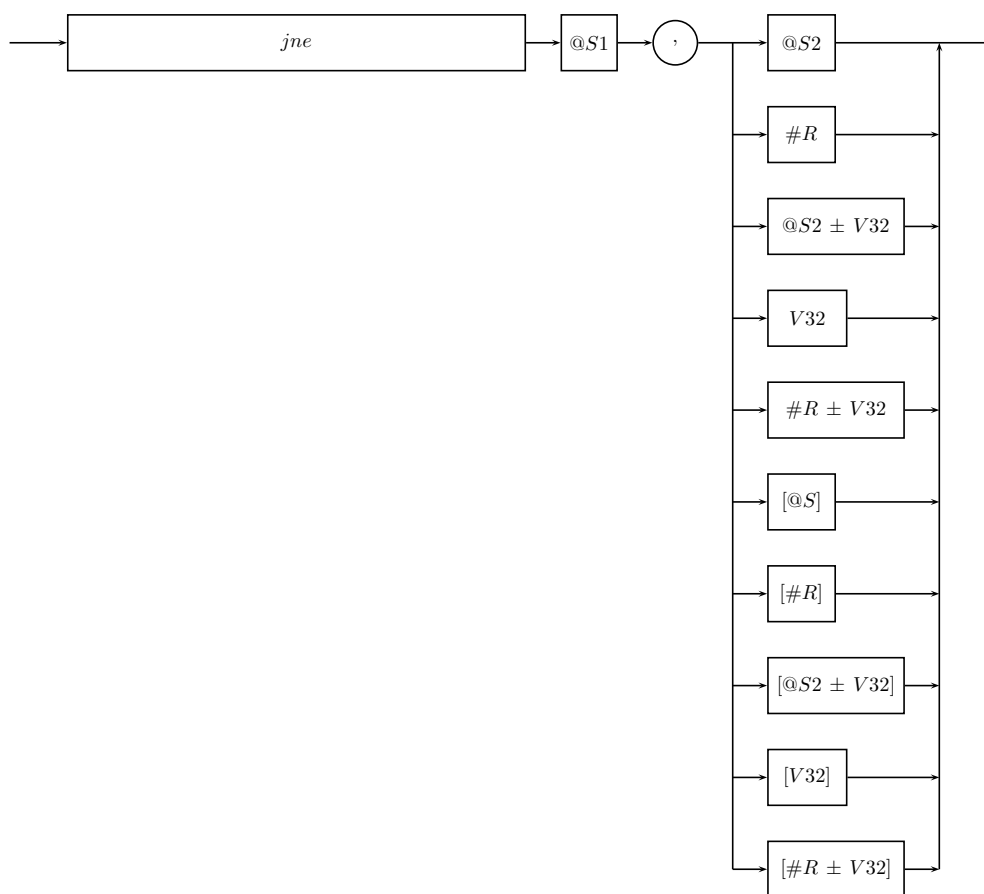


Рис. 1.66: Синтаксическое описание команды *jne*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг нуля (*ZF*) аргумента *ARG1* равен нулю, т. е. аргумент *ARG1* не равен нулю;

1.4.4.63 jno (Jump if Overflow flag unset)

Установка адреса следующего параграфа, условная

jno ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

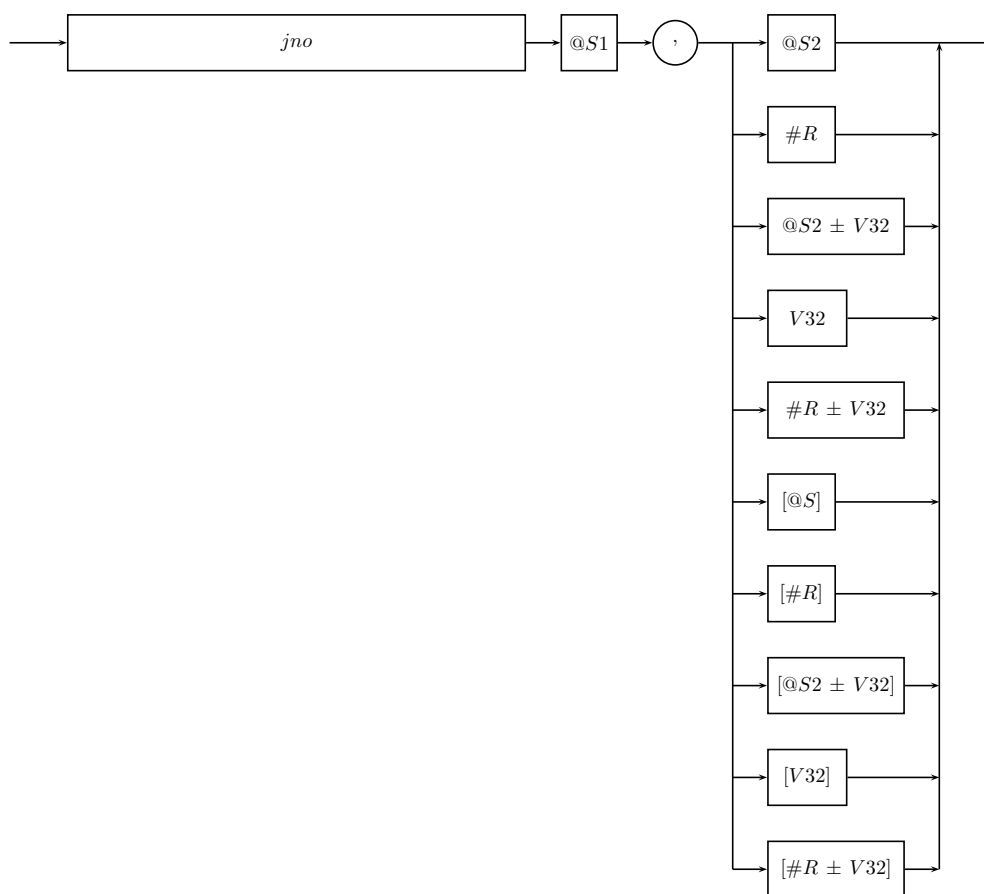


Рис. 1.67: Синтаксическое описание команды *jno*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг переполнения (*OF*) аргумента *ARG1* равен нулю;

1.4.4.64 jns (Jump if Sign flag unset)

Установка адреса следующего параграфа, условная

jns ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

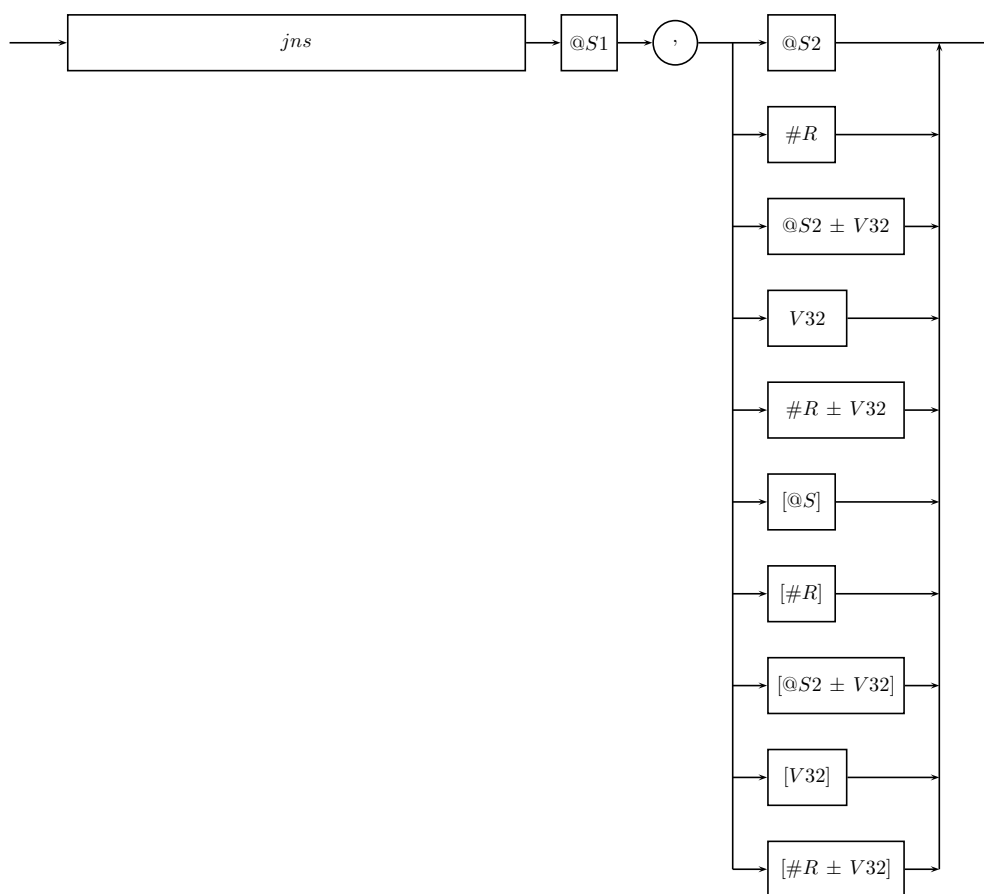


Рис. 1.68: Синтаксическое описание команды *jns*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента ARG2 (фактический переход будет осуществлён по окончании текущего параграфа), если флаг знака (*SF*) аргумента ARG1 равен нулю;

1.4.4.65 jo (Jump if Overflow flag set)

Установка адреса следующего параграфа, условная

jo ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

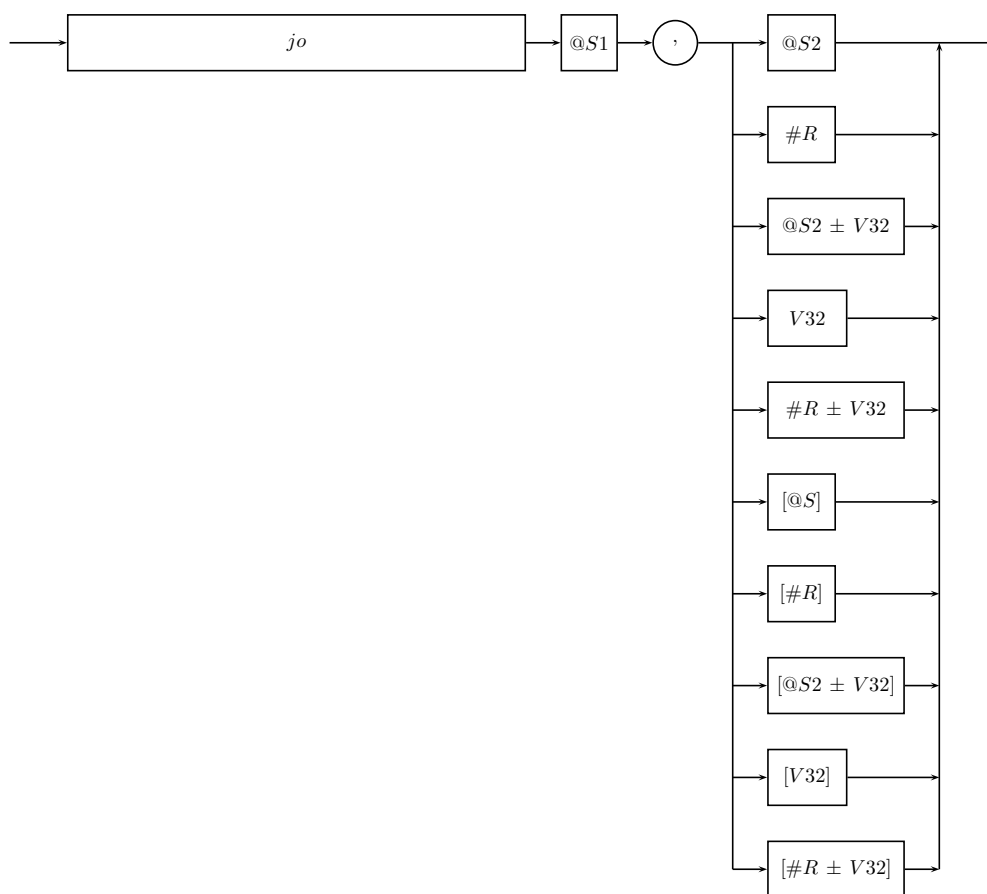


Рис. 1.69: Синтаксическое описание команды *jo*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг переполнения (*OF*) аргумента *ARG1* равен единице;

1.4.4.66 js (Jump if Sign flag set)

Установка адреса следующего параграфа, условная

js ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

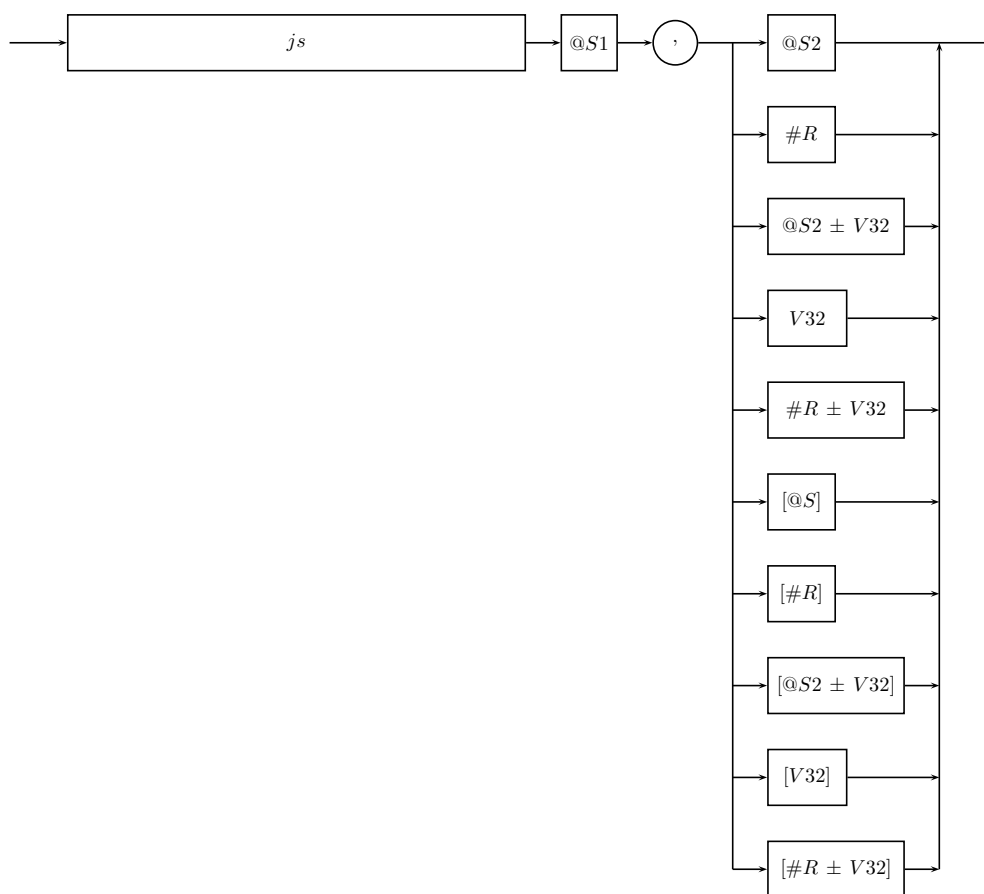


Рис. 1.70: Синтаксическое описание команды *js*

Наличие результата: нет

Алгоритм работы:

- установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если флаг знака (*SF*) аргумента *ARG1* равен единице;

1.4.4.67 lt (Less Than)

Проверка условия «меньше».

lt ARG1, ARG2

Назначение: команда сравнения двух аргументов по критерию меньшего

Синтаксис:

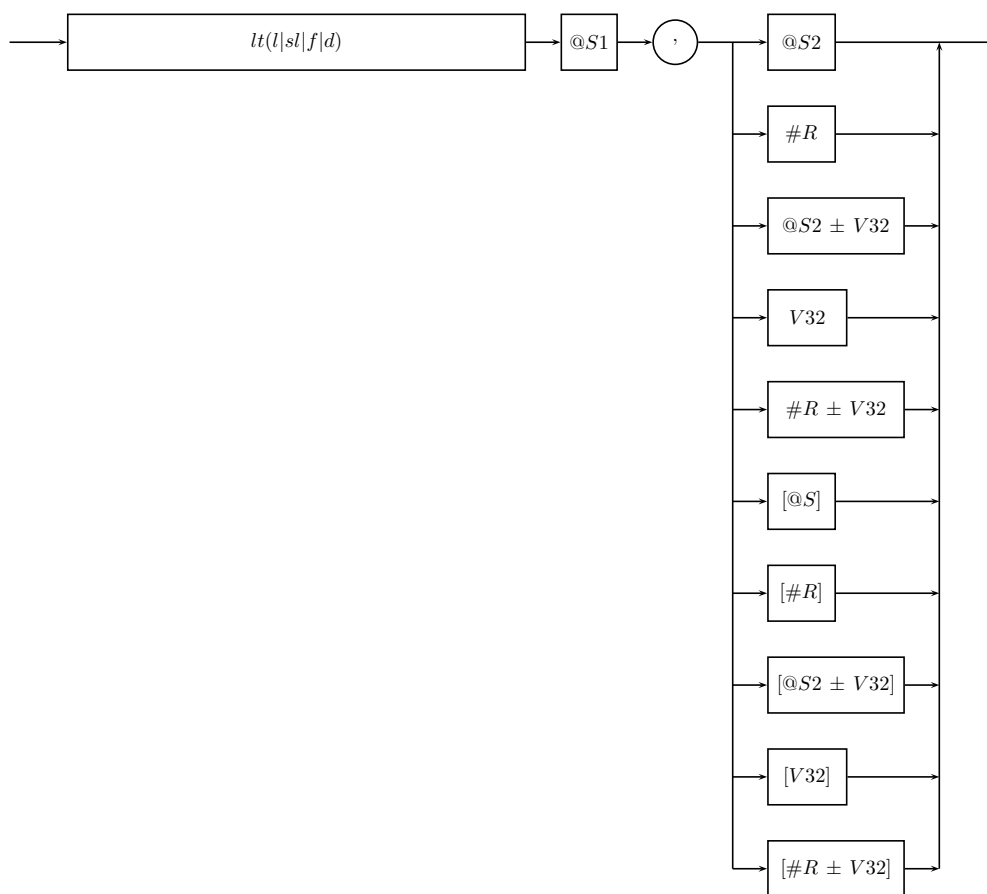


Рис. 1.71: Синтаксическое описание команды *lt*

Наличие результата: да

Алгоритм работы:

- выполнить команду сравнения двух аргументов, в качестве результата выдать «1», если *ARG1* меньше, чем *ARG2*, иначе выдать «0»;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции одноразрядный, старшие 63 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.68 madd (Multiplication with ADDition of packed arguments)

Параллельное умножение со сложением

madd ARG1, ARG2

Назначение: операция параллельного умножения со сложением

Синтаксис:

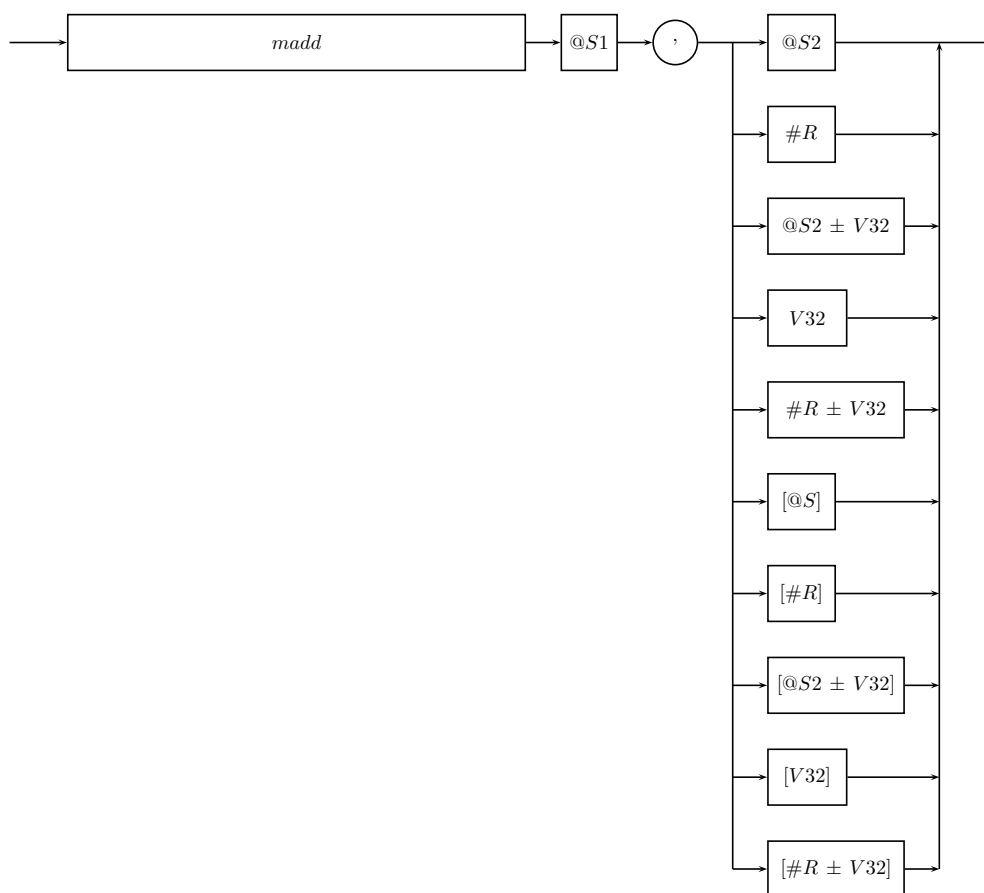


Рис. 1.72: Синтаксическое описание команды *madd*

Наличие результата: да

Алгоритм работы:

- выполнить операцию сложения произведений старших и младших частей упакованных чисел (допустим $ARG1 = (a, b)$, $ARG2 = (c, d)$, где a, c — старшие части упакованных чисел, заданных аргументами $ARG1$ и $ARG2$ соответственно, а b, d — младшие части упакованных чисел, заданных аргументами $ARG1$ и $ARG2$ соответственно, тогда результат равен $a * c + b * d$);

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если все биты экспоненты результата равны единице, либо расширенный бит экспоненты все биты экспоненты результата равен единице, либо хотя бы один из аргументов операций будет равен бесконечности, либо хотя бы один из аргументов операций сопровождается выставленным флагом OF , иначе – $OF = 0$;

флаг потери точности $IF = 1$, если в результате произошла потеря точности вследствие выравнивания мантисс или округления до ближайшего чётного, либо хотя бы один из аргументов операций сопровождается выставленным в '1' соответствующим битом в векторе статуса, либо результатом выполнения операции является NaN, иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.69 max (MAXimum)

Максимум

$max\ ARG1, ARG2$

Назначение: операция выбора наибольшего из двух аргументов

Синтаксис:

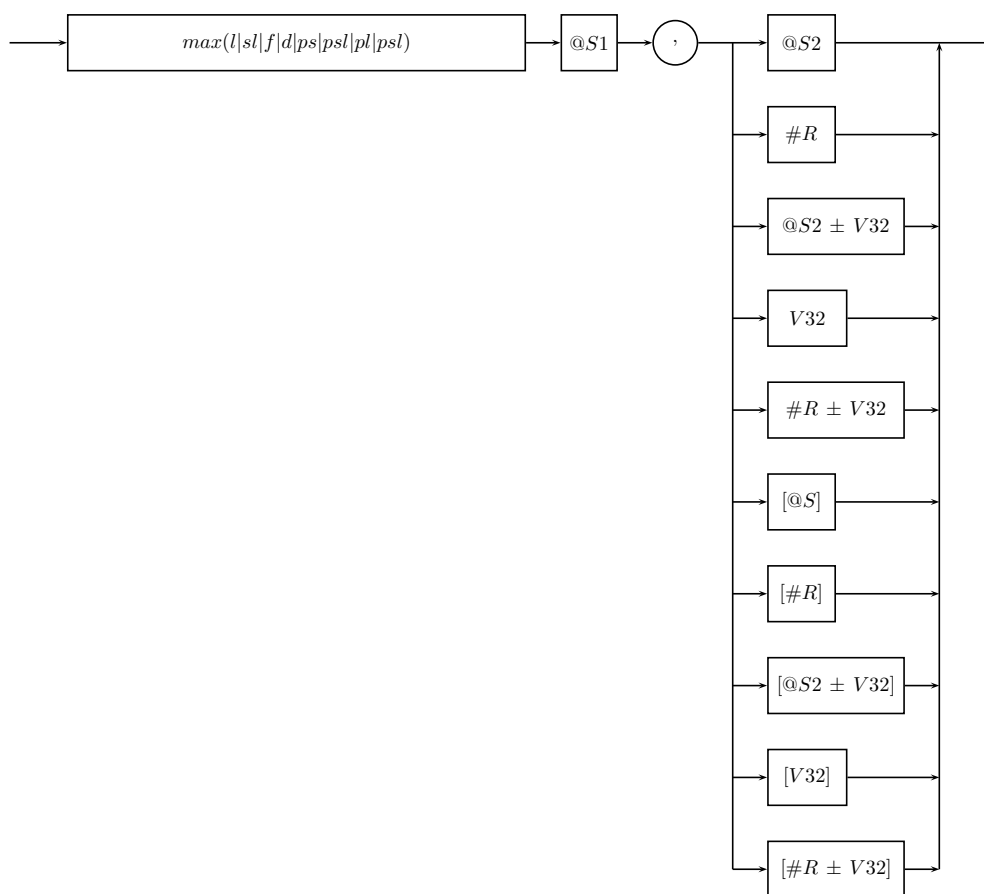


Рис. 1.73: Синтаксическое описание команды *max*

Наличие результата: да

Алгоритм работы:

- выполнить операцию $max(ARG1, ARG2)$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг переноса $CF = 0$;
 - флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.
- для типов операций **float, double**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг потери точности $IF = 0$;
 - флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.
- для типов операций **packed short, packed signed short, packed long, packed signed long, packed, complex**:
 - флаг нуля $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг переноса $CF = 0$;
 - флаг знака $SF = 0$.

1.4.4.70 min (MINimum)

Минимум

$\min ARG1, ARG2$

Назначение: операция выбора наименьшего из двух аргументов

Синтаксис:

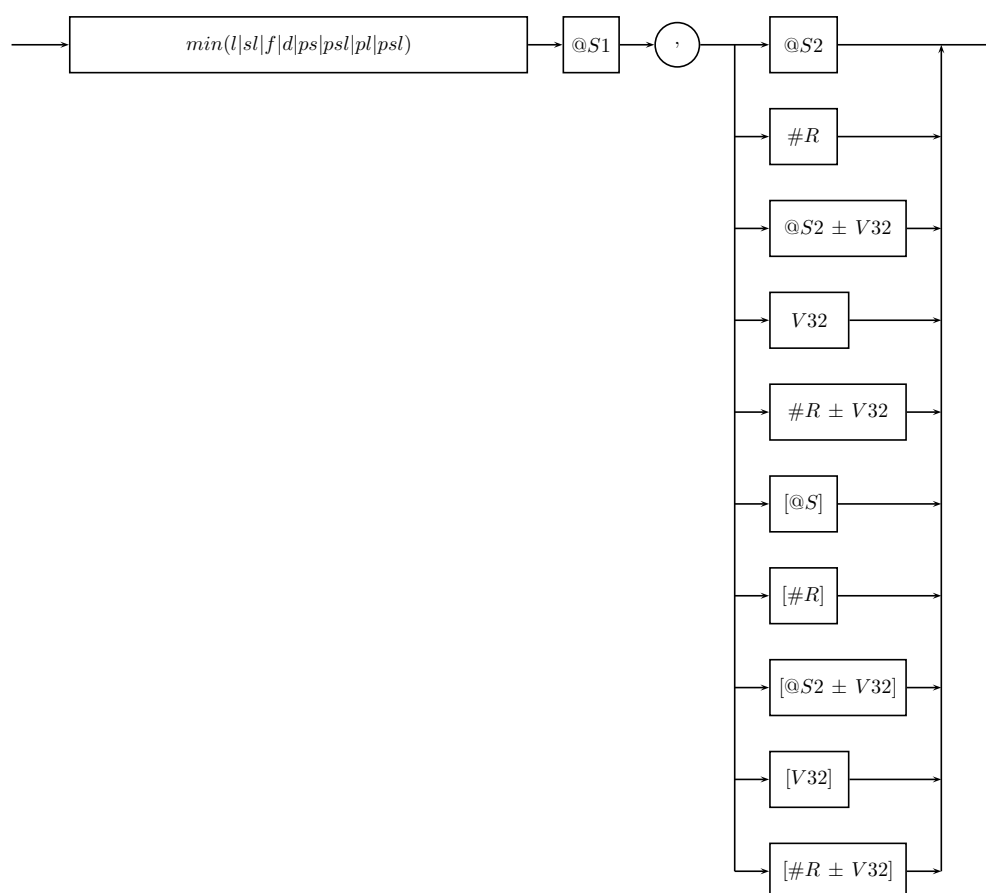


Рис. 1.74: Синтаксическое описание команды *min*

Наличие результата: да

Алгоритм работы:

- выполнить операцию $\min(ARG1, ARG2)$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг переноса $CF = 0$;
 - флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.
- для типов операций **float, double**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг потери точности $IF = 0$;
 - флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.
- для типов операций **packed short, packed signed short, packed long, packed signed long, packed, complex**:
 - флаг нуля $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг переноса $CF = 0$;
 - флаг знака $SF = 0$.

1.4.4.71 move (MOVE)

Передача значения из коммутатора с сохранением флагов в коммутатор

move ARG

Назначение: команда передачи значения из коммутатора с сохранением флагов в коммутатор

Синтаксис:

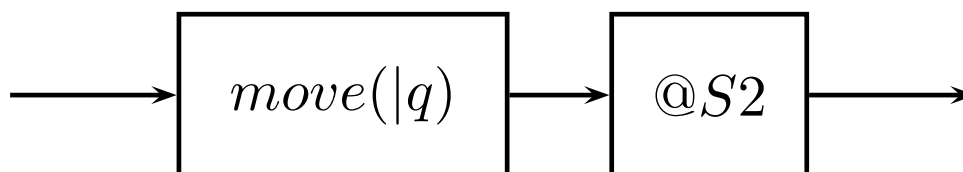


Рис. 1.75: Синтаксическое описание команды *move*

Наличие результата: да

Алгоритм работы:

- передать 64-х разрядный аргумент *ARG* вместе с флагами в коммутатор;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- флаг нуля равен значению флага нуля аргумента *ARG*;
- флаг переполнения равен значению флага переполнения аргумента *ARG*;
- флаг переноса равен значению флага переноса аргумента *ARG*;
- флаг знака равен значению флага знака аргумента *ARG*.

Применение: команда *move* предназначена для проброса значения, ранее сохранённого в коммутаторе, с сохранением флагов аргумента *ARG*.

1.4.4.72 mul (MULtiply)

Умножение

mul ARG1, ARG2

Назначение: операция умножения двух аргументов

Синтаксис:

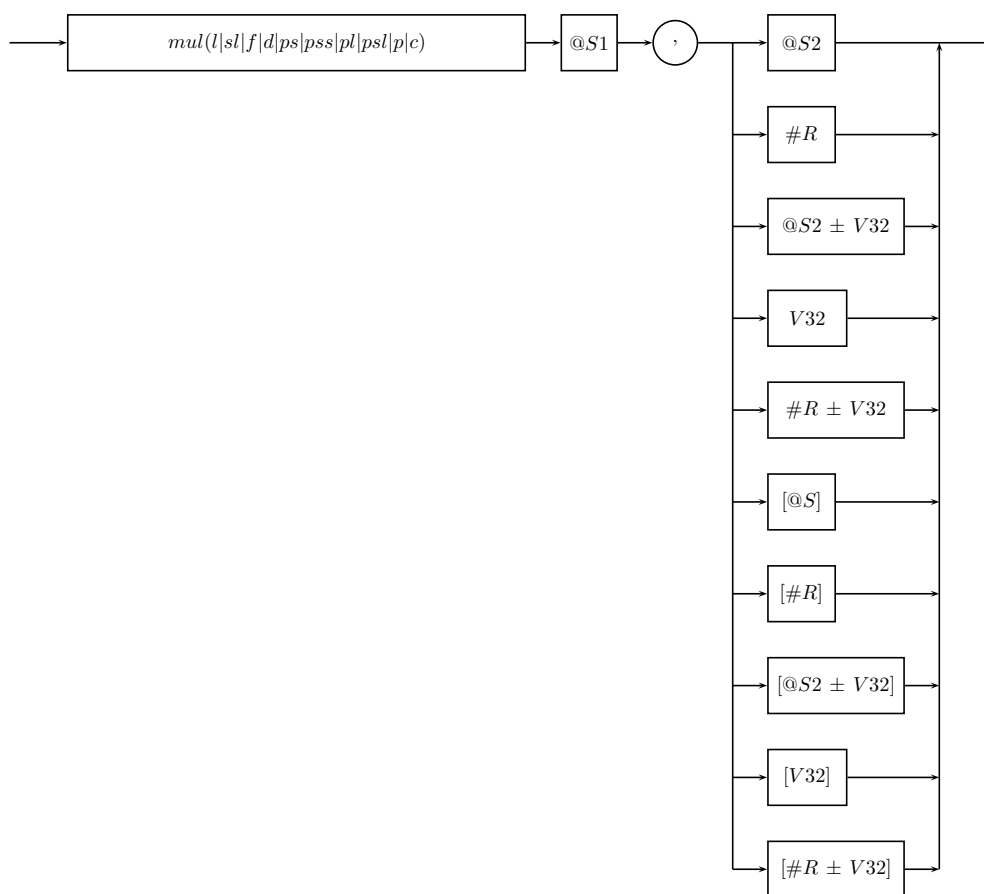


Рис. 1.76: Синтаксическое описание команды *mul*

Наличие результата: да

Алгоритм работы:

- выполнить умножение $ARG1 * ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если результат умножения выходит за границы 32-х разрядной сетки, иначе – $OF = 0$;

флаг переноса $CF = OF$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **float, double**:

флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если все биты экспоненты результата равны единице, иначе – $OF = 0$;

флаг потери точности $IF = 1$, если в результате произошла потеря точности вследствие сдвига мантиссы во время постнормализации или округления до ближайшего чётного, иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long, packed, complex**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

Применение: команда `mul` используется для умножения двух операндов, значение которых интерпретируется согласно типу операции.

Пример:

```
1  .data
2
3  B:
4      .float \
5          0f12.8 , 0f-5.6 , \
6          0f-1.78 , 0f0.19
7
8  .text
9
10 A:
11     rdq B
12     rdq B + 8
13     mulc @1, @2
14     wrq @1, B + 16
15 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) `B`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера `.float` в текущую секцию ассемблирования записывается четыре 32-х разрядных вещественных числа, начиная с текущего адреса ассемблирования (символ обратной косой черты `\` в конце строки используется для продолжения строки, т. е. строки №№4,5,6 логически являются одной строкой);
- в строке №8 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №10 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;

- в строках №№11,12 командами `rdq`, читаются из памяти данных два 64-х разрядных целых беззнаковых числа по адресам B и $B + 8$ соответственно и помещаются в коммутатор;
- в строке №13 командой `mulc` выполняется операция умножения результатов выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №11, @2 — результат выполнения команды чтения в строке №12; оба аргумента команды `mulc` согласно суффиксу с интерпретируются как комплексные числа размерностью 64 бита, поэтому операция умножения выполняется по правилам комплексной арифметики; результат выполнения команды также интерпретируются как 64-х разрядное комплексное число и помещается в коммутатор;
- в строке №14 командой `wrq` осуществляется запись в память данных по адресу $B + 16$ результата выполнения предшествующей команды: @1 — результат выполнения команды сложения в строке №13;
- в строке №15 командой `complete` завершается текущий параграф.

1.4.4.73 norm (NORmalization)

Нормализация

norm ARG

Назначение: операция вычисления необходимого количества сдвигов для нормализации числа с фиксированной точкой

Синтаксис:

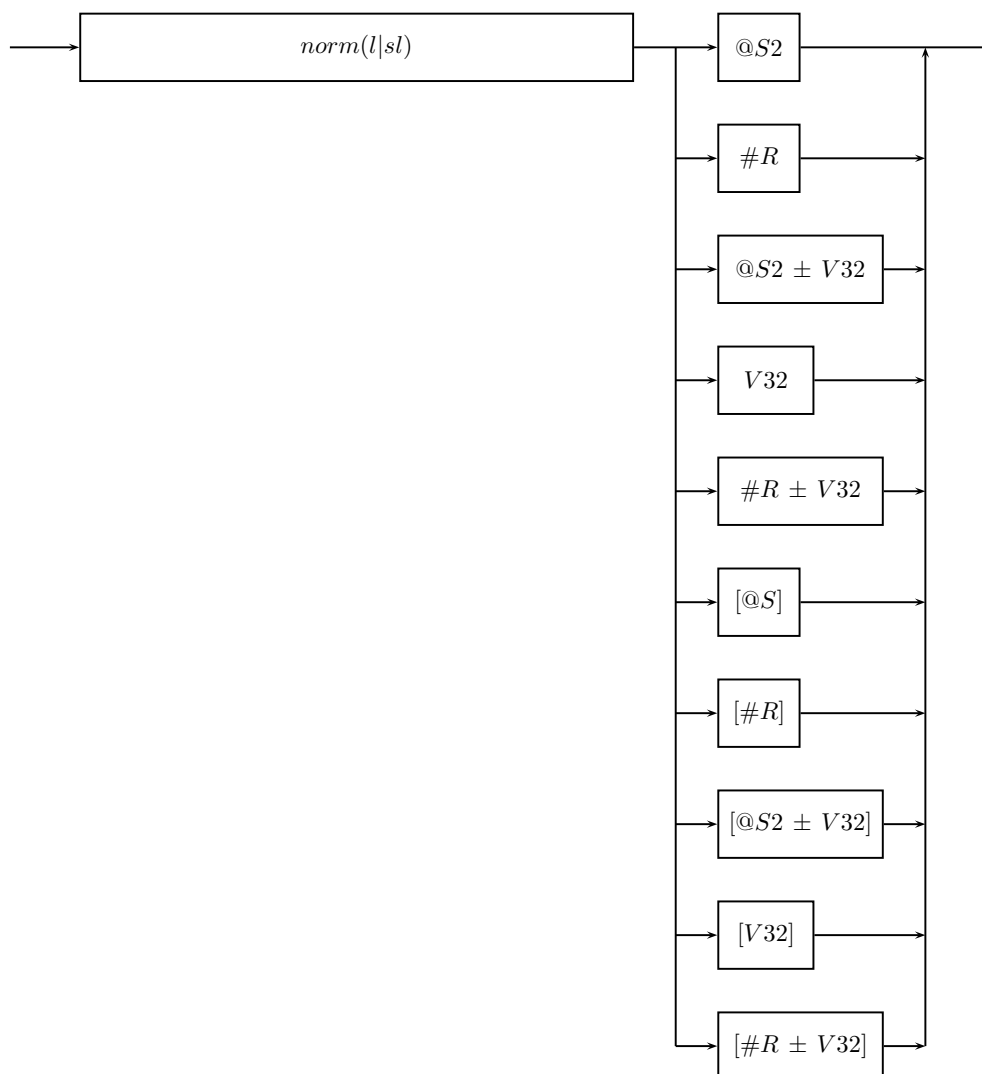


Рис. 1.77: Синтаксическое описание команды *norm*

Наличие результата: да

Алгоритм работы:

- вычислить необходимое количество сдвигов для нормализации числа с фиксированной точкой;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 5-ти разрядный, старшие 59 разрядов ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = OF$;

флаг знака $SF = 0$.

1.4.4.74 not (NOT)

Логического отрицание

not ARG

Назначение: операция логического отрицания аргумента

Синтаксис:

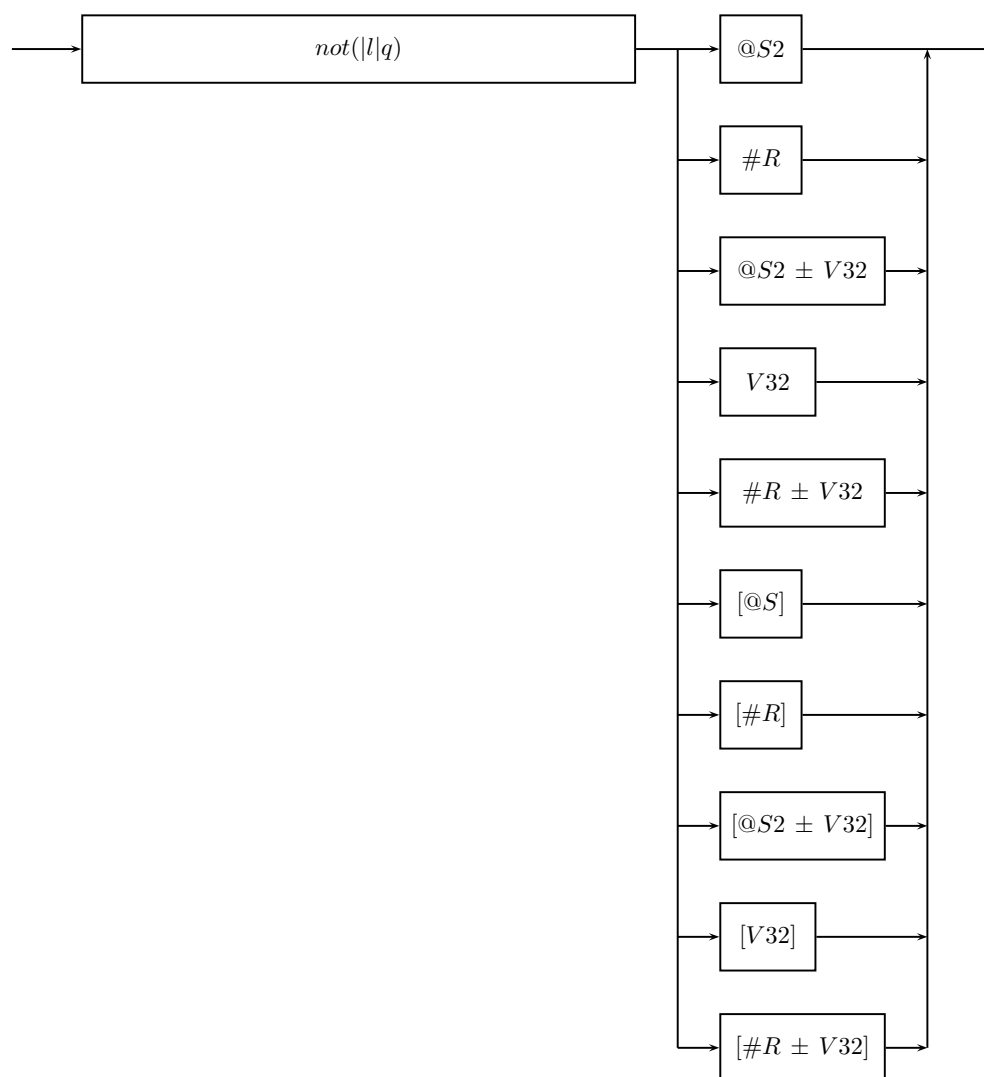


Рис. 1.78: Синтаксическое описание команды *not*

Наличие результата: да

Алгоритм работы:

- выполнить логическое отрицание $\sim ARG1$;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = OF$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.75 or (OR)

Логического сложение

or ARG1, ARG2

Назначение: операция логического сложения двух аргументов

Синтаксис:

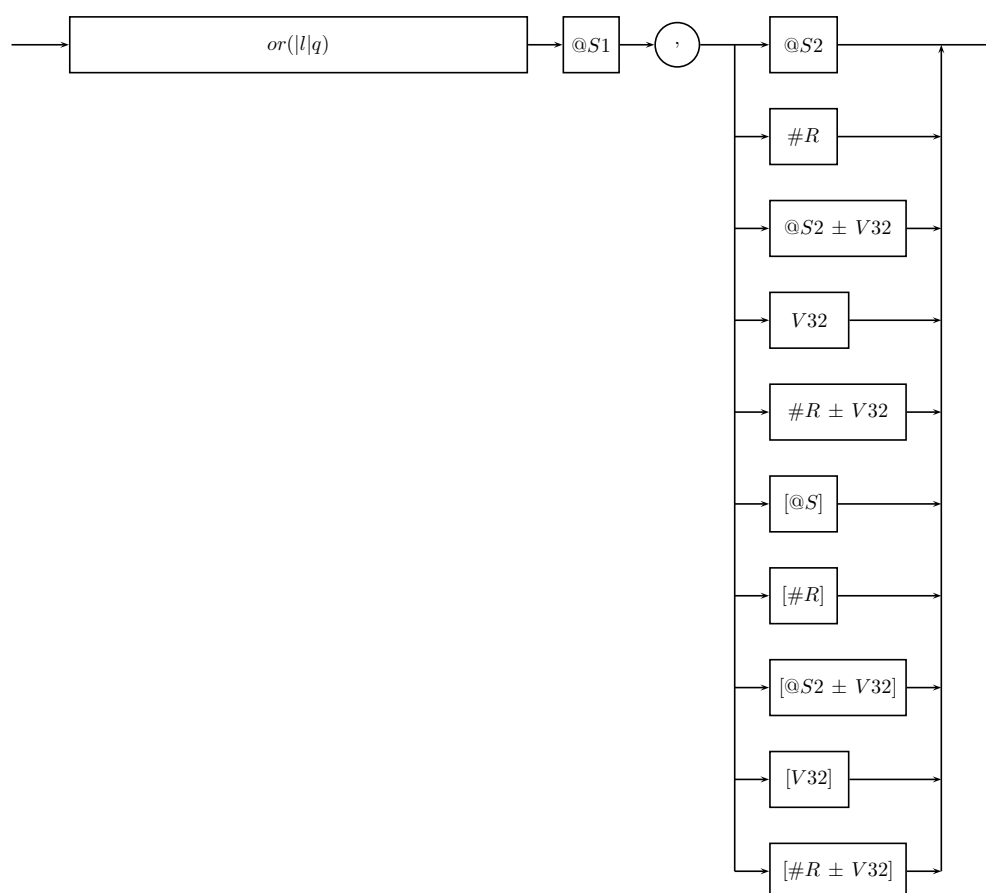


Рис. 1.79: Синтаксическое описание команды or

Наличие результата: да

Алгоритм работы:

- выполнить логическое сложение ARG1 | ARG2;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = OF$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.76 pack (PACK)

Упаковка

pack ARG1, ARG2

Назначение: операция формирования упакованного числа

Синтаксис:

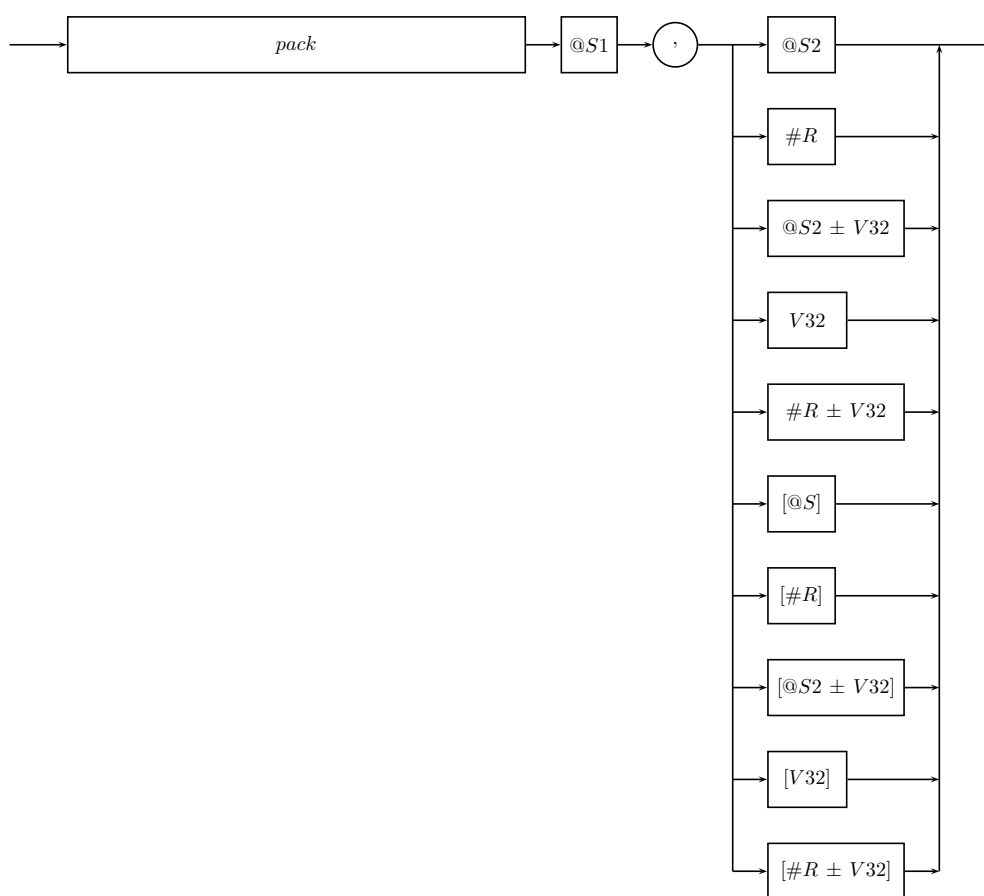


Рис. 1.80: Синтаксическое описание команды *pack*

Наличие результата: да

Алгоритм работы:

- сформировать 64-х разрядное значение результата, старшие 32 разряда (с 32 по 63) которого равны младшим 32 разрядам (с 0 по 31) аргумента *ARG1*, а младшие 32 разряда (с 0 по 31) — старшим 32 разрядам (с 32 по 63) аргумента *ARG2*;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = OF$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.77 patch (PATCH)

Склейка

patch ARG1, ARG2

Назначение: операция формирования склеенного числа

Синтаксис:

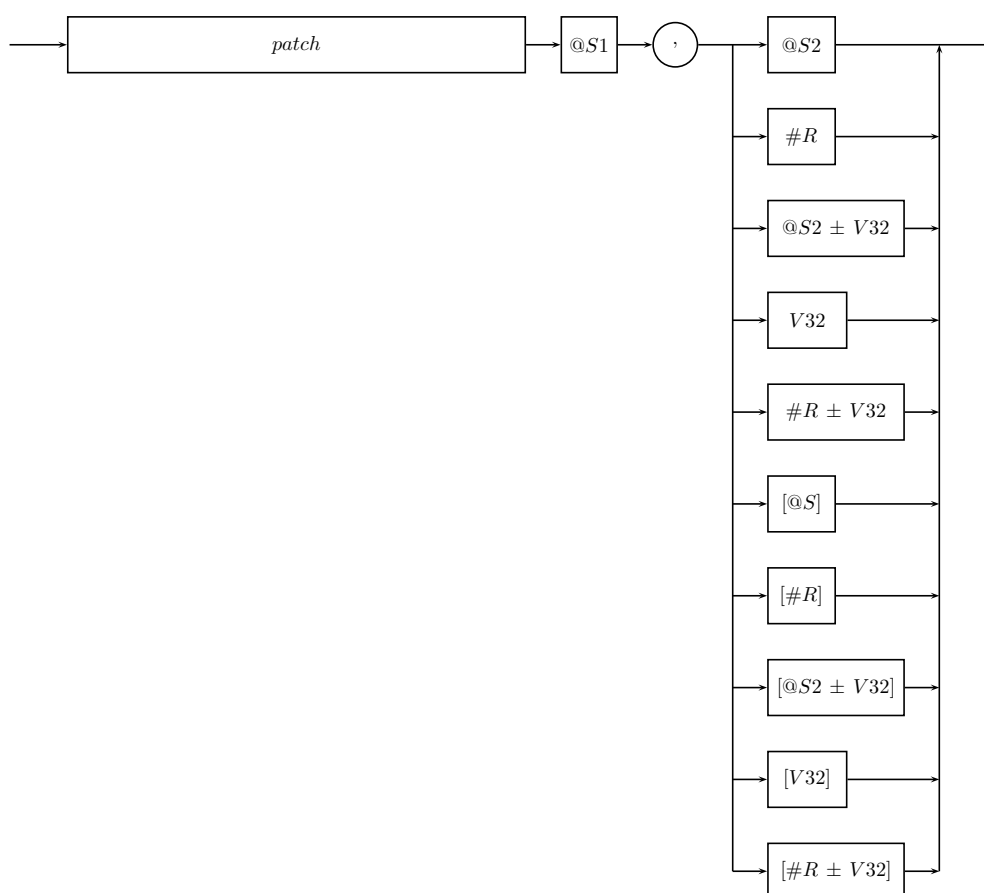


Рис. 1.81: Синтаксическое описание команды *patch*

Наличие результата: да

Алгоритм работы:

- сформировать 64-х разрядное значение результата, старшие 32 разряда (с 32 по 63) которого равны младшим 32 разрядам (с 0 по 31) аргумента *ARG1*, а младшие 32 разряда (с 0 по 31) — младшим 32 разрядам (с 0 по 31) аргумента *ARG2*;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = OF$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.78 rd (ReaD)

Чтение

rd ARG

Назначение: операция чтения значения из памяти

Синтаксис:

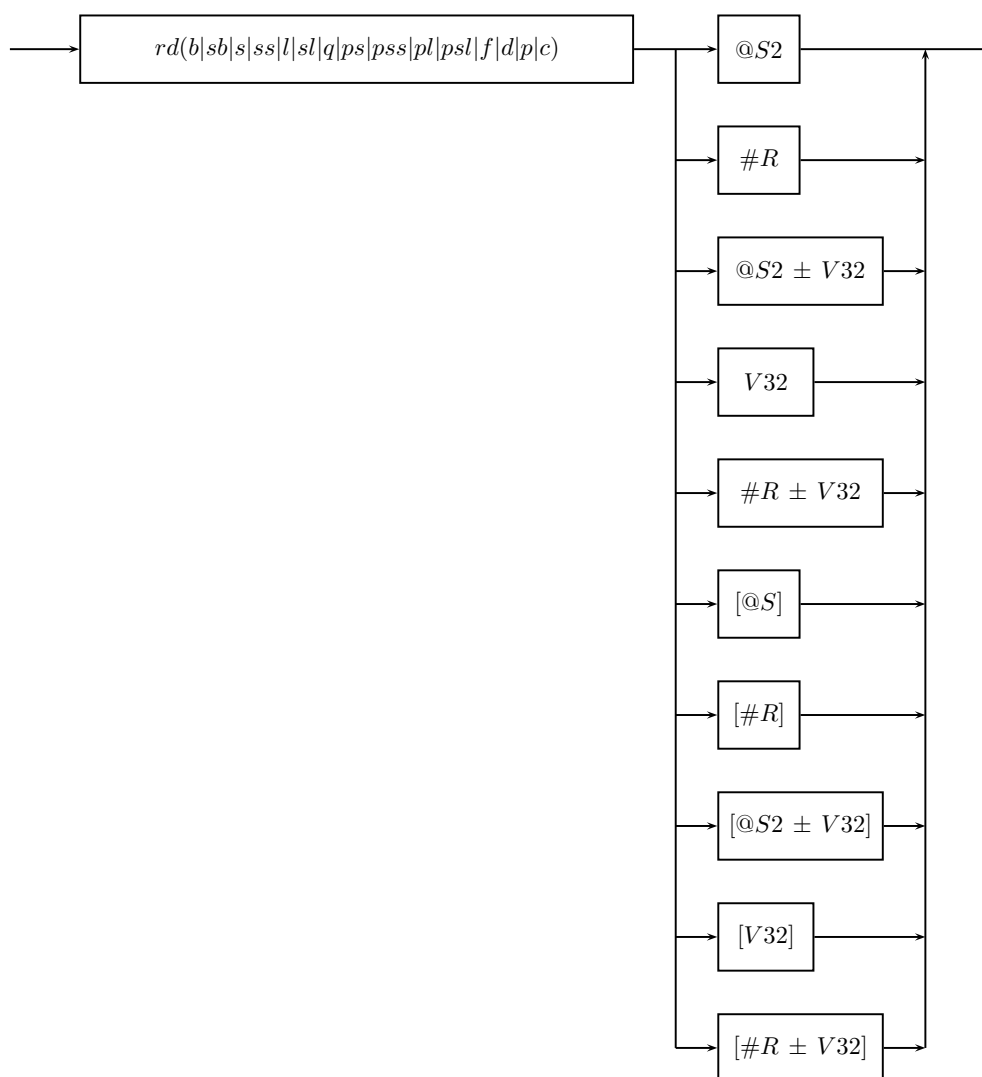


Рис. 1.82: Синтаксическое описание команды *rd*

Наличие результата: да

Алгоритм работы:

- прочитать в зависимости от шестого бита регистра PSW (см. 1.2.2.3.1) и типа команды значение размером байт (знаковый или беззнаковый), полуслово (знаковое или беззнаковое), слово (знаковое или беззнаковое), двойное слово из памяти по адресу, заданному аргументом *ARG*;
- в случае чтения знакового байта (**signed byte**), полуслова (**signed short**) размножить знак до 31 разряда;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Интерпретация значения аргумента: согласно выше описанному алгоритму работы, значение аргумента всегда интерпретируется как адрес памяти, по которому осуществляется чтение значения, помещаемого в качестве результата данной команды в коммутатор. Другими словами, данная команда всегда обращается к памяти вне зависимости от варианта формирования значения аргумента. Если аргумент команды указан в квадратных скобках (`[]` - операция чтения аргумента из памяти), то командой `rd` выполняется два чтения: читается значение из памяти по адресу, указанному в квадратных скобках, а затем снова читается значение из памяти по адресу, равному значению, которое было считано в первый раз. Если для формирования значения аргумента используется значение регистра или результат предшествующей команды, т. е. используется ссылка на коммутатор (`@S`), то старшие 32 разряда (с 32 по 63) игнорируются.

Результат:

- для типа операции **byte** результат операции 32-х разрядный, разряды с 8 по 31 заполняются нулём, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **signed byte** результат операции 32-х разрядный, разряды с 8 по 31 заполняются значением 7 разряда (размножение знака), старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 32-х разрядный, разряды с 16 по 31 заполняются нулём, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **signed short** результат операции 32-х разрядный, разряды с 16 по 31 заполняются значением 15 разряда (размножение знака), старшие 32 разряда ячейки коммутатора, в которую помещается результат,

обнуляются;

- для типов операций **long, signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **byte, signed byte, short, signed short, long, signed long, quad, signed quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **float, double**:

флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long, packed, complex**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

Применение: команда **rd** используется для чтения значения, как знакового, так и беззнакового, из памяти. При чтении знакового значения выполняется размножение знакового разряда до 31 разряда. По возможности следует располагать данную команду в начале параграфа.

Пример:

```
1  .data
2
3  A:
4      .long 1, 2
5
6  .text
7
8  B:
9      rdl A
10     rdl A + 4
11     addl @1, @2
12     wr1 @1, A + 8
13 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера `.long` в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 32-х разрядное число 1, следом за которым по текущему адресу ассемблирования плюс 4 байта записывается 32-х разрядное число 2;
- в строке №6 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №8 объявляется символ (идентификатор) `B`, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строках №№9,10 командами `rdl`, читаются из памяти данных два 32-х разрядных целых беззнаковых числа по адресам `A` и `A + 4` соответственно и помещаются в коммутатор;
- в строке №11 командой `addl` складываются результаты выполнения двух пред-

шествующих команд: @1 — результат выполнения команды чтения в строке №10, @2 — результат выполнения команды чтения в строке №9; оба аргумента команды `addl` согласно суффиксу `l` интерпретируются как 32-х разрядные целые беззнаковые числа; результат выполнения команды также интерпретируются как 32-х разрядное целое беззнаковое число и помещается в коммутатор;

- в строке №12 командой `wrl` осуществляется запись в память данных по адресу $A + 8$ результата выполнения предшествующей команды: @1 — результат выполнения команды сложения в строке №11;
- в строке №13 командой `complete` завершается текущий параграф.

1.4.4.79 rdd (ReaD Direct)

Непосредственное чтение

rdd ARG

Назначение: операция непосредственного чтения значения из памяти

Синтаксис:

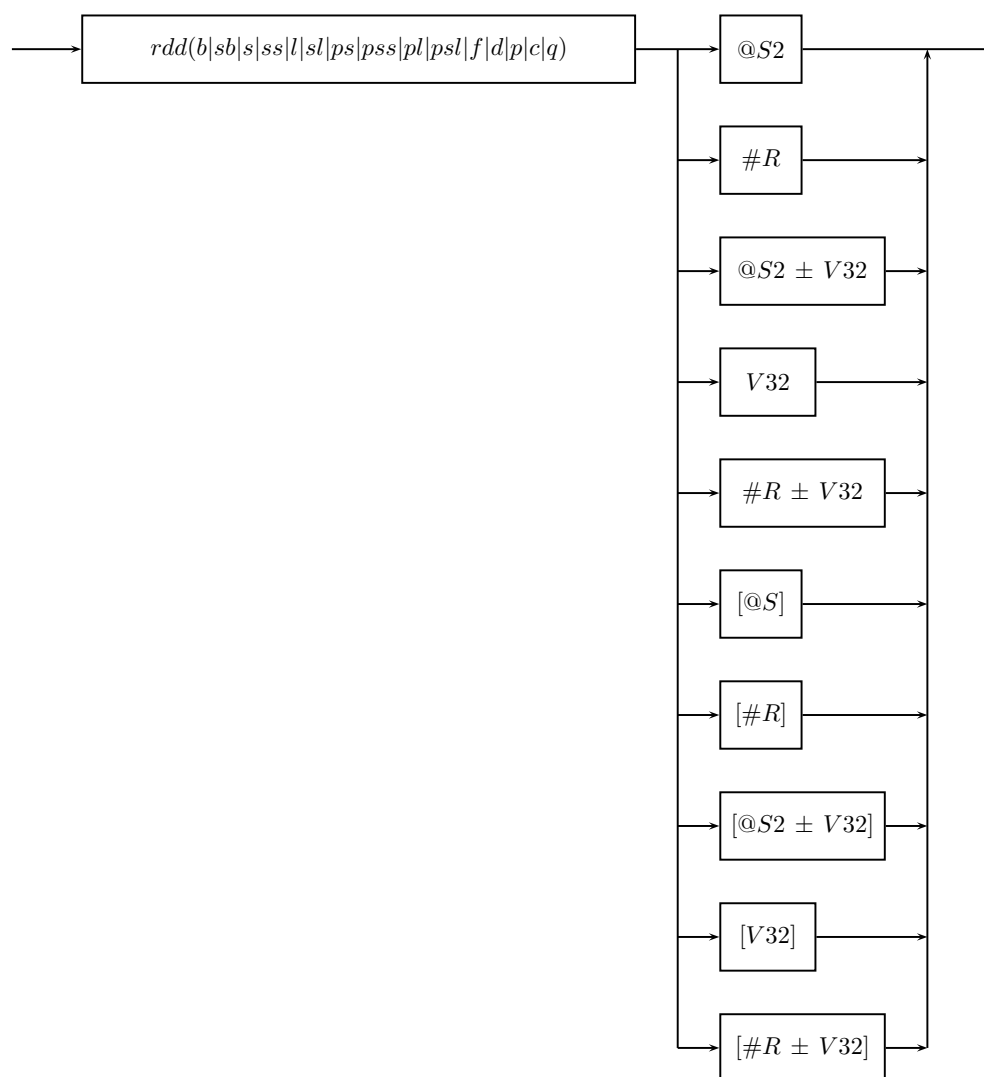


Рис. 1.83: Синтаксическое описание команды *rdd*

Наличие результата: да

Алгоритм работы:

- безотлагательно прочитать в зависимости от типа команды значение размером байт (знаковый или беззнаковый), полуслово (знаковое или беззнаковое), слово (знаковое или беззнаковое), двойное слово из памяти по адресу, заданному аргументом *ARG*;
- в случае чтения знакового байта (**signed byte**), полуслова (**signed short**) размножить знак до 31 разряда;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Интерпретация значения аргумента: согласно выше описанному алгоритму работы, значение аргумента всегда интерпретируется как адрес памяти, по которому осуществляется чтение значения, помещаемого в качестве результата данной команды в коммутатор. Другими словами, данная команда всегда обращается к памяти вне зависимости от варианта формирования значения аргумента. Если аргумент команды указан в квадратных скобках (`[]` - операция чтения аргумента из памяти), то командой `rd` выполняется два чтения: читается значение из памяти по адресу, указанному в квадратных скобках, а затем снова читается значение из памяти по адресу, равному значению, которое было считано в первый раз. Если для формирования значения аргумента используется значение регистра или результат предшествующей команды, т. е. используется ссылка на коммутатор (`@S`), то старшие 32 разряда (с 32 по 63) игнорируются.

Результат:

- для типа операции **byte** результат операции 32-х разрядный, разряды с 8 по 31 заполняются нулём, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **signed byte** результат операции 32-х разрядный, разряды с 8 по 31 заполняются значением 7 разряда (размножение знака), старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 32-х разрядный, разряды с 16 по 31 заполняются нулём, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **signed short** результат операции 32-х разрядный, разряды с 16 по 31 заполняются значением 15 разряда (размножение знака), старшие 32 разряда ячейки коммутатора, в которую помещается результат,

обнуляются;

- для типов операций **long, signed long, float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **byte, signed byte, short, signed short, long, signed long, quad, signed quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **float, double**:

флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long, packed, complex**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

Применение: команда `rdd` используется для чтения значения, как знакового, так и беззнакового, из памяти. При чтении знакового значения выполняется разложение знакового разряда до 31 разряда. По возможности следует располагать данную команду в начале параграфа.

1.4.4.80 rol (ROtate Left)

Сдвиг циклический аргумента влево

rol ARG1, ARG2

Назначение: операция циклического сдвига аргумента влево

Синтаксис:

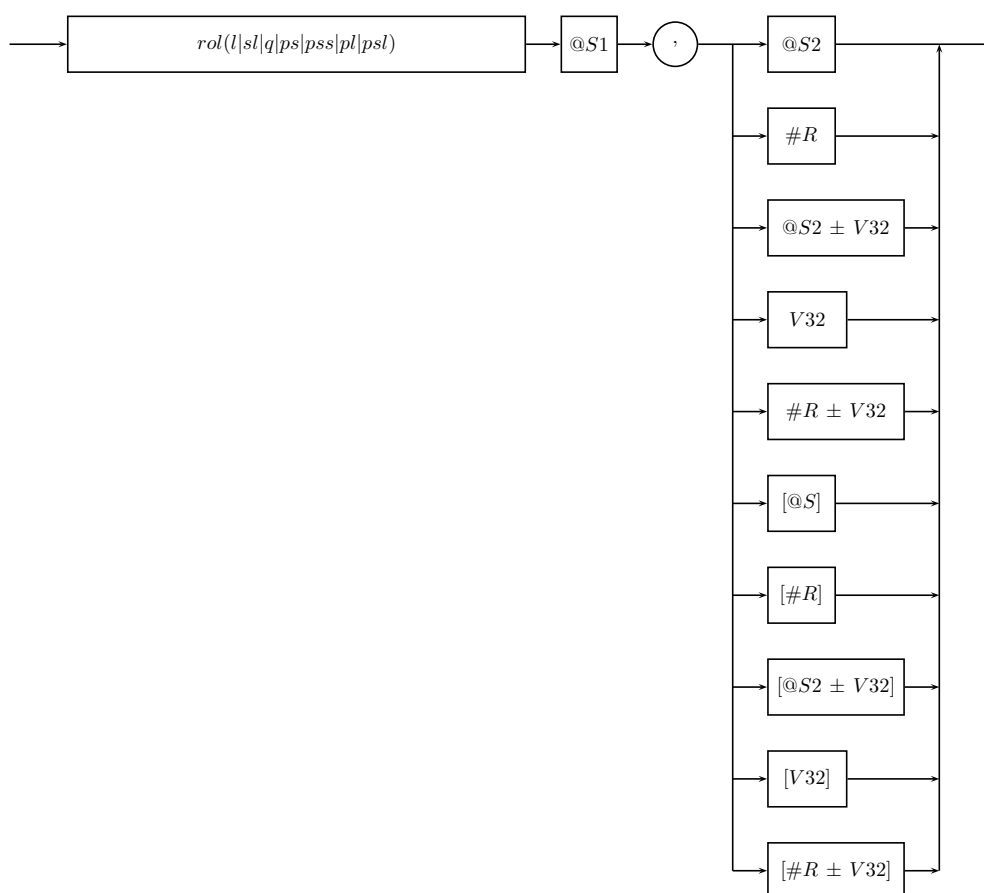


Рис. 1.84: Синтаксическое описание команды *rol*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента *ARG1* влево на количество бит, заданных в аргументе *ARG2*;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long, quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если старший бит аргумента $ARG1$ не равен старшему биту результата, иначе $OF = 0$;

флаг переноса $CF = 1$, если последний сдвигаемый бит равен 1, иначе – $CF = 0$;

флаг знака $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.81 ror (ROtate Right)

Сдвиг циклический аргумента вправо

ror ARG1, ARG2

Назначение: операция циклического сдвига аргумента вправо

Синтаксис:

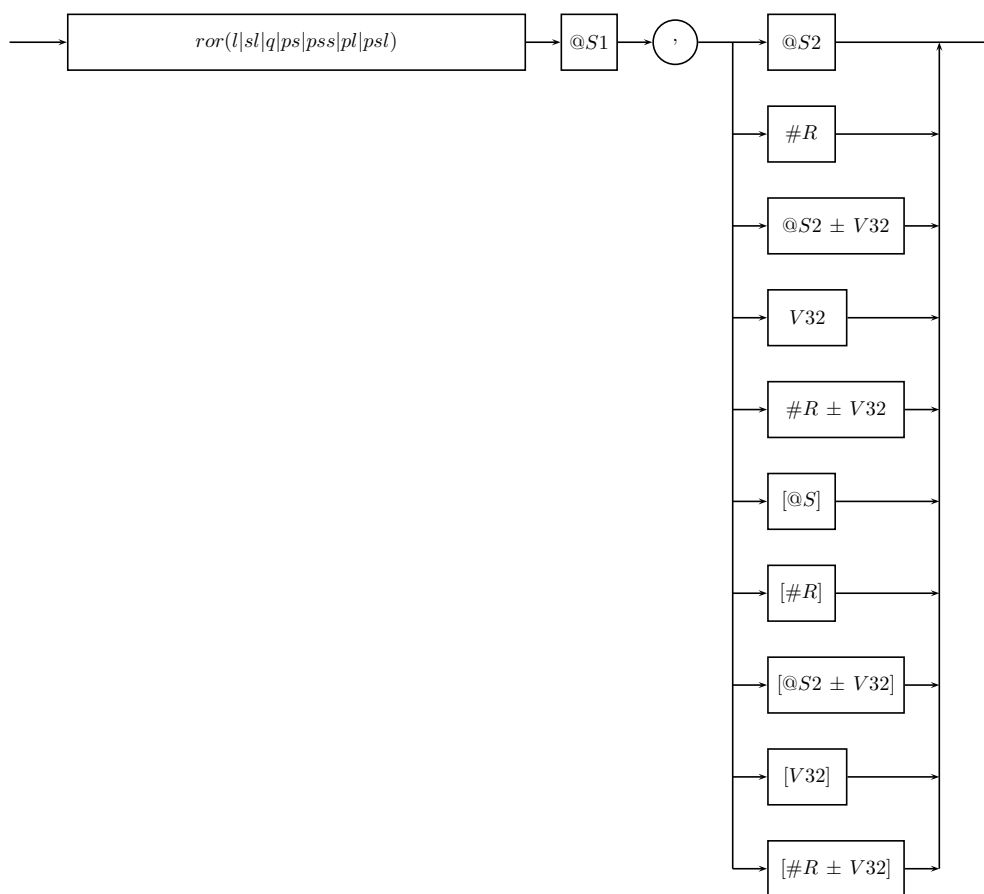


Рис. 1.85: Синтаксическое описание команды *ror*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента *ARG1* вправо на количество бит, заданных в аргументе *ARG2*;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long, quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если старший бит аргумента $ARG1$ не равен старшему биту результата, иначе $OF = 0$;

флаг переноса $CF = 1$, если последний сдвигаемый бит равен 1, иначе – $CF = 0$;

флаг знака $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.82 sa (Select if Above)

Выбор второго аргумента по условию

sa ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

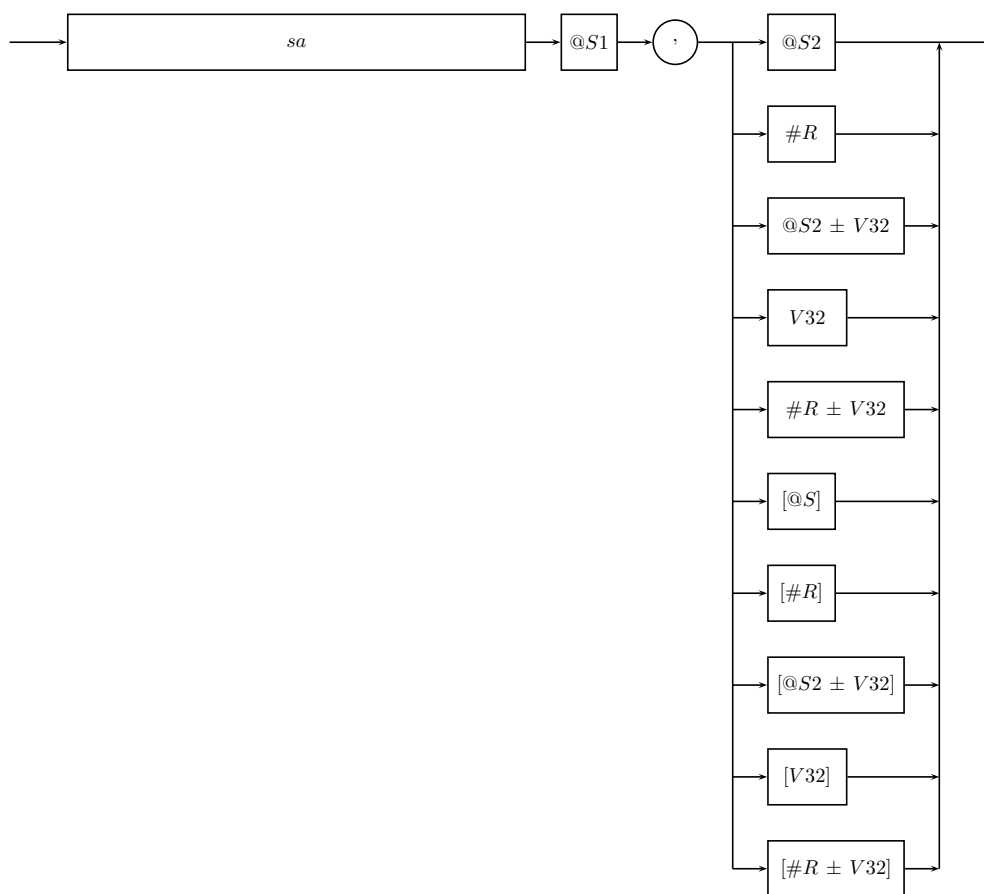


Рис. 1.86: Синтаксическое описание команды *sa*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаги переноса (*CF*) и нуля (*ZF*) аргумента *ARG1* равны нулю, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.83 sae (Select if Above and Equal) / snc (Select if Carry flag unset)

Выбор второго аргумента по условию

$(sae/snc) ARG1, ARG2$

Назначение: команда условного выбора второго аргумента

Синтаксис:

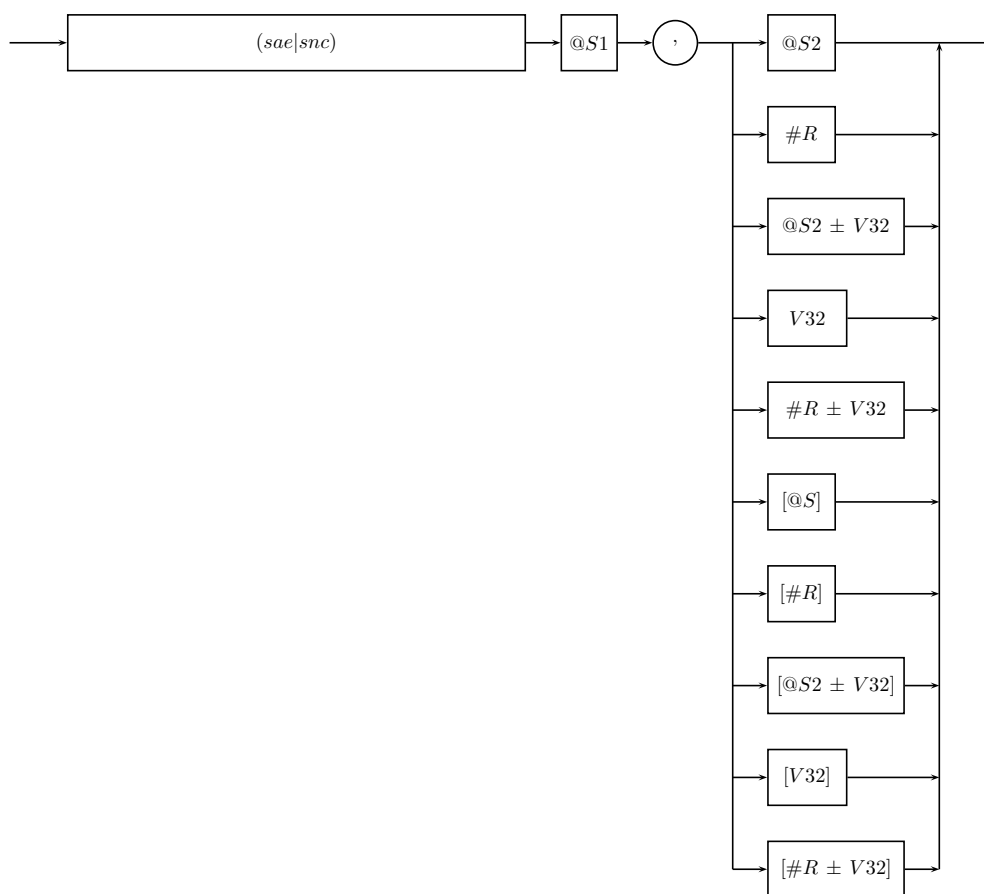


Рис. 1.87: Синтаксическое описание команды *sae/snc*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаг переноса (*CF*) аргумента *ARG1* равен нулю, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.84 sar (Shift Arithmetic Right)

Сдвиг арифметический аргумента вправо

sar ARG1, ARG2

Назначение: операция арифметического сдвига аргумента вправо

Синтаксис:

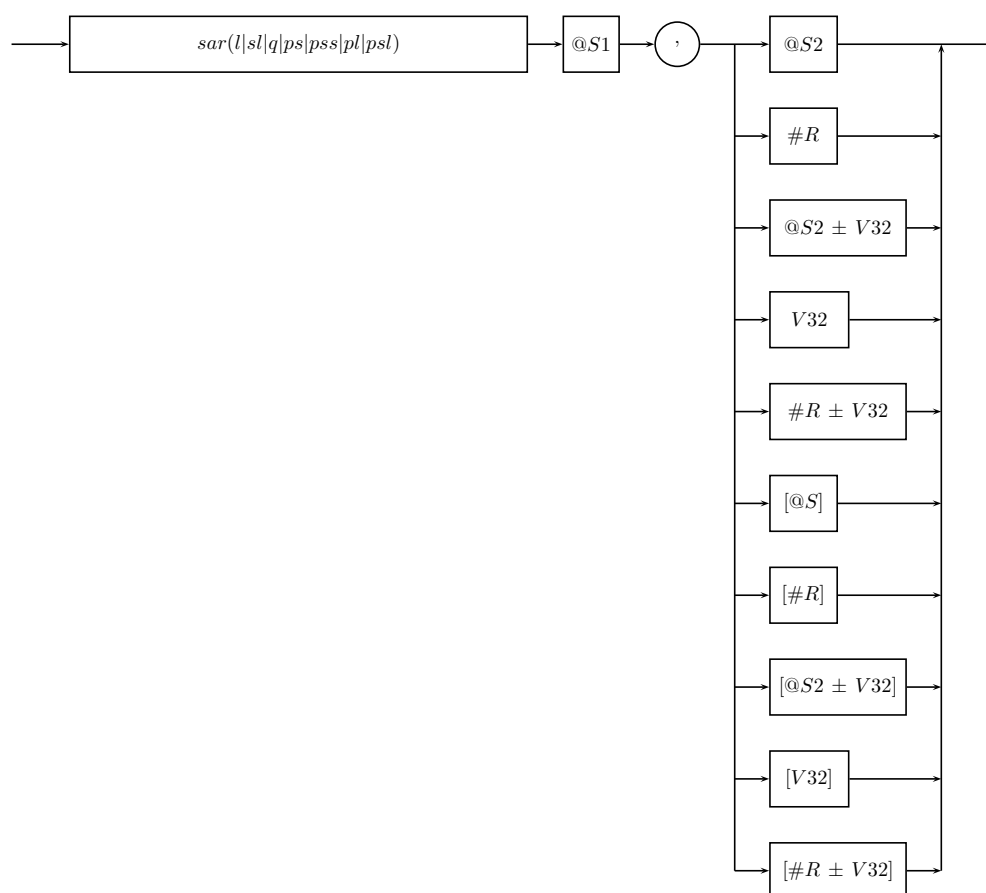


Рис. 1.88: Синтаксическое описание команды *sar*

Наличие результата: да

Алгоритм работы:

- выполнить арифметический сдвиг аргумента *ARG1* вправо на количество бит, заданных в аргументе *ARG2*;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long, quad**:
 - флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;
 - флаг переполнения $OF = 1$, если старший бит аргумента $ARG1$ не равен старшему биту результата, иначе $OF = 0$;
 - флаг переноса $CF = 1$, если последний сдвигаемый бит равен 1, иначе – $CF = 0$;
 - флаг знака $SF = 0$.
- для типов операций **packed short, packed signed short, packed long, packed signed long**:
 - флаг нуля $ZF = 0$;
 - флаг переполнения $OF = 0$;
 - флаг переноса $CF = 0$;
 - флаг знака $SF = 0$.

1.4.4.85 sb (Select if Below) / sc (Select if Carry flag set)

Выбор второго аргумента по условию

(sb/sc) ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

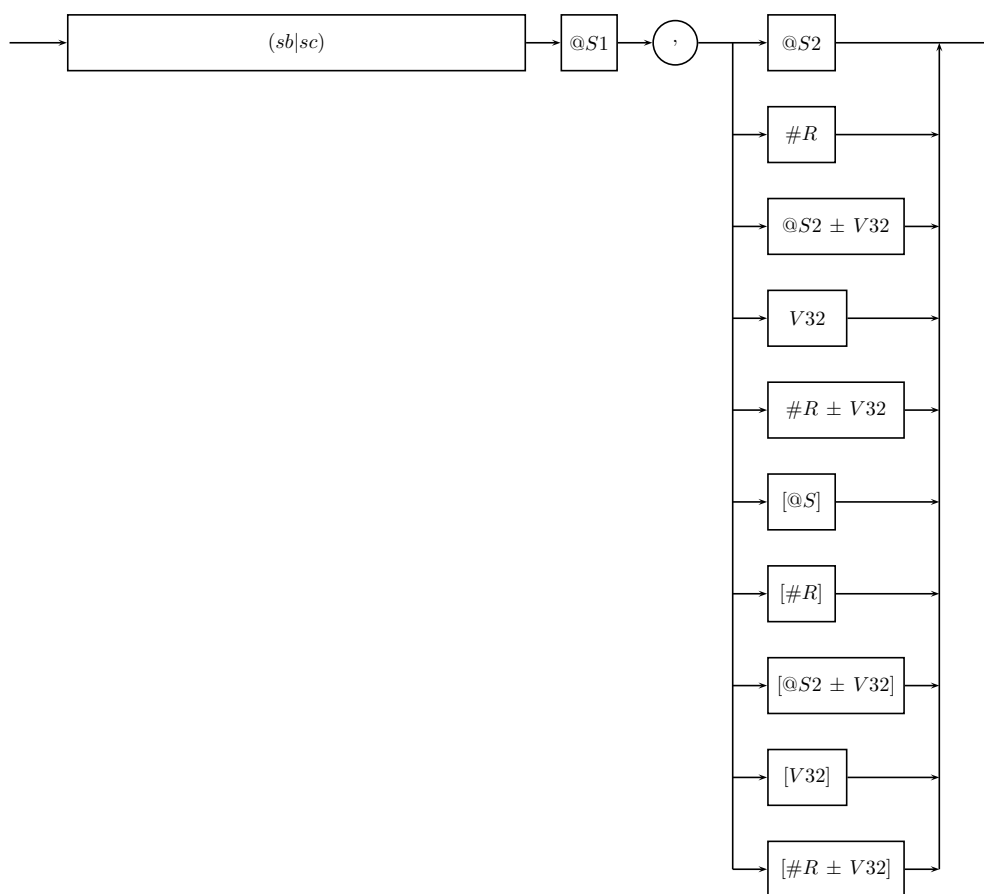


Рис. 1.89: Синтаксическое описание команды sb/sc

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента ARG2, если флаг переноса (CF) аргумента ARG1 равен единице, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.86 sbb (SuBtract with Barrow (Carry Flag))

Вычитание с заёмом

sbb ARG1, ARG2

Назначение: операция целочисленного вычитания с учётом флага переноса (*CF*) результата ранее выполненной команды

Синтаксис:

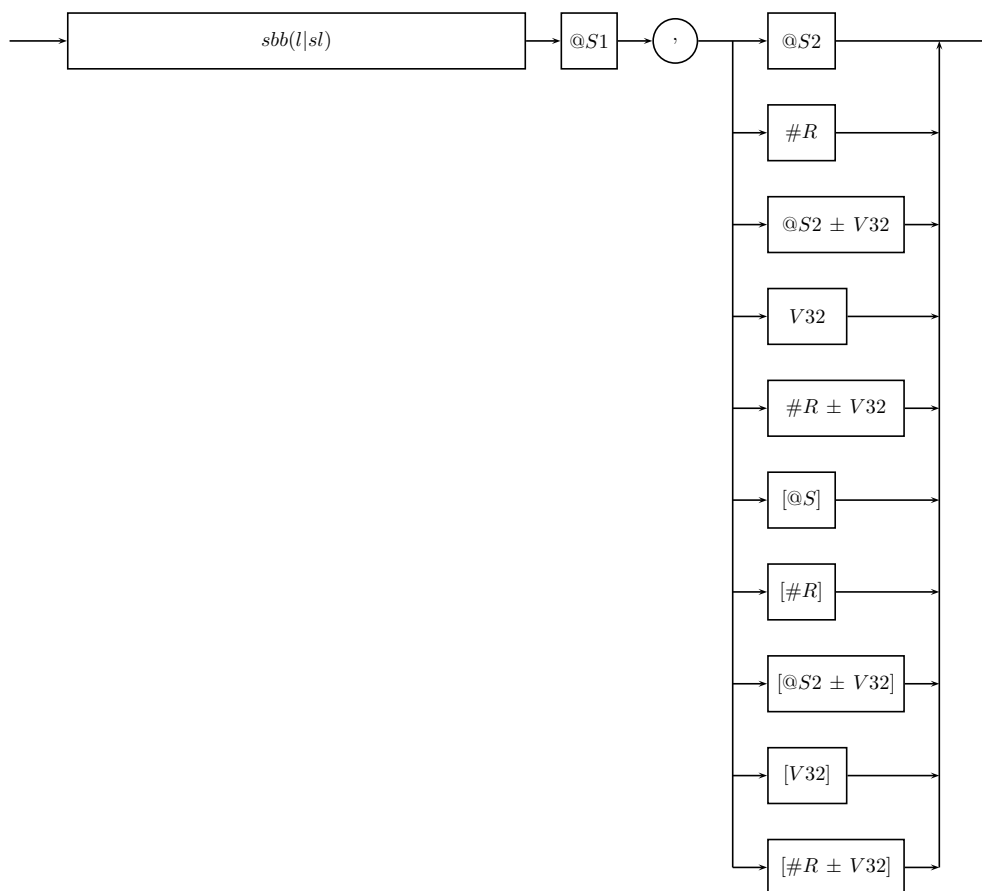


Рис. 1.90: Синтаксическое описание команды *sbb*

Наличие результата: да

Алгоритм работы:

- выполнить вычитание из значения аргумента *ARG2* значение флага переноса (*CF*) аргумента *ARG1*;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются.

Состояние флагов результата после выполнения команды:

- для типа операции **long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = CF$;

флаг переноса $CF = 1$, если значение аргумента $ARG2$ равно 1, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типа операции **signed long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.87 sbe (Select if Below and Equal)

Выбор второго аргумента по условию

sbe ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

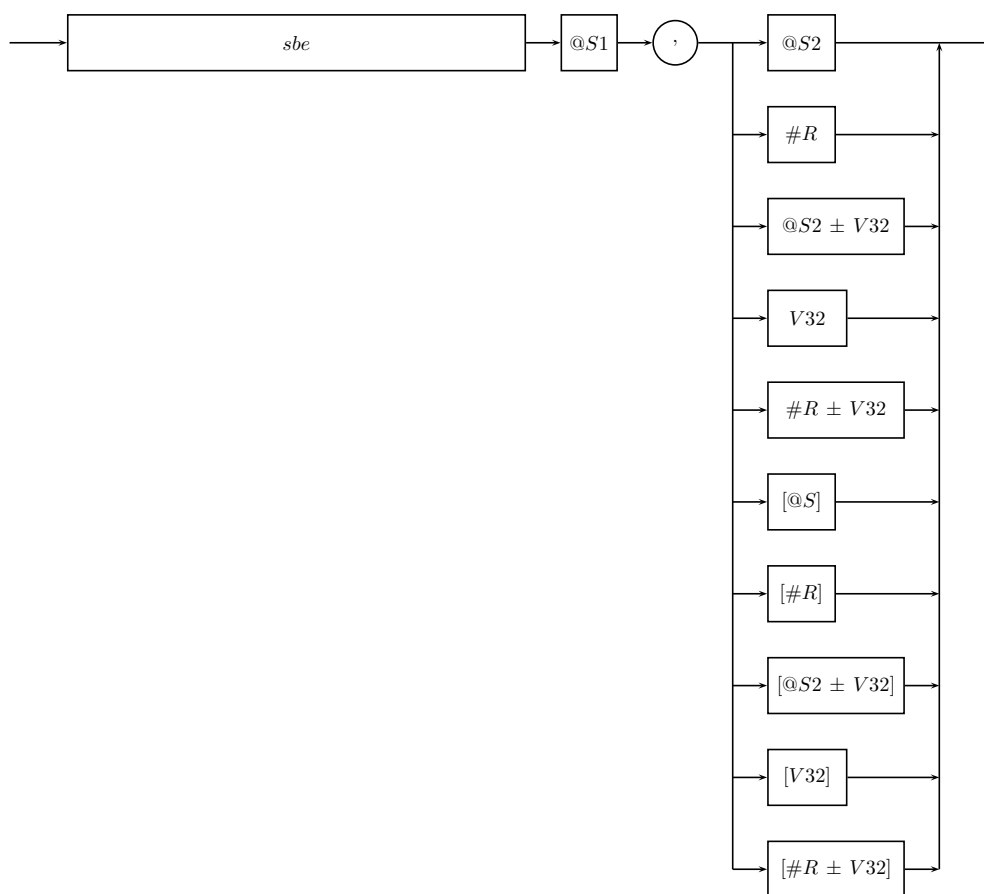


Рис. 1.91: Синтаксическое описание команды *sbe*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаги переноса (*CF*) и нуля (*ZF*) аргумента *ARG1* равны единице, иначе – равным нулю;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.88 se (Select if Equal)

Выбор второго аргумента по условию

se ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

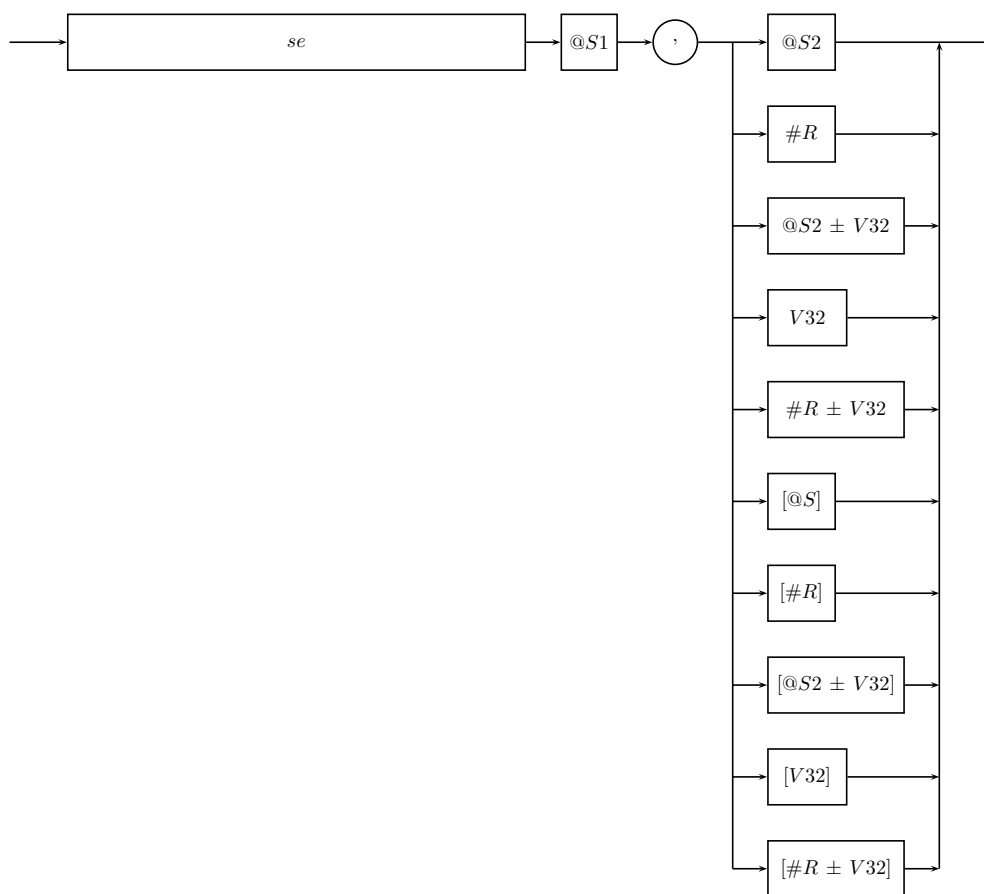


Рис. 1.92: Синтаксическое описание команды *se*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаг нуля (*ZF*) аргумента *ARG1* равен единице, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.89 set (SET)

Установка регистра

set ARG1, ARG2

Назначение: операция установки значения регистра

Синтаксис:

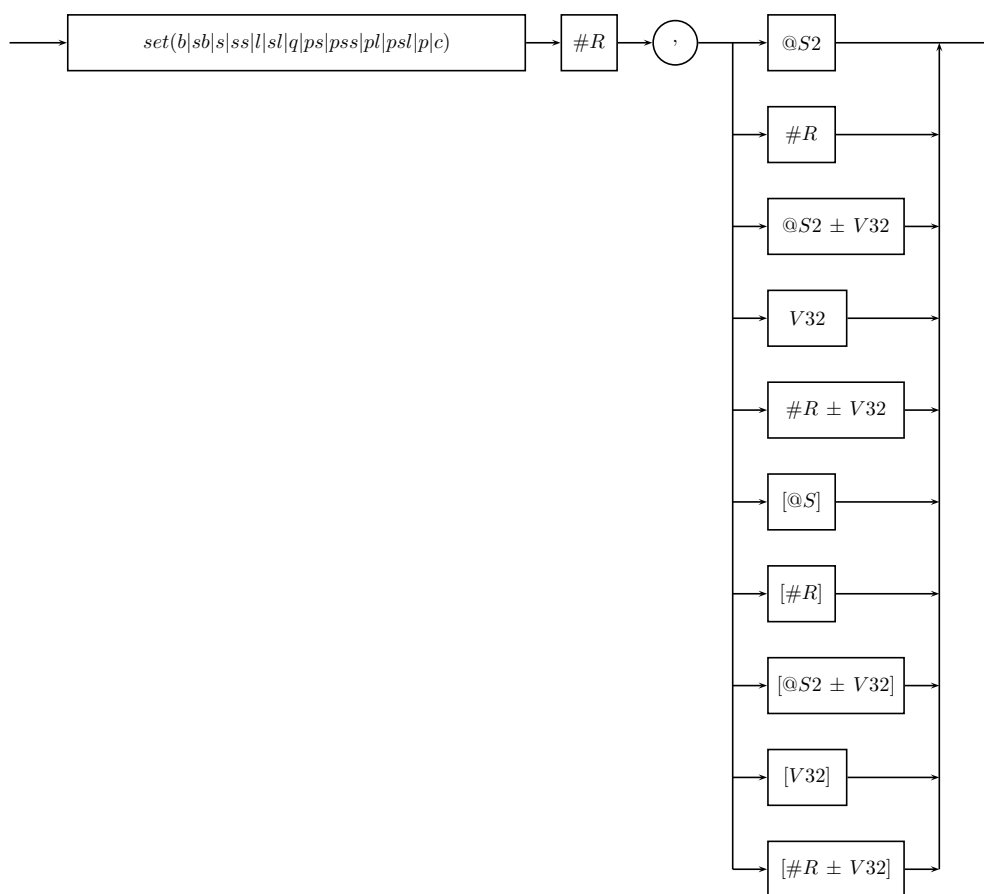


Рис. 1.93: Синтаксическое описание команды *set*

Наличие результата: нет

Алгоритм работы:

- установить значение регистра, заданного аргументом *ARG1*, равным значению аргумента *ARG2*, размером байт (знаковый или беззнаковый)), полу-слово (знаковое или беззнаковое), слово (знаковое или беззнаковое), двойное слово в зависимости от типа команды;

- для типов операции **signed byte**, **signed short** размножить знак до 31 разряда;

Интерпретация значений аргументов: согласно выше описанным синтаксису и алгоритму работы, первый аргумент команды определяет номер или имя регистра (в отличие от большинства остальных команд, первый аргумент которых определяет номер ссылки на результат предшествующей команды), значение которого необходимо установить; значение второго аргумента, сохраняемое в указанном регистре, формируется согласно общим правилам формирования второго аргумента команд.

Применение: команда `set` используется для установки 64-х разрядного значения регистра, равным значению, как знакового, так и беззнакового, второго аргумента. В случае использования знакового типа команды (`setsb`, `setss`, `setsl`) значения старших разрядов регистра устанавливаются равными знаковому разряду; для беззнаковых типов операций (`setb`, `sets`, `setl`, `setq`) значения старших разрядов регистра устанавливаются равными нулю.

Особенности выполнения: фактическая установка значения регистра осуществляется по окончании параграфа. Таким образом значение какого-либо регистра в рамках одного параграфа должно устанавливаться однократно, использование нового значения регистра возможно только в следующем параграфе. Если в одном параграфе значение одного и того же регистра устанавливается несколько раз, ассемблером будет выведено соответствующее предупреждение, а значение регистра по окончании параграфа будет установлено в значение, сформированное последней **выполненной** (выполнение команд не упорядочено: команда выполняется по готовности её аргументов) командой установки.

Пример:

```
1  .data
2
3  A:
4      .float 0f-1.2548E2
5  B:
6      .quad 0x01020304050607F8
7
8  .text
9
10 C:
```

```
11      jmp D
12
13      rdl A
14      setl #0, @1
15 complete
16
17 D:
18      setb #32, 0xAA
19
20      rdq B
21      setq #PSW, @1
22      setsb #0, @2
23
24      getsl #0
25      addsl @1, 1
26      setsl #1, @1
27
28      setl #32, 0xBBCCDDEE
29 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера `.float` в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 32-х разрядное число с плавающей точкой одинарной точности $-1.2548 * 10^2$;
- в строке №5 объявляется символ (идентификатор) `B`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №6 директивой ассемблера `.quad` в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 64-х разрядное целое беззнаковое число `0x01020304050607F8`;

- в строке №8 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №10 объявляется символ (идентификатор) *C*, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №11 командой `jmp` безусловно устанавливается адрес следующего параграфа, равным *D*;
- в строке №13 командой `rdi`, читается из памяти данных 32-х разрядное целое беззнаковое числа по адресу *A* и помещаются в коммутатор;
- в строке №14 командой `seti` в регистр общего назначения №0 помещается результат выполнения предшествующей команды: @1 — результат выполнения команды чтения в строке №13;
- в строке №15 командой `complete` завершается текущий параграф; происходит фактическая установка значения регистра общего назначения №0.
- в строке №17 объявляется символ (идентификатор) *D*, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №18 командой `setb` в индексный регистр №32 помещается значение `0xAA` (значения старших разрядов с 8 по 63 будут нулевыми, так как тип операции беззнаковый байт);
- в строке №20 командой `rdq`, читается из памяти данных 64-х разрядное целое беззнаковое число по адресу *B* и помещается в коммутатор;
- в строке №21 командой `setq` в управляющий регистр *PSW* помещается результат выполнения предшествующей команды: @1 — результат выполнения команды чтения в строке №20;
- в строке №22 командой `setsb` в регистр общего назначения №0 помещается результат выполнения предшествующей команды: @2 — результат выполнения команды чтения в строке №20 (помещается значение `0xF8`, значения старших разрядов с 8 по 63 будут равны значению 7 разряда, так как тип операции знаковый байт);
- в строке №24 командой `getsl` извлекается значение регистра общего назначения №0, установленное в предыдущем параграфе (не смотря на наличие команды `setsb` в строке №22);

- в строке №25 командой `addsl` складываются результат выполнения предшествующей команды: `@1` — результат выполнения команды извлечения в строке №24 (число `0x42faf5c3` — представление числа $-1.2548 * 10^2$ в IEEE754), и константное значение 1; оба аргумента команды `addsl` согласно суффиксу `sl` интерпретируются как 32-х разрядные целые знаковые числа; результат выполнения команды также интерпретируется как 32-х разрядное целое знаковое число и помещается в коммутатор;
- в строке №26 командой `setsl` в регистр общего назначения №1 помещается результат выполнения предшествующей команды: `@1` — результат выполнения команды сложения в строке №25 (помещается значение `0x42faf5c4`, значения старших разрядов с 32 по 63 будут равны значению 31 разряда, так как тип операции знаковый длинный);
- в строке №28 командой `setl` в индексный регистр №32 помещается значение `0xBBCCDDEE` (значения старших разрядов с 32 по 63 будут нулевые, так как тип операции беззнаковый длинный);
- в строке №29 командой `complete` завершается текущий параграф; происходит фактическая установка значения регистров общего назначения №№0,1, управляющего регистра *PSW*, индексного регистра №32 в одно из двух возможных значений (строки №№18,28).

1.4.4.90 sg (Select if Greater)

Выбор второго аргумента по условию

sg ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

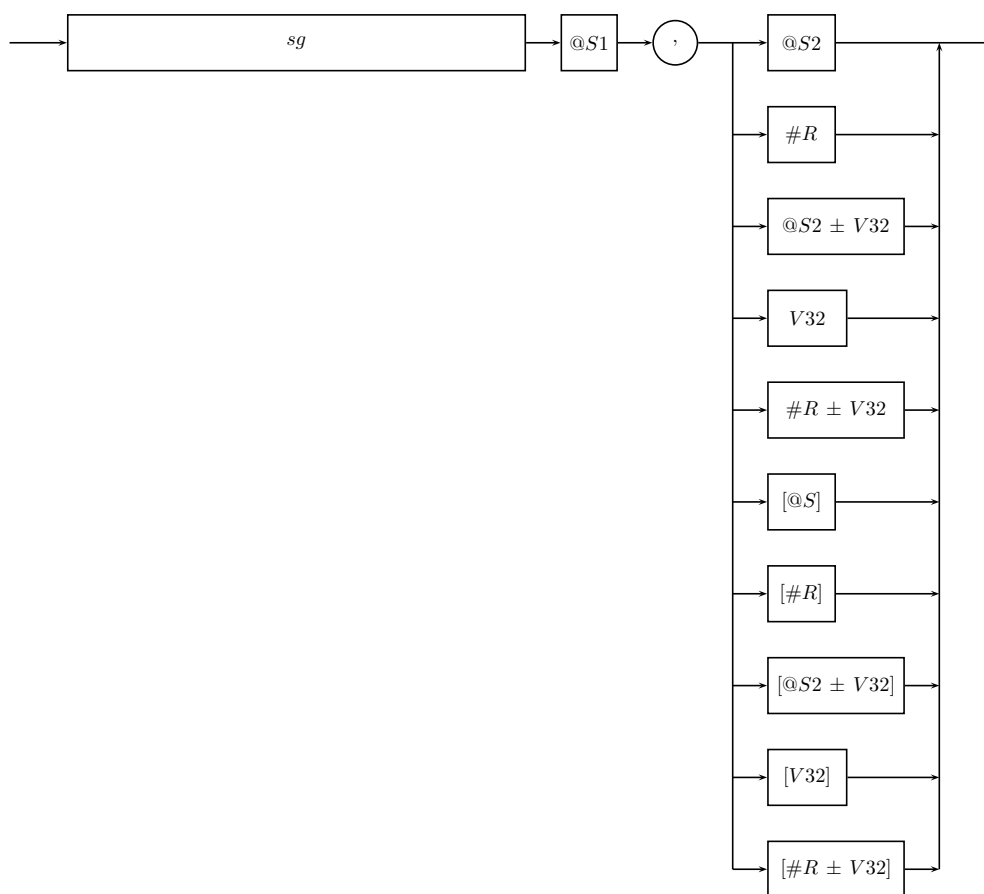


Рис. 1.94: Синтаксическое описание команды *sg*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* равны и флаг нуля (*ZF*) аргумента *ARG1* равен нулю, иначе – равным нулю;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.91 sge (Select if Greater and Equal)

Выбор второго аргумента по условию

sge ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

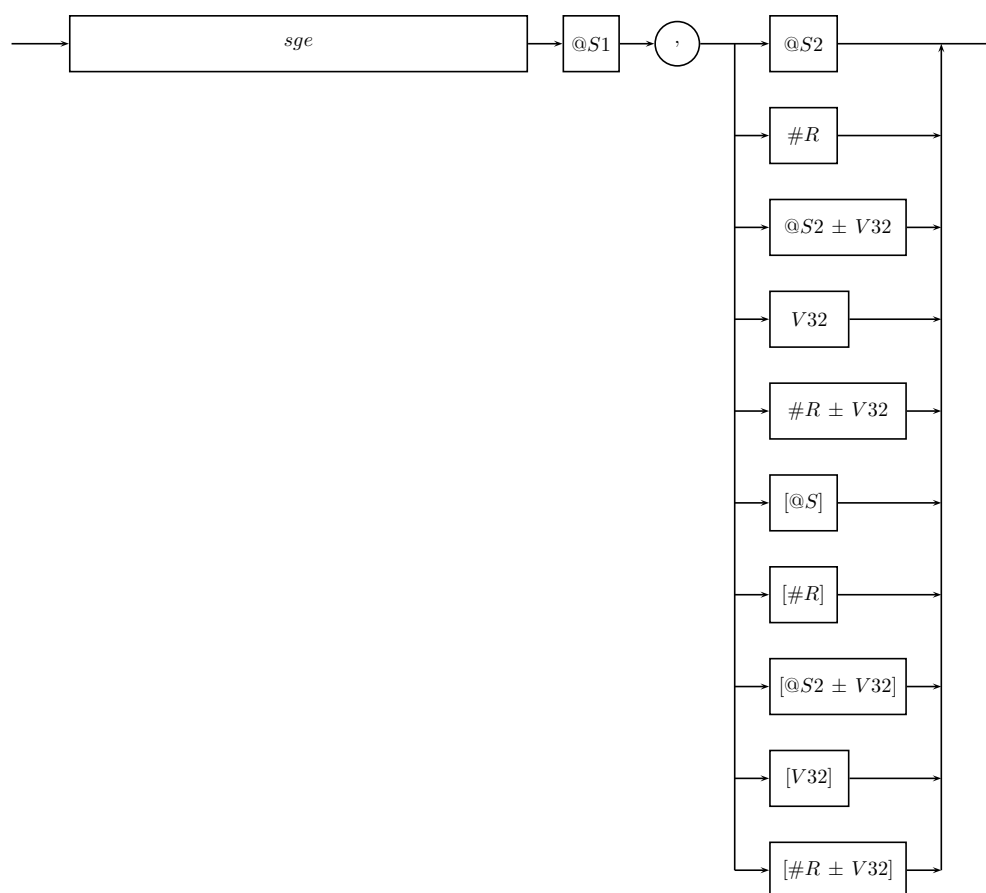


Рис. 1.95: Синтаксическое описание команды *sge*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента ARG2, если флаги знака (SF) и переполнения (OF) аргумента ARG1 равны, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.92 sl (Select if Less)

Выбор второго аргумента по условию

sl ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

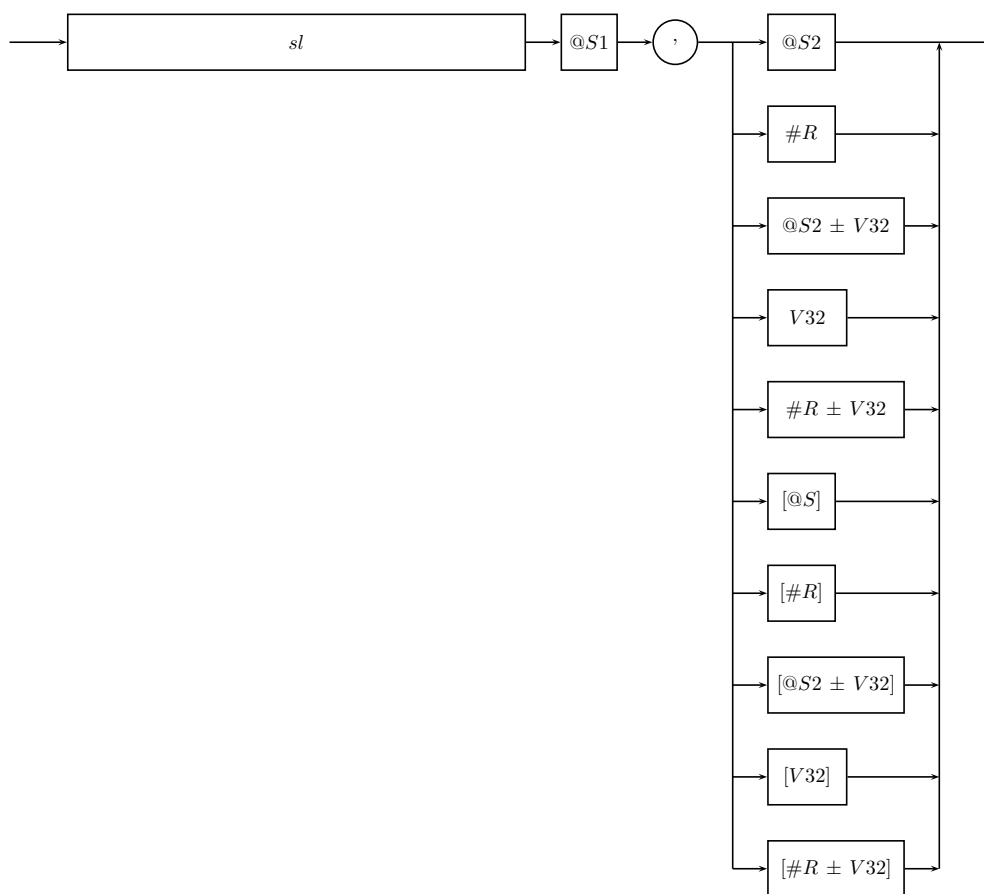


Рис. 1.96: Синтаксическое описание команды *sl*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* не равны, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.93 sle (Select if Less and Equal)

Выбор второго аргумента по условию

sle ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

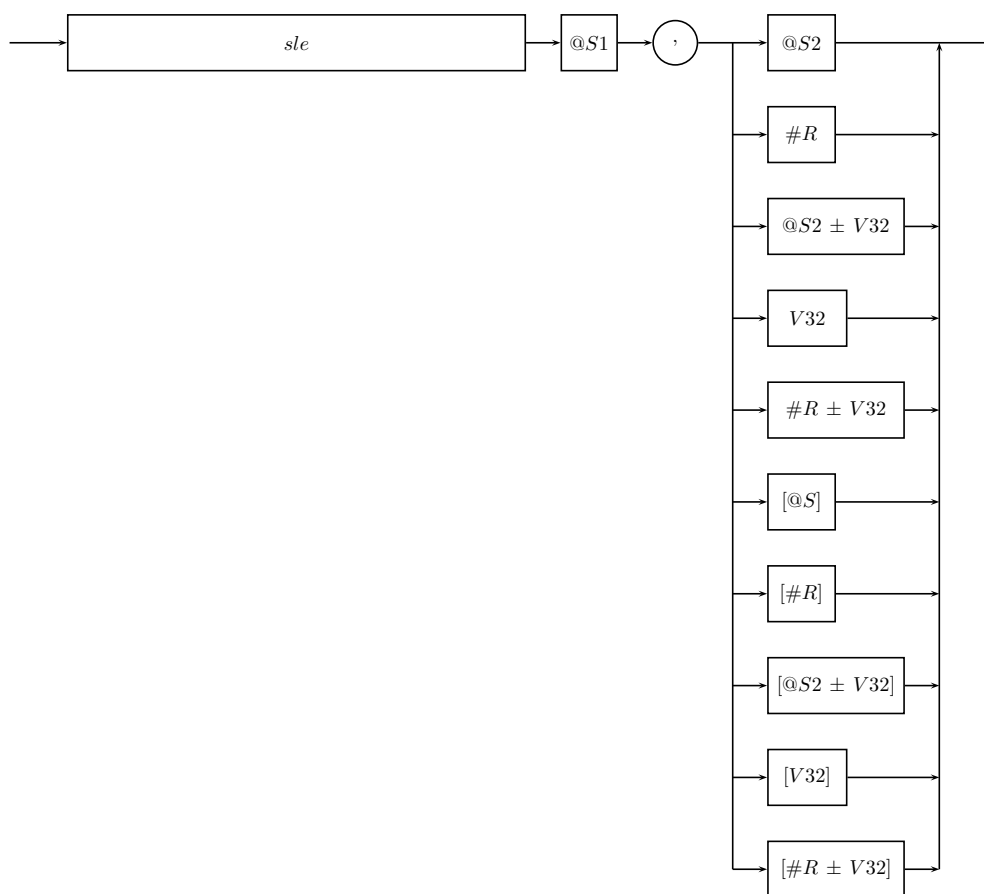


Рис. 1.97: Синтаксическое описание команды *sle*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаги знака (*SF*) и переполнения (*OF*) аргумента *ARG1* не равны или флаг нуля (*ZF*) аргумента *ARG1* равен единице, иначе – равным нулю;
- установить флаги;

- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.94 sll (Shift Logical Left) / sal (Shift Arithmetic Left)

Сдвиг логический/арифметический аргумента влево

$(sll|sal) ARG1, ARG2$

Назначение: операция логического/арифметического сдвига аргумента влево

Синтаксис:

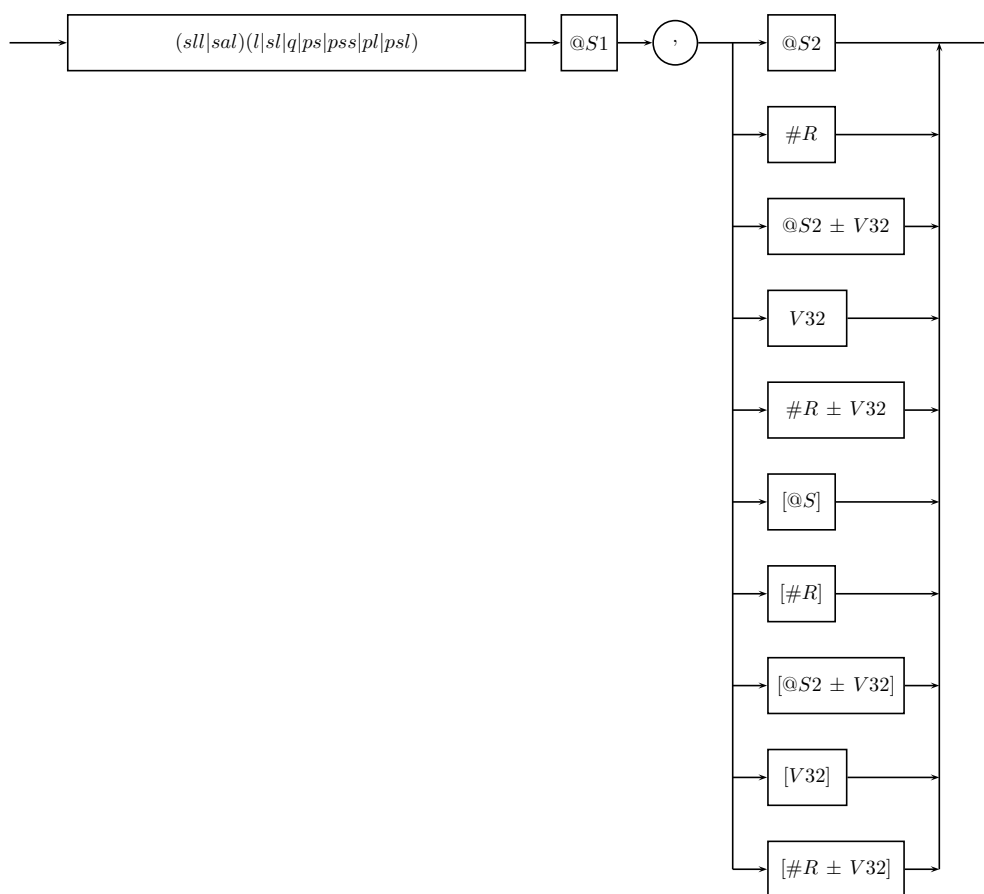


Рис. 1.98: Синтаксическое описание команды sll/sal

Наличие результата: да

Алгоритм работы:

- выполнить логический/арифметический сдвиг аргумента $ARG1$ влево на количество бит, заданных в аргументе $ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long, quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если старший бит аргумента $ARG1$ не равен старшему биту результата, иначе $OF = 0$;

флаг переноса $CF = 1$, если последний сдвигаемый бит равен 1, иначе – $CF = 0$;

флаг знака $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.95 slr (Shift Logical Right)

Сдвиг логический аргумента вправо

slr ARG1, ARG2

Назначение: операция логического сдвига аргумента вправо

Синтаксис:

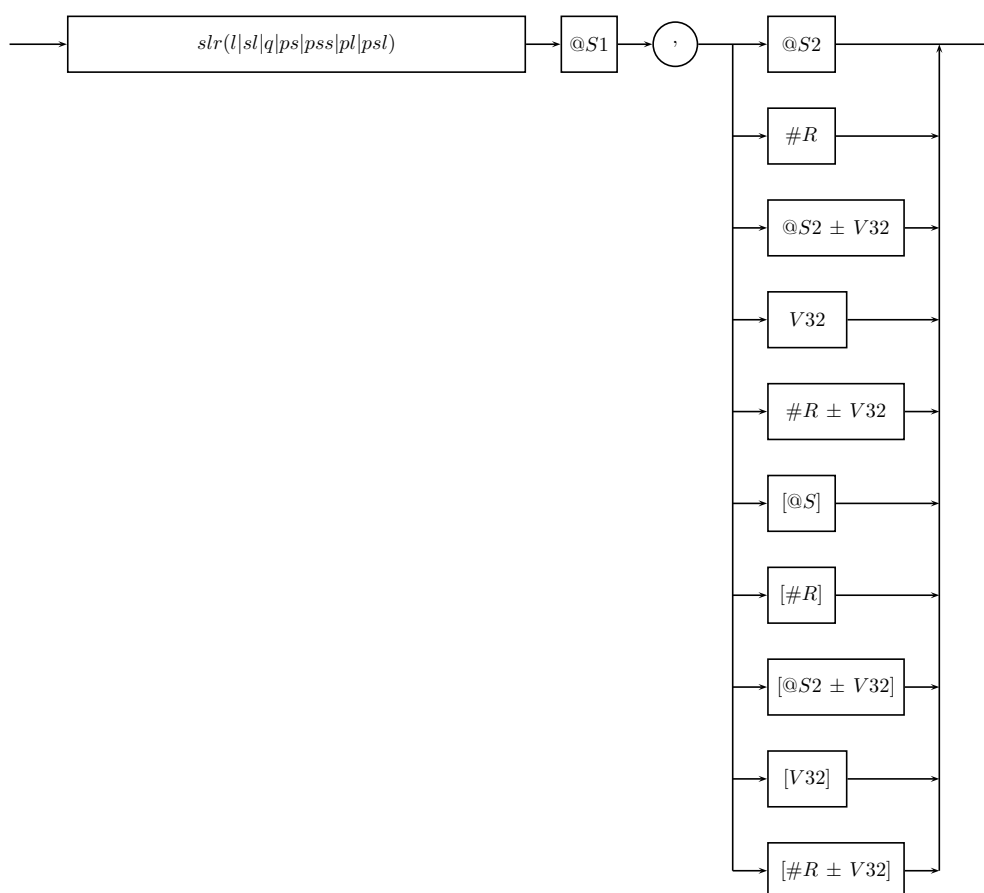


Рис. 1.99: Синтаксическое описание команды *slr*

Наличие результата: да

Алгоритм работы:

- выполнить логический сдвиг аргумента *ARG1* вправо на количество бит, заданных в аргументе *ARG2*;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операций **long, signed long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типов операций **long, signed long, quad**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если старший бит аргумента $ARG1$ не равен старшему биту результата, иначе $OF = 0$;

флаг переноса $CF = 1$, если последний сдвигаемый бит равен 1, иначе – $CF = 0$;

флаг знака $SF = 0$.

- для типов операций **packed short, packed signed short, packed long, packed signed long**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

1.4.4.96 sne (Select if Not Equal)

Выбор второго аргумента по условию

sne ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

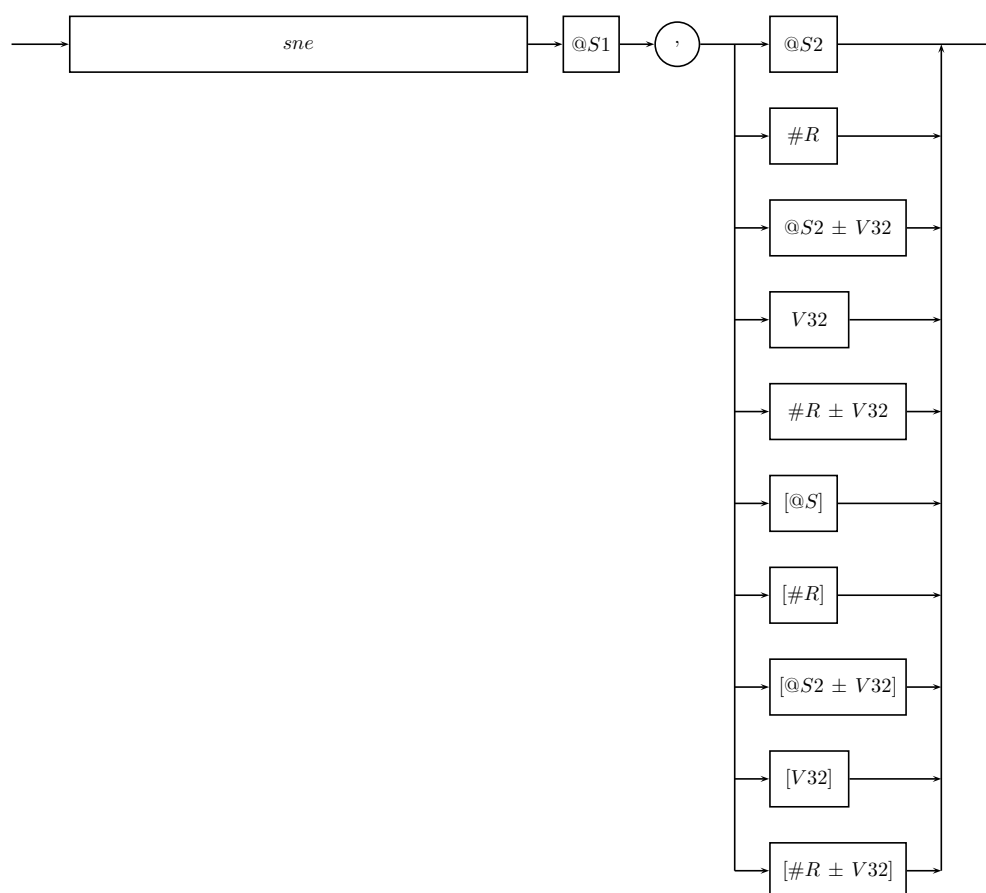


Рис. 1.100: Синтаксическое описание команды *sne*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаг нуля (*ZF*) аргумента *ARG1* равен нулю, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.97 sno (Select if Overflow flag unset)

Выбор второго аргумента по условию

sno ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

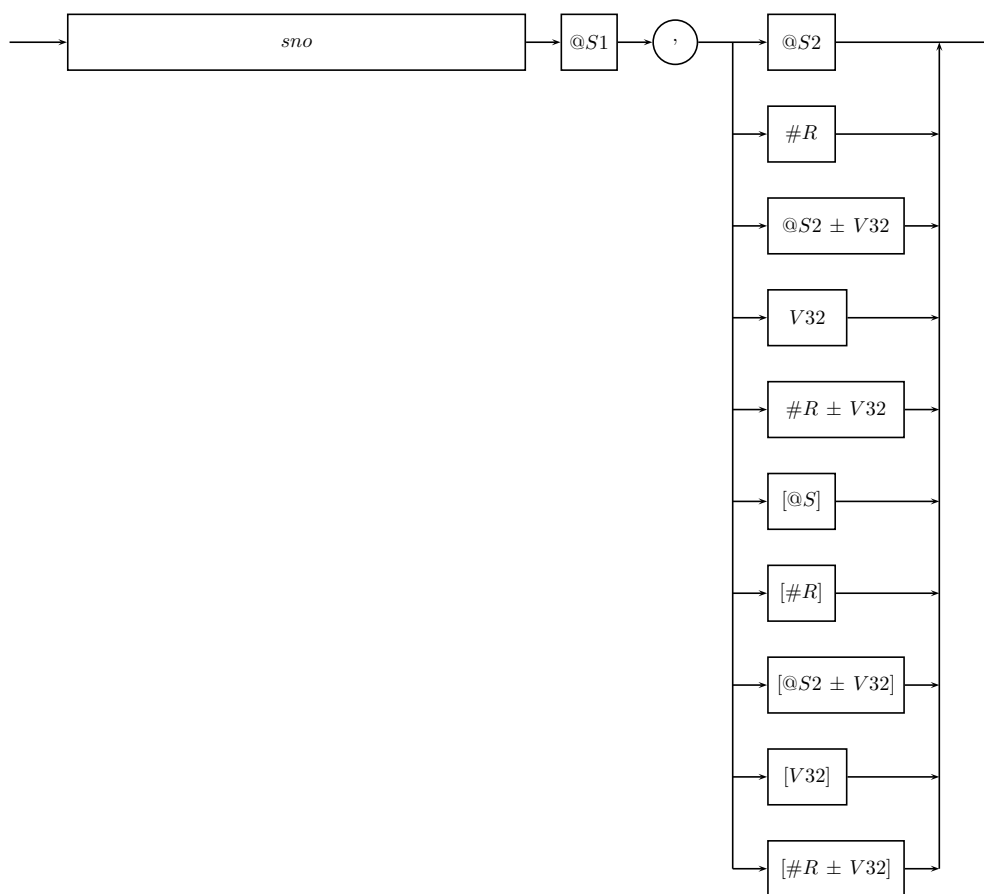


Рис. 1.101: Синтаксическое описание команды *sno*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG2*, если флаг переполнения (*OF*) аргумента *ARG1* равен нулю, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.98 sns (Select if Sign flag unset)

Выбор второго аргумента по условию

sns ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

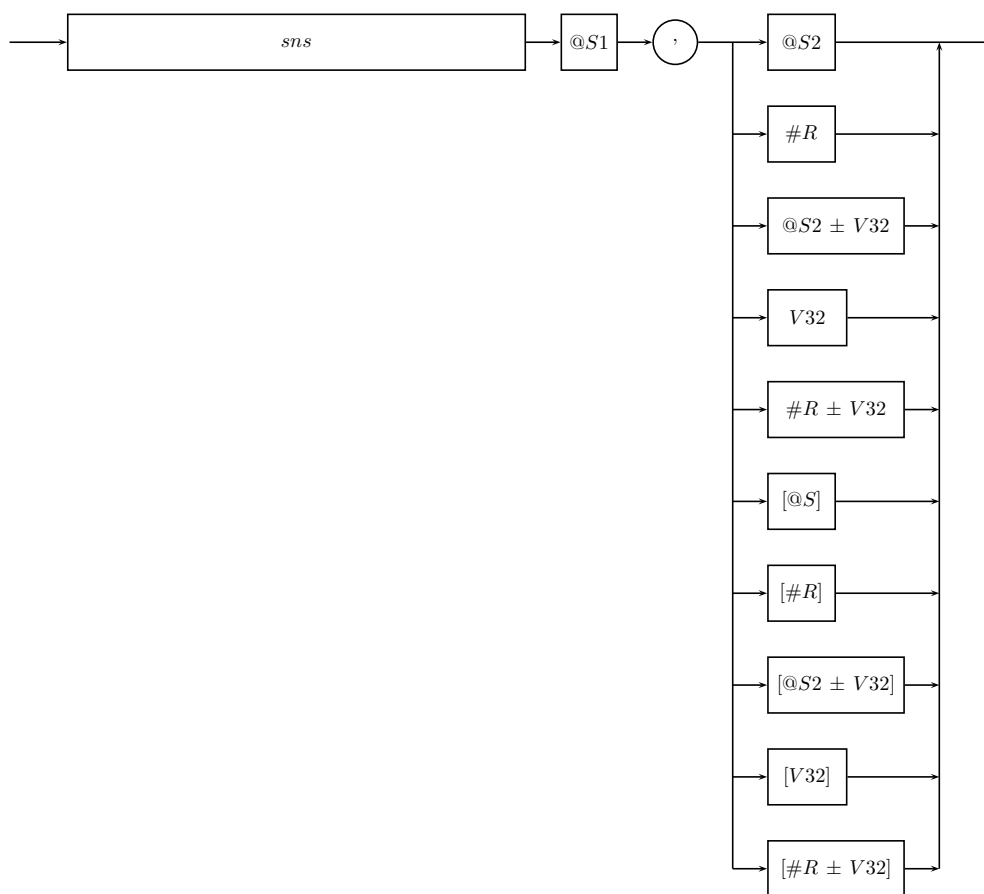


Рис. 1.102: Синтаксическое описание команды *sns*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг знака (*ZF*) аргумента *ARG2* равен нулю, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.99 so (Select if Overflow flag set)

Выбор второго аргумента по условию

so ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

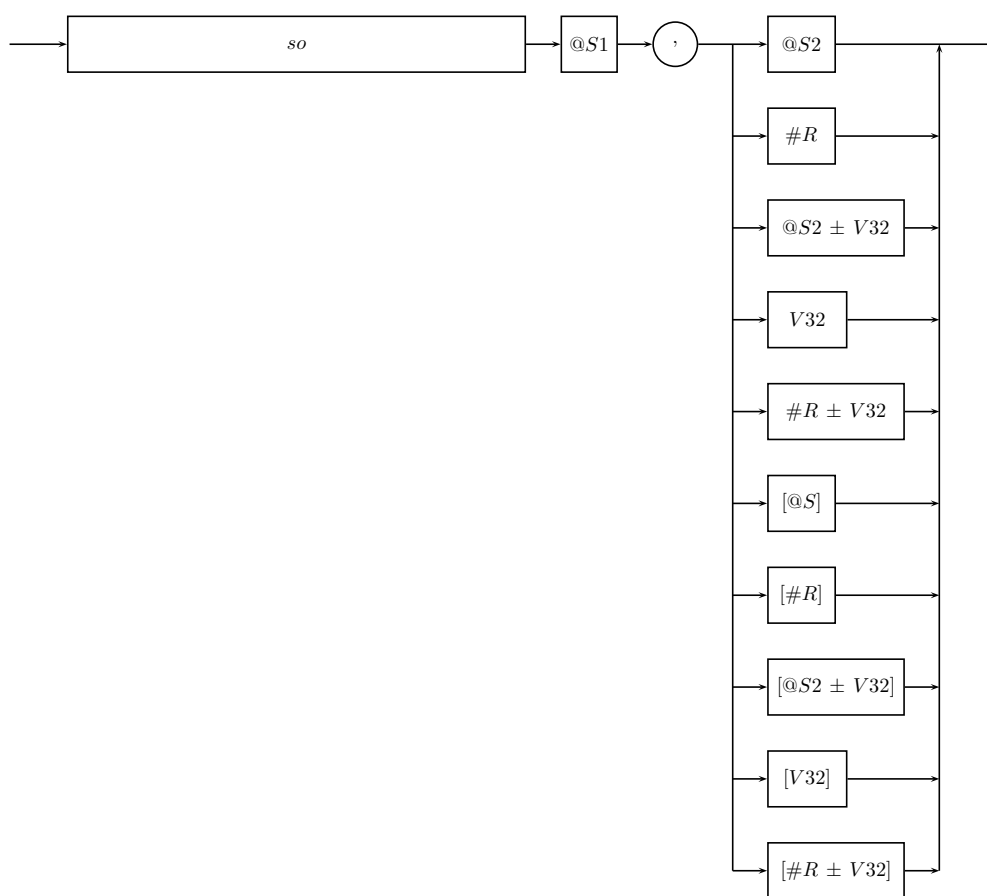


Рис. 1.103: Синтаксическое описание команды *so*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг переполнения (*OF*) аргумента *ARG2* равен единице, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.100 ss (Select if Sign flag set)

Выбор второго аргумента по условию

ss ARG1, ARG2

Назначение: команда условного выбора второго аргумента

Синтаксис:

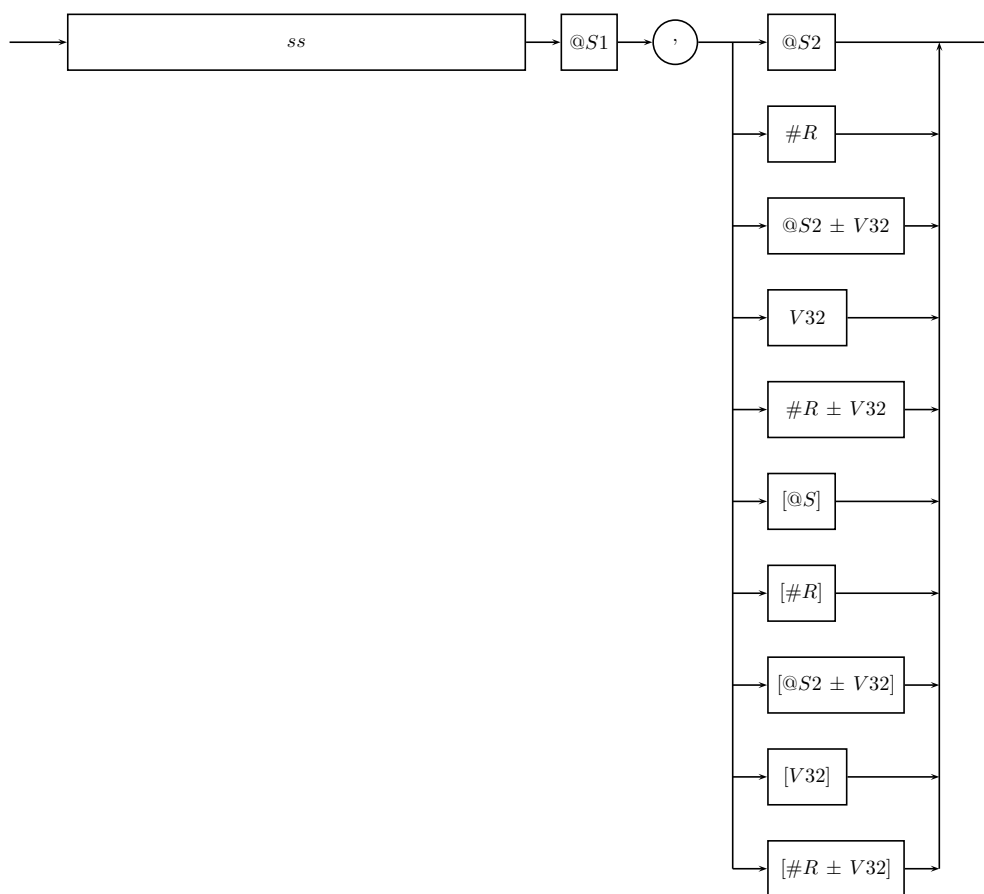


Рис. 1.104: Синтаксическое описание команды *ss*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента *ARG1*, если флаг знака (*SF*) аргумента *ARG2* равен единице, иначе – равным нулю;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат: результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.4.4.101 sqrt (SQuare RooT)

Квадратный корень

sqrt ARG

Назначение: операция извлечения квадратного корня

Синтаксис:

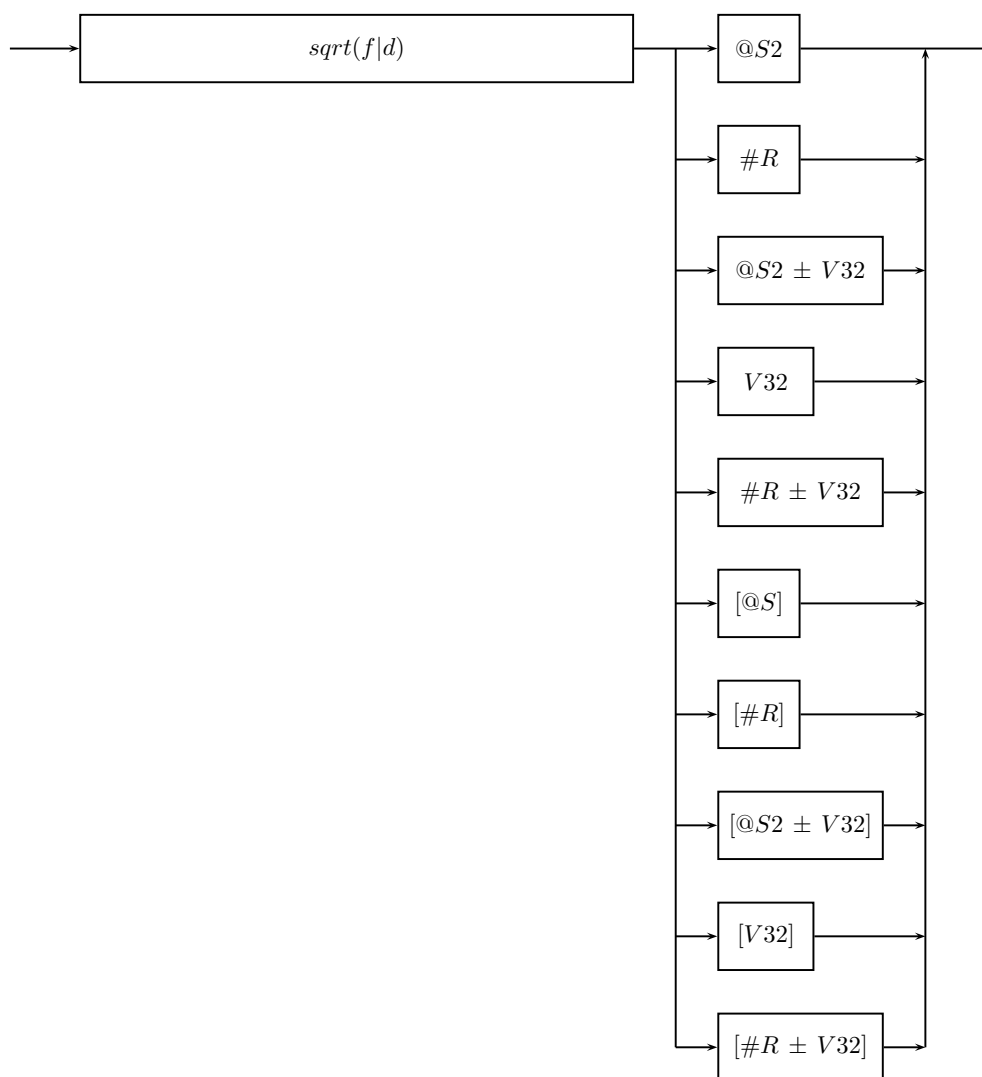


Рис. 1.105: Синтаксическое описание команды *sqrt*

Наличие результата: да

Алгоритм работы:

- выполнить извлечение квадратного корня аргумента $\sqrt{ARG}$;

- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **double** результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 1$, если в результате произошла потеря точности вследствие постнормализации (округления до ближайшего чётного), иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `sqrt` используется для извлечения квадратного корня из аргумента, значение которого интерпретируется в зависимости от типа операции.

1.4.4.102 sub (SUBtract)

Вычитание

sub ARG1, ARG2

Назначение: операция вычитания двух аргументов

Синтаксис:

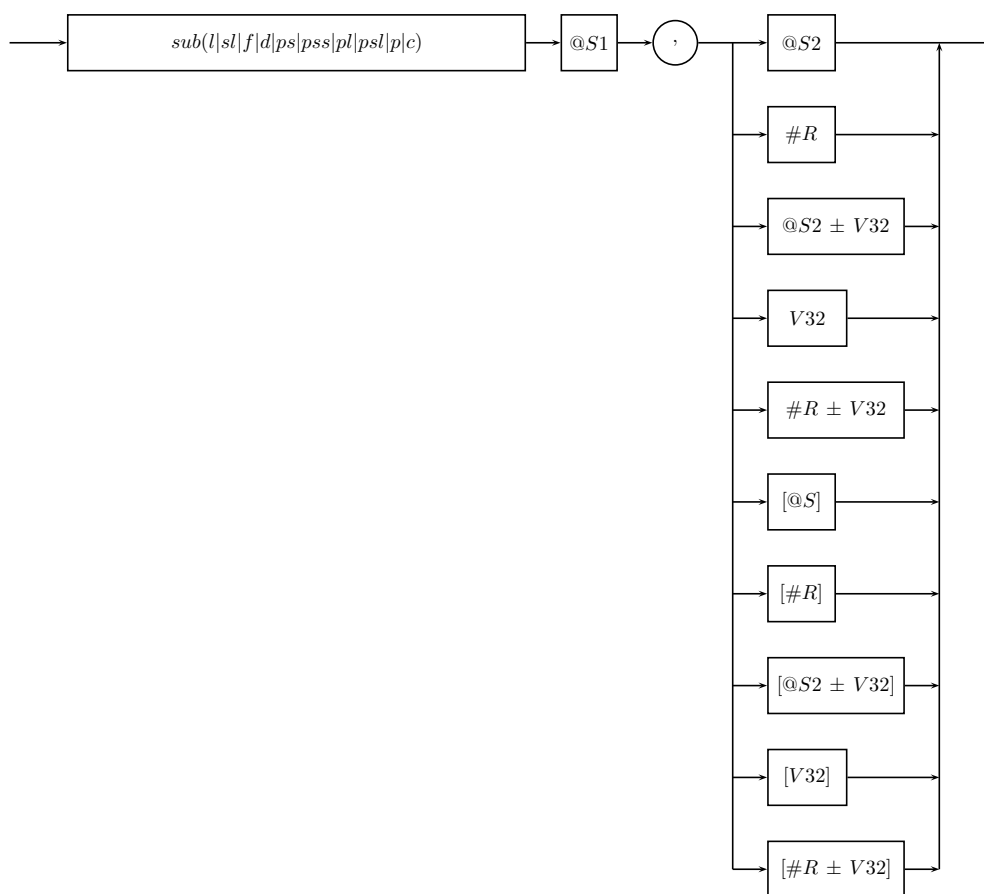


Рис. 1.106: Синтаксическое описание команды *sub*

Наличие результата: да

Алгоритм работы:

- выполнить вычитание $ARG1 - ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типов операции **long**, **signed long**, **float** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

- для типа операции **long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = CF$;

флаг переноса $CF = 1$, если $ARG1 < ARG2$, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типа операции **signed long**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если флаг переноса CF не равен старшему биту результата и $ARG1[31] = sARG2[31]$, где число в квадратных скобках есть номер бита аргумента, а $sARG2 = (not ARG2) + 1$ (*not* — операция побитового отрицания), иначе – $OF = 0$;

флаг переноса $CF = 1$, если расширенный бит результата равен единице, а также старший бит аргумента $ARG1$ равен единице и старший бит аргумента $ARG2$ равен нулю, иначе – $CF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

- для типов операций **packed short**, **packed signed short**, **packed long**, **packed signed long**:

флаг нуля $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = 0$;

флаг знака $SF = 0$.

– для типов операций **float, double**:

флаг нуля $ZF = 1$, если все биты результата равны нулю (т. е. значение -0 не приводит к установке данного флага), иначе – $ZF = 0$;

флаг переполнения $OF = 1$, если все биты экспоненты результата равны единице или расширенный бит экспоненты результата равен единице, либо хотя бы один из аргументов равен бесконечности, иначе – $OF = 0$;

флаг потери точности $IF = 1$, если в результате произошла потеря точности вследствие выравнивания мантисс или округления до ближайшего чётного, либо результатом является NaN иначе – $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

– для типов операций **packed float, complex float**:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг потери точности $IF = 0$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

Применение: команда `sub` используется для вычитания двух операндов, значение которых интерпретируется согласно типу операции.

Пример:

```
1  .data
2
3  B:
4      .float 0f12.8
5
6  .text
7
8  A:
9      rdl B
10     subf @1, 0f-5.6
11     wr1 @1, B + 4
12 complete
```


Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) *B*, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера `.float` в текущую секцию ассемблирования записывается 32-х разрядное вещественное число по текущему адресу ассемблирования;
- в строке №6 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №8 объявляется символ (идентификатор) *A*, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №9 командой `rdl` читается из памяти данных 32-х разрядное целое беззнаковое число по адресу *B* и помещается в коммутатор;
- в строке №10 командой `subf` выполняется операция вычитания результата выполнения предшествующей команды: `@1` — результат выполнения команды чтения в строке №9, и константы `–5.6`; оба аргумента команды `subf` согласно суффиксу `f` интерпретируются как вещественные числа одинарной точности размерностью 32 бита; результат выполнения команды также интерпретируется как 32-х разрядное вещественное число одинарной точности и помещается в коммутатор;
- в строке №11 командой `wrl` осуществляется запись в память данных по адресу $B + 4$ результата выполнения предшествующей команды: `@1` — результат выполнения команды вычитания в строке №10;
- в строке №12 командой `complete` завершается текущий параграф.

1.4.4.103 wr (WRite)

Запись

wr ARG1, ARG2

Назначение: операция записи значения в память

Синтаксис:

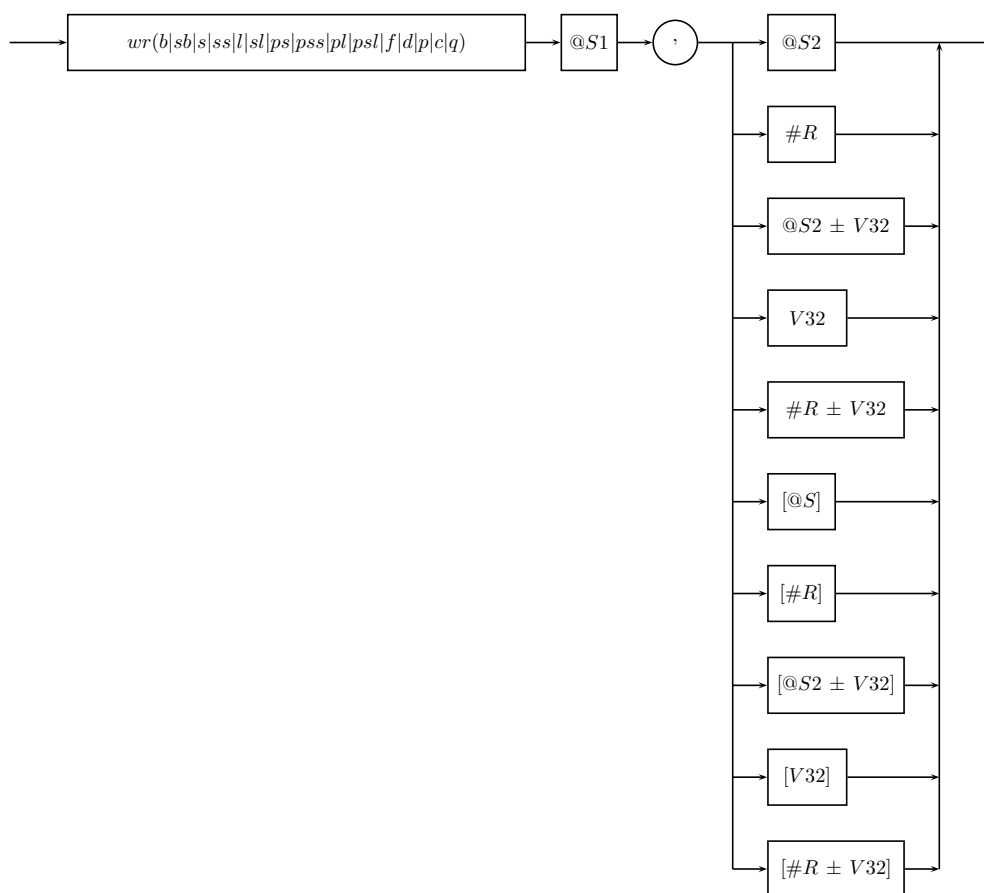


Рис. 1.107: Синтаксическое описание команды *wr*

Наличие результата: нет

Алгоритм работы:

- записать в зависимости от шестого бита регистра PSW (см. 1.2.2.3.1) и типа команды значение размером байт (знаковый или беззнаковый), полуслово (знаковое или беззнаковое), слово (знаковое или беззнаковое), двойное слово, заданное аргументом *ARG1*, в память данных по адресу, заданному аргументом *ARG2*;

Интерпретация значений аргументов: согласно выше описанному алгоритму работы, значение первого аргумента интерпретируется как оно есть (непосредственно используется), значение второго аргумента всегда интерпретируется как адрес памяти, по которому осуществляется запись значения первого аргумента. Другими словами данная команда всегда обращается к памяти вне зависимости от варианта формирования значения второго аргумента. Если аргумент команды указан в квадратных скобках (`[]` - операция чтения аргумента из памяти), то командой `wf` сначала выполняется чтение значения из памяти по адресу, указанному в квадратных скобках, а затем осуществляется запись значения аргумента `ARG1` в память по адресу, равному значению, которое было считано. Если для формирования значения аргумента используется значение регистра или результат предшествующей команды, т. е. используется ссылка на коммутатор (`@S`), то старшие 32 разряда (с 32 по 63) игнорируются.

Применение: команда `wf` используется для записи значения, как знакового, так и беззнакового, в память данных. По возможности следует располагать данную команду в конце параграфа.

Пример:

```
1  .data
2
3  A:
4      .float 0f-1.25E-1
5
6  .text
7
8  B:
9      rdl A
10     getl 0x3f028f5c; представление числа 0.51 в IEEE754
11     addf @1, @2
12     wrf @1, A + 4
13 complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера `.data` устанавливается текущая секция ассемблирования — секция инициализированных данных `data`;
- в строке №3 объявляется символ (идентификатор) `A`, который является меткой в текущей секции ассемблирования (`data`), и инициализируется текущим

значением адреса ассемблирования;

- в строке №4 директивой ассемблера `.float` в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 32-х разрядное число с плавающей точкой одинарной точности $-1.25 * 10^{-1}$;
- в строке №6 директивой ассемблера `.text` устанавливается текущая секция ассемблирования — секция исполняемых инструкций `text`;
- в строке №8 объявляется символ (идентификатор) *B*, который является меткой в текущей секции ассемблирования (`text`), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строке №9 командой `rdl`, читается из памяти данных 32-х разрядное целое беззнаковое число по адресу *A* и помещается в коммутатор;
- в строке №10 командой `getl` помещается в коммутатор 32-х разрядное целое беззнаковое число (константа `0x3f028f5`);
- в строке №11 командой `addf` складываются результаты выполнения двух предшествующих команд: `@1` — результат выполнения команды извлечения в строке №10, `@2` — результат выполнения команды чтения в строке №9; оба аргумента команды `addf` согласно суффиксу `f` интерпретируются как вещественные числа с плавающей точкой одинарной точности; результат выполнения команды также интерпретируются как вещественное число с плавающей точкой одинарной точности и помещается в коммутатор;
- в строке №12 командой `wrl` осуществляется запись в память данных по адресу *A* + 4 результата выполнения предшествующей команды: `@1` — результат выполнения команды сложения в строке №11;
- в строке №13 командой `complete` завершается текущий параграф.

1.4.4.104 wrd (WRite Direct)

Запись

wrd ARG1, ARG2

Назначение: операция записи значения в память

Синтаксис:

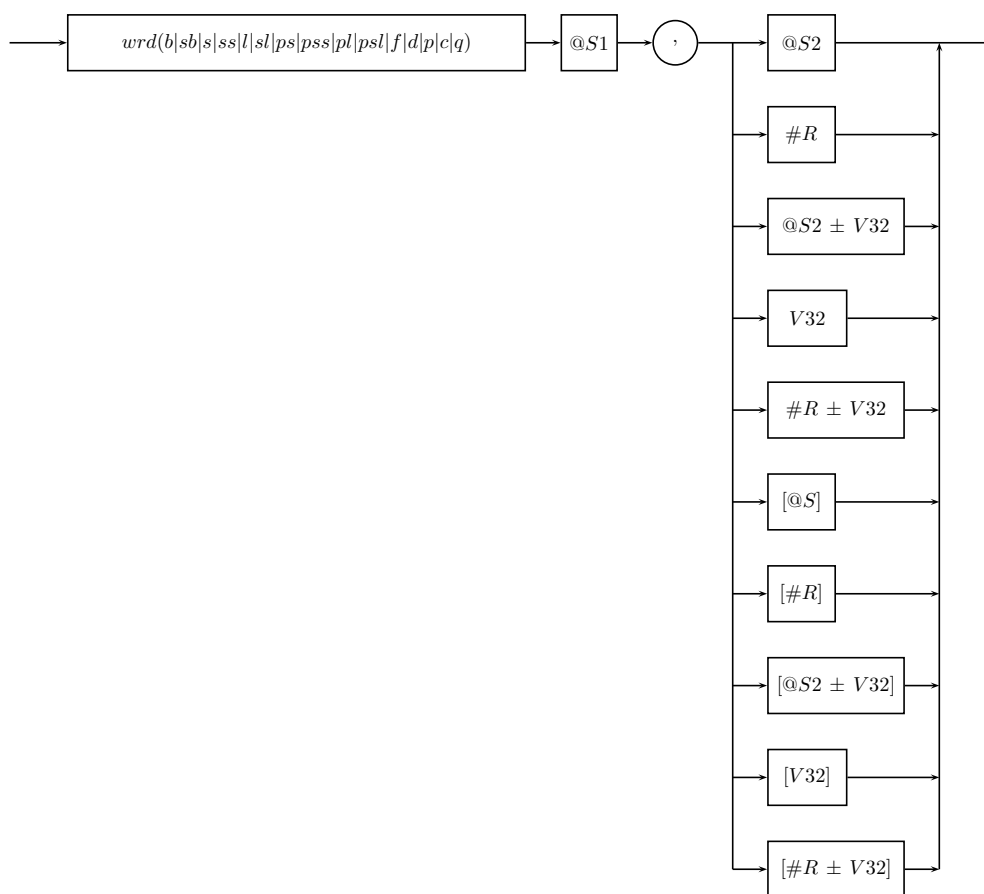


Рис. 1.108: Синтаксическое описание команды *wrd*

Наличие результата: нет

Алгоритм работы:

- безотлагательно записать в зависимости от типа команды значение размером байт (знаковый или беззнаковый), полуслово (знаковое или беззнаковое), слово (знаковое или беззнаковое), двойное слово, заданное аргументом *ARG1*, в память данных по адресу, заданному аргументом *ARG2*;

Интерпретация значений аргументов: согласно выше описанному алгоритму работы, значение первого аргумента интерпретируется как оно есть (непосредственно используется), значение второго аргумента всегда интерпретируется как адрес памяти, по которому осуществляется запись значения первого аргумента. Другими словами данная команда всегда обращается к памяти вне зависимости от варианта формирования значения второго аргумента. Если аргумент команды указан в квадратных скобках (`[]` - операция чтения аргумента из памяти), то командой `wtd` сначала выполняется чтение значения из памяти по адресу, указанному в квадратных скобках, а затем осуществляется запись значения аргумента *ARG1* в память по адресу, равному значению, которое было считано. Если для формирования значения аргумента используется значение регистра или результат предшествующей команды, т. е. используется ссылка на коммутатор (`@S`), то старшие 32 разряда (с 32 по 63) игнорируются.

Применение: команда `wtd` используется для записи значения, как знакового, так и беззнакового, в память данных. По возможности следует располагать данную команду в конце параграфа.

1.4.4.105 xor (XOR)

Логического сложение

xor ARG1, ARG2

Назначение: операция логического сложения по модулю 2 двух аргументов

Синтаксис:

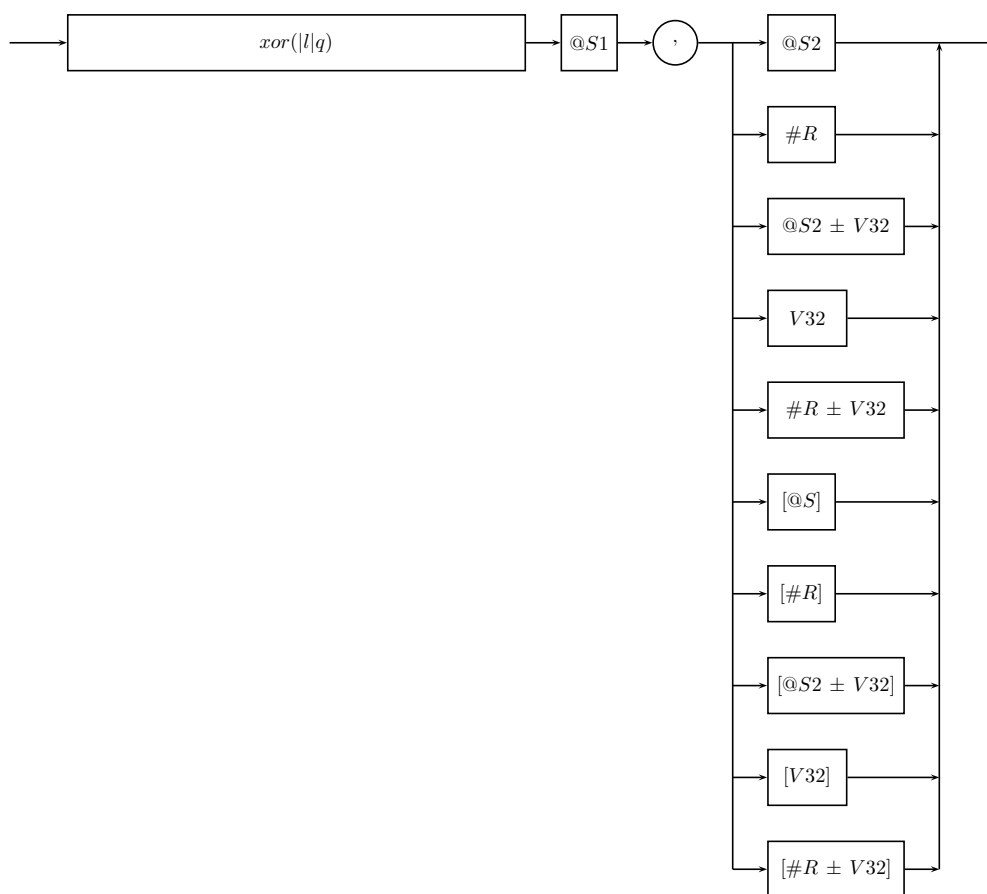


Рис. 1.109: Синтаксическое описание команды *xor*

Наличие результата: да

Алгоритм работы:

- выполнить логическое сложение по модулю 2 $ARG1 \wedge ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Результат:

- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

Состояние флагов результата после выполнения команды:

флаг нуля $ZF = 1$, если все биты результата равны нулю, иначе – $ZF = 0$;

флаг переполнения $OF = 0$;

флаг переноса $CF = OF$;

флаг знака $SF = 1$, если старший бит результата равен единице, иначе – $SF = 0$.

1.5 Система директив ассемблера

Все директивы ассемблера имеют имена, начинающиеся с символа точки «.», непосредственно за которой следуют символы латинского алфавита, как правило, в нижнем регистре.

1.5.1 `.alias name value`

Директива `.alias` служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Каждый алиас должен быть объявлен в одной строке до первого использования. Значение алиаса `value` состоит из любого набора символов и начинается с первого непробельного символа, следующего за `name`, и заканчивается либо концом строки, либо символом `';` или `'//'` (начало однострочного комментария). Допускается переопределение значения алиаса. Эта директива заменяет все последующие вхождения идентификатора `name` на его значение (`value`).

1.5.2 `.align abs_expr, abs_expr, abs_expr`

Директива `.align` предназначена для увеличения текущего адреса ассемблирования до заданной границы, путём пропуска необходимого количества единиц адресации.

Первое выражение, результат вычисления которого должен быть целым положительным числом, задаёт необходимое выравнивание в байтах.

Второе выражение, результат вычисления которого должен быть целым положительным числом, задаёт значение заполнителя, которое используется для инициализации пропускаемых байтов. Данное выражение (и запятая) могут быть пропущены. В этом случае значение заполнителя равно нулю.

Третье выражение, результат вычисления которого должен быть целым положительным числом, задаёт значение максимального количества байт, которые могут быть пропущены данной директивой для достижения запрошенного выравнивания. Если для достижения запрошенного выравнивания необходимо пропустить большее количество байтов, чем указанный максимум, то пропуск байтов совсем не выполняется. Другими словами, выполнить выравнивание только в том случае, если количество пропускаемых байтов меньше, либо равно указанного максимума. Данное выражение (и запятая) могут быть пропущены. В этом случае будет пропущено необходимое для выравнивания

количество байт. При использовании данной директивы возможен также пропуск значения заполнителя (второго аргумента) путём указания двух запятых после первого выражения.

Например, `'align 8'` продвинет текущий адрес ассемблирования вперёд до значения кратного 8. Если текущий адрес ассемблирования уже кратен 8, то не выполняется никаких действий.

1.5.3 `.ascii "string" ...`

`.ascii` ожидает ноль или более строковых литералов, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждая строка без завершающего нулевого байта размещается последовательно в инициализируемой области памяти данных (секция `.data`), начиная с текущего адреса ассемблирования. Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных.

1.5.4 `.asciiz "string" ...`

`.asciiz` ожидает ноль или более строковых литералов, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждая строка с завершающим нулевым байтом размещается последовательно в инициализируемой области памяти данных (секция `.data`), начиная с текущего адреса ассемблирования. Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных.

1.5.5 `.bss subsection`

`.bss` сообщает ассемблеру о необходимости ассемблировать все ниже следующие инструкции в конец подсекции `bss` с номером `subsection`. Значение `subsection` должно быть абсолютным выражением. Если значение `subsection` не задано, предполагается значение 0.

1.5.6 .byte expressions

.byte ожидает ноль или более выражений, разделенных запятыми. Данная директива может располагаться в секциях .data или .bss. Для каждого выражения эмитируется 8-ми разрядное число, которое, в процессе выполнения, является результатом вычисления этого выражения. Вычисленное 8-ми разрядное число размещается последовательно в инициализируемой области памяти данных (секция .data), начиная с текущего адреса ассемблирования. Если директива размещена в секции .bss, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если значение expression не задано, предполагается значение 0.

1.5.7 .comm symbol, length, align

.comm определяет общий (совместно используемый) символ (идентификатор) с именем symbol. В процессе компоновки общий символ из одного объектного файла может быть объединен с определенным или другим общим символом из другого объектного файла с тем же самым именем symbol. Если в процессе компоновки ни в одном объектном файле нет определенного символа с именем symbol, то для символа symbol будет выделено length байт в неинициализируемой области памяти данных (секция .bss), причём, если объявлено несколько одноименных общих символов symbol, будет выбран символ с максимальным значение length. Значение length должно быть абсолютным выражением. Значение align задаёт желаемое выравнивание символа symbol и должно быть абсолютным выражением.

1.5.8 .data subsection

.data сообщает ассемблеру о необходимости ассемблировать все ниже следующие инструкции в конец подсекции data с номером subsection. Значение subsection должно быть абсолютным выражением. Если значение subsection не задано, предполагается значение 0.

1.5.9 .double flonums

.double ожидает ноль или более чисел с плавающей точкой двойной точности, разделенных запятыми. Данная директива может располагаться в секциях .data или .bss. Каждое число является 64-х разрядным значением и размещается последовательно

в инициализируемой области памяти данных (секция `.data`), начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если значение `floatnum` не задано, предполагается значение 0.

1.5.10 `.else`

`.else` является частью ассемблерной директивы `.if`, которые обеспечивают поддержку условного ассемблирования кода. `.else` отмечает начало блока кода, который необходимо ассемблировать в случае, если условие предшествующей директивы `.if/.elsief` ложно.

1.5.11 `.elseif`

`.elseif` является частью ассемблерной директивы `.if`, которые обеспечивают поддержку условного ассемблирования кода. `.elseif` отмечает начало блока кода, который необходимо ассемблировать в случае, если условие предшествующей директивы `.if/.elsief` ложно, а условие данной директивы истинно.

1.5.12 `.end`

`.end` отмечает конец ассемблерного файла. Всё, что расположено ниже данной директивы игнорируется.

1.5.13 `.endif`

`.endif` является частью ассемблерной директивы `.if` и отмечает конец блока кода, который ассемблируется условно.

1.5.14 `.endr`

`.endr` является терминирующей директивой для директивы `.rept`.

1.5.15 .equ symbol, expression

.equ является синонимом директивы .set и устанавливает значение символа symbol равным результату вычисления выражения expression, а также изменяет атрибут типа соответственно новому значению. Атрибут связывания не изменяется. Изменять значение символа при помощи данной директивы возможно многократно. В таблице символов объектного файла будет сохранено последнее значение.

1.5.16 .equiv symbol, expression

.equiv устанавливает значение символа symbol равным результату вычисления выражения expression, если символ не был ранее определён.

1.5.17 .eqv symbol, expression

.eqv устанавливает значение символа symbol равным выражению expression, если символ не был ранее определён. Вычисление результата выражения происходит только при использовании символа symbol.

1.5.18 .err

При ассемблировании данной директивы в поток вывода ошибок выводится сообщение об ошибке, а также не генерируется объектный файл. Данная директива может быть использована для сигнализации ошибки в условно компилируемом блоке кода (при использовании директив условной компиляции)

1.5.19 .error “string”

При ассемблировании данной директивы в поток вывода ошибок выводится сообщение об ошибке с текстом string, а также не генерируется объектный файл. Данная директива может быть использована для сигнализации ошибки в условно компилируемом блоке кода (при использовании директив условной компиляции)

1.5.20 .fail expression

Сгенерировать сообщение или предупреждение. Если значение выражения `expression` больше, либо равно 500, ассемблер напечатает предупреждение. Если значение выражения `expression` меньше 500, ассемблер напечатает ошибку. Сообщение будет содержать значение выражения `expression`.

1.5.21 .file fileno filename

`.file` используется для добавления файла `filename` в секцию `.debug_line` при генерировании отладочной информации в формате DWARF2.

Аргумент `fileno` является выражением, результат вычисления которого должен быть уникальным целым положительным числом, которое используется в качестве индекса файла `filename`.

Аргумент `filename` является C строкой, заканчивающейся нулевым символом.

1.5.22 .fill repeat, size, value

`repeat`, `size` и `value` являются абсолютными выражениями. Для выражения `value` эмитируется `repeat` копий каждая размером `size` байт. Значение `size` должно быть в диапазоне от 0 до 8. Если значение `size` больше 8, то в качестве значения `size` используется значение 8. `size` и `value` являются опциональными. Если вторая запятая и `value` отсутствуют, то `value` равно 0. Если первая запятая и последующие токены отсутствуют `size` равно 1.

1.5.23 .float flonums

`.float` является синонимом директивы `.single` и ожидает ноль или более чисел с плавающей точкой одинарной точности, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждое число является 32-х разрядным значением и размещается последовательно в инициализируемой области памяти данных (секция `.data`), начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если значение `floatnum` не задано, предполагается значение 0.

1.5.24 .global names, .globl names

.global устанавливает атрибут связывания каждого символа, разделенных запятыми, из списка names в значение «GLOBAL», в результате чего в процессе компоновки символ будет видимым всем объектным файлам. Если символ не существует, он будет создан.

1.5.25 .if absolute_expression

.if обеспечивают поддержку условного ассемблирования кода и отмечает начало блока кода, который должен быть ассемблирован, если результат вычисления выражения absolute_expression не равен нулю. Также поддерживаются следующие варианты директивы .if:

.ifdef symbol

ассемблирует следующий блок кода, если символ symbol был ранее определен.

.ifb text

ассемблирует следующий блок кода, если операнд text пустой (не задан).

.ifeq absolute_expression

ассемблирует следующий блок кода, если результат вычисления выражения absolute_expression равен нулю.

.ifeqs string1, string2

ассемблирует следующий блок кода, если строки одинаковые (string1 == string2). Сравнение строк производится с учетом регистра литералов строки. Строки должны быть заключены в двойные кавычки.

.ifge absolute_expression

ассемблирует следующий блок кода, если результат вычисления выражения absolute_expression больше либо равен нулю.

.ifgt absolute_expression

ассемблирует следующий блок кода, если результат вычисления выражения absolute_expression строго больше нуля.

.ifle absolute_expression

ассемблирует следующий блок кода, если результат вычисления выражения `absolute_expression` меньше либо равен нулю.

.iflt absolute_expression

ассемблирует следующий блок кода, если результат вычисления выражения `absolute_expression` строго меньше нуля.

.ifnb text

ассемблирует следующий блок кода, если операнд `text` не пустой (задан).

.ifndef symbol или **.ifnotdef symbol**

ассемблирует следующий блок кода, если символ `symbol` не был ранее определён.

.ifne absolute_expression

ассемблирует следующий блок кода, если результат вычисления выражения `absolute_expression` не равен нулю.

.ifnes string1, string2

ассемблирует следующий блок кода, если строки различные (`string1 != string2`). Сравнение строк производится с учетом регистра литералов строки. Строки должны быть заключены в двойные кавычки.

1.5.26 **.include “file”**

`.include` обеспечивает способ включения вспомогательных (дополнительных) файлов в определенной позиции текущего файла. Исходный код из файла `file` ассемблируется так, как если бы он следовал в текущем файле с позиции расположения директивы `.include`. По окончании ассемблирования вспомогательного файла `file` продолжается ассемблирование текущего файла. Пути поиска вспомогательного файла `file` можно задать, используя опцию командной строки `'-I'`.

1.5.27 **.lcomm symbol, length**

`.lcomm` резервирует `length` байт для локального общего символа `symbol` в неинициализируемой области памяти данных (секция `.bss`). `length` должно быть абсолютным выражением. Атрибут связывания символа `symbol` устанавливается в значение «LOCAL», т. е. в процессе компоновки символ не будет виден другим объектным файлам.

1.5.28 .loc fileno lineno [column] [options]

При генерации отладочной информации в формате DWARF2 директива `.loc` добавляет в программу для машины состояний, ассемблируемую в секцию `.debug_line`, необходимые инструкции, исходя из значений аргументов `fileno`, `lineno` и необязательных аргументов `column`, `options`, которые будут выполнены машиной состояний перед формированием очередной записи таблицы строк, необходимой для отладки программы.

Аргумент `options` представляет из себя последовательность следующих токенов, разделённых запятыми, в любом порядке:

`basic_block` наличие данной опции приведёт к установке значения регистра `basic_block` в истинное значение при выполнении программы, содержащейся в секции `.debug_line`, машиной состояний;

`prologue_end` наличие данной опции приведёт к установке значения регистра `prologue_end` в истинное значение при выполнении программы, содержащейся в секции `.debug_line`, машиной состояний;

`epilogue_begin` наличие данной опции приведёт к установке значения регистра `epilogue_begin` в истинное значение при выполнении программы, содержащейся в секции `.debug_line`, машиной состояний;

`is_stmt value` наличие данной опции приведёт к установке значения `is_stmt` в значение `value`, которое должно быть 0 или 1, при выполнении программы, содержащейся в секции `.debug_line`, машиной состояний;

`isa value` наличие данной опции приведёт к установке значения `isa` в значение `value`, которое должно быть целым беззнаковым числом, при выполнении программы, содержащейся в секции `.debug_line`, машиной состояний;

`discriminator value` наличие данной опции приведёт к установке значения `discriminator` в значение `value`, которое должно быть целым беззнаковым числом, при выполнении программы, содержащейся в секции `.debug_line`, машиной состояний.

1.5.29 .local names

`.local` устанавливает атрибут связывания каждого символа, разделённых запятыми, из списка `names` в значение «LOCAL», в следствие чего данные символы не будут видимы другим объектным файлам в процессе компоновки. Если символ не существует, он будет создан.

1.5.30 .long expressions

.long ожидает ноль или более выражений, разделенных запятыми. Данная директива может располагаться в секциях .data или .bss. Для каждого выражения эмитируется 32-х разрядное число, которое, в процессе выполнения, является результатом вычисления этого выражения. Вычисленное 32-х разрядное число размещается последовательно в инициализируемой области памяти данных (секция .data), начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции .bss, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если значение expression не задано, предполагается значение 0.

1.5.31 .p2align abs_expr, abs_expr, abs_expr

Директива .p2align предназначена для увеличения текущего адреса ассемблирования до заданной границы, путём пропуска необходимого количества единиц адресации.

Первое выражение, результат вычисления которого должен быть целым положительным числом, задаёт количество младших разрядов адреса ассемблирования, значения которых должны быть нулевыми после выравнивания.

Второе выражение, результат вычисления которого должен быть целым положительным числом, задаёт значение заполнителя, которое используется для инициализации пропускаемых байтов. Данное выражение (и запятая) могут быть пропущены. В этом случае значение заполнителя равно нулю.

Третье выражение, результат вычисления которого должен быть целым положительным числом, задаёт значение максимального количества байт, которые могут быть пропущены данной директивой для достижения запрошенного выравнивания. Если для достижения запрошенного выравнивания необходимо пропустить большее количество байтов, чем указанный максимум, то пропуск байтов совсем не выполняется. Другими словами, выполнить выравнивание только в том случае, если количество пропускаемых байтов меньше, либо равно указанного максимума. Данное выражение (и запятая) могут быть пропущены. В этом случае будет пропущено необходимое для выравнивания количество байт. При использовании данной директивы возможен также пропуск значения заполнителя (второго аргумента) путём указания двух запятых после первого выражения.

Например, 'p2align 3' продвинет текущий адрес ассемблирования вперёд до значения

кратного 8. Если текущий адрес ассемблирования уже кратен 8, то не выполняется никаких действий.

1.5.32 .print string

Во время ассемблирования данной директивы ассемблер выведет в стандартный поток вывода строку string. Строка string должна быть заключена в двойные кавычки.

1.5.33 .quad expressions

.quad ожидает ноль или более выражений, разделенных запятыми. Данная директива может располагаться в секциях .data или .bss. Для каждого выражения эмитируется 64-х разрядное число, которое, в процессе выполнения, является результатом вычисления этого выражения. Вычисленное 64-х разрядное число размещается последовательно в инициализируемой области памяти данных (секция .data), начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции .bss, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если значение expression не задано, предполагается значение 0.

1.5.34 .rept count

Данная директива используется для организации повторения count раз последовательности инструкций, расположенных между данной директивой (.rept) и терминирующей её директивой .endr.

1.5.35 .section name [, "flags" [, @type [,flag_specific_arguments]]]

.section используется для указания ассемблеру необходимости ассемблировать последующий код в секцию name.

Необязательный аргумент flags используется для явного определения флагов секции name и представляет из себя заключённую в двойные кавычки строку, которая может содержать любую комбинацию следующих литералов:

- а данные секции name являются размещаемыми (allocatable);

- w данные секция name доступны для изменения (writable);
- x данные секции name содержат исполняемые инструкции (executable);
- M данные секции name могут быть объединены с данными других секций (mergeable);
- S данные секции name строки, заканчивающиеся нулевым символом (zero terminated).

Необязательный аргумент type может быть задан одним из следующих значений:

- @progbits секция name содержит данные;
- @nobits секция name не содержит данные, т. е. занимает только место в памяти;
- @note секция name содержит вспомогательные данные, которые не имеют никакого отношения к программе.

Если поле flags содержит литеру M, тогда вместе с типом type должен быть задан дополнительный аргумент entsize, например, .section DATA, "M", @progbits, 4. Секции, у которых установлен флаг M, но сброшен флаг S должны содержать константные данные фиксированного размера длиной entsize байт. Секции, у которых установлены оба флага M и S должны содержать строки, заканчивающиеся нулевым символом, где размер каждого символа равен entsize байт. entsize является выражением, результат вычисления которого должен быть целым положительным числом.

1.5.36 .set symbol, expression

.set является синонимом директивы .set и устанавливает значение символа symbol равным результату вычисления выражения expression, а также изменяет атрибут типа соответственно новому значению. Атрибут связывания не изменяется. Изменять значение символа при помощи данной директивы возможно многократно. В таблице символов объектного файла будет сохранено последнее значение.

1.5.37 .short expressions

.short ожидает ноль или более выражений, разделенных запятыми. Данная директива может располагаться в секциях .data или .bss. Для каждого выражения эмитируется 16-ти разрядное число, которое, в процессе выполнения, является результатом вычисления этого выражения. Вычисленное 16-ти разрядное число размещается последовательно в инициализируемой области памяти данных (секция .data), начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little

endian). Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если значение `expression` не задано, предполагается значение 0.

1.5.38 `.single flonums`

`.single` является синонимом директивы `.float` и ожидает ноль или более чисел с плавающей точкой одинарной точности, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждое число является 32-х разрядным значением и размещается последовательно в инициализируемой области памяти данных (секция `.data`), начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если значение `floatnum` не задано, предполагается значение 0.

1.5.39 `.size name, expression`

`.size` устанавливает размер в байтах символа `name` равным результату вычисления выражения `expression`.

1.5.40 `.skip size, fill`

`.skip` является синонимом директивы `.size` и размещает последовательно в инициализируемой области памяти данных (секция `.data`), начиная с текущего адреса ассемблирования, `size` байт, значение которого равно `fill`. `size` и `fill` являются абсолютными выражениями. Данная директива может располагаться в секциях `.data` или `.bss`. Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если запятая и значение `fill` не заданы, предполагается значение 0.

1.5.41 `.sleb128 expressions`

`.sleb128` ожидает ноль или более выражений, разделенных запятыми. Результат вычисления каждого из выражений, который должен быть целым знаковым числом, данной директивой кодируется в компактное представление переменной длины в формате

знакового LEB128 числа. Данная директива в частности используется при генерации отладочной информации в формате DWARF2.

1.5.42 `.space size, fill`

`.space` является синонимом директивы `.skip` и размещает последовательно в инициализируемой области памяти данных (секция `.data`), начиная с текущего адреса ассемблирования, `size` байт, значение которого равно `fill`. `size` и `fill` являются абсолютными выражениями. Данная директива может располагаться в секциях `.data` или `.bss`. Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Если запятая и значение `fill` не заданы, предполагается значение 0.

1.5.43 `.string "str"`

`.string` ожидает ноль или более строковых 8-ми битовых литералов, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждая литерал строки, а также завершающий нулевой байт, размещаются последовательно в инициализируемой области памяти данных (секция `.data`) в 8-ми разрядах, начиная с текущего адреса ассемблирования. Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных. Также поддерживаются следующие варианты директивы `.string`:

`.string8 "str"`

`.string8` ожидает ноль или более строковых 8-ми битовых литералов, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждая литерал строки, а также завершающий нулевой байт, размещаются последовательно в инициализируемой области памяти данных (секция `.data`) в 8-ми разрядах, начиная с текущего адреса ассемблирования. Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных.

`.string16 "str"`

`.string16` ожидает ноль или более строковых 8-ми битовых литералов, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждая литерал строки, а также завершающий нулевой байт, размещаются последовательно в инициализируемой области памяти данных (секция `.data`) в 16-ти

разрядах, начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных.

.string32 “str”

`.string32` ожидает ноль или более строковых 8-ми битовых литералов, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждая литерал строки, а также завершающий нулевой байт, размещаются последовательно в инициализируемой области памяти данных (секция `.data`) в 32-х разрядах, начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных.

.string64 “str”

`.string64` ожидает ноль или более строковых 8-ми битовых литералов, разделенных запятыми. Данная директива может располагаться в секциях `.data` или `.bss`. Каждая литерал строки, а также завершающий нулевой байт, размещаются последовательно в инициализируемой области памяти данных (секция `.data`) в 64-х разрядах, начиная с текущего адреса ассемблирования. Порядок расположения байтов от младшего к старшему (little endian). Если директива размещена в секции `.bss`, изменяется только значение текущего адреса ассемблирования без реального размещения данных.

1.5.44 .subsection abs_expr

`.subsection` изменяет номер подсекции, в которую ассемблируется последующий код, текущей секции.

Выражение `abs_expr`, результат вычисления которого должен быть целым положительным числом, определяет номер подсекции.

1.5.45 .syntax version

`.syntax` динамически переключает используемый синтаксис команд. Значение `version` должно быть либо V1, либо V2 (подробнее см. 1.4.3).

1.5.46 .text subsection

.text сообщает ассемблеру о необходимости ассемблировать все ниже следующие инструкции в конец подсекции text с номером subsection. Значение subsection должно быть абсолютным выражением. Если значение subsection не задано, предполагается значение 0.

1.5.47 .type name, type

.type устанавливает атрибут типа символа name равным значению type. Поддерживаемые типы (возможные значения type):

STT_NOTYPE тип символа name не определён

STT_OBJECT тип символа name является объектом данных (секции .data, .bss)

STT_FUNC тип символа name является функцией (секция .text)

STT_COMMON тип символа name является общим объектом данных (секция .bss)

1.5.48 .uleb128 expressions

.uleb128 ожидает ноль или более выражений, разделенных запятыми. Результат вычисления каждого из выражений, который должен быть целым беззнаковым числом, данной директивой кодируется в компактное представление переменной длины в формате беззнакового LEB128 числа. Данная директива в частности используется при генерации отладочной информации в формате DWARF2.

1.5.49 .warning “string”

При ассемблировании данной директивы в поток вывода ошибок выводится предупреждение с текстом string. Данная директива может быть использована для сигнализации ошибки в условно компилируемом блоке кода (при использовании директив условной компиляции)

1.5.50 .weak names

.weak устанавливает атрибут связывания каждого символа, разделенных запятыми, из списка names в значение «WEAK», в результате чего в процессе компоновки символ будет видимым всем объектным файлам. Если символ не существует, он будет создан.

1.6 Принципы программирования на ассемблере для мультиклеточного процессора

Программа на ассемблере представляет собой исходный текст со всеми включёнными в него файлами, который подаётся на вход ассемблера.

Исходный текст со всеми включёнными в него файлами называется единицей трансляции. Включение файлов в исходный код осуществляется директивой ассемблера `.include`.

В результате компиляции такой единицы трансляции ассемблер создаёт объектный файл, который затем, возможно, с другими объектными файлами, собирается редактором связей в исполняемую программу.

Исходный текст программы на ассемблере состоит из последовательности инструкций. Под инструкцией в данном случае понимается команда процессора или директива ассемблера. Каждая инструкция должна располагаться в отдельной строке, т. е. должна заканчиваться переводом строки, либо началом комментария.

Для указания области памяти, в которую будут ассемблироваться ниже следующие инструкции исходного текста программы, в ассемблере представлены следующие директивы: `.data`, `.bss`, `.text`.

Использование директивы `.data` в исходном тексте программы переключает текущую секцию ассемблирования на секцию `data`. В данной секции сохраняются начальные данные программы. Для инициализации данной секции в исходном тексте программы могут быть использованы такие директивы ассемблера как `.ascii`, `.asciz`, `.byte`, `.double`, `.float`, `.long`, `.short`, `.single`, `.string` и её варианты, `.quad`.

Использование директивы `.bss` в исходном тексте программы переключает текущую секцию ассемблирования на секцию `bss`. Данная секция используется для резервирования необходимого размера неинициализированного блока памяти данных, каждый байт которого в процессе загрузки программы инициализируется нулевым значением. Для выделения неинициализированного блока памяти данных в исходном тексте программы могут быть использованы такие директивы ассемблера как `.ascii`, `.asciz`, `.byte`, `.double`, `.float`, `.long`, `.short`, `.single`, `.string` и её варианты, `.quad` без явного указания инициализирующего значения (если инициализирующее значение указано, оно будет проигнорировано). Для резервирования необходимого размера неинициализированного блока памяти без переключения текущей секции ассемблирования могут быть использованы директивы ассемблера `.lcomm` и `.comm`.

Использование директивы `.text` в исходном тексте программы переключает текущую секцию ассемблирования на секцию `text`. В данной секции сохраняются исполняемые инструкции программы.

В каждой из выше перечисленных секций возможна установка метки. Метка определяется как символ, за которым следует двоеточие «:». Метки используются в качестве адреса памяти. Так, например, метка, объявленная в секции `.data` или `.bss`, может быть использована в командах процессора чтения или записи, а метка, объявленная в секции `.text`, — в командах процессора установки адреса следующего параграфа.

Кроме того, если текущей секцией ассемблирования является секция исполняемых инструкций (`.text`), то метка также интерпретируется как начало параграфа. Метки внутри параграфа не допускаются (если внутри параграфа указана метка, она будет проигнорирована). Каждый параграф заканчивается командой `complete`. Другими словами, параграф является макрокомандой, все инструкции которого должны быть выполнены; невозможно начать выполнения параграфа с середины (с не первой команды параграфа).

В общем случае вариант шаблона программы на ассемблере может быть представлен в следующем виде:

```
1  /*
2   Данная программа предназначена ...
3  */
4
5  /*
6   Размещение исходных данных в памяти данных
7  */
8  .data; текущая секция ассемблирования
9
10 Da: // объявление метки
11     /* директивы инициализации паямти данных, например */
12     .long 1, 0xABCD
13     .ascii "Some string"
14
15 Db: // объявление метки
16     /* директивы инициализации паямти данных */
17
18  /*
19   Резервирование неинициализированного блока памяти данных
```

необходимого размера в байтах без изменения текущей секции ассемблирования

20 **/*

21 **.lcomm** Ba, 24

22

23 */**

24 *Резервирование неинициализированного блока памяти данных*

25 **/*

26 **.bss** ; *текущая секция ассемблирования*

27

28 Bb: *// объявление метки*

29 */* директивы инициализации паямти данных, например */*

30 **.long** , , ,

31 **.byte**

32 **.skip** 12

33

34 Bc: *// объявление метки*

35 */* директивы инициализации паямти данных */*

36

37 */**

38 *Размещение исполняемых инструкций в памяти программ*

39 **/*

40 **.text** ; *текущая секция ассемблирования*

41

42 Ta: *// объявление метки, а также начало нового параграфа*

43 **rdl** Da

44 **rdl** Da + 4

45 **mull** @1, @2

46 **wrl** @1, Bc

47 **complete** *// завершение параграфа*

48

49 Tb: *// объявление метки, а также начало нового параграфа*

50 */* команды процессора */*

51 **complete** *// завершение параграфа*

1.7 Прерывания и их обработка

Система прерываний мультиклеточного процессора МСр042R100102 допускает обработку 44 прерываний. Прерывания можно условно разделить на две группы:

- системные прерывания, занимают адреса с 0 по 15 в контроллере прерываний;
- прерывания от периферии, занимают адреса с 16 по 46.

Структура прерываний

Номер	Описание
Немаскируемые системные прерывания	
0	Немаскируемое внешнее прерывание (ENMI)
1	Зарезервировано
2	Немаскируемое исключение в аппаратной части (PERE)
3	Немаскируемое программное исключение (PRGE)
4	Зарезервировано
5	Зарезервировано
Маскируемые системные прерывания	
6	Маскируемое прерывание: внешний сигнал wake_up (WKUP)
7	Маскируемое прерывание от контроллера транзакций (DTC0)
8	Маскируемое прерывание от системного таймера (ST0)
9	Маскируемое прерывание от системного таймера (ST1)
10	Маскируемое прерывание от системного таймера (ST2)
11	Маскируемое прерывание от системного таймера (ST3)
12	Маскируемое программное прерывание (SW0)
13	Маскируемое программное прерывание (SW1)
14	Маскируемое программное прерывание (SW2)
15	Маскируемое программное прерывание (SW3)
Прерывания от периферии	
16	Маскируемое прерывание от ADC0
17	Маскируемое прерывание от DAC0
18	Маскируемое прерывание от ADC1
19	Маскируемое прерывание от DAC1
20	Маскируемое прерывание от UART0
21	Маскируемое прерывание от UART1
Продолжение на следующей странице	

Номер	Описание
22	Маскируемое прерывание от UART2
23	Маскируемое прерывание от UART3
24	Маскируемое прерывание от I2C0
25	Маскируемое прерывание от I2C1
26	Маскируемое прерывание от SPI0
27	Маскируемое прерывание от SPI1
28	Маскируемое прерывание от I2S0
29	Маскируемое прерывание от GPTIM0
30	Маскируемое прерывание от GPTIM1
31	Маскируемое прерывание от GPTIM2
32	Маскируемое прерывание от GPTIM3
33	Маскируемое прерывание от WDT
34	Маскируемое прерывание от RTC
35	Маскируемое прерывание от GPIOA
36	Маскируемое прерывание от GPIOB
37	Маскируемое прерывание от GPIOC
38	Маскируемое прерывание от GPIOD
39	Маскируемое прерывание от GPIOE
40	Маскируемое прерывание от GPIOF
41	Маскируемое прерывание от ETHERNET0
42	Маскируемое прерывание от USB0
43	Маскируемое прерывание от USB0_EPI
44	Маскируемое прерывание от USB0_EPO
45	Маскируемое прерывание от PWM0
46	Маскируемое прерывание от STAT
47 — 63	Зарезервировано

Источник с номером «0» имеет наивысший приоритет при обработке прерываний.

Немаскируемые прерывания:

Прерывания с номерами 0, 2, 3 являются немаскируемыми. Немаскируемые прерывания приводят к немедленному переходу на программу обработки прерываний. Их нельзя запретить, они разрешены сразу после начала работы ядра.

Маскируемые прерывания:

Прерывания с номерами 6 — 46 являются маскируемыми, они глобально разрешаются битом ONIRQS в регистре PSW (1.2.2.3.1). Индивидуальное разрешение задается регистром MSKR (1.2.2.3.3).

Для работы системы прерывания и функционирования программ обработки прерываний имеются следующие регистры: INTR (1.2.2.3.2), MSKR (1.2.2.3.3), ER (1.2.2.3.4), FORCE (1.2.2.3.6), IRETADDR (1.2.2.3.5), IHOOKADDR (1.2.2.3.7), INTNUMR (1.2.2.3.8).

Регистры INTR, MSKR, ER, FORCE, INTNUMR относятся к управлению контроллером прерываний. Регистры IRETADDR, IHOOKADDR используются программным алгоритмом.

Порядок обработки прерываний:

В случае возникновения события, вызывающего запрос на обработку прерывания, оно фиксируется в регистре INTR. Регистр MSKR является фильтром, который допускает или нет сигналы из регистра INTR к дальнейшей обработке. После регистра MSKR сигналы попадают в схему определения приоритета, где происходит определение самого приоритетного прерывания в данном такте, его номер заносится в регистр INTNUMR. Переход на обработчик прерывания происходит в конце параграфа, когда завершаются все записи в память и регистры, если таковые были в параграфе.

Адрес начального обработчика прерываний находится в регистре IHOOKADDR. Ядро работает только с одним адресом перехода на обработчик прерываний. По этому адресу может находиться любой алгоритм на усмотрение пользователя. Определение порядка работы с конкретным запросом прерывания возложено на программиста. Например, программист может расположить в первичном обработчике прерываний код, который осуществляет диспетчеризацию и перенаправление программы на необходимый адрес, где располагается код обработчика для конкретного прерывания.

Адрес возврата автоматически запоминается в регистре IRETADDR, программист имеет к нему полный доступ.

Латентность времени перехода на обработчик прерывания имеет зависимость от программы пользователя. Рекомендуется перед стартом программы очистить регистр INTR записью 1 во все биты регистра.

Важно заметить, что для каждой группы клеток можно задавать обработчик прерываний в регистре IHOOKADDR, задавать MSKR, т. к. у каждой группы клеток имеется свой набор этих регистров.

1.8 Динамическая реконфигурация

Способность мультиклеточной архитектуры программно перераспределять свои ресурсы называется динамической реконфигурацией. При этом если мультиклеточное ядро разделяет task-группу на несколько task-групп, то это называется декомпозиция. Процесс объединения нескольких task-групп в одну task-группу — называется композицией. **При этом, для обеспечения корректной работы необходимо, чтобы каждая клетка входила в одну и только в одну task-группу.**

Архитектура мультиклеточного процессора не налагает каких-либо ограничений на количество task-групп и количество клеток в task-группе. Все клетки могут быть объединены в одну task-группу. Может быть сформировано 4 task-группы, в каждой будет по одной клетке. Конкретное распределение клеток по task-группам определяется программистом и может неоднократно меняться в процессе вычислений.

Так как каждая клетка мультиклеточного процессора имеет свой набор регистров общего назначения, а также индексных и управляющих регистров, то необходимо учитывать то факт, что при динамической реконфигурации все клетки сохраняют значения выше перечисленных регистров, которые они получили при работе в составе предыдущих task-групп. Следовательно, при декомпозиции во всех клетках новой task-группы значения регистров будут равны. При композиции клеток у каждой клетки может быть свой набор значений. Для того чтобы обеспечить равенство значений, требуемое содержимое регистров необходимо записать в память до объединения клеток и восстановить его после композиции.

Для динамической реконфигурации клеток в мультиклеточном процессоре используются два системных регистра: NEWADDR и ICR.

Регистр NEWADDR предназначен для формирования адреса перехода на следующий параграф для группы клеток. Значение данного регистра должно быть задано в следующем формате:

- в разрядах 00..31 указывается значение адреса перехода на следующий параграф;
- в разрядах 32..35 указывается 4-х битная маска, определяющая клетки, для которых назначается указанное значение (адрес перехода) (разряд 35 определяет клетку с физическим номером 0, 34 — 1, 33 — 2, 32 — 3).

Так например, фрагмент следующего кода

```
1    test :  
2    .....
```



```
3      getl 0x05
4      patch @1, 0xA0B0
5      setl #NEWADDR, @1
6      .....
7      complete
```

устанавливает значение регистра NEWADDR только клеток №№1,3 со значением адреса перехода на следующий параграф 0xA0B0.

Регистр ICR предназначен для определения подмножества клеток образующих task-группу. Значение данного регистра должно быть задано в следующем формате:

- в разрядах 00..03 указывается 4-х битное значение, которое определяет принадлежность клетки к данной task-группе (разряд 3 определяет клетку с физическим номером 0, 2 — 1, 1 — 2, 0 — 3);
- в разрядах 32..35 указывается 4-х битная маска, определяющая клетки, для которых назначается указанное значение (разряд 35 определяет клетку с физическим номером 0, 34 — №1, 33 — №2, 32 — №3).

1.8.1 Декомпозиция

Для task-группы клеток, которую необходимо разделить, требуется выполнить следующие действия:

1. установить значения битов RCFG и CLNBUF регистра PSW в состояние 1. RCFG — это бит динамической реконфигурации, а CLNBUF — указание на необходимость завершения выполнения всех команд следующего параграфа до выполнения перехода.

```
1      test :
2      getl #PSW
3      orl @1, 0x180 ; 7 (RCFG) и 8 (CLNBUF) биты регист
                     па PSW
4      setl #PSW, @1
5      jmp reconf
6      complete
```

2. далее, следующим параграфом после установки битов RCFG и CLNBUF регистра PSW должен быть параграф, формирующий новые значения регистра ICR

и адреса перехода для новых task-групп, который должен находиться в регистре NEWADDR. Запись этих новых значений выполняется только при установленных битах RCFG и CLNBUF в 1 регистра PSW.

```
1      ; разбиение на две группы
2      reconf:
3          getl 0x8
4          getl 0x7
5          patch @2, @2
6          patch @2, @2
7          patch @4, test_1000
8          patch @4, test_0111
9          setq #ICR, @4 ; 1000 установка в регистр ICR
                        task—группы 1, состоящий из клетки 0
10         setq #ICR, @4 ; 0111 установка в регистр ICR
                        task—группы 2, состоящий из клеток 1,2,3
11         setq #NEWADDR, @4 ; 1000 & addr установка в регис
                        тр адреса следующего параграфа для
                        task—группы 1
12         setq #NEWADDR, @4 ; 0111 & addr установка в регис
                        тр адреса следующего параграфа для
                        task—группы 2
13     complete
```

1.8.2 Композиция

Для task-групп клеток, которые необходимо собрать в одну task-группу, требуется выполнить следующие действия

1. установить значения битов RCFG и CLNBUF регистра PSW в состояние 1. RCFG — это бит динамической реконфигурации, а CLNBUF — указание на необходимость завершения выполнения всех команд следующего параграфа до выполнения перехода.

```
1      test:
2          getl #PSW
3          orl @1, 0x180 ; 7 (RCFG) и 8 (CLNBUF) биты регист
                        ра PSW
4          setl #PSW, @1
```

```
5      jmp reconf
6      complete
```

2. далее, следующим параграфом после установки битов RCFG и CLNBUF регистра PSW в task-группе, которая подключается, должен быть параграф без команды перехода

```
1      stop :
2      getl 0
3      complete
```

3. в task-группе, к которой происходит подключение, должен быть параграф с установкой новых групп, например, к группе из двух клеток 2, 3 подключается клетка 1:

```
1      reconf :
2      getl 0x7
3      patch @1, @1
4      patch @2, test_0111
5      setq #ICR, @2 ; 0111 установка в регистр ICR
                       task—группы, состоящий из клеток 1,2,3
6      setq #NEWADDR, @2 ; 0111 & addr установка в регис
                       тр адреса следующего параграфа для
                       task—группы
7      complete
```

2. Руководство пользователя по редактору связей

2.1 Общие сведения о редакторе связей

Компоновщик или редактор связей для мультиклеточного процессора осуществляет сборку образа исполняемой программы путём объединения данных одного или нескольких объектных файлов и, в случае необходимости, объектных файлов статических библиотек, а также связывание символов объектных файлов, так чтобы все ссылки на эти символы имели верные адреса времени выполнения.

Объектные файлы представляют собой результат компиляции ассемблером исходного кода программы.

Статические библиотеки представляют собой обычные архивы объектных файлов, для создания которых используется команда `ar`.

2.2 Использование редактора связей

Представленный редактор связей запускается из командной строки командой *mc-ld*, аргументами которой являются один или несколько объектных файлов. Поддерживаются также следующие опции:

-l, -library=namespec — добавить файл архива namespec к списку файлов, используемых для сборки. Данная опция может использоваться многократно. Редактор связей будет искать путь к библиотеке для имени файла libnamespec.a. Поиск архивного файла осуществляется редактором связей только один раз, в том месте, где он указан в командной строке. Если в архиве определяется символ, который был неопределённым в каком-либо объектном файле и этот объектный файл указан в командной строке до файла архива, компоновщик добавит в результирующий файл образа соответствующие объектные файлы архива. В тоже время, наличие неопределённого символа в каком-либо объектном файле, указанного в командной строке после файла архива, не приведёт к повторному поиску символа в ранее указанном архивном файле. Любой архивный файл может быть указан в командной строке несколько раз.

-L, -library-path=searchdir — добавить путь searchdir в список путей, в которых редактор связей будет искать статические библиотеки (архивные файлы, задаваемые опцией -l). Данная опция может использоваться многократно. Директории используются в том порядке, в котором они указаны в командной строке. Все значения опции -L применяются ко всем значениям опции -l, независимо от того, в каком порядке эти опции указаны.

-o, -output=FILE — поместить вывод в файл образа FILE. Если данная опция не используется, то имя выходного файла по умолчанию — image.bin.

-T, -script=FILE — прочитать файл скрипта управления редактором связей.

-M, -print-map — вывести в стандартный поток вывода информацию о размещении данных объектных файлов в памяти и значениях, назначенных символам.

-h, -help — показать это сообщение и выйти.

2.3 Скрипты

Скрипты редактора связей предназначены для управления процессом сборки образа исполняемой программы из одного или нескольких объектных файлов.

Скрипт представляет из себя набор команд. Каждая команда является либо ключевым словом, за которым могут следовать аргументы, либо присваиванием значения символу. Команды разделяются символом точки с запятой (;). Пробельные символы игнорируются.

Строки, состоящие из символов латинского алфавита ([a..zA..Z]), точки (.), подчёркивания (_), десятичных цифр ([0..9]), и не начинающиеся с цифры, могут быть заданы без заключения их в двойные кавычки ("). Все остальные строки должны быть заключены в двойные кавычки.

В скрипте можно использовать следующие типы комментариев:

- однострочный комментарий

данный тип комментария начинается с '/' и заканчивается концом строки

- многострочный комментарий

данный тип комментария начинается с '/*' и заканчивается '*/', т. е. все, что находится между этими ограничителями игнорируется; вложенные комментарии не допускаются.

В основном скрипты редактора связей используются для описания расположения секций входных объектных файлов в секциях выходного файла, а также распределения выходных секций в памяти целевой машины.

Для передачи редактору связей пользовательского скрипта необходимо использовать опцию командной строки '-T'. Если скрипт не был предоставлен пользователем, используется скрипт по умолчанию.

2.3.1 Выражения

Синтаксис выражений, используемых в скриптах редактора связей, идентичен синтаксису выражений языка C. В выражениях могут быть использованы только целочисленные константы, ссылки на глобальные символы, а также встроенные функции.

2.3.1.1 Константы

Все константы целые.

Также, как и в C, редактор связей рассматривает целые числа, начинающиеся с '0', как восьмеричные, и целые, начинающиеся с '0x' или '0X', как шестнадцатеричные. Также допускается использование суффиксов 'K' и 'M' для умножения констант на 1024 или 1024*1024 соответственно. Например, в следующем примере определяется одно и тоже значение:

```
fourk1 = 4K;
```

```
fourk2 = 4096;
```

```
fourk3 = 0x1000.
```

2.3.1.2 Имена символов

Без двойных кавычек имя символа может начинаться с буквенной литеры, подчёркивания, точки и содержать буквенно-цифровые символы, подчёркивание, точку. Имена символов не должны конфликтовать с ключевыми словами. Возможно определить символ, имя которого будет содержать недопустимые литеры, заключив его в двойные кавычки. Например, "with a space".

2.3.1.3 Счётчик текущего местоположения

Специальная переменная точка '.', которая может быть использована в выражениях, всегда содержит значение текущего местоположения выходной секции. Символ '.' можно использовать в выражении точно также, как и обычный символ.

Присваивание значения символу '.' вызовет перемещение счётчика текущего местоположения. Это может быть использовано для создания пропусков в выходной секции. Счётчик текущего местоположения не может быть уменьшен как внутри выходной секции, а также за её пределами, если это вызовет перекрытие згрузочных адресов (LMAs).

```
SECTIONS
```

```
{
```

```
    output : {
```

```
        file1(.text)
```

```
        . = . + 1000;
```

```
        file2(.text)
```

```
        . += 100;
        file3(.text)
    } = 0xFFAAFFAA;
}
```

В данном примере секция '.text' из файла file1 размещается в начале выходной секции 'output'. Далее следует пропуск размером 1000 байтов. Затем секция '.text' из файла file2, за которой также следует пропуск размером 100 байтов перед секцией '.text' из файла file3. Обозначение '= 0xFFAAFFAA' определяет записываемые в пропуски данные.

Замечание: в действительности '.' ссылается на байтовое смещение внутри текущего объекта, в котором содержатся данные. Обычно это утверждение (statement) `SECTIONS`, чей начальный адрес 0, следовательно '.' может быть использована как абсолютный адрес. Если, тем не менее, '.' используется внутри описания секции, она ссылается на байтовое смещение от начала этой секции, а не на абсолютный адрес. Таким образом, в скрипте подобному ниже:

```
SECTIONS
{
    . = 0x100;
    .text: {
        *(.text)
        . = 0x200;
    }
    . = 0x1000;
    .data: {
        *(.data)
        . += 0x400;
    }
}
```

Стартовый адрес секции '.text' будет приравнен значению 0x100, а размер будет в точности соответствовать 0x200 байтам, даже, если окажется недостаточно данных во входных секциях '.text' для заполнения указанного пространства. Если данных окажется слишком много, то это вызовет ошибку, так как будет осуществлена попытка уменьшить текущее значение счётчика текущего местоположения.

Стартовый адрес секции '.data' будет приравнен значению 0x1000, а также будет иметь дополнительных 0x400 байтов пространства после всех данных из входных секций '.data'

и перед концом самой выходной секции `‘.data’`.

2.3.1.4 Операторы

Редактор связей распознаёт стандартный набор арифметических операторов языка C, со стандартными уровнями приоритета и связываниями:

Поддерживаемые операции в порядке снижения приоритета

Лексемы	Операции	Приоритет
<i>symbol</i>	символ	13
<i>func(...)</i>	вызов функции	12
<i>unary +</i>	унарный плюс	11
<i>unary −</i>	унарный минус	11
<i>unary !</i>	логическое НЕ	11
<i>unary ~</i>	побитовое НЕ	11
<i>*, /, %</i>	мультипликативные операции	10
<i>+, −</i>	аддитивные операции	9
<i><<, >></i>	сдвиг влево и вправо	8
<i><, >, <=, >=</i>	отношения	7
<i>==, !=</i>	равенство/неравенство	6
<i>&</i>	побитовое И	5
<i>^</i>	побитовое исключающее ИЛИ	4
<i> </i>	побитовое ИЛИ	3
<i>&&</i>	логическое И	2
<i> </i>	логическое ИЛИ	1
<i>? :</i>	условная операция	0

2.3.1.5 Оценка и вычисление выражения

Редактор связей оценивает выражения ленивым способом: он вычисляет значение выражения только тогда, когда это необходимо.

Редактору связей нужна некоторая информация: значение стартового адреса первой

секции, начало и размер регионов памяти для того, чтобы в принципе осуществлять связывание. Эти значения вычисляются как можно раньше.

Однако, другие значения (например, значения символов) не известны или не нужны до момента распределения памяти. Такие значения оцениваются позже, когда другая информация (например, размеры выходных секций) станет доступна при присваивании символам значения выражения.

Размеры секций не могут быть известны до момента распределения памяти, поэтому присваивания, зависящие от распределения памяти, не выполняются.

Некоторые выражения, которые зависят от значения счётчика текущего местоположения '.', должны вычисляться во время распределения секций в памяти.

Если результат выражения необходим, но значение не доступно, тогда генерируется ошибка. Например, следующий скрипт

```
SECTIONS
{
    .text 9 + non_constant_symbol : { *(.text) }
}
```

вызовет сообщение об ошибке 'non constant expression'.

2.3.1.6 Секция результата вычисления выражения

Адреса и символы могут быть абсолютными, либо определены в какой-либо секции. Символы, связанные с секцией, являются перемещаемыми. С другой стороны, абсолютный символ всегда будет иметь одно и то же значение на протяжении всех последующих операций связывания.

Некоторые члены выражения являются адресами. Это справедливо для символов, значение которых определено в какой-либо секции, а также для встроенных функций, которые возвращают адрес, таких как ADDR, LOADADDR, ORIGIN и SEGMENT_START. Другие члены выражения являются просто числами или встроенными функциями, результат которых не является адресом, например, LENGTH. В выражениях, которые используются за пределами описания выходной секции, все числа трактуются как абсолютные адреса. В выражениях, которые используются внутри описания выходной секции, все символы с абсолютным значением трактуются как числа.

В следующем примере

SECTIONS

```
{
  . = 0x100;
  __executable_start = 0x100;
  .data :
  {
    . = 0x10;
    __data_start = 0x10;
    *(.data)
  }
}
```

в первых двух присваиваниях `.` и `__executable_start` устанавливаются равными абсолютному адресу `0x100`, затем в следующих двух присваиваниях `.` и `__data_start` устанавливаются равными `0x10` относительно секции `.data`.

Для оценки членов выражения, содержащих числа, относительные и абсолютные адреса, редактор связей использует следующие правила:

- для унарных операций над относительным адресом и бинарных операций над двумя относительными адресами в одной и той же секции или между относительным адресом и числом, оператор применяется к смещению части адреса(ов);
- для унарных операций над абсолютным адресом, бинарных операций над одним или несколькими абсолютными адресами или над двумя относительными адресами из различных секций, перед применением оператора осуществляется конвертация неабсолютного члена к абсолютному адресу.

Секция результата каждого подвыражения определяется следующим образом:

- результатом операции только над числами будет число;
- результатом операций сравнения `'&&'` и `'||'` будет число;
- результатом любой другой бинарной арифметической или логической операции над двумя относительными адресами в одной и той же секции или над двумя абсолютными адресами (после конвертирования) будет число;
- результатом других операций над относительными адресами или над одним относительным адресом и числом, будет относительный адрес в той же самой секции, что и секция операнда(ов);
- результатом других операций над абсолютными адресами (после конвертирова-

ния) будет абсолютный адрес.

Для принудительного приведения результата выражения, который является относительным адресом, к абсолютному адресу необходимо использовать встроенную функцию `ABSOLUTE`. Например, чтобы создать абсолютный символ, значение которого равно адресу конца выходной секции `‘.data’`, необходимо написать в скрипте:

```
SECTIONS
{
    .data : { *(.data) _edata = ABSOLUTE(.); }
}
```

Если встроенная функция `‘ABSOLUTE’` не была использована, то значение `‘_edata’` было бы относительным адресом в секции `‘.data’`.

Использование встроенной функции `LOADADDR` также преобразует относительный адрес в абсолютный.

2.3.1.7 Встроенные функции

В выражениях скрипта редактора связей могут быть использованы встроенные функции.

2.3.1.7.1 `ABSOLUTE(expr)`

Возвращает абсолютное (неперемещаемое) значение выражения `expr`. В основном используется для присваивания абсолютного значения символу внутри описания секции, где значения символов, как правило, являются относительными.

2.3.1.7.2 `ADDR(section)`

Возвращает адрес (VMA) именованной секции. Местоположение данной секции должно быть ранее определено в скрипте. В следующем примере символам `start_of_output_1`, `symbol_1` и `symbol_2` присваиваются эквивалентные значения, за исключением того, что символ `symbol_1` будет относительным в секции `.output1`, а два других абсолютными:

```
SECTIONS
{
```

```
.output1 :  
{  
    start_of_output_1 = ABSOLUTE(.);  
    ...  
}  
.output :  
{  
    symbol_1 = ADDR(.output1);  
    symbol_2 = start_of_output_1;  
}  
}
```

2.3.1.7.3 ALIGN(align) / ALIGN(expr, align)

Возвращает значение счётчика текущего местоположения (.) или значение результата произвольного выражения, выравненное на следующую границу align. Одно операндный ALIGN не изменяет текущее значение счётчика текущего местоположения, а только выполняет арифметическую операцию и возвращает результат. Двух операндный ALIGN выполняет арифметическую операцию над результатом произвольного выражения expr и возвращает результат. (ALIGN(align) является эквивалентом ALIGN(., align)).

Далее следует пример, в котором адрес выходной секции .data устанавливается равным текущему значению счётчика текущего местоположения выровненным по границе 0x2000 байтов после предыдущей секции, а также устанавливается значение переменной A внутри секции равным значению выравнивания по границе 0x8000 байтов после всех входных секций:

```
SECTIONS  
{  
    ...  
    .data ALIGN(0x2000) :  
    {  
        *(.data)  
        A = ALIGN(0x8000);  
    }  
    ...  
}
```

Первое использование встроенной функции ALIGN в этом примере определяет местоположение секции, так как результат данной функции используется в качестве необязательного адресного атрибута при описании выходной секции. Следующее использование встроенной функции ALIGN в этом примере определяет значение символа A.

2.3.1.7.4 ALIGNOF(section)

Возвращает значение выравнивания в байтах именованной секции, если секция была распределена в памяти. Если секция не была распределена в памяти на момент выполнения данной функции, тогда генерируется сообщение об ошибке. В следующем примере, значение выравнивания выходной секции .output сохраняется в самой секции в качестве первого значения.

```
SECTIONS
{
    ...
    .output
    {
        LONG(ALIGNOF(.output))
        ...
    }
    ...
}
```

2.3.1.7.5 ASSERT(expr, msg)

Возвращает результат вычисления выражения expr, а также выводит сообщение об ошибке с текстом msg, если результат нулевой.

2.3.1.7.6 DEFINED(symbol)

Возвращает 1, если символ находится в глобальной таблице символов редактора связей и определён в скрипте до использования данной функции, иначе возвращает 0. Данную функцию можно использовать для установки значения символа по умолчанию.

Следующий пример показывает, как установить значение глобального символа 'start', равным начальному адресу секции '.text', если данный символ ещё не определён:

SECTIONS

```
{  
  ...  
  .text :  
  {  
    start = DEFINED(begin) ? begin : . ;  
    ...  
  }  
  ...  
}
```

2.3.1.7.7 LENGTH(memory)

Возвращает длину (размер) именованного региона памяти memory.

2.3.1.7.8 LOADADDR(section)

Возвращает абсолютный загрузочный адрес (LMA) именованной секции.

2.3.1.7.9 MAX(exp1, exp2)

Возвращает максимум из exp1 и exp2.

2.3.1.7.10 MIN(exp1, exp2)

Возвращает минимум из exp1 и exp2.

2.3.1.7.11 ORIGIN(memory)

Возвращает стартовый адрес именованного региона памяти memory.

2.3.1.7.12 SEGMENT_START(segment, default)

Возвращает базовый адрес именованного сегмента. Если значение для сегмента было указано явно, используя опцию командной строки '-T', будет возвращено именно это

значение, иначе будет возвращено значение по умолчанию.

2.3.1.7.13 SIZEOF(section)

Возвращает размер в байтах именованной секции, если секция была размещена. Если при выполнении данной функции секция не была размещена, редактор связей сгенерирует ошибку.

В следующем примере значения символов `symbol_1` и `symbol_2` устанавливаются равными одинаковым значениям:

```
SECTIONS
{
    ...
    .output
    {
        .begin = . ;
        ...
        .end = . ;
    }
    symbol_1 = .end - .begin;
    symbol_2 = SIZEOF(.output);
    ...
}
```

2.3.2 Операции присваивания

В скриптах допускается операция присваивания символу значения. Данная операция устанавливает значение символа и размещает его в таблице символов с глобальной областью видимости.

2.3.2.1 Простые присваивания

Для того, чтобы присвоить значение символу, необходимо использовать один из ниже перечисленных операторов:

symbol = *expression*;

symbol += *expression*;

symbol -= *expression*;

symbol *= *expression*;

symbol /= *expression*;

symbol <<= *expression*;

symbol >>= *expression*;

symbol &= *expression*;

symbol |= *expression*;

Первый оператор устанавливает значение символа, равным результату выражения. Во всех остальных случаях символ должен быть уже определён и его значение будет изменено согласно операции.

Каждая операция присваивания должна заканчиваться точкой с запятой (;).

В скриптах редактора связей символьные присваивания могут быть текстуально расположены только в трёх местах:

- вне команд, допустимых в скрипте редактора связей;
- внутри команды SECTIONS;
- внутри описания выходной секции в команде STCTIONS.

Секция, в которой определён символ, соответствует секции результата выражения.

Пример использования операции присваивания в трёх разных вариантах:

```
A = 0;
SECTIONS
{
    .text :
    {
        *(.text)
        end_of_text = .;
    }
    B = ALIGN(4);
    .data : { *(.data) }
}
```

В этом примере символ 'A' будет определён значением ноль. Значением символа 'end_of_text' будет адрес, следующий за последней входной секцией '.text'. Значением сим-

вола 'B' будет адрес, следующий за выходной секцией '.text', выравненный на 4 байта.

2.3.2.2 HIDDEN символы

С помощью команды HIDDEN определяется символ, который будет скрыт и не будет экспортирован. Синтаксис:

$$HIDDEN(symbol = expression)$$

2.3.2.3 PROVIDE символы

В некоторых случаях, желательно, чтобы символ был определён в скрипте только в том случае, если на него имеются ссылки и в тоже время он не был определён ни во одном объектном файле. Синтакс:

$$PROVIDE(symbol = expression)$$

2.3.2.4 PROVIDE_HIDDEN символы

В некоторых случаях, желательно, чтобы символ был определён в скрипте только в том случае, если на него имеются ссылки и в тоже время он не был определён ни во одном объектном файле. Синтакс:

$$PROVIDE_HIDDEN(symbol = expression)$$

Если данный символ будет определён, то он будет скрыт и не будет экспортирован.

2.3.3 Команды

В скриптах редактора связей поддерживается следующий набор команд:

- команда для установки точки входа (ENTRY);
- команды для работы с файлами (INCLUDE, INPUT, GROUP, OUTPUT, SEARCH_DIR, STARTUP);
- команды для определения регионов памяти (MEMORY, REGION_ALIAS);
- команда для описания выходных секций (SECTIONS);
- команда для определения программных сегментов (PHDRS);

- прочие команды (ASSERT, DEBUG, OUTPUT_ARCH).

2.3.3.1 ASSERT(expr, msg)

Команда ASSERT гарантирует, что результат вычисления выражения `expr` не нулевой. Если нулевой, редактор связей выводит сообщение об ошибке с текстом `msg` и завершает работу.

2.3.3.2 DEBUG(fmt_str, expr_list)

Команда DEBUG предназначена для вывода результатов вычисления выражений, определённых списком `expr_list`, согласно заданной форматной строке `fmt_str`.

Данная команда может использоваться для отладки скрипта и никак не влияет на процесс сборки образа исполняемой программы.

Форматная строка `fmt_str` представляет из себя обычную строку символов, которая может содержать, а может и не содержать директивы: обычные символы, за исключением символов '%' и '\', которые печатаются в неизменном виде, ескаре-последовательности, начинающиеся с символа '\', а также номер результата выражения, начинающийся с символа '%', который необходимо вывести на печать. Номеруция результатов выражений начинается с единицы; максимальное значение номера — 9. Между символом '%' и номером результата выражения может быть указан один из следующих флагов:

- d — напечатать результат выражения в виде целого знакового десятичного числа;
- u — напечатать результат выражения в виде целого беззнакового десятичного числа;
- o — напечатать результат выражения в восьмеричной системе счисления;
- x, X — напечатать результат выражения в шестнадцатичной системе счисления.

Поддерживаемые ескаре-последовательности:

- \a — звуковой сигнал (alert);
- \b — забой (backspace);
- \f — новая страница (formfeed);
- \n — перевод строки (newline);
- \r — возврат каретки (carriage return);

- `\t` — горизонтальная табуляция (horizontal tab);
- `\v` — вертикальная табуляция (vertical tab);
- `\\` — соответствует литералу `'\'`.

Для вывода на печать символа `'%'` в форматной строке `fmt_str` необходимо написать два символа `'%'` подряд (`'%%'`).

В случае невалидного номера результата выражения (указанный номер результата превышает количество выражений в списке `expr_list`) или ошибки в форматной строке в соответствующем месте на печать будет выведено три вопросительных знака `'???'`.

2.3.3.3 ENTRY(symbol)

Первая инструкция, с которой начинается выполнение программы называется точкой входа. Для того, чтобы установить точку входа, необходимо использовать команду `ENTRY`. Аргументом команды является имя символа `symbol`.

Так как процессор `MCp042R100102` не поддерживают старт программы с произвольного адреса, то данная команда поддерживается парсером, но игнорируется кодогенератором, о чём редактор связей выводит соответствующее предупреждение.

2.3.3.4 GROUP(file, file, ...)

Команда `GROUP` указывает редактору связей интерпретировать перечисленные файлы как имена архивов (библиотек) и включить их в процесс сборки программы, как будто эти файлы были переданы в командной строке с помощью опции командной строки `'-l'`. Другими словами, команда `GROUP` подобна команде `INPUT`, за исключением того, что все файлы должны быть архивами (библиотеками).

Архивы (библиотеки) просматриваются неоднократно до тех пор, пока не останется ни одного неопределённого символа.

2.3.3.5 INCLUDE filename

Команда `INCLUDE` предназначена для включения скрипта `filename` редактора связей в месте расположения данной команды. Поиск файла осуществляется в текущей директории, а также в любой из директорий, заданных опцией командной строки `'-L'`. Данную

директиву можно использовать на самом верхнем уровне, в командах MEMORY или SECTIONS, а также в описаниях выходной секции.

2.3.3.6 INPUT(file, file, ...)

Команда INPUT указывает редактору связей включить перечисленные файлы в процесс сборки программы, как будто эти файлы были переданы в командной строке.

Поиск файлов осуществляется в текущей директории; если файл не найден, то поиск файла будет осуществлён в директориях, заданных опцией командной строки '-L'.

Если использована команда 'INPUT (-lfile)', то имя будет преобразовано редактором связей в libfile.a, точно также, как при использовании опции командной строки '-l'.

2.3.3.7 Команда MEMORY

Конфигурация редактора связей по умолчанию позволяет распределять всю доступную память. Используя команду MEMORY, данное поведение можно изменить.

Команда MEMORY описывает местоположение и размер блоков памяти целевой машины. Её можно использовать, чтобы описать какие регионы памяти могут быть использованы редактором связей, а какие стоит избегать. После определения регионов памяти их можно назначить выходным секциям. Редактор связей установит адрес секции основываясь на регионах памяти, а также выдаст предупреждения о переполнении региона памяти. Редактор связей не будет пытаться переставлять местами секции, чтобы уместить их в доступных регионах.

Скрипт редактора связей может содержать несколько команд MEMORY, однако в конечном итоге они будут объединены в одну. Допускается определять неограниченное число регионов памяти внутри данной команды.

Синтаксис команды MEMORY:

```
MEMORY
{
  name [(attr)] : ORIGIN = origin, LENGTH = len
  ...
}
```

name — это имя, которое используется в скрипте для ссылки на данный регион памяти. Имя name региона не имеет никакого значения за пределами скрипта. Имена регио-

нов сохраняются в отдельном пространстве имён, и никогда не будут конфликтовать с именами символов, файлов или секций. Каждый регион памяти должен иметь индивидуальное имя внутри команды MEMORY. Однако позже можно добавить алиас на имя существующего региона памяти, используя команду REGION_ALIAS.

Строка attr задаёт необязательный список атрибутов, которые определяют использовать ли специфичные регионы памяти для входных секций, размещение которых явно не указано в скрипте. Если для некоторой входной секции неопределена выходная секция, редактор связей создаст выходную одноимённую секцию. Если определены атрибуты региона, редактор связей будет использовать их для выбора региона памяти, в котором будет создана выходная секция.

Строка attr должна состоять только из следующих символов:

- 'R' — read-only секция;
- 'W' — read/write секция;
- 'X' — исполняемая секция;
- 'A' — размещаемая секция;
- 'I' — инициализированная секция;
- '!' — инверсия следующего атрибута.

Если входная секция, размещение которой не определено в скрипте, соответствует любому из перечисленных атрибутов, кроме '!', то она будет размещена в этом регионе памяти. Атрибут '!' изменяет проверку на противоположную, поэтому секция будет размещена в данном регионе только, если она не соответствует ни одному из перечисленных атрибутов.

origin — это числовое выражение, результат вычисления которого является начальным адресом региона памяти. Результат вычисления выражения должен быть константным, а само выражение не должно содержать никаких символов. Ключевое слово ORIGIN может быть сокращено до 'org' или 'o' (но не, например, ORG).

len — это числовое выражение, результат вычисления которого определяет размер в байтах региона памяти. Результат вычисления выражения должен быть константным, а само выражение не должно содержать никаких символов. Ключевое слово LENGTH может быть сокращено до 'len' или 'l'.

В следующем примере определяется два региона памяти, доступных для размещения: первый начинается с адреса 0 и имеет размер 128 килобайт, а второй начинается с адреса

0x400000 и имеет размер 256 килобайт. Редактор связей поместит в регион памяти `rom` каждую секцию, размещение которой явно не указано в скрипте и которая либо доступна только для чтения, либо исполняемая. Остальные секции редактор связей поместит в регион памяти `ram`.

```
MEMORY
{
    rom (rx)  : ORIGIN = 0, LENGTH = 128K
    ram (!rx) : org = 0x400000, l = 256K
}
```

Как только регион памяти определён, можно указать редактору связей разместить специфичные выходные секции в этом регионе, используя `'>region'`. Например, если определён регион памяти с именем `mem_reg`, необходимо использовать `'>mem_reg'` в описании выходной секции. Если для выходной секции неопределён адрес, редактор связей установит его значение равным первому доступному адресу в регионе памяти. Если размер всех включённых в регион памяти выходных секций превышает размер данного региона памяти, редактор связей выдаст сообщение об ошибке.

Для того, чтобы получить начальный адрес и размер региона памяти, в выражении необходимо использовать функции *ORIGIN(memory)* и *LENGTH(memory)* соответственно, например, `addr_of_ram = ORIGIN(ram);`.

2.3.3.8 OUTPUT(filename)

Команда `OUTPUT` определяет имя выходного файла. Использование `OUTPUT(filename)` в скрипте в точности совпадает с использованием опции командной строки `'-o filename'`. Если используются оба варианта, предпочтение отдаётся опции командной строки.

2.3.3.9 OUTPUT_ARCH(arch)

Команда `OUTPUT_ARCH` определяет архитектуру, для которой осуществляется компоновка программы и является аналогом опции командной строки `'--arch'`.

Допустимыми аргументами данной команды для мультиклеточных процессоров являются следующие значения: `MCp0411100101`, `MCp042R100102`.

Если используется как опция командной строки `'--arch'`, так и команда `OUTPUT_ARCH`, предпочтение отдаётся опции командной строки.

2.3.3.10 Команда PHDRS

Формат ELF объектного файла использует программные заголовки, которые описывают так называемые сегменты, а именно как следует размещать программу (сегменты) в памяти.

При загрузке ELF программы загрузчик читает программные заголовки для того, чтобы выяснить, как загружать программу. Это работает только в том случае, когда программные заголовки выставлены корректно.

Редактор связей создаёт обоснованные программные заголовки по умолчанию. Однако, в некоторых случаях, может возникнуть необходимость определить программные заголовки более точно. В этом случае необходимо использовать команду PHDRS. При обнаружении команды PHDRS в скрипте, редактор связей не будет создавать никакие другие программные заголовки, кроме явно определённых.

Синтаксис команды PHDRS.

```
PHDRS
{
    name type [ FILEHDR ] [ PHDRS ] [ AT ( address ) ] [ FLAGS ( flags ) ] ;
}
```

PHDRS, FILEHDR, AT, FLAGS являются ключевыми словами.

name используется только для ссылок внутри команды SECTIONS в скрипте и никак не проявляется в выходном файле. Имена программных заголовков сохраняются в отдельном пространстве имён и никак не конфликтуют с именами символов, файлов и секций. Каждый программный заголовок должен иметь уникальное имя. Заголовки обрабатываются в порядке возрастания загрузочного адреса сопоставляемых секций.

Некоторые типы программных заголовков описывают сегменты памяти, которые загрузчик будет загружать из файла. В скрипте определяется содержимое данных сегментов путём размещения распределяемых (allocatable) выходных секций в сегментах. Для размещения секции в специфическом сегменте можно использовать атрибут ':phdr' выходной секции.

Допустимо размещать некоторые секции в нескольких сегментах. Это просто означает, что один сегмент памяти содержит другой. Можно однократно использовать ':phdr' для каждого сегмента, который должен содержать секцию.

Если разместить секцию более, чем в одном сегменте, используя ':phdr', то редактор связей разместит все последующие распределяемые секции, для которых не определён

'phdr', в том же самом сегменте. Это сделано для удобства, так как обычно весь набор соприкасающихся секций будет размещаться в одном сегменте.

Ключевые слова FILEHDR и PHDRS после программного заголовка поддерживаются парсером, но игнорируются кодогенератором.

Тип сегмента может принимать следующие значения (номер указывает значение ключевого слова).

PT_NULL (0) — неиспользуемый программный заголовок;

PT_LOAD (1) — данный сегмент должен быть загружен из файла;

PT_DYNAMIC (2) — в данном сегменте следует искать информацию о динамическом связывании (не поддерживается);

PT_INTERP (3) — в данном сегменте следует искать информацию об имени интерпретатора программы;

PT_NOTE (4) — в данном сегменте содержатся примечания;

PT_SHLIB (5) — зарезервированный тип программного заголовка;

PT_PHDR (6) — в данном сегменте содержится таблица программных заголовков;

expression — результатом вычисления выражения является числовое значение программного заголовка. Может быть использовано для типов не описанных выше.

Используя 'AT expression', можно определить конкретный адрес, по которому сегмент должен быть загружен в память. Аналогично атрибуту AT, который используется в описании выходной секции. Команда AT программного заголовка переписывает значение атрибута выходной секции.

Обычно редактор связей устанавливает флаги сегментов, основываясь на флагах секций, которые являются содержимым данного сегмента. Для явного определения флагов сегмента, можно использовать ключевое слово FLAGS. Значение флага должно быть целым числом. Оно используется в качестве значения поля p_flags программного заголовка.

Пример.

```
PHDRS
{
    text PT_LOAD;
    data PT_LOAD;
```

```
}
```

```
SECTIONS
```

```
{  
  .text : { *(.text) } :text  
  .data : { *(.data) } :data  
}
```

2.3.3.11 REGION_ALIAS(alias, region)

Команда REGION_ALIAS предназначена для добавления алиасов на существующие регионы памяти, которые ранее были созданы командой MEMORY. Каждый алиас может соответствовать только одному региону памяти.

Команда REGION_ALIAS создаёт алиас alias для региона памяти region, что позволяет гибко распределять выходные секции в регионах памяти.

2.3.3.12 SEARCH_DIR(path)

Команда SEARCH_DIR добавляет путь к списку путей, где редактор связей осуществляет поиск библиотек. Использование SEARCH_DIR(path) в точности совпадает с использованием опции командной строки '-L path'. Если используются оба варианта, тогда и редактор связей использует оба пути. Пути, заданные с использованием опции командной строки, используются первыми.

2.3.3.13 Команда SECTIONS

Команда SECTIONS указывает, как редактору связей осуществлять расположение входных секций в выходных, и как разместить выходные секции в памяти.

Формат команды SECTIONS:

```
SECTIONS  
{  
  sections-command  
  sections-command  
  ...  
}
```

Каждая sections-command может быть:

- командами ASSERT, DEBUG, ENTRY;
- символьным присваиванием;
- описанием выходной секции.

Команды ASSERT, DEBUG, ENTRY и символьное присваивание допускается использовать внутри рассматриваемой команды для удобства использования счётчика местоположения. Это облегчает понимание скрипта, потому что данные команды могут быть использованы в значимых точках при описании макета выходного файла.

Если в скрипте не используется команда SECTIONS, редактор связей разместит каждую входную секцию в одноимённой выходной секции в порядке их размещения во входных файлах. Если, например, все входные секции представлены в первом входном файле, порядок секций в выходном файле будет соответствовать порядку их следования в первом входном файле. Первая секция будет размещена с нулевого адреса.

Синтаксис описания выходной секции:

```
section_name [address] [(type)] :  
    [AT(lma)]  
    [ALIGN(section_align)]  
    [SUBALIGN(subsection_align)]  
    {  
        output-section-command  
        output-section-command  
        ...  
    } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

Двоеточие и круглые скобки в описании выходной секции являются обязательными. Символы перевода строки и остальные пробелы опциональны.

Каждая из output-section-command может быть:

- символьным присваиванием;
- описанием входной секции;
- данными, которые включаются непосредственно с помощью специальных команд.

2.3.3.13.1 Атрибуты выходной секции

Ниже описаны атрибуты выходной секции, которые могут быть определены при описании выходной секции.

Как правило, большинство из опциональных атрибутов (указаны в квадратных скобках []) не используется.

2.3.3.13.1.1 Имя выходной секции. `section_name` — имя выходной секции. Если имя выходной секции `section_name` содержит недопустимые символы, его необходимо заключить в двойные кавычки.

2.3.3.13.1.2 Виртуальный адрес (VMA) выходной секции. `address` — выражение, результат вычисления которого является адресом (VMA) выходной секции.

Выражение, определяющее адрес, опционально, а в случае его явного указания адрес выходной секции будет в точности ему соответствовать (без учёта значений выравнивания входных секций, содержащихся в выходной секции).

Если выходной адрес не определён, тогда адрес будет вычислен согласно следующему алгоритму:

- если для секции определён выходной регион памяти, то секция будет добавлена в этот регион и её адрес будет равен первому свободному адресу данного региона памяти;
- если была использована команда MEMORY для создания списка регионов памяти, тогда будет выбран первый регион из списка, атрибуты которого совместимы с атрибутами выбранной секции; выходной адрес секции будет равен следующему свободному адресу в регионе;
- если регионы памяти не были определены, либо ни один из определённых регионов памяти не является подходящим для размещения секции, то выходной адрес секции будет вычислен на основе текущего значения счётчика местоположения.

Вычисленный адрес также будет выровнен согласно атрибуту выравнивания выбранной выходной секции. Значение выравнивания выходной секции будет самым строгим (максимальным) среди всех значений выравнивания входных секций, содержащихся в выходной секции.

Определение адреса выходной секции вызывает изменение значения счётчика местоположения для непустых секций (пустые секции игнорируются).

2.3.3.13.1.3 Тип выходной секции. Каждая выходная секция может иметь тип. Тип — это ключевое слово в круглых скобках `{{(type)}}`. Допустимо использование следующих типов: NOLOAD (секция должна быть помечена как незагружаемая, т. е. она не будет загружаться в память при запуске программы).

Данный атрибут поддерживается парсером редактора связей, однако его значение игнорируется.

2.3.3.13.1.4 Адрес загрузки (LMA) выходной секции. Каждая секция имеет виртуальный адрес (VMA) и загрузочный адрес (LMA). Виртуальный адрес описан выше. Загрузочный адрес определяется ключевым словом 'AT' или 'AT>'. Определение загрузочного адреса опционально.

Ключевое слово 'AT' в качестве аргумента требует выражение, результат вычисления которого точно определяет загрузочный адрес секции. Ключевое слово 'AT>' в качестве аргумента требует имя региона памяти, при этом загрузочный адрес секции устанавливается равным следующему свободному адресу региона, выровненному, в случае необходимости, согласно значению выравнивания секции.

Если ни одно из ключевых слов 'AT' или 'AT>' не задано для размещаемой секции, редактор связей будет использовать следующую процедуру определения загрузочного адреса секции:

- если задан конкретный виртуальный адрес секции (VMA), тогда он используется в качестве загрузочного адреса (LMA);
- если для секции определён выходной регион памяти, тогда загрузочный адрес (LMA) равен виртуальному адресу (VMA);
- если секция не является размещаемой, тогда загрузочный адрес (LMA) равен виртуальному адресу (VMA);
- если найден регион памяти, подходящий для размещения текущей секции, и этот регион уже содержит хотя бы одну секцию, тогда загрузочный адрес (LMA) устанавливается таким образом, чтобы разность между VMA и LMA совпадала с разностью между VMA и LMA последней размещённой секции в данном регионе;

- если не было объявлено ни одного региона памяти, тогда используется регион памяти, который охватывает всё адресное пространство, а загрузочный адрес (LMA) устанавливается согласно алгоритму, описанному на предыдущем шаге;
- если подходящий для размещения секции регион памяти не был найден или не было предыдущей секции в найденном регионе, тогда LMA устанавливается равным VMA.

2.3.3.13.1.5 Принудительное выравнивание выходной секции. Для определения минимального значения выравнивания выходной секции необходимо использовать атрибут ALIGN, аргументом которого является выражение. Результат вычисления выражения должен быть неотрицательным целым числом кратным степени 2, за исключением значения '-1', которое отменяет выравнивание.

2.3.3.13.1.6 Принудительное выравнивание входной секции. Для определения минимального значения выравнивания входной секции внутри выходной секции необходимо использовать атрибут SUBALIGN, аргументом которого является выражение. Результат вычисления выражения должен быть неотрицательным целым числом кратным степени 2, за исключением значения '-1', которое отменяет выравнивание.

2.3.3.13.1.7 Регион памяти выходной секции. Для того, чтобы назначить секции ранее определённый регион памяти необходимо использовать '>region'. Например:

```
MEMORY
{
    rom : ORIGIN = 0x1000, LENGTH = 0x1000
}

SECTIONS
{
    ROM : { *(.text) } >rom
}
```

2.3.3.13.1.8 Программный заголовок (сегмент) выходной секции. Для того, чтобы назначить секцию ранее определённому программному сегменту необходимо использовать ':phdr'. Если секция назначена одному или более сегменту, то все последу-

ющие размещённые секции будут назначены также тем сегментам, пока не будет явного указания модификатора `':phdr'`. Пример:

```
PHDRS
{
    text PT_LOAD;
}

SECTIONS
{
    .text : { *(.text) } :text
}
```

2.3.3.13.1.9 Заполнитель выходной секции. Для того, чтобы установить шаблон заполнителя для всей секции, необходимо использовать `'=fillexp'`. `fillexp` является выражением. Любые неопределённые регионы памяти внутри выходной секции будут заполнены этим значением.

Последовательность байт в числе, которым является результат вычисления выражения `fillexp`, — `big-endian`.

Шаблон заполнителя можно изменить, используя команду `FILL` внутри команд выходной секции. Пример:

```
SECTIONS
{
    .text : { *(.text) } =0x90909090
}
```

2.3.3.13.2 Описание входных секций

Наиболее часто используемая команда выходной секции — это команда описания входной секции.

Описание входной секции является самой базовой командой скрипта редактора связей. Описания выходных секций используются для указания редактору связей карты расположения программы в памяти. Описания входных секций используются для указания редактору связей карты расположения входных секций в выходных секциях.

2.3.3.13.2.1 Базовое описание входной секции. Описание входной секции состоит из имени файла, за которым опционально следует список имён входных секций в круглых скобках.

Имя файла и имя секции может быть задано в виде шаблона имён, которые будут описаны позже.

Наиболее общее описание входной секции указывает редактору связей на включение всех входных секций с определённым именем в (одноимённую) выходную секцию. Например, для того, чтобы включить все входные секции `'.text'`, необходимо написать:

```
*(.text)
```

Здесь символ `'*'` является шаблоном, который соответствует любому имени файла. Для того, чтобы исключить список файлов из соответствия имени файла шаблону можно использовать команду `EXCLUDE_FILE`. Например:

```
*(EXCLUDE_FILE (file1.o *_xx.o) .data)
```

означает включить все секции `.data` из всех файлов, за исключением секций `.data` из файлов `file1.o` и `*_xx.o` (`*` — шаблон, соответствующий любой последовательности символов).

Существует два способа включить более, чем одну входную секцию в выходную:

```
*(.text .rdata)
```

```
*(.text) *(.rdata)
```

Разница между двумя представленными вариантами в порядке включения входных секций `'.text'` и `'.rdata'` в выходную секцию. В первом случае, они будут смешаны, т. е. расположены в выходной секции в порядке их появления на входе редактора связей. Во втором случае, сначала будут расположены все входные секции `'.text'`, за которыми будут следовать все входные секции `'.rdata'`.

Для того, чтобы включить входные секции из определённых файлов, необходимо явно указать их имена. Например:

```
data.o(.data)
```

Для того, чтобы включать входные секции на основе их флагов, необходимо использовать команду `INPUT_SECTION_FLAGS`. Например:

```
SECTIONS
```

```
{
```

```
    .text1 : { INPUT_SECTION_FLAGS(SHF_MERGE & SHF_STRINGS) *(.text) }
```



```
.text2 : { INPUT_SECTION_FLAGS(!SHF_WRITE) *(.text) }  
}
```

В этом примере, выходная секция `.text1` будет содержать любую из одноимённых входных секций, флаги `SHF_MERGE` и `SHF_STRINGS` которых будут установлены. Выходная секция `.text2` будет содержать любую из одноимённых входных секций, флаг `SHF_WRITE` которых будет сброшен.

Для указания файлов внутри архива необходимо использовать шаблон имени архива, затем символ `':'`, а затем шаблон имени файла, БЕЗ пробельных символов вокруг символа `':'` (при этом полученная строка должна быть заключена в двойные кавычки):

- `"archive:file"` — соответствует файлу `file` внутри архива `libarchive.a`;
- `"archive:"` — соответствует всем файлам внутри архива `libarchive.a`;
- `":file"` — соответствует файлу `file`, который не найден ни в одном из архивов. Другими словами, файл `file` НЕ должен быть частью архива, т. е. это обычный объектный файл.
- `"file"` — соответствует файлу, который может быть как частью любого архива, так и обычным объектным файлом.

Как имя архива, так и имя файла, может содержать шаблон имени. `"archive:file"` может быть использован внутри списка команды `EXCLUDE_FILE`, но не может присутствовать в других контекстах скрипта редактора связей. Например, недопустимо извлекать файл из архива, используя `"archive:file"` внутри команды `INPUT`.

В случае использования имени файла БЕЗ списка секций, все секции входного файла будут включены в выходную секцию. Например:

```
data.o
```

2.3.3.13.2.2 Шаблоны подстановки для входных секций В описании входной секции, как в имени файла, так и в имени секции, можно использовать шаблоны подстановки. Так, например, имя файла `''` означает любое имя файла.

Шаблоны подстановки:

`''` — соответствует любому количеству любых символов (ноль и более);

`'?'` — соответствует любому одному символу;

`'[chars]'` — соответствует одному символу из набора символов, символ `'-'` используется для задания диапазона символов, например, `'[a-z]'` соответствует любому

латинскому символу в нижнем регистре;

'\' — экранирует следующий символ.

Когда имя файла соответствует шаблону, символы подстановки не будут соответствовать символу '/' (используется для разделения имён директорий в Unix). Шаблон, состоящий из одного символа '*' является исключением; он всегда соответствует любому имени файла, вне зависимости от того, содержит имя файла символ '/' или нет. В имени секции символы подстановки будут соответствовать символу '/'.

Шаблоны имени файла применяются только к файлам, которые явно определены в командной строке или в команде INPUT. Редактор связей не осуществляет поиск директорий для расширения шаблонов.

Если имя файла соответствует более, чем одному шаблону, или имя файла задано явно и при этом также соответствует какому-либо шаблону, то редактор связей будет использовать первое совпадение в скрипте. Например, последовательность описаний входных секций, приведённая ниже, возможно является ошибочной, так как правило data.o никогда не будет использовано:

```
.data : { *(.data) }  
.data1 : { data.o(.data) }
```

Следующий пример показывает, как шаблоны подстановки могут быть использованы для разделения файлов. Скрипт указывает редактору связей разместить все входные секции '.text' в выходной секции '.text' и все входные секции '.bss' в выходной секции '.bss'. Редактор связей разместит входные секции '.data' из входных файлов, имя которых начинается с заглавной буквы, в выходной секции '.DATA'; для всех других файлов, редактор связей разместит входные секции '.data' в выходной секции '.data'.

```
SECTIONS  
{  
  .text : { *(.text) }  
  .DATA : { [A-Z]*(.data) }  
  .data : { *(.data) }  
  .bss : { *(.bss) }  
}
```

2.3.3.13.2.3 Входная секция для общих символов. Общие символы во входных файлах будут размещены в секции '.bss' выходного файла.

2.3.3.13.2.4 Пример описания входной секции. Следующий скрипт указывает редактору связей прочитать все секции из входного файла `all.o` и разместить их в начале выходной секции `'outputA'`, которая начинается с адреса `0x10000`. Все секции `'input1'` из входного файла `file1.o` разместить непосредственно за всеми секциями входного файла `all.o` в той же самой выходной секции `'outputA'`. Все секции `'input2'` из входного файла `file1.o` размещаются в выходной секции `'outputB'`, за которыми сразу следуют все входные секции `'input1'` из входного файла `file2.o`. Все остальные входные секции `'input1'` и `'input2'` из любых других входных файлов размещаются в выходной секции `'outputC'`.

```
SECTIONS
{
    outputA 0x10000 :
    {
        all.o
        file1.o (.input1)
    }
    outputB :
    {
        file1.o (.input2)
        file2.o (.input1)
    }
    outputC :
    {
        *(.input1)
        *(.input2)
    }
}
```

2.3.3.13.3 Данные выходной секции, непосредственно включаемые с помощью специальных команд

В выходную секцию можно явно включить байты данных, используя команды выходной секции `BYTE`, `SHORT`, `LONG`, `QUAD`. За каждым ключевым словом следует выражение в круглых скобках, результат вычисления которого и будет сохранён в выходной секции. Значение сохраняется по текущему местоположению счётчика текущего местоположения.

Команды BYTE, SHORT, LONG и QUAD записывают в выходную секцию соответственно один, два, четыре и восемь байтов. После записи байтов, счётчик текущего местоположения увеличивается на соответствующее количество.

Например, ниже следующие команды приведут к последовательной записи в выходную секцию одного байта, равным значению 0x01, и четырём байтов, равных значению символа addr:

```
BYTE(1)
LONG(addr)
```

Порядок следования байт при выполнении команд BYTE, SHORT, LONG, QUAD определяется целевой архитектурой.

Описанные выше команды могут быть использованы только внутри описания выходной секции, поэтому следующий пример вызовет ошибку:

```
SECTIONS
{
    .text : { *(.text) }
    LONG(1)
    .data : { *(.data) }
}
```

в то время как следующий будет правильно отработан:

```
SECTIONS
{
    .text : { *(.text) LONG(1) }
    .data : { *(.data) }
}
```

Для определения шаблона заполнителя для текущей выходной секции необходимо использовать команду FILL, за которой следует выражение в круглых скобках. Любые неопределённые регионы памяти внутри данной выходной секции (например, пропуски, вызванные выравниванием входных секций) заполняются значением выражения, которое повторяется необходимое количество раз. Допускается использование различных шаблонов заполнителя в различных частях выходной секции.

Пример заполнения неопределённых промежутков значением 0x90:

```
FILL(0x90909090)
```

Команда FILL схожа с атрибутом '=fillexp' описания выходной секции, за исключением того, что она применяется к части секции, которая следует за ней ('=fillexp' применяется ко всей секции). Если одновременно используются два варианта, предпочтение отдаётся команде FILL.

2.3.3.13.4 Отбрасывание выходной секции

Редактор связей не будет создавать секции с пустым содержимым (без данных). Это сделано для возможности описания входных секций в скрипте, которые могут или НЕ могут присутствовать во входных файлах. Например:

```
.other\_data : { *(.other\_data\_1 .other\_data\_2) }
```

выходная секция '.other_data' будет создана в выходном файле только в том случае, если имеется хотя бы одна непустая (с данными) входная секция '.other_data_1' или '.other_data_2' в любом из входных файлов.

2.3.3.14 STARTUP(filename)

Команда STARTUP подобна команде INPUT, за исключением того, что заданный файл будет первым в списке обрабатываемых файлов вне зависимости от места положения данной команды.

3. Руководство пользователя по функциональной модели

3.1 Общие сведения о функциональной модели процессора

Функциональная модель процессора предназначена для проверки правильности выполнения ранее собранной программы путем её выполнения на персональном компьютере. Текущее состояние памяти данных в шестнадцатеричном виде выводятся в текстовые файлы (по умолчанию: DMin.txt и DM.txt) до и после процесса моделирования.

В текущей версии модели реализован обмен данными через Uart по протоколу TCP (порт по умолчанию: 16400).

3.2 Использование функциональной модели процессора

Представленная функциональная модель запускается из командной строки командой *model*, аргументом которой является исполняемый бинарный файл. Поддерживаются также следующие опции:

-h или **-help** — показать это сообщение и выйти.

-inst-raw — включает вывод в консоль машинных слов в сыром виде соответствующих выполняемым командам.

-inst-render — включает вывод в консоль выполняемых команд в упакованном виде. Каждое командное машинное слово разбивается на следующие поля: код операции (cop), суффикс (sfx), тип операции (top), признак завершения параграфа (cmp), поля аргументов (F1 и F2).

-inst-asm — включает вывод в консоль выполняемых команд в дизассемблированном виде.

-inst-strange — включает вывод в консоль аргументов и результата выполняемых команд. В результате выполнения каждой команды на экран выводится одна строка из четырех колонок:

1. выставленные операцией флаги (z - флаг нуля, o - флаг переполнения, c - флаг заема, s - флаг знака);
2. результат операции;
3. первый аргумент операции;
4. второй аргумент операции.

-dump-raw — после выполнения программы выводить в консоль заданный диапазон из памяти данных в числовом виде. Вместе с этой опцией необходимо использовать опции **-dump-from**, **-dump-to**. Эта опция исключает использование **-dump-symbol**.

-dump-symbol — после выполнения программы выводить в консоль заданный диапазон из памяти данных в текстовом виде. Вместе с этой опцией необходимо использовать опции **-dump-from**, **-dump-to**. Эта опция исключает использование **-dump-raw**.

-dump-byte — выводить данные порциями по байту. Вместе с этой опцией необходимо использовать опции **-dump-row**, **-dump-from**, **-dump-to**.

-dump-short — выводить данные порциями по 2 байта. Вместе с этой опцией необходимо использовать опции **-dump-row**, **-dump-from**, **-dump-to**.

-dump-long — выводить данные порциями по 4 байта. Вместе с этой опцией необходимо использовать опции **-dump-row**, **-dump-from**, **-dump-to**.

-dump-float — выводить данные порциями по 4 байта в виде вещественного числа одинарной точности. Вместе с этой опцией необходимо использовать опции **-dump-row**, **-dump-from**, **-dump-to**.

-dump-quad — выводить данные порциями по 8 байт. Вместе с этой опцией необходимо использовать опции **-dump-row**, **-dump-from**, **-dump-to**.

-dump-addr — выводить адрес перед строкой данных. Вместе с этой опцией необходимо использовать опции **-dump-row**, **-dump-from**, **-dump-to**.

-dump-length dec_digits — задает количество столбцов, в которые выводится поток данных. Вместе с этой опцией необходимо использовать опции **-dump-row**, **-dump-from**, **-dump-to**.

-dump-from 0xhex_digits или dec_digits — задает адрес начала выводимой секции в памяти данных.

-dump-to 0xhex_digits или dec_digits — задает адрес конца выводимой секции в памяти данных.

-dump-file path_to_file — задает имя файла, в который будет сброшено текущее состояние секции памяти данных на момент начала и конца выполнения программы.

-without-hardware-errors — отключает моделирование известных аппаратных ошибок.

Примеры включения различных опций:

-inst-asm -inst-strange — выводить в консоль дизассемблированные команды, а также значение их аргументов и возвращаемый ими результат.

-inst-asm -inst-strange -inst-render -inst-row — выводить всю информацию о выполняемых командах.

-dump-row-dump-long-dump-addr-dump-length 10-dump-from 0x0-dump-

to 0x1FFFF — после выполнения программы вывести в консоль весь диапазон секции памяти данных в числовом виде порциями по 4 байта в 10 столбцов, выводя адрес перед каждой строкой.

4. Руководство пользователя по отладчику

4.1 Общие сведения об отладчике mc-dbg

Отладчик mc-dbg используется для отладки программ, написанных для мультиклеточных процессоров. Отладчик mc-dbg производит отладку в режиме работы с программой-эмулятором мультиклеточного микропроцессора.

В текущей версии реализован обмен данными через Uart по протоколу TCP (порт по умолчанию: 16400).

4.2 Процесс использования отладчика mc-dbg

Представленный отладчик запускается из командной строки. Обязательным аргументом является исполняемый бинарный файл. Для полноценной работы отладчика данный файл необходимо компилировать вместе с отладочной информацией (процесс описан в документации к mc-as). В качестве аргументов команд допустимо использовать числа в восьмеричной, десятиричной и шестнадцатеричной форме.

Поддерживаются также следующие опции:

-p или -port <dec number> — установить адрес порта, используемого TCP подключением, для обмена данными по Uart с отладчиком (по умолчанию 16400).

4.3 Набор команд

Отладчик поддерживает следующие группы команд:

- 4.3.1 Команды общего назначения
- 4.3.2 Команды управления процессом выполнения
- 4.3.3 Команды управления секцией памяти данных
- 4.3.4 Команды управления точками останова
- 4.3.5 Команды просмотра доступных меток
- 4.3.6 Другие команды

4.3.1 Команды общего назначения

4.3.1.1 help

Синтаксис: h или help

Назначение: выводит подробную информацию об указанной в аргументе команде.

Аргументы: в качестве единственного аргумента передается имя команды. В случае отсутствия аргументов выводит список поддерживаемых команд.

Пример:

h run

Результат: run||r runs the program without breakpoints

4.3.1.2 quit

Синтаксис: q или quit

Назначение: завершает работу отладчика.

Аргументы: отсутствуют.

4.3.2 Команды управления процессом выполнения

4.3.2.1 run

Синтаксис: r или run

Назначение: запускает программу с игнорированием всех точек останова.

Аргументы: отсутствуют.

Пример:

r

Результат: Model stopped at stop: (0x2a)

4.3.2.2 next

Синтаксис: n или step или next

Назначение: выполняет следующий параграф программы.

Аргументы: отсутствуют.

Пример:

n

Результат: Model stopped at start: (0x0) before L1: (0x4)

4.3.2.3 continue

Синтаксис: c или continue

Назначение: выполняет программу до следующей точки останова или её завершения.

Аргументы: отсутствуют.

Пример:

c

Результат: Model stopped by break point at start: (0x0) before L1: (0x4)

4.3.2.4 stop

Синтаксис: st или stop

Назначение: останавливает выполнение программы.

Аргументы: отсутствуют.

4.3.2.5 kill

Синтаксис: k или kill

Назначение: останавливает выполнение программы и сбрасывает данные в начальное состояние.

Аргументы: отсутствуют.

4.3.2.6 until

Синтаксис: until

Назначение: выполняет программу до тех пор пока адрес начала выполняемого параграфа не станет больше адреса, указанного в аргументе команды.

Аргументы: команда должна иметь 1 аргумент.

- *<Addr in machine words> — адрес машинного слова.
- <Label in program memory section> — метка в программной секции (имя параграфа).
- <Number of line> — номер строки в файле, определенном командой file.

Пример:

until L1

Результат: Model stopped at L1: (0xc) before L2: (0x10)

4.3.2.7 where

Синтаксис: where

Назначение: выводит на экран цепочку переходов от текущего параграфа до указанной в аргументе метки или конца программы.

Аргументы: команда должна иметь 1 аргумент.

- ***<Addr in machine words>** — адрес машинного слова.
- **<Label in program memory section>** — метка в программной секции (имя параграфа).
- **<Number of line>** — номер строки в файле, определенном командой file.

Пример:

where L3

Результат: ->start: (0x0)-> L1: (0x4)-> L1: (0x4)-> L1: (0x4) -> L1: (0x4)->
L2: (0xc)-> L3: (0x10)

4.3.2.8 jump

Синтаксис: j или jump

Назначение: устанавливает адрес выполнения следующего параграфа.

Аргументы: команда должна иметь 1 аргумент.

- ***<Addr in machine words>** — адрес машинного слова.
- **<Label in program memory section>** — метка в программной секции (имя параграфа).
- **<Number of line>** — номер строки в файле, определенном командой file.

Пример:

j L5

Результат: Jump to L5 (0x1F)

4.3.3 Команды управления секцией памяти данных

4.3.3.1 print

Синтаксис: p или print

Назначение: выводит на экран данные, адресуемые аргументом.

Использование: print **<EXPRESSION>**

EXPRESSION: поддерживаются выражения в стиле языка C, в том числе операции:

- * арифметические операции (сложение, вычитание, умножение, деление, взятие остатка);
- * не прямое обращение (например, *a);
- * обращение к элементу массива (например, a[b]);
- * обращение к члену структуры (например, a.b);
- * не прямое обращение к члену структуры (например, a->b);
- * явное приведение типов (например, (int)a);
- * операция вычисления размера типа (например, sizeof(int));
- * поразрядное НЕ (например, ~a);
- * логическое НЕ (например, !a);
- * условные операции (например, a>=b);
- * поразрядный сдвиг вправо и влево (например, a>>3);
- * операции проверки на равенство (например, a!=b или a==b);
- * поразрядное И (например, a&0xff);
- * поразрядное исключающее ИЛИ (например, a^3);
- * поразрядное ИЛИ (например, a|3);
- * логическое И (например, a&&b);
- * логическое ИЛИ (например, a||b).

Следует отметить, что наряду с типами языка C, также поддерживаются типы MC_Quad (целое, беззнаковое, 64 бита), MC_Long (целое, беззнаковое, 32 бита), MC_Short (целое, беззнаковое, 16 бит), MC_Byte (целое, беззнаковое, 8 бит).

Также поддерживается вывод диапазона значений:

1. `print <SOURCE>:<SOURCE>` — выводит значения в диапазоне от первого до второго параметра.
2. `print <SOURCE>@<COUNT>` — выводит значения в количестве COUNT машинных слов начиная с указанного параметра.

Примеры:

1. `p *(float*)0x3` — вывести значение начиная с четвертого байта в виде вещественного числа одинарной точности.

Результат:

`*(int*)0x3 = 0x00000000`

2. `p #34:#36` — вывести значения регистров с 34 по 36.
3. `p (unsigned char)Start@8` — вывести значения восьми байтов после метки Start.
4. `p *(&Start+10)` — вывести значение десятого байта после Start.
5. `p &Start-1:&Stop+1` — вывести значения начиная с байта перед меткой Start и заканчивая байтом после метки Stop.

4.3.3.2 display

Синтаксис: `di` или `display`

Назначение: выводит на экран данные, адресуемые аргументом после каждой остановки выполнения программы.

Использование: `display <EXPRESSION>`

EXPRESSION: Поддерживаются выражения в синтаксисе языка C, в том числе операции:

- * арифметические операции (сложение, вычитание, умножение, деление, взятие остатка);
- * не прямое обращение (например, `*a`);
- * обращение к элементу массива (например, `a[b]`);
- * обращение к члену структуры (например, `a.b`);
- * не прямое обращение к члену структуры (например, `a->b`);
- * явное приведение типов (например, `(int)a`);
- * операция вычисления размера типа (например, `sizeof(int)`);
- * поразрядное НЕ (например, `a`);
- * логическое НЕ (например, `!a`);

- * условные операции (например, $a \geq b$);
- * поразрядный сдвиг вправо и влево (например, $a \gg 3$);
- * операции проверки на равенство (например, $a \neq b$ или $a == b$);
- * поразрядное И (например, $a \& 0xff$);
- * поразрядное исключающее ИЛИ (например, $a \wedge 3$);
- * поразрядное ИЛИ (например, $a | 3$);
- * логическое И (например, $a \& \& b$);
- * логическое ИЛИ (например, $a || b$).

Следует отметить, что наряду с типами языка C, также поддерживаются типы MC_Quad (целое, беззнаковое, 64 бита), MC_Long (целое, беззнаковое, 32 бита), MC_Short (целое, беззнаковое, 16 бит), MC_Byte (целое, беззнаковое, 8 бит).

Также поддерживается вывод диапазона значений:

1. `display <SOURCE>:<SOURCE>` — выводит значения в диапазоне от первого до второго параметра.
2. `display <SOURCE>@<COUNT>` — выводит значения в количестве COUNT машинных слов начиная с указанного параметра.

Примеры:

1. `display *(float*)0x3` — вывести значение начиная с четвертого байта в виде вещественного числа одинарной точности, после каждой остановки в выполнении программы.

Результат:

`*(int*)0x3 = 0x00000000`

2. `di #34:#36` — вывести значения регистров с 34 по 36, после каждой остановки в выполнении программы.
3. `di (unsigned char)Start@8` — вывести значения восьми байтов после метки Start, после каждой остановки в выполнении программы.
4. `di *(&Start+10)` — вывести значение десятого байта после Start, после каждой остановки в выполнении программы.

5. di &Start-1:&Stop+1 — вывести значения начиная с байта перед меткой Start и заканчивая байтом после метки Stop, после каждой остановки в выполнении программы.

4.3.3.3 set

Синтаксис: s или set

Назначение: устанавливает указанное значение по указанному адресу (регистру или памяти данных).

Использование: set или s <destination> = <value>

<destination> - адрес записи

<value> - записываемое значение

Оба аргумента обрабатываются аналогично аргументу команды print.

Пример:

set #5 = 0x5

Результат: Set #5 to 0x5

set Label = 0x5

Результат: Set Label to 0x5

4.3.4 Команды управления точками останова

4.3.4.1 break

Синтаксис: b или break

Назначение: устанавливает точки останова согласно аргументу в памяти программ.

Использование: break <parameters>

Типы параметров:

- <label> — метка в области памяти программ.
- *<addr in machine words> — адрес машинного слова.

– <number of line> — номер строки в выбранном файле.

Пример:

b L2 L3 *0x6

Результат: Set breakpoint at L2(0xC)

Set breakpoint at L3(0x10)

Set breakpoint at 0x6

4.3.4.2 watch

Синтаксис: wa или watch

Назначение: устанавливает точки останова согласно аргументу в памяти данных.

Использование: watch <SOURCE>

SOURCE: Выражение в стиле синтаксиса языка C.

Также поддерживается установка нескольких точек останова:

1. watch <SOURCE>:<SOURCE> — устанавливает точки останова от первого до второго источника.
2. watch <SOURCE>@<COUNT> — устанавливает точки останова на COUNT машинных слов начиная с указанного источника.

Пример:

wa 0x0:0x3

Результат: Set watch at *0x0

Set watch at *0x1

Set watch at *0x2

Set watch at *0x3

4.3.5 Команды просмотра доступных меток

4.3.5.1 file

Синтаксис: f или file

Назначение: устанавливает файл, из которого будет осуществляться выборка доступных меток командой list. Кроме того, при передаче номера строки в качестве аргумента команде break, точка останова будет выставлена в указанный файл.

Аргументы:

При отсутствии аргументов выводится список доступных файлов.

В качестве единственного аргумента выступает имя одного из доступных файлов.

Пример:

1. f

Результат: Working with crt0.s

Available files:

crt0.s

math.core.s

math.coref.s

math.expf.s

2. f math.core.s

Результат: Working with math.core.s

4.3.5.2 list

Синтаксис: l или list

Назначение: выводит на экран метки из установленного командой file файла.

Аргументы:

При отсутствии аргументов выводит 10 следующих меток.

<-> Выводит 10 предыдущих меток.

Пример:

list

Результат: data memory: irq.desc.tbl

data memory: master.isr.retaddr

data memory: tsk.retval

program memory: INMI.isr

program memory: MPRGE.isr

program memory: STUB.isr

program memory: SWT.isr

program memory: SWT.isr.epilogue

program memory: __libc__main

program memory: divl.arg.check

4.3.6 Другие команды

4.3.6.1 info

Синтаксис: i или info

Назначение: выводит информацию. Обязательно наличие одной из опций.

Опции:

 или <break> — выводит информацию о точках останова в секции памяти программ.

<wa> или <watch> — выводит информацию о точках останова в секции памяти данных.

<di> или <display> — выводит информацию о списках отображения данных.

<add> или <address> *<address in machine words> — выводит информацию о метках по заданному в аргументе адресу.

<sym> или <symbol> <label in data memory or .txt section> — выводит адрес в машинных словах заданной в аргументе метки.

Пример:

i b

Результат:

	Index	Enabled	Address	Name
1.		true	0x39	divl.crt
2.		true	0x37	divsl.epilogue
3.		true	0x3F	divl.prepare

4.3.6.2 enable

Синтаксис: enable

Использование: enable <b или di> <all или list identifier>

Назначение: включает указанную точку останова или область отображения данных по указанному идентификационному номеру.

Аргумент:

<list identifier> — идентификационный номер, полученный командой info.

<all> — распространит действие команды на все точки останова или весь список выводимой информации.

Опции:

 — определяет указанный идентификатор как точку останова.

<di> — определяет указанный идентификатор как список выводимой информации.

Пример:

enable b 3

Результат:

Enable :

3. divl.prepare 0x3F true

4.3.6.3 disable

Синтаксис: disable

Использование: disable <b или di> <all или list identifier>

Назначение: выключает указанную точку останова или область отображения данных по указанному идентификационному номеру.

Аргумент:

<list identifier> — идентификационный номер, полученный командой info.

<all> — распространит действие команды на все точки останова или весь список выводимой информации.

Опции:

 — определяет указанный идентификатор как точку останова.

<di> — определяет указанный идентификатор как список выводимой информации.

Пример:

```
disable b 3
```

Результат:

```
Disable :
```

```
3. divl.prepare 0x3F false
```

4.3.6.4 delete

Синтаксис: d или delete

Использование: delete или d <b или di> <all или list identifier>

Назначение: удаляет указанную точку останова или область отображения данных по указанному идентификационному номеру.

Аргумент:

<list identifier> — идентификационный номер, полученный командой info.

<all> — распространит действие команды на все точки останова или весь список выводимой информации.

Опции:

`<b>` — определяет указанный идентификатор как точку останова.

`<di>` — определяет указанный идентификатор как список выводимой информации.

Пример:

```
delete b 3
```

Результат:

Delete :

```
3. divl.prepare 0x3F false
```

4.3.6.5 model

Синтаксис: `m` или `model`

Назначение: конфигурирует параметры функциональной модели.

Опции:

asm или **a** или **inst-asm** — включает вывод в консоль выполняемых команд в дизассемблированном виде.

asm- или **a-** или **inst-asm-** — отключает вывод в консоль выполняемых команд в дизассемблированном виде.

machine или **m** или **inst-raw** — включает вывод в консоль машинных слов в сыром виде соответствующих выполняемым командам.

machine- или **m-** или **inst-raw-** — отключает вывод в консоль машинных слов в сыром виде.

render или **r** или **inst-render** — включает вывод в консоль выполняемых команд в упакованном виде. Каждое командное машинное слово разбивается на следующие поля: код операции (cop), суффикс (sfx), тип операции (top), признак завершения параграфа (cmp), поля аргументов (F1 и F2).

render- или **r-** или **inst-render-** — отключает вывод в консоль выполняемых команд в упакованном виде.

switchboard или **sw** или **inst-strange** — включает вывод в консоль аргументов и результата выполняемых команд. В результате выполнения каждой команды на экран выводится одна строка из четырех колонок:

1. выставленные операцией флаги (z — флаг нуля, o — флаг переполнения, c — флаг заема, s — флаг знака).
2. результат операции.
3. первый аргумент операции.
4. второй аргумент операции.

switchboard- или **sw-** или **inst-strange-** — отключает вывод в консоль аргументов и результата выполняемых команд.

addr или **dump-addr** — выводить адрес перед строкой данных. В месте с этой опцией необходимо использовать опции **dump-raw**, **dump-from**, **dump-to**.

addr-или **dump-addr-** — отключает опцию **-dump-addr**.

raw или **dump-raw** — после каждой точки останова в выполнении программы выводить в консоль заданный диапазон из памяти данных в числовом виде. В месте с этой опцией необходимо использовать опции **dump-from**, **dump-to**. Эта опция исключает использование **symbol**.

symbol или **dump-symbol** — после каждой точки останова в выполнении программы выводить в консоль заданный диапазон из памяти данных в текстовом виде. В месте с этой опцией необходимо использовать опции **dump-from**, **dump-to**. Эта опция исключает использование **raw**.

DisableCmdOutput или **dco** — отключает вывод на экран данных о выполняемых командах.

DisableDataOutput или **ddo** — отключает опции **dump-raw** и **dump-symbol**.

byte или **b** или **dump-byte** — выводить данные порциями по байту. В месте с этой опцией необходимо использовать опции **raw**, **dump-from**, **dump-to**.

short или **sh** или **dump-short** — выводить данные порциями по 2 байта. В месте с этой опцией необходимо использовать опции **raw**, **dump-from**, **dump-to**.

long или **l** или **dump-long** — выводить данные порциями по 4 байта. В месте

с этой опцией необходимо использовать опции **raw**, **dump-from**, **dump-to**.

float или **f** или **dump-float** — выводить данные порциями по 4 байта в виде вещественного числа одинарной точности. В месте с этой опцией необходимо использовать опции **raw**, **dump-from**, **dump-to**.

quad или **q** или **dump-quad** — выводить данные порциями по 8 байт. В месте с этой опцией необходимо использовать опции **raw**, **dump-from**, **dump-to**.

length или **ln** или **dump-length dec _digits** — задает количество столбцов, в которые выводятся данные. В месте с этой опцией необходимо использовать опции **raw**, **dump-from**, **dump-to**.

dump-from или **from 0xhex _digits** или **dec _digits** — задает адрес начала выводимой секции в памяти данных.

dump-to или **to 0xhex _digits** или **dec _digits** — задает адрес конца выводимой секции в памяти данных.

dump-file или **dump path _to _file** — задает имя файла, в который будет сброшено текущее состояние секции памяти данных.

without-hardware-errors или **wh** — отключает моделирование известных аппаратных ошибок.

Примеры:

m a sw — выводить выполняемые команды в виде дизассемблера и триад.

m dump DataMemory.txt — сбросить текущее состояние памяти данных в файл DataMemory.txt.

4.3.6.6 stat

Синтаксис: stat

Назначение: управляет средствами накопления статистики.

Опции:

begin или **b** — включает сбор статистики, предыдущие данные сбрасываются.

end или **e** — выключает сбор статистики.

«**File path**» — выводит накопленную статистику в файл по указанному пути.

4.4 Процесс использования отладчика mc-dbg в графическом режиме

Присутствует возможность отладки в графическом режиме в Geany IDE.

Официальный сайт [Geany IDE](#).

Данное руководство содержит следующие разделы:

- Установка
- Менеджер проектов
- Инструменты сборки
- Инструменты отладки
- Настройка горячих клавиш
- Пример

4.4.1 Установка

Под ОС Windows установка может быть проведена вручную или при помощи [инсталлятора](#) на нашем сайте. Установка при помощи инсталлятора включает следующие шаги:

- Скачать [инсталлятор](#) и произвести при помощи него установку.
- Установить драйвер ftdi.
- Перезагрузить компьютер.
- Запустить Geany.
- Во вкладке «Инструменты/Менеджер модулей» активировать модуль MC-DBG.

Для подключения отладчика к Geany IDE в качестве плагина под ОС Windows вручную необходимо выполнить следующие действия:

- Скачать [geany-1.23.1_nogtk_setup.exe](#) и установить его.
- Скачать [gtk+ 2.24](#) и распаковать его в папку с Geany.
- Установить [MSYS 1.0.11](#) и добавить путь к утилите make.exe в переменную среды Path.
- Скачать плагин с сайта [Мультиклет](#) и распаковать его в папку с Geany.

- Запустить Geany.
- Во вкладке «Инструменты/Менеджер модулей» активировать модуль MC-DBG.

Под ОС Linux установка сводится к следующим шагам:

- Установить из репозитория Geany и libpthread.
- Установить файл библиотеки libftd2xx.so, распространяемые вместе с плагином. Если данная библиотека уже установлена у вас на компьютере, настоятельно рекомендуется заменить её распространяемой в архиве файлом.
- Запустить Geany.
- Перейти во вкладку главного меню «Правка/Настройки/Общее».
- Указать путь до файлов плагина MC-DBG-lin-32.so и MC-DBG-lin-64.so в переменной «Дополнительный путь для модулей».
- Перезапустить Geany.
- Во вкладке «Инструменты/Менеджер модулей» активировать модуль MC-DBG.

Вкладки, появляющиеся при включении плагина, изображены на рис. 4.1

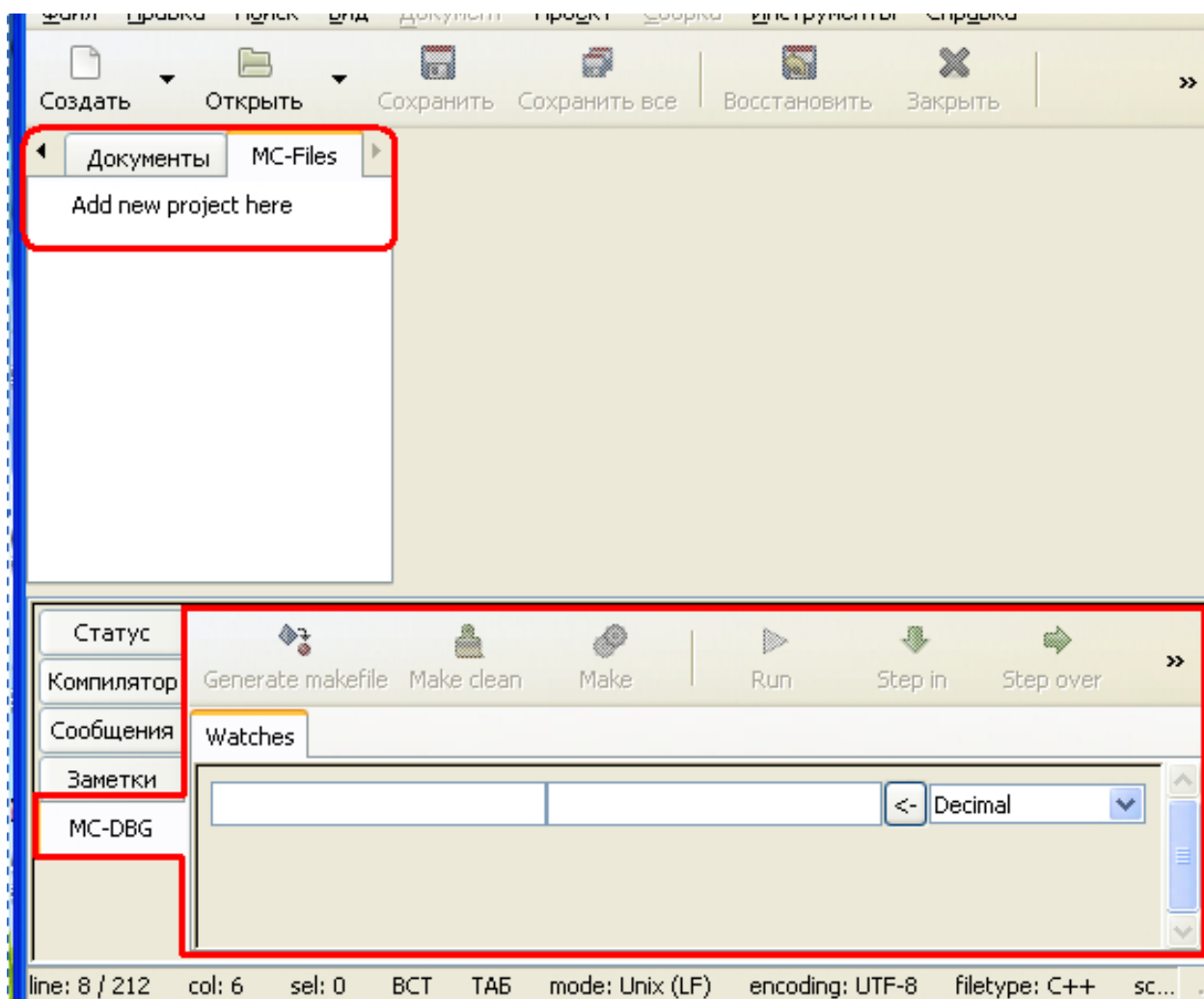


Рис. 4.1: Окна, появляющиеся после включения плагина

4.4.2 Менеджер проектов

Общие настройки менеджера проектов доступны во вкладке «Edit/Plugin Preferences». Эти настройки включают настройки компиляции, сборки, а также загрузки и отладки. Пример заполнения полей приведен ниже (см. рис. 4.2).

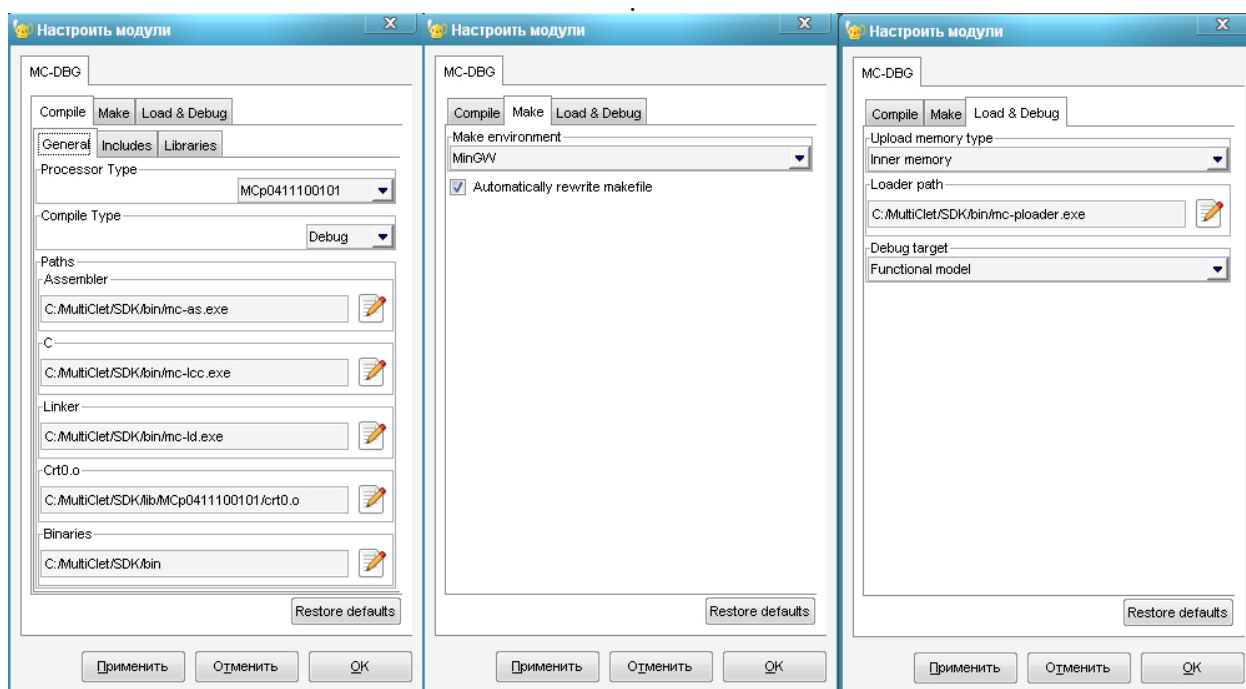


Рис. 4.2: Настройка модуля MC-DBG

Настройки компиляции включают в себя:

- Выбор типа процессора.
- Управление режимом компиляции (включение отладочного режима).
- Установку полного имени для ассемблера, компилятора и линкера.
- Установку полного имени для crt0 и папки Multiclet/bin.
- Список директорий с заголовочными файлами, подключаемых ко всем проектам.
- Список библиотек, подключаемых ко всем проектам.

Настройки сборки включают в себя:

- Выбор среды сборки.
- Опцию отключения автоматического пересоздания make файла при повторной сборке.

Настройки загрузки и отладки включают в себя:

- Установку полного имени для загрузчика.
- Опционально для R1: выбор между загрузкой программы в ПЗУ или ОЗУ.

- Опционально для R1: выбор между отладкой на функциональной модели или через JTAG.

Доступ к функциям менеджера текущего проекта предоставляется в главном меню во вкладке «Проект», а также в окне «MC-Files». Присутствует возможность работы сразу с несколькими проектами. При этом, активным при сборке и отладке будет считаться тот проект, элемент которого выбран в данный момент в окне «MC-Files». Параметры проекта могут быть вызваны через вкладку «Проект/MS-Project Properties» или через контекстное меню проекта в окне «MC-Files». Настраиваются следующие параметры:

- Список файлов проекта. Сборка файлов производится в порядке их следования в этом списке. Согласно расширению файла он передается компилятору ассемблера (расширения .s, .asm), компилятору C (расширения .c) или сразу редактору связей (все остальные). Пример списка файлов проекта изображен на рис. 4.3

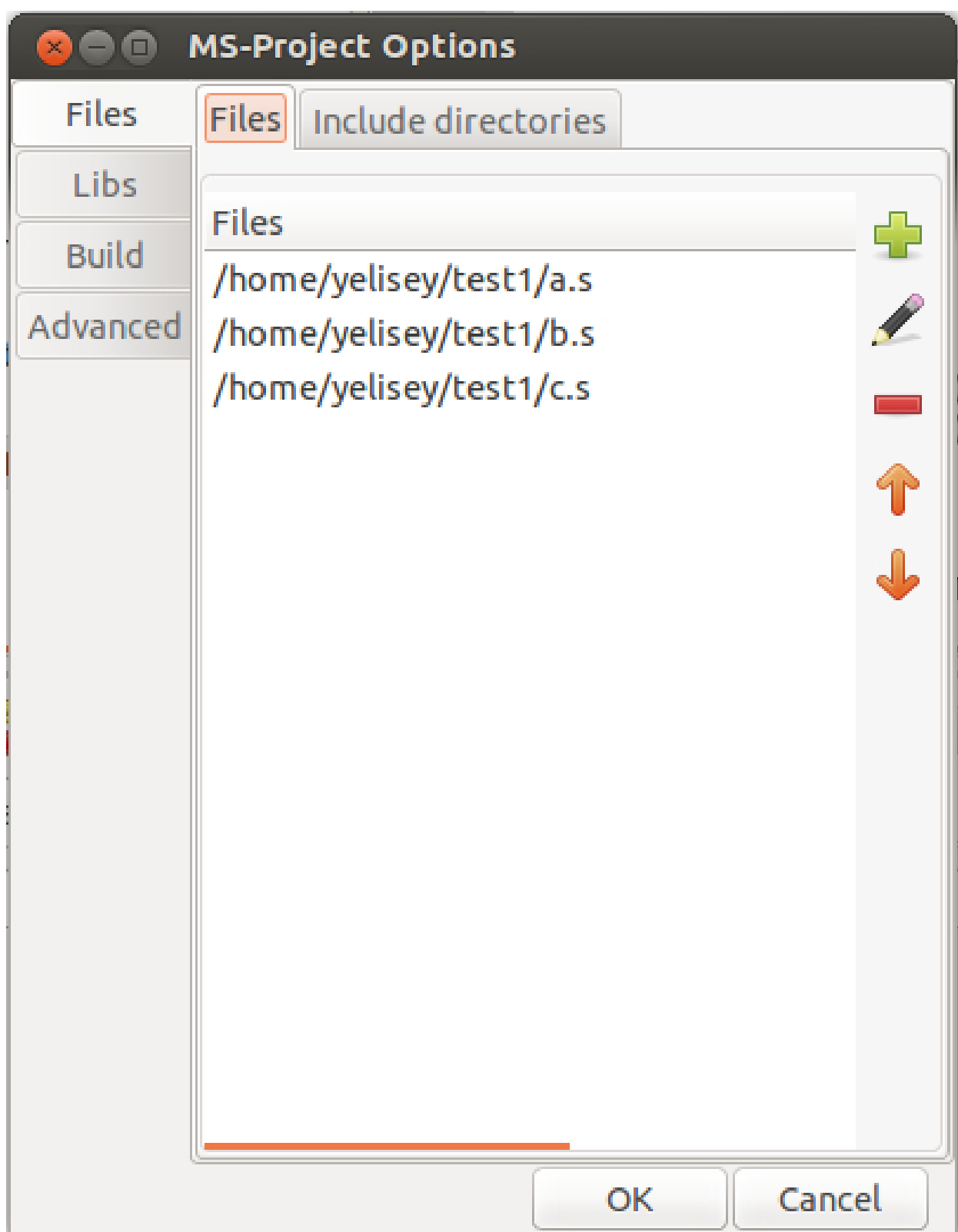


Рис. 4.3: Список файлов проекта

- Список директорий с заголовочными файлами.

- Список подключаемых библиотек.
- Список директорий с подключаемыми библиотеками.
- Директория сборки проекта.
- Команды для компилятора ассемблерных файлов (по умолчанию `mc-as`), для компилятора C файлов (по умолчанию `mc-icc`) и редактору связей (по умолчанию `mc-ld`), а также списки дополнительных опций к этим командам. В каждом конкретном проекте имеется возможность переключения между настройками цепи сборки по умолчанию (задаются в окне общих настроек менеджера проекта) и индивидуальной для текущего проекта цепи сборки. Первоначально для каждого проекта применяется цепь сборки по умолчанию. Управление переключением между цепями сборки осуществляется при помощи пункта «Use default toolchain» во вкладке «Build» настроек проекта.

4.4.3 Инструменты сборки

Плагин MC-DBG предлагает следующие инструменты сборки (см. рис. 4.4):

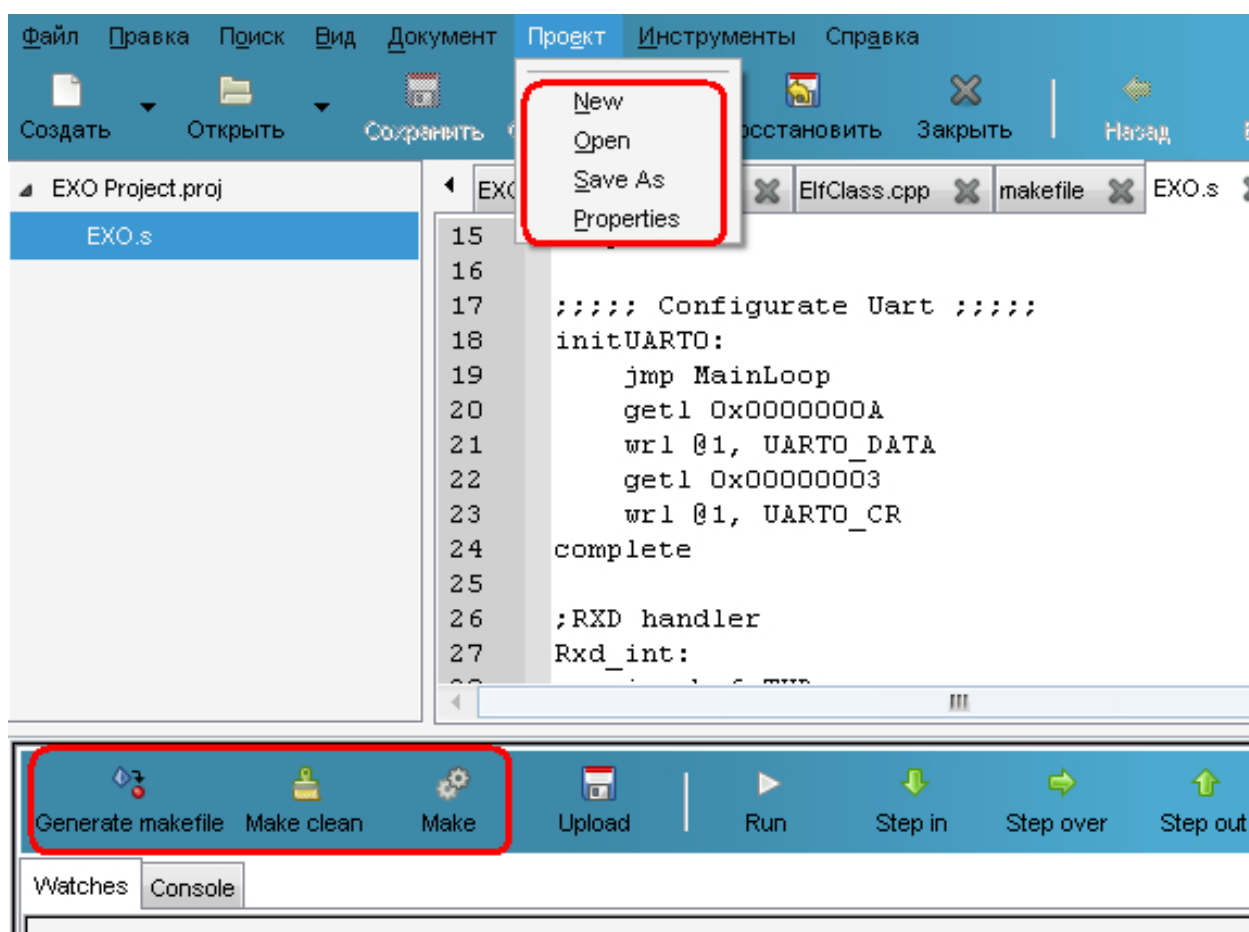


Рис. 4.4: Инструменты сборки

- Генерация файла сборки проекта.
- Сборка проекта (с помощью команды make).
- Сборка отдельных файлов проекта.
- Очистка проекта.

Данные функции доступны на панели инструментов окна MC-DBG, а также в контекстном меню файлов и проектов окна MC-Files. В результате успешной сборки проекта, будет создан бинарный файл, который будет размещен в директории сборки проекта. В случае возникновения каких-либо ошибок, они будут отображены в окне «Console». Если в сообщении об ошибке указано имя файла и номер строки, в которой допущена ошибка, то щелчком мыши по записи об ошибке может быть осуществлен переход к ее местоположению.

4.4.4 Инструменты отладки

Инструменты отладки плагина MC-DBG могут быть использованы только после успешной сборки проекта. Весь процесс отладки выполняется на функциональной модели. Эти инструменты расположены на панели инструментов окна MC-DBG (см. рис. 4.5).



Рис. 4.5: Инструменты отладки

К ним относятся:

- выполнение программы с игнорированием всех точек останова (Run);
- выполнение программы пошагово (Step In). Одним шагом при отладке ассемблерных файлов мультиклеточной архитектуры считается параграф;
- выполнение программы пошагово без остановки во вложенных лексических блоках (Step Over);
- выполнение программы пошагово без остановки до выхода из текущего лексического блока (Step Out);
- выполнение программы до следующей точки останова (Continue);
- выполнение программы до текущего местоположения курсора (Run to cursor);
- принудительная установка адреса выполнения следующего параграфа (Jump to cursor);
- прекращение отладки (Stop);
- приостановление выполнения программы (Pause).

В процессе отладки зеленой строкой отмечается первая команда следующего параграфа. Для того чтобы установить точку останова в память программ, необходимо левым щелчком мыши поставить маркер на соответствующей строке. Механизм работы точек останова имеет важную особенность, обусловленную нюансами мультиклеточной архитектуры. Она заключается в том, что в случае установки точки останова на первую команду в параграфе (см. рис. 4.6), выполнение программы будет приостановлено дважды (перед выполнением параграфа и после его выполнения). В случае установки точки останова не на первую команду в параграфе, выполнение программы будет приостановлено один раз после выполнения данного параграфа.

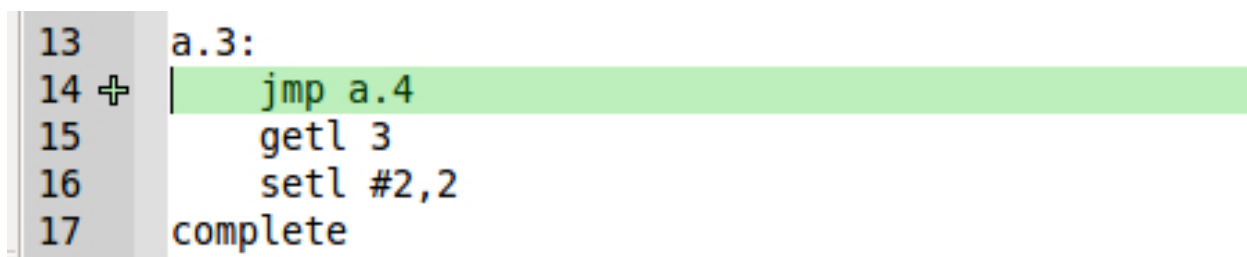


Рис. 4.6: Точка останова на первой команде в параграфе

Для того чтобы добавить какую-либо переменную в список Watches, необходимо ввести ее имя или адрес в левое поле таблицы окна MC-DBG. Значение переменной появится в правом поле таблицы. Для того чтобы вывести значение регистра, требуется перед номером регистра поставить символ '#' (например, #1). В случае, если в имени переменной допущена ошибка, она будет выведена в правую колонку таблицы (см. рис. 4.7).

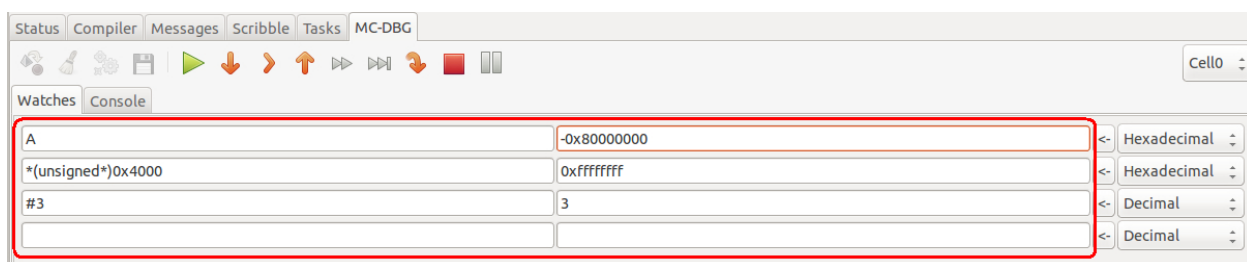


Рис. 4.7: Переменные в списке Watches

Для того чтобы выставить значение какой-либо переменной, занесенной в список Watches, необходимо записать требуемое значение в правую колонку таблицы соответствующей строки и нажать кнопку Set со стрелкой влево, расположенную правее (см. рис. 4.8).

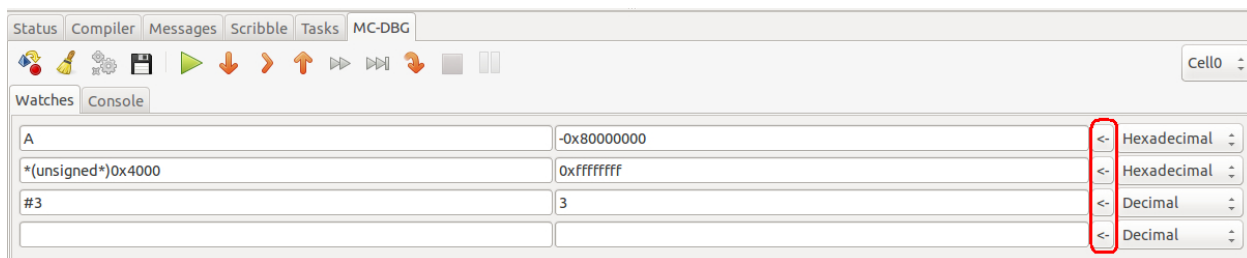


Рис. 4.8: Кнопки Set

В полях правее кнопки set (см. рис. 4.9) задается формат отображения данных (Hexadecimal — шестнадцатеричный, Decimal — десятичный, Octal — восьмиричный, Symbolic — символьный).

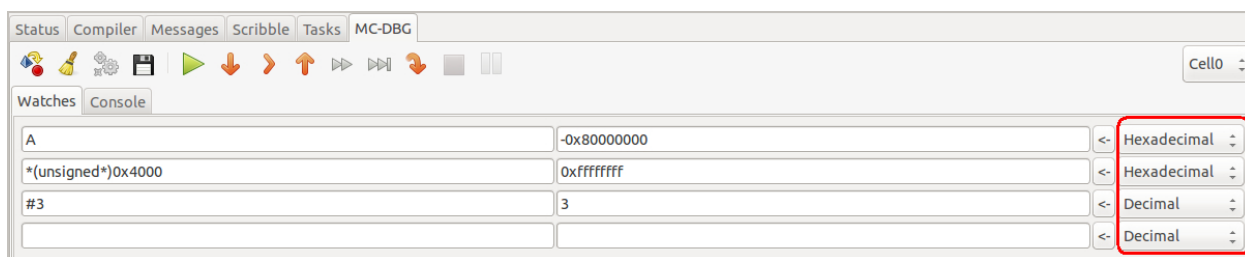


Рис. 4.9: Кнопки настройки отображения выводимой информации

Поддерживаются следующие операции (см. рис. 4.10):

- арифметические операции (сложение, вычитание, умножение, деление, взятие остатка);
- не прямое обращение (например, *a);
- обращение к элементу массива (например, a[b]);
- обращение к члену структуры (например, a.b);
- не прямое обращение к члену структуры (например, a->b);
- явное приведение типов (например, (int)a);
- операция вычисления размера типа (например, sizeof(int));
- поразрядное НЕ (например, ~a);
- логическое НЕ (например, !a);
- условные операции (например, a>=b);
- поразрядный сдвиг вправо и влево (например, a>>3);
- операции проверки на равенство (например, a!=b или a==b);
- поразрядное И (например, a&0xff);
- поразрядное исключающее ИЛИ (например, a^3);
- поразрядное ИЛИ (например, a|3);
- логическое И (например, a&&b);
- логическое ИЛИ (например, a||b);
- вывод блока данных через конструкции: **Адрес первого выводимого элемента:Адрес последнего выводимого элемента** (например, a:b) или **Адрес первого выводимого элемента@количество выводимых байт** (например, a@16).

Следует отметить, что наряду с типами языка C, также поддерживаются типы MC_Quad (целое, беззнаковое, 64 бита), MC_Long (целое, беззнаковое, 32 бита), MC_Short (целое, беззнаковое, 16 бит), MC_Byte (целое, беззнаковое, 8 бит).

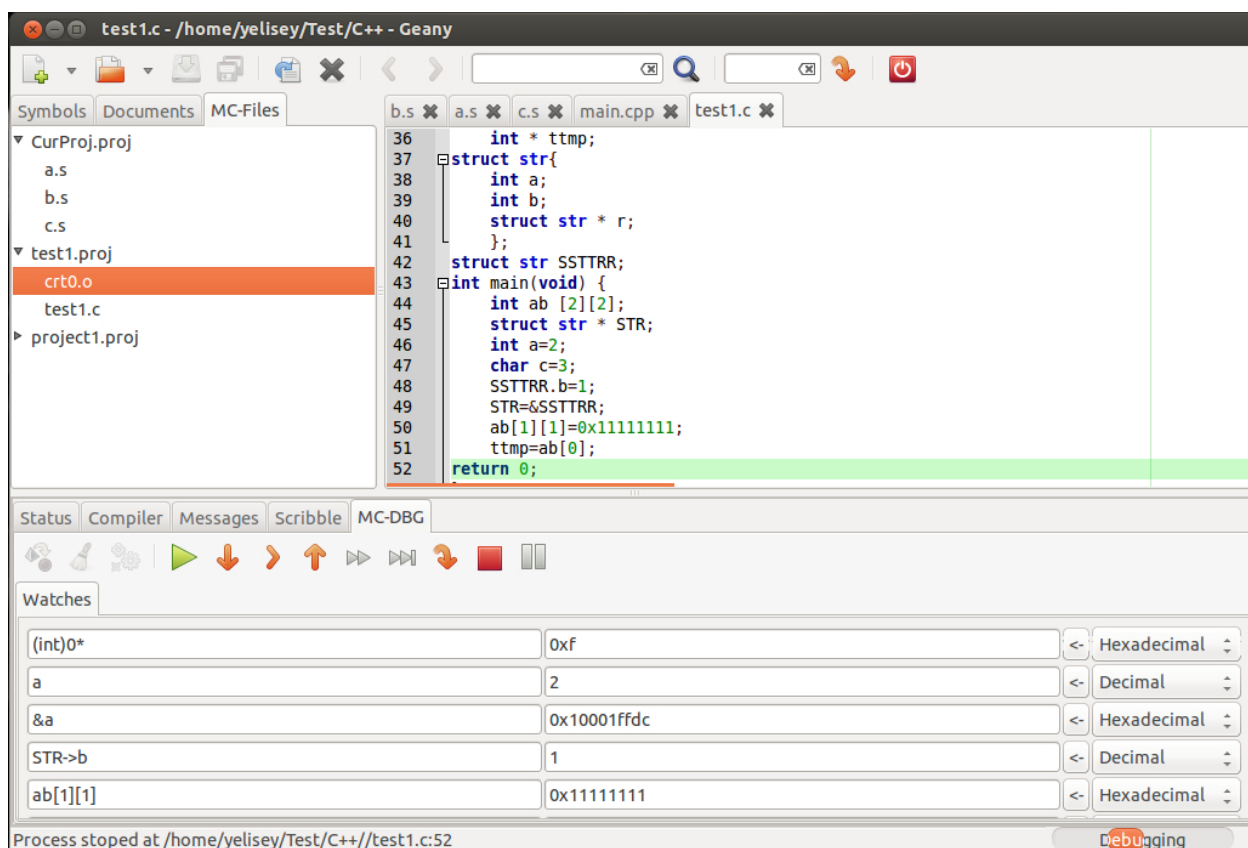


Рис. 4.10: Пример заполнения полей Watches при отладке кода, написанного на C

4.4.5 Настройка горячих клавиш

Для настройки горячих клавиш необходимо открыть «Правка/Настройки/Сочетание клавиш» и настроить их по своему вкусу (клавиши плагина в конце списка) (см. рис. 4.11).

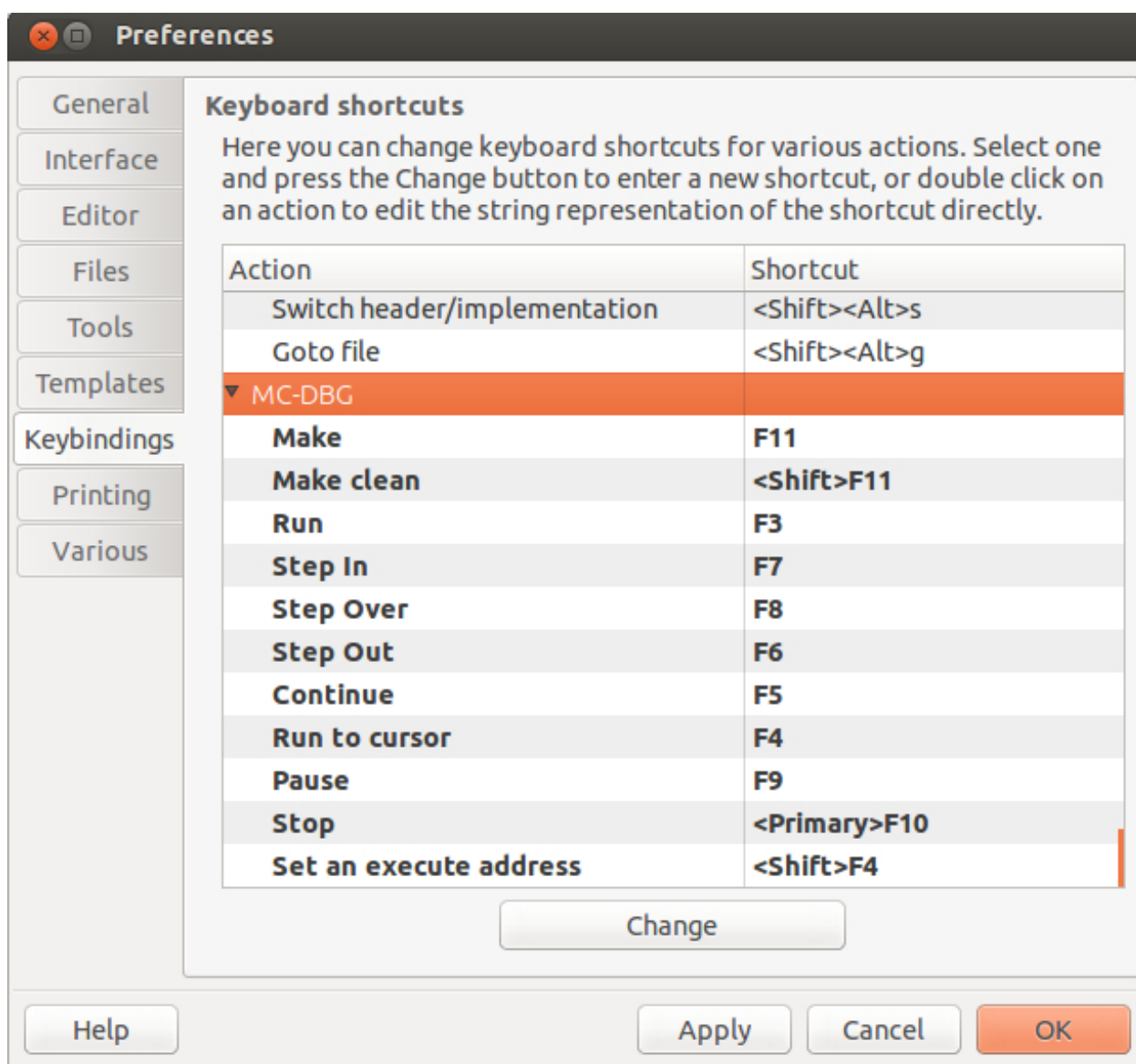


Рис. 4.11: Пример заполнения полей Watches при отладке кода, написанного на C

4.4.6 Пример

Напишем на ассемблере простейший эхо сервер, работающий через Uart. В качестве клиентской стороны будем использовать tcp_client.exe из стандартного набора разработчика компании Мультиклет.

Для начала создадим новый файл EXO.s стандартными средствами Geany. Запишем в него код эхо сервера, написанный на базе примера uart0_inv.asm, доступного в [архиве](#). Код приведен ниже:

```
1 .text
```



```
2  .include "HDL50001_pcf.inc"
3
4  .alias Get_byte 2
5
6  .text
7
8  start:
9      jmp initUART0
10     getl 0x00000300
11     wrl @1, GPIOB_BPS
12     getl 0x00000104
13     wrl @1, UART0_BDR
14 complete
15
16     ;;;; Configure Uart ;;;;
17
18 initUART0:
19     jmp MainLoop
20     getl 0x0000000A
21     wrl @1, UART0_DATA
22     getl 0x00000003
23     wrl @1, UART0_CR
24 complete
25
26     ;RXD handler
27
28 Rxd_int:
29     jmp buf_TXD
30     rdl UART0_DATA
31     setl #Get_byte, @1
32 complete
33
34     ;test FIFO on full
35
36 buf_TXD:
37     rdl UART0_ST
38     getl 0x00000200
```

```
39      and @1, @2
40      jne @1, buf_TXD
41      je @2, Send_byte
42  complete
43
44  ;Send inverse byte
45
46  Send_byte:
47      jmp MainLoop
48      getl #Get_byte
49      wr1 @1, UART0_DATA
50  complete
51
52  ;;Loop
53
54  MainLoop:
55      rdl UART0_ST
56      getl 0x00000001
57      and @2,@1
58      je @1, MainLoop
59      jne @2, Rxd_int
60  complete
```

Создаем новый проект EXOProject. Данное действие возможно выполнить либо через вызов контекстного меню списка текущих проектов и выбора пункта «New», либо через главное меню «Проект/New MC-Project». Откроется диалоговое окно, в котором требуется указать имя нового файла проекта. После успешного создания проекта его имя отобразится в окне MC-Files.

Добавляем файл EXO.s в наш проект при помощи кнопки «Add File» контекстного меню проекта в окне MC-Files.(см. рис. 4.12)

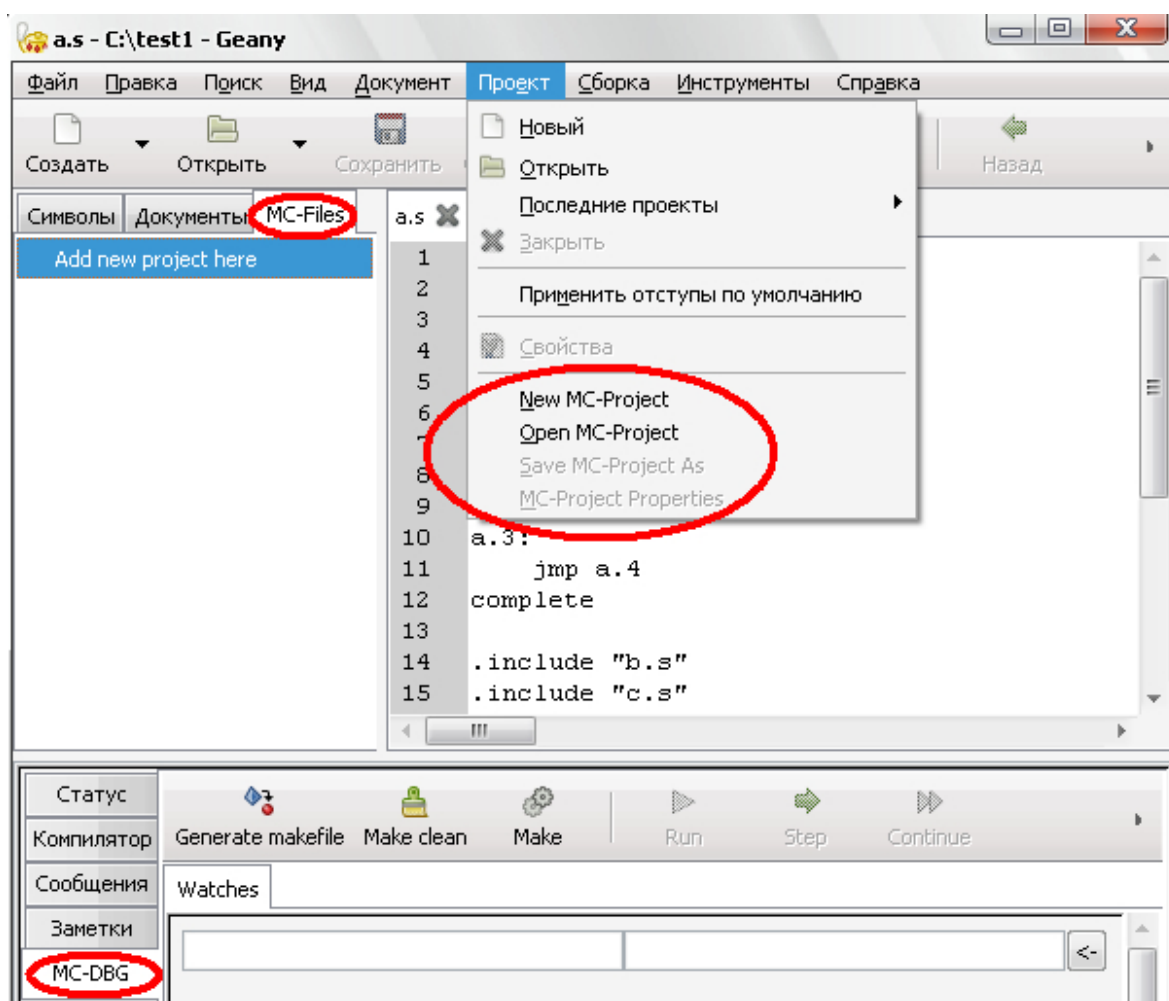


Рис. 4.12: Вид окна после успешного создания проекта и добавления в него файла

Затем необходимо настроить проект. Вызываем «Проект/MC-Project Properties» и добавляем к «Files/Include directories» папку с HDL50001_pcf.inc. Теперь можно попробовать собрать проект. Для этого нажимаем кнопку Make. В результате получим сообщение (см. рис. 4.13).

Статус	make -f C:/exo/makefile (в каталоге: ./)
Компилятор	mc-as C:/exo/EXO.s --include-path=C:/MultiClet/SDK/include/MCp0411100101 -g --arch=MCp0411100101 --output=C:/exo/EXO.o
Сообщения	mc-ld C:/exo/EXO.o --output=C:/exo/image.bin
Заметки	Сборка прошла успешно.

Рис. 4.13: Результат нажатия клавиши Make

Теперь, когда сборка прошла успешно, мы можем приступить к отладке. Выполняем программу по параграфам кнопкой Step (см. рис. 4.14). Зеленым подсвечивается первая команда следующего параграфа.

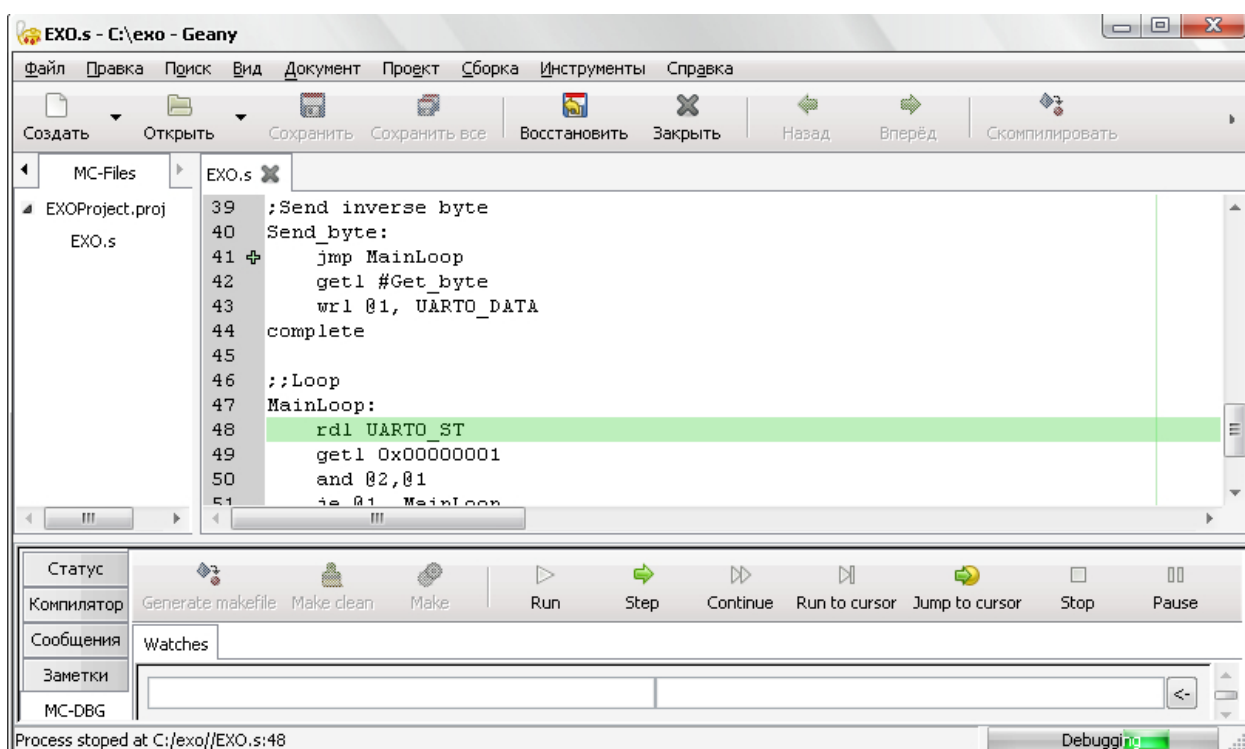


Рис. 4.14: В процессе отладки

Видим, что программа заиклилась на параграфе MainLoop до тех пор, пока на Uart не поступят новые данные. Запускаем tcp_client.exe и отправляем по Uart 0x1 (см. рис. 4.15).

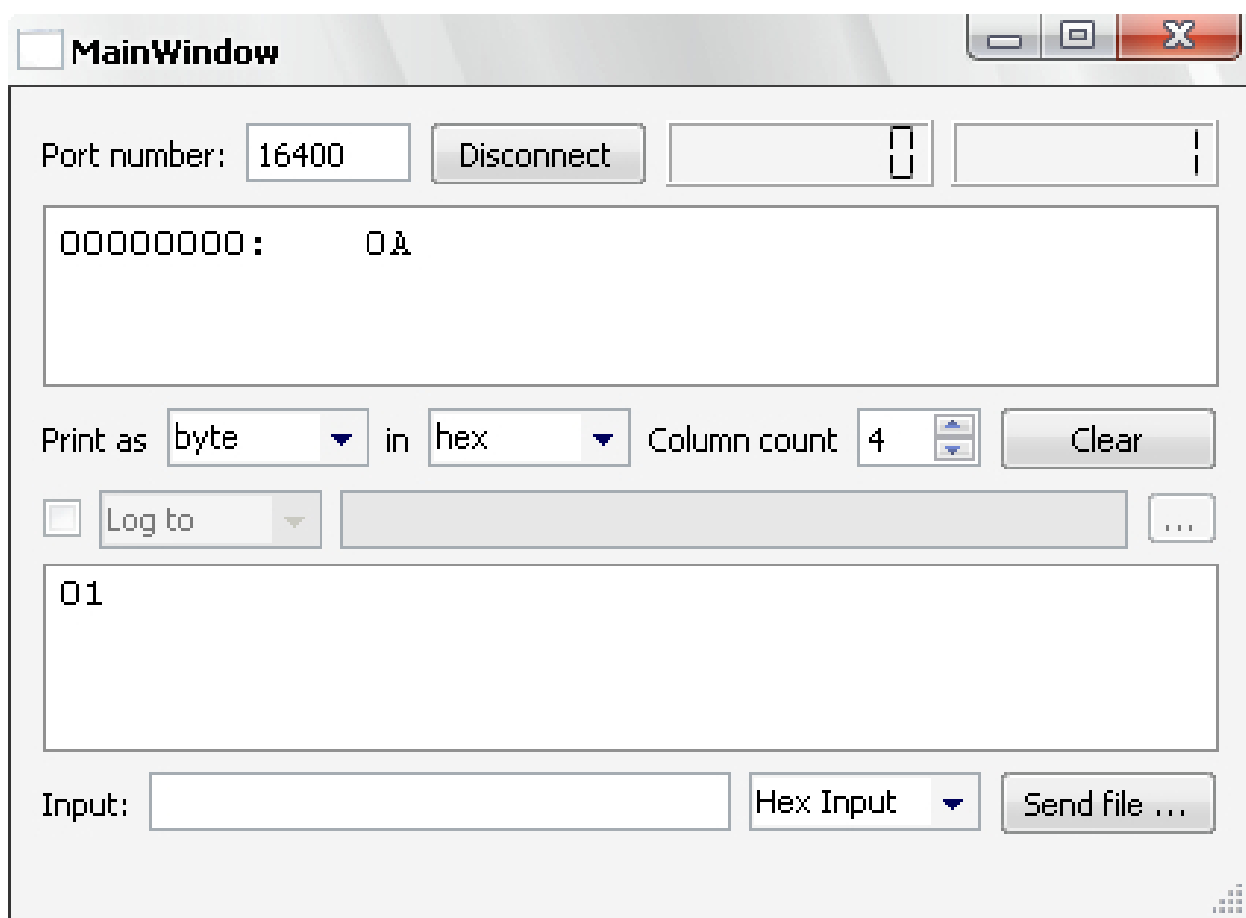


Рис. 4.15: Окно приложения tcp_client.exe

Возвращаемся в Geany. Ставим точку останова на первую команду параграфа Send_byte и нажимаем Continue. Выполнение программы прекращается перед параграфом Send_byte. Во второй регистр записано полученное по Uart значение. Для того чтобы получить его значение вводим #2 в левое поле Watches (# обозначает регистр, для чтения из DM вводится просто адрес например 2 или 0x2)(см. рис. 4.16).

Важное замечание. Механизм работы точек останова имеет важную особенность, обусловленную нюансами мультиклеточной архитектуры. Она заключается в том, что в случае установки точки останова на первую команду в параграфе, выполнение программы будет приостановлено дважды (перед выполнением параграфа и после его выполнения). В случае установки точки останова не на первую команду в параграфе, выполнение программы будет приостановлено один раз после выполнения данного параграфа.

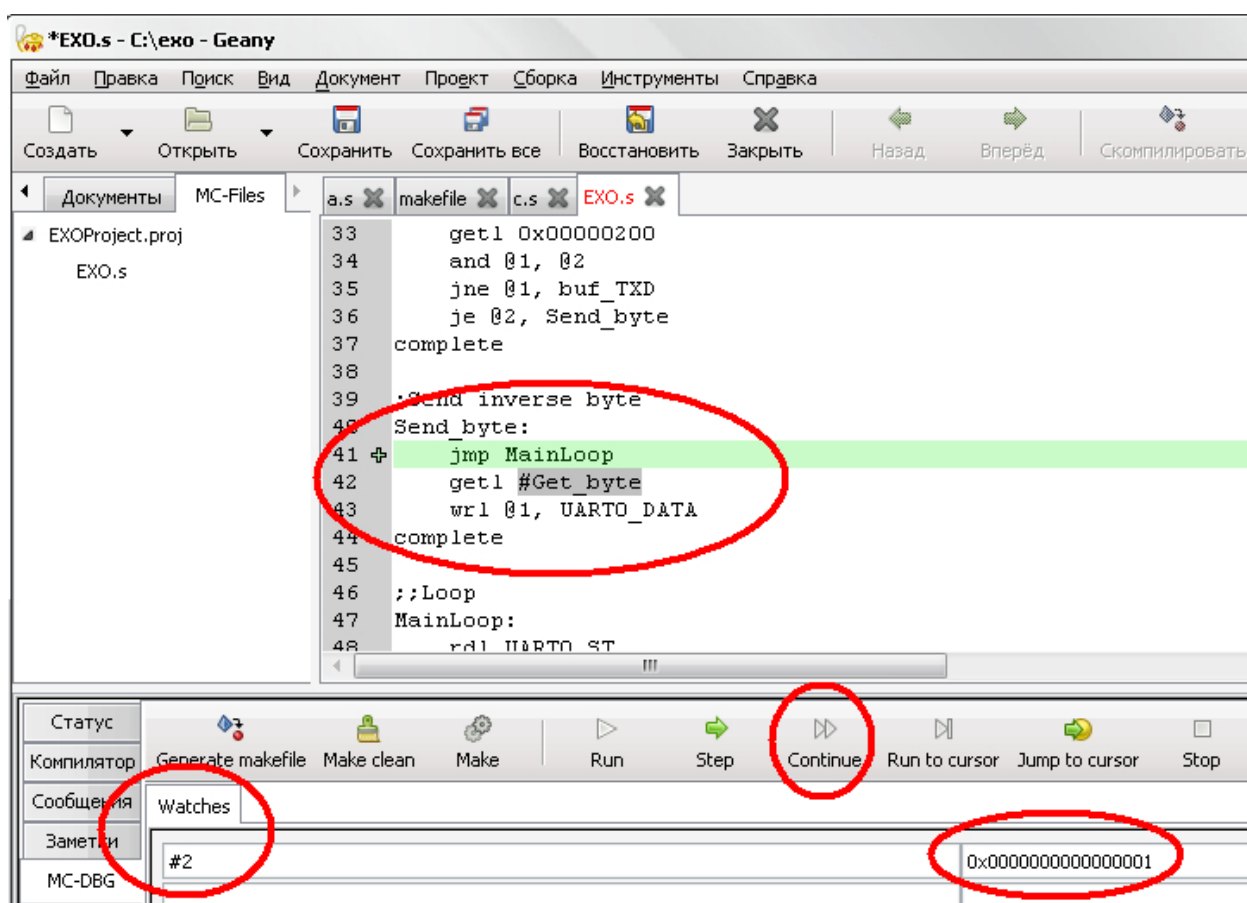


Рис. 4.16: Окно отладчика после остановки перед параграфом Send_byte и чтения #2

Теперь запишем во второй регистр число 5 средствами отладчика. Для этого заменим число 1 на число 5 в правом поле и нажмем кнопку Set (на ней нарисована стрелка влево) (см. рис. 4.17).

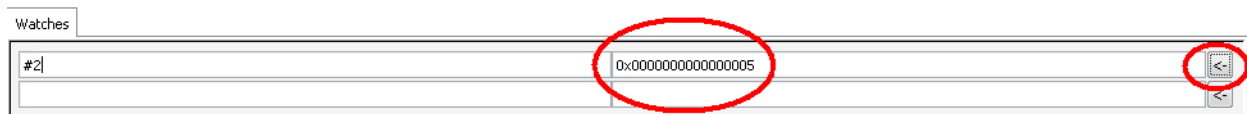


Рис. 4.17: Присвоение #2 значения 5

Вновь нажмем Continue (выполнение программы продолжится). В tcp_client.exe получим закономерный результат (см. рис. 4.18). Остановим отладку кнопкой Stop.

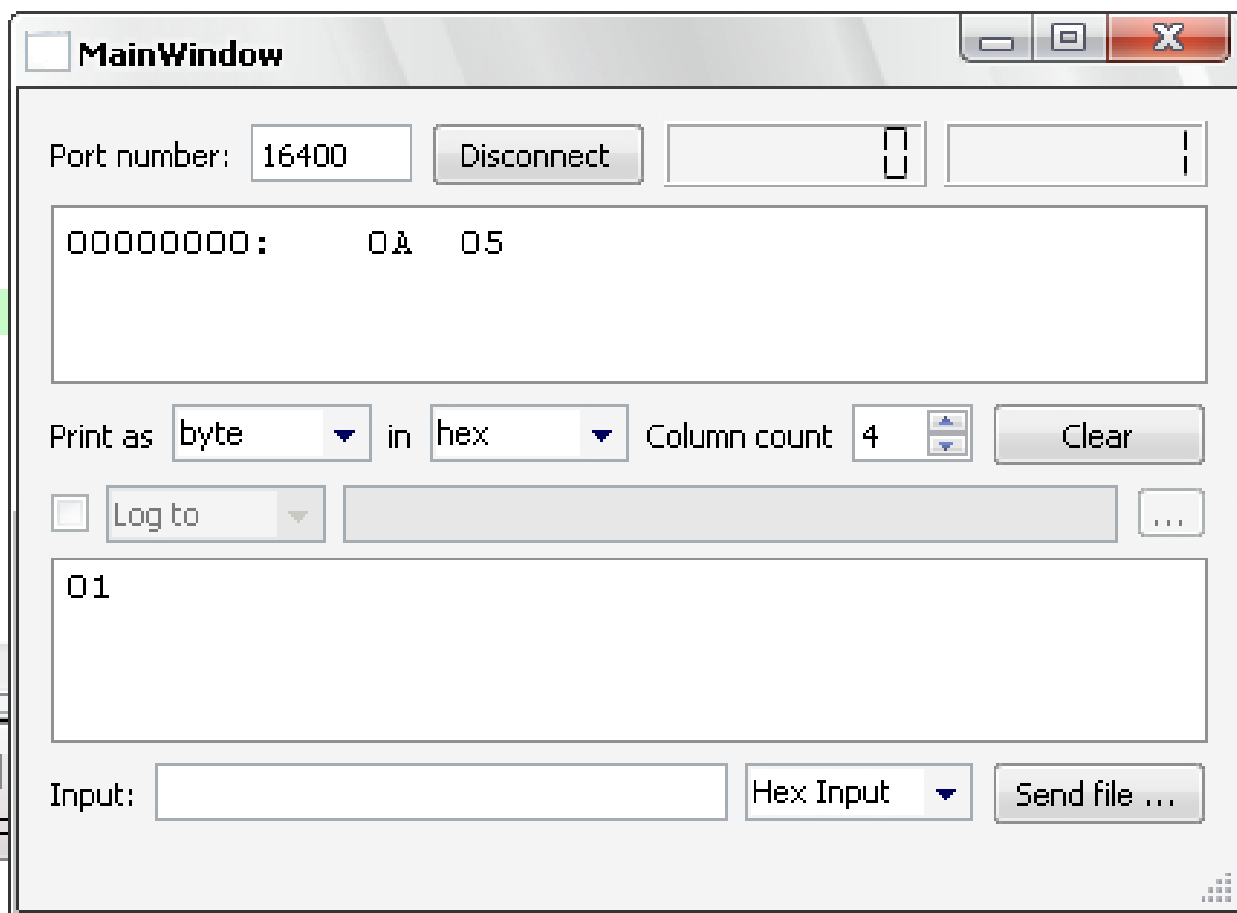


Рис. 4.18: Результат отладки