Руководство пользователя по ассемблеру MultiClet S1

Данный документ описывает процесс и особенности создания программ, написанных на ассемблере для мультиклеточного процессора S1.

Содержание

1 Общие сведения об ассемблере

Представленный ассемблер используется для написания программ для мультиклеточных процессора S1.

1.1 Запуск ассемблера и опции командной строки

Ассемблер запускается из командной строки командой ***mc-as-s1***, 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

-v, --verbose — показать подробный лог ассемблирования

-t, --tcom — сгенерировать файлы с выходными данными в текстовом формате

-V — напечатать номер версии ассемблера

-h, --help — показать это сообщение и выйти

2 Общие сведения о мультиклеточном процессоре

Процессор Multiclet S1 имеет в своем составе мультиклеточное процессорное ядро — процессорное ядро с принципиально новой мультиклеточной архитектурой российской разработки. Мультиклеточный процессор предназначен для решения широкого круга задач управления и цифровой обработки сигналов в приложениях, требующих минимального энергопотребления и высокой производительности.

2.1 Организация памяти

Память для процессора Multiclet S1 с программной точки зрения представляет собой массив с байтовой адресацией, диапазон адресов $[0, 2^{32})$.

2.2 Регистры

Мультиклеточный процессор Multiclet S1 имеет в своем составе следующие регистры:

- регистры общего назначения (РОН [GPR — General Purpose Register]);
- регистры управляющие (РУ [CR — Control Register]);

Обращение к регистру осуществляется по его номеру или имени, которому предшествует символ диеза '#', с помощью специализированных команд. 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.

2.2.1 Регистры общего назначения

Используются в качестве сверхбыстрой памяти (Scratchpad memory). Для обращения к какому-либо регистру данного типа используются либо номера от 6 до 37, либо имена $GPR0 - GPR32$. Начальное значение регистров данного типа не определено.

2.2.2 Регистры управляющие

Процессор имеет в своем составе следующие управляющие регистры:

Номер регистра	**Имя регистра**	**Права доступа**	**Описание**
0	PSW	R/W	Регистр состояния процессора

1	RETA	R	Регистр адреса продолжения
2	NXTP	R	Регистр адрес следующего параграфа исполняемой программы
3	BP	R/W	Регистр базового адреса локальных переменных исполняемой программы
4	SP	R/W	Регистр указателя начального адреса свободной памяти данных
5	RETV	R/W	Регистр возвращаемого значения
6	AOR	R/W	Регистр смещения адреса исполняемой программы

2.2.2.1 Регистр PSW состояния процессора

Регистр состояния процессора 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		признак состояния ядра, если данный бит «0», то ядро занято, иначе свободно*, признак формируется аппаратно, в исходном состоянии равен единице
1		признак отмены очередности исполнения команд чтения/записи, если данный бит «0», то установлен режим контроля очередности исполнения команд чтения/записи, иначе контроль отменен, в исходном состоянии равен нулю
2		признак готовности адреса следующего параграфа, если он равен «0», то адрес не готов, иначе готов**, в исходном состоянии признак равен нулю
3		признак «complete». В исходном состоянии он равен нулю – предыдущий параграф не закончен. В «1» признак устанавливается при декодировании команды с признаком «complete» и отсутствии на данный момент адреса следующего параграфа (2-ой бит равен «0»)
4 — 63		Зарезервировано

* ядро считается свободным, если нет обращений в память программ, в буферах клеток нет команд и не сформирован адрес следующего параграфа.

** признак готовности адреса следующего параграфа формируется, если команды текущего параграфа еще не выбраны, а адрес уже сформирован. В этом случае адрес записывается в регистр NXTR а во второй бит PSW записывается единица.

2.2.2.2 Регистр RETA адреса продолжения

Содержимое регистра формируется автоматически при обработке команды с признаком «complete». В регистр записывается адрес команды, следующей за командой с признаком . Начальное значение регистра равно нулю.

Структура регистра RETA

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

2.2.2.3 Регистр NXTP адрес следующего параграфа исполняемой программы

Устанавливается программно. Начальное значение регистра равно нулю.

Структура регистра NXTP

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

2.2.2.4 Регистр BP базового адреса локальных переменных исполняемой программы

Устанавливается программно. Начальное значение регистра равно нулю.

Структура регистра BP

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

2.2.2.5 Регистр SP указателя начального адреса свободной памяти данных

Устанавливается программно. Начальное значение регистра равно нулю.

Структура регистра SP

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

2.2.2.6 Регистр RETV возвращаемого значения

Устанавливается программно. Начальное значение регистра равно нулю.

Структура регистра RETV

№№ битов	Обозначение	Описание
0 — 63		Значение

2.2.2.7 Регистр AOR смещения адреса исполняемой программы

Устанавливается программно. Начальное значение регистра равно нулю.

Структура регистра AOR

№№ битов	Обозначение	Описание
0 — 63		Значение

2.3 Коммутатор

Коммутатор используется для обмена результатами команд между командами внутри одного параграфа, а также между командами различных параграфов.

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.

3 Основные понятия языка

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.

3.2 Константы

В ассемблере поддерживаются числовые и символьные (литеральные) константы.

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.

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 на 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). mc-as-s1 использует ASCII кодировку символов (литералов): ´A имеет целочисленное значение 65, ´B имеет целочисленное значение 66, и так далее.

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

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.

3.4 Символы

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

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 подчеркивания («_») и символа точки («.»), при этом имя не может начинаться с цифры. Учитывается регистр букв в имени.

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.

3.4.3 Метки на результат команды

Метка на результат команды определяется как символ, за которым следует «:=». Этот символ представляет результат выполнения какой-либо команды и может быть использован в других командах в качестве аргумента.

3.4.4 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.

3.4.5 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 неопределенной 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 директива 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.

3.5 Выражения

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

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.

3.5.2 Целочисленные выражения

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

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.

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

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 аргументы 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.

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 или 96 бит. 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.

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»).

$V, V1, V2$ — 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

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 бит);
- 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 четыре байта (с 0 по 31 биты) представляют собой младшую часть (тип старшей и младшей 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 9 показаны мнемоники суффиксов типов операций.

Таблица 9: Мнемоника типов операций

Тип операции	**Обозначение**
Целый, размерностью один байт	b
Целый, размерностью два байта	s
Целый, размерностью четыре байта	l
Целый, размерностью восемь байтов	q
Вещественный одинарной точности, размерностью четыре байта	f
Вещественный двойной точности, размерностью восемь байтов	d
16-ти разрядный целый упакованный, размерностью восемь байтов	ps
32-х разрядный целый упакованный, размерностью восемь байтов	pl

Вещественный одинарной точности упакованный, размерностью восемь байтов	pf

4.3 Общий принцип построения команд в ассемблере

Ассемблером поддерживается следующий синтаксис команд мультиклеточного процессора:

- синтаксис команд с двумя аргументами, который изображён на рисунке 1
- синтаксис команд с одним аргументом, который изображён на рисунке 2

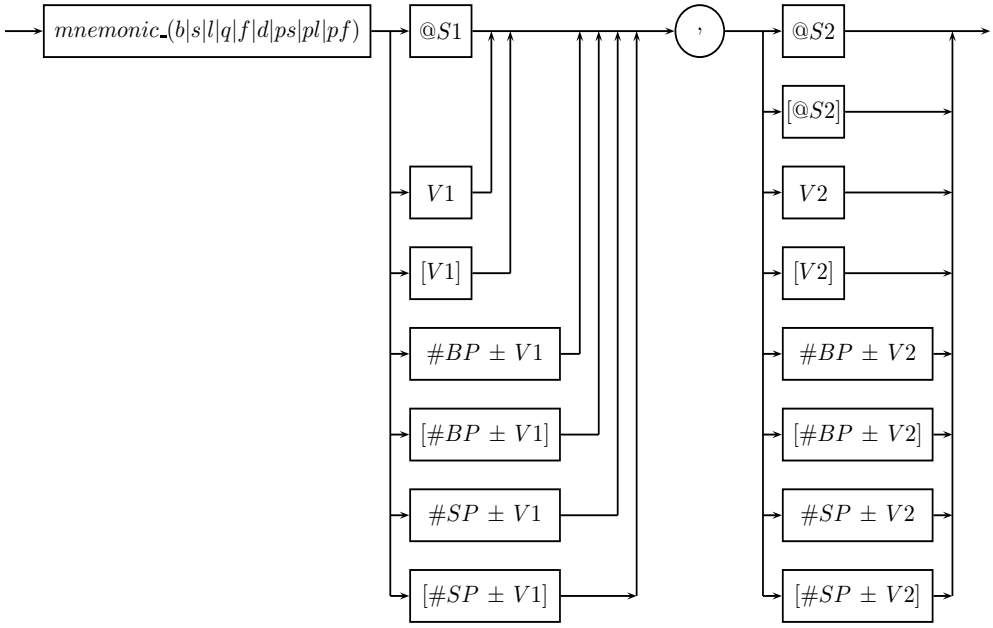

Рис. 1: Синтаксис команд с двумя аргументами мультиклеточного процессора

4.3.1 Общие правила формирования аргументов команд и их интерпретация

Согласно синтаксическому описанию (рис. 1 и 2) первый аргумент двухаргументной команды может быть задан с использованием одного из следующих вариантов:

- используется результат выполнения одной из 63-х предыдущих команд, содержащийся в коммутаторе и заданный ссылкой на это значение
- используется непосредственное 32-х разрядное значение $V1$, заданное в командном слове

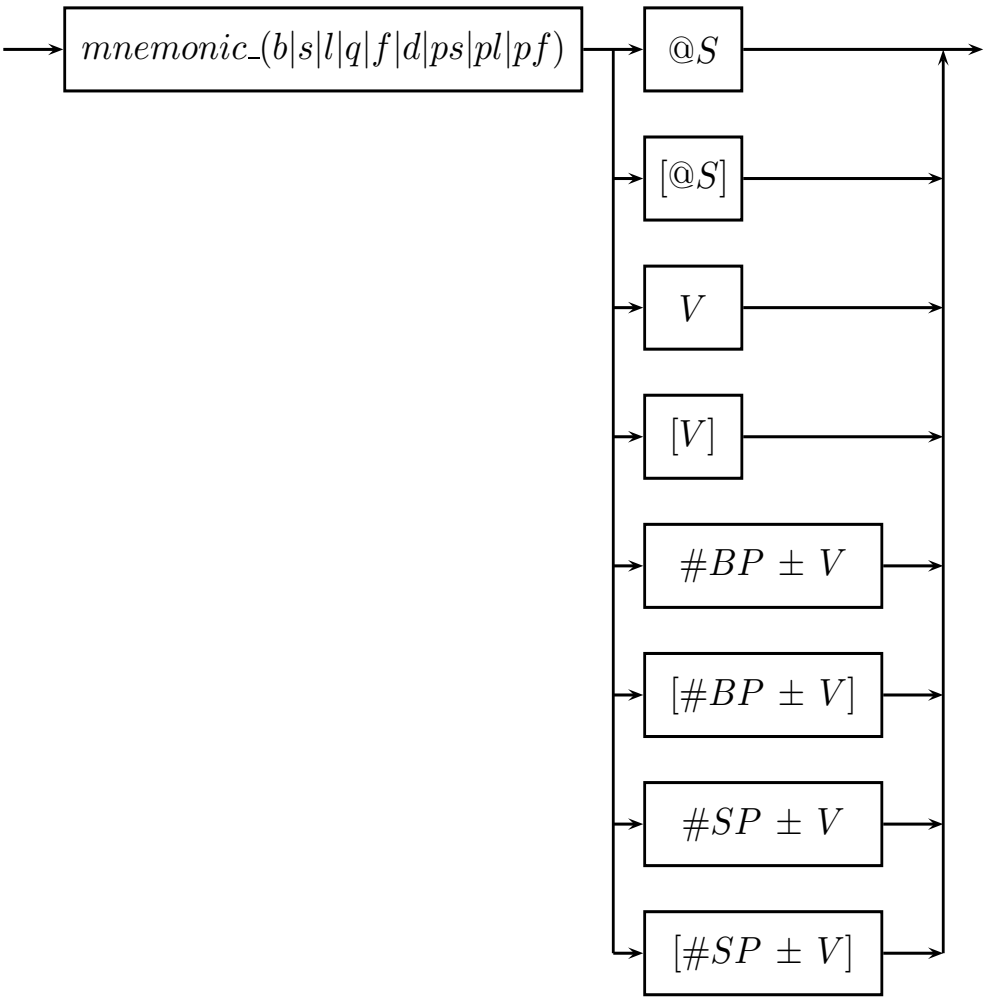

Рис. 2: Синтаксис команд с двумя аргументами мультиклеточного процессора

- используется значение, вычисляемое на основании значения регистра $\#BP$ и непосредственного 32-х разрядного значения $V1$, заданного в командном слове, согласно формуле

$$\#BP \pm V1$$

- используется значение, считанное из памяти данных по адресу, вычисляемому на основании значения регистра $\#BP$ и непосредственного 32-х разрядного значения $V1$, заданного в командном слове, согласно формуле

$$\#BP \pm V1$$

- используется значение, вычисляемое на основании значения регистра $\#SP$ и непосредственного 32-х разрядного значения $V1$, заданного в командном слове, согласно формуле

$$\#SP \pm V1$$

- используется значение, считанное из памяти данных по адресу, вычисляемому на основании значения регистра $\#SP$ и непосредственного 32-х разрядного значения $V1$, заданного в командном слове, согласно формуле

$$\#SP \pm V1$$

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

- используется результат выполнения одной из 63-х предыдущих команд, содержащийся в коммутаторе и заданный ссылкой на это значение
- 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 непосредственное 32-х разрядное значение $V2$, заданное в командном слове
- используется значение, вычисляемое на основании значения регистра $\#BP$ и непосредственного 32-х разрядного значения $V2$, заданного в командном слове, согласно формуле

$$\#BP \pm V2$$

- используется значение, считанное из памяти данных по адресу, вычисляемому на основании значения регистра $\#BP$ и непосредственного 32-х разрядного значения $V2$, заданного в командном слове, согласно формуле

$$\#BP \pm V2$$

- используется значение, вычисляемое на основании значения регистра $\#SP$ и непосредственного 32-х разрядного значения $V2$, заданного в командном слове, согласно формуле

$$\#SP \pm V2$$

- используется значение, считанное из памяти данных по адресу, вычисляемому на основании значения регистра $\#SP$ и непосредственного 32-х разрядного значения $V2$, заданного в командном слове, согласно формуле

$$\#SP \pm V2$$

В общем случае команды формируют результат, сохраняемый в коммутаторе.

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

4.4 Описание команд

4.4.1 abs (ABSolute value)

Абсолютное значение

abs ARG

Назначение: команда вычисления абсолютного значения аргумента

Синтаксис:

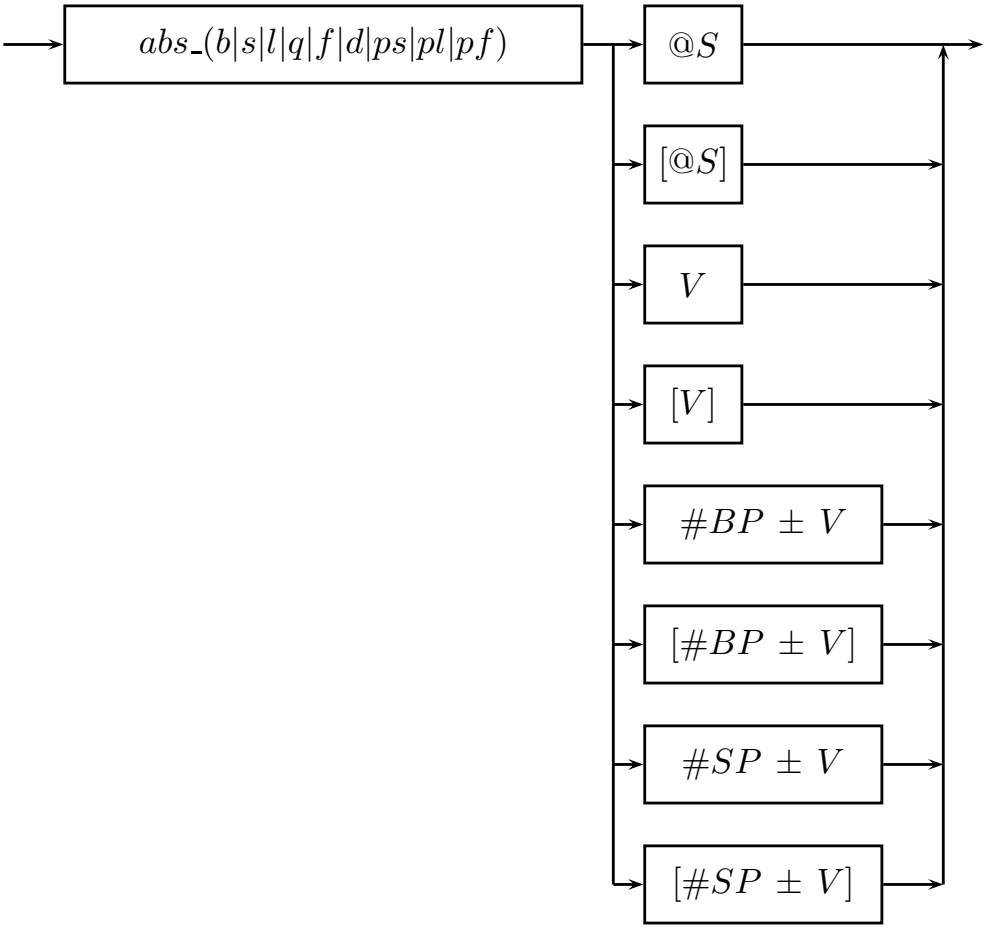

Рис. 3: Синтаксическое описание команды *abs*

Наличие результата: да

Алгоритм работы:

- вычислить абсолютное значение аргумента ARG;
- поместить результат в коммутатор;

Результат:

– для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;

– для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;

– для типов операции **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.

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.

Пример:

```
1  .text
2
3  A:
4        load_l 0x80000000
5        abs_l @1
6        abs_l 0xF0010203
7        abs_f 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 load_l помещается в коммутатор 32-х разрядное целое число (константа $0x80000000$);

– в строке №5 командой abs_l вычисляется абсолютное значение результата выполнения предшествующей команды: @1 — результат выполнения команды извлечения в строке №4; аргумент команды abs_l согласно суффиксу l интерпретируются как целое знаковое размерностью 4 байта; результат выполнения команды интерпретируются также как целое знаковое размерностью 4 байта и помещается в коммутатор;

– в строках №№6,7 показаны другие варианты использования команды abs;

– в строке №8 командой complete завершается текущий параграф.

4.4.2 add (ADDition)

Сложение

add $ARG1$, $ARG2$

Назначение: команда сложения двух аргументов

Синтаксис:

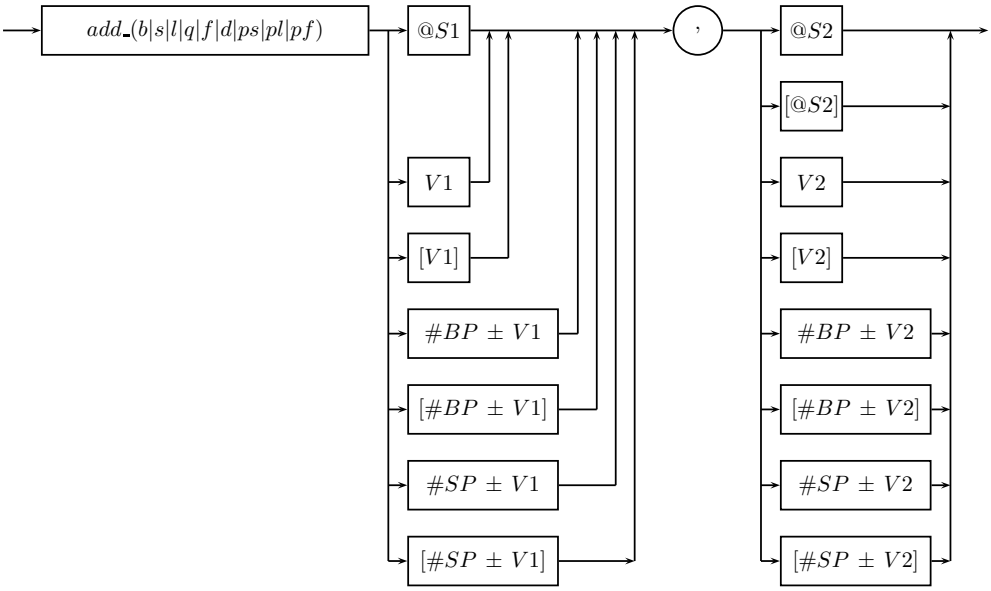

Рис. 4: Синтаксическое описание команды *add*

Наличие результата: да

Алгоритм работы:

- выполнить сложение $ARG1 + ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типов операции **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.

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

Пример:

```
1  .data
2
3  B:
4      .double\
5          0f12.8\
6          0f-1.78
7
8  .text
9
10 A:
11     load_q  [B]
12     load_q  [B + 8]
13     add_d  @1, @2
14     wr_q  @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 .double в текущую секцию ассемблирования записывается два 64-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 load_q, читаются из памяти данных два 64-х разрядных числа по адресам B и $B + 8$ соответственно и помещаются в коммутатор;

– в строке №13 командой add_d 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 add_d согласно суффиксу d интерпретируются как вещественные числа двойной точности размерностью 64 бита; результат выполнения команды также интерпретируются как вещественное число двойной точности размерностью 64 бита и помещается в коммутатор;

– в строке №14 командой wr_q 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.

4.4.3 and (AND)

Логического умножение

and $ARG1$, $ARG2$

Назначение: команда логического умножения двух аргументов

Синтаксис:

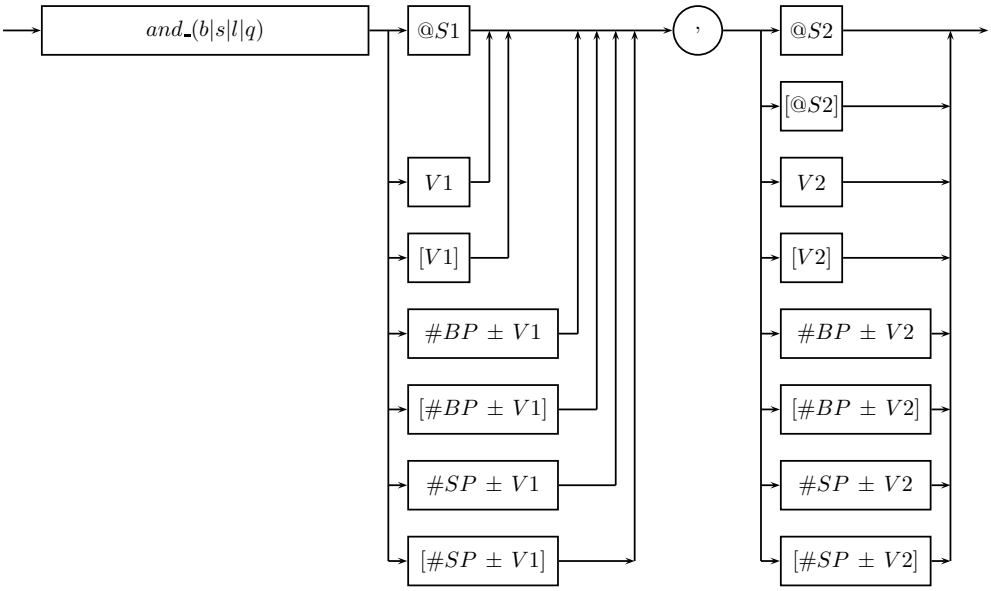

Рис. 5: Синтаксическое описание команды *and*

Наличие результата: да

Алгоритм работы:

- выполнить логическое умножение $ARG1$ & $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **quad** результат операции 64-х разрядный.

4.4.4 bc (Bit Count)

Вычисление числа битов, равных единице

bc ARG

Назначение: команда вычисления числа битов, равных единице

Синтаксис:

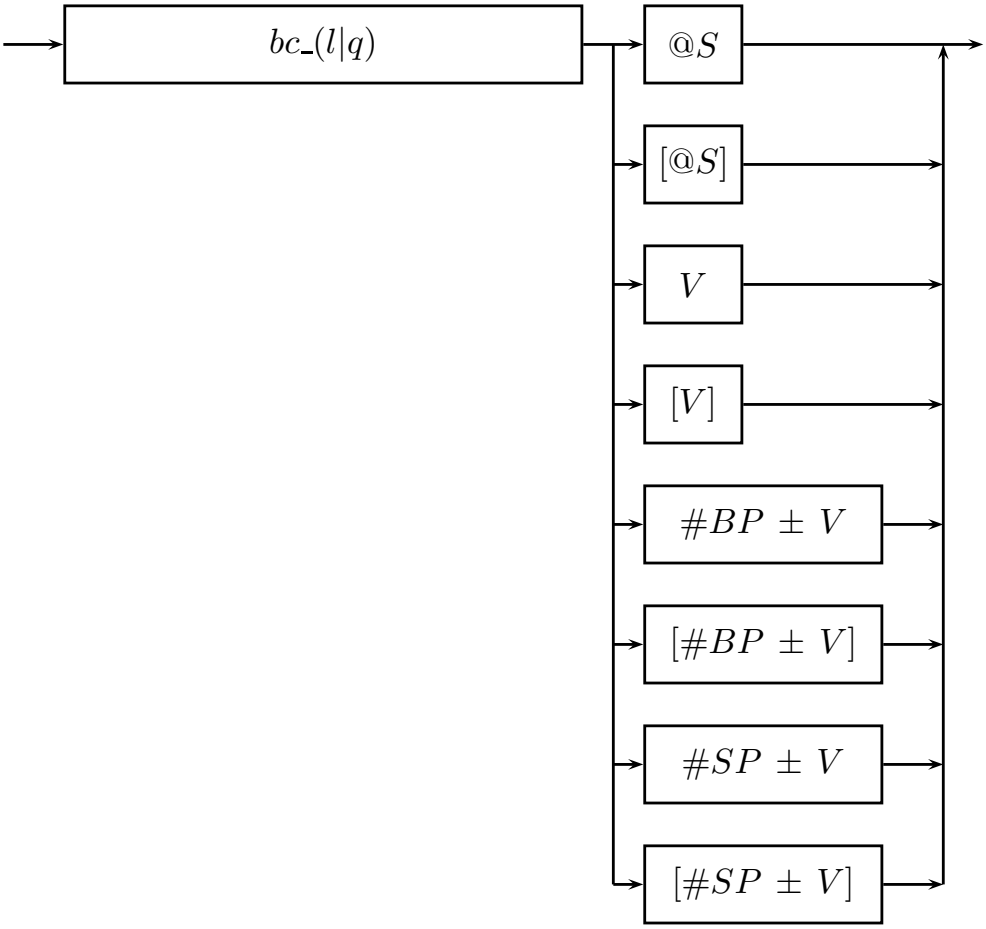

Рис. 6: Синтаксическое описание команды *bc*

Наличие результата: да

Алгоритм работы:

- вычислить число бит, равных единице;
- поместить результат 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.

4.4.5 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:

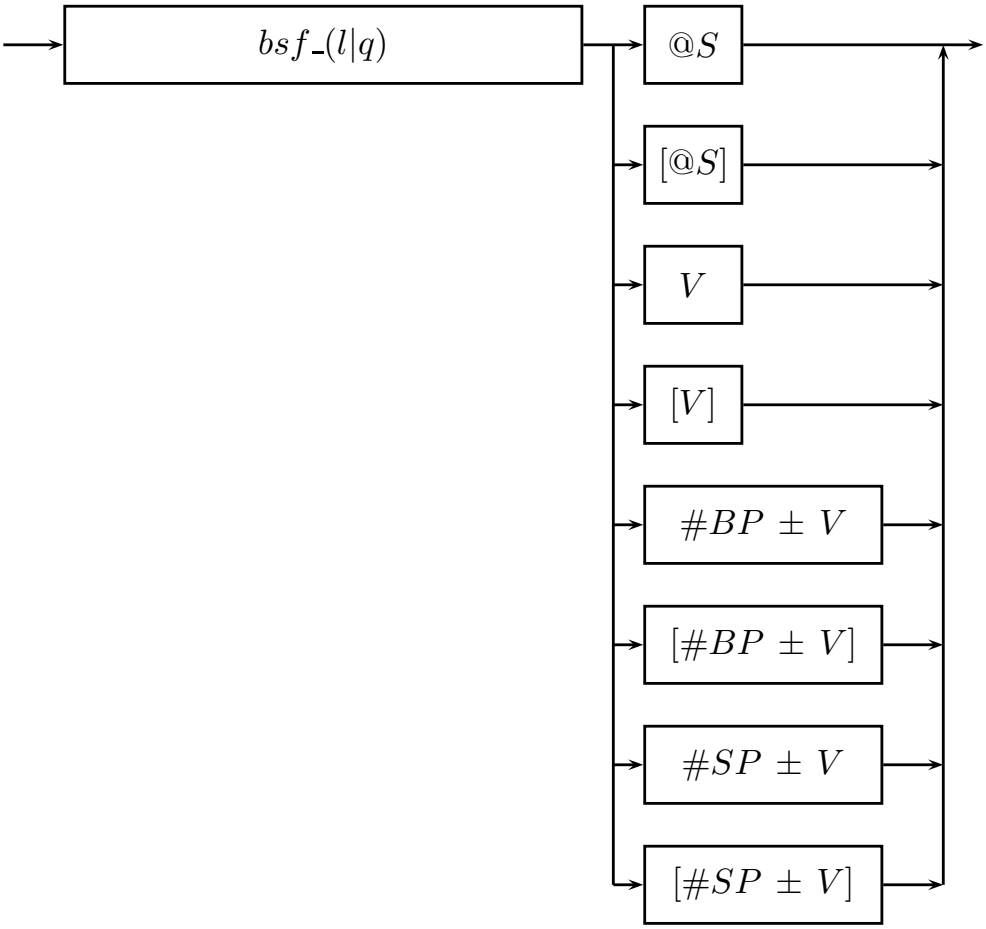

Рис. 7: Синтаксическое описание команды *bsf*

Наличие результата: да

Алгоритм работы:

– вычислить номер первого бита, равного единице, обнаруженного при сканировании аргумента в направлении от младшего бита к старшему;

– поместить результат 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.

4.4.6 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:

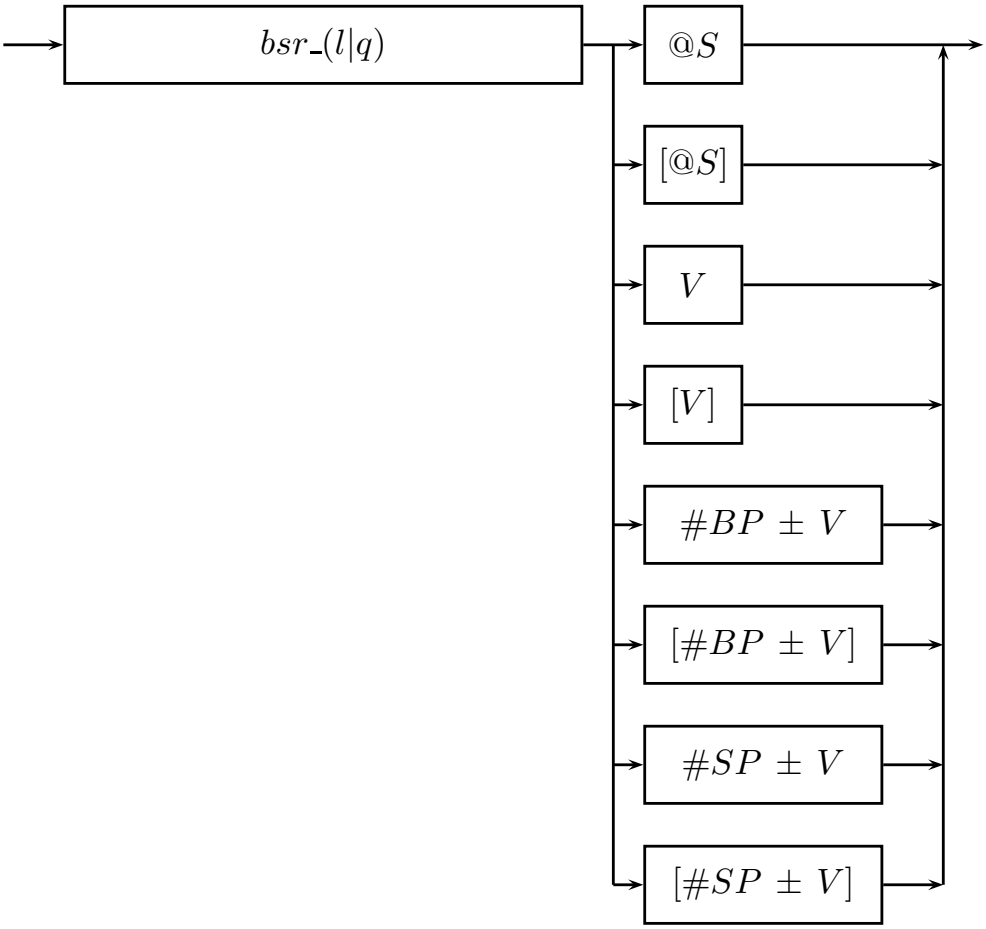

Рис. 8: Синтаксическое описание команды *bsr*

Наличие результата: да

Алгоритм работы:

- вычислить номер первого бита, равного единице, обнаруженного при сканировании аргумента в направлении от старшего бита к младшему;
- поместить результат 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.

4.4.7 bswap (Byte SWAP)

Перестановка байтов

bswap ARG

Назначение: команда перестановки байтов

Синтаксис:

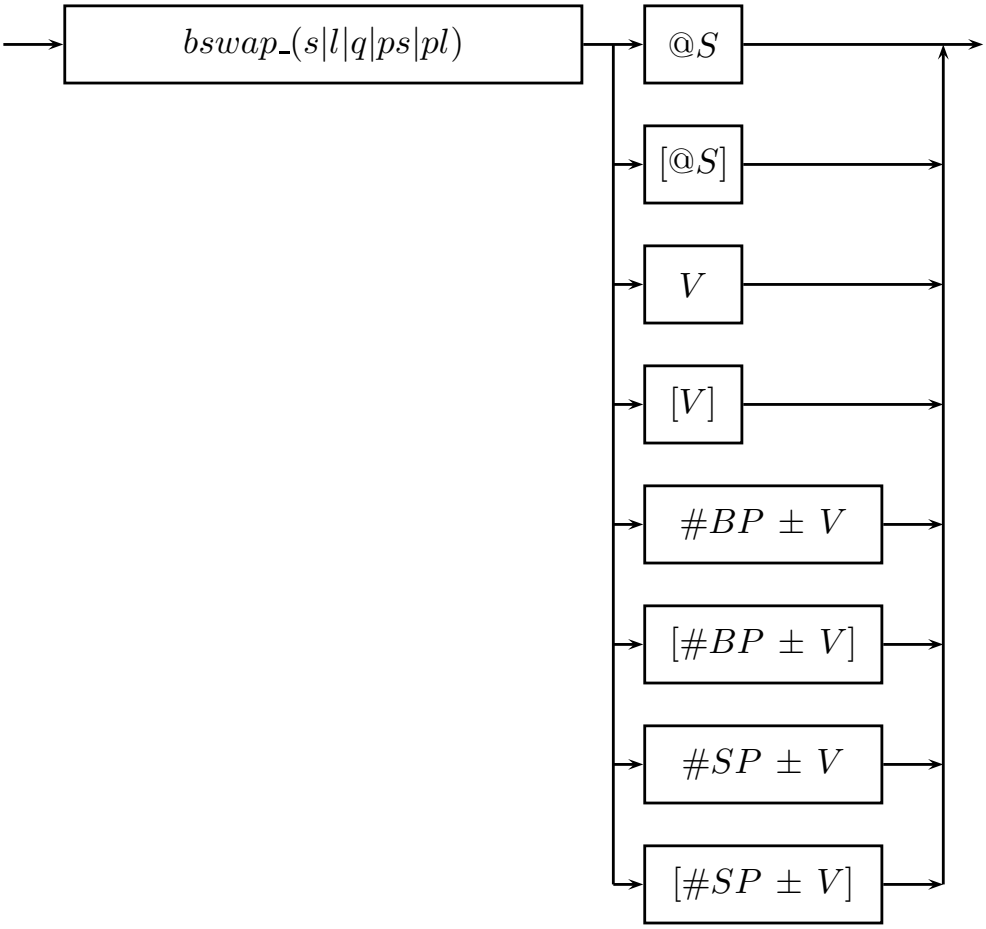

Рис. 9: Синтаксическое описание команды *bswap*

Наличие результата: да

Алгоритм работы:

– выполнить перестановку байтов аргумента согласно типу операции:

для типа операции **short**:

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

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**:

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

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

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

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

– поместить результат в коммутатор;

Результат:

– для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;

– 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.

4.4.8 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:

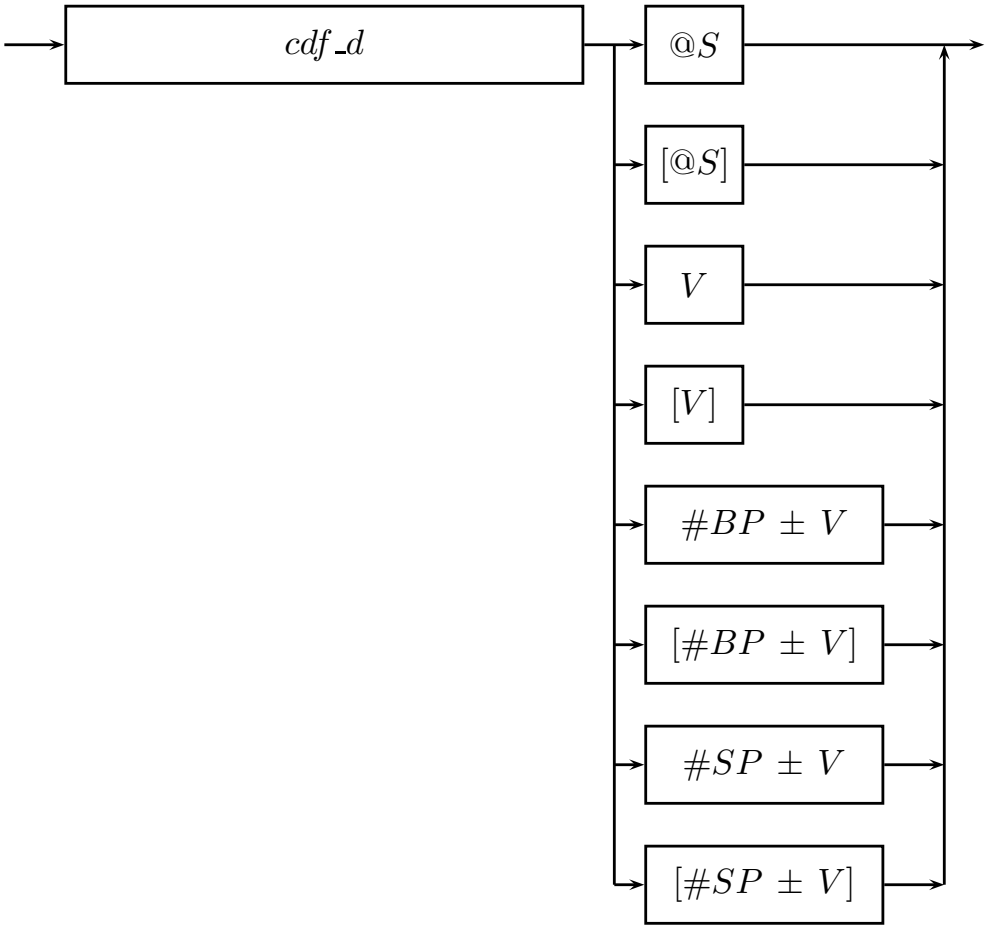

Рис. 10: Синтаксическое описание команды *cdf*

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;

– поместить результат в коммутатор;

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

4.4.9 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:

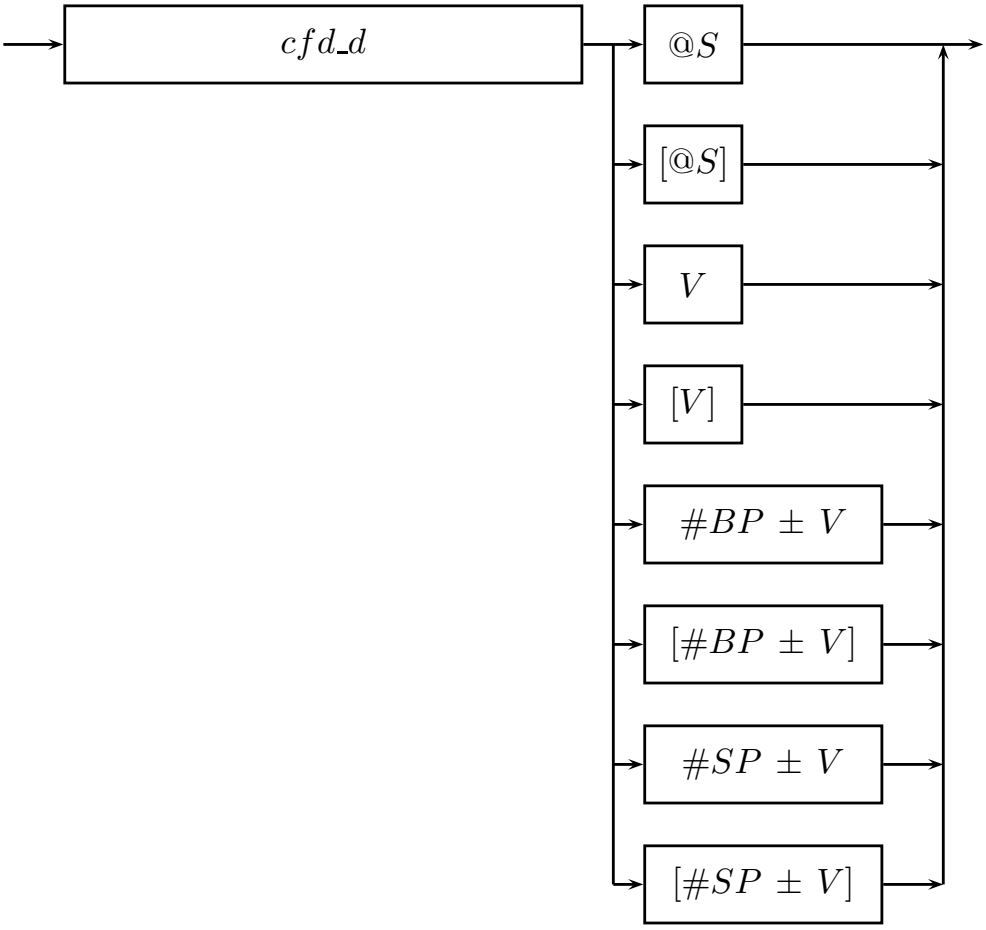

Рис. 11: Синтаксическое описание команды cfd

Наличие результата: да

Алгоритм работы:

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

Результат: результат операции 64-х разрядный.

4.4.10 cfi (Convert Float to Signed Integer)

Преобразование типа

cfi ARG

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

Синтаксис:

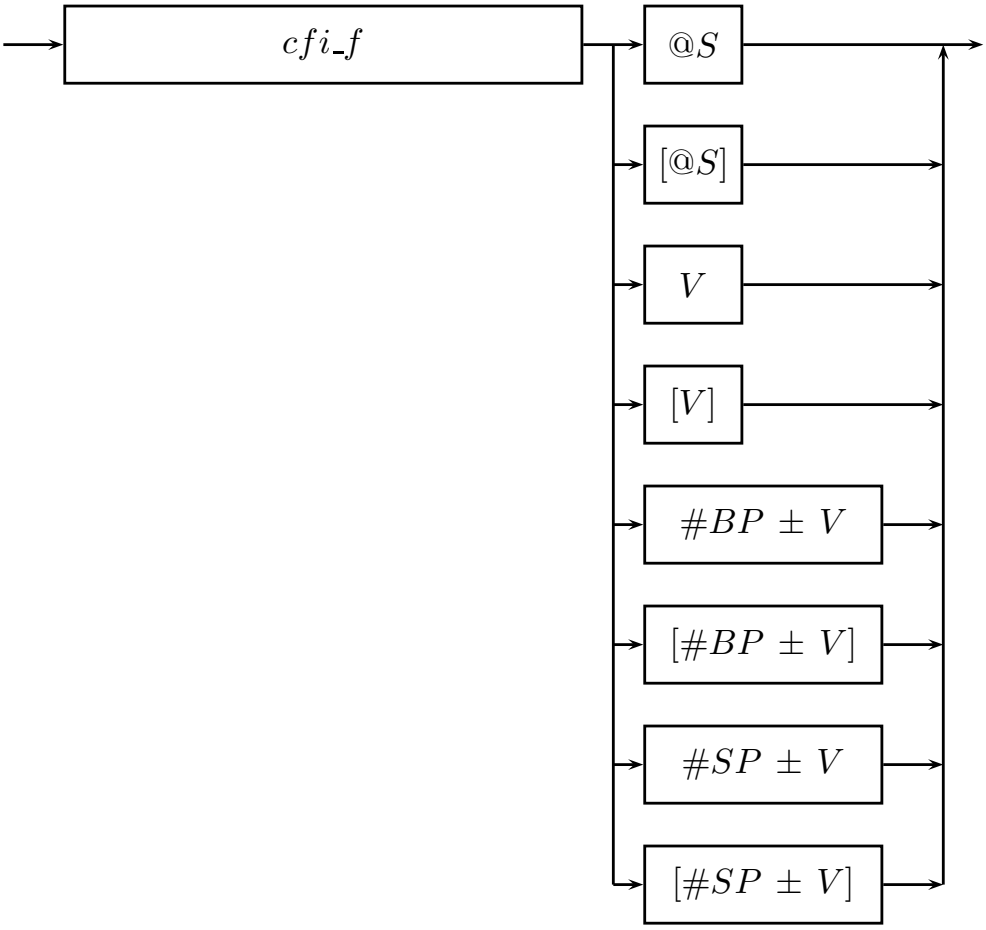

Рис. 12: Синтаксическое описание команды *cfi*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование числа с плавающей точкой одинарной точности в знаковое целое;
- поместить результат в коммутатор;

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

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

Пример:

```
1  .text
2
3  A:
4        cfi_f  1394.593
5        wr_l @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 cfi_f выполняется операция преобразования вещественного числа одинарной точности 1394.593 к целому знаковому 32-х разрядному числу;
- в строке №5 командой wr_l 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.

4.4.11 cif (Convert Signed Integer to Float)

Преобразование типа

cif ARG

Назначение: команда преобразования знакового целого в число с плавающей точкой

Синтаксис:

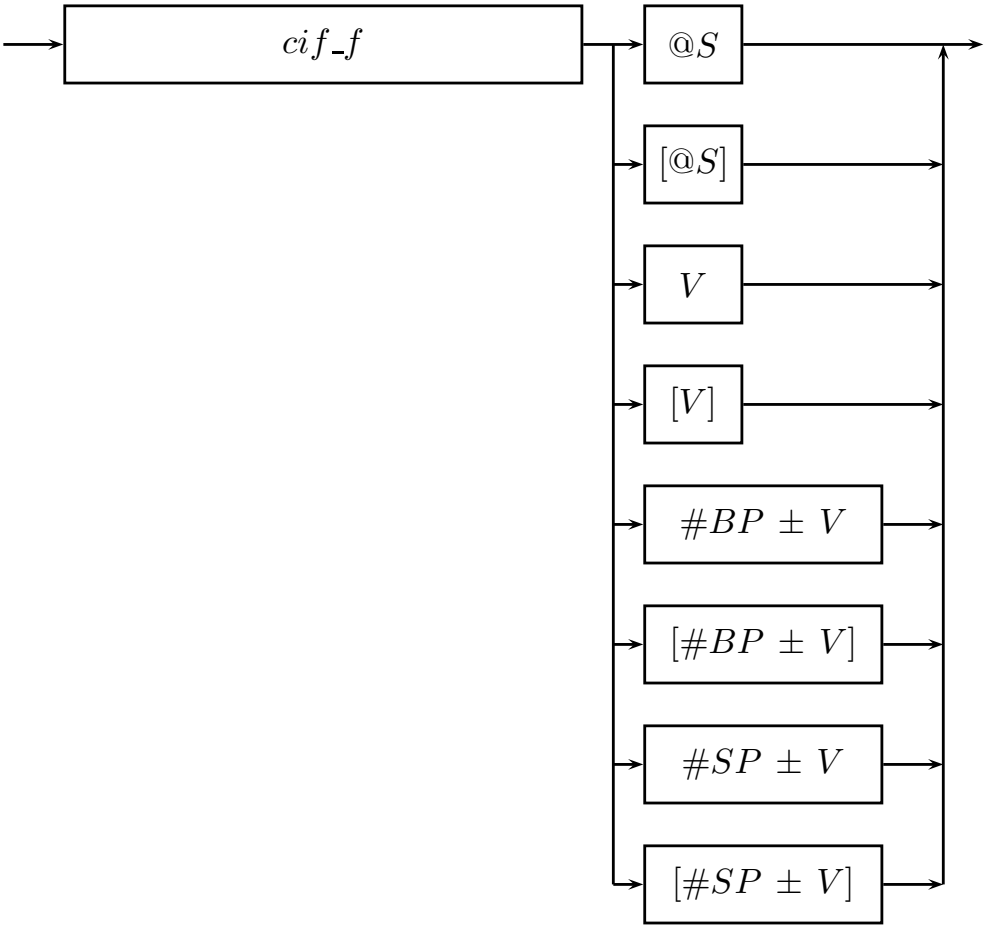

Рис. 13: Синтаксическое описание команды *cif*

Наличие результата: да

Алгоритм работы:

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

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

Применение: команда cif 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        cif_f  -1394
5        wr_l @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 cif_f выполняется операция преобразования целого знакового 32-х разрядного числа -1394 к вещественному числу одинарной точности;
- в строке №5 командой wr_l 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.

4.4.12 cmul (Complex Multiplication)

Комплексное умножение

mul $ARG1$, $ARG2$

Назначение: команда комплексного умножения двух аргументов

Синтаксис:

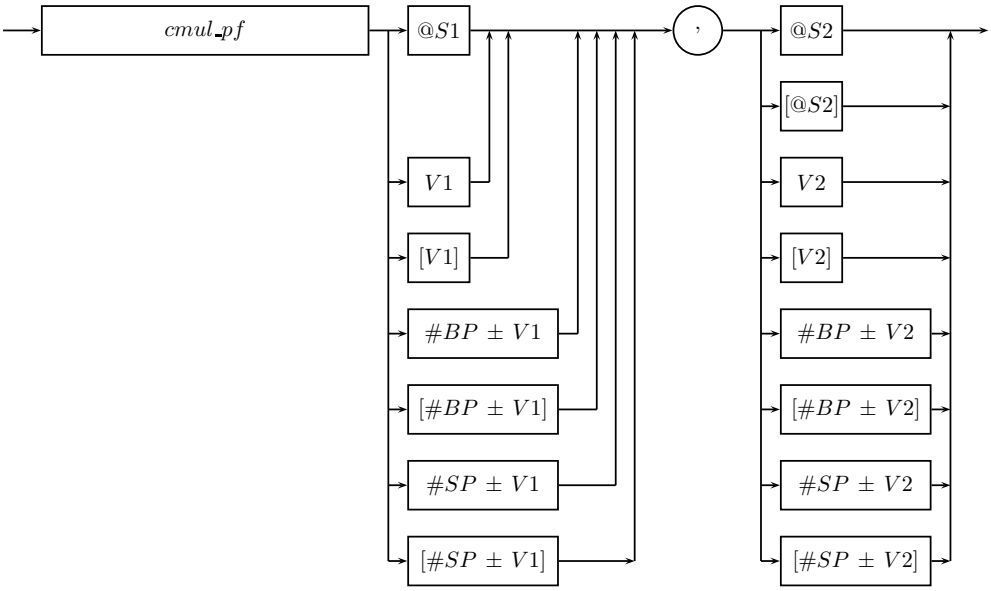

Рис. 14: Синтаксическое описание команды *cmul*

Наличие результата: да

Алгоритм работы:

- выполнить комплексное умножение $ARG1 * ARG2$;
- поместить результат в коммутатор;

Результат: результат операции 64-х разрядный.

Применение: команда cmul используется для умножения двух операндов, значение которых интерпретируется как комплексное число (действительная и мнимая части числа являются вещественными числами одинарной точности).

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      load_pl B
12      load_pl B + 8
13      cmul_pf @1, @2
14      wr_pl @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 load_pl, читаются из памяти данных два 64-х разрядных упакованных числа по адресам B и $B + 8$ соответственно и помещаются в коммутатор;
- в строке №13 командой cmul_pf выполняется операция комплексного 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 cmul_pf интерпретируются как комплексные числа размерностью 64 бита; результат выполнения команды также интерпретируются как 64-х разрядное комплексное число и помещается в коммутатор;
- в строке №14 командой wr_pl 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.

4.4.13 div (DIVision)

Деление

*div ARG*1, *ARG*2

Назначение: команда деления двух аргументов

Синтаксис:

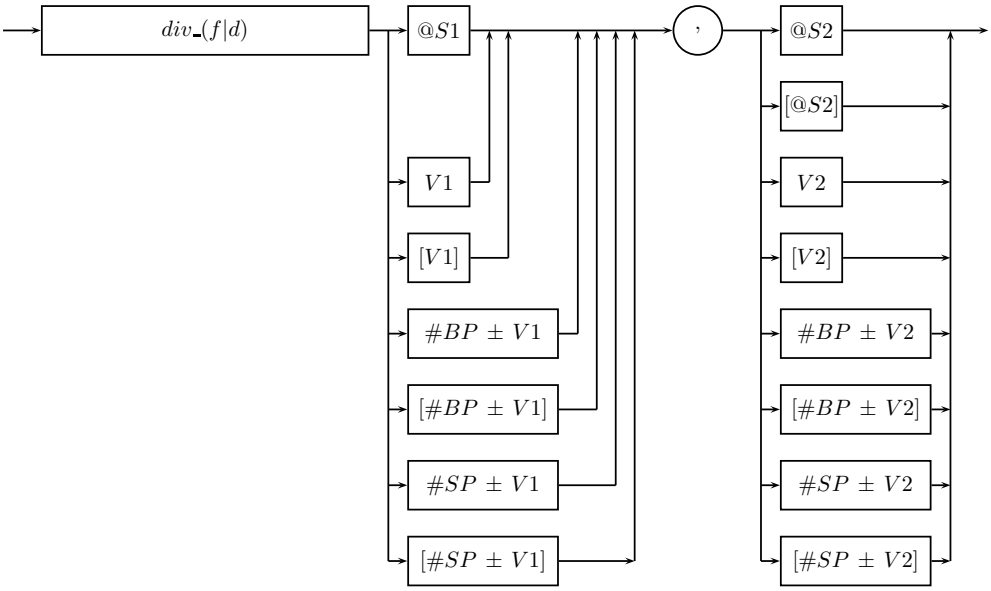

Рис. 15: Синтаксическое описание команды *div*

Наличие результата: да

Алгоритм работы:

- вычислить частное аргументов $ARG1/ARG2$;
- поместить результат 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.

Применение: команда div используется для деления двух операндов, значение которых интерпретируется согласно типу операции.

Пример:

```
1  .data
```

```
2
3  B:
4          .float  0f12.8e12 , 0f-5.6e-4
5
6  .text
7
8  A:
9          load_l B
10         load_l B + 4
11         div_f @1, @2
12         wr_l @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 load_l, 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 div_f 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 div_f согласно суффиксу f интерпретируются как вещественные числа одинарной точности; результат выполнения команды также интерпретируются как вещественное число одинарной точности и помещается в коммутатор;
- в строке №12 командой wr_l 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.

4.4.14 divrem (DIVision with REMinder)

Деление с остатком знаковое

fcpy ARG1, ARG2

Назначение: команда деления с остатком двух целых знаковых 32-х разрядных чисел.

Синтаксис:

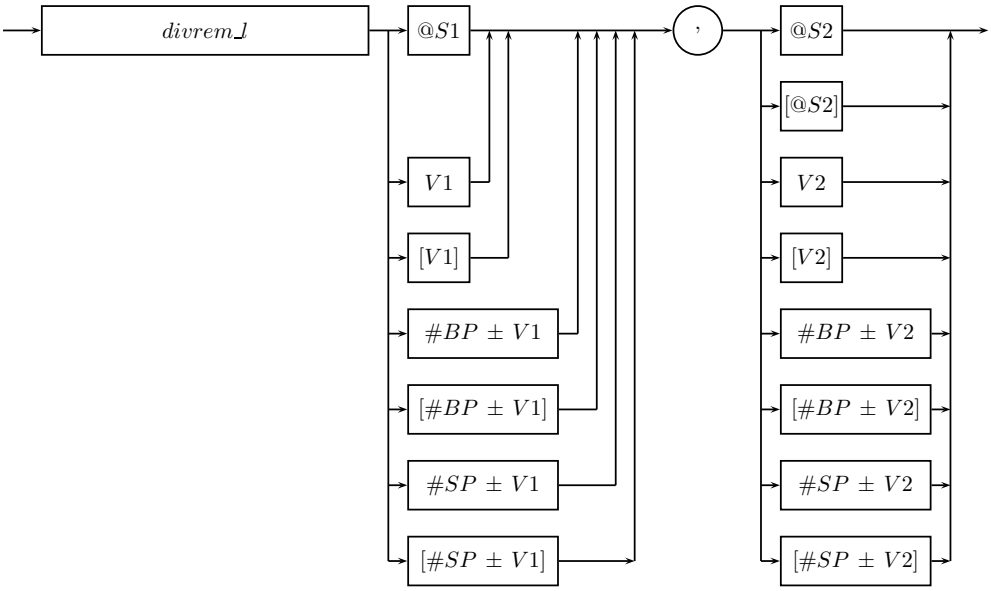

Рис. 16: Синтаксическое описание команды *divrem*

Наличие результата: да

Алгоритм работы:

- вычислить частное и остаток аргументов $ARG1/ARG2$;
- поместить результат в коммутатор;

Результат: результат операции 64-х разрядный: результат деления находится в младших 32-х разрядах, остаток - в старших.

4.4.15 divremu (DIVision with REMinder Unsigned)

Деление с остатком беззнаковое

fcpy $ARG1$, $ARG2$

Назначение: команда деления с остатком двух целых беззнаковых 32-х разрядных чисел.

Синтаксис:

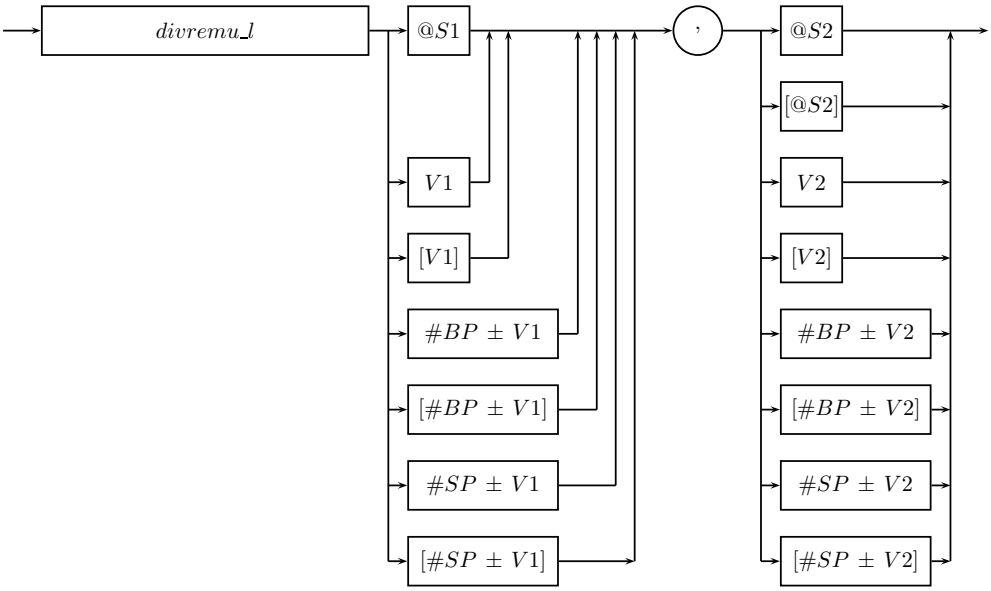

Рис. 17: Синтаксическое описание команды *divremu*

Наличие результата: да

Алгоритм работы:

- вычислить частное и остаток аргументов $ARG1/ARG2$;
- поместить результат в коммутатор;

Результат: результат операции 64-х разрядный: результат деления находится в младших 32-х разрядах, остаток - в старших.

4.4.16 dload (Direct LOAD)

Непосредственная загрузка (чтение) значения с размножением знака

dload ARG

Назначение: команда непосредственной загрузки (чтения) значения с размножением знака в коммутатор без контроля очерёдности доступа к памяти

Синтаксис:

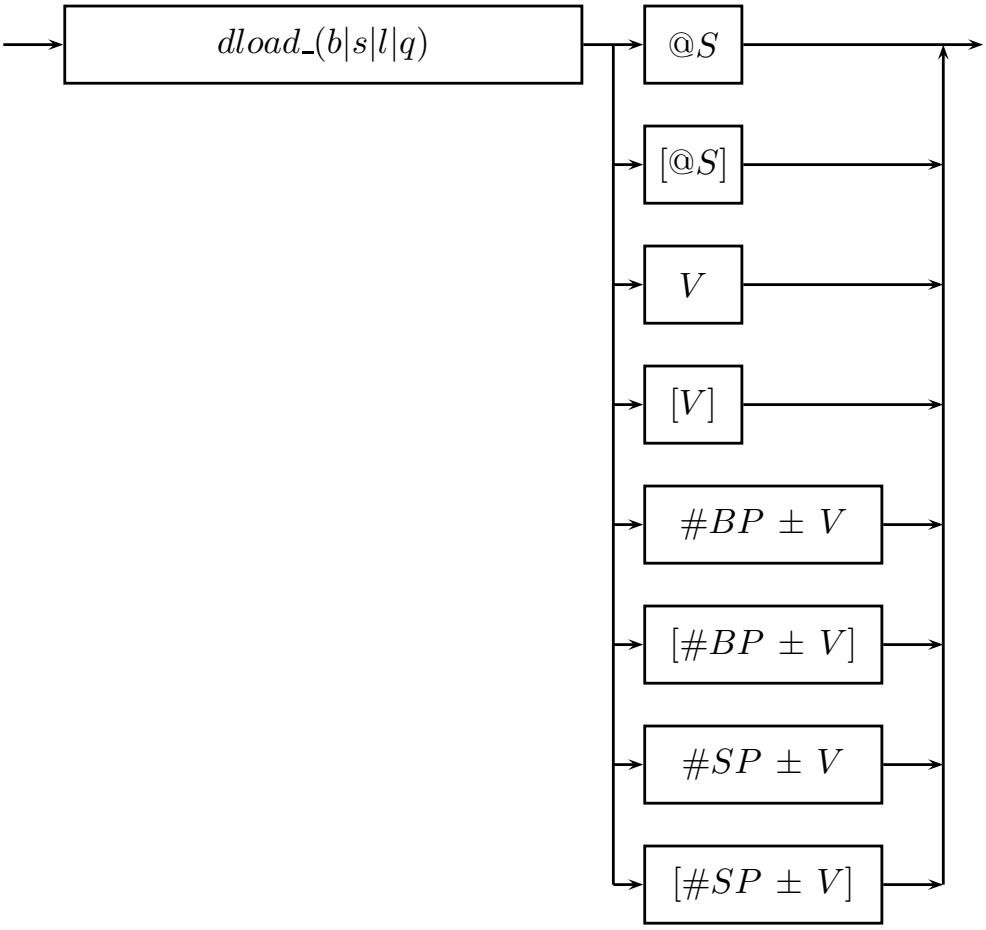

Рис. 18: Синтаксическое описание команды *dload*

Наличие результата: да

Алгоритм работы:

– безотлагательно загрузить в зависимости от типа и формата команды значение размером байт, полуслово, слово, двойное слово в коммутатор, заданному аргументом *ARG*;

– в случае загрузки байта (**byte**), полуслова (**short**), слова (**long**) размножить знак

до 63 разряда;

– поместить результат в коммутатор;

Результат:

– для типа операции **byte** результат операции 64-х разрядный, разряды с 8 по 63 заполняются значением 7 разряда (размножение знака);

– для типа операции **short** результат операции 64-х разрядный, разряды с 16 по 63 заполняются значением 15 разряда (размножение знака);

– для типа операций **long** результат операции 64-х разрядный, разряды с 32 по 63 заполняются значением 31 разряда (размножение знака);

– для типа операций **quad** результат операции 64-х разрядный.

4.4.17 dloadu (Direct LOAD Unsigned)

Непосредственная загрузка (чтение) значения без размножения знака

dloadu ARG

Назначение: команда непосредственной загрузки (чтения) значения без размножения знака в коммутатор без контроля очерёдности доступа к памяти

Синтаксис:

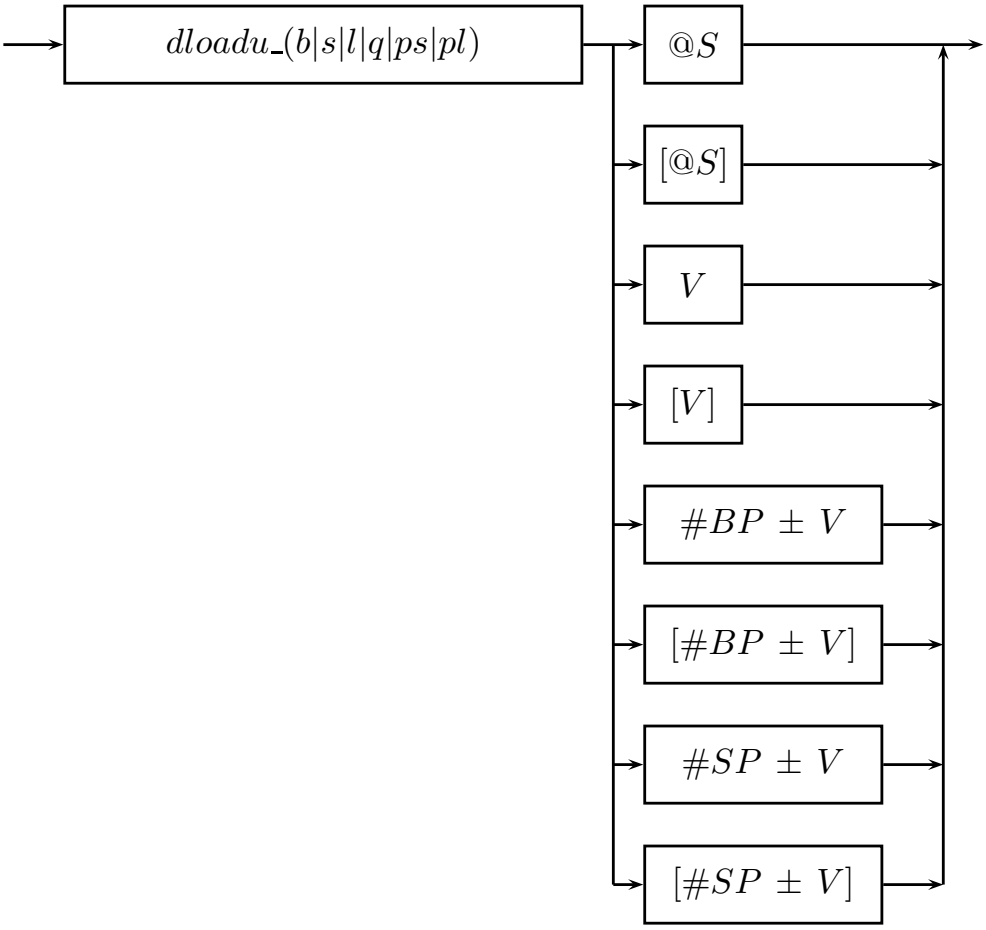

Рис. 19: Синтаксическое описание команды *dloadu*

Наличие результата: да

Алгоритм работы:

– безотлагательно загрузить в зависимости от типа и формата команды значение размером байт, полуслово, слово, двойное слово в коммутатор, заданному аргументом *ARG*;

– поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 64-х разрядный, разряды с 8 по 63 обнуляются;
- для типа операции **short** результат операции 64-х разрядный, разряды с 16 по 63 обнуляются;
- для типа операций **long** результат операции 64-х разрядный, разряды с 32 по 63 обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

4.4.18 dmod (Direct MODification)

Непосредственная модификация содержимого памяти данных

dmod $ARG1$, $ARG2$

Назначение: команда непосредственной модификации содержимого памяти данных без очерёдности доступа к памяти

Синтаксис:

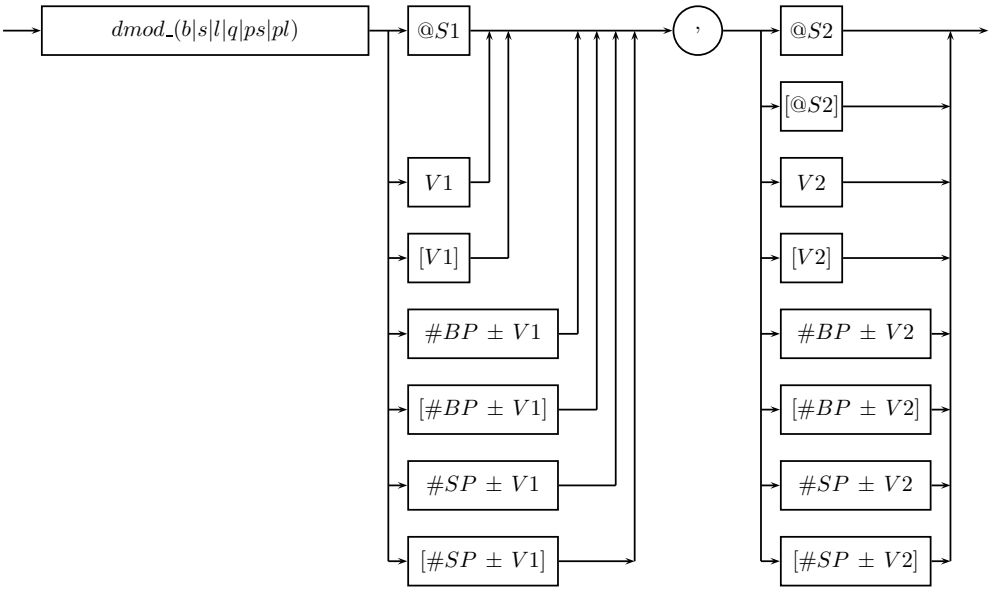

Рис. 20: Синтаксическое описание команды *dmod*

Наличие результата: нет

Алгоритм работы:

– изменить значение размером байт, полуслово, слово, двойное слово в памяти данных в зависимости от типа команды, согласно следующей формуле:
$DM(ARG2) := DM(ARG2) + ARG1$.

Интерпретация значения аргумента: значение аргумента $ARG2$ интерпретируется как адрес памяти, по которому осуществляется изменение значения. Если для формирования значения аргумента $ARG2$ используется результат предшествующей команды, т. е. используется ссылка на коммутатор ($@S$), то старшие 32 разряда (с 32 по 63) игнорируются.

4.4.19 dtas (Direct Test And Set)

Непосредственное чтение содержимого памяти данных с одновременной записью

dtas ARG

Назначение: команда непосредственного чтения содержимого памяти данных с одновременной записью без очерёдности доступа к памяти

Синтаксис:

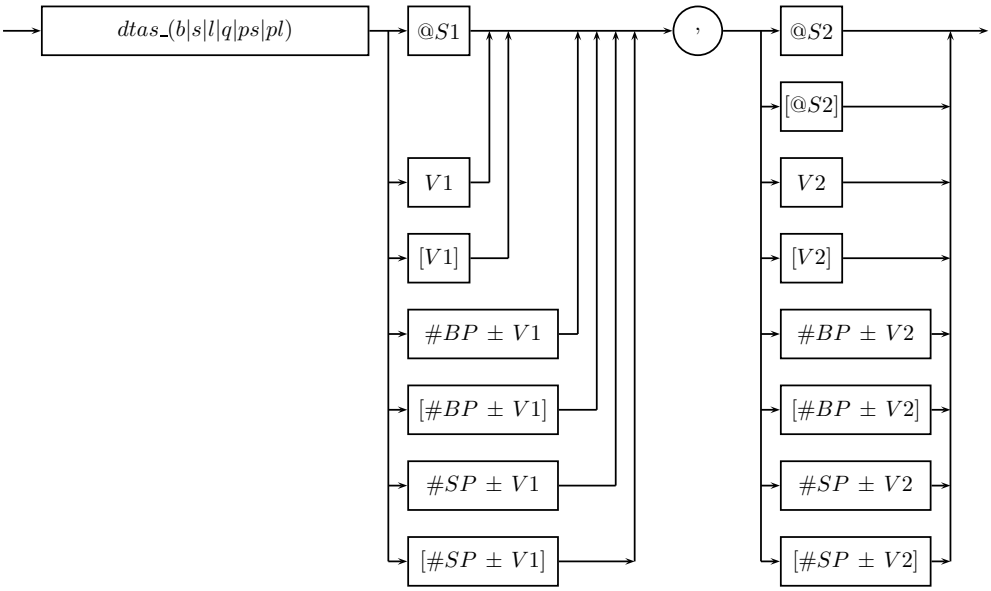

Рис. 21: Синтаксическое описание команды *dtas*

Наличие результата: да

Алгоритм работы:

- безотлагательно загрузить значение размером байт, полуслово, слово, двойное слово из памяти данных в зависимости от типа команды, заданному аргументом $ARG2$, с одновременной его установкой в памяти данных равным значению, заданным аргументом $ARG1$;
- поместить результат в коммутатор;

Интерпретация значения аргумента: согласно выше описанному алгоритму работы, значение аргумента $ARG2$ интерпретируется как адрес памяти, по которому осуществляется загрузкаи модификация значения, помещаемого в качестве результата данной команды в коммутатор. Если для формирования значения аргумента используется 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 64-х разрядный, разряды с 8 по 63 обнуляются;
- для типа операции **short** результат операции 64-х разрядный, разряды с 16 по 63 обнуляются;
- для типа операций **long** результат операции 64-х разрядный, разряды с 32 по 63 обнуляются;
- для остальных типов операций результат операции 64-х разрядный.

4.4.20 dwr (Direct WRite)

Запись

dwr $ARG1$, $ARG2$

Назначение: команда записи значения в память без очерёдности доступа к памяти

Синтаксис:

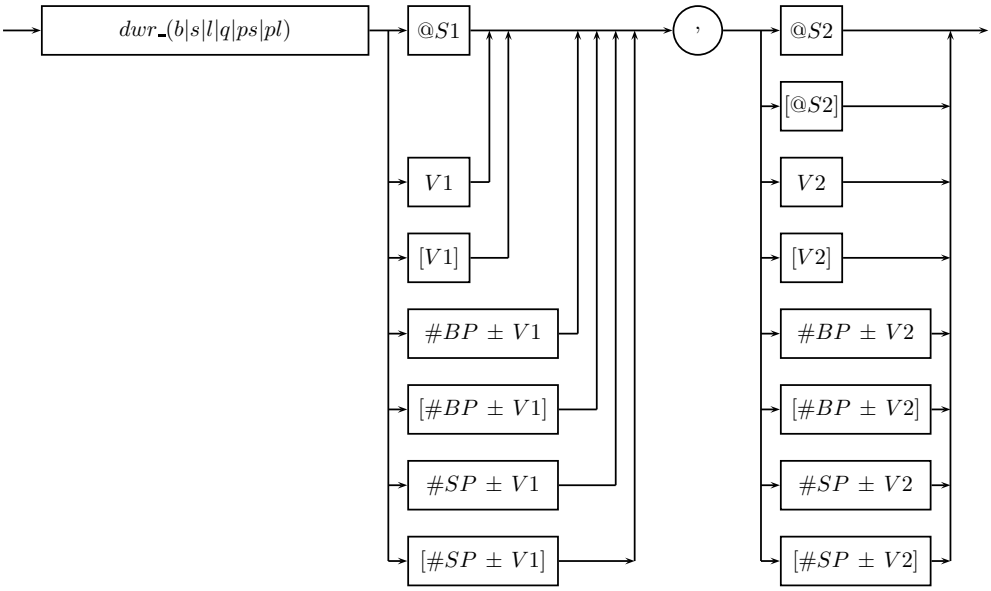

Рис. 22: Синтаксическое описание команды *dwr*

Наличие результата: нет

Алгоритм работы:

– безотлогательно записать в зависимости от типа команды значение размером байт, полуслово, слово, 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 dwr 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 результат предшествующей

команды, т. е. используется ссылка на коммутатор ($@S$), то старшие 32 разряда (с 32 по 63) игнорируются.

4.4.21 expb (EXPand Bytes)

Развёртка байтов

expb ARG

Назначение: команда развёртки байтов

Синтаксис:

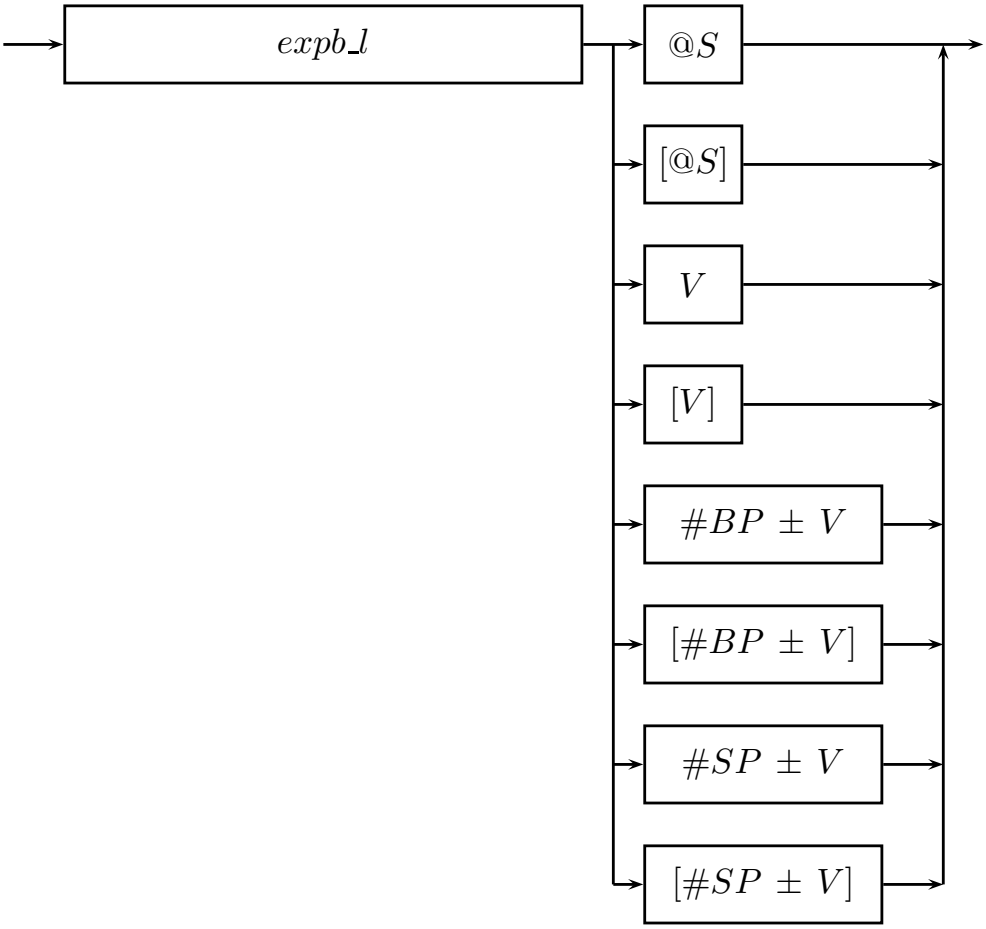

Рис. 23: Синтаксическое описание команды *expb*

Наличие результата: да

Алгоритм работы:

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

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

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

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

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

– поместить результат в коммутатор;

Результат: результат операции 64-х разрядный.

4.4.22 expbs (EXPand Bytes Signed)

Развёртка байтов с размножением знака

expbs ARG

Назначение: команда развёртки байтов с размножением знака

Синтаксис:

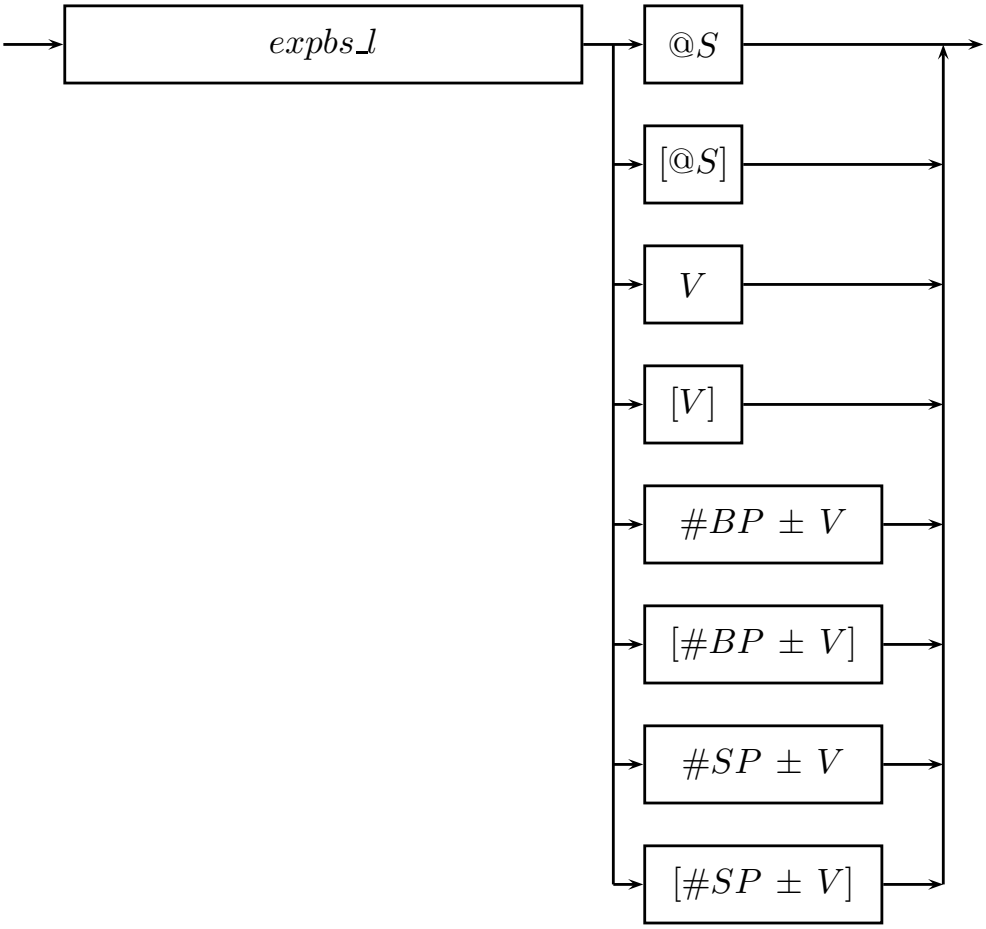

Рис. 24: Синтаксическое описание команды *expbs*

Наличие результата: да

Алгоритм работы:

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

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

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

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

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

– поместить результат в коммутатор;

Результат: результат операции 64-х разрядный.

4.4.23 ge (Greater or Equal)

Проверка условия «больше или равно» знаковых чисел.

ge $ARG1$, $ARG2$

Назначение: команда сравнения двух знаковых аргументов по критерию большего или равного

Синтаксис:

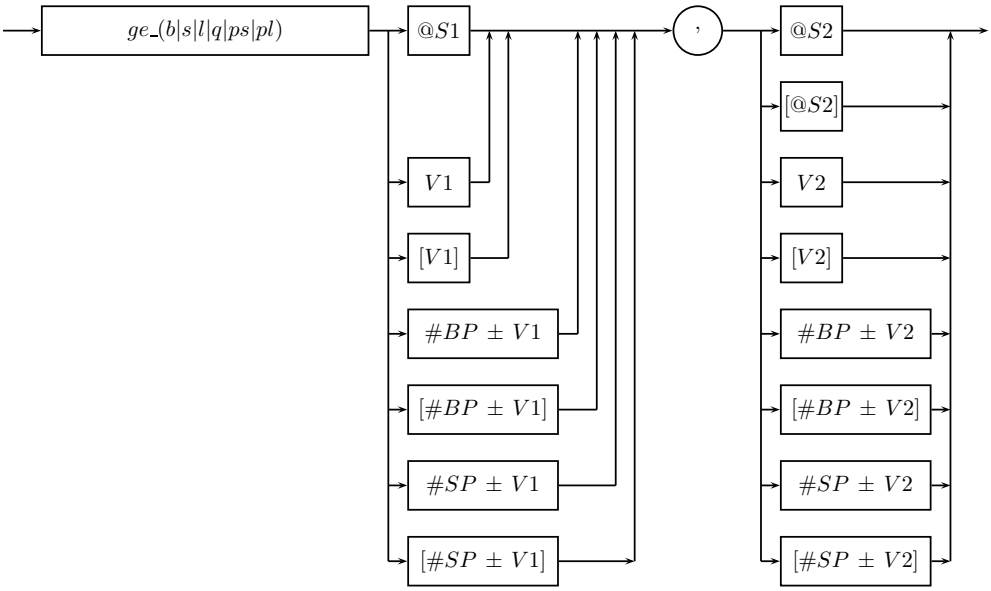

Рис. 25: Синтаксическое описание команды *ge*

Наличие результата: да

Алгоритм работы:

- выполнить команду сравнения двух знаковых аргументов, в качестве результата выдать «1», если $ARG1$ больше или равен $ARG2$, иначе выдать «0»;
- поместить результат в коммутатор;

Результат:

- для типов операций **byte, short, long, quad** результат операции одноразрядный, старшие 63 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **packed short** результат операции 64-х разрядный (0, 16, 32, 48 разряды либо «1», либо «0» в зависимости от аргументов, остальные разряды - обнуляются;

– для типа операции **packed long** результат операции 64-х разрядный (0, 32 разряды либо «1», либо «0» в зависимости отаргументов, остальные разряды - обнуляются.

4.4.24 getrg (GET ReGister value)

Извлечение значения регистра в коммутатор

getrg ARG

Назначение: команда извлечения значения регистра в коммутатор

Синтаксис:

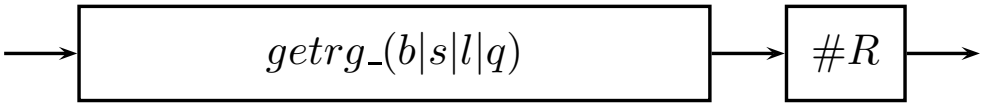

Рис. 26: Синтаксическое описание команды *getrg*

Наличие результата: да

Алгоритм работы:

- извлечь значение, заданное регистром R;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, разряды с 8 по 63 обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, разряды с 16 по 63 обнуляются;
- для типа операций **long** результат операции 32-х разрядный, разряды с 32 по 63 обнуляются;;
- для типа операции **quad** результат операции 64-х разрядный.

Пример:

```
1  .text
2
3  A:
4        getrg_l #PSW
5        wr_l @1, 0x100
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 getrg_l извлекается из регистра PSW 32-х разрядное значение и помещается в коммутатор;

– в строке №5 командой wr_l сохраняется в память данных по адресу $0x100$ ранее сохранённое в коммутаторе значение, т. е. сохраняется результат выполнения предшествующей команды: @1 — результат выполнения команды извлечения значения регистра в строке №4;

– в строке №6 командой complete завершается текущий параграф.

4.4.25 geu (Greater or Equal Unsigned)

Проверка условия «больше или равно» беззнаковых чисел.

geu $ARG1$, $ARG2$

Назначение: команда сравнения двух беззнаковых аргументов по критерию большего или равного

Синтаксис:

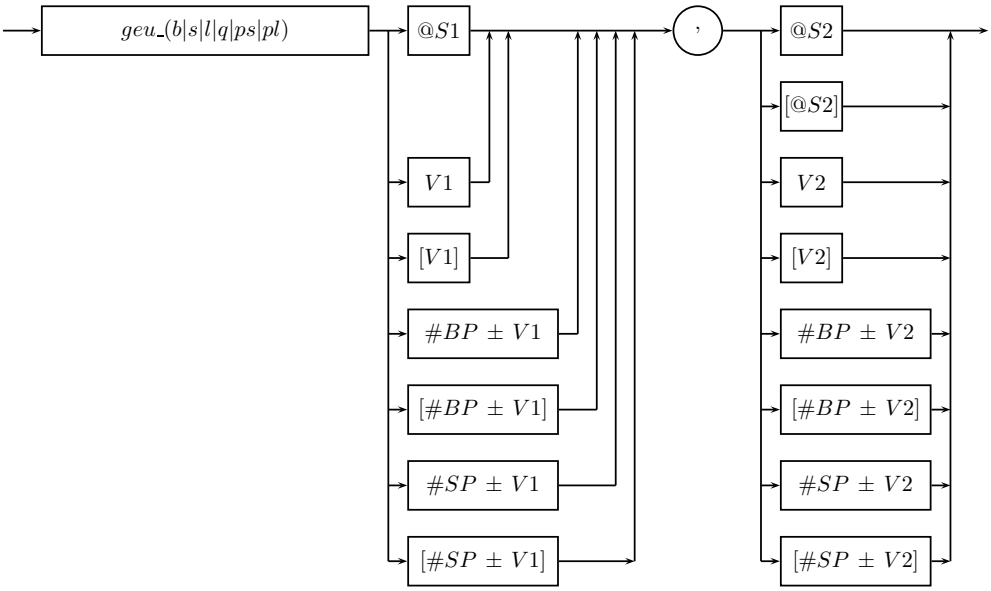

Рис. 27: Синтаксическое описание команды *geu*

Наличие результата: да

Алгоритм работы:

- выполнить команду сравнения двух знаковых аргументов, в качестве результата выдать «1», если $ARG1$ больше или равен $ARG2$, иначе выдать «0»;
- поместить результат в коммутатор;

Результат:

- для типов операций **byte, short, long, quad** результат операции одноразрядный, старшие 63 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **packed short** результат операции 64-х разрядный (0, 16, 32, 48 разряды либо «1», либо «0» в зависимости от аргументов, остальные разряды - обнуляются;

– для типа операции **packed long** результат операции 64-х разрядный (0, 32 разряды либо «1», либо «0» в зависимости отаргументов, остальные разряды - обнуляются.

4.4.26 load (LOAD)

Загрузка (чтение) значения с размножением знака

load ARG

Назначение: команда загрузки (чтения) значения с размножением знака в коммутатор с контролем очерёдности доступа к памяти

Синтаксис:

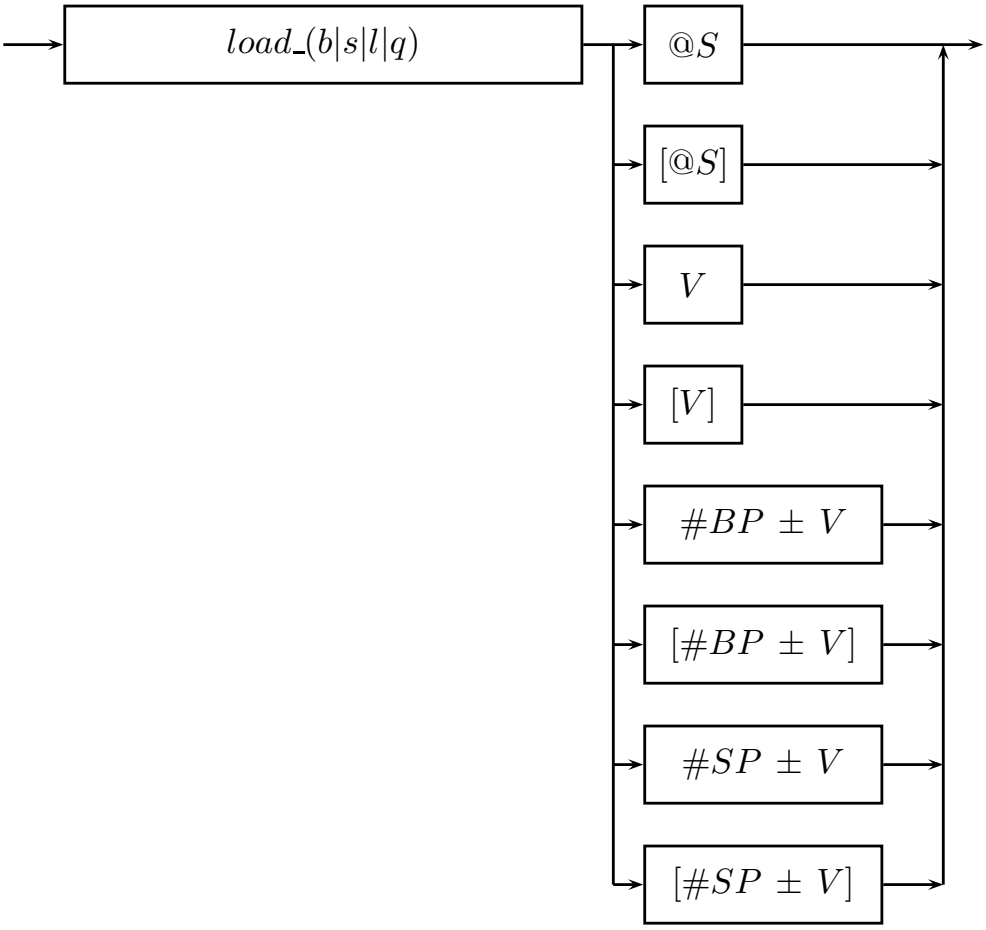

Рис. 28: Синтаксическое описание команды *load*

Наличие результата: да

Алгоритм работы:

– загрузить в зависимости от типа и формата команды значение размером байт, полуслово, слово, двойное слово в коммутатор, заданному аргументом *ARG*;

– в случае загрузки байта (**byte**), полуслова (**short**), слова (**long**) размножить знак до 63 разряда;

– поместить результат в коммутатор;

Результат:

– для типа операции **byte** результат операции 64-х разрядный, разряды с 8 по 63 заполняются значением 7 разряда (размножение знака);

– для типа операции **short** результат операции 64-х разрядный, разряды с 16 по 63 заполняются значением 15 разряда (размножение знака);

– для типа операций **long** результат операции 64-х разрядный, разряды с 32 по 63 заполняются значением 31 разряда (размножение знака);

– для типа операций **quad** результат операции 64-х разрядный.

Пример:

```
1  .data
2
3  A:
4      .long 1, 2
5
6  .text
7
8  B:
9      load_l [A]
10     load_l [A + 4]
11     add_l @1, @2
12     wr_l @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 load_l, загружаются в коммутатор, считанные из памяти данных по адресам A и $A + 4$ соответственно, два 32-х разрядных целых беззнаковых числа;

– в строке №11 командой add_l складываются результаты выполнения двух предшествующих команд: @1 — результат выполнения команды загрузки значения в строке №10, @2 — результат выполнения команды загрузки значения в строке №9; оба аргумента команды add_l 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 wr_l 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.

4.4.27 loadu (LOAD Unsigned)

Загрузка (чтение) значения без размножения знака

loadu ARG

Назначение: команда загрузки (чтения) значения без размножения знака в коммутатор с контролем очерёдности доступа к памяти

Синтаксис:

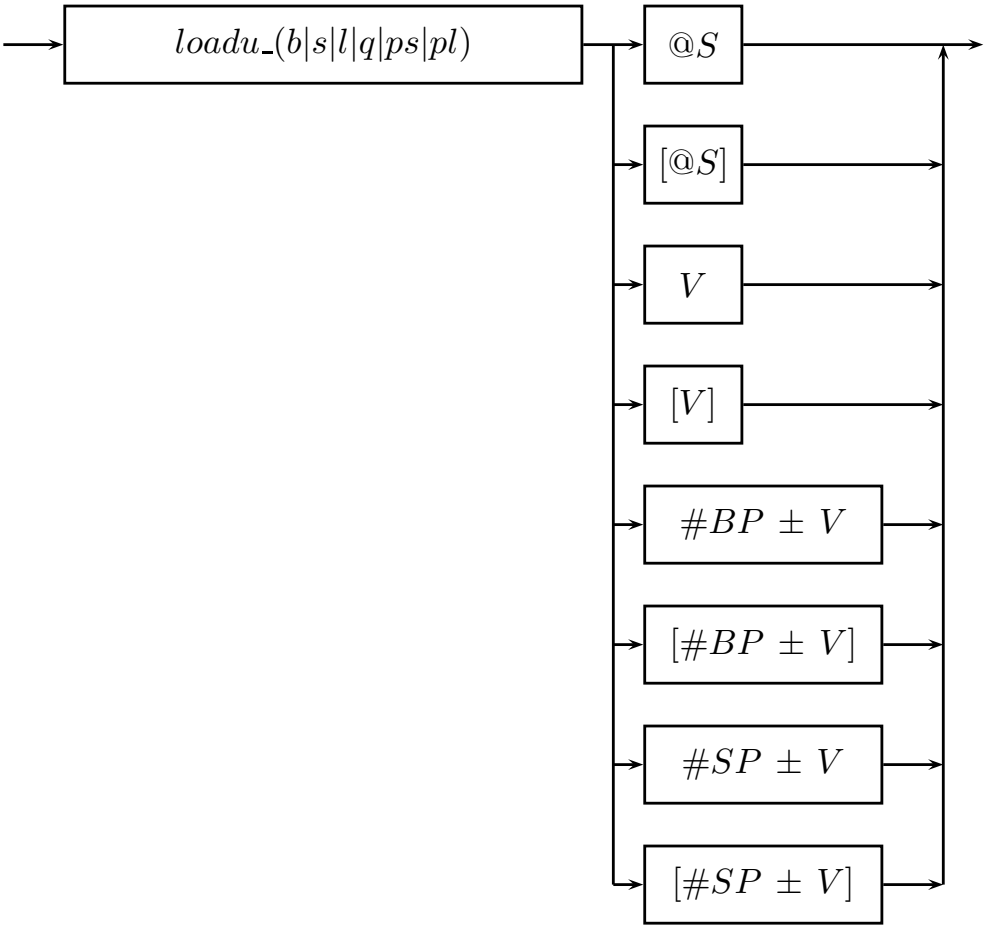

Рис. 29: Синтаксическое описание команды *loadu*

Наличие результата: да

Алгоритм работы:

- загрузить в зависимости от типа и формата команды значение размером байт, полуслово, слово, двойное слово в коммутатор, заданному аргументом *ARG*;
- поместить результат в коммутатор;

Результат:

– для типа операции **byte** результат операции 64-х разрядный, разряды с 8 по 63 обнуляются;

– для типа операции **short** результат операции 64-х разрядный, разряды с 16 по 63 обнуляются;

– для типа операций **long** результат операции 64-х разрядный, разряды с 32 по 63 обнуляются;

– для остальных типов операций результат операции 64-х разрядный.

4.4.28 lt (Less Than)

Проверка условия «меньше».

lt *ARG*1, *ARG*2

Назначение: команда сравнения двух знаковых аргументов по критерию меньшего

Синтаксис:

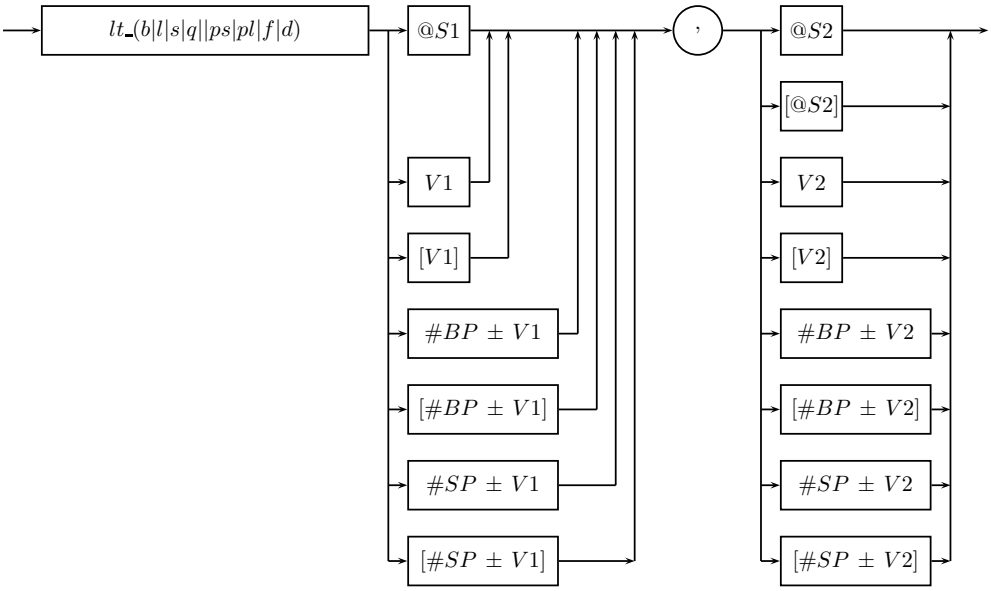

Рис. 30: Синтаксическое описание команды *lt*

Наличие результата: да

Алгоритм работы:

- выполнить команду сравнения двух знаковых аргументов, в качестве результата выдать «1», если *ARG*1 меньше, чем *ARG*2, иначе выдать «0»;
- поместить результат в коммутатор;

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

4.4.29 ltu (Less Than Unsigned)

Проверка условия «меньше».

ltu $ARG1$, $ARG2$

Назначение: команда сравнения двух беззнаковых аргументов по критерию меньшего

Синтаксис:

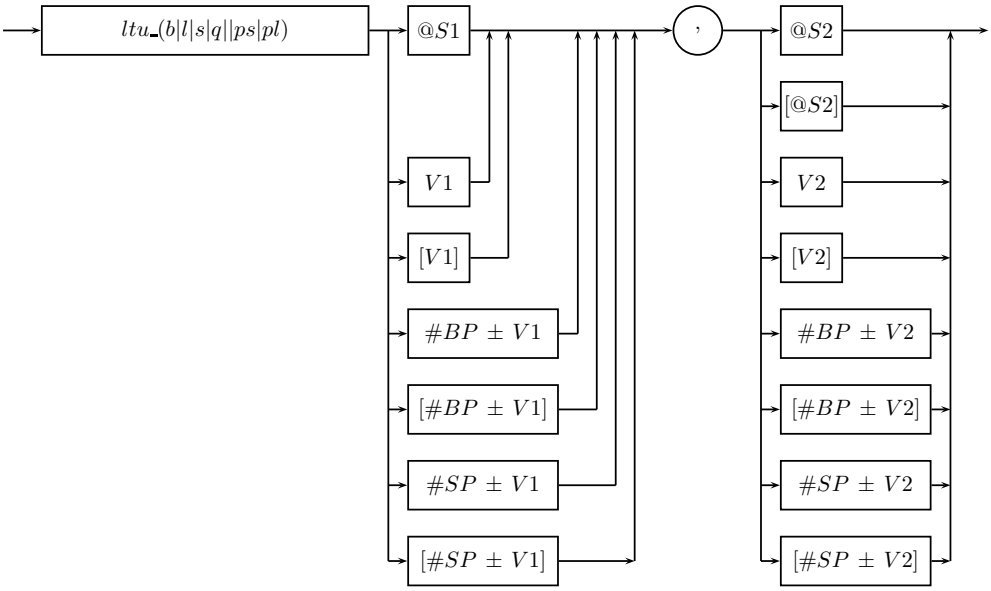

Рис. 31: Синтаксическое описание команды *ltu*

Наличие результата: да

Алгоритм работы:

- выполнить команду сравнения двух беззнаковых аргументов, в качестве результата выдать «1», если $ARG1$ меньше, чем $ARG2$, иначе выдать «0»;
- поместить результатв коммутатор;

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

4.4.30 max (MAXimum)

Выбор большего знакового числа.

max *ARG*1, *ARG*2

Назначение: команда выбора большего знакового числа из двух знаковых аргументов

Синтаксис:

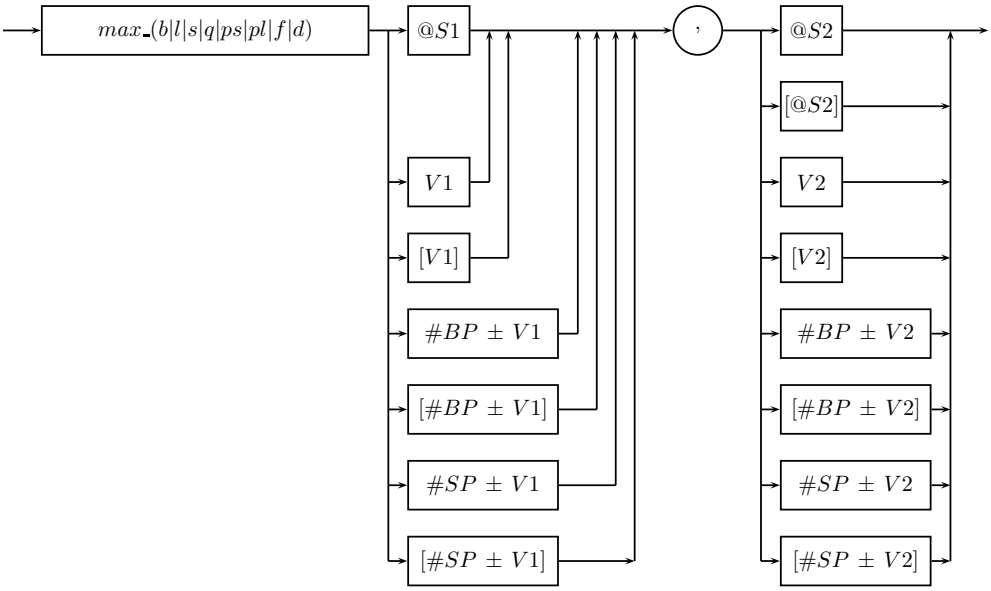

Рис. 32: Синтаксическое описание команды *max*

Наличие результата: да

Алгоритм работы:

- выполнить команду выбора большего знакового числа из двух знаковых аргументов, в качестве результата выдать *ARG*1, если *ARG*1 больше, чем *ARG*2, иначе выдать *ARG*2;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типов операции **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.

4.4.31 maxu (MAXimum Unsigned)

Выбор большего беззнакового числа.

maxu $ARG1$, $ARG2$

Назначение: команда выбора большего беззнакового числа из двух беззнаковых аргументов

Синтаксис:

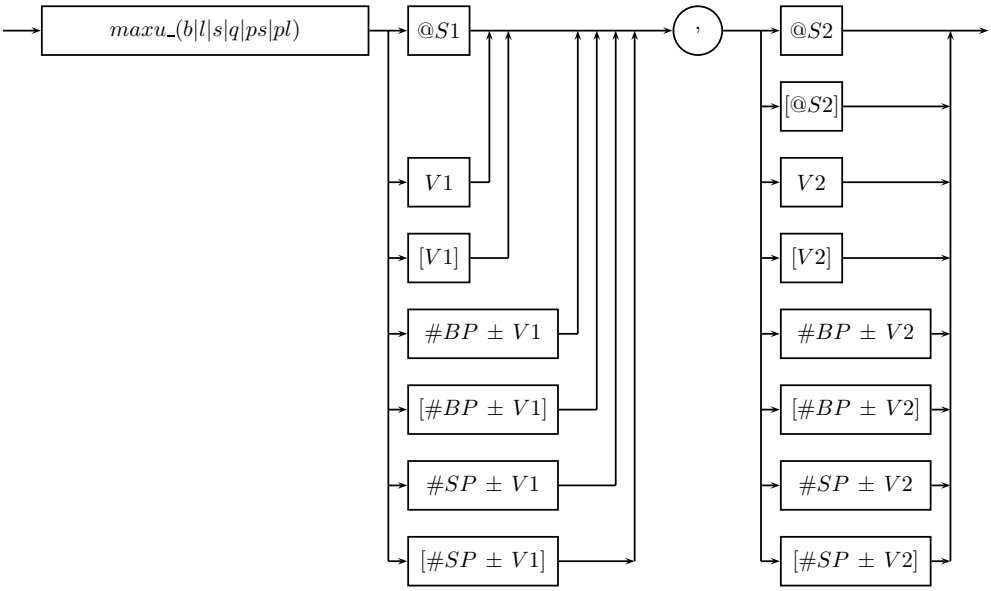

Рис. 33: Синтаксическое описание команды *maxu*

Наличие результата: да

Алгоритм работы:

- выполнить команду выбора большего беззнакового числа из двух беззнаковых аргументов, в качестве результата выдать $ARG1$, если $ARG1$ больше, чем $ARG2$, иначе выдать $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;

– для типов операции **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.

4.4.32 min (MINimum)

Выбор меньшего знакового числа.

min *ARG*1, *ARG*2

Назначение: команда выбора меньшего знакового числа из двух знаковых аргументов

Синтаксис:

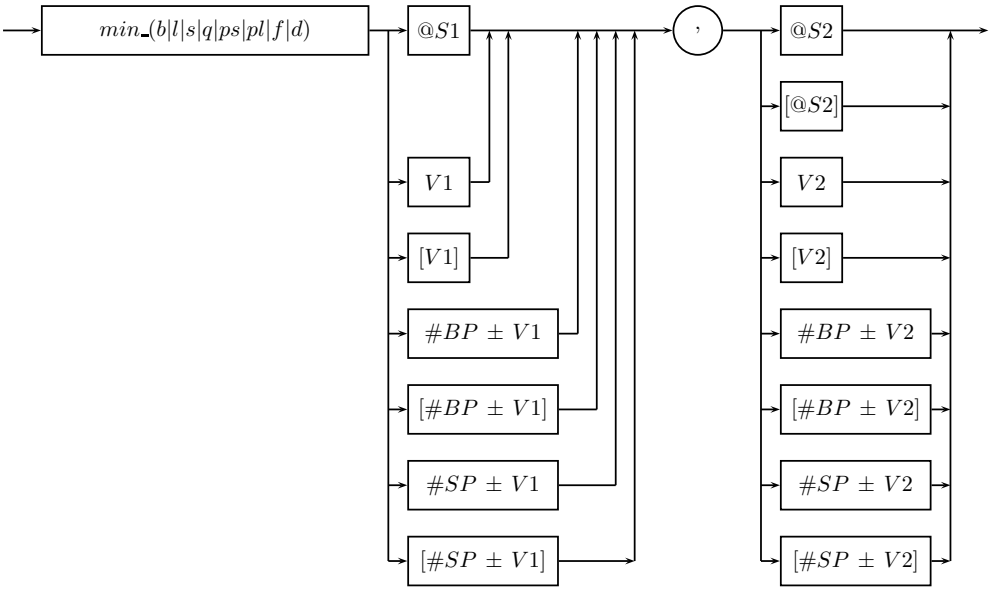

Рис. 34: Синтаксическое описание команды *min*

Наличие результата: да

Алгоритм работы:

- выполнить команду выбора меньшего знакового числа из двух знаковых аргументов, в качестве результата выдать *ARG*1, если *ARG*1 меньше, чем *ARG*2, иначе выдать *ARG*2;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типов операции **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.

4.4.33 minu (MINimum Unsigned)

Выбор меньшего беззнакового числа.

minu $ARG1$, $ARG2$

Назначение: команда выбора меньшего беззнакового числа из двух беззнаковых аргументов

Синтаксис:

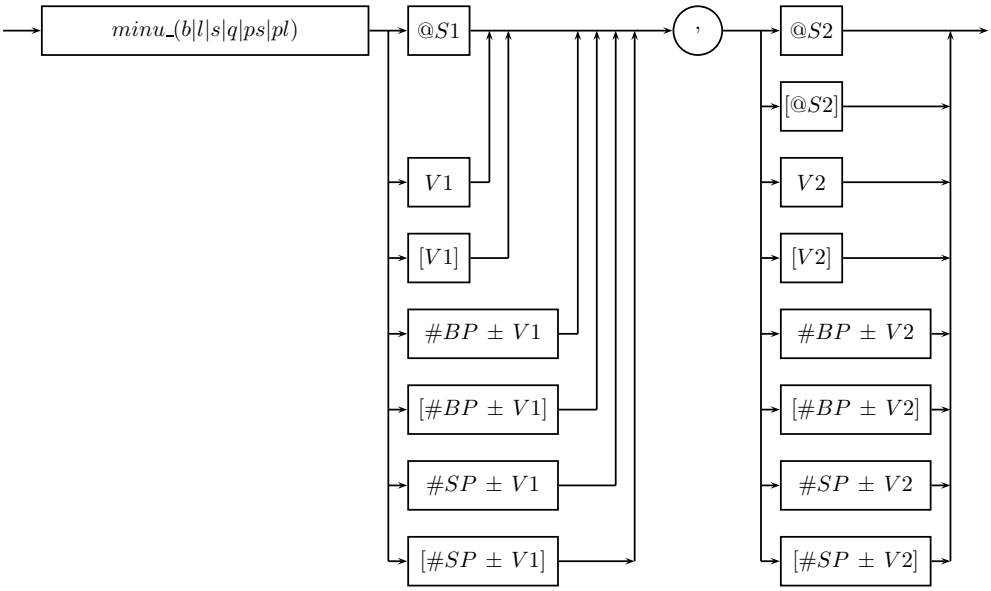

Рис. 35: Синтаксическое описание команды *minu*

Наличие результата: да

Алгоритм работы:

- выполнить команду выбора меньшего беззнакового числа из двух беззнаковых аргументов, в качестве результата выдать $ARG1$, если $ARG1$ меньше, чем $ARG2$, иначе выдать $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;

– для типов операции **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.

4.4.34 mod (MODification)

Модификация содержимого памяти данных

mod ARG

Назначение: команда модификации содержимого памяти данных с очерёдностью доступа к памяти

Синтаксис:

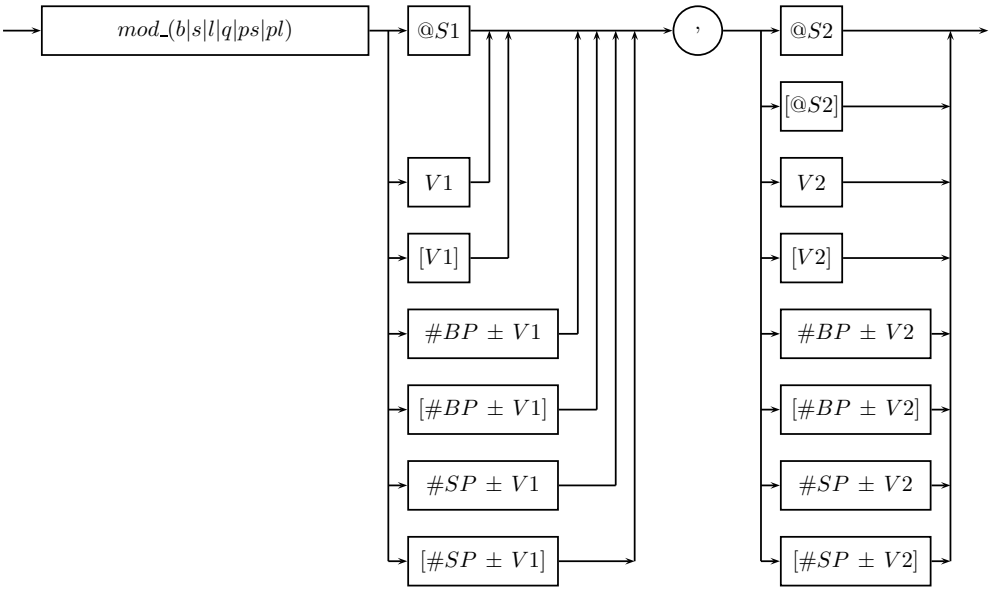

Рис. 36: Синтаксическое описание команды *mod*

Наличие результата: нет

Алгоритм работы:

– изменить значение размером байт, полуслово, слово, двойное слово в памяти данных в зависимости от типа команды, согласно следующей формуле:
$DM(ARG2) := DM(ARG2) + ARG1$.

Интерпретация значения аргумента: значение аргумента $ARG2$ интерпретируется как адрес памяти, по которому осуществляется изменение значения. Если для формирования значения аргумента $ARG2$ используется результат предшествующей команды, т. е. используется ссылка на коммутатор ($@S$), то старшие 32 разряда (с 32 по 63) игнорируются.

4.4.35 modrg (MODification ReGister value)

Изменение значения регистра

modrg $ARG1$, $ARG2$

Назначение: команда изменения значения регистра

Синтаксис:

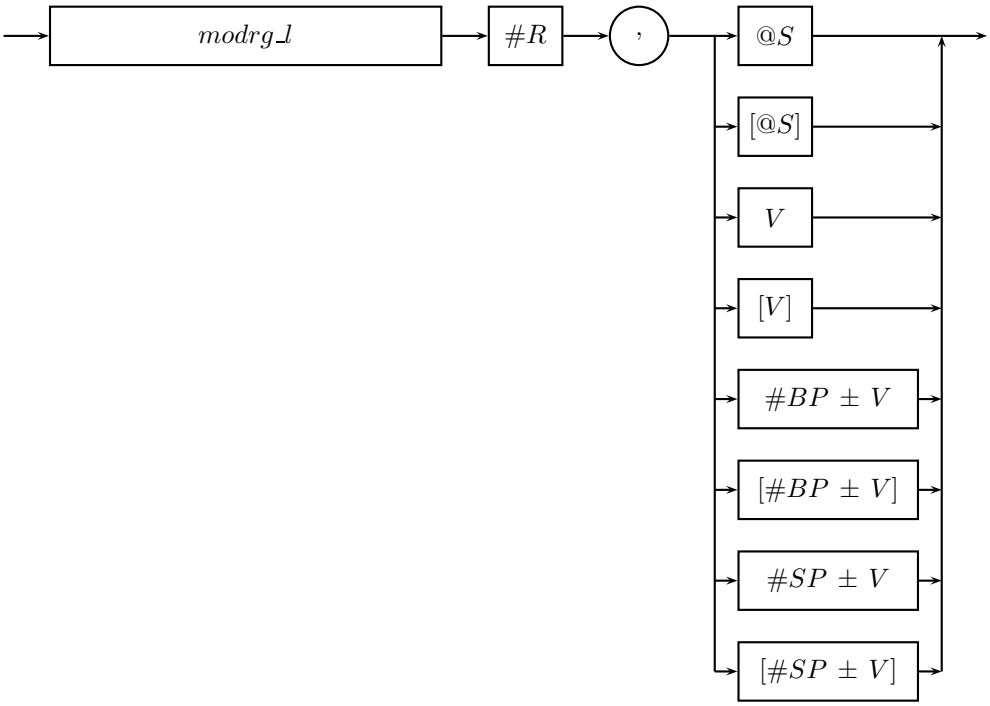

Рис. 37: Синтаксическое описание команды *modrg*

Наличие результата: нет

Алгоритм работы:

– изменить значение, заданное регистром R, согласно формуле: $\#R := \#R + ARG2$.

4.4.36 mul (MULtiply)

Умножение

mul $ARG1$, $ARG2$

Назначение: команда умножения двух знаковых аргументов

Синтаксис:

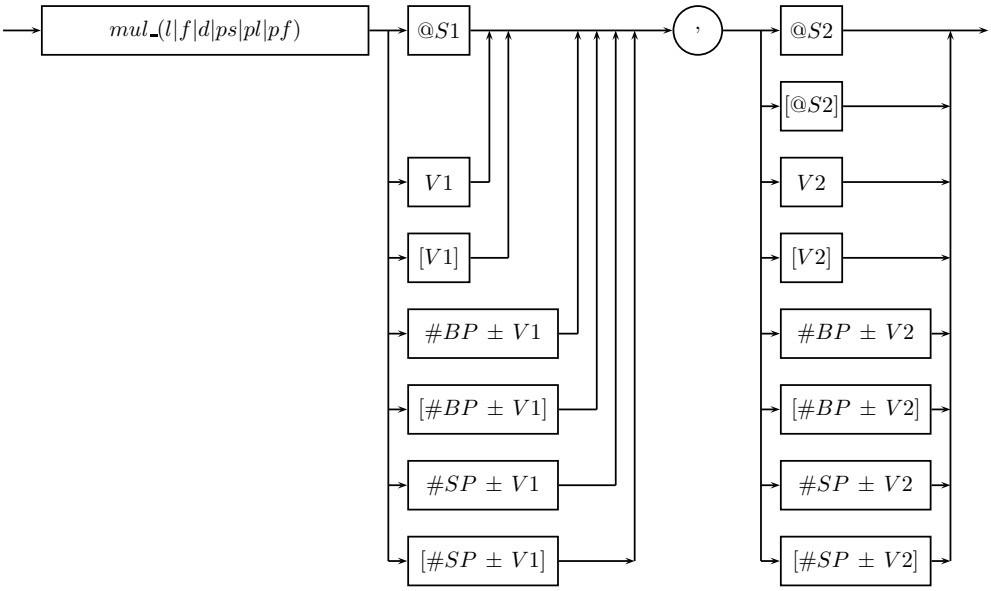

Рис. 38: Синтаксическое описание команды *mul*

Наличие результата: да

Алгоритм работы:

- выполнить знаковое умножение $ARG1 * ARG2$;
- поместить результат в коммутатор;

Результат:

- для типов операций **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.

Применение: команда mul используется для умножения двух знаковых 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         load_pl B
12         load_pl B + 8
13         mul_pf @1, @2
14         wr_pl @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 load_pl, читаются из памяти данных два 64-х разрядных упакованных числа по адресам B и $B + 8$ соответственно и помещаются в коммутатор;
- в строке №13 командой mul_pf 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 mul_pf согласно суффиксу pf интерпретируются как упакован-

ные числа размерностью 64 бита; результат выполнения команды также интерпретируются как 64-х разрядное упакованное число и помещается в коммутатор;

– в строке №14 командой wr_pl осуществляется запись в память данных по адресу $B+16$ результата выполнения предшествующей команды: @1 — результат выполнения команды умножения в строке №13;

– в строке №15 командой complete завершается текущий параграф.

4.4.37 mulu (MULtiply Unsigned)

Умножение

mulu $ARG1$, $ARG2$

Назначение: команда умножения двух беззнаковых аргументов

Синтаксис:

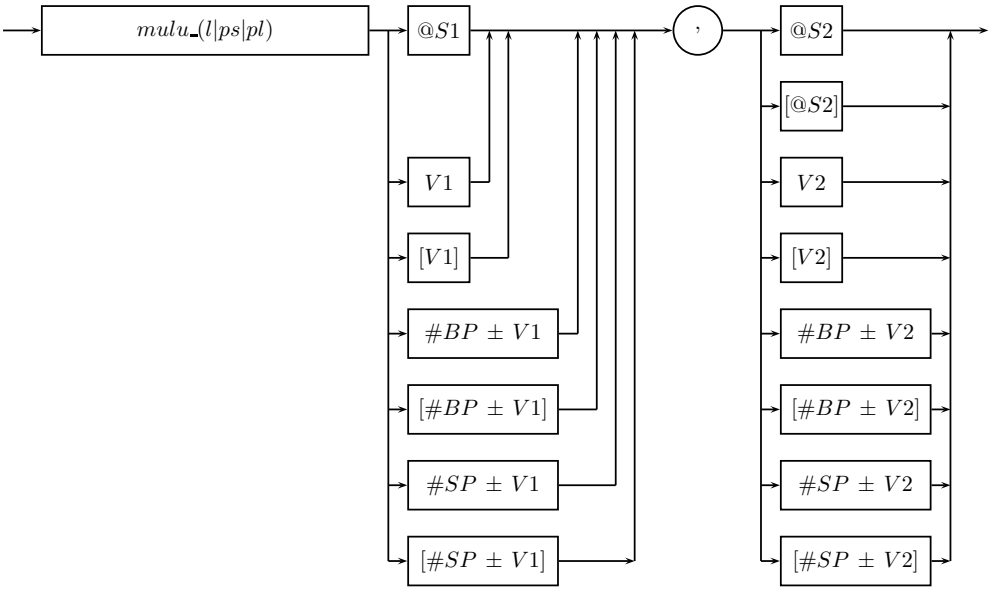

Рис. 39: Синтаксическое описание команды *mulu*

Наличие результата: да

Алгоритм работы:

- выполнить беззнаковое умножение $ARG1 * ARG2$;
- поместить результат в коммутатор;

Результат:

- для типов операций **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.

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

4.4.38 not (NOT)

Логического отрицание

not ARG

Назначение: команда логического отрицания аргумента

Синтаксис:

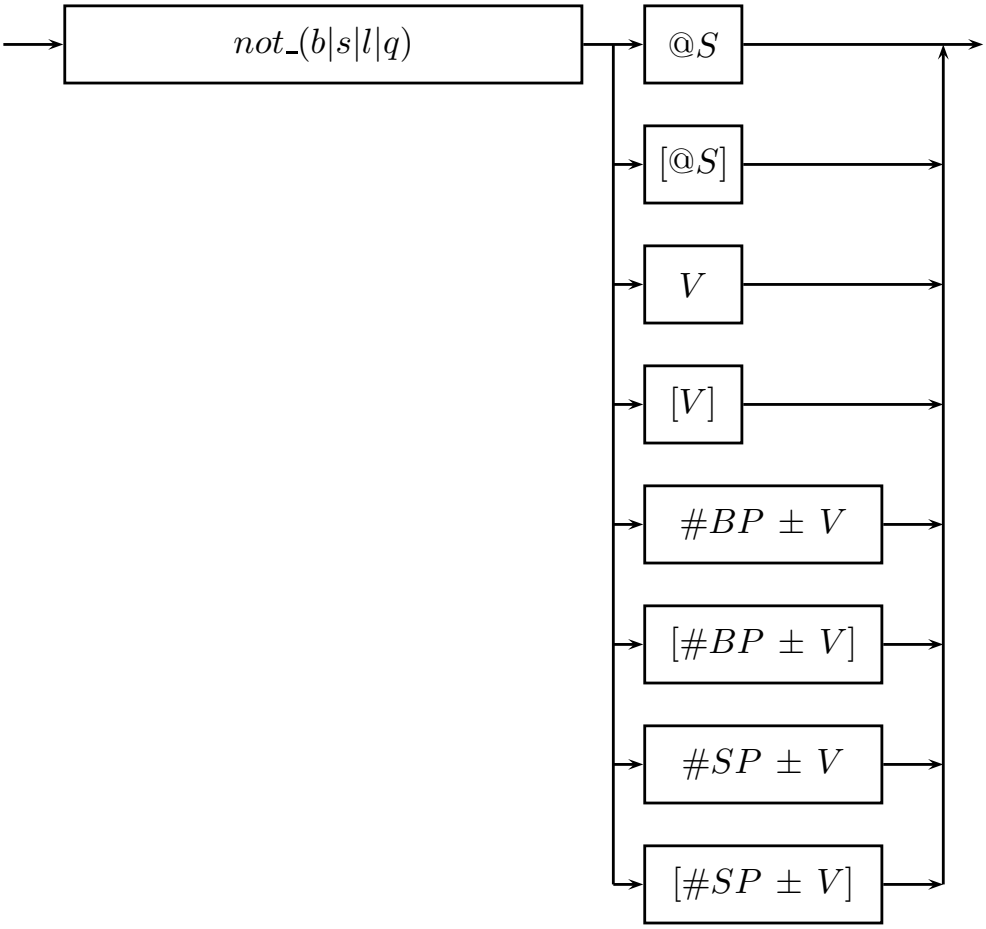

Рис. 40: Синтаксическое описание команды *not*

Наличие результата: да

Алгоритм работы:

– выполнить логическое отрицание $\sim ARG1$;

– поместить результат в коммутатор;

Результат:

– для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов

ячейки коммутатора, в которую помещается результат, обнуляются;

– для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;

– для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;

– для типа операции **quad** результат операции 64-х разрядный.

4.4.39 or (OR)

Логического сложение

or $ARG1$, $ARG2$

Назначение: команда логического сложения двух аргументов

Синтаксис:

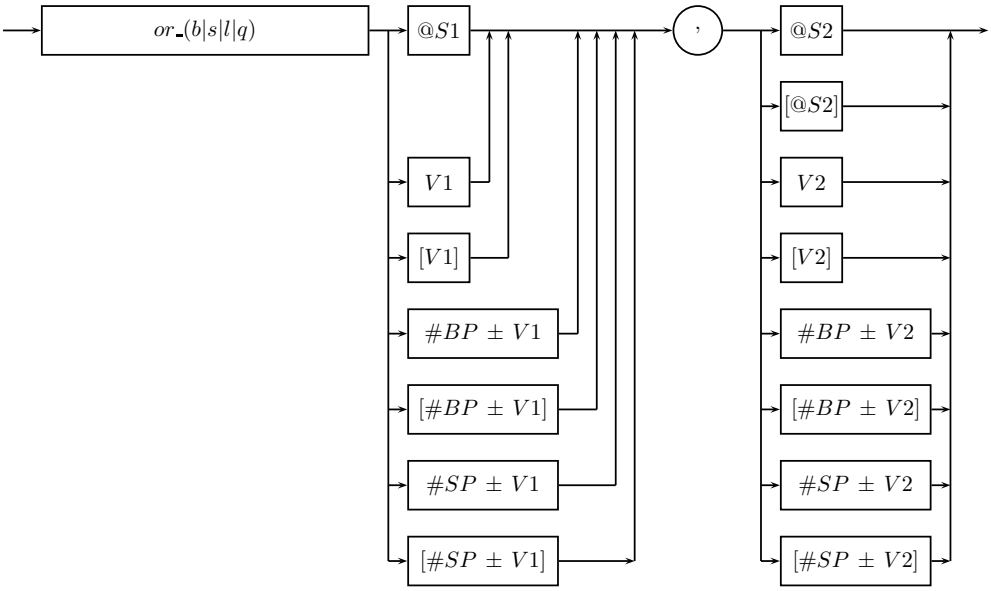

Рис. 41: Синтаксическое описание команды *or*

Наличие результата: да

Алгоритм работы:

- выполнить логическое сложение $ARG1 \mid ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **quad** результат операции 64-х разрядный.

4.4.40 pack (PACK)

Упаковка

pack $ARG1$, $ARG2$

Назначение: команда формирования упакованного числа

Синтаксис:

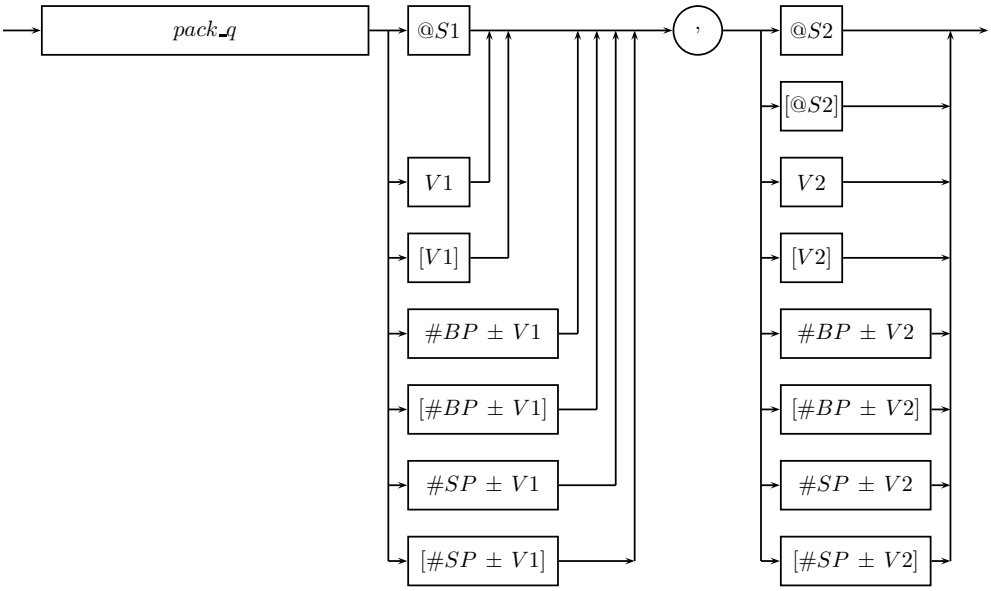

Рис. 42: Синтаксическое описание команды *pack*

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 $ARG2$;

– поместить результат в коммутатор;

Результат: результат операции 64-х разрядный.

4.4.41 patch (PATCH)

Склейка

patch $ARG1$, $ARG2$

Назначение: команда формирования склеенного числа

Синтаксис:

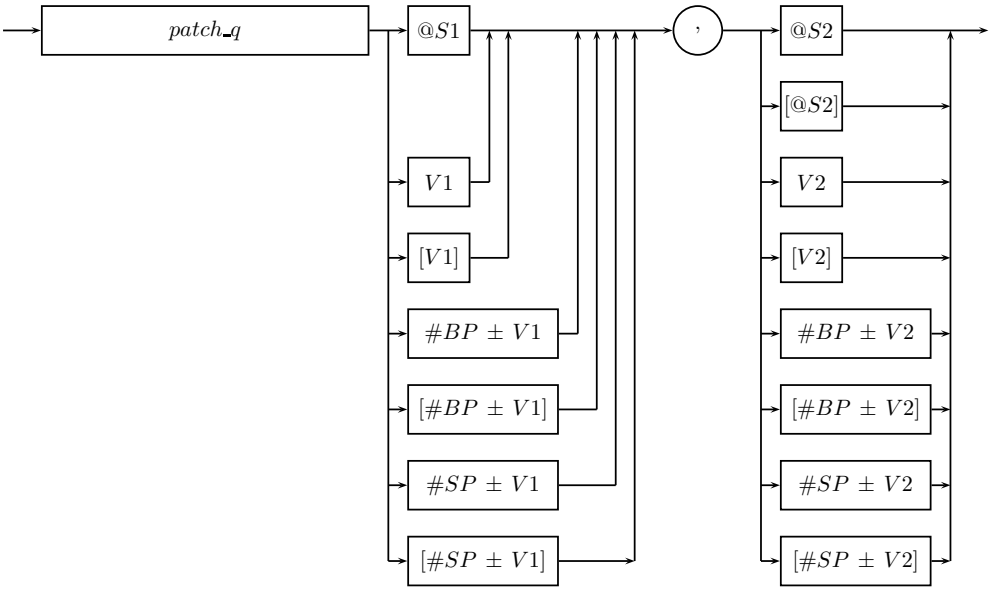

Рис. 43: Синтаксическое описание команды *patch*

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 $ARG2$;
- поместить результат в коммутатор;

Результат: результат операции 64-х разрядный.

4.4.42 pckb (Convolute Byte)

Свёртка байтов

pckb ARG

Назначение: команда свёртки байтов

Синтаксис:

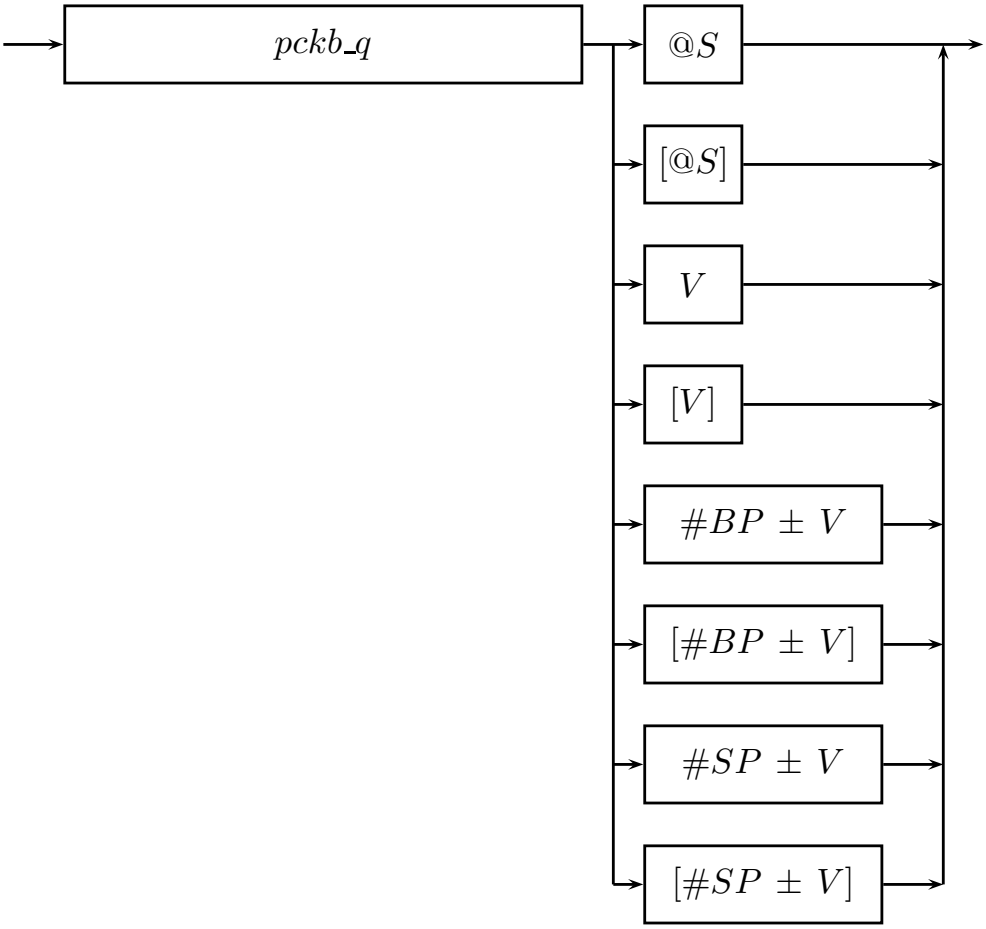

Рис. 44: Синтаксическое описание команды *pckb*

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**:

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

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

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

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

– поместить результат в коммутатор;

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

4.4.43 pckbs (Convolute Byte with SaTuration)

Свёртка байтов с насыщением

pckbs ARG

Назначение: команда свёртки байтов с насыщением

Синтаксис:

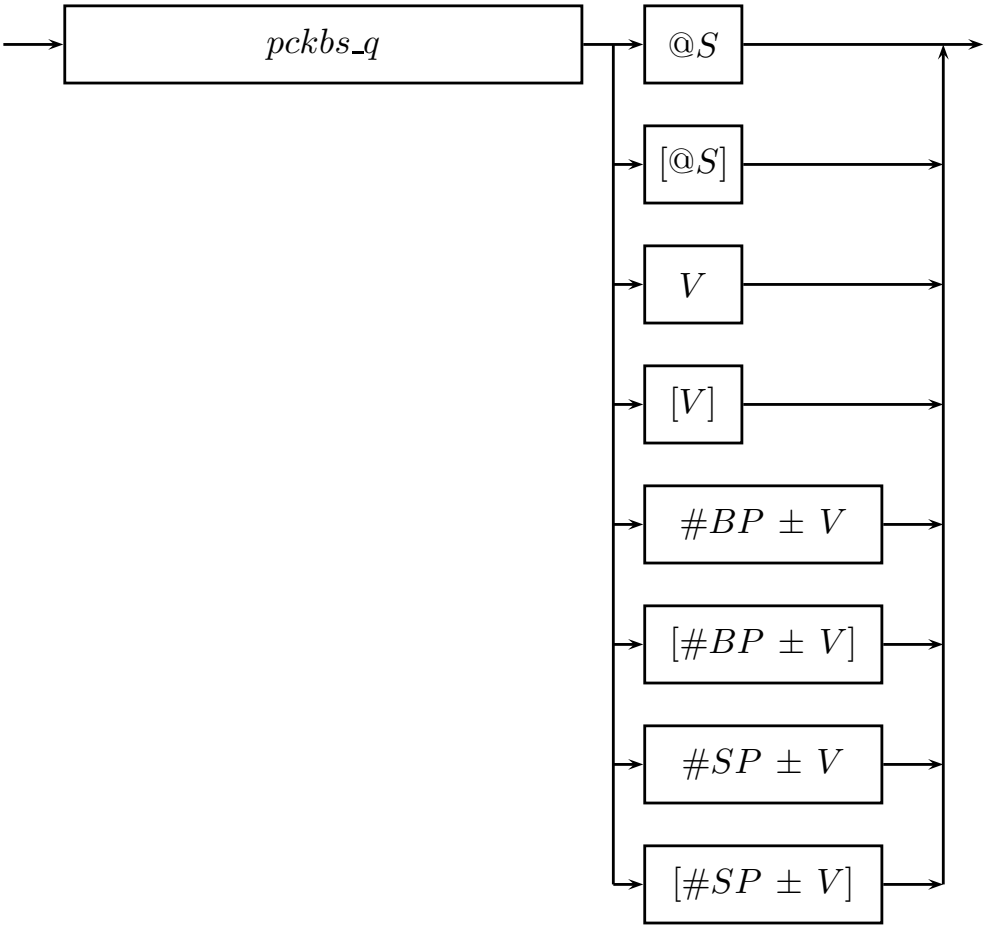

Рис. 45: Синтаксическое описание команды *pckbs*

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**:

$$RES[31:24] = \begin{cases} ARG[55:48], & \text{если } ARG[63:48] <= 255 \\ 255, & \text{если } ARG[63:48] > 255 \end{cases}$$

$$RES[23:16] = \begin{cases} ARG[39:32], & \text{если } ARG[47:32] <= 255 \\ 255, & \text{если } ARG[47:32] > 255 \end{cases}$$

$$RES[15:08] = \begin{cases} ARG[23:16], & \text{если } ARG[31:16] <= 255 \\ 255, & \text{если } ARG[31:16] > 255 \end{cases}$$

$$RES[07:00] = \begin{cases} ARG[07:00], & \text{если } ARG[15:00] <= 255 \\ 255, & \text{если } ARG[15:00] > 255 \end{cases}$$

для типа операции **signed byte**:

$$RES[31:24] = \begin{cases} ARG[55:48], & \text{если } -128 <= ARG[63:48] <= 127 \\ -128, & \text{если } ARG[63:48] < -128 \\ 127, & \text{если } ARG[63:48] > 127 \end{cases}$$

$$RES[23:16] = \begin{cases} ARG[39:32], & \text{если } -128 <= ARG[47:32] <= 127 \\ -128, & \text{если } ARG[47:32] < -128 \\ 127, & \text{если } ARG[47:32] > 127 \end{cases}$$

$$RES[15:08] = \begin{cases} ARG[23:16], & \text{если } -128 <= ARG[31:16] <= 127 \\ -128, & \text{если } ARG[31:16] < -128 \\ 127, & \text{если } ARG[31:16] > 127 \end{cases}$$

$$RES[07:00] = \begin{cases} ARG[07:00], & \text{если } -128 <= ARG[15:00] <= 127 \\ -128, & \text{если } ARG[15:00] < -128 \\ 127, & \text{если } ARG[15:00] > 127 \end{cases}$$

– поместить результат в коммутатор;

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

4.4.44 rol (ROtate Left)

Сдвиг циклический аргумента влево

rol $ARG1$, $ARG2$

Назначение: команда циклического сдвига аргумента влево

Синтаксис:

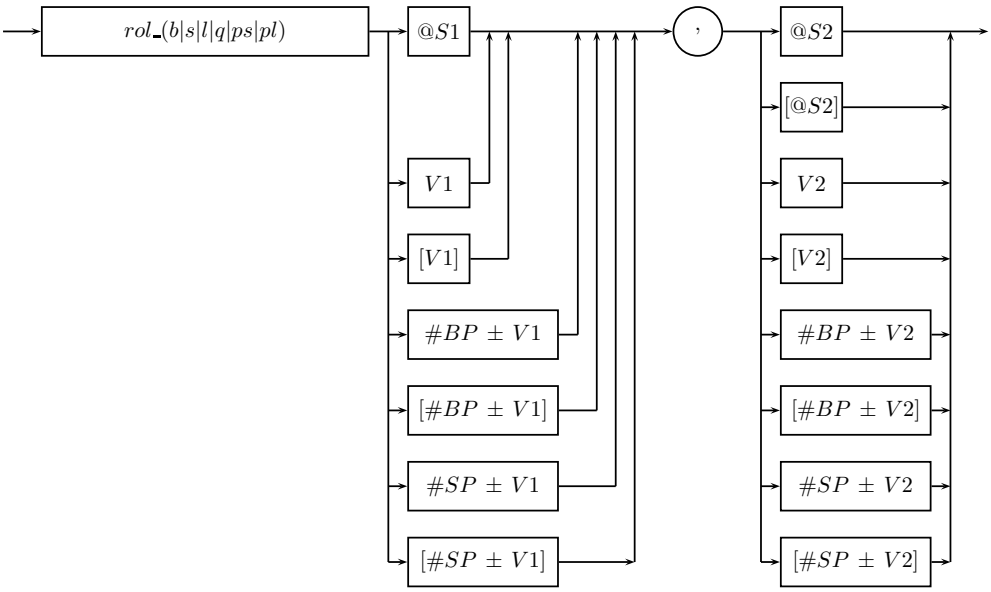

Рис. 46: Синтаксическое описание команды *rol*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента $ARG1$ влево на количество бит, заданных в аргументе $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.45 rolcn (ROtate Left on Constant)

Сдвиг циклический аргумента влево на константу

rolcn ARG1, ARG2

Назначение: команда циклического сдвига аргумента влево на константу

Синтаксис:

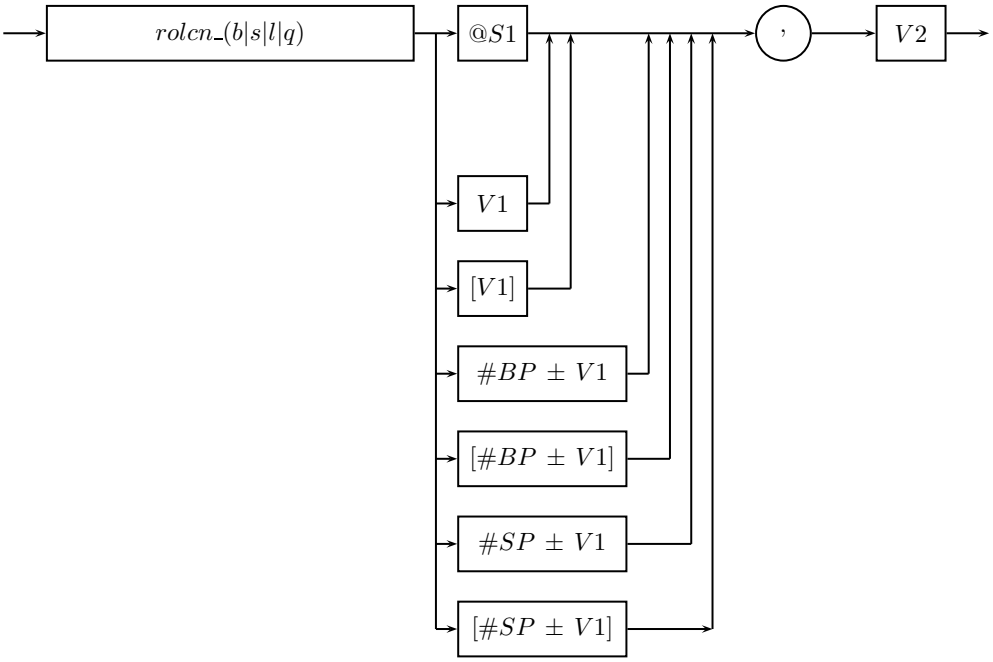

Рис. 47: Синтаксическое описание команды *rolcn*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента *ARG1* влево на константу на количество бит, заданных в аргументе *ARG2*, которое является целым положительным числом;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.46 ror (ROtate Right)

Сдвиг циклический аргумента вправо

ror $ARG1$, $ARG2$

Назначение: команда циклического сдвига аргумента вправо

Синтаксис:

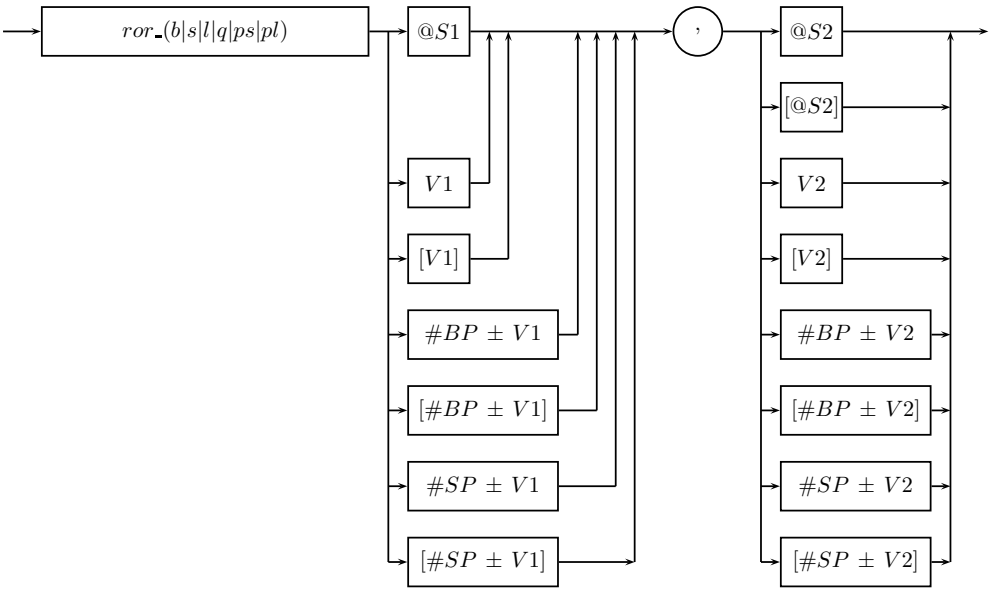

Рис. 48: Синтаксическое описание команды *ror*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента $ARG1$ вправо на количество бит, заданных в аргументе $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.47 rorcn (ROtate Right on Constant)

Сдвиг циклический аргумента вправо

rorcn $ARG1$, $ARG2$

Назначение: команда циклического сдвига аргумента вправо на константу

Синтаксис:

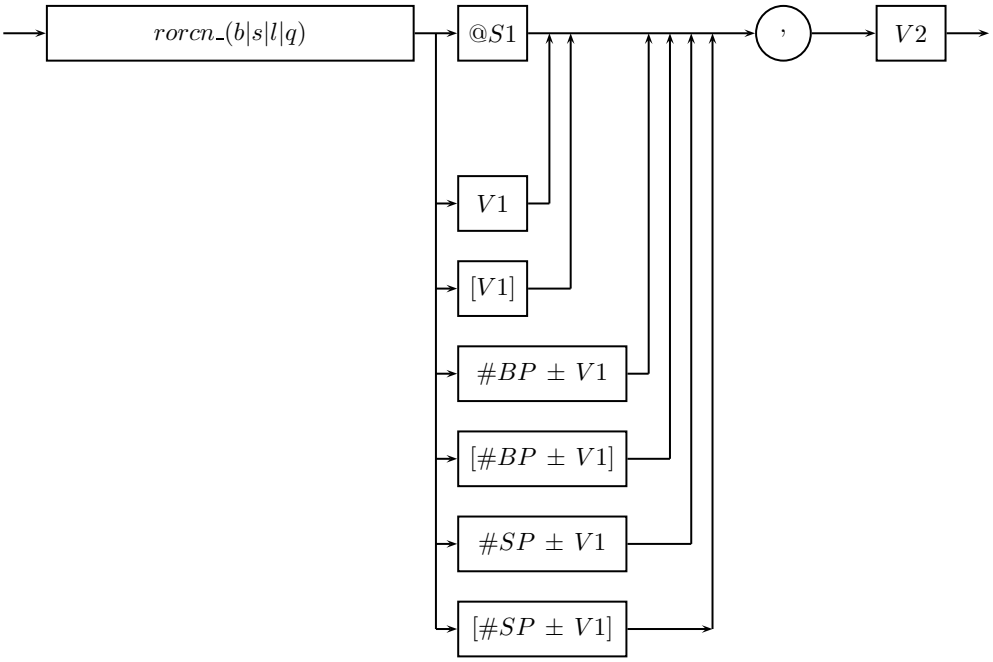

Рис. 49: Синтаксическое описание команды *rorcn*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента $ARG1$ вправо на константу на количество бит, заданных в аргументе $ARG2$, которое является целым положительным числом;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.48 sar (Shift Arithmetic Right)

Сдвиг арифметический аргумента вправо

sar $ARG1$, $ARG2$

Назначение: команда арифметического сдвига аргумента вправо

Синтаксис:

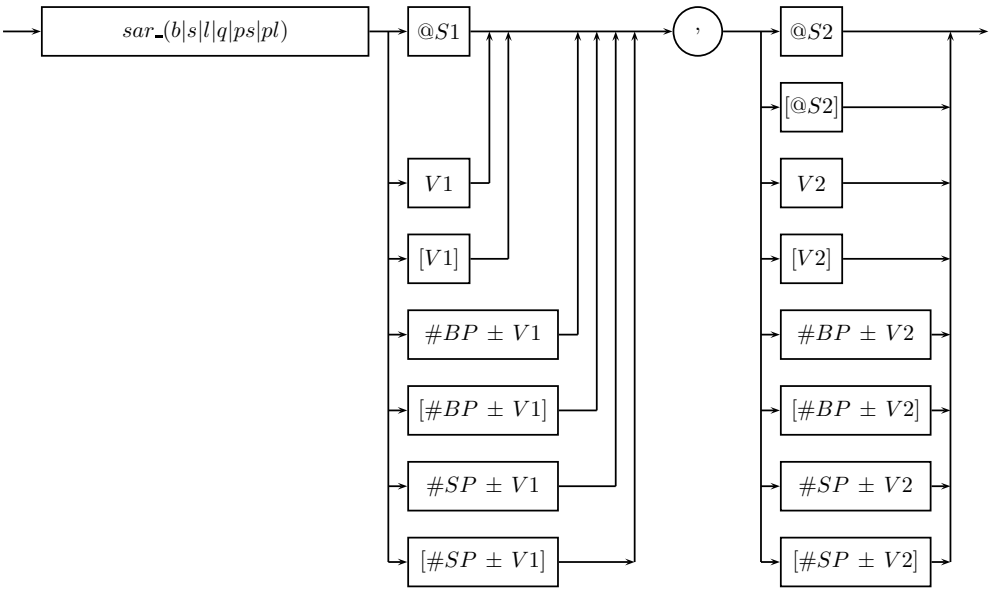

Рис. 50: Синтаксическое описание команды *sar*

Наличие результата: да

Алгоритм работы:

- выполнить арифметический сдвиг аргумента $ARG1$ вправо на количество бит, заданных в аргументе $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.49 sarcn (Shift Arithmetic Right on Constant)

Сдвиг арифметический аргумента вправо на константу

sarcn $ARG1$, $ARG2$

Назначение: команда арифметического сдвига аргумента вправо на константу

Синтаксис:

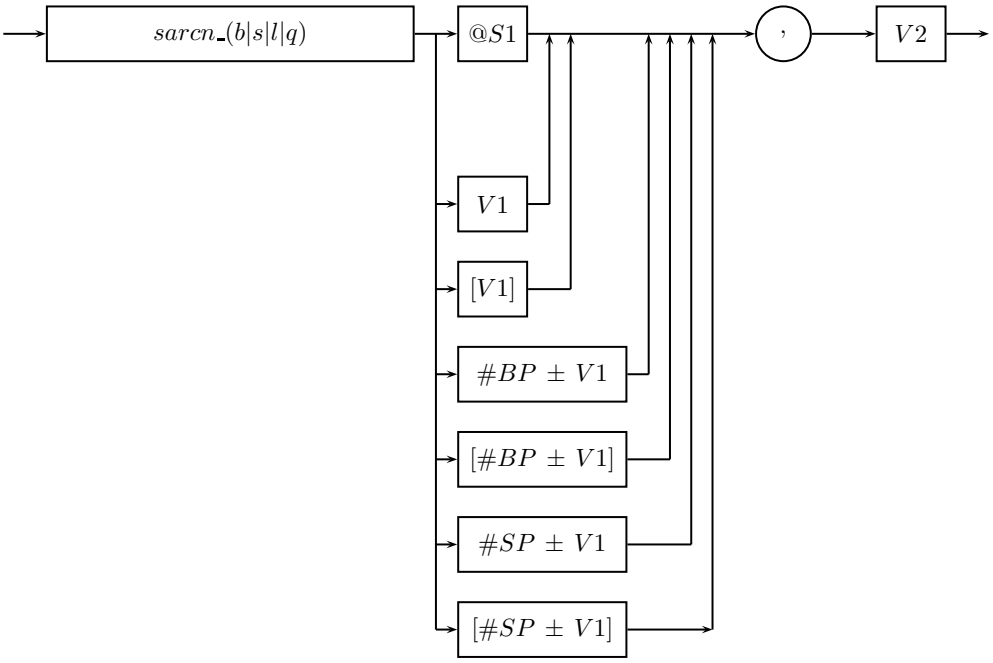

Рис. 51: Синтаксическое описание команды *sarcn*

Наличие результата: да

Алгоритм работы:

- выполнить арифметический сдвиг аргумента $ARG1$ вправо на константу на количество бит, заданных в аргументе $ARG2$, которое является целым положительным числом;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;

– 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.

4.4.50 setjelse (SET Jump ELSE)

Безусловная установка адреса следующего параграфа по умолчанию

setjelse ARG

Назначение: команда безусловной установки адреса следующего параграфа

Синтаксис:

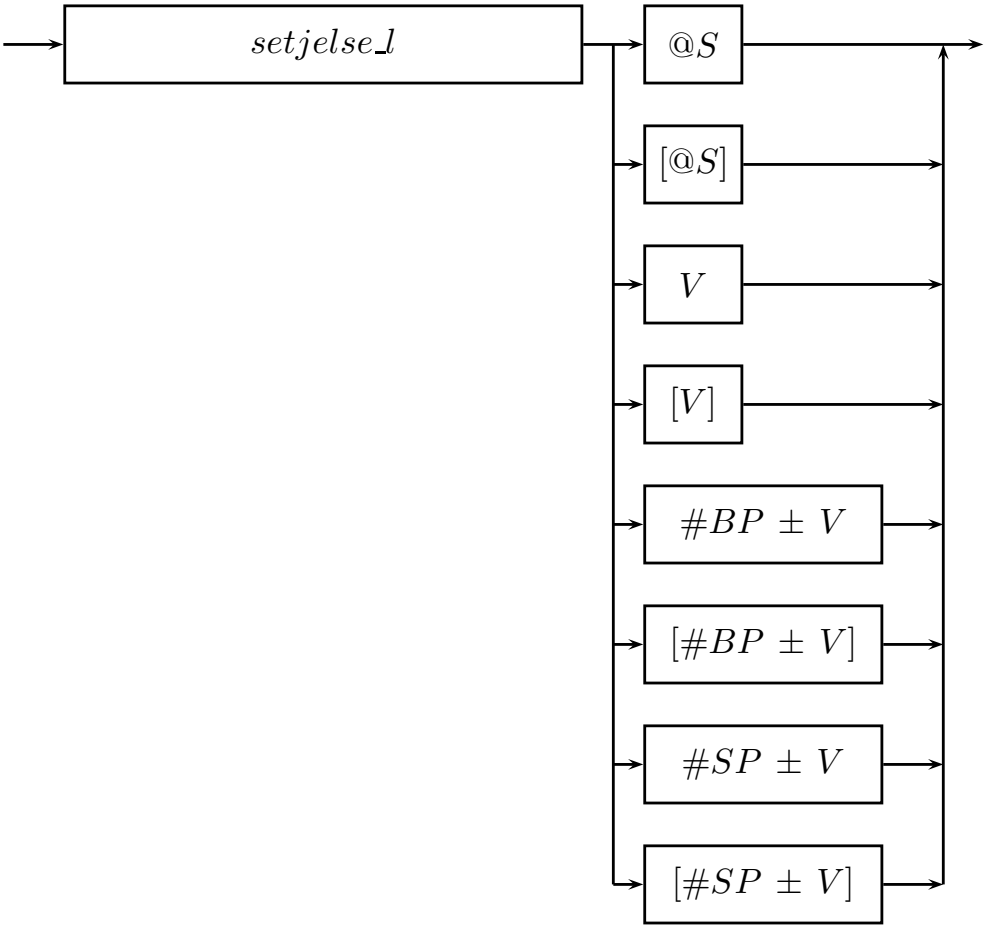

Рис. 52: Синтаксическое описание команды *setjelse*

Наличие результата: нет

Алгоритм работы:

– установить адрес следующего параграфа, равным значению аргумента ARG, если он ещё не был установлен одной из команд $setjf\_l$ или $setjt\_l$ (фактический переход будет осуществлён по окончании текущего параграфа);

4.4.51 setjf (SET Jump False)

Условная установка адреса следующего параграфа

setjf ARG1, ARG2

Назначение: команда условной установки адреса следующего параграфа

Синтаксис:

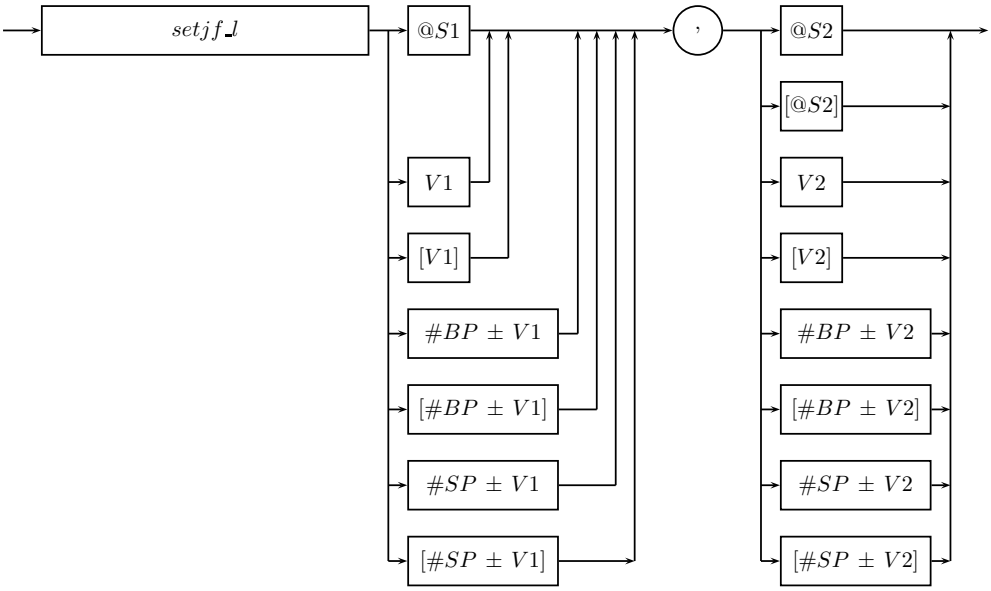

Рис. 53: Синтаксическое описание команды *setjf*

Наличие результата: нет

Алгоритм работы:

– установить адрес следующего параграфа, равным значению аргумента *ARG2* (фактический переход будет осуществлён по окончании текущего параграфа), если значение аргумента *ARG1* равно нулю;

4.4.52 SET Jump True)

Условная установка адреса следующего параграфа

setjt $ARG1$, $ARG2$

Назначение: команда условной установки адреса следующего параграфа

Синтаксис:

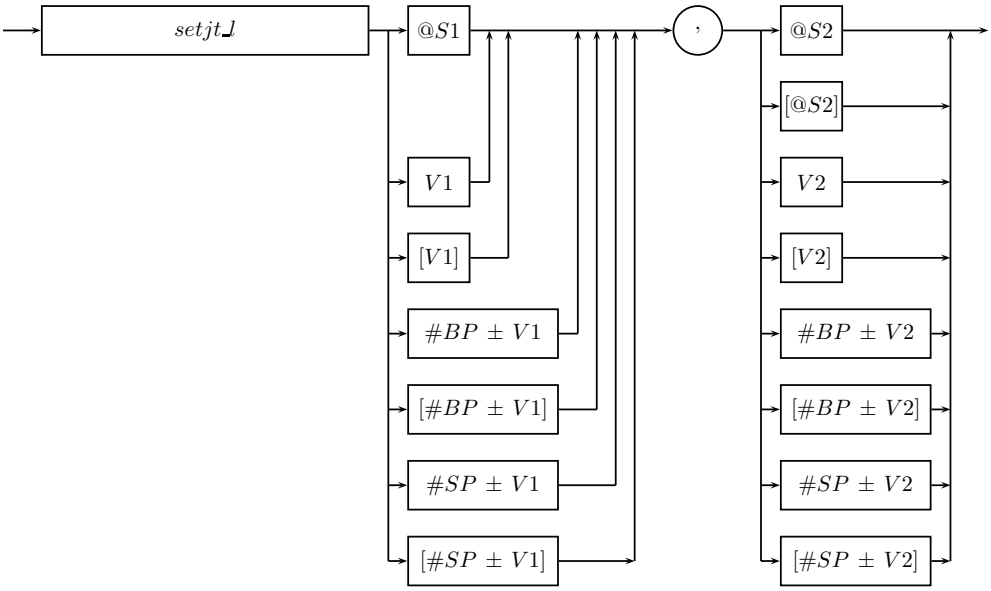

Рис. 54: Синтаксическое описание команды *setjt*

Наличие результата: нет

Алгоритм работы:

– установить адрес следующего параграфа, равным значению аргумента $ARG2$ (фактический переход будет осуществлён по окончании текущего параграфа), если значение аргумента $ARG1$ не равно нулю;

4.4.53 setrg (SET ReGister)

Установка регистра

setrg $ARG1$, $ARG2$

Назначение: команда установки значения регистра

Синтаксис:

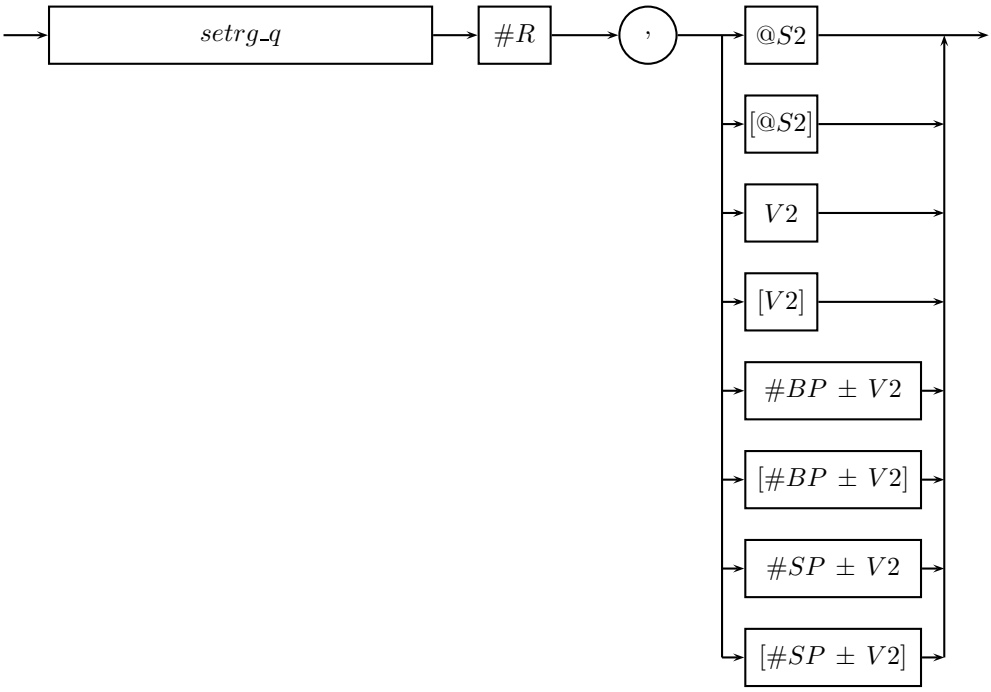

Рис. 55: Синтаксическое описание команды *setrg*

Наличие результата: нет

Алгоритм работы:

– установить значение регистра, заданного аргументом $ARG1$, равным значению аргумента $ARG2$;

Интерпретация значений аргументов: первый аргумент команды определяет номер или имя регистра (в отличие от большинства остальных команд), значение которого необходимо установить; значение второго аргумента, сохраняемое в указанном регистре, формируется согласно общим правилам формирования второго аргумента команд.

Применение: команда setrg используется для установки 64-х разрядного значения регистра, равным значению, второго аргумента.

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      .long 0x00000004
7
8  .text
9
10 C:
11     setjelse_l D
12
13     loadu_l A
14     setrg_q #16, @1
15 complete
16
17 D:
18     loadu_l B
19     setrg_q #PSW, @1
20 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 .long в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 32-х разрядное целое число $0x00000004$;

– 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 setjelse_l безусловно устанавливается адрес следующего параграфа, равным D;

– в строке №13 командой loadu_l читается из памяти данных 32-х разрядное целое беззнаковое число по адресу A и помещаются в коммутатор;

– в строке №14 командой setrg_q в регистр общего назначения №16 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 №16.

– 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 loadu_l, читается из памяти данных 32-х разрядное целое беззнаковое число по адресу B и помещается в коммутатор;

– в строке №19 командой setrg_q в управляющий регистр PSW помещается результат выполнения предшествующей команды: @1 — результат выполнения команды чтения в строке №18;

– в строке №20 командой complete завершается текущий параграф; происходит фактическая установка значения управляющего регистра PSW.

4.4.54 sf (Select if False)

Выбор второго аргумента по условию

$sf\ ARG1,\ ARG2$

Назначение: команда условного выбора второго аргумента

Синтаксис:

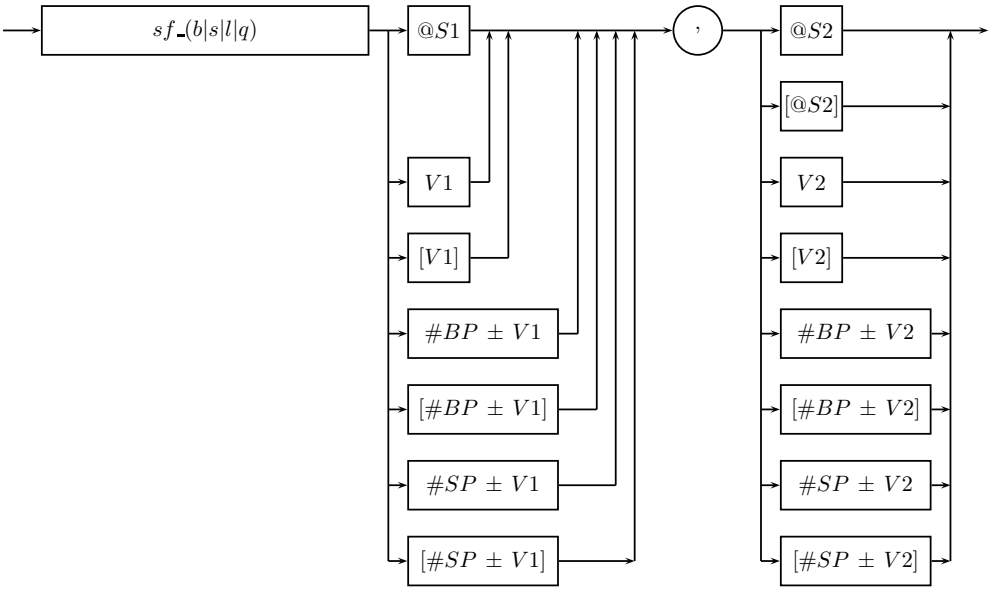

Рис. 56: Синтаксическое описание команды sf

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента $ARG2$, если значение аргумента $ARG1$ равно нулю, иначе – равным нулю;
- поместить результат в коммутатор;

Результат: результат операции 64-х разрядный.

4.4.55 sll (Shift Logical Left)

Сдвиг логический аргумента влево

sll $ARG1$, $ARG2$

Назначение: команда логического (арифметического) сдвига аргумента влево

Синтаксис:

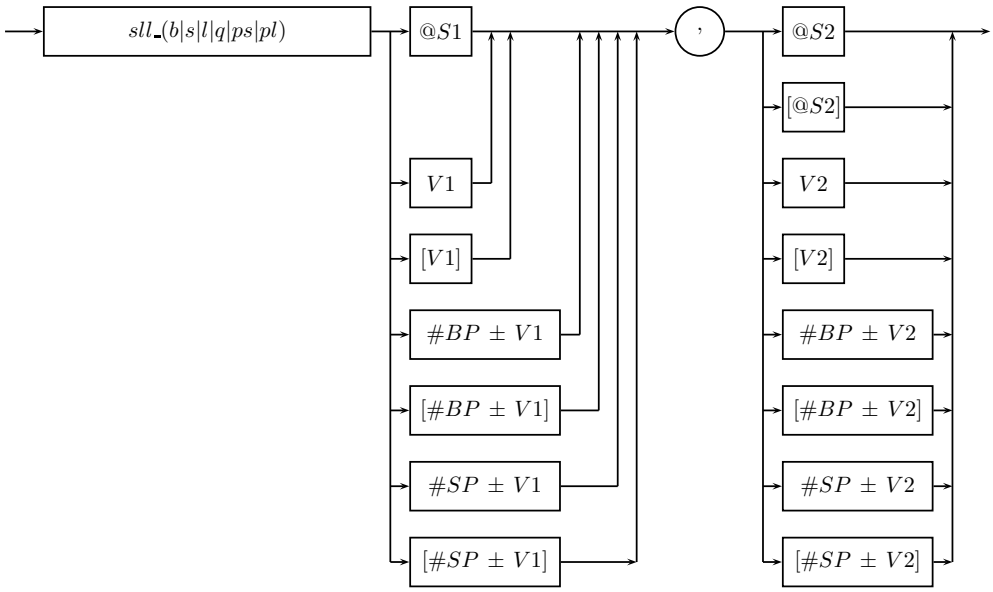

Рис. 57: Синтаксическое описание команды *sll*

Наличие результата: да

Алгоритм работы:

- выполнить логический сдвиг аргумента $ARG1$ влево на количество бит, заданных в аргументе $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.56 sllcn (Shift Logical Left on Constant)

Сдвиг логический аргумента влево на константу

sllcn $ARG1$, $ARG2$

Назначение: команда логического сдвига аргумента влево на константу

Синтаксис:

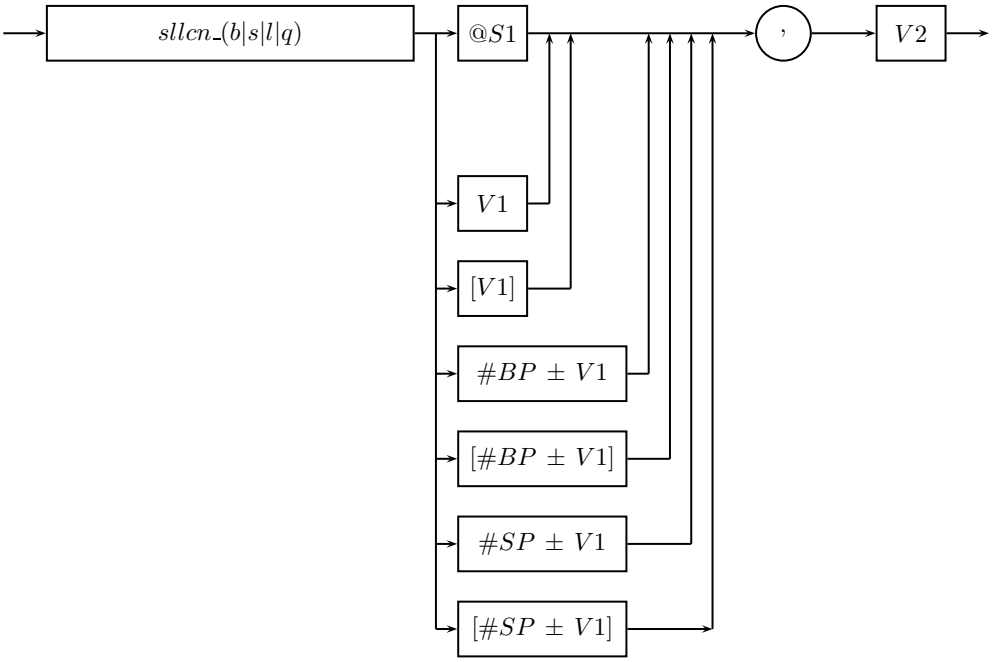

Рис. 58: Синтаксическое описание команды *sllcn*

Наличие результата: да

Алгоритм работы:

- выполнить логический сдвиг аргумента $ARG1$ влево на константу на количество бит, заданных в аргументе $ARG2$, которое является целым положительным числом;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.57 slr (Shift Logical Right)

Сдвиг логический аргумента вправо

slr $ARG1$, $ARG2$

Назначение: команда логического сдвига аргумента вправо

Синтаксис:

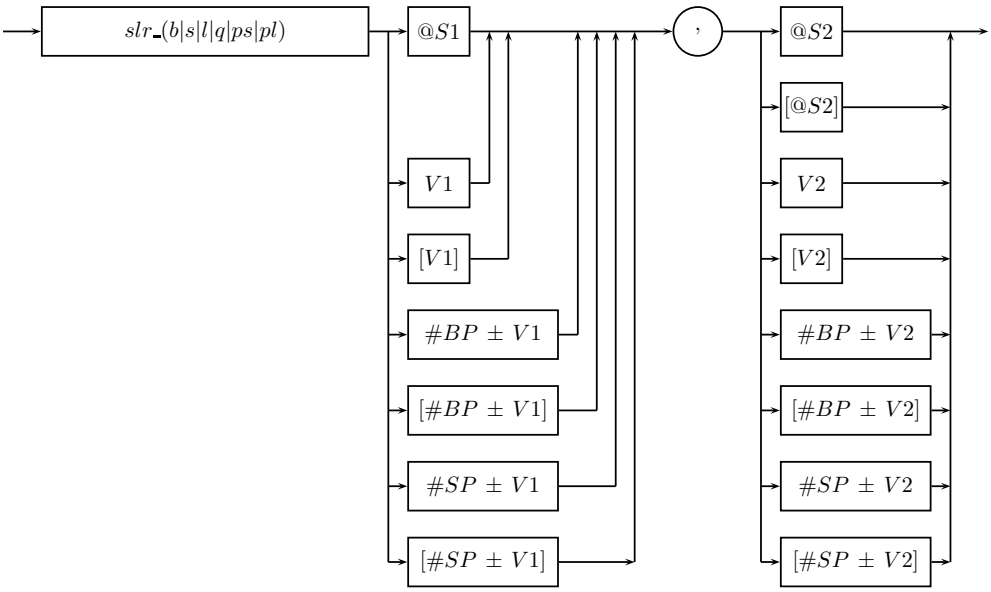

Рис. 59: Синтаксическое описание команды *slr*

Наличие результата: да

Алгоритм работы:

- выполнить логический сдвиг аргумента $ARG1$ вправо на количество бит, заданных в аргументе $ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.58 slrcn (Shift Logical Right on Constant)

Сдвиг логический аргумента вправо на константу

slrcn $ARG1$, $ARG2$

Назначение: команда логического сдвига аргумента вправо на константу

Синтаксис:

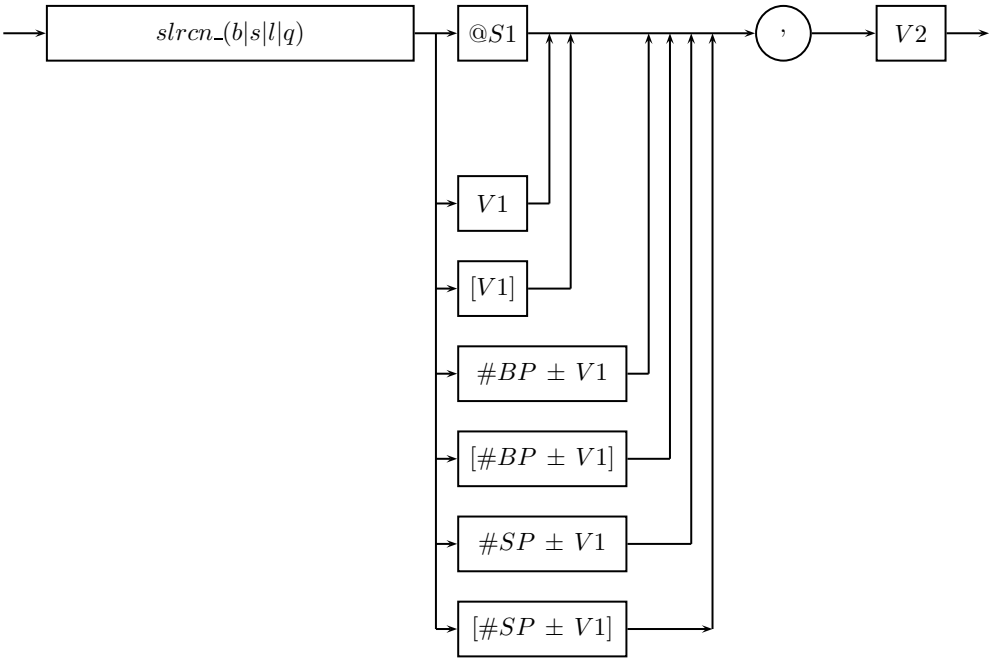

Рис. 60: Синтаксическое описание команды *slrcn*

Наличие результата: да

Алгоритм работы:

- выполнить логический сдвиг аргумента $ARG1$ вправо на константу на количество бит, заданных в аргументе $ARG2$, которое является целым положительным числом;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- 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.

4.4.59 sqrt (SQuare RooT)

Квадратный корень

sqrt ARG

Назначение: команда извлечения квадратного корня

Синтаксис:

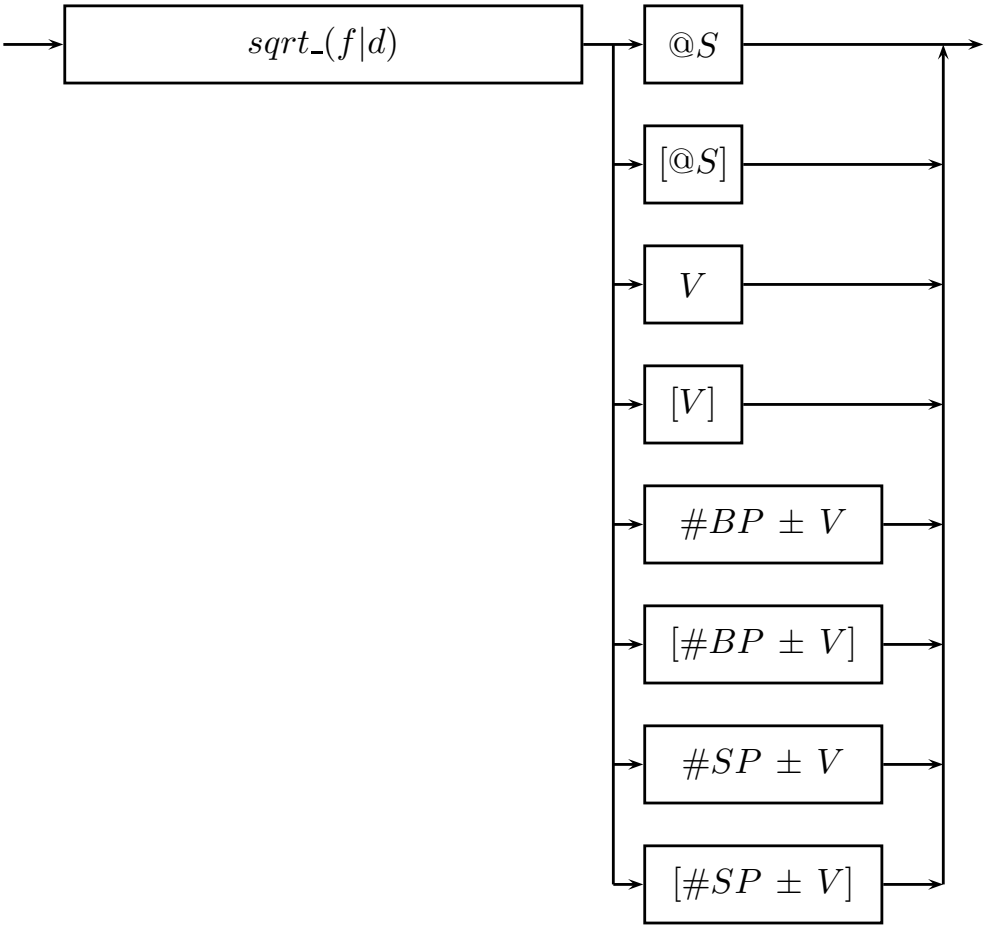

Рис. 61: Синтаксическое описание команды *sqrt*

Наличие результата: да

Алгоритм работы:

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

– поместить результат 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.

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

4.4.60 st (Select if True)

Выбор второго аргумента по условию

st $ARG1$, $ARG2$

Назначение: команда условного выбора второго аргумента

Синтаксис:

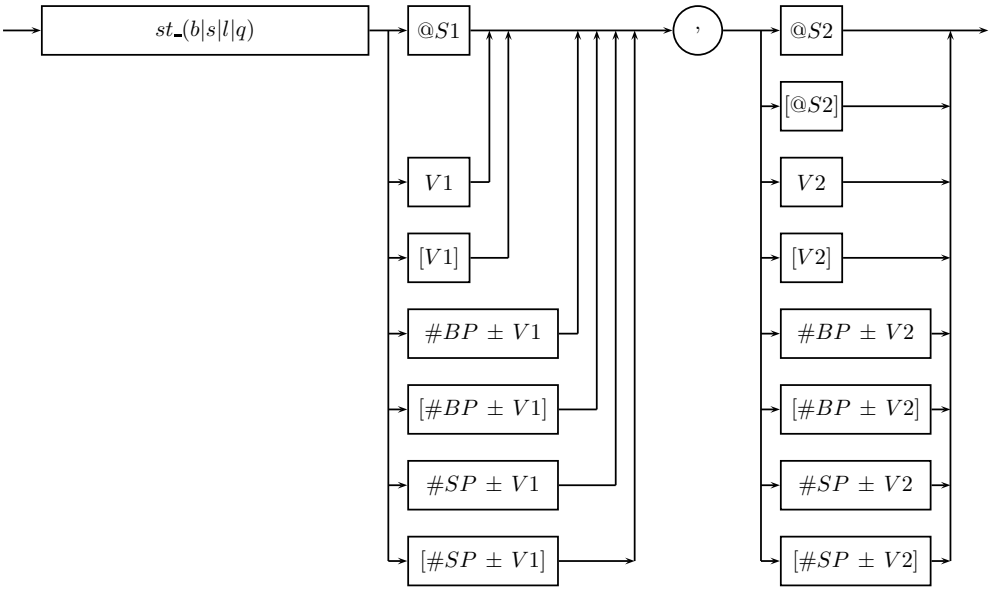

Рис. 62: Синтаксическое описание команды *st*

Наличие результата: да

Алгоритм работы:

- установить результат, равным значению аргумента $ARG2$, если значение аргумента $ARG1$ не равно нулю, иначе – равным нулю;
- поместить результат в коммутатор;

Результат: результат операции 64-х разрядный.

4.4.61 sub (SUBtract)

Вычитание

sub $ARG1$, $ARG2$

Назначение: команда вычитания двух аргументов

Синтаксис:

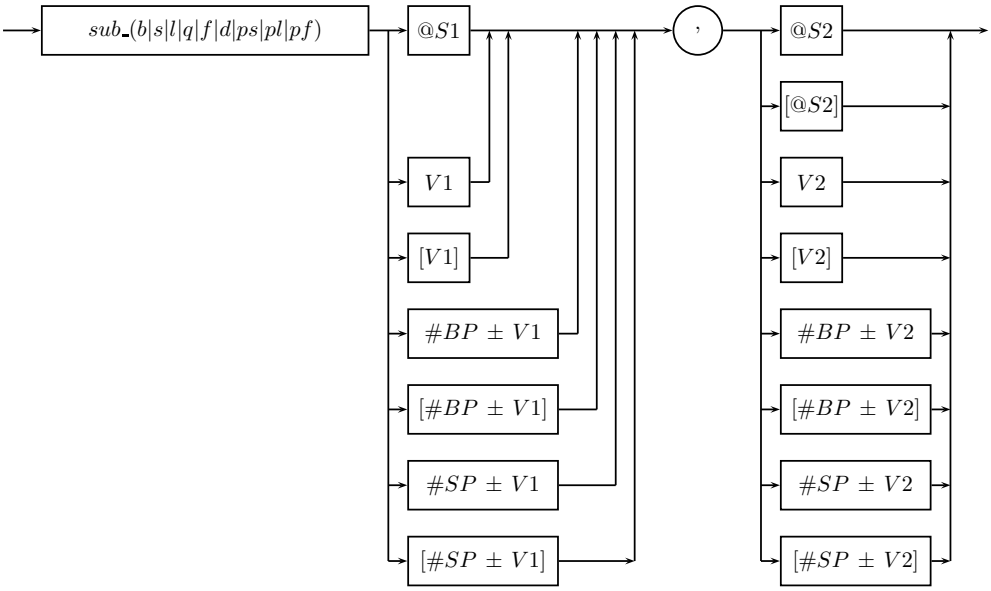

Рис. 63: Синтаксическое описание команды *sub*

Наличие результата: да

Алгоритм работы:

- выполнить вычитание $ARG1 - ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типов операции **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.

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      load_l [B]
10     sub_f @1, 0f-5.6
11     wr_l @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 load_l читается из памяти данных 32-х разрядное целое беззнаковое число по адресу B и помещается в коммутатор;
- в строке №10 командой sub_f выполняется операция вычитания результата выполнения предшествующей команды: @1 — результат выполнения команды чтения в строке №9, и константы -5.6; оба аргумента команды sub_f согласно суффиксу f интерпретируются как вещественные числа одинарной точности размерностью 32 бита; результат выполнения команды также интерпретируются как 32-х разрядное

вещественное число одинарной точности и помещается в коммутатор;

– в строке №11 командой wr_l 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.

4.4.62 tas (Test And Set)

Чтение содержимого памяти данных с одновременной записью

tas ARG

Назначение: команда чтения содержимого памяти данных с одновременной записью с очерёдностью доступа к памяти

Синтаксис:

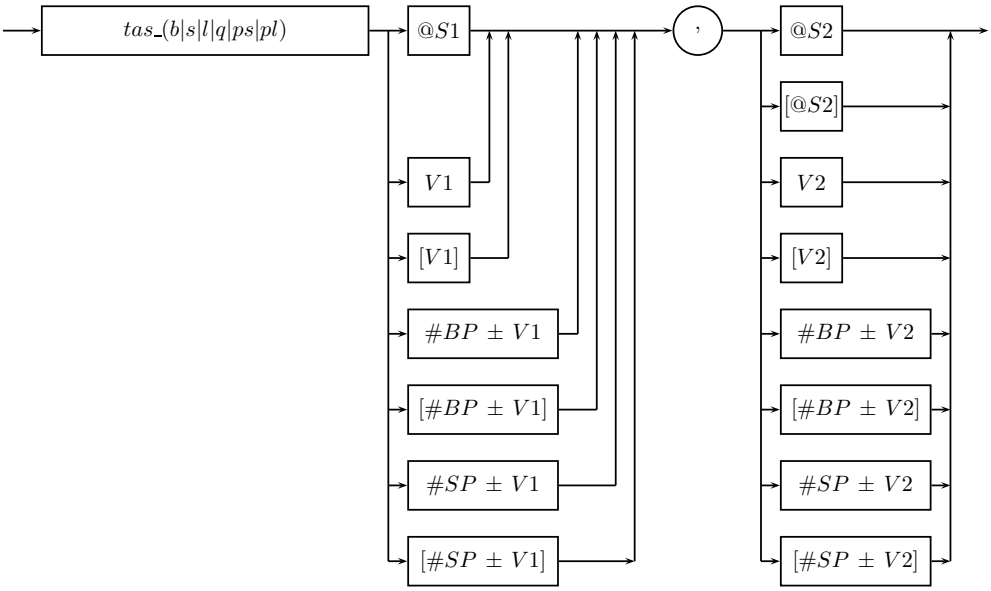

Рис. 64: Синтаксическое описание команды *tas*

Наличие результата: да

Алгоритм работы:

- загрузить значение размером байт, полуслово, слово, двойное слово из памяти данных в зависимости от типа команды, заданному аргументом $ARG2$, с одновременной его установкой в памяти данных равным значению, заданным аргументом $ARG1$;
- поместить результат в коммутатор;

Интерпретация значения аргумента: согласно выше описанному алгоритму работы, значение аргумента $ARG2$ интерпретируется как адрес памяти, по которому осуществляется загрузка значения, помещаемого в качестве результата данной команды в коммутатор. Если для формирования значения аргумента используется 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 64-х разрядный, разряды с 8 по 63 обнуляются;

– для типа операции **short** результат операции 64-х разрядный, разряды с 16 по 63 обнуляются;

– для типа операций **long** результат операции 64-х разрядный, разряды с 32 по 63 обнуляются;

– для остальных типов операций результат операции 64-х разрядный.

4.4.63 wr (WRite)

Запись

wr *ARG*1, *ARG*2

Назначение: команда записи значения в память

Синтаксис:

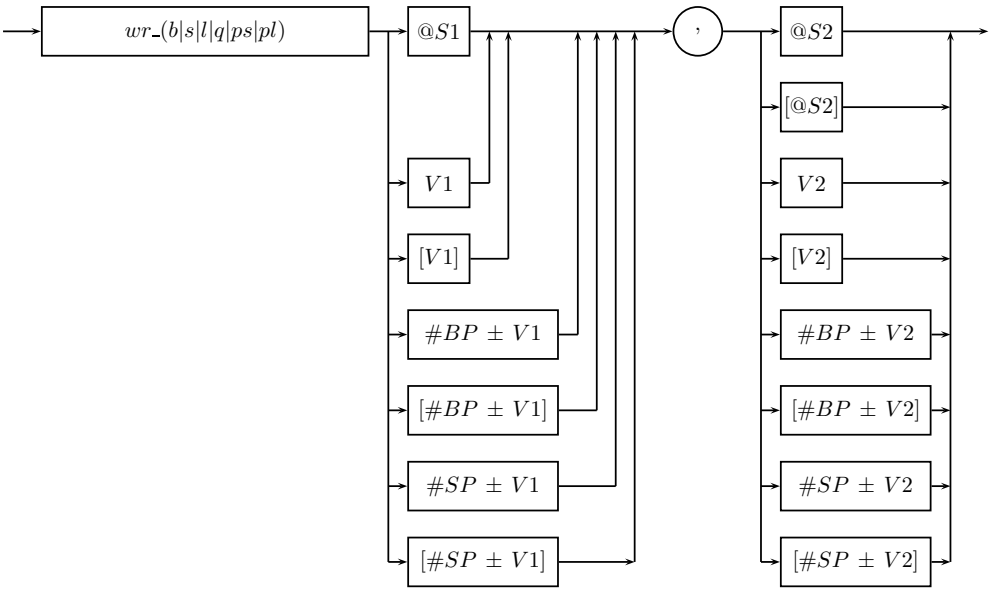

Рис. 65: Синтаксическое описание команды *wr*

Наличие результата: нет

Алгоритм работы:

– записать в зависимости от типа команды значение размером байт, полуслово, слово, двойное слово, заданное аргументом *ARG*1, в память данных по адресу, заданному аргументом *ARG*2;

Интерпретация значений аргументов: согласно выше описанному алгоритму работы, значение второго аргумента всегда интерпретируется как адрес памяти, по которому осуществляется запись значения первого аргумента. Если для формирования значения аргумента используется результат предшествующей команды, т. е. используется ссылка на коммутатор (@*S*), то старшие 32 разряда (с 32 по 63) игнорируются.

Применение: команда wr используется для записи значения в память данных.

Пример:

```
1 .data
2
```

```
3  A:
4      .float  0f-1.25E-1
5
6  .text
7
8  B:
9      load_l  [A]
10     load_l  0x3f028f5c ; представление числа 0.51 в IEEE754
11     add_f  @1, @2
12     wr_l  @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 load_l, читается из памяти данных 32-х разрядное целое беззнаковое число по адресу A и помещается в коммутатор;
- в строке №10 командой load_l помещается в коммутатор 32-х разрядное целое беззнаковое число (константа $0x3f028f5$);
- в строке №11 командой add_f 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 add_f 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 wr_l 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.

4.4.64 xor (XOR)

Логического сложение

xor $ARG1$, $ARG2$

Назначение: команда логического сложения по модулю 2 двух аргументов

Синтаксис:

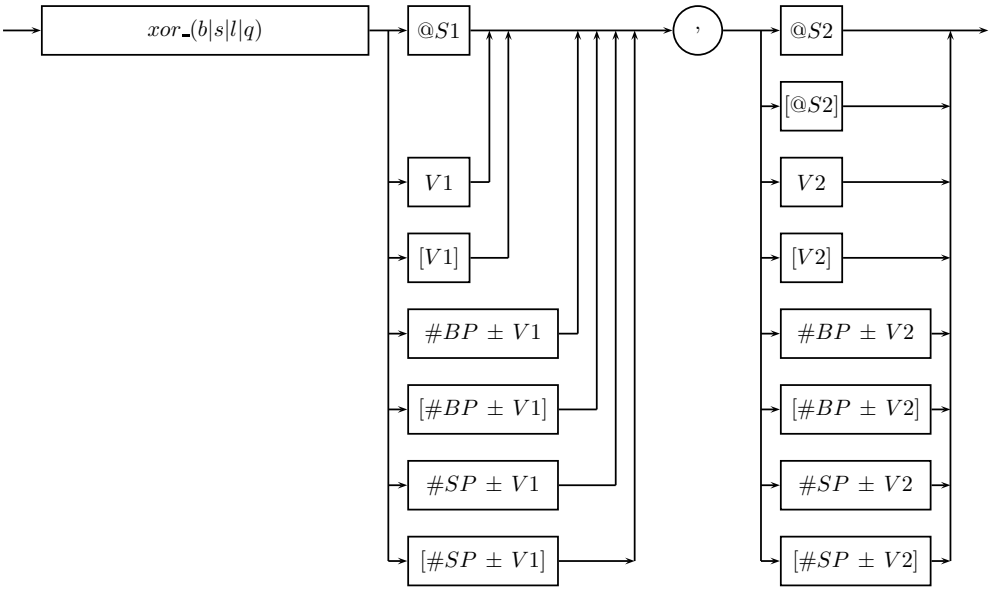

Рис. 66: Синтаксическое описание команды *xor*

Наличие результата: да

Алгоритм работы:

- выполнить логическое сложение по модулю 2 $ARG1$ ^$ARG2$;
- поместить результат в коммутатор;

Результат:

- для типа операции **byte** результат операции 8-ми разрядный, старшие 56 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **short** результат операции 16-ти разрядный, старшие 48 разрядов ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **long** результат операции 32-х разрядный, старшие 32 разряда ячейки коммутатора, в которую помещается результат, обнуляются;
- для типа операции **quad** результат операции 64-х разрядный.

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.

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).

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.

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.

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.

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.

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.

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.

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.

5.9 .double flonums

.double ожидает ноль или более чисел с плавающей точкой двойной точночти, 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.

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.

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.

5.12 .end

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

5.13 .endif

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

5.14 .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.

5.15 .equiv symbol, expression

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

5.16 .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)

5.17 .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)

5.18 .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 выраженния expression.

5.19 .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.

5.20 .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.

5.21 .float flonums

.float является синонимом директивы .single и ожидает ноль или более чисел с плавающей точкой одинарной точночти, 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.

5.22 .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.

5.23 .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 определён.

.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.

5.24 .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’.

5.25 .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.

5.26 .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 ргистра basic_block в истинное значение при выполнении программы, содержащейся в секции .debug_line, машиной состояний;

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

epilogue_begin наличие данной опции приведёт к установке значения 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.

5.27 .local names

.local устанавливает атрибут связывания каждого символа, разделенных 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.

5.28 .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.

5.29 .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.

5.30 .print string

Во время ассемблирования данной директивы ассемблер выведет в стандартный поток вывода строку string. Строка string должна быть заключена в двойные кавычки.

5.31 .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.

5.32 .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:

a данные секции name является 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 содержтит 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.

5.33 .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.

5.34 .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.

5.35 .single flonums

.single является синонимом директивы .float и ожидает ноль или более чисел с плавающей точкой одинарной точночти, 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.

5.36 .size name, expression

.size устанавливает размер в байтах символа name равным результату вычисления выражения expression.

5.37 .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.

5.38 .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.

5.39 .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.

5.40 .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.

5.41 .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.

5.42 .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.

5.43 .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)

5.44 .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.

5.45 .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)

5.46 .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.

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 располагается 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, .section.

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 необхо
       димого размера в байтах без изменения текущей секции ассембл
       ирования
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 памяти данных, например */
30      .long  , , ,
31      .byte
32      .skip 12
33
34  Bc: // объявление метки
35      /* директивы инициализации памяти 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      load_l [Da]
44      loadd_l [Da + 4]
45      mul_l @1, @2
46      wr_l @1, Bc
47  complete // завершение параграфа
48
49  Tb: // объявление метки, а также начало нового параграфа
50      /* команды процессора */
51  complete // завершение параграфа
```