MULTICLET®

www.multiclet.com

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ
ПО ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

СОДЕРЖАНИЕ

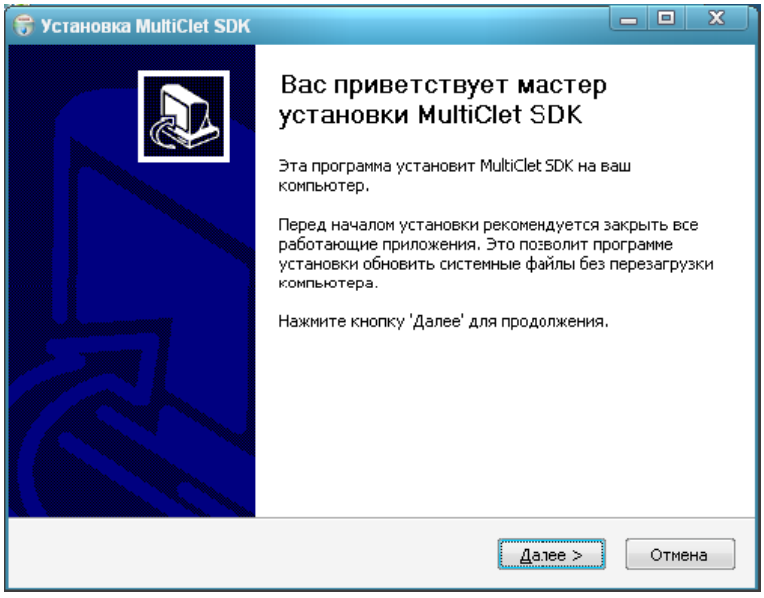
1. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ ПО УСТАНОВКЕ И УДАЛЕНИЮ ПО

1.1. Установка программного обеспечения и документации.

Установка программного обеспечения и документации (далее MultiCletSDK) осуществляется специально созданным инсталлятором.

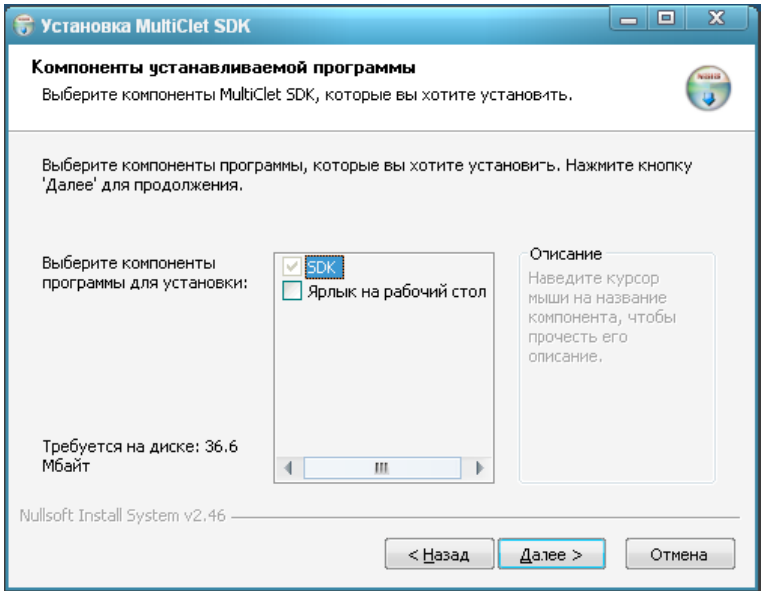
Для начала установки нужно запустить файл MultiCletSDK.exe.

После запуска установки отображается окно приветствия с краткой информацией об устанавливаемом продукте.



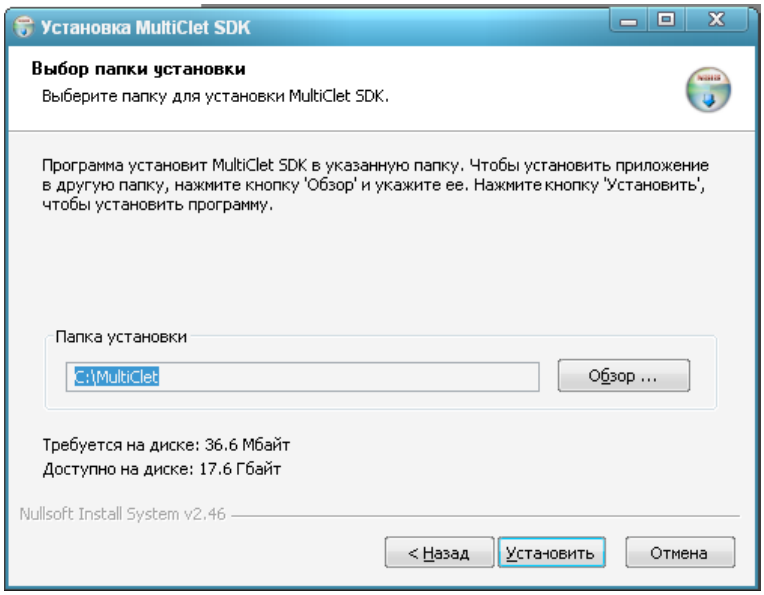
Для продолжения установки необходимо нажать кнопку «Далее».

На втором шаге отображается окно со списком доступных для установки компонентов, в котором можно выбрать параметр «Ярлык на рабочий стол». По умолчанию он выключен. Его включение приведёт к тому, что после установки на рабочем столе появится ярлык, указывающий на главную папку с документацией, тестами и проектами MultiCletSDK.



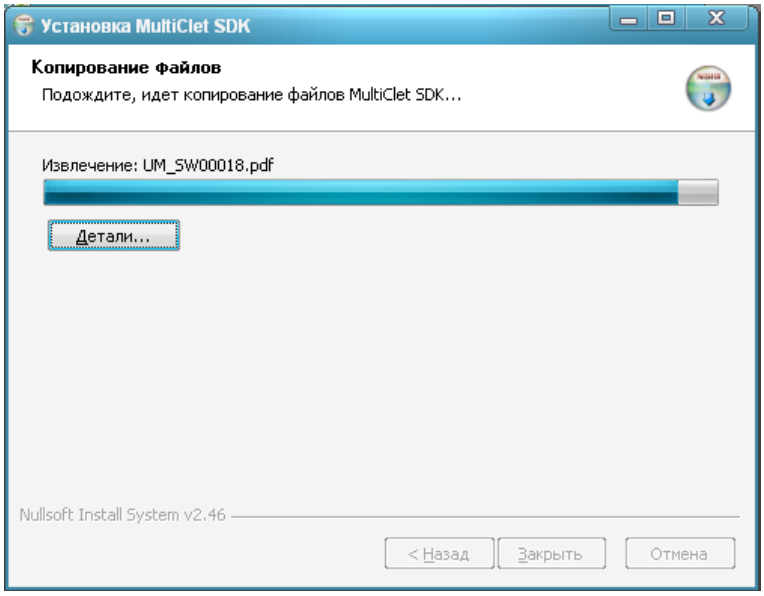
Затем нужно нажать кнопку «Далее».

На третьем шаге можно выбрать каталог, куда будут установлены файлы MultiCletSDK.

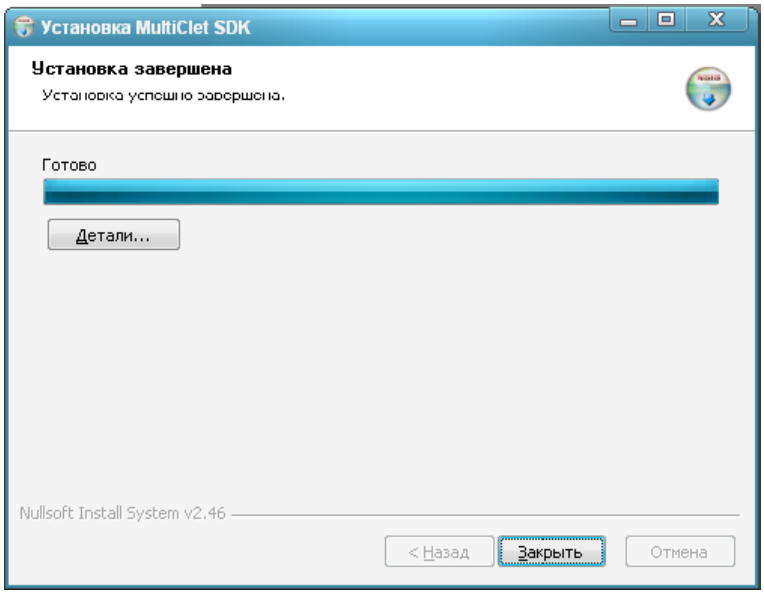


Затем нужно нажать кнопку «Установить».

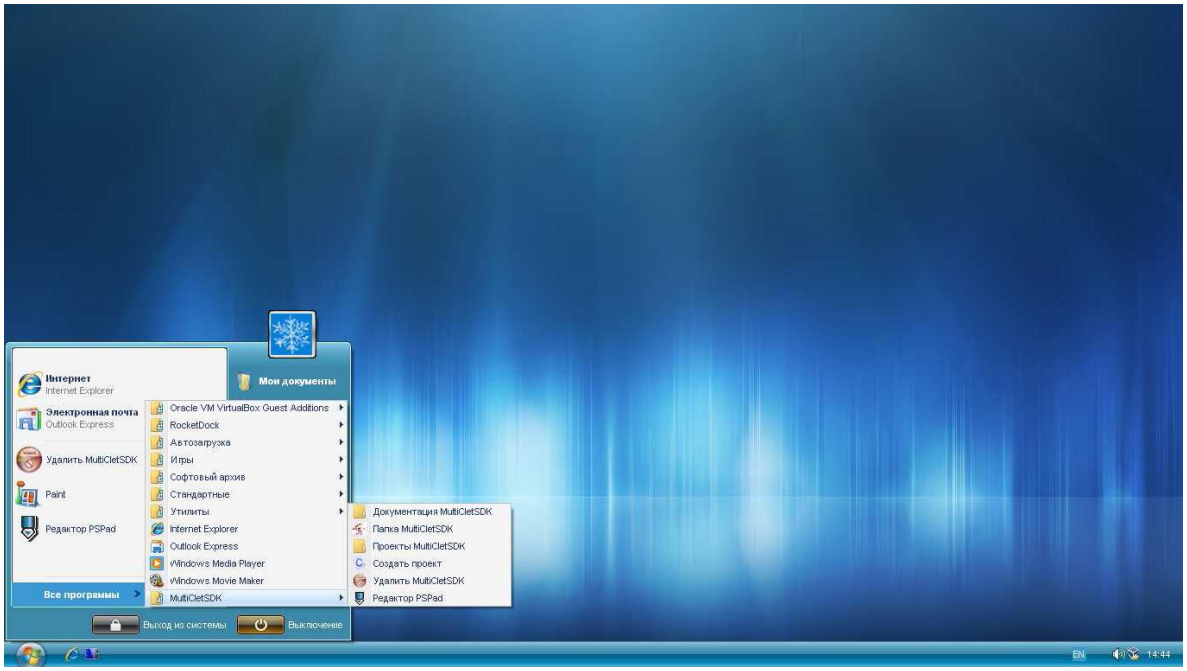
В процессе установки будет отображаться ход копирования файлов.



По завершении нужно нажать кнопку «Закрыть». На этом установка считается завершённой.



В результате завершения программы установки в меню «Пуск» будет добавлена группа ярлыков MultiCletSDK.



Чтобы приступить к работе необходимо выбрать в меню нужный пункт:

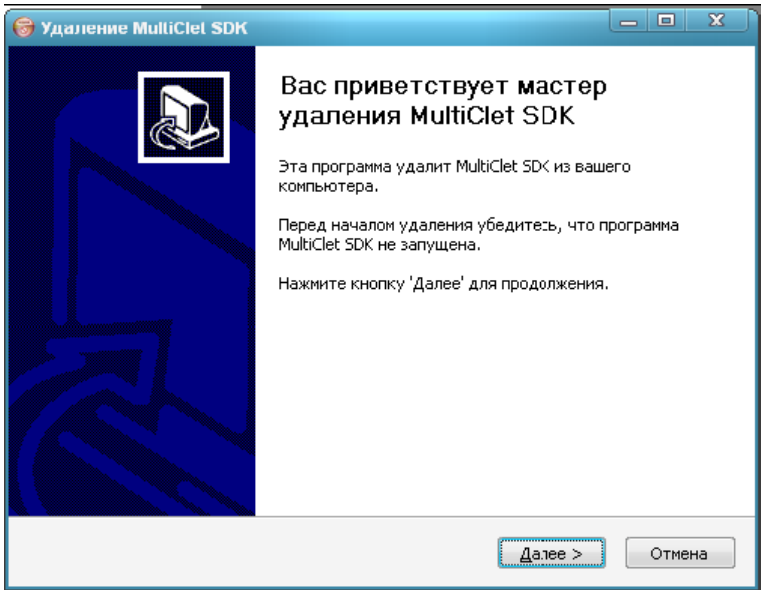
- Папка MultiCletSDK;
- Проекты MultiCletSDK;
- Создать проект;
- Удалить SDK;
- Редактор PSPad (запуск редактора PSPad).

1.2. Удаление программного обеспечения и документации

Удаление программного обеспечения и документации (далее MultiCletSDK) осуществляется специально созданным деинсталлятором.

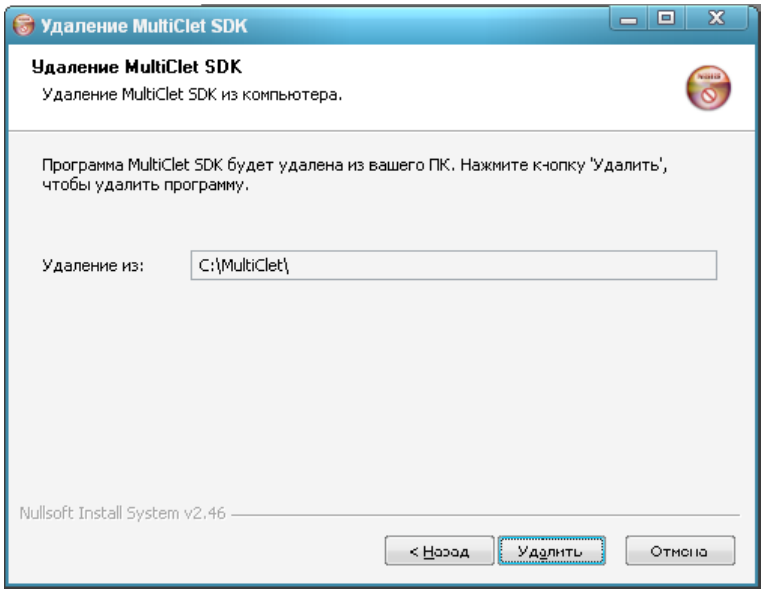
Для запуска процесса удаления нужно запустить файл uninstall.exe, который расположен в установочной директории MultiCletSDK, либо выбрать в меню «Пуск» пункт «MultiCletSDK\Удалить MultiCletSDK».

После запуска процесса удаления отображается окно приветствия с краткой информацией об удаляемом продукте.



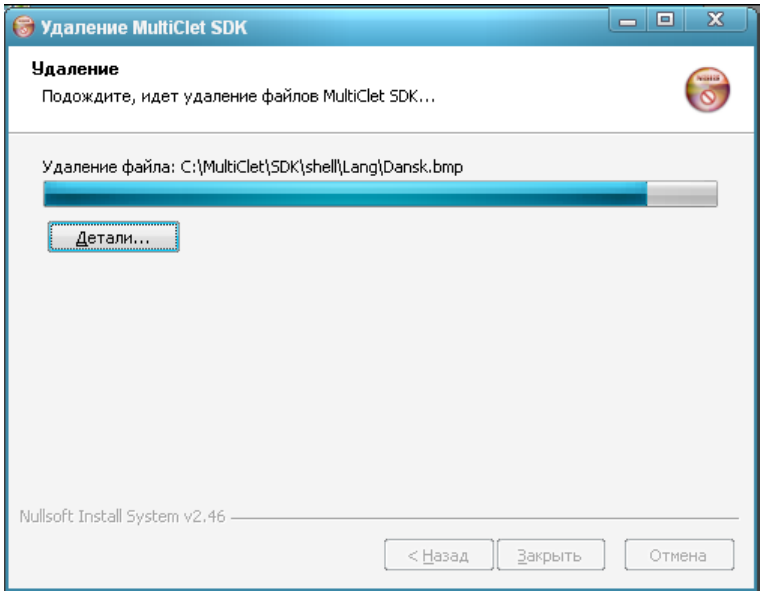
Для продолжения установки необходимо нажать кнопку «Далее».

На втором шаге отображается окно подтверждения удаления MultiCletSDK.

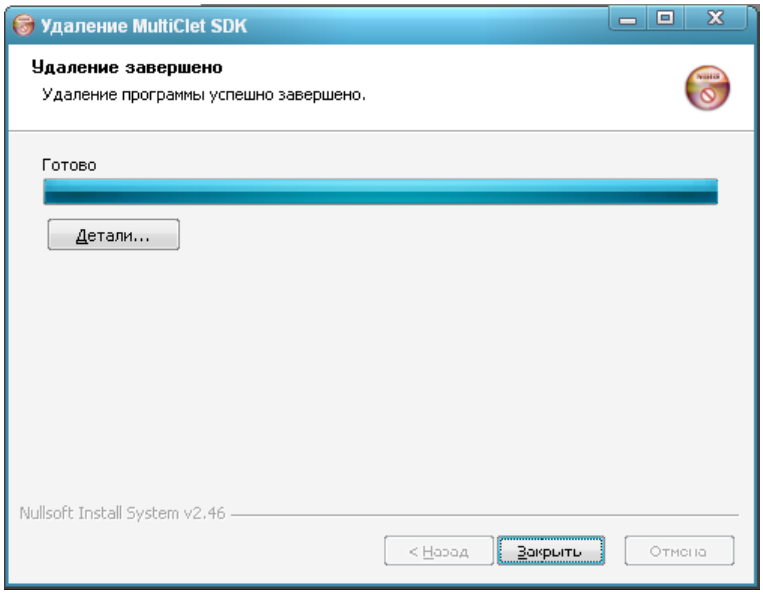


Для подтверждения удаления MultiCletSDK нужно нажать кнопку «Удалить».

В процессе удаления будет отображаться ход удаления файлов.



По завершении нужно нажать кнопку «Закрыть». На этом процесс удаления продукта считается завершённым.



2. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ ПО СРЕДЕ РАЗРАБОТКИ

2.1. Описание оболочки

Оболочка основана на базе свободно распространяемого редактора PSPad (http://www.pspad.com) со специальными настройками для работы с проектами MultiClet.

Редактор обладает следующими характеристиками:

- работа с проектами;
- работа над несколькими документами одновременно (MDI);
- сохранение экранной сессии, т. е. при следующем входе PSPad автоматически откроет все
- открытые на момент закрытия файлы;
- FTP клиент — вы можете редактировать файлы прямо с web-сервера;
- поддержка макросов: записывать, сохранять и загружать макросы;
- поиск и замена в файлах;
- сравнение текстов с разноцветной подсветкой различий;
- шаблоны (HTML-теги, скрипты, шаблоны кода...);
- инсталляция содержит шаблоны для HTML, PHP, Pascal, JScript, VBScript, MySQL, MSDos, Perl,...;
- подсветка синтаксиса применяется автоматически согласно типу файла;
- определяемые пользователем стили подсветки для экзотических синтаксисов;
- автокоррекция;
- интеллектуальный встроенный HTML-предпросмотр используя IE и Mozilla;
- полноценный HEX редактор;
- вызов внешних программ, отдельно для каждой среды разработки;
- внешний компилятор с перехватом вывода, окном лога и парсер логов для каждой среды
- создают эффект "IDE";
- цветная подсветка синтаксиса для печати и допечатный предпросмотр;
- интегрирована TiDy-библиотека для форматирования и проверки HTML-кода, конверсии
- в CSS, XML, XHTML;
- встроенная свободная версия CSS-редактора TopStyle Lite;

- экспорт кода с подсветкой в форматах RTF, HTML, TeX в файл или буфер обмена;
- вертикальное выделение, закладки, метки, нумерация строк ...;
- переформатирование и сжатие HTML-кода, изменения регистра слов, тэгов, букв;
- сортировка строк с возможностью сортировать по заданному столбцу, с параметром удаления дубликатов;
- таблица ASCII-символов с приведением соответствия HTML-мнемоник;
- навигатор кода для Pascal, INI, HTML, XML, PHP, и многих других в будущем;
- проверка правописания;
- встроенный веб-браузер с поддержкой APACHE;
- подсветка парных скобок.

Более подробно обо всех возможностях редактора и о том, как им пользоваться можно узнать из встроенной справке редактора, т. к. описать их все в рамках данного документа не представляется возможным.

Далее будут описаны лишь те особенности, которые присущи проектам MultiClet.

В процессе инсталляции за редактором закрепляются следующие расширения:

*.ppr - файл проекта;

*.asm - файл ассемблера MultiClet.

Расширения *.c, *.h не регистрируются, чтобы не перекрыть настройки системы по умолчанию, т. к. высока вероятность того, что в системе уже установлен редактор этих файлов.

2.2. Создание проектов

Создать новый проект можно двумя путями:

• через запуск скрипта создания нового проекта;

• средствами самого редактора (см. справку редактора).

В главном меню и в папке Multiclet есть ссылка на скрипт создания проекта: «Создать проект». При запуске скрипта создаётся папка нового проекта в папке с проектами, которая по умолчанию находится в папке "C:/Multiclet/Projects". В проект включён один файл-шаблон. Дальнейшее добавление файлов в проект производится при помощи средств самого редактора. Так же можно создать проект и в самом редакторе. Процедура создания нового проекта подробно описана во встроенной справке.

2.3. Компиляция проектов

Редактор настроен на компиляцию проектов Си и ассемблера MultiClet.

Для запуска компиляции должен быть открыт соответствующий проект.

Для запуска компиляции необходимо нажать Ctrl+F9, что приведёт к компиляции всех файлов Си и ассемблера, расположенных в директории и всех поддиректориях проекта. Ход компиляции (журнал) отображается в нижней части окна редактора. По завершении компиляции в журнале отображается результат компиляции. Если компиляция завершилась успешно, то код завершения равен 0, иначе отображается код ошибки.

2.4. Загрузка программы в ПЗУ

Для загрузки программы в ПЗУ необходимо нажать Alt+F9, что приведет к запуску окна загрузки программы в память. Процесс загрузки и очистки памяти пройдет автоматически. В качестве альтернативы данному сочетанию клавиш можно использовать соответствующие кнопки на панели управления PSPad.

2.5. Запуск программ на модели

После компиляции полученный файл образа памяти можно запустить на модели. Для запуска модели нужно нажать F9. В процессе выполнения модели могут появляться диагностические сообщения и другие подсказки.

2.6. Документация

Документация на SDK MultiClet находится в папке "C:/MultiClet/Docs":

Manual_Soft.pdf — Руководство пользователя по программному обеспечению.

Ограничения

В данной версии не поддерживается отладка программ. После завершения компиляции все действия по загрузке программы в процессор или запуск модели нужно осуществлять вручную.

3. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ ПО КОМПИЛЯТОРУ СИ

Компилятор для процессора MCp создан на основе LCC (перенацеливаемый компилятор для ANSI C -- стандарт C89). Используемый в проекте LCC 4.2 обеспечивает фасад компилятора: препроцессор, лексический анализатор, синтаксический анализатор, некоторую оптимизацию промежуточного кода (например, распространение констант), некоторые библиотеки. Компанией «Мультиклет» разработаны генератор ассемблерного кода из промежуточного представления LCC 4 и управляющий процессом компиляции драйвер, который позволяет получить образ памяти для загрузки на отладочную плату или в модель процессора.

3.1. Описание

В настоящее время генератор выдаёт код, соответствующий архитектуре и системе команд MCp0411100101. Поэтому, операции целочисленного деления и взятия остатка от деления реализованы через вызов функций, написанных на ассемблере. Они содержатся в файле crt0.s системной библиотеки.

В состав компилятора входят три программы:

- драйвер (lcc в Linux / lcc.exe в Windows);
- препроцессор (cpp / cpp.exe);
- транслятор (rcc / rcc.exe).

Драйвер использует так же исполняемые файлы:

- ассемблера (as / as.exe);
- редактора связей (ld / ld.exe);
- средства подготовки образа (convert / convert.exe).

Эти программы входят в предоставляемые компанией «Мультиклет» пакеты ассемблера (as) и редактора связей (ld, convert), а так же в интегрированную среду разработки.

3.2. Подготовка к работе

Для того, чтобы драйвер компилятора мог обеспечить полную сборку программы до пригодного к загрузке и исполнению образа памяти, необходимо в одной директории разместить исполняемые файлы:

- lcc

- cpp
- rcc
- as
- ld

В системах, с поддержкой ссылок на файлы вместо копий самих файлов в этой директории можно разместить ссылки на них.

3.3. Пакет MultiCletSDK

Пакет MultiCletSDK содержит все необходимые программы и файлы с ассемблером, подготовленные к работе.

3.4. Трансляция и сборка при помощи драйвера

Текущая версия компилятора требует указания в командной строки опции **-lccdir = LCCDIR**, где LCCDIR - директория с исполняемыми файлами: lcc, cpp, rcc, as, ld.

Поэтому, трансляция набора исходных файлов C89 в набор объектных файлов осуществляется командой:

- lcc -lccdir=LCCDIR -c f1.c ... fN.c

Сборка готового к исполнению образа памяти может быть осуществлена командой:

- lcc -lccdir=LCCDIR f1.c ... fN.c asm1.s ... asmM.s o1.o ...
- oL.o CRTDIR/crt0.0 -o image

где CRTDIR - директория, содержащая crt0.0. В этой команде проиллюстрировано то, что драйвер способен собрать образ программы из:

- исходных текстов на C89;
- исходных текстов на ассемблере MCp;
- заранее подготовленных объектных файлов.

Если опция -o, задающая имя собираемого образа, не будет использована в командной строке, то этот образ получит имя a.out в текущей директории.

3.5. Передача аргументов редактору связей

Иногда полезно передать некоторые аргументы командной строки редактору связей. Например, ключ -M, на который редактор откликается выдачей информации о размещении

символов, что полезно для отладки и анализа памяти после запуска программ на отладочной памяти или на программной модели процессора.

Опции редактору связей могут быть переданы при помощи ключа -Wm так, как это делается в следующем примере (в одной из Linux-систем разработчиков; \ указывает перенос команды на следующую строку):

```
$ cd /tmp/flc/MC/bld
$ lcc \
        -lccdir=./ \
        -Wm-M \
        ../lcc/multiclet/symbolic/tst/factorial.c \
        ..<INST_DIR>/SDK/lib/MCp0411100101/crt0.0 \
        -o fact.img
```

<INST_DIR> - установочная директория

3.6. Некоторые другие полезные ключи драйвера компиляции

-S драйвер обеспечит только лишь трансляцию исходных файлов на Си в ассемблер процессора MCp. Другие стадии построения исполняемого пройдены не будут.

3.7. Диагностические сообщения

Так как фасад компилятора построен на базе LCC, то и диагностические сообщения, связанные с лексической, синтаксической и семантической корректностью исходной программы являются стандартными для этой системы. Они достаточно подробны и информативны. Детальнее с ними можно познакомиться в инструкциях на сайте http://www.cs.virginia.edu/~lcc-win32/.

Кодогенератор компилятора может так же выдавать диагностические сообщения, связанные с текущими особенностями его реализации. Для текущей версии список сообщений таков:

1. <координаты в исходных текстах>: expression is too complex; please, consider to decompose it -- это сообщение выдаётся в случае, если выражение стоящее в указанной позиции

в исходных текстах (файл, строка) компилятору не удалось уложить в ограничения, накладываемые на структуру параграфа.

В будущих версиях проблема будет решена.

3.8. Особенности ранней версии компилятора.

Текущая версия компилятора не поддерживает операции присваивания блоков данных. То есть, следующие конструкции:

1. Инициализация константами автоматических массивов и структур.

2. Присваивание одной структуры другой, то есть, конструкции вида:

```
struct S s1, s2;

...

s1 = s2;
```

3. Вызовы функций, возвращающих структуры, или принимающие их в качестве аргументов (именно структуры, на указатели никаких ограничений не накладывается).

В будущих версиях компилятора (выходящих после 22.07.2012) проблема будет решена.

3.9. Сборка программы

Рассмотрим пример программы, написанной на языке Си, реализующей алгоритм сортировки методом пузырька.

```
#define SIZE 256
int array[SIZE];

void main(void)
{
        int i;
        int j;
        int c = 0;
```

```
        for(i = 0; i < SIZE; i += 1)
        {
                c = 255 - i;
                array[i] = c;
        }

        for(j = SIZE-1; j; j -= 1)
        {
                for(i = 0; i < j; i += 1)
                {
                        if(array[i] > array[i + 1])
                        {
                                int tmp = array[i];
                                array[i] = array[i + 1];
                                array[i + 1] = tmp;
                        }
                }
        }
}
```

Предположим, что данная программа сохранена в файле с именем bubble.c. Процесс сборки программы, или, другими словами, получение файла, содержащего образы памяти программ и памяти данных исполняемой программы, необходимого для загрузки в ПЗУ отладочной платы или выполнения на модели, состоит из следующих этапов:

1. Обработка файла bubble.c препроцессором Си, для чего необходимо в командной строке выполнить следующую команду:

   ```
   cpp bubble.c bubble.i
   ```

   Препроцессор выполняет удаление из исходного кода комментариев, а также осуществляет обработку препроцессорных директив, начинающихся с символа «#», таких как #define, #include и других.

2. Компиляция файла bubble.i компилятором Си, для чего необходимо в командной строке выполнить следующую команду:

```
rcc bubble.i bubble.s
```

Компилятор Си выполняет трансляцию программы на языке Си в эквивалентную программу на языке ассемблера.

3. Компиляция файла bubble.s компилятором ассемблерного кода, для чего необходимо в командной строке выполнить следующую команду:

```
as -obubble.o bubble.s
```

Ассемблирование исходного кода программы на языке ассемблера в объектный файл, содержащего блоки машинного кода и данных программы, с неопределенными адресами символов на данные и процедуры в других объектных файлах, а также список своих процедур и данных.

Опция компилятора ассемблерного кода «-o» задаёт имя выходного объектного файла.

4. Сборка файла, содержащего образы памяти программ и памяти данных исполняемой программы из одного или нескольких объектных файлов, для чего необходимо в командной строке выполнить следующую команду:

```
ld -M -oimage.bin crt0.o bubble.o
```

Опция компоновщика «-M» указывает на необходимость вывода в стандартный поток вывода информации о размещении данных объектных файлов в памяти и значениях, назначенных символам.

Опция компоновщика «-o» задаёт имя выходного файла, содержащего образы памяти программ и памяти данных исполняемой программы.

Объектный файл crt0.o предоставляет набор стартовых исполняемых процедур, которые выполняют необходимую инициализацию перед вызовом главной функции программы. Данный файл должен располагаться первым в списке файлов, передаваемых на вход компоновщику. Для программ, исходный код которых написан только на языке ассемблера, файл crt0.o может не использоваться.

Процесс сборки программы может быть выполнен драйвером сборки lcc, для чего необходимо в командной строке выполнить следующую команду:

```
lcc -target=mcp/win32 -Wl-M -oimage.bin crt0.o bubble.s
```

В этом случае драйвер выполнит описанную выше последовательность действий самостоятельно.

Полученный файл образов памяти исполняемой программы (image.bin) может быть:

- загружен в ПЗУ отладочной платы, для чего необходимо в командной строке выполнить следующую команду:

```
ploader image.bin
```

- выполнен на функциональной модели, для чего необходимо в командной строке выполнить, например, следующую команду:

```
model \
    -dump-raw \
    -dump-addr \
    -dump-long \
    -dump-length 4 \
    -dump-from 0x00000088 \
    -dump-to 0x00000488 \
    image.bin
```

Опции «-dump-from» и «-dump-to» задают начальный и конечный адреса блока памяти данных, содержимое которого необходимо вывести на печать. Начиная с адреса 0x00000088 памяти данных, компоновщиком был размещён массив array, элементы которого в результате выполнения программы будут упорядочены по возрастанию.

Описание всех опций запуска функциональной модели можно узнать выполнив в командной строке команду:

```
model.exe --help
```

4. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ ПО АССЕМБЛЕРУ

4.1. Общие сведения о мультиклеточном процессоре

Мультиклеточное процессорное ядро — первое 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.

Мультиклеточный процессор состоит из 4 клеток (когерентных процессорных блоков), объединенных интеллектуальной коммутационной средой, и с раздельной памятью программ и данных.

Система команд ядра мультиклеточного процессора имеет два формата (*AA* - размерностью слово (32 бита) и *AV* — размерностью двойного слова (64 бита)).

Команды ядра работают со следующими типами данных:

- знаковый/беззнаковый байт, размерностью 8 бит;
- знаковое/беззнаковое слово, размерностью 32 бита;
- беззнаковое двойное слово, размерностью 64 бита;
- знаковый вещественный, размерностью 32 бита;
- знаковый вещественный упакованный, размерностью 64 бита;
- знаковый вещественный комплексный, размерностью 64 бита.

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

Мультиклеточное ядро аппаратно обеспечивает реализацию параллелизма на операторном уровне «естественным» образом, без решения задачи распараллеливания.

4.1.1. Память программ (ПП)

Память программ представляет собой независимые блоки статических оперативно-запоминающих устройств (ПП0 – ПП3), число которых равняется числу процессорных блоков (клеток). Соответственно, каждый процессорный блок имеет свою собственную память программ. Указанные блоки памяти не связаны между собой и функционируют независимо.

Для пользователя память программ работает только в режиме чтения и используется только для хранения программного алгоритма. Для констант используется выделенная для этого область в памяти данных. Адресация происходит к 64-х разрядному двойному слову.

На рис. 1 показана логическая адресация ПП для процессора, состоящего из 4-х процессорных блоков.

ПП0	ПП1	ПП2	ПП3
$0x0000$	$0x0000$	$0x0000$	$0x0000$
$0x0001$	$0x0001$	$0x0001$	$0x0001$
...	...	...	...
$0x0fff$	$0x0fff$	$0x0fff$	$0x0fff$

Рис.1 – Структура памяти данных

Программа процессора рассматривается как набор последовательно размещенных параграфов. Параграф — это группа предложений, которая записана последовательно, имеет один вход и один выход. В свою очередь, предложение — группа информационно связанных операций. Каждый параграф размещается, начиная с ПП0. Команды параграфа размещаются последовательно. Каждая очередная команда размещается в сегменте ПП, принадлежащем очередному процессорному блоку. При этом, команда формата □□ полностью размещается в одном сегменте. Так, например, следующая последовательность команд, образованная двумя параграфами:

«av0,av1,aa2,aa3,aa4,av5» «aa6,aa7,av8,av9,aa10,av11,av12»

может быть размещена так, как показано на рис. 2. Порядок считывания инструкций показан увеличением насыщенности цветового тона.

Адрес	ПП0		ПП1		ПП2		ПП3	
$0x0000$	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
$0x0zzz + 0x00$	$av0$	$av0$	$av1$	$av1$	$aa2$	0	$aa3$	0
$0x0zzz + 0x01$	$aa4$	0	$av5$	$av5$	0	0	0	0
$0x0zzz + 0x02$	$aa6$	$aa10$	$aa7$	$av11$	$av8$	$av8$	$av9$	$av9$
$0x0zzz + 0x03$		$aa10$	$aa7$	$av11$	$av8$	$av8$	$av9$	$av9$
...	...	...	...	...	...	...	...	...
$0x0fff$	...	...	...	...	...	...	...	...

Рис.2 – Размещение параграфов в ПП

4.1.2. Память данных (ПД)

Память данных представляет собой независимые блоки статических оперативно-запоминающих устройств (ПД0 – ПД3), число которых равняется числу процессорных блоков (клеток), адресация происходит побайтно. Особенностью организации памяти данных является то, что ячейки со смежными адресами находятся в разных блоках памяти данных. Для сокращения времени доступа к памяти рекомендуется данные выравнивать на 8 байт.

Адресация памяти данных для 4-х сегментов показана на рис. 3.

Адрес (базовый)	ПД0	ПД1	ПД2	ПД3
	Смещение адреса			
0x00000	0x00000 ... 0x00007	0x00008 ... 0x0000f	0x00010 ... 0x00017	0x00018 ... 0x0001f
0x00020	0x00000 ... 0x00007	0x00008 ... 0x0000f	0x00010 ... 0x00017	0x00018 ... 0x0001f
...	...	...	...	...
0x1ffe0	0x00000 ... 0x00007	0x00008 ... 0x0000f	0x00010 ... 0x00017	0x00018 ... 0x0001f

Рис.3 – Структура памяти данных

4.1.3. Регистры

Мультиклеточный процессор имеет в своем составе следующие регистры:

• регистры общего назначения (РОН [GPR — General Purpose Register]);

• регистры индексные (РИ [IR — Index Register]);

• регистры управляющие (РУ [CR — Control Register]).

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

4.1.3.1. Регистры общего назначения

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

4.1.3.2. Регистры индексные

Используются для косвенной адресации. Логическая структура индексного регистра показана на рис.4.

Номера битов	63…48	47…32	31…0
	Индекс (*Index*)	Маска (*Mask*)	База (*Base*)

Рис.4 – Структура индексного регистра

Для обращения к какому-либо регистру данного типа используются номера от 32 до 39. Имен у данного типа регистров нет. В общем случае (см. исключения в описании конкретной команды в разделе «Система команд ассемблера») при использовании регистра данного типа в качестве аргумента операции значение этого аргумента формируется согласно следующему алгоритму:

1. вычисление исполнительного адреса, согласно следующей формуле:

$$Address = Index + Base$$

2. обращение к памяти данных по исполнительному адресу $Address$ для чтения/записи значения аргумента согласно типу используемой команды.

Модификация значения индексного регистра осуществляется аппаратно по завершению параграфа в том случае, если установлен соответствующий бит регистра MODR маски изменения индексных регистров (см. раздел «Регистры управления»), согласно следующей формуле:

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

где | — операция побитового «ИЛИ», & — операция побитового «И», ~ — операция побитового инвертирования.

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

4.1.3.3. Регистры управляющие

Процессор имеет в своем составе следующие управляющие регистры:

Номер регистра	Имя регистра	Права доступа	Описание
48	PSW	R/W	Регистр управления вычислительным процессом
49	INTR	R/W	Регистр прерываний
50	MSKR	R/W	Регистр маски прерываний
51	ER	R	Регистр исключений
52	IRETADDR	R	Регистр адреса возврата
53	STVALR	R/W	Регистр периода системного таймера
54	STCR	R/W	Регистр управления системным таймером
55	IHOOKADDR	R/W	Регистр первичного обработчика прерываний
56	INTNUMR	R	Регистр номера прерывания
57	MODR	R/W	Регистр маски модификации индексных регистров

4.1.3.4. Регистр PSW управления вычислительным процессом

Регистр управления вычислительным процессом предназначен для управления вычислительным процессом

Структура регистра управления вычислительным процессом показана на рис.5

PSW	Регистр управления PSW																												
Номер бита	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Операции	R = 0																									RW	RW	RW	R	RW	R	RW
Начальное состояние	0																									0	0	0	0	0	0	0
Описание	RESERVED																									RW	SWR	STOP	RESERVED	SUSPEND	RESERVED	ONIRQS

Рис.5 – Структура регистра управления вычислительным процессом

Назначение разрядов регистра:

Для всех разрядов регистра:

0 — отсутствие признака или события

1 — наличие признака или события

0 Разрешение обработки немаскируемых прерываний. Признак не блокирует прием прерываний регистром INTR, а блокирует только их дальнейшую обработку. Устанавливается всегда только программно, снимается программно и аппаратно (при переходе на программу обработки прерываний).

1 Зарезервировано

2 Переход в режим ожидания. По окончании параграфа, в котором устанавливается данный бит, ядро переходит в режим ожидания. В этом состоянии оно находится до прихода прерывания. По приходу прерывания данный признак снимается аппаратно и управление передается на программу обслуживания прерываний. По окончании обслуживания прерывания, если программистом не заданы какие-либо действия, продолжается выполнения приостановленной программы.

3 Зарезервировано

4 Остановка ядра. При установке в 1, по завершению текущего параграфа, ядро прекращает выборку команд. Возобновление работы возможно только извне, путем аппаратного сброса или подачи на вход «wake_up» процессора напряжения уровня логической «1».

5 Программный сброс. При установке в 1, по завершению параграфа процессор проходит полную инициализацию, ядро автоматически начинает работу с исходного состояния, регистры сбрасываются в исходное состояние. Содержимое оперативной памяти не сохраняется.

6 Очередность исполнения команд чтения/записи:

0 — команды записи исполняются только после выполнения всех команд чтения данного параграфа

1 — контроль очередности не выполняется

7 - 31 Зарезервировано

4.1.3.5. Регистр INTR прерываний

Регистр прерываний предназначен для сигнализации наличия прерываний, а также инициации программных прерываний.

Структура регистра прерываний на рис. 6

INTR

Регистр прерываний INTR																															
Номер бита	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Операции	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Начальное состояние	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Описание	USB0_IRQ	ETHERNET0_IRQ	GPIOD_IRQ	GPIOC_IRQ	GPIOB_IRQ	GPIOA_IRQ	RTC_IRQ	PWM0_IRQ	GPTIM6_IRQ	GPTIM5_IRQ	GPTIM4_IRQ	GPTIM3_IRQ	GPTIM2_IRQ	GPTIM1_IRQ	GPTIM0_IRQ	I2S0_IRQ	SPI2_IRQ	SPI1_IRQ	SPI0_IRQ	I2C1_IRQ	I2C0_IRQ	UART3_IRQ	UART2_IRQ	UART1_IRQ	UART0_IRQ	SWI	SWT	MPRGE	PPGE	PERE	ENMI	INMI

Рис.6 – Структура регистра прерываний

Назначение разрядов регистра:

Для всех разрядов регистра:

0 — отсутствие признака или события

1 — наличие признака или события

0 Немаскируемое внутреннее прерывание (INMI)

1 Немаскируемое внешнее прерывание (ENMI)

2 Немаскируемое исключение в аппаратной части (PERE)

3 Немаскируемое программное исключение (PPGE)

4 Маскируемое программное исключение (MPRGE)

5 Прерывание от системного таймера (SWT)

6 Программное прерывание (SWI)

7 Маскируемое прерывание от UART0

8 Маскируемое прерывание от UART1

9 Маскируемое прерывание от UART2

10 Маскируемое прерывание от UART3

11 Маскируемое прерывание от I2C0

12 Маскируемое прерывание от I2C1

13 Маскируемое прерывание от SPI0

14 Маскируемое прерывание от SPI1

15 Маскируемое прерывание от SPI2

16 Маскируемое прерывание от I2S0

17 Маскируемое прерывание от GPTIM0

18 Маскируемое прерывание от GPTIM1

19 Маскируемое прерывание от GPTIM2

20 Маскируемое прерывание от GPTIM3

21 Маскируемое прерывание от GPTIM4

22 Маскируемое прерывание от GPTIM5

23 Маскируемое прерывание от GPTIM6

24 Маскируемое прерывание от PWM0

25 Маскируемое прерывание от RTC

26 Маскируемое прерывание от GPIOA

27 Маскируемое прерывание от GPIOB

28 Маскируемое прерывание от GPIOC

29 Маскируемое прерывание от GPIOD

30 Маскируемое прерывание от ETHERNET0

31 Маскируемое прерывание от USB0

4.1.3.6. Регистр MSKR маски прерываний

Регистр маски прерываний предназначен для маскирования прерываний.

Структура регистра маски прерываний на рис. 7

MSKR

Регистр маски прерываний MSKR																													
Номер бита	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Операции	$R=0$																RW															
Начальное состояние	0																0															
Описание	Зарезервировано																Маскируемые прерывания															

Рис. 7 – Структура регистра маски прерываний

Назначение разрядов регистра:

0 — 27 Маска для 4 — 31 битов регистра INTR

0 — запрет обработки запроса прерывания

1 — разрешение обработки прерывания

27 — 31 Зарезервировано

4.1.3.7. Регистр ER ошибок

Структара регистра ошибок на рис. 8

ER	Регистр ошибок ER																													
Номер бита	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Операции	R = 0																RC	RC	RC	RC	RC	RC	RC	RC	RC	RC	RC	RC	R = 0	R = 0	RC	
Начальное состояние	0																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Описание	Зарезервировано																II3	II1	II2	II0	AI3	AI2	AI1	AI0	DZ3	DZ2	DZ1	DZ0	RESERVED	RESERVED	AMBATR_ER	

Рис. 8 – Структура регистра ошибок

Назначение разрядов регистра:

Для всех разрядов регистра:

0 — отсутствие признака или события

1 — наличие признака или события

0 — 1 Ошибки с шиной

2 Зарезервировано

3 Зарезервировано

4 Ошибка «деление на ноль» в 0 ядре

5 Ошибка «деление на ноль» в 1 ядре

6 Ошибка «деление на ноль» во 2 ядре

7 Ошибка «деление на ноль» в 3 ядре

8 Ошибка «неверный адрес» в 0 ядре, сформирован не правильный адрес

9 Ошибка «неверный адрес» в 1 ядре

10 Ошибка «неверный адрес» во 2 ядре

11 Ошибка «неверный адрес» в 3 ядре

12 Ошибка «неверная инструкция» в 0 ядре, из памяти программ прочитана несуществующая инструкция

13 Ошибка «неверная инструкция» в 1 ядре

14 Ошибка «неверная инструкция» во 2 ядре

15 Ошибка «неверная инструкция» в 3 ядре

16 - 31 Зарезервировано

4.1.3.8. Регистр IRETADDR адреса возврата

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

Структара регистра первичного обработчика прерываний на рис. 9

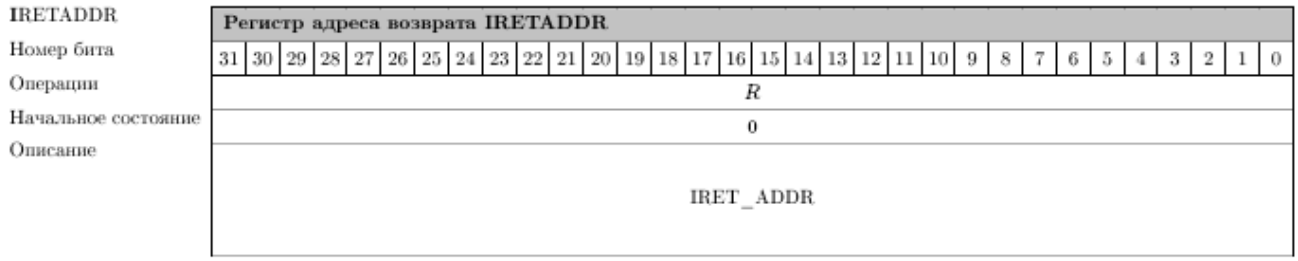


Рис. 9 – Структура регистра адреса возврата

Назначение разрядов регистра:

0 — 7 Адрес возврата, формируемый только при прерывании

8 — 31 Зарезервировано

4.1.3.9. Регистр STVALR периода системного таймера

Регистр периода системного таймера предназначен для сохранения значения периода системного таймера и доступен только для чтения.

Структура регистра периода системного таймера на рис. 10

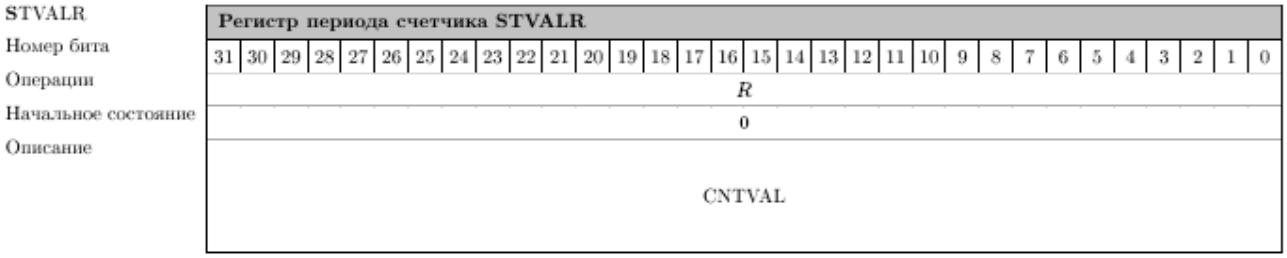


Рис. 10 – Структура регистра периода системного таймера

Назначение разрядов регистра:

0 — 31 Значение периода счетчика в периодах системной частоты после предделителя системного таймера

4.1.3.10. Регистр STCR управления системным таймером

Регистр управления системным таймером предназначен для управления системным таймером. Системный таймер предназначен для формирования заданных периодических или однократных временных интервалов. Таймер представляет собой декрементирующий счетчик с делителем тактового сигнала на входе. Начальное значения счетчика записывается в регистр STVALR, управление осуществляется через регистр STCR. По истечении заданного временного интервала формируется запрос на обработку прерывания

Структура регистра управления системным таймером на рис. 11

STCR

Регистр управления счетчиком STCR																												
Номер бита	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Операции	$R=0$																RW								$R=0$					RW	RW	RW
Начальное состояние	0																0								0					0	0	0
Описание	Зарезервировано																PREDIV								Зарезервировано					CNTCMP	ENCCNT	EN

Рис. 11 – Структура регистра управления системным таймером

Назначение разрядов регистра:

0 Разрешение работы счетчика

0 — запрещено

1 — разрешено

1 Разрешение циклической работы таймера (если запрещена циклическая работа, то таймер после одного периода выключится, бит 0 данного регистра будет установлен в «0»):

0 — запрещено

1 — разрешено

2 Признак завершения счета периода, заданного в STVAL (по данному признаку вырабатывается запрос на обработку прерывания от таймера):

0 — запрещено

1 — разрешено

3- 7 Зарезервировано

8 – 15 Значение предделителя счетчика. Системная частота делится на значение, заданное в данных битах. Поделенная частота является тактовой для счетчика таймера.

16 - 31 Зарезервировано

4.1.3.11. Регистр IHOOKADDR первичного обработчика прерываний

Регистр первичного обработчика прерываний предназначен для сохранения адреса памяти программ первичного обработчика прерывания.

Структура регистра первичного обработчика прерываний на рис. 12

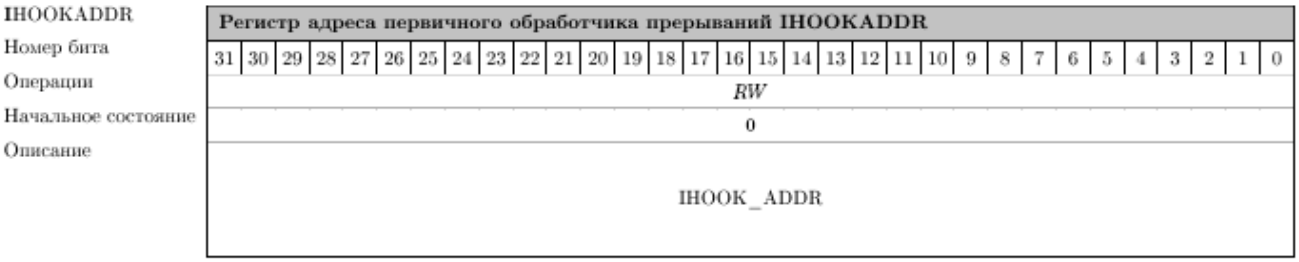


Рис. 12 – Структура регистра первичного обработчика прерываний

Назначение разрядов регистра:

0 - 7 Значение адреса первичного обработчика прерываний

8 - 31 Зарезервировано

4.1.3.12. Регистр INTNUMR номера прерывания

Регистр номера прерывания предназначен для сохранения номера самого приоритетного разрешенного прерывания на данный момент и доступен только для чтения.

Структура регистра номера прерывания показана на рис. 13

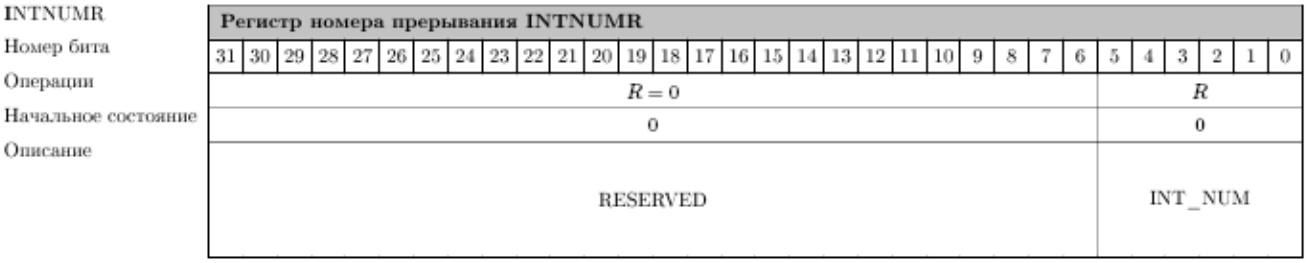


Рис. 13 – Структура регистра номера прерывания

Назначение разрядов регистра:

0 - 5 Номер самого приоритетного разрешенного прерывания на данный момент

6 - 31 Зарезервировано

4.1.3.13. Регистр MODR маски модификации индексных регистров

Регистр маски модификации индексных регистров предназначен для указания необходимости пересчёта значения индексного регистра, номер которого определяется согласно формуле 32 + i, где i — номер бита регистра MODR, по завершению выполнения операций параграфа. Правило изменения значения индексного регистра смотри в разделе «Регистры индексные, 16.3.2».

Структура регистра маски модификации индексных регистров показана на рис. 14.

MODR	Регистр модификации индексных регистров MODR																											
Номер бита	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Операции	R = 0																								RW	RW	RW	RW	RW	RW	RW	RW
Начальное состояние	0																								0	0	0	0	0	0	0	0
Описание	RESERVED																															

Рис. 14 – Структура регистра маски модификации индексных регистров

Назначение разрядов регистра:

0 Разрешение модификации индексного регистра номер 32:

0 — запрещено

1 — разрешено

1 Разрешение модификации индексного регистра номер 33:

0 — запрещено

1 — разрешено

2 Разрешение модификации индексного регистра номер 34:

0 — запрещено

1 — разрешено

3 Разрешение модификации индексного регистра номер 35:

0 — запрещено

1 — разрешено

4 Разрешение модификации индексного регистра номер 36:

0 — запрещено

1 — разрешено

5 Разрешение модификации индексного регистра номер 37:

0 — запрещено

1 — разрешено

6 Разрешение модификации индексного регистра номер 38:

0 — запрещено

1 — разрешено

7 Разрешение модификации индексного регистра номер 39:

0 — запрещено

1 — разрешено

8 - 31 Зарезервировано

4.1.4. Коммутатор

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

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

Предложение — группа информационно связанных операций.

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

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

$$N = Nreq - Nres,$$

где $Nreq$ — номер команды, запрашивающей результат текстуально предшествующей команды, $Nres$ — 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.

4.1.5. Выборка команды

Начальный адрес памяти прогграмм, с которого процессорные блоки начинают выборку команд 0.

Необходимыми условиями выборки и последующего декодирования очередной команды всеми процессорными блоками является завершение декодирования предыдущей команды всеми процессорными блоками, считывание и размещение очередной команды на регистре команд всеми процессорными блоками (у каждого процессорного блока свой регистр команд).

Выборка и декодирование команд продолжается до тех пор пока не будет выбрана последняя команда параграфа. Адрес нового параграфа может поступить как в любой момент выборки команд текущего параграфа, так и после завершения выборки команд. Он поступает всем процессорным блокам одновременно. Если к моменту выборки последней команды параграфа адрес следующего параграфа не вычислен, то выборка приостанавливается до получения адреса. Если после завершения выполнения всех команд параграфа адрес следующего параграфа не сформирован, то процессор перходит в состояние ожидания сигнала «start» (ожидание внешнего перезапуска).

4.2. Общие сведения об ассемблере

Представленный ассемблер используется для написания программ для мультиклеточного процессора. Ассемблер обладает следующими характеристиками:

- использование раздельной памяти команд для каждого процессорного блока (клетки) и общей памяти данных;
- 32-х разрядная адресация памяти команд;
- 32-х разрядная адресация памяти данных;

Ассемблер может быть запущен на IBM PC совместимых компьютерах обладающих следующими характеристиками:

• операционная система: Microsoft Windows 2000, XP, Vista, 7.

Минимальные требования к аппаратной части:

• процессор не ниже 500МГц;

• оперативная память не менее 512Мб;

• видеокарта должна поддерживать минимально разрешение 1024x768 при 256 цветах и аппаратное ускорение функций DirectX версий 5.0 и выше.

4.2.1. Запуск ассемблера и опции командной строки

Представленный ассемблер запускается из командной строки командой as, аргументом которой является файл с исходным кодом. Поддерживаются также следующие опции:

-I, - -include-path=DIR — добавить директорию DIR в список директорий, используемых для поиска файла, подключаемого директивой ассемблера «.include»

-o, - -output=objfile — поместить вывод в объектный файл objfile

-V — напечатать номер версии ассемблера

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

4.3. Основные понятия языка

4.3.1. 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.

4.3.2. Константы

В ассемблере поддерживаются числовые и символьные (литеральные) константы.

4.3.2.1. 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.

4.3.2.2. 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 выражение является машинный код, размером в один байт, символа (литерала). as использует ASCII кодировку символов (литералов): ´A имеет целочисленное значение 65, ´B имеет целочисленное значение 66, и так далее.

4.3.3. 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. Следует заметить, что ввиду особой организации памяти программ мультиклеточного процессора, в которой размещаются исполняемые инструкции программы, длина секции .text, в которой ассемблер расположил исполняемые инструкции программы, может увеличиться из-за выравнивания параграфов (подробнее см. раздел «Память программ (PM)»).

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, которые в процессе компоновки будут размещены в памяти программ.

2. секция .data. В этой секции располагаются начальные данные программы, которые в процессе компоновки будут размещены в инициализируемой области памяти данных.

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.

В процессе создания объектного файла секция .text размещается с адреса 0 памяти программ; секция .data размещается с адреса 0 памяти данных, следом за которой размещается секция .bss.

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».

4.3.3.1. Подсекции

Ассемблированные байты 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.

4.3.4. Символы

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

4.3.4.1. 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.

4.3.4.2. 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.

4.3.4.3. Символы с абсолютным значением

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

4.3.4.4. Атрибуты символов

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

4.3.5. Выражения

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

4.3.5.1. 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.

4.3.5.2. Целочисленные выражения

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

4.3.5.3. Аргументы

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

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

Числа, как правило, являются целыми. Если выражение состоит из одного аргумента, который представляет собой число с плавающей точкой, то результатом вычисления выражения будет именно это число без каких-либо преобразований. Если выражение состоит из нескольких аргументов, разделенных операторами, либо из префиксного оператора, за которым следует один единственный аргумент, то при вычислении выражения значение аргумента, которое представляет собой число с плавающей точкой, будет заменено на нулевое значение, а также выведено соответствующее предупреждение.

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

4.3.5.4. Операторы

Операторы представляют собой арифметические функции и делятся на префиксные и инфиксные.

Префиксные операторы являются одно аргументными. 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 двух аргументными. 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.

4.4. Система команд ассемблера

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

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

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

4.4.1. Условные обозначения

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

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

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

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

$\#IRindex$, $\#IRmask$, $\#IRbase$ — обозначает индекс, маску и базу соответственно индексного регистра IR.

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

V 8, V 32 — обозначает непосредственное значение размером байт (8 бит), 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

4.4.2. 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 четыре байта (32 бита);

• беззнаковый целый, размерностью восемь байтов (64 бита);

• знаковый вещественный одинарной точности, размерностью четыре байта (32 бита);

• знаковый упакованный, размерностью 8 байтов (64 бита), старшршие 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);

• знаковый комплексный, размерностью 8 байтов (64 бита), старшршие четыре байта (с 32 по 63 биты) представляют собой реальную часть, а младшие четыре байта (с 0 по 31биты) представляют собой мнимую часть (тип реальной и мнимой частей — вещественный одинарной точности);

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

В таблице ниже показаны мнемоники суффиксов типов операций.

Тип операции	Беззнаковый	Знаковый
Целый, размерностью один байт	b	sb
Целый, размерностью четыре байта	l	sl
Целый, размерностью восемь байтов	q	-
Знаковый вещественный одинарной точности, размерностью четыре байта	-	f
Знаковый упакованный, размерностью 8 байтов	-	p
Знаковый комплексный, размерностью 8 байтов	-	c

4.4.3. Общий принцип построения команд в ассемблере

На рисунке 15 изображён общий синтаксис команд мультиклеточного процессора.

4.4.3.1. Общие правила формирования аргументов команд и их интерпретация

В общем случае команды могут быть двух-агументными, одно-аргументными и без аргументов. Согласно синтаксическому описанию (рис. 15), в качестве первого аргумента двух-аргументной команды всегда используется результат выполнения одной из 63-х предыдущих команд, содержащийся в коммутаторе и заданный ссылкой на это значение.

Второй аргумент двух-аргументной команды или аргумент одно-аргументной 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 регистра общего назначения ($\#GPR$) или значение управляющего

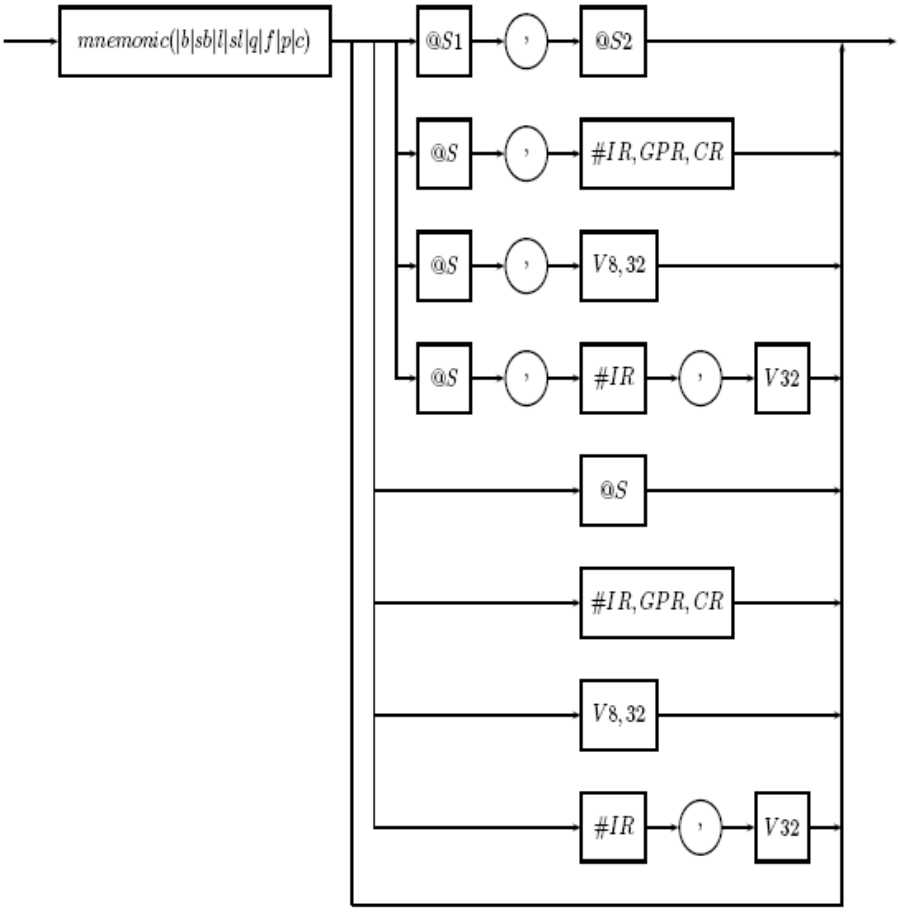


Рис.15 – Синтаксическое описание команды мультиклеточного процессора

регистра ($\#CR$)

• используется значение, считанное из памяти данных по адресу, вычисляемому на основании значения индексного регистра ($\#IR$), согласно формуле

$$\#IRbase + \#IRindex$$

• используется непосредственное значение, заданное в командном слове, размерностью байт (V 8) или слово (V 32) в зависимости от типа команды

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

$$\#IRbase + \#IRindex + V\ 32$$

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

4.4.3.2. Правила формирования результатов команд

В общем случае команды формируют результат, сохраняемый в коммутаторе. Размер значения результата любой команды соответствует размеру коммутатора — 64 бита.

Результат нагружен значениями флагов:

Флаг	Перевод	Описание
ZF (Zero Flag)	Флаг нуля	Устанавливается в случае равенства нулю всех разрядов результата команды
OF (Overflow Flag)	Флаг переполнения	Устанавливается в случае потери значащего бита
CF (Carry Flag)	Флаг переноса	Устанавливается в случае выхода за разрядную сетку результата команды
SF (Sign Flag)	Флаг знака	Устанавливается равным значению старшего (знакового) разряда результата команды

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

4.4.4. Описание команд

4.4.4.1. abs (ABSolute value)

Абсолютное значение

$abs\ ARG$

Назначение: операция вычисления абсолютного значения аргумента

Синтаксис:

Наличие результата: да

Алгоритм работы:

– вычислить абсолютное значение аргумента ARG;

– установить флаги;

– поместить результат вместе с флагами в коммутатор;

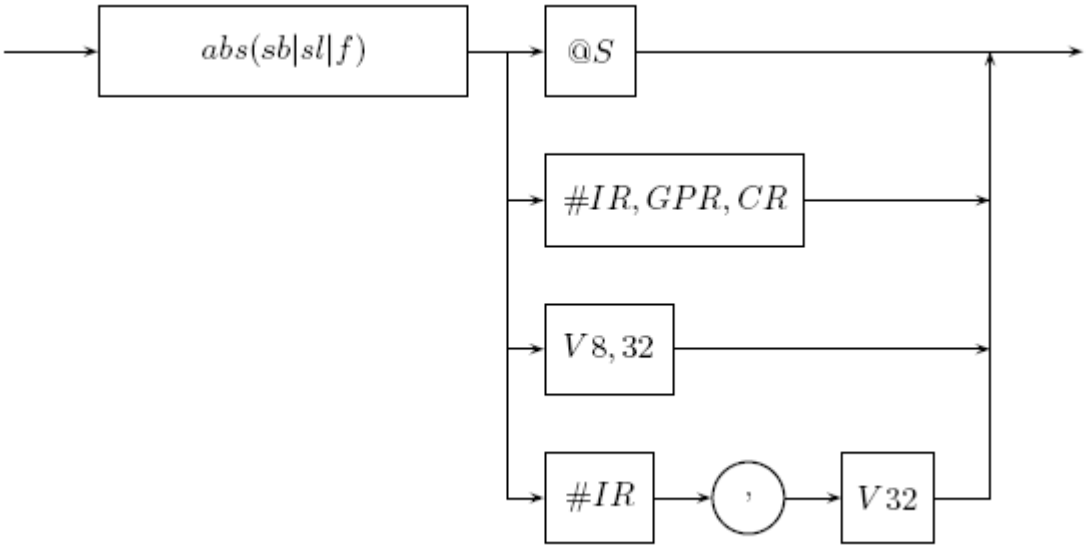


Рис.16 – Синтаксическое описание команды abs

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	r	r

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

Пример:

```
1 . t e x t
2
3 A:
4 ge t sb −128
5 abssb @1
6 a b s s l 0xF0010203
7 abs f 0 f−12 . 8 5
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) □, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;

– в строке №4 командой getsb помещается в коммутатор 8-ми разрядное целое знаковое число (константа −128);

– в строке №5 командой abssb вычисляется абсолютное значение результата выполнения предшествующей команды: @1 — результат выполнения команды извлечения в строке №4; аргумент команды abssb согласно суффиксу sb интерпретируются как целое знаковое размерностью байт; результат выполнения команды интерпретируются также как целое знаковое размерностью байт и помещается в комутатор с установленным флагом переполнения (□□), так как значение 128 выходит за границы разрядной сетки знакового целого числа размерностью байт;

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

– в строке №8 командой complete завершается текущий параграф.

4.4.4.2. adc (ADdition with Carry)

Сложение с переносом

$adc\ ARG1, ARG2$

Назначение: операция целочисленного сложения с учётом флага переноса (CF) результата предыдущего сложения командой add

Синтаксис:

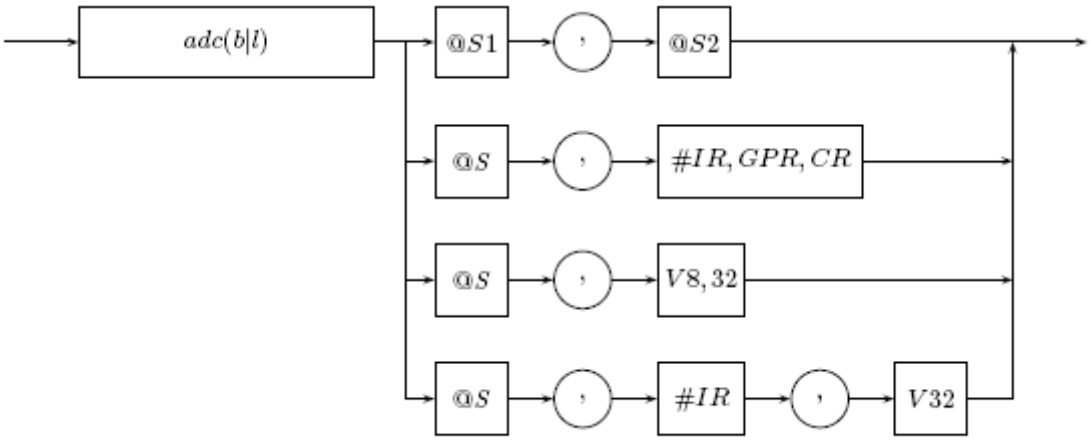


Рис.17 – Синтаксическое описание команды adc

Наличие результата: да

Алгоритм работы:

– выполнить сложение значения флага переноса (CF) аргумента $ARG1$ со значением аргумента $ARG2$;

– установить флаги;

– поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

Применение: команда adc используется при сложении длинных целых чисел. В отличии от одноимённой команды большинства других процессоров, команда adc мультиклеточного

SF	CF	OF	ZF
r	r	r	r

процессора складывает значение второго аргумента с единицей, если флаг переноса (CF) первого аргумента установлен, в противном случае в качестве результата возвращается просто значение второго аргумента.

Пример:

```
1  /*
2   * Программа сложения двух 64-х разрядных чисел
3  */
4
5  .data
6
7  A:
8       .long 0x00010203, 0xAABBCCDD
9  B:
10      .long 0x04050607, 0xEEFF0908
11 C:
12      .long 0x00000000, 0x00000000
13
14 .text
15
16 D:
17      rdl A
18      rdl A + 4
19      rdl B
20      rdl B + 4
21      addl @3, @1; результат 0x99BAD5E5, OF == 1
22      addl @5, @3; результат 0x0406080A
23      adcl @2, @1; результат 0x0406080B
24      wrl @1, C
25      wrl @4, C + 4
26 complete
```

Пояснения к примеру

– в строке №5 директивой ассемблера .data устанавливается текущая секция ассемблирования - секция инициализированных данных data;

– в строках №7,№9,№11 объявляются символы (идентификаторы) A, B, C, которые являются метками в текущей секции ассемблирования (data), и инициализируется текущими значениями адреса ассемблирования;

– в строках №№8,10,12 директивой ассемблера .long в текущую секцию ассемблирования по текущему адресу ассемблирования записываются 32-х рарядные числа: $0x00010203$, $0xAABBCCDD$, $0x04050607$, $0xEEFF0908$, $0x00000000$, $0x00000000$;

– в строке №14 директивой ассемблера .text устанавливается текущая секция ассемблирования - секция исполняемых инструкций text;

– в строке №16 объявляется символ (идентификатор) D, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;

– в строках №№17,18,19,20 командами rdl читаются из памяти данных четыре 32-х разрядных целых беззнаковых числа по адресам A, $A + 4$, B, $B + 4$ соответственно и помещаются в коммутатор;

– в строке №21 командой addl складываются результаты выполнения двух предшествующих команд: @3 — результат выполнения команды чтения в строке №18, @1 — результат выполнения команды чтения в строке №20; оба аргумента команды addl согласно суффиксу l интерпретируются как 32-х разрядные целые беззнаковые числа; результат выполнения команды также интерпретируются как 32-х разрядное целое беззнаковое число и помещается в коммутатор;

– в строке №22 командой addl складываются результаты выполнения двух предшествующих команд: @5 — результат выполнения команды чтения в строке №17, @3 — результат выполнения команды чтения в строке №19; оба аргумента команды addl согласно суффиксу l интерпретируются как 32-х разрядные целые беззнаковые числа; результат выполнения команды также интерпретируются как 32-х разрядное целое беззнаковое число и помещается в коммутатор;

– в строке №23 командой adcl складываются результаты выполнения двух предшествующих команд: @2 — результат выполнения команды сложения в строке №20 (фактически используется только значение флага переполнения OF), @1 — результат выполнения команды сложения в строке №21; оба аргумента команды addl согласно суффиксу l интерпретируются как

32-х разрядные целые беззнаковые числа; результат выполнения команды также интерпретируются как 32-х разрядное целое беззнаковое число и помещается в коммутатор;

– в строке №24 командой wrl осуществляется запись в память данных по адресу C результата выполнения предшествующей команды: @1 — результат выполнения команды сложения с учётом флага переноса в строке №24;

– в строке №25 командой wrl осуществляется запись в память данных по адресу $C + 4$ результата выполнения предшествующей команды: @4 — результат выполнения команды сложения в строке №21;

– в строке №26 командой complete завершается текущий параграф.

4.4.4.3. add (ADDition)

Сложение

$add\ ARG1, ARG2$

Назначение: операция сложения двух аргументов

Синтаксис:

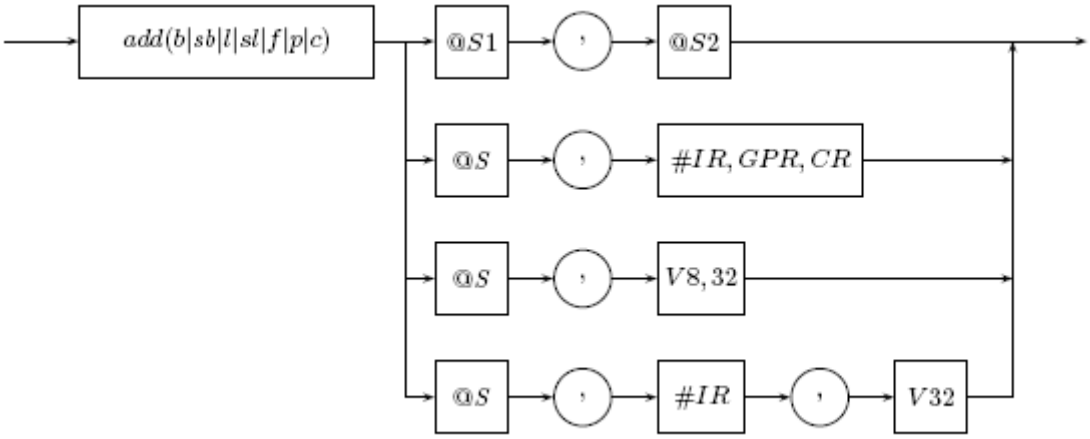


Рис.18 – Синтаксическое описание команды add

Наличие результата: да

Алгоритм работы:

– выполнить сложение $ARG1 + ARG2$;

– установить флаги;

– поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	r	r	r

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

Пример:

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

Пояснения к примеру

– в строке №1 директивой ассемблера .data устанавливается текущая секция ассемблирования — секция инициализированных данных data;

– в строке №3 объявляется символ (идентификатор) *B*, который является меткой в текущей секции ассемблирования (data), и инициализируется текущим значением адреса ассемблирования;

– в строке №4 директивой ассемблера .float в текущую секцию ассемблирования записывается четыре 32-х рарядных вещественных числа, начиная с текущего адреса ассемблирования (символ обратной косой черты \ в конце строки используется для продолжения строки, т. е. строки №№4,5,6 логически являются одной строкой);

– в строке №8 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;

– в строке №10 объявляется символ (идентификатор) *A*, который является меткой в

текущей секции ассемблирования (text), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;

– в строках №№11,12 командами rdq, читаются из памяти данных два 64-х разрядных целых беззнаковых числа по адресам B и B + 8 соответственно и помещаются в коммутатор;

– в строке №13 командой addc выполняется операция сложения результатов выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №11, @2 — результат выполнения команды чтения в строке №12; оба аргумента команды addc согласно суффиксу с интерпретируются как комплексные числа размерностью 64 бита, старшие 32 разряда которых представляют действительные части чисел, а младшие 32 разряда — мнимые; результат выполнения команды также интерпретируются как 64-х разрядное комплексное число и помещается в комутатор;

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

– в строке №15 командой complete завершается текущий параграф.

4.4.4.4. and (AND)

Логического умножение

and ARG1 , ARG2

Назначение: операция логического умножения двух аргументов

Синтаксис:

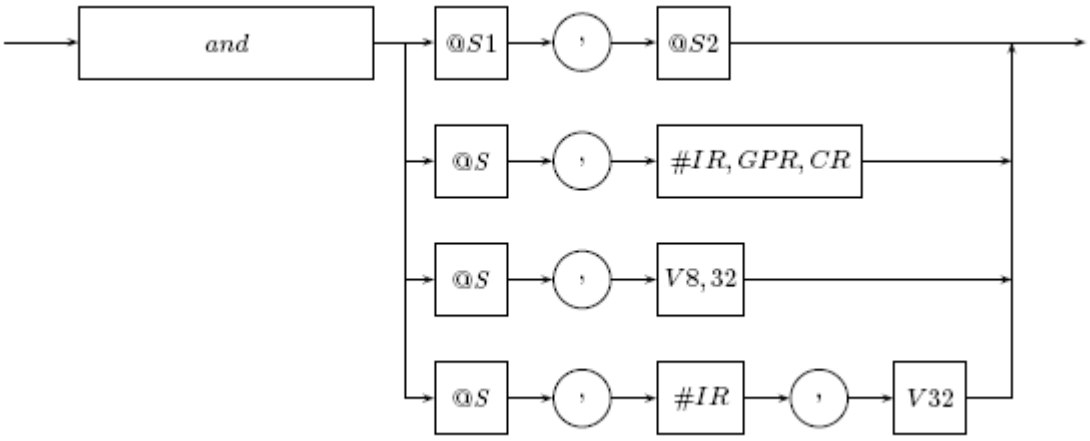


Рис.19 – Синтаксическое описание команды and

Наличие результата: да

Алгоритм работы:

– выполнить логическое умножение $ARG1$ & $ARG2$;

– установить флаги;

– поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

4.4.4.5. cfsl (Convert Float to Signed Long)

Преобразование типа

$cfsl\ ARG$

Назначение: операция преобразования типа

Синтаксис:

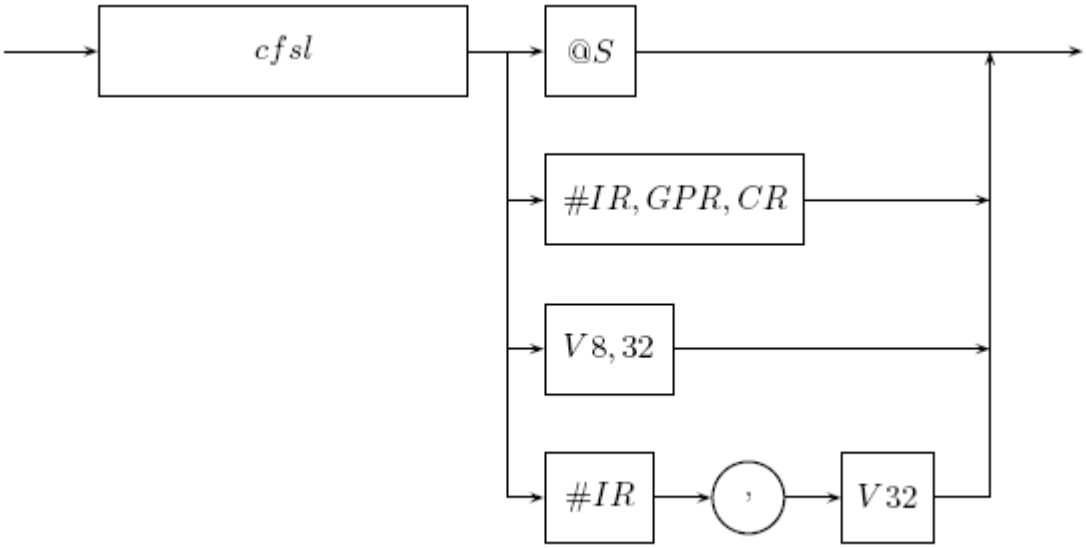


Рис.20 – Синтаксическое описание команды cf sl

Наличие результата: да

Алгоритм работы:

– выполнить округление вещественного числа одинарной точности, заданного аргументом ARG2, до ближайшего целого;

– выполнить преобразование округлённого вещественного числа одинарной точности в знаковое 32-х разрядное целое число;

– установить флаги;

– поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
r	0	r	r

Применение: команда cfsl используется для преобразования с предварительным округлением до ближайшего целого значения операнда, интерпретируемого как вещественное число одинарной точности, к знаковому целому 32-х разрядному числу.

Пример:

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

Пояснения к примеру

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

– в строке №3 объявляется символ (идентификатор) *A*, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;

– в строке №4 командой cfsl выполняется операция преобразования с предварительным округлением до ближайшего целого вещественного числа одинарной точности 13.94 к целому знаковому 32-х разрядному числу (результат выполнения операция — 14);

– в строке №5 командой wrl в память данных по адресу 0 записывается значение результата выполнения предшествующей команды: @1 — результат выполнения команды преобразования типа в строке №4;

– в строке №6 командой complete завершается текущий параграф.

4.4.4.6. clf (Convert Long to Float)

Преобразование типа

clf ARG

Назначение: операция преобразования типа

Синтаксис:

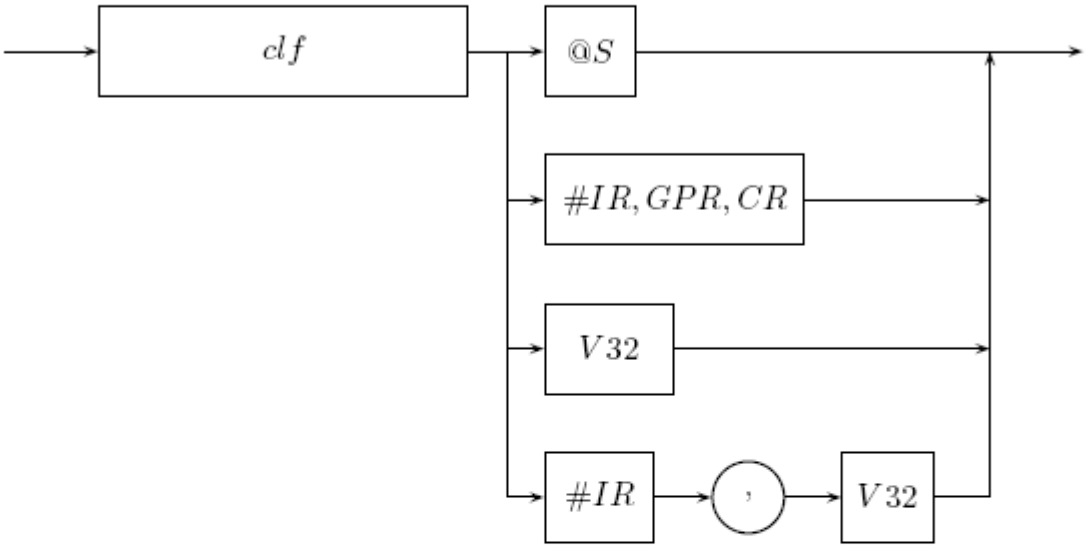


Рис. 21: Синтаксическое описание команды *clf*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование беззнакового 32-х разрядного целого числа, заданного аргументом *ARG*, в число с плавающей точкой;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

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

Пример:

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

Пояснения к примеру

- в строке №1 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;
- в строке №3 объявляется символ (идентификатор) A, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №4 командой cslf выполняется операция преобразования целого беззнакового 32-х разрядного числа 1394 к вещественнуму числу одинарной точности;
- в строке №5 командой wrl в память данных по адресу 0 записывается значение результата выполнения предшествующей команды: @1 — результат выполнения команды преобразования типа в строке №4;
- в строке №6 командой complete завершается текущий параграф.

4.4.4.7. cslf (Convert Signed Long to Float)

Преобразование типа

cslf ARG

Назначение: операция преобразования типа

Синтаксис:

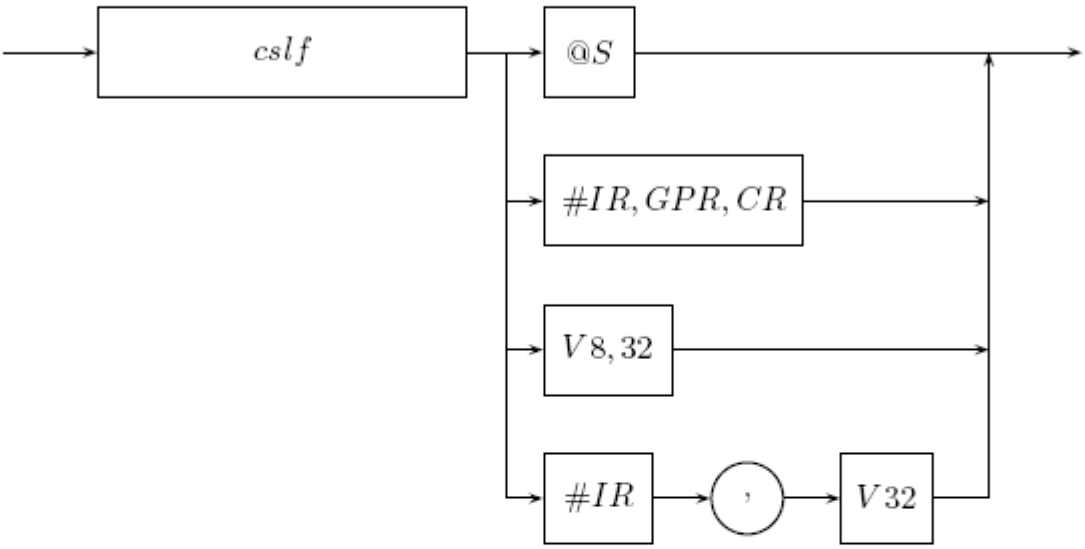


Рис. 22: Синтаксическое описание команды *cslf*

Наличие результата: да

Алгоритм работы:

- выполнить преобразование знакового 32-х разрядного целого числа, заданного аргументом *ARG*, в число с плавающей точкой;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

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

Пример:

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

Пояснения к примеру

- в строке №1 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;
- в строке №3 объявляется символ (идентификатор) A, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №4 командой cslf выполняется операция преобразования целого знакового 32-х разрядного числа -1394 к вещественнуму числу одинарной точности;
- в строке №5 командой wrl в память данных по адресу 0 записывается значение результата выполнения предшествующей команды: @1 — результат выполнения команды преобразования типа в строке №4;
- в строке №6 командой complete завершается текущий параграф.

4.4.4.8. div (DIVide)

Сложение

div ARG1, ARG2

Назначение: операция деления двух аргументов

Синтаксис:

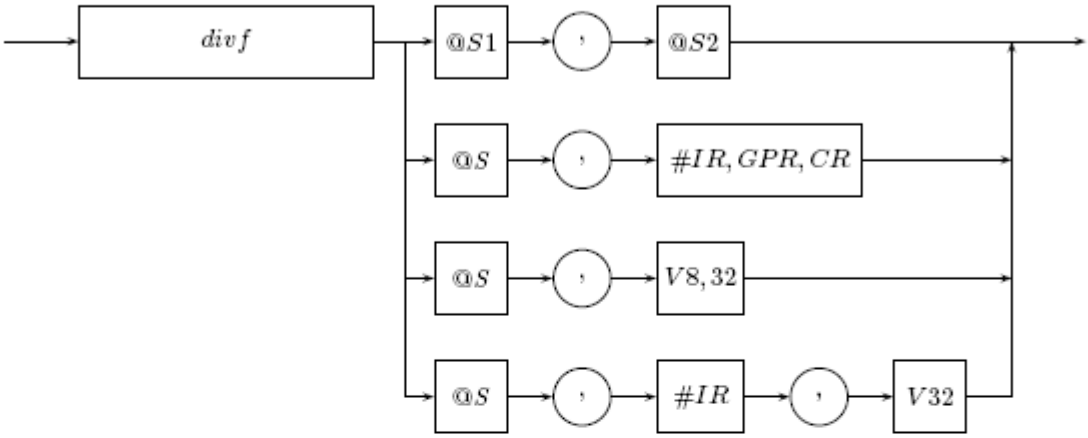


Рис. 23: Синтаксическое описание команды *div*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	r	r

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

Пример:

```
1  .data
2
3  B:
4        .float  0f12.8e12 , 0f-5.6e-4
5
6  .text
7
8  A:
9        rdl B
10       rdl B + 4
11       divf @1, @2
12       wrl @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 рарядных вещественных числа, начиная с текущего адреса ассем-

блирования;

- 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 адресса ассемблирования, а также начинает новый параграф;
- в строках №№9,10 командами rdl, читаются из памяти данных два 32-х разрядных целых беззнаковых числа по адресам B и $B + 4$ соответственно и помещаются в комутатор;
- в строке №11 командой divf выполняется операция деления результатов выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №19, @2 — результат выполнения команды чтения в строке №10; оба аргумента команды divf согласно суффиксу f интерпретируются как вещественные числа одинарной точности; результат выполнения команды также интерпретируются как вещественное число одинарной точности и помещается в комутатор;
- в строке №12 командой wrl осуществляется запись в память данных по адресу $B + 8$ результата выполнения предшествующей команды: @1 — результат выполнения команды деления в строке №11;
- в строке №13 командой complete завершается текущий параграф.

4.4.4.9. exa (EXacutive Address)

Исполнительный адрес

exa ARG

Назначение: операция формирования исполнительного адреса памяти данных

Синтаксис:

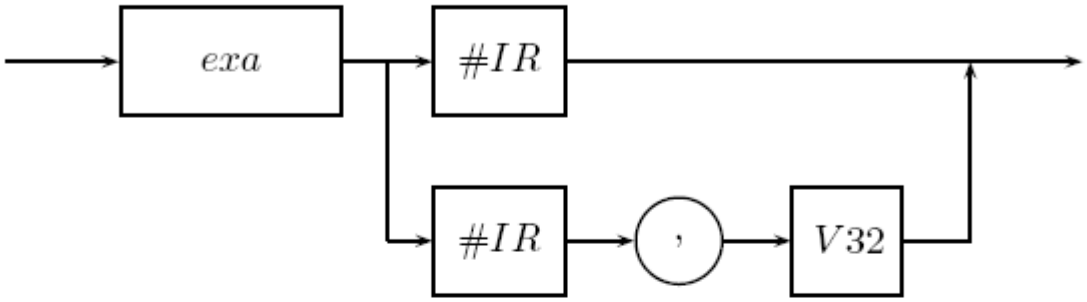


Рис. 24: Синтаксическое описание команды *exa*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
0	0	0	r

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

от большенства других команд процессора, не осуществляет чтение значения из памяти данных по сформированному адресу. Правила формирования исполнительного адреса описаны в разделе «Регистры индексные» (16.3.2). В общем случае, интерпретация значения, сохранённого данной командой в коммутаторе, зависит от алгоритма программы.

Пример:

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

```
25      setq #32, @1
26      getl 0x0000001C
27      patch @1, A
28      setq #33, @1
29      setl #MODR, 3
30  complete
31
32  MUL:
33      exa #32
34      je @1, FINISH
35      jne @2, MUL
36
37      rdl #33
38      rdl #33, B - A
39      mull @1, @2
40      wrl @1, #33, C - A
41  complete
42
43  FINISH:
44      jmp FINISH
45  complete
```

Пояснения к примеру

- в строке №10 директивой ассемблера .data устанавливается текущая секция ассемблирования — секция инициализированных данных data;
- в строке №12 объявляется символ (идентификатор) *A*, который является меткой в текущей секции ассемблирования (data), и инициализируется текущим значением адреса ассемблирования;
- в строке №13 директивой ассемблера .long в текущую секцию ассемблирования запи-

сывается пять 32-х рарядных целых числа, начиная с текущего адреса ассемблирования;

- в строке №14 объявляется символ (идентификатор) B, который является меткой в текущей секции ассемблирования (data), и инициализируется текущим значением адреса ассемблирования;
- в строке №15 директивой ассемблера .long в текущую секцию ассемблирования записывается пять 32-х рарядных целых числа, начиная с текущего адреса ассемблирования;
- в строке №12 объявляется символ (идентификатор) C, который является меткой в текущей секции ассемблирования (data), и инициализируется текущим значением адреса ассемблирования;
- в строке №18 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;
- в строке №20 объявляется символ (идентификатор) $INIT$, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №21 командой jmp безусловно устанавливается адрес следующего параграфа равным MUL;
- в строке №23 командой getl помещается в коммутатор 32-х разрядное целое беззнаковое число (константа $0x00040007$);
- в строке №24 командой patch формируется 64-х разрядное число, старшие 32 разряда которого инициализируются результататом выполнения предшествующей команды: @1 — результат выполнения команды извлечения в строке №23, а младшие 32 разряда — значением 0;
- в строке №25 командой setq в индексный регистр №32 помещается результат выполнения предшествующей команды: @1 — результат выполнения команды склейки в

строке №24;

- в строке №26 командой getl помещается в коммутатор 32-х разрядное целое беззнаковое число (константа $0x0000001C$);
- в строке №27 командой patch формируется 64-х разрядное число, старшие 32 разряда которого инициализируются результататом выполнения предшествующей команды: @1 — результат выполнения команды извлечения в строке №23, а младшие 32 разряда — значением метки A;
- в строке №28 командой setq в индексный регистр №33 помещается результат выполнения предшествующей команды: @1 — результат выполнения команды склейки в строке №24;
- в строке №29 командой setl устанавливается значение регистра маски модификации индексных регистров, равное 3, что соответствует необходимости пересчёта значений индексных регистров №№32,33 по завершении каждого параграфа;
- в строке №30 командой complete завершается текущий параграф; происходит фактическая установка значений индексных регистров №№32,33 и реистра маски модификации индексных регистров.
- в строке №32 объявляется символ (идентификатор) MUL, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №33 командой exa помещается в коммутатор 32-х разрядное целое беззнаковое число, сформированное на основании значения индексного регистра №32;
- в строке №34 командой je устанавливается адрес следующего параграфа равным FINISH, если установлен флаг нуля (ZF) результата выполнения предшествующей команды: @1 — результат выполнения команды формирования исполнительного адреса в строке №33;
- в строке №35 командой jne устанавливается адрес следующего параграфа равным

MUL, если сброшен флаг нуля (ZF) результата выполнения предшествующей команды: @1 — результат выполнения команды формирования исполнительного адреса в строке №33;

- в строке №37 командой rdl, читается из памяти данных 32-х разрядное целое беззнаковое число по адресу, сформированному на основании значения индексного регистра №33 и помещается в коммутатор;
- в строке №38 командой rdl, читается из памяти данных 32-х разрядное целое беззнаковое число по адресу, сформированному на основании значения индексного регистра №33 и смещения $B - A$ и помещается в коммутатор;
- в строке №39 командой mull выполняется операция умножения результатов выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №38, @2 — результат выполнения команды чтения в строке №37; результат выполнения команды также интерпретируются как 32-х разрядное целое знаковое число и помещается в комутатор;
- в строке №40 командой wrl осуществляется запись в память данных по адресу, сформированному на основании значения индексного регистра №33 и смещения $C - A$, результата выполнения предшествующей команды: @1 — результат выполнения команды умножения в строке №39;
- в строке №41 командой complete завершается текущий параграф; происходит пересчёт значений индексных регистров №№32,33.
- в строке №43 объявляется символ (идентификатор) $FINISH$, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №44 командой jmp безусловно устанавливается адрес следующего параграфа равным $FINISH$;
- в строке №45 командой complete завершается текущий параграф.

В приведённом примере индексный регистр №32 используется для организации циклического выполнения параграфа MUL; индексный регистр №33 используется для формирования исполнительного адреса памяти данных в командах чтения и записи. Исходя из начальных значений индексных регистров №№32,33, параграф MUL будет выполнен пять раз, а значение исполнительного адреса памяти данных в командах чтения и записи будет увеличиваться на четыре при каждом завершении выполнения параграфа MUL.

4.4.4.10. get (GET value)

Извлечение значения

get ARG

Назначение: операция извлечения значения

Синтаксис:

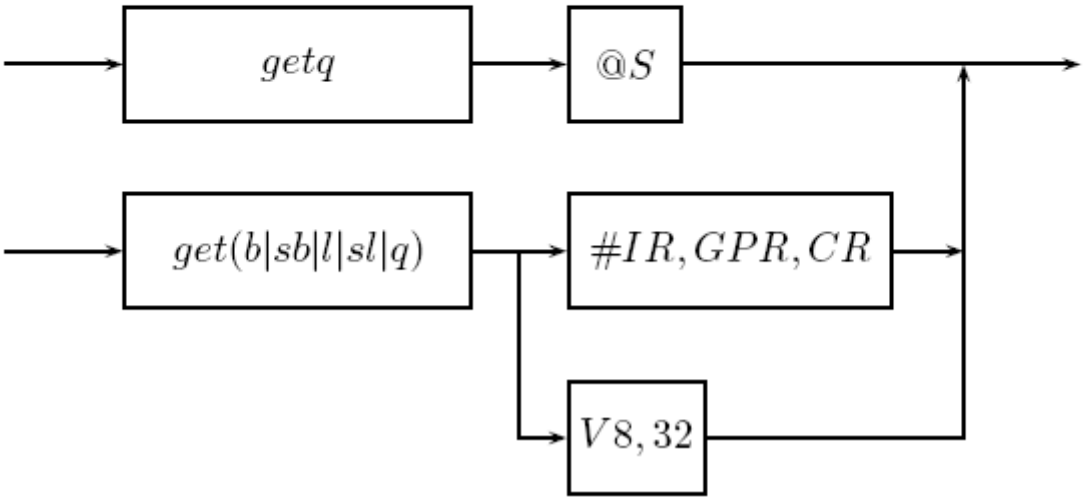


Рис. 25: Синтаксическое описание команды *get*

Наличие результата: да

Алгоритм работы:

- извлечь значение, заданное аргументом *ARG*;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
r	0	0	r

Применение: команда get используется

- для проброса значения, ранее сохранённого в комутаторе;
- для извлечения значения из регистра любого типа и сохранения его в комутаторе, т. е. в не зависимости от типа регистра извлекается именно значение этого регистра;
- для сохранения константы в комутаторе.

Пример:

```
1  .text
2
3  A:
4        getsb  -128
5        getq  #0
6        getl  #32
7        getl  #PSW
8        getq  @4
9  complete
```

Пояснения к примеру

- в строке №1 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;
- в строке №3 объявляется символ (идентификатор) A, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №4 командой getsb помещается в коммутатор 8-ми разрядное целое знаковое число (константа -128);
- в строке №5 командой getq извлекается значение регистра общего назначения №0 и сохраняется в коммутаторе;

– в строке №6 командой getl извлекается значение базы индексного регистра №32 и сохраняется в коммутаторе (не смотря на то, что команде getl указан в качестве аргумента индексный регистр, формирование исполнительного адреса памяти данных, а также обращение к памяти данных, не происходит, просто извлекаются младшие 32 разряда индексного регистра и сохраняются в коммутаторе);

– в строке №7 командой getl извлекается значение управляющего регистра PSW и сохраняется в коммутаторе;

– в строке №8 командой getq сохраняется в комутаторе ранее сохранённое в комутаторе значение, т. е. сохраняется результат выполнения предшествующей команды с обновлением значений флагов результата: @4 — результат выполнения команды извлечения в строке №4;

– в строке №9 командой complete завершается текущий параграф.

4.4.4.11. insub (INversion SUBtract)

Обратное вычитание

insub *ARG*1, *ARG*2

Назначение: операция обратного вычитания двух аргументов

Синтаксис:

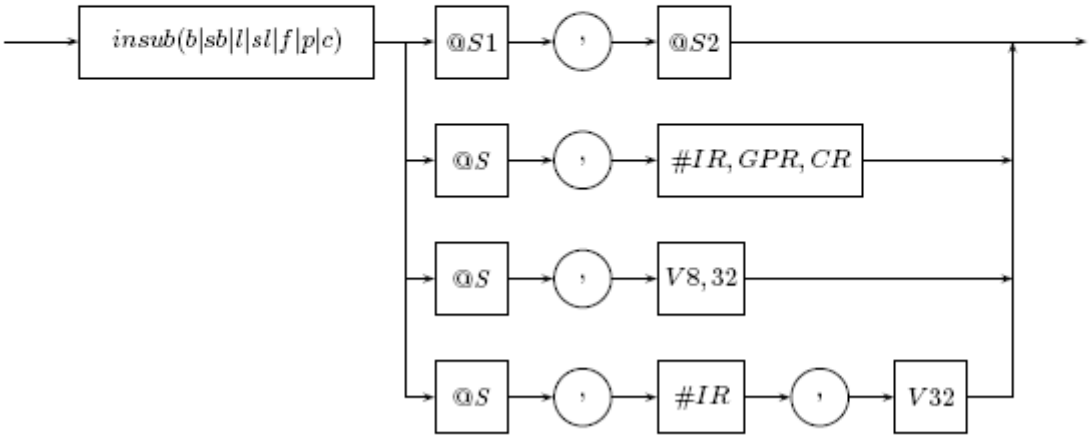


Рис. 26: Синтаксическое описание команды *insub*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
r	r	r	r

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

Пример:

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

Пояснения к примеру

- в строке №1 директивой ассемблера .data устанавливается текущая секция ассемблирования — секция инициализированных данных data;
- в строке №3 объявляется символ (идентификатор) B, который является меткой в текущей секции ассемблирования (data), и инициализируется текущим значением адреса ассемблирования;
- в строке №4 директивой ассемблера .float в текущую секцию ассемблирования записывается четыре 32-х рарядных вещественных числа, начиная с текущего адреса ассемблирования (символ обратной косой черты \ в конце строки используется для

продолжения строки, т. е. строки №№4,5,6 логически являются одной строкой);

- в строке №8 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;
- в строке №10 объявляется символ (идентификатор) A, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адреса ассемблирования, а также начинает новый параграф;
- в строках №№11,12 командами rdq, читаются из памяти данных два 64-х разрядных целых беззнаковых числа по адресам B и $B+8$ соответственно и помещаются в коммутатор;
- в строке №13 командой insubp выполняется операция обратного вычитания результатов выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №11, @2 — результат выполнения команды чтения в строке №12; оба аргумента команды insubp согласно суффиксу p интерпретируются как упакованные числа размерностью 64 бита, поэтому операция обратного вычитания выполняется независимо для старшей и младшей частей чисел; результат выполнения команды также интерпретируются как 64-х разрядное упакованное число и помещается в коммутатор;
- в строке №14 командой wrq осуществляется запись в память данных по адресу $B+16$ результата выполнения предшествующей команды: @1 — результат выполнения команды обратного вычитания в строке №13;
- в строке №15 командой complete завершается текущий параграф.

4.4.4.12. ja (Jump if Above)

Установка адреса следующего параграфа, условная

ja ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

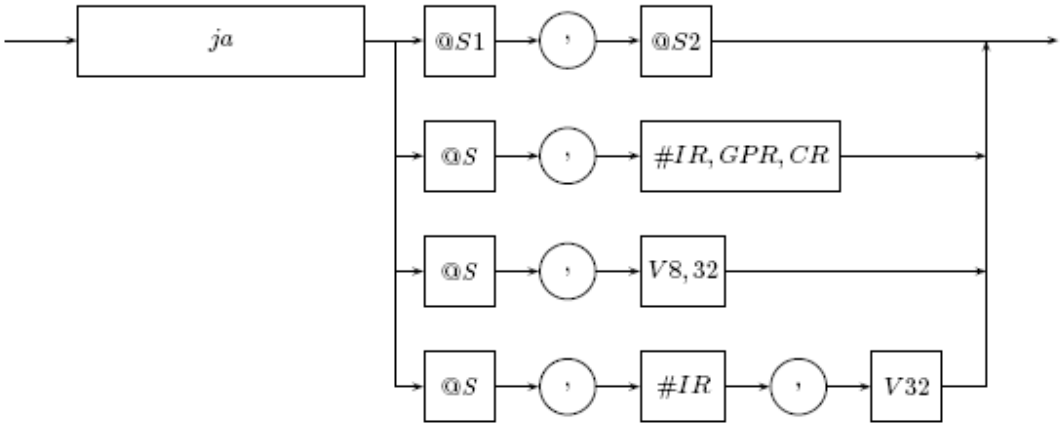


Рис. 27: Синтаксическое описание команды *ja*

Наличие результата: нет

Алгоритм работы:

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

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

Установка адреса следующего параграфа, условная

$(jae|jnc)\ ARG1,\ ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

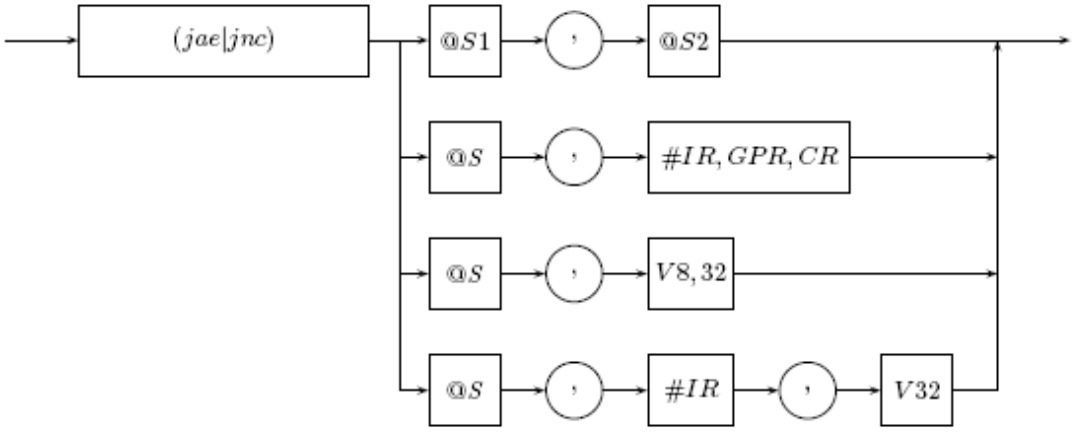


Рис. 28: Синтаксическое описание команды jae/jnc

Наличие результата: нет

Алгоритм работы:

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

4.4.4.14. jb (Jump if Below) / jc (Jump if Carry flag set)

Установка адреса следующего параграфа, условная

$(jb|jc)\ ARG1,\ ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

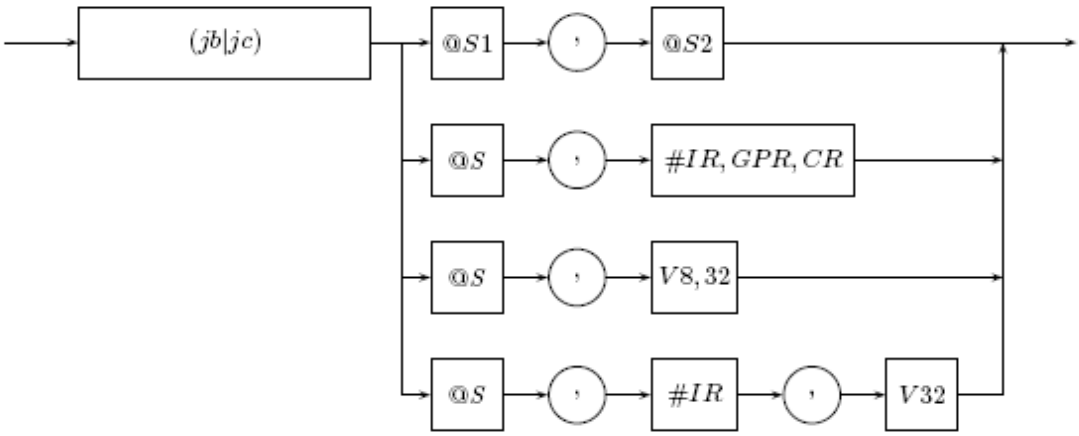


Рис. 29: Синтаксическое описание команды jb/jc

Наличие результата: нет

Алгоритм работы:

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

4.4.4.15. jbe (Jump if Below and Equal)

Установка адреса следующего параграфа, условная

jbe $ARG1$, $ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

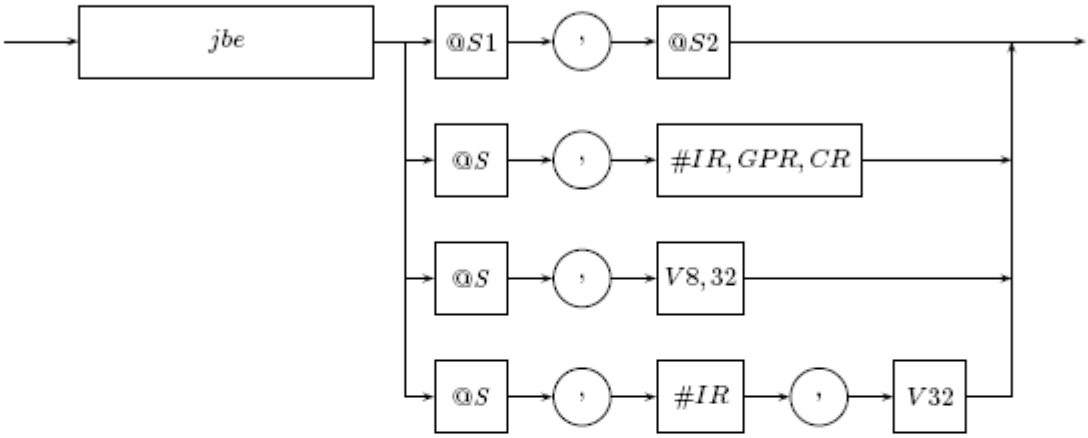


Рис. 30: Синтаксическое описание команды *jbe*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.16. je (Jump if Equal)

Установка адреса следующего параграфа, условная

je ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

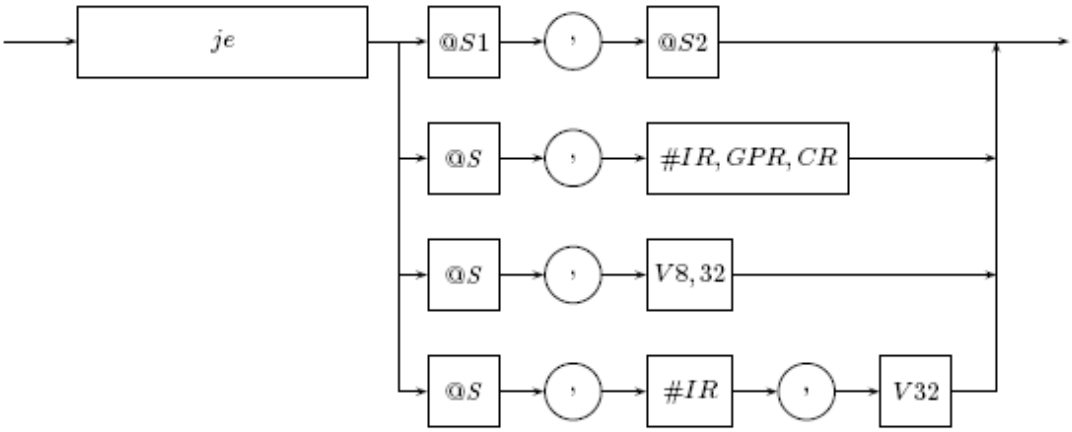


Рис. 31: Синтаксическое описание команды *je*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.17. jg (Jump if Greater)

Установка адреса следующего параграфа, условная

$jg\ ARG1,\ ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

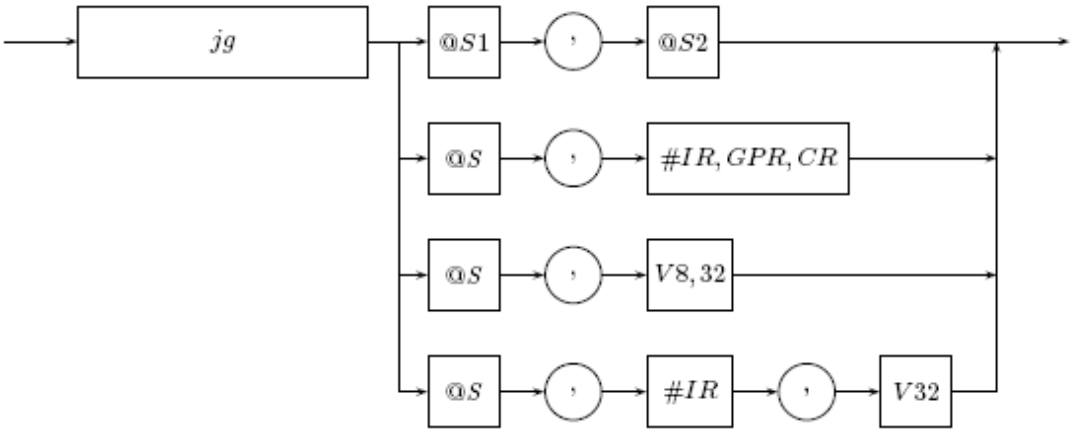


Рис. 32: Синтаксическое описание команды jg

Наличие результата: нет

Алгоритм работы:

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

4.4.4.18. jge (Jump if Greater and Equal)

Установка адреса следующего параграфа, условная

jge $ARG1$, $ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

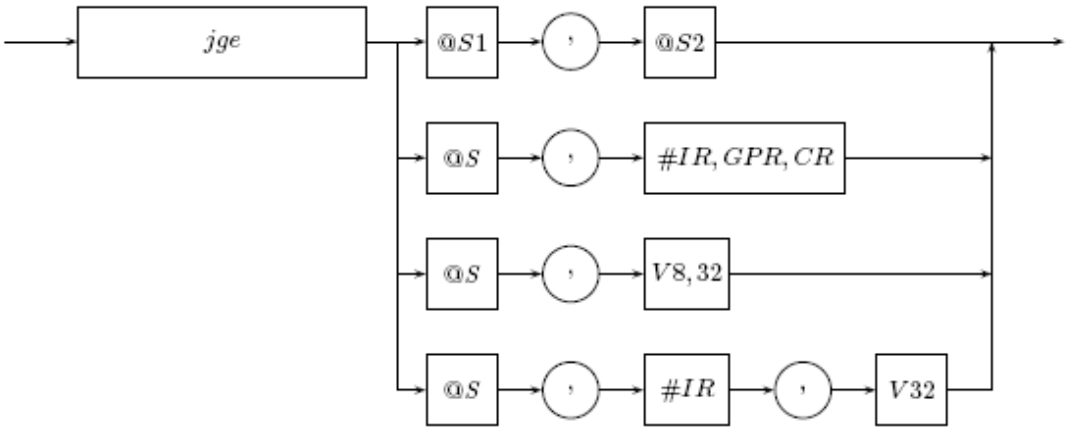


Рис. 33: Синтаксическое описание команды *jge*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.19. jl (Jump if Less)

Установка адреса следующего параграфа, условная

jl ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

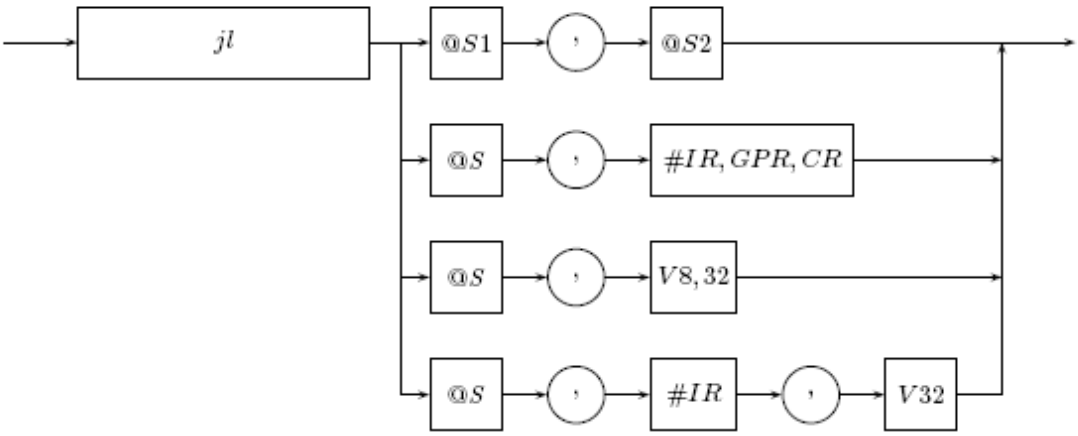


Рис. 34: Синтаксическое описание команды *jl*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.20. jle (Jump if Less and Equal)

Установка адреса следующего параграфа, условная

jle $ARG1,\ ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

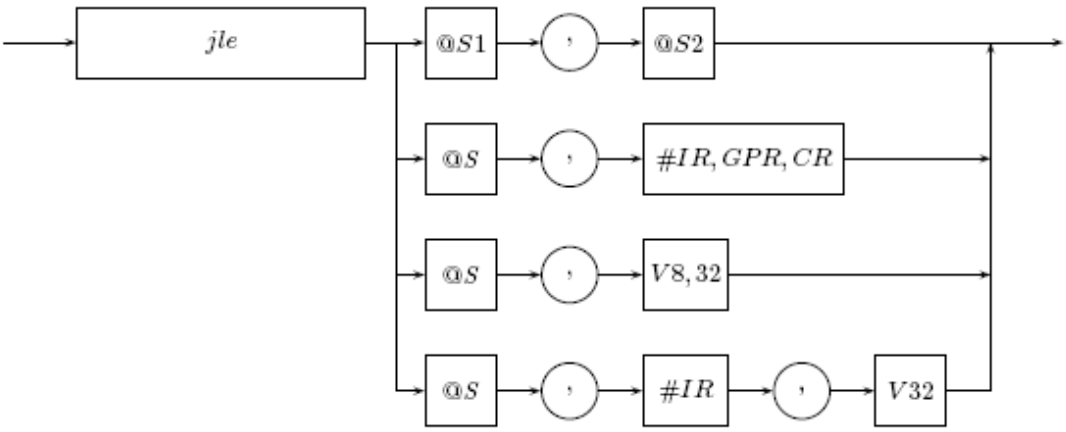


Рис. 35: Синтаксическое описание команды *jle*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.21. jmp (unconditional JuMP)

Установка адреса следующего параграфа, безусловная

jmp ARG

Назначение: операция безусловной установки адреса следующего параграфа

Синтаксис:

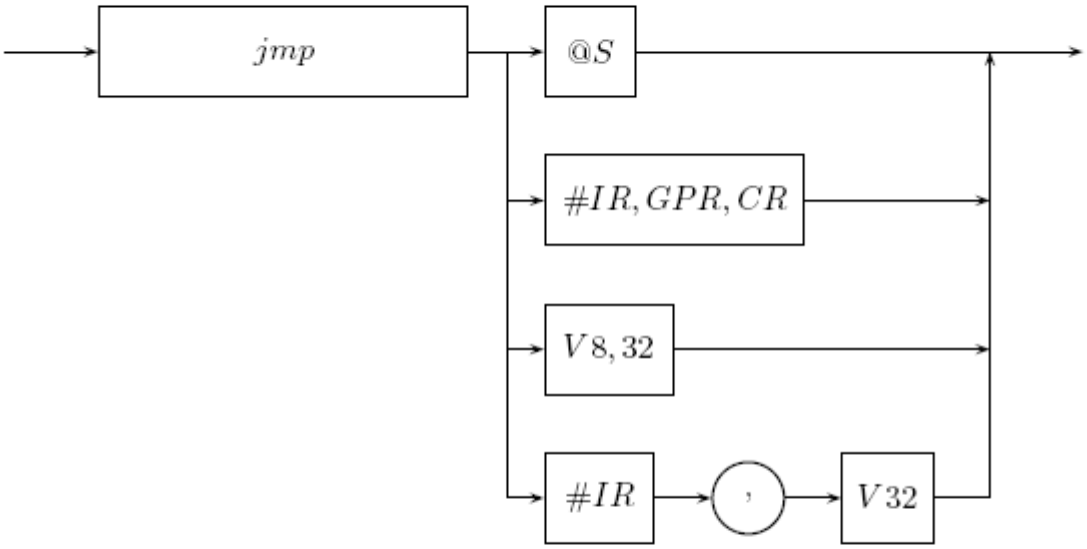


Рис. 36: Синтаксическое описание команды *jmp*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.22. jne (Jump if Not Equal)

Установка адреса следующего параграфа, условная

jne $ARG1$, $ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

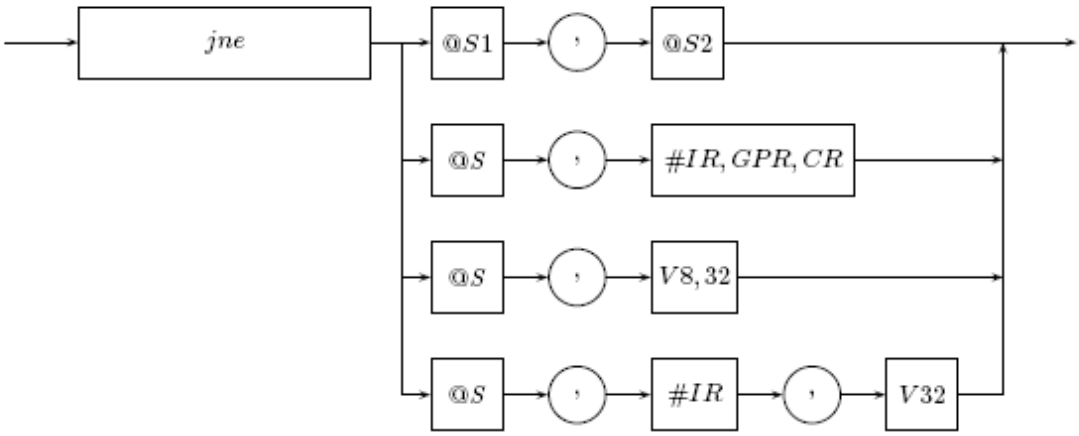


Рис. 37: Синтаксическое описание команды *jne*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.23. jno (Jump if Overflow flag unset)

Установка адреса следующего параграфа, условная

jno $ARG1,\ ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

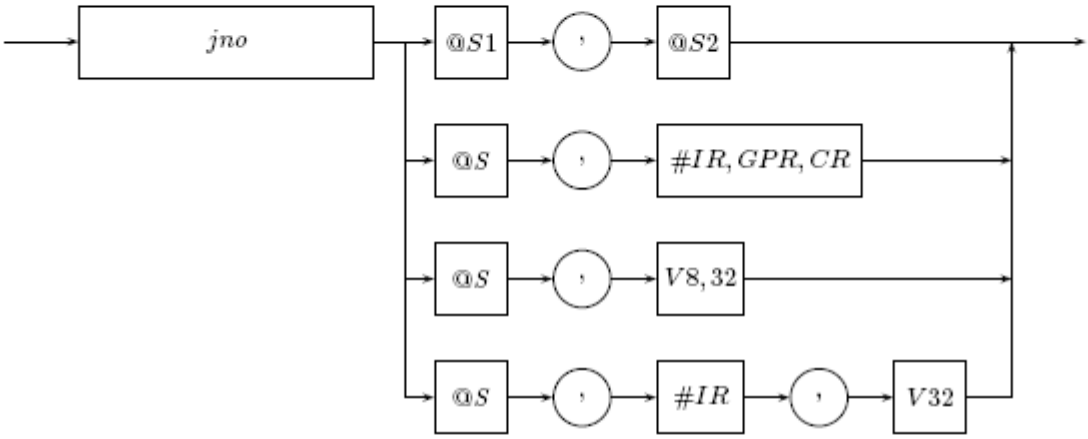


Рис. 38: Синтаксическое описание команды *jno*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.24. jns (Jump if Sign flag unset)

Установка адреса следующего параграфа, условная

jns $ARG1$, $ARG2$

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

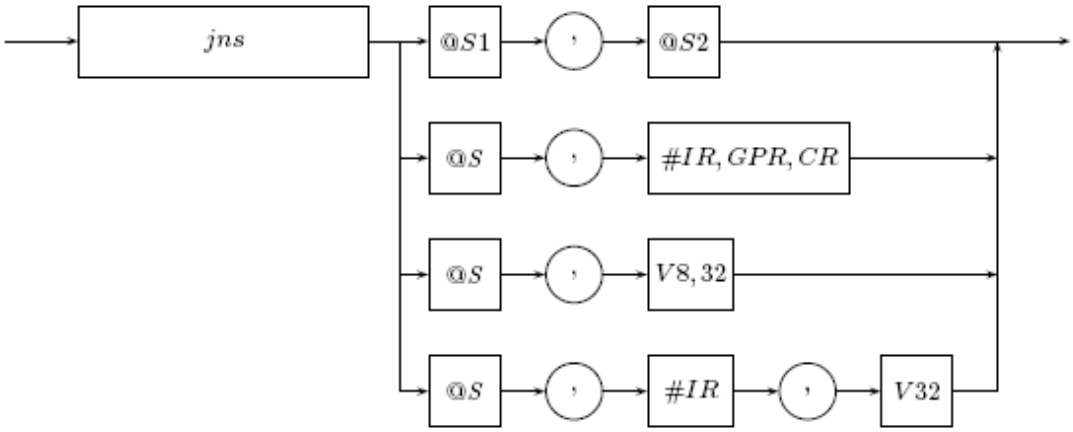


Рис. 39: Синтаксическое описание команды *jns*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.25. jo (Jump if Overflow flag set)

Установка адреса следующего параграфа, условная

jo ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

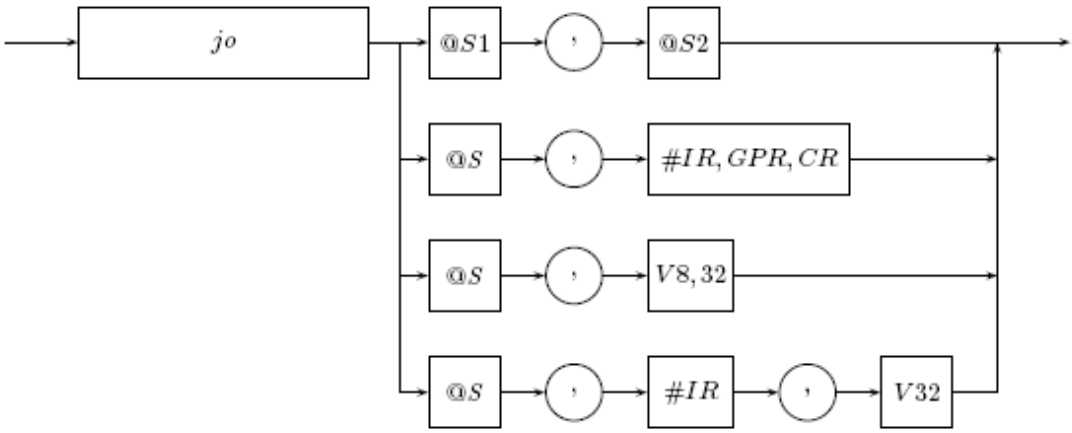


Рис. 40: Синтаксическое описание команды *jo*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.26. js (Jump if Sign flag set)

Установка адреса следующего параграфа, условная

js ARG1, ARG2

Назначение: операция условной установки адреса следующего параграфа

Синтаксис:

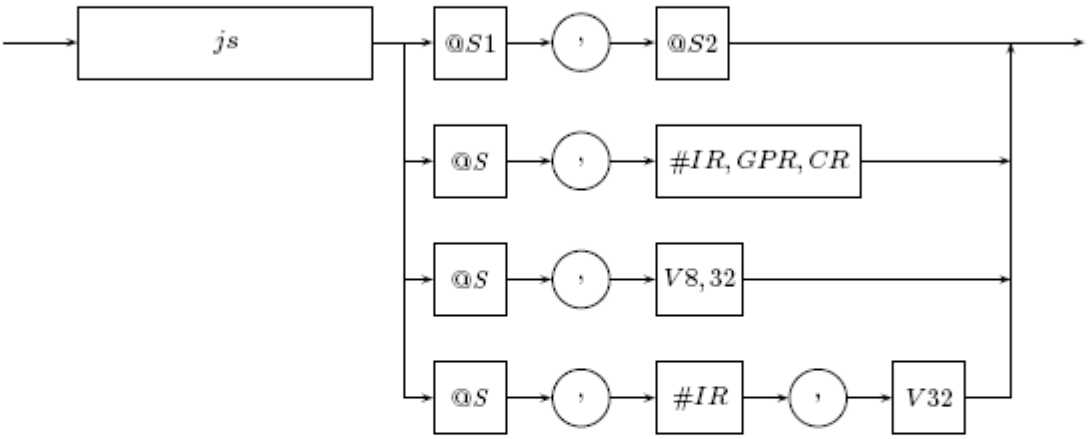


Рис. 41: Синтаксическое описание команды *js*

Наличие результата: нет

Алгоритм работы:

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

4.4.4.27. madd (Multiplication with ADDition of packed arguments)

Умножение со сложением

madd $ARG1$, $ARG2$

Назначение: операция упакованного умножения со сложением

Синтаксис:

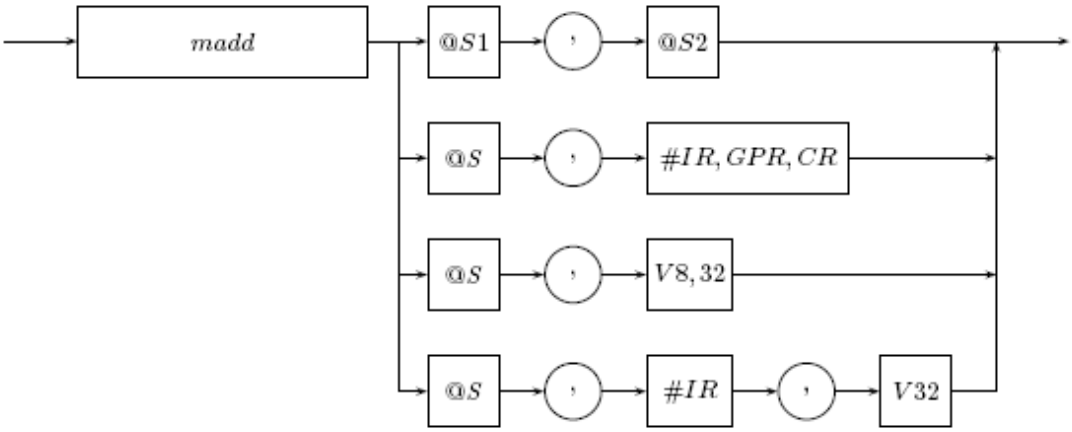


Рис. 42: Синтаксическое описание команды *madd*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	r	r

4.4.4.28. max (MAXimum)

Максимум

$max\ ARG1,\ ARG2$

Назначение: операция выбора наибольшего из двух аргументов

Синтаксис:

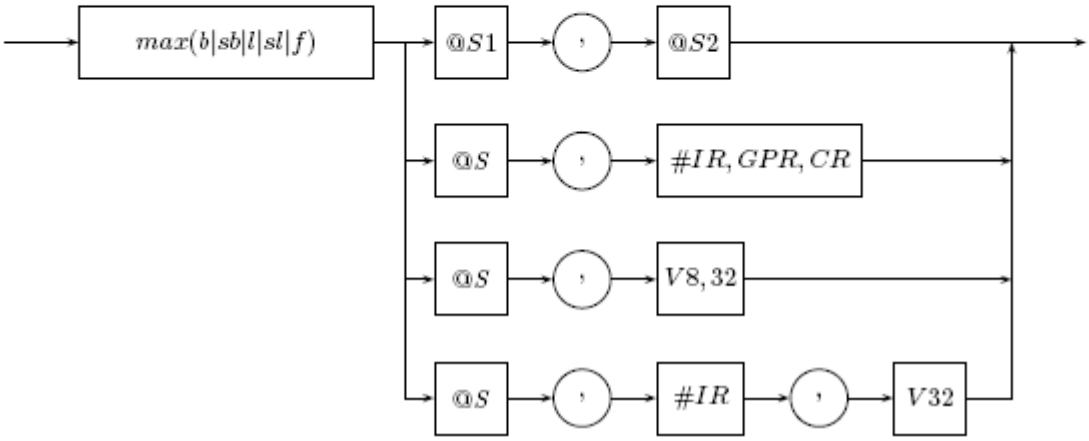


Рис. 43: Синтаксическое описание команды *max*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

4.4.4.29. min (MINimum)

Минимум

$min\ ARG1,\ ARG2$

Назначение: операция выбора наименьшего из двух аргументов

Синтаксис:

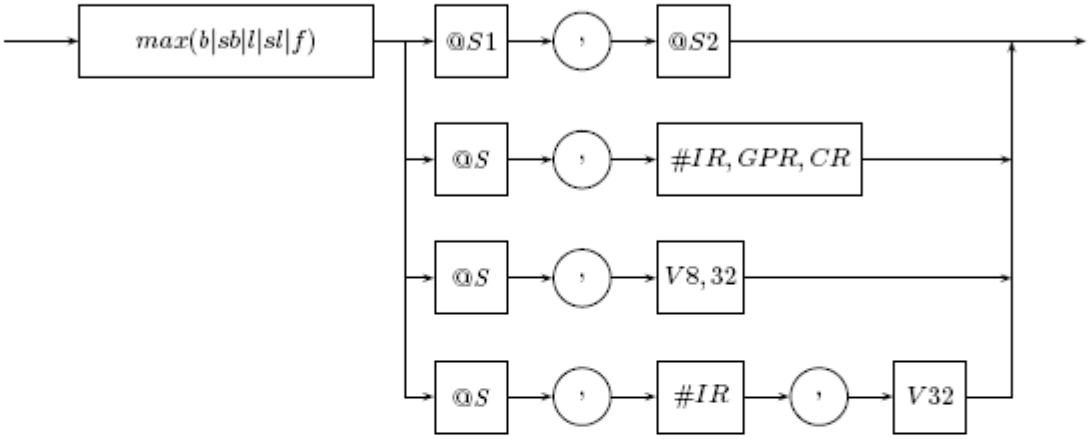


Рис. 44: Синтаксическое описание команды *min*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

4.4.4.30. mul (MULtiply)

Умножение

mul $ARG1$, $ARG2$

Назначение: операция умножения двух аргументов

Синтаксис:

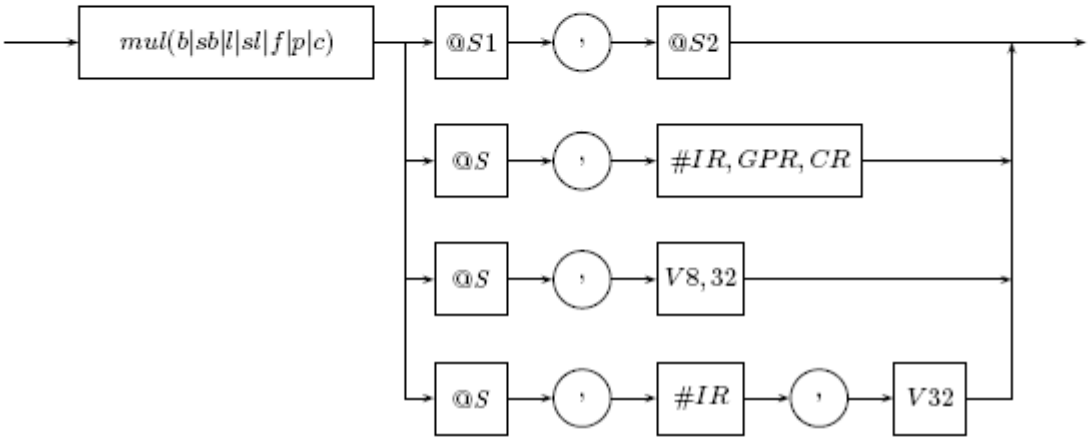


Рис. 45: Синтаксическое описание команды *mul*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	r	r	r

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

Пример:

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

Пояснения к примеру

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

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

4.4.4.31. norm (NORmalization)

Нормализация

norm ARG

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

Синтаксис:

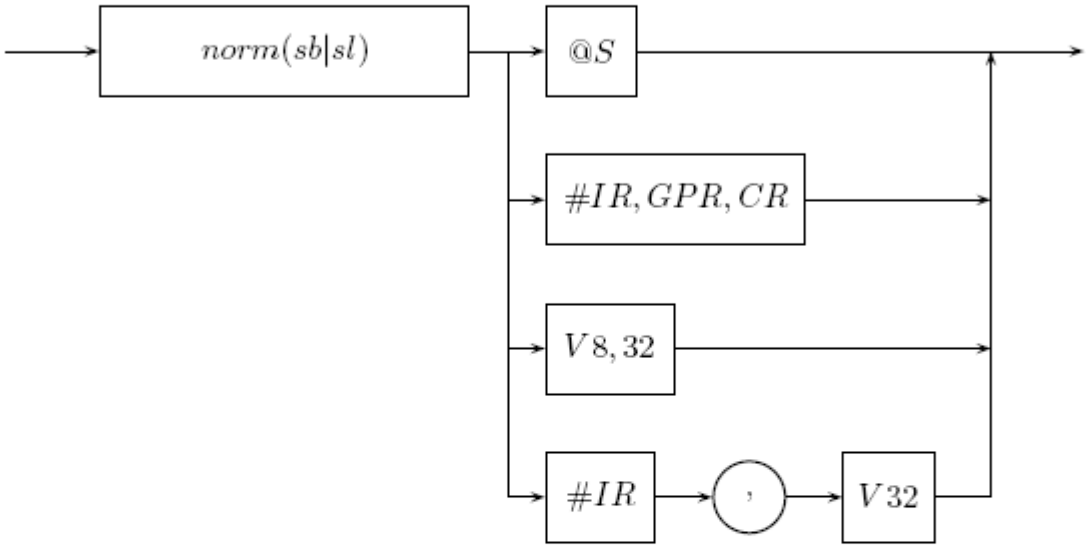


Рис. 46: Синтаксическое описание команды *norm*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
0	0	0	r

4.4.4.32. not (NOT)

Логического отрицание

not ARG

Назначение: операция логического отрицания аргумента

Синтаксис:

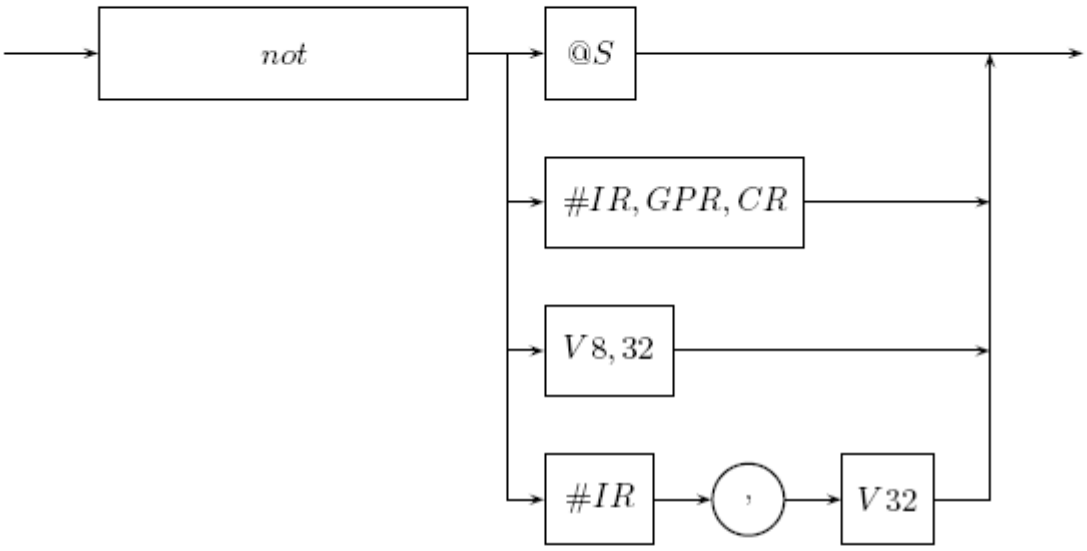


Рис. 47: Синтаксическое описание команды *not*

Наличие результата: да

Алгоритм работы:

- выполнить логическое отрицание $\sim ARG1$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
r	0	0	r

4.4.4.33. or (OR)

Логического сложение

or $ARG1$, $ARG2$

Назначение: операция логического сложения двух аргументов

Синтаксис:

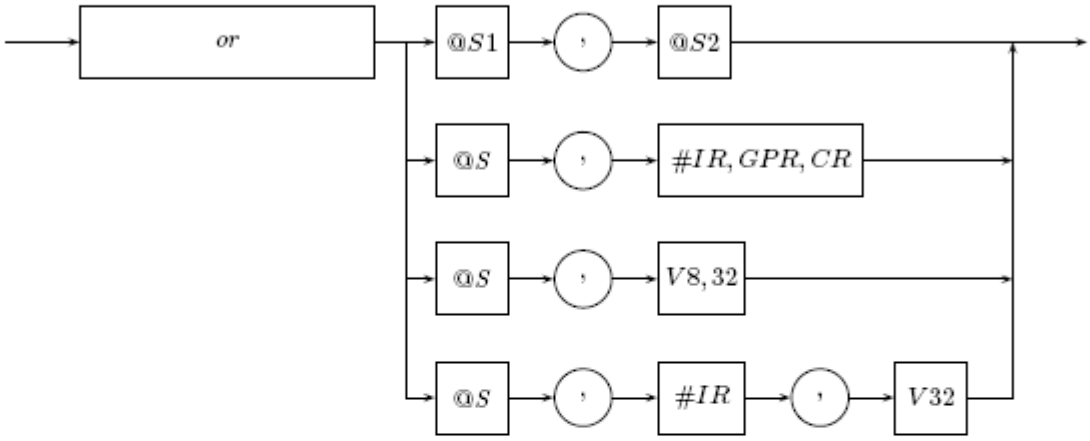


Рис. 48: Синтаксическое описание команды *or*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

4.4.4.34. pack (PACK)

Упаковка

pack $ARG1$, $ARG2$

Назначение: операция формирования упакованного числа

Синтаксис:

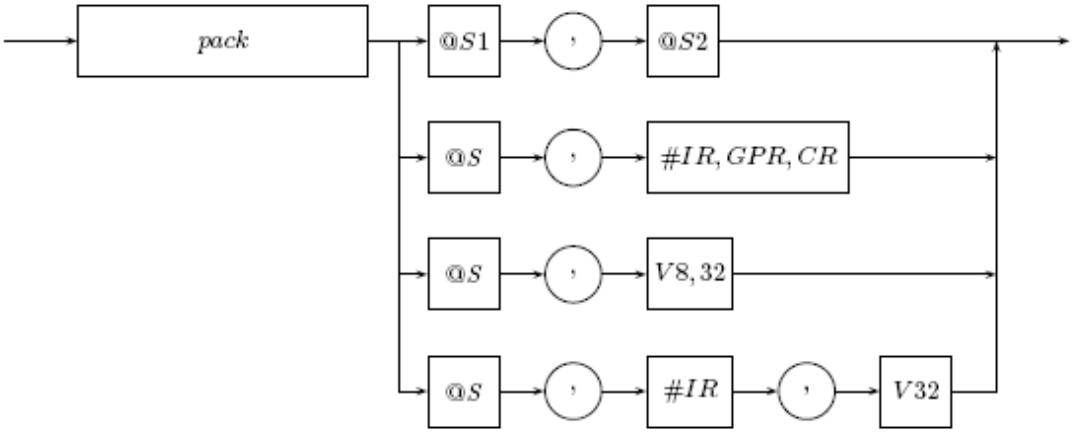


Рис. 49: Синтаксическое описание команды *pack*

Наличие результата: да

Алгоритм работы:

- сформировать 64-х разрядное значение результата, старшие 32 разряда (с 32 по 63) которого равны младшим 32 разрядам (с 0 по 31) аргумента $ARG1$, а младшие 32 разряда (с 0 по 31) — старшим 32 разрядам (с 32 по 63) аргумента $ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

SF	CF	OF	ZF
r	0	0	r

Состояние флагов результата после выполнения команды:

4.4.4.35. patch (PATCH)

Склейка

patch $ARG1$, $ARG2$

Назначение: операция формирования склеенного числа

Синтаксис:

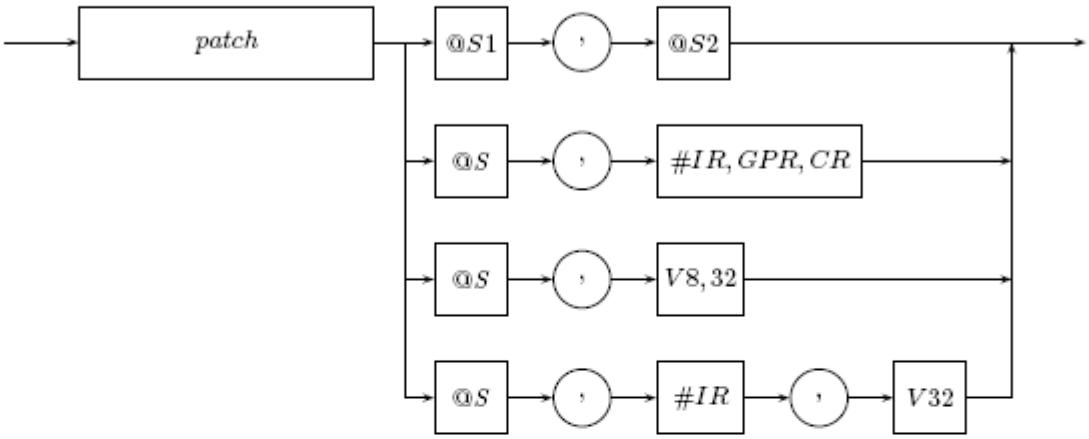


Рис. 50: Синтаксическое описание команды *patch*

Наличие результата: да

Алгоритм работы:

- сформировать 64-х разрядное значение результата, старшие 32 разряда (с 32 по 63) которого равны младшим 32 разрядам (с 0 по 31) аргумента $ARG1$, а младшие 32 разряда (с 0 по 31) — младшим 32 разрядам (с 0 по 31) аргумента $ARG2$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

4.4.4.36. rd (ReaD)

Чтение

rd ARG

Назначение: операция чтения значения из памяти данных

Синтаксис:

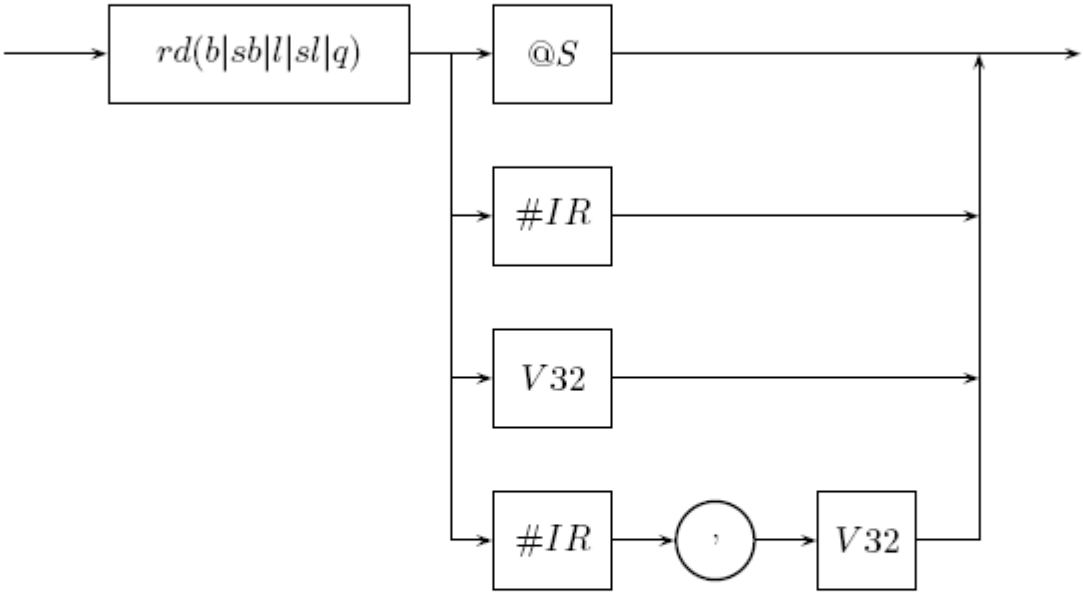


Рис. 51: Синтаксическое описание команды *rd*

Наличие результата: да

Алгоритм работы:

– прочитать в зависимости от типа команды значение размером байт (знаковый или беззнаковый), слово (знаковое или беззнаковое), двойное слово из памяти данных по адресу, заданному аргументом *ARG*;

– в случае чтения знакового байта или слова размножить знак;

– установить флаги;

– поместить результат вместе с флагами в коммутатор;

Интерпретация значения аргумента: согласно выше описанному алгоритму работы, значение аргумента всегда интерпретируется как адрес памяти данных, по которому осуществляется чтение значения, помещаемого в качестве результата данной команды в коммутатор. Другими словами, данная команда всегда обращается к памяти данных вне зависимости от варианта формирования значения 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.

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	0	0	r

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

Пример:

```
1  .data
2
3  A:
4      .long 1, 2
5
6  .text
7
8  B:
```

```
 9       rdl A
10       rdl A + 4
11       addl @1, @2
12       wrl @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 адресса ассемблирования;
- в строке №4 директивой ассемблера .long в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 32-х рарядное число 1, следом за которым по текущему адресу ассемблирования плюс 4 байта записывается 32-х рарядное 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 адресса ассемблирования, а также начинает новый параграф;
- в строках №№9,10 командами rdl, читаются из памяти данных два 32-х разрядных целых беззнаковых числа по адресам A и $A + 4$ соответственно и помещаются в коммутатор;
- в строке №11 командой addl складываются результаты выполнения двух предшествующих команд: @1 — результат выполнения команды чтения в строке №10, @2 — результат выполнения команды чтения в строке №9; оба аргумента команды addl со-

гласно 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 комутатор;

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

4.4.4.37. rol (ROtate Left)

Сдвиг циклический аргумента влево

rol ARG1, ARG2

Назначение: операция циклического сдвига аргумента влево

Синтаксис:

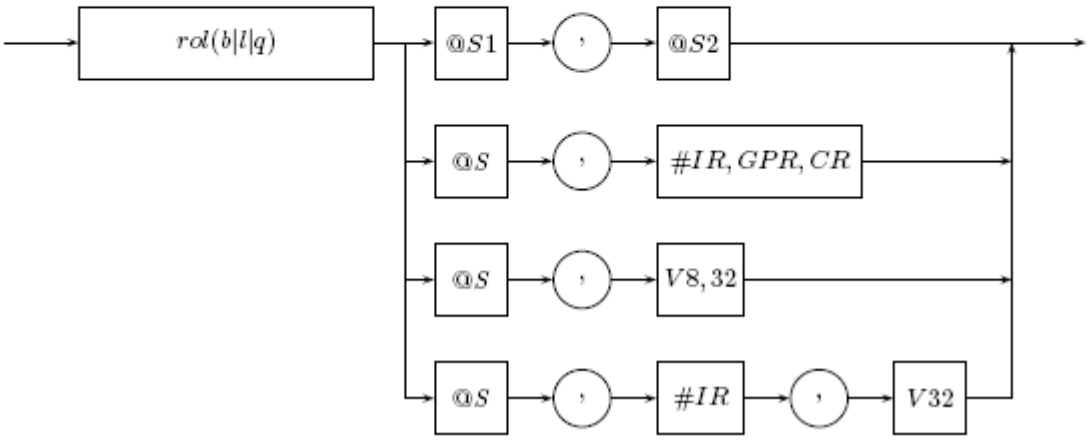


Рис. 52: Синтаксическое описание команды *rol*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента *ARG1* влево на *ARG2* разряда;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
r	r	r	r

4.4.4.38. ror (ROtate Right)

Сдвиг циклический аргумента вправо

ror $ARG1$, $ARG2$

Назначение: операция циклического сдвига аргумента вправо

Синтаксис:

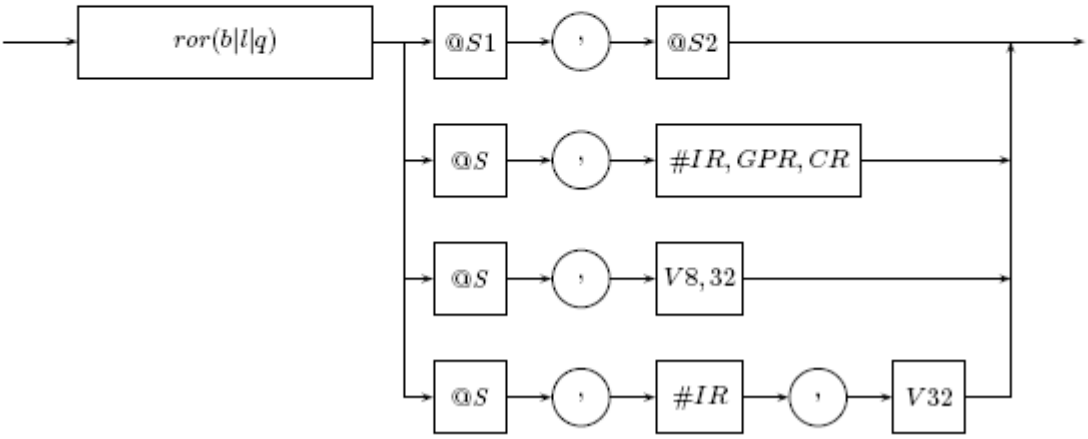


Рис. 53: Синтаксическое описание команды *ror*

Наличие результата: да

Алгоритм работы:

- выполнить циклический сдвиг аргумента $ARG1$ вправо на $ARG2$ разряда;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	r	r	r

4.4.4.39. sar (Shift Arithmetic Right)

Сдвиг арифметический аргумента вправо

sar $ARG1$, $ARG2$

Назначение: операция арифметического сдвига аргумента вправо

Синтаксис:

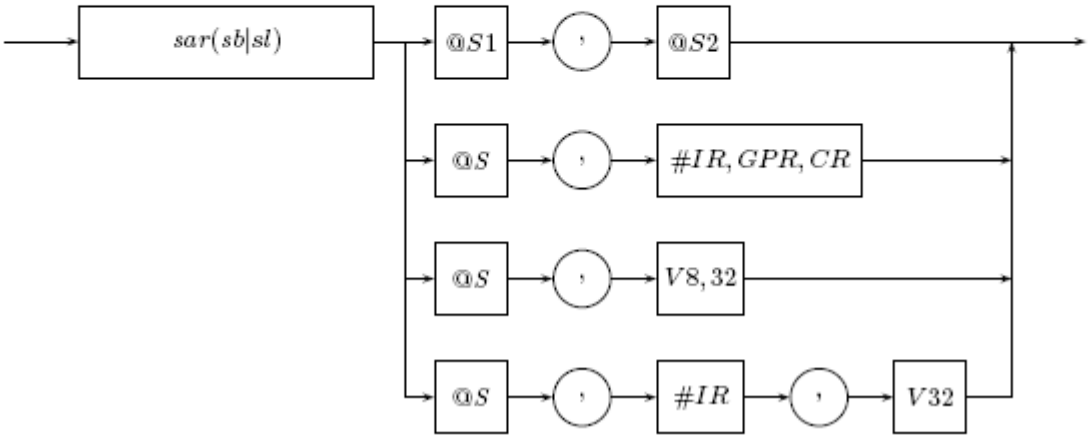


Рис. 54: Синтаксическое описание команды *sar*

Наличие результата: да

Алгоритм работы:

- выполнить арифметического сдвиг аргумента $ARG1$ вправо на $ARG2$ разряда;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	r	r	r

4.4.4.40. sbb (SuBtract with Barrow (Carry Flag))

Вычитание с заёмом

sbb $ARG1,\ ARG2$

Назначение: операция целочисленного вычитания с учётом флага переноса (CF) результата предыдущего вычитания командой *sub*

Синтаксис:

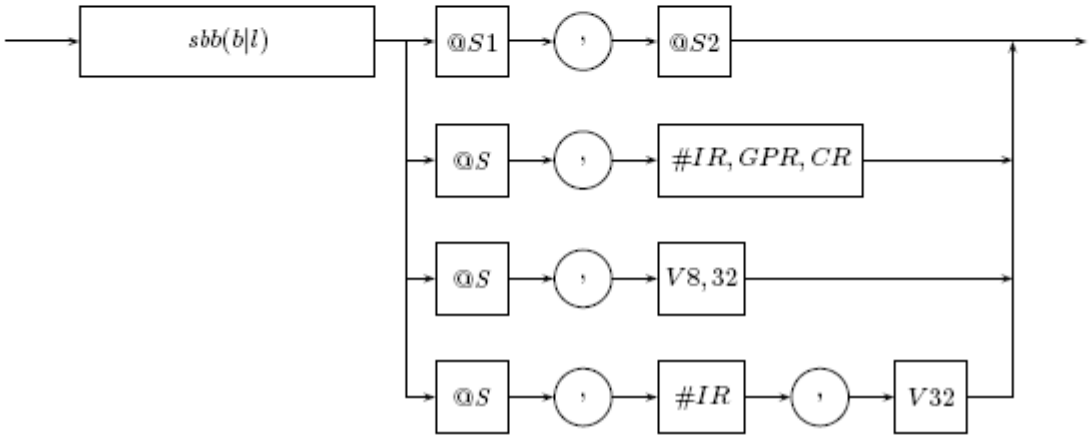


Рис. 55: Синтаксическое описание команды *sbb*

Наличие результата: да

Алгоритм работы:

- выполнить вычитание из значения аргумента $ARG2$ значение флага переноса (CF) аргумента $ARG1$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	r	r	r

4.4.4.41. set (SET)

Установка

set $ARG1$, $ARG2$

Назначение: операция установки значения регистра

Синтаксис:

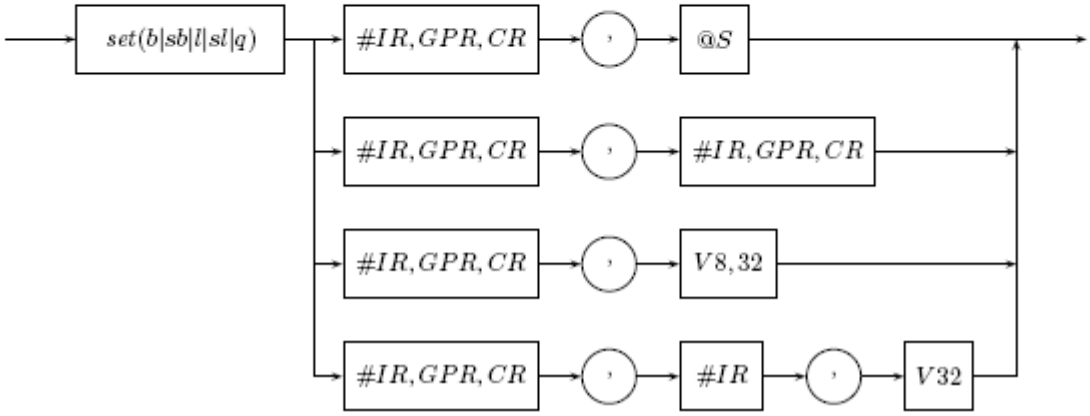


Рис. 56: Синтаксическое описание команды *set*

Наличие результата: нет

Алгоритм работы:

– установить значение регистра, заданного аргументом $ARG1$, равным значению аргумента $ARG2$, размером байт (знаковый или беззнаковый), слово (знаковое или беззнаковое), двойное слово в зависимоти от типа команды;

Интерпретация значений аргументов: согласно выше описанным синтаксису и алгоритму работы, первый аргумент команды определяет номер или имя регистра (в отличие от большенства остальных команд, первй аргумент которых определяет номер ссылки на

результат предшествующей команды), значение которого необходимо установить; значение второго аргумента, сохраняемое в указанном регистре, формируется согласно общим правилам формирования второго аргумента команд.

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

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

Пример:

```
1  .data
2
3  A:
4       .float 0f-1.2548E2
5  B:
6       .quad 0x010203040506O7F8
7
8  .text
9
10 C:
11      jmp D
```

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

Пояснения к примеру

- в строке №1 директивой ассемблера .data устанавливается текущая секция ассемблирования — секция инициализированных данных data;
- в строке №3 объявляется символ (идентификатор) *A*, который является меткой в текущей секции ассемблирования (data), и инициализируется текущим значением адресса ассемблирования;
- в строке №4 директивой ассемблера .float в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 32-х рарядное 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 адресса ассемблирования;
- в строке №6 директивой ассемблера .quad в текущую секцию ассемблирования по текущему адресу ассемблирования записывается 64-х рарядное целое беззнаковое число $0x01020304050607F8$;
- в строке №8 директивой ассемблера .text устанавливается текущая секция ассемблирования — секция исполняемых инструкций text;
- в строке №10 объявляется символ (идентификатор) C, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №11 командой jmp безусловно устанавливается адрес следующего параграфа, равным D;
- в строке №13 командой rdl, читается из памяти данных 32-х разрядное целое беззнаковое числа по адресу A и помещаются в коммутатор;
- в строке №14 командой setl в регистр общего назначения №0 помещается результат выполнения предшествующей команды: @1 — результат выполнения команды чтения в строке №13;
- в строке №15 командой complete завершается текущий параграф; происходит фактическая установка значения регистра общего назначения №0.
- в строке №17 объявляется символ (идентификатор) D, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;
- в строке №18 командой setb в индексный регистр №32 помещается значение $0xAA$ (значения старших разрядов с 8 по 63 будут нулевые, так как тип операции беззнаковый байт);

- в строке №20 командой rdq, читается из памяти данных 64-х разрядное целое беззнаковое число по адресу B и помещается в коммутатор;
- в строке №21 командой setq в управляющий регистр PSW помещается результат выполнения предшествующей команды: @1 — результат выполнения команды чтения в строке №20;
- в строке №22 командой setsb в регистр общего назначения №0 помещается результат выполнения предшествующей команды: @2 — результат выполнения команды чтения в строке №20 (помещается значение 0xF8, значения старших разрядов с 8 по 63 будут равны значению 7 разряда, так как тип операции знаковый байт);
- в строке №24 командой getsl извлекается значение регистра общего назначения №0, установленное в предыдущем параграфе (не смотря на наличие команды setsb в строке №22);
- в строке №25 командой addsl складываются результат выполнения предшествующей команды: @1 — результат выполнения команды извлечения в строке №24 (число $0x42faf5c3$ — представление числа $-1.2548 * 10^2$ в IEEE754), и константное значение 1; оба аргумента команды addsl согласно суффиксу sl интерпретируются как 32-х разрядные целые знаковые числа; результат выполнения команды также интерпретируются как 32-х разрядное целое знаковое число и помещается в комутатор;
- в строке №26 командой setsl в регистр общего назначения №1 помещается результат выполнения предшествующей команды: @1 — результат выполнения команды сложения в строке №25 (помещается значение $0x42faf5c4$, значения старших разрядов с 32 по 63 будут равны значению 31 разряда, так как тип операции знаковый длинный);
- в строке №28 командой setl в индексный регистр №32 помещается значение $0xBBCCDDEE$ (значения старших разрядов с 32 по 63 будут нулевые, так как тип операции беззнаковый длинный);
- в строке №29 командой complete завершается текущий параграф; происходит фактическая установка значения регистров общего назначения №№0,1, управляющего ре-

гисра PSW, индексного регистра №32 в одно из двух возможных значений (строки №№18,28).

4.4.4.42. sll (Shift Logical Left) / sal (Shift Arithmetic Left)

Сдвиг логический/арифметический аргумента влево

$(sll|sal)$ $ARG1$, $ARG2$

Назначение: операция логического/арифметического сдвига аргумента влево

Синтаксис:

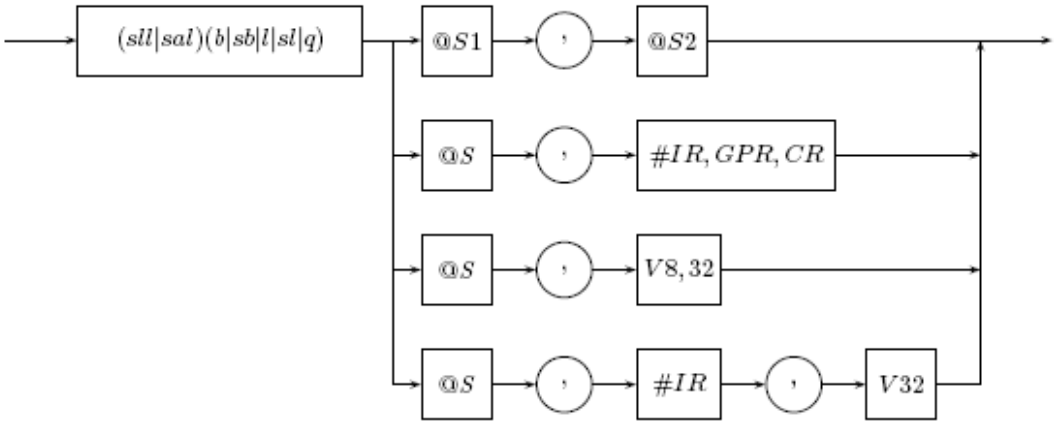


Рис. 57: Синтаксическое описание команды sll/sal

Наличие результата: да

Алгоритм работы:

- выполнить логический/арифметический сдвиг аргумента $ARG1$ влево на $ARG2$ разряда;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	r	r	r

4.4.4.43. slr (Shift Logical Right)

Сдвиг логический аргумента вправо

slr ARG1, ARG2

Назначение: операция логического сдвига аргумента вправо

Синтаксис:

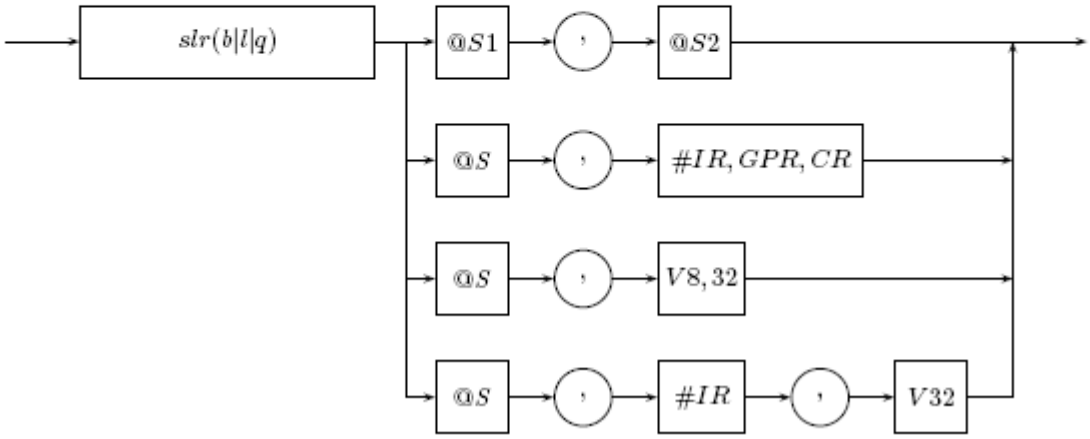


Рис. 58: Синтаксическое описание команды *slr*

Наличие результата: да

Алгоритм работы:

- выполнить логический сдвиг аргумента $ARG1$ вправо на $ARG2$ разряда;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	CF	OF	ZF
r	r	r	r

4.4.4.44. sqrt (SQuare RooT)

Обратное вычитание

sqrt ARG

Назначение: операция извлечения квадратного корня вещественного аргумента

Синтаксис:

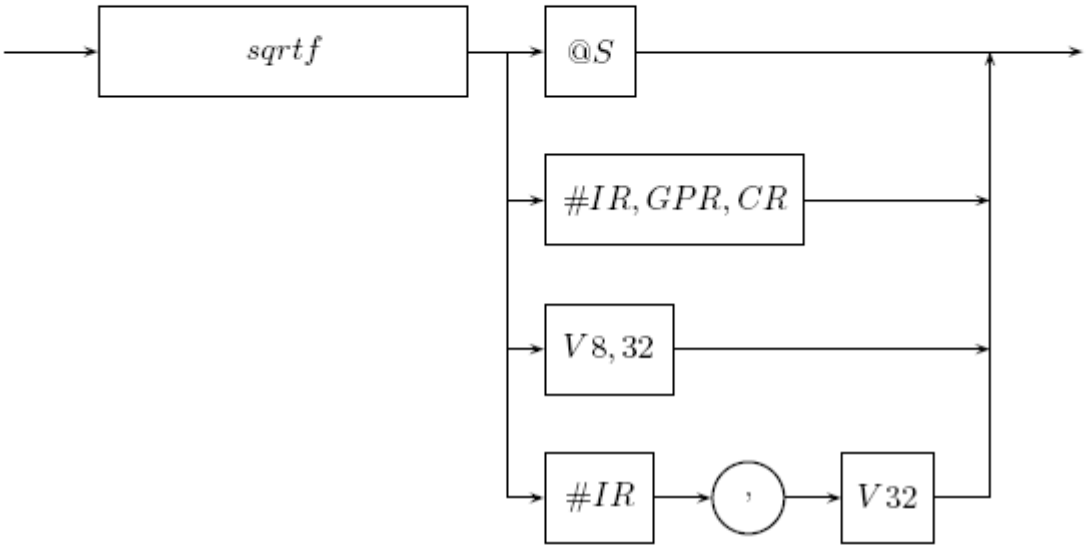


Рис. 59: Синтаксическое описание команды *sqrt*

Наличие результата: да

Алгоритм работы:

- выполнить извлечение квадратного корня вещественного аргумента $\sqrt{ARG}$;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
0	0	0	r

4.4.4.45. sub (SUBtract)

Вычитание

sub ARG1, ARG2

Назначение: операция вычитания двух аргументов

Синтаксис:

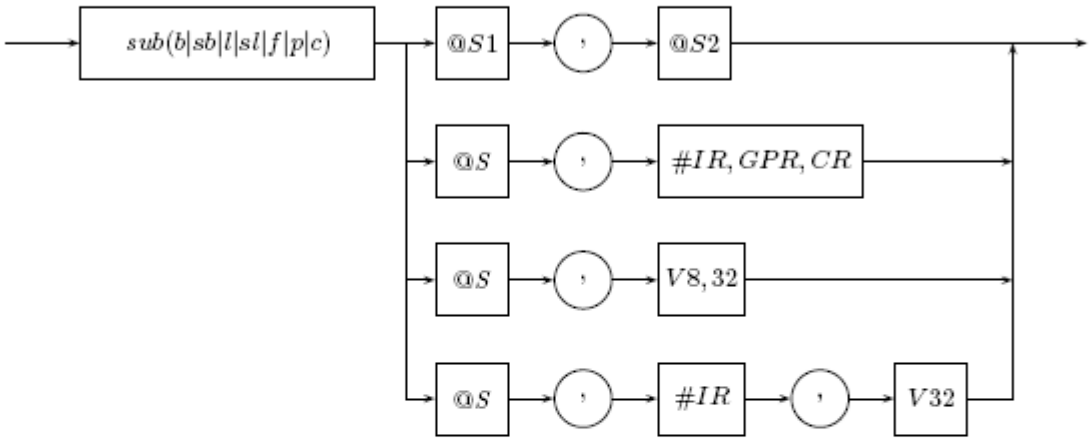


Рис. 60: Синтаксическое описание команды *sub*

Наличие результата: да

Алгоритм работы:

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

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
r	r	r	r

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

Пример:

```
1  .data
2
3  B:
4        .float 0f12.8
5
6  .text
7
8  A:
9        rdl B
10       subf @1, 0f-5.6
11       wrl @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 адресса ассемблирования;
- в строке №4 директивой ассемблера .float в текущую секцию ассемблирования записывается 32-х рарядное вещественное число по текущему адресу ассемблирования;
- в строке №6 директивой ассемблера .text устанавливается текущая секция ассембли-

рования — секция исполняемых инструкций text;

– в строке №8 объявляется символ (идентификатор) A, который является меткой в текущей секции ассемблирования (text), и инициализируется текущим значением адресса ассемблирования, а также начинает новый параграф;

– в строке №9 командой rdl читается из памяти данных 32-х разрядное целое беззнаковое число по адресу B и помещается в коммутатор;

– в строке №10 командой subf выполняется операция вычитания результата выполнения предшествующей команды: @1 — результат выполнения команды чтения в строке №9, и константы -5.6; оба аргумента команды subf согласно суффиксу f интерпретируются как вещественные числа одинарной точности размерностью 32 бита; результат выполнения команды также интерпретируются как 32-х разрядное вещественное число одинарной точности и помещается в комутатор;

– в строке №11 командой wrl осуществляется запись в память данных по адресу $B + 4$ результата выполнения предшествующей команды: @1 — результат выполнения команды вычитания в строке №10;

– в строке №12 командой complete завершается текущий параграф.

4.4.4.46. wr (WRite)

Запись

$wr\ ARG1,\ ARG2$

Назначение: операция записи значения в память данных

Синтаксис:

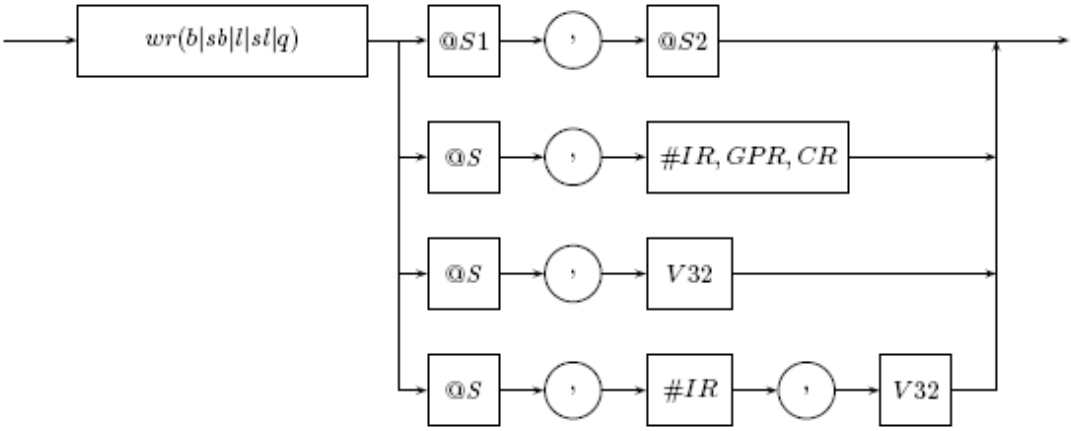


Рис. 61: Синтаксическое описание команды wr

Наличие результата: нет

Алгоритм работы:

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

Интерпретация значений аргументов: согласно выше описанному алгоритму работы, значение первого аргумента интерпретируется как оно есть (непосредственно используется), значение второго аргумента всегда интерпретируется как адрес памяти данных, по которому осуществляется запись значения первого аргумента. Другими словами данная команда всегда обращается к памяти данных вне зависимости от варианта формирования значения второго аргумента. Если для формирования значения второго 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, как знакового, так и беззнакового, в память данных. По возможности следует располагать данную команду в конце параграфа.

Пример:

```
1  .data
2
3  A:
4        .float  0f-1.25E-1
5
6  .text
7
8  B:
9        rdl A
10       getl 0x3f028f5c; представление числа 0.51 в IEEE754
11       addf @1, @2
12       wrl @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 рарядное 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 адресса ассемблирования, а также начинает новый параграф;
- в строке №9 командой rdl, читается из памяти данных 32-х разрядное целое беззнаковое числа по адресу A и помещаются в коммутатор;
- в строке №10 командой getl помещается в коммутатор 32-х разрядное целое беззнаковое число (константа $0x3f028f5$);
- в строке №11 командой addf складываются результаты выполнения двух предшествующих команд: @1 — результат выполнения команды извлечения в строке №10, @2 — результат выполнения команды чтения в строке №9; оба аргумента команды addf согласно суффиксу f интерпретируются как вещественные числа с плавающей точкой одинарной точности; результат выполнения команды также интерпретируются как вещественное число с плавающей точкой одинарной точности и помещается в коммутатор;
- в строке №12 командой wrl осуществляется запись в память данных по адресу $A + 4$ результата выполнения предшествующей команды: @1 — результат выполнения команды сложения в строке №11;
- в строке №13 командой complete завершается текущий параграф.

4.4.4.47. xor (XOR)

Логического сложение

xor *ARG*1, *ARG*2

Назначение: операция логического сложения по mod2 двух аргументов

Синтаксис:

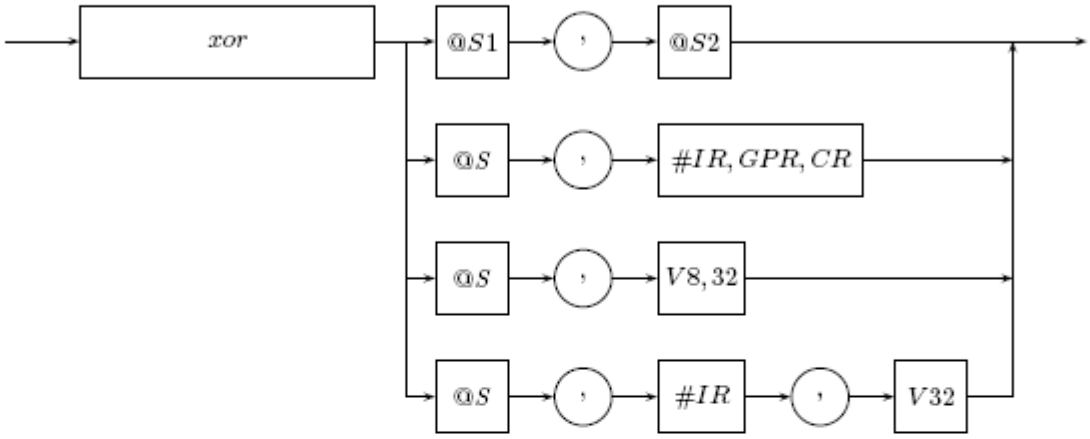


Рис. 62: Синтаксическое описание команды *xor*

Наличие результата: да

Алгоритм работы:

- выполнить логическое сложение по mod2 *ARG*1 ^*ARG*2;
- установить флаги;
- поместить результат вместе с флагами в коммутатор;

Состояние флагов результата после выполнения команды:

SF	*CF*	*OF*	*ZF*
r	0	0	r

4.5. Система директив ассемблера

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

4.5.1. .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).

4.5.2. .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. Есдли для достижения за-

прошенного 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.

4.5.3. .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.

4.5.4. .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.

4.5.5. .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.

4.5.6. .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.

4.5.7. .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.

4.5.8. .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.

4.5.9. .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.

4.5.10. .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.

4.5.11. .end

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

4.5.12. .endif

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

4.5.13. .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.

4.5.14. .equiv symbol, expression

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

4.5.15. .eqv symbol, expression

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

4.5.16. .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)

4.5.17. .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)

4.5.18. .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.

4.5.19. .float flonums

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

4.5.20. .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.

4.5.21. .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.

4.5.22. .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’.

4.5.23. .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.

4.5.24. .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.

4.5.25. .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.

4.5.26. .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. Есдли 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.

4.5.27. .print string

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

4.5.28. .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.

4.5.29. .rept count

Данная директива используется для организации повторения count раз последовательности инструкций, расположенных между данной директивой (.rept) и терминирующей её директивой .endr.

4.5.30. .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.

4.5.31. .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.

4.5.32. .single flonums

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

4.5.33. .size name, expression

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

4.5.34. .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.

4.5.35. .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.

4.5.36. .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.

4.5.37. .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.

4.5.38. .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)

4.5.39. .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)

4.5.40. .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.

4.6. 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 компоновщиком (линковщиком) 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.

В мультиклеточном процессоре используется раздельная память программ для каждого процессорного блока и общая паямть данных. Для указания области памяти, в которую будут ассемблироваться ниже следующие инструкции исходного текста программы, в ассемблере представлены следующие директивы: .data, .bss, .text.

Использование директивы .data в исходном тексте программы переключает текущую секцию ассемблирования на секцию data. В данной секции сохраняются начальные данные программы, которые в процессе загрузки программы располагаются в памяти данных процессора. Для инициализации данной секции в исходном тексте программы могут быть использованы такие директивы ассемблера как .ascii, .asciz, .byte, .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, .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 данных 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 парограммы, которые в процессе загрузки программы располагаются в памяти программ процессора. Распределение исполняемых инструкций между процессорными блоками (у каждого процессорного блока своя память программ) осуществляется на этапе компоновки программы компоновщиком (линковщиком).

В каждой из выше перечисленных секций возможна установка метки. Метка определяется как символ, за которым следует двоеточие «:». Метки используются в качестве адреса паямти данных или программ. 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 итерпретируется 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 необход
       имого размера в байтах без изменения текущей секции ассемблир
       ования
20 */
21 .lcomm Ba, 24
22
23 /*
24   Резервирование неинициализированного блока памяти данных
25 */
26 .bss; текущая секция ассемблирования
27
28 Bb: // объявление метки
29     /* директивы инициализации паямти данных, например */
```

```
30      .long  ,,,
31      .byte
32      .skip  12
33
34  Bc: // объявление метки
35      /* директивы инициализации паямти данных */
36
37  /*
38    Размещение исполняемых инструкций в памяти программ
39  */
40  .text; текущая секция ассемблирования
41
42  Ta: // объявление метки, а также начало нового параграфа
43      rdl Da
44      rdl Da + 4
45      mull @1, @2
46      wrl @1, Bc
47  complete // завершение параграфа
48
49  Tb: // объявление метки, а также начало нового параграфа
50      /* команды процессора */
51  complete // завершение параграфа
```

4.7. Прерывания и их обработка

Система прерываний мультиклеточного процессора допускает обработку 32 прерываний.

Номер	Описание
0	Немаскируемое внутреннее прерывание (INMI)
1	Немаскируемое внешнее прерывание (ENMI)
2	Немаскируемое исключение в аппаратной части (PERE)
3	Немаскируемое программное исключение (PPGE)
4	Маскируемое программное исключение (MPRGE)
5	Прерывание от системного таймера (SWT)
6	Программное прерывание (SWI)
7	Маскируемое прерывание от UART0
8	Маскируемое прерывание от UART1
9	Маскируемое прерывание от UART2
10	Маскируемое прерывание от UART3
11	Маскируемое прерывание от I2C0
12	Маскируемое прерывание от I2C1
13	Маскируемое прерывание от SPI0
14	Маскируемое прерывание от SPI1
15	Маскируемое прерывание от SPI2
16	Маскируемое прерывание от I2S0
17	Маскируемое прерывание от GPTIM0
18	Маскируемое прерывание от GPTIM1
19	Маскируемое прерывание от GPTIM2
20	Маскируемое прерывание от GPTIM3
21	Маскируемое прерывание от GPTIM4
22	Маскируемое прерывание от GPTIM5
23	Маскируемое прерывание от GPTIM6

Продолжение на следующей странице

Номер	Описание
24	Маскируемое прерывание от PWM0
25	Маскируемое прерывание от RTC
26	Маскируемое прерывание от GPIOA
27	Маскируемое прерывание от GPIOB
28	Маскируемое прерывание от GPIOC
29	Маскируемое прерывание от GPIOD
30	Маскируемое прерывание от ETHERNET0
31	Маскируемое прерывание от USB0

Источник с номером «0» имеет наивысший приоритет при обработке прерываний.

Немаскируемые прерывания:

Прерывания с номерами 0 — 3 являются немаскируемыми. Немаскируемые прерывания приводят к немедленному переходу на программу обработки прерываний. Их нельзя запретить, они разрешены сразу после начала работы ядра.

Маскируемые прерывания:

Прерывания с номерами 4 — 31 являются маскируемыми, они глобально разрешаются битом ONIRQS в регистре PSW (16.3.4). Индивидуальное разрешение задается регистром MSKR (16.3.6).

Для работы системы прерывания и фунционирования программ оборабокти прерываний имеются следующие регистры: INTR (16.3.5), MSKR (16.3.6), ER (16.3.7), IRETADDR (16.3.8), IHOOKADDR (16.3.11), INTNUMR (16.3.12).

Регистры INTR, MSKR, ER, INTNUMR относятся к управлению контроллером прерываний. Регистры IRETADDR, IHOOKADDR используются программным алгоритмом. В памяти мультиклеточного процессора не выделено какой-либо фиксированной зоны для размещения обработчика прерываний. Программист может разместить обработчики прерываний в любом месте адресного пространства памяти программ. При возникновении прерывания ядро перей-

дет по адресу, записанному в регистре IHOOKADDR. По данному адресу программист может расположить первичный обработчик прерываний, который может осуществить диспетчеризацию и перенаправить программу на необходимый адрес, где располагается обработчик для конкретного прерывания. Адрес возврата автоматически запоминается в регистре IRETADDR, программист имеет к нему полный доступ.

Порядок обработки прерываний:

При возникновении прерывания, контроллер прерываний определяет самое приоритетное прерывание и формирует сигнал, который приводит к установке соответствующего бита регистра INTR. Вся программа разбита на «параграфы». На время выполнения «параграфа» все прерывания запрещены, кроме немаскируемых, которые могут прервать работу процессорного ядра в любое время. После завершения «параграфа» возможен переход на адрес первичного обработчика прерываний, если был запрос прерывания, прерывание не маскировано (выставлен соответствующий бит регистра MSKR) и нет глобального запрета прерываний (бит ONIRQS в регистре PSW). В регистр IRETADDR автоматически записывается адрес следующего «параграфа», в регистр INTNUMR — номер запрошенного прерывания, а также устанавливается глобальный запрет прерываний. Ядро переходит на выполнение алгоритма, размещенного по адресу, записанному в регистре IHOOKADDR. После завершения работы обработчиков прерываний программный алгоритм должен выставить глобальное разрешение прерываний (бит ONIRQS в регистре PSW), в случае необходимости, и осуществить выход из прерывания.

5. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ ПО РЕДАКТОРУ СВЯЗЕЙ

5.1. Общие сведения о редакторе связей

Компоновщик или редактор связей для мультиклеточного процессора осуществляет сборку образа исполняемой программы путём объединения данных одного или нескольких объектных файлов и, в случае необходимости, объектных файлов статических библиотек, а также связывание символов объектных файлов, так чтобы все ссылки на эти символы имели верные адреса времени выполнения.

Объектные файлы представляют собой результат компиляции ассемблером исходного кода программы.

Статические библиотеки представляют собой обычные архивы объектных файлов, для создания которых используется команда ar.

5.2. Использование редактора связей

Представленный редактор связей запускается из командной строки командой ld, аргументами которой являются один или несколько объектных файлов. Поддерживаются также следующие опции:

-L, --library=namespec — добавить файл архива namespec к списку файлов, используемых для сборки. Данная опция может использоваться многократно. Редактор связей будет искать путь к библиотеке для имени файла libnamespec.a. Поиск архивного файла осуществляется редактором связей только один раз, в том месте, где он указан в командной строке. Если в архиве определяется символ, который был неопределённым в каком-либо объектном файле и этот объектный файл указан в командной строке до файла архива, компоновщик добавит в результирующий файл образа соответствующие объектные файлы архива. В тоже время, наличие неопределённого символа в каком-либо объектном файле, указанного в командной строке после файла архива, не приведёт к повторному поиску символа в ранее указанном архивном файле. Любой архивный файл може быть указан в камандной строке несколько раз.

-L, - -library-path=searchdir — добавить путь searchdir в список путей, в которых редактор связей будет искать статические библиотеки (архивные файлы, задаваемые опцией -l). Данная опция может использоваться многократно. Директории используются в том порядке, в котором они указаны в коммандной строке. Все значения опции -L применяются ко всем значениям опции -l, независимо от того, в каком порядке эти опции указаны.

-e, --entry=entry_point — использовать значение символа entry_point в качестве

стартового адреса начала выполнения программы. Если символ entry_point не найден, то будет осуществлена попытка разбора entry_point и преобразования его в число и, в случае успеха, использования данного числа в качестве стартового адреса начала выполнения программы.

-o, --output=FILE — поместить вывод в файл образа FILE. Если данная опция не используется, то имя выходного файла по умолчанию — image.bin.

-M, --print-map — вывести в стандартный поток вывода информацию о размещении данных объектных файлов в памти и значениях, назначенных символам.

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

6. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ ПО ЗАГРУЗЧИКУ

6.1. Общие сведения о загрузчике

Программный загрузчик для мультиклеточного процессора осуществляет загрузку образов памяти исполняемой программы в ПЗУ отладочной платы.

Файл образов памяти исполняемой программы представляет собой результат сборки программы редактором связей: файл образов памяти программ и памяти данных исполняемой программы.

6.2. Использование загрузчика

Программный загрузчик запускается из командной строки командой ploader, аргументом которой является файл образов памяти исполняемой программы. Поддерживаются также следующие опции:

-l, --list - показать список доступных ftdi устройств.

-d, --device=deviceName - установить имя ftdi устройства в deviceName, используемого для загрузки.

-f, --frequency=frequencyValue - установить частоту ftdi устройства в frequencyValue, используемого для загрузки (значение по умолчанию 10000 кГц).

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

Предположим, что image.bin является файлом образов памяти исполняемой программы. Тогда для загрузки этого образа в ПЗУ отладочной платы необходимо в командной строке выполнить следующую команду:

```
ploader image.bin
```

В этом случае будет использовано первое найденное подходящее ftdi устройство для загрузки файла image.bin в ПЗУ отладочной платы с частотой 10000 кГц.

Для использования конкретного ftdi устройства с заданной частатой может быть использована следующая команда:

```
ploader image.bin -d"PicoTAP A" -f20000
```