Кросс-средства разработки программ

Описание языка ассемблера

Справочное руководство

НАУЧНО-ТЕХНИЧЕСКИЙ ЦЕНТР

Содержание

Список иллюстраций

Список Таблиц

Предисловие

В предисловии описывается назначение и состав документа, приводится краткий обзор разделов, определяется стиль и символьные нотации, используемые в документе.

Примечание

В данном руководстве содержится информация об устройстве одного из программно совместимых процессоров серии 1879ВМ, достаточная для того, чтобы разрабатывать программы на языке ассемблера.

О руководстве

Данное руководство содержит необходимые сведения по языку ассемблера. Ассемблер является одной из программ кросс-средств разработки программ для программно совместимых процессоров серии 1879ВМ (NM640x), построенных на архитектуре NeuroMatrix™. (Далее в тексте сокращенно именуется КРОСС-СРЕДСТВА, или указывается с помощью аббревиатуры – NM SDK) Базовой структурной единицей устройства является ядро nmc (neuro matrix core), таких ядер не микросхеме может быть несколько. Каждое ядро nmc имеет свой поток команд, интерфейс к памяти, набор периферии и вообще представляет собой относительно независимую единицу.

Данное руководство содержит следующую информацию:

- описание структуры и функционирования основных узлов процессора с точки зрения программиста;
- описание регистров процессора;
- описание механизмов использования внутренних блоков памяти векторного процессора (ВП);
- описание всех конструкций языка ассемблера с примерами их использования;
- структурированный список всех скалярных и векторных команд;
- описание различий в языке ассемблера процессоров архитектуры NeuroMatrix™ разных поколений.

Как организован данный документ

Данный документ структурно подразделяется на разделы, подразделы, пункты и подпункты.

Документ разбит на шесть разделов:

Раздел 1 **Обзор Структуры Процессора**

Содержит справочную информацию о структуре процессора с точки зрения программиста. Даёт общее представление о функционировании основных вычислительных узлов.

Раздел 2 **Обзор Основных Элементов Языка Ассемблера**

Содержит полную информацию о структуре и правилах построения программ на языке ассемблера, включает примеры использования различных синтаксических конструкций.

Раздел 3 **Описание Регистров Процессора**

Содержит подробную информацию о составе и различных типах регистров, описание полей регистров управления, правила использования и назначение векторных регистров, примеры использования регистров в различных ассемблерных инструкциях.

Раздел 4 **Формат Ассемблерных Инструкций**

Содержит информацию о формате скалярных и векторных инструкций ассемблера, описание состава и структуры машинных команд.

Раздел 5 **Набор Инструкций Языка Ассемблера**

Содержит полный набор существующих инструкций процессора, их синтаксис, положение тех или иных операций или команд процессора в ассемблерной инструкции, примеры ассемблерной инструкции с использованием той или иной операции или команды.

Соглашения о нотациях

В данном справочном руководстве используются следующие типографические нотации:

`Courier` Так помечается текст, который может быть набран пользователем с клавиатуры: исходные тексты на языке ассемблера.

`Courier`	Отмечает текст, который должен быть заменен пользовательской информацией, например, реальным значением константы.
Текст или Текст	Так помечается текст, на который необходимо обратить особое внимание.
//Текст	Так помечаются комментарии к программам.

Примечание

Данное примечание представляет собой пример того, как оформлены все важные замечания и комментарии, возникающие по ходу описания.

Как оформлять замечания и предложения

По документации

Если при работе с документацией, описывающей процессор и его кросс-средства разработки программ, у Вас возникли сложности, например: Вы считаете, что материал изложен недостаточно подробно для понимания, пожалуйста, присылайте свои замечания в следующем виде:

- обозначение и наименование документа;
- номера страниц, на которые приводится ссылка;
- краткое описание замечания.

По работе программ из состава NMSDK

Если при работе с какой либо из программ NMSDK у Вас возникли сложности, пожалуйста, присылайте свои замечания в следующем виде:

- номер версии программы из состава кросс-средств разработки программ, при работе с которой у Вас возникли сложности;
- небольшой фрагмент исходного кода, на котором проявляется ошибка;
- краткое описание ее проявления и возможные предположения о причинах ее возникновения, если таковые имеются.

Замечания присылайте по адресу: nm-support@module.ru

1 Краткий Обзор Структуры Процессора

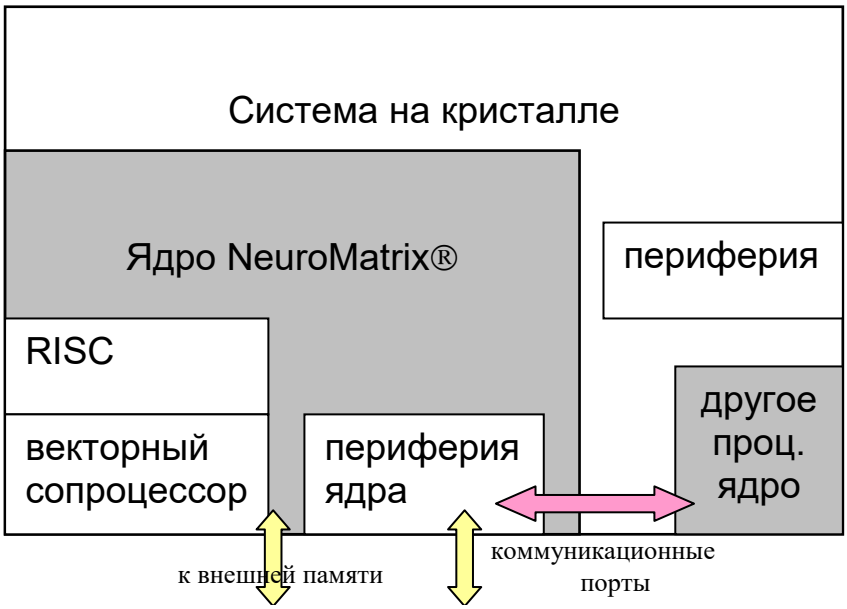
1.1 Введение

Прежде чем приступить к описанию языка ассемблера, необходимо ввести некоторые понятия, связанные со структурой процессора, на которые в дальнейшем можно будет ссылаться при описании тех или иных конструкций языка..

1.2 Внешний интерфейс процессора

Структура процессора, описанная в данном разделе, отражает взгляд программиста, поэтому некоторые понятия, которые редко используются при программировании, опущены. Более детальный обзор процессора приведен в руководстве по эксплуатации микросхемы.

Рис. 1-1 Условная схема устройства



Устройство использует 32-х разрядные адреса.

Помимо работы с внешней памятью процессор может принимать и передавать данные через коммуникационные порты.

1.3 Общее описание внутренней структуры процессора

Основной особенностью архитектуры NeuroMatrix является векторный процессор, он работает совместно со скалярным процессором и периферийными устройствами. До появления устройств на базе nmc четвёртого поколения (nmc4), существовал лишь один вариант векторного процессора, - устройство для перемножения матриц чисел с фиксированной точкой, данный вариант будем считать базовым.

Данный 64-х разрядный процессор предназначен для выполнения параллельных операций и способен выполнять одновременно до 2048 операций за один такт. Его архитектура позволяет менять отношение производительность/точность, изменяя разбиение данных (векторов) на элементы с одного бита до 64-х.

Векторный процессор выполняет множественные действия над данными за одну инструкцию, так называемая SIMD (single instruction, multiple data) параллельная обработка.

Начиная с nmc4, архитектура NeuroMatrix имеет более гибкую векторную подсистему: появляется возможность применять сопроцессор(ы) другого типа. В частности, NM6407 - это многопроцессорная система, состоящая из двух процессорных ядер NeuroMatrix, одно с традиционным сопроцессором, а другое с векторным сопроцессором плавающей точки.

Минимальной адресуемой единицей внешней памяти для процессора NM6403 является 32-х разрядное слово. Процессор имеет набор команд, состоящий как из 32-х, так и из 64-х разрядных команд. Если в команде содержится 32-х разрядная константа, то команда трактуется, как 64-х разрядная, в остальных случаях она является 32-х разрядной.

Приведем примеры 32-х и 64-х разрядных команд процессора:

```
ar0 = 80808080h;        // 64-х разрядная команда
                        // (содержит константу).
gr0 = [ar0];            // 32-х разрядная команда.
```

Более подробную информацию о системе команд можно найти в разделе 5.

1.3.1 Описание основных элементов скалярного процессора

Скалярный процессор представляет собой RISC ядро, отвечающее за подготовку данных для векторного процессора. Он также может использоваться и как самостоятельный вычислительный блок.

СП имеет 8 адресных регистров и 8 регистров общего назначения. Помимо этого, существует набор специализированных регистров, подробная информация о которых содержится в подразделе 3.2.Специальные регистры.

СП позволяет осуществлять следующие операции:

- различные виды адресации с модификацией адресных регистров;
- чтение/запись в память как 32-х разрядных слов, так и пар слов, образующих 64-х разрядное число.
- все виды арифметических и логических операций над регистрами общего назначения с модификацией и без модификации флагов состояния;
- различные типы сдвигов, в том числе на произвольное количество битов;
- условные и безусловные переходы, в том числе отложенные переходы (см. пункт 5.1.9);
- вызовы функций с записью в стек адреса возврата, в том числе и отложенные вызовы функций;
- пошаговое умножение;
- управление периферийными устройствами;
- управление векторным процессором путем задания его конфигурации.

Полный набор скалярных команд процессора приведен в подразделе 5.1.Скалярные инструкции.

1.4 Векторный сопроцессор чисел с плавающей точкой

Сопроцессор состоит из четырёх одинаковых обрабатывающих ячеек и устройства переупаковки.

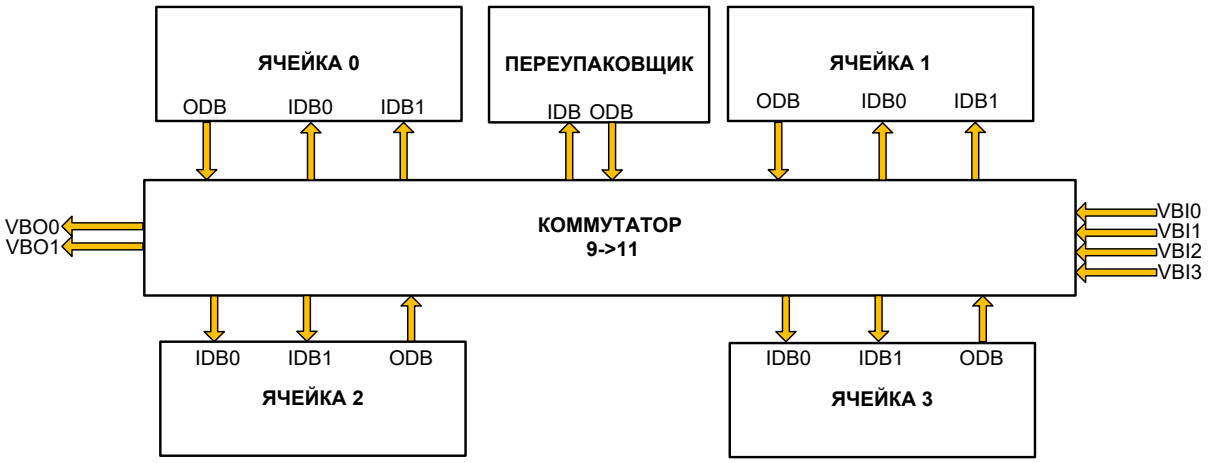


Рис.2 Структурная схема матрично-векторного сопроцессора

Каждая из обрабатывающих ячеек способна выполнять операции сложения, вычитания, умножения, формирования флагов, а также маскирования элементов над векторами чисел с плавающей точкой. Устройство переупаковки способно преобразовывать 32-х и 64-х разрядные данные как с плавающей, так и с фиксированной точкой между собой.

На приведённой схеме узла стоит обратить внимание на стрелки, представляющие шины данных, они отражают ограничения на возможность одновременной пересылки, к примеру, каждая ячейка может отдавать наружу только один вектор данных одновременно, а весь сопроцессор может писать в память одновременно только два вектора (при условии записи в разные банки, иначе вообще только один вектор).

Устройство позволяет менять настройки, такие как режим округления, и отслеживает исключительные ситуации, такие как переполнение, потеря точности или неправильная команда, при необходимости формируя прерывания.

Каждая векторная инструкция FPU относится к какой-то одной ячейке, (за исключением пересылок векторов между ячейками и операций переупаковщик-память).

В языке ассемблера номер ячейки указывается в начале векторной инструкции:

```
fpu 0   .float    vreg7= vreg0 * vreg0;
```

1.4.1 Ячейка сопроцессора

Каждая ячейка содержит операционный узел и набор векторных регистров.

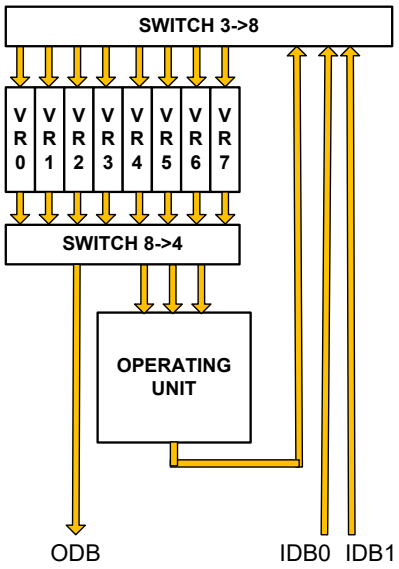


Рис. 3 Структура ячейки сопроцессора

Схема отражает порядок работы с устройством, - операционный узел общается с внешним миром только через векторные регистры.

Для векторной операции с плавающей точкой счётчик можно задать как традиционным образом, так и с помощью специального регистра.

1.4.1.1 Вычисления на FPU

Сценарий работы на FPU примерно таков: прочитанные из памяти данные при необходимости преобразуются в формат с плавающей точкой, записываются в векторные регистры. Над данными производится несколько арифметических операций, возможно, с пересылками промежуточных данных между разными ячейками и, если необходимо, после обратного преобразования, записываются в память.

Набор арифметических операций включает сложение, умножение с накоплением и взятие модуля, доступны также маскирование и формирование флагов для ветвления.

В арифметической инструкции указывается тип данных элемента, с помощью ключевых слов `.float`, `.double`, `.complex`, `.matrix`:

```
fpu 0 .float vreg7= vreg0 * vreg0;
```

1.4.1.2 Векторные регистры FPU

Векторный регистр хранит до тридцати двух 64-х разрядных слов, у каждой ячейки сопроцессора таких регистров восемь, их имена в языке ассемблера vreg0…vreg7

Содержимое регистра интерпретируется либо как вектор чисел double, либо как вектор чисел float, упакованных по две штуки в 64-х разрядное слово. Регистр «не знает» какого именно вида его данные, интерпретация содержимого задаётся в операционной векторной команде.

```
fpu 0  rep 32 vreg0= [ar0++]; // Чтение – тип не указываем
fpu 0 .float vreg7= vreg0 * vreg0;
```

Можно применить float-операцию к double-данным и получить мусор:

```
fpu 0  .float   vreg7= vreg0 * vreg0;   //   ок
fpu 0  .double  vreg2= vreg7 * vreg7; //
  Недетектируемая ошибка выполнения – неверная
интерпретация содержимого вектора
```

В то же время, длина вектора, определяемая количеством 64-х разрядных слов в регистре, контролируется счётчиком, привязанным к регистру, то есть, в арифметической инструкции указывается тип данных, но не количество элементов вектора, а количество указывается в операциях с памятью. Размер вектора – результата арифметической инструкции равен размерам векторов-аргументов, размеры аргументов, в свою очередь должны быть согласованы.

```
fpu 0  rep 32 vreg0= [ar0++];
fpu 0  rep 16 vreg1= [ar0++];
fpu 0 .float vreg7= vreg0 * vreg1; // Ошибка выполнения –
несогласованы размеры аргументов, будет сгенерировано
исключение
```

При любой записи в регистр старое содержимое перезаписывается. Есть возможность записывать по маске, при этом новое содержимое будет скомбинировано со старым. Можно, например, сравнить элементы вектора с нулём, а полученную маску использовать для перезаписи только отрицательных чисел

При чтении вектора из регистра содержимое сохраняется. Есть операции, считывающие из векторного регистра один 64-

элемент, который используется как константа на протяжении всей операции, так можно, например, умножить вектор на скаляр. Прочитанный таким способом аргумент остаётся в векторе, а выбирается элемент так: для первой инструкции после формирования вектора будет выбран первый элемент, для следующей второй и так далее и по кругу.

Для указания того, что операндом должен быть скаляр используется ключевое слово `.retrive`. Скаляром можно объявить любой операнд кроме самого первого.

```
fpu 0  rep 32 vreg0= [ar0++]; //   10,20,30,40...
fpu 0  rep 2 vreg1= [ar1++];  //   3,5
fpu 0  .double   vreg7= vreg0 * .retrive(vreg1);
fpu 0  rep 1 [ar2++]= vreg7;       //   30,60,90,120...
(3)
fpu 0  .double   vreg7= vreg0 * .retrive(vreg1);
fpu 0  rep 1 [ar2++]= vreg7;       //   50,100,150,200...
(5)
fpu 0  .double   vreg7= vreg0 * .retrive(vreg1);
fpu 0  rep 1 [ar2++]= vreg7;       //   30,60,90,120...
(3)
```

Обратите внимание, что две идентичные инструкции, третья и пятая, вызванные с одинаковыми векторами – аргументами, дают разный результат благодаря свойствам `.retrive`.

Другой важный момент: если `.retrive` используется совместно с типом float, то множитель – скаляр должен быть продублирован, - помещён в обе половины 64-разрядного слова.

Скопировать содержимое векторного регистра в регистр другой ячейки можно с помощью инструкции вида

```
fpu 1 vreg2= fpu 3 vreg4;
```

1.4.1.3 Пары векторных регистров

Есть несколько инструкций, которые могут работать сразу с парой векторных регистров. Например, при конвертации форматов из float в double для хранения результата нужно вдвое больше места, в этом случае, можно использовать регистровую пару для записи результата.

Другой случай – умножение с участием матрицы float 2x2, когда строки матрицы берутся из разных половин пары.

Векторные регистры объединяются в пары по правилу: перваячасть пары – регистр с чётным номером, вторая часть пары имеет номер на единицу больший, чем первая. Таким образом, всего есть четыре возможные пары: (0,1), (2,3), (4,5), (6,7).

```
fpu 1 .packer= (vreg4,vreg5) with .float <= .double;
fpu 3 .matrix vreg7= -vreg2* (vreg0,vreg1) - vreg3;
fpu 3 .matrix vreg7= -vreg2* .retrive(vreg0,vreg1) - vreg3;
```

При расчёте производительности следует учесть, что передача регистровой пары между ячейками и переупаковщиком выполнится медленнее, чем передача регистра.

1.4.1.4 Операции с памятью

Операция загрузки векторного регистра FPU во многом аналогична загрузке векторного FIFO процессора фиксированной точки, главное, поддерживает все обычные для NeuroMatrix способы адресации, но занимает отдельную инструкцию, а также позволяет указывать длину загружаемого вектора через специальный регистр vlen. **Важно:** в регистр vlen следует писать число **на единицу меньшее** желаемой длины вектора.

```
fpu 0  rep 32 vreg0= [ar0++]; //   чтение 32 64-бит. слов
//   ниже то же самое с помощью регистра
vlen= 31;
fpu 0  rep vlen vreg0= [ar0++];
```

В команде записи содержимого векторного регистра также следует явно указывать количество записываемых длинных слов. Запись в память не меняет состояние вектора.

1.4.1.5 Умножение с накоплением

FPU предоставляет несколько способов работы с векторами данных:

1. Почленное перемножение с накоплением векторов данных в формате double.
2. Почленное перемножение с накоплением векторов данных в формате float.
3. Почленное перемножение с накоплением векторов комплексных чисел с компонентами в формате float.
4. Векторная операция в которой на каждом шаге происходит умножение пары float-ов на матрицу 2x2 с накоплением. Матрица-множитель может как быть константой на протяжении выполнения операции, так и выбираться каждый

такт новая. В качестве источника данных для матрицы используется векторная регистровая пара из двух векторных регистров с последовательными номерами.

Операция умножения позволяет поэлементно перемножить два вектора и записать результат в третий. Тип данных

```
fpu 3   .float    vreg7= vreg0 * vreg1;
fpu 3   .float    vreg7= vreg0 * vreg1 + vreg2;
```

1.4.1.6 Сумма, разность и сравнение

1. Сумма и разность делается аналогично умножению с накоплением:

```
fpu 0   .double    vreg7= vreg0 + vreg1;
```

Главная особенность этих операций в возможности выполнить сравнение, - сформировать маску и флаги по полученному значению.

Сравнение позволяет поэлементно сравнить два вектора с записью результата в регистры масок. При этом номер бита в регистре масок связан с позицией элемента в векторе. Для векторов float-ов регистров масок два, – отдельно для младших и старших слов. Кроме того, признаки сравнения последней(?) пары элементов фиксируются отдельно, и их можно использовать для ветвления основной программы.

Флаги формируются по-умолчанию, чтобы не формировать их, используйте ключевое слово noflags, аналогично с вычислениями на скалярном процессоре:

```
fpu 3   .float    vreg7= vreg0 - vreg0 noflags;
```

Чтобы сформировать маску, необходимо операцию сложения или вычитания дополнить указанием предиката, по которому будет формироваться маска, доступны такие предикаты:

```
fpu 3   .float    vreg7= vreg0 - vreg1, set mask if >;
fpu 3   .float    vreg7= vreg0 - vreg1, set mask if <;
fpu 3   .float    vreg7= vreg0 - vreg1, set mask if =0;
fpu 3   .float    vreg7= vreg0 - vreg1, set mask if <>0;
fpu 3   .float    vreg7= vreg0 - vreg1, set mask if >=;
fpu 3   .float    vreg7= vreg0 - vreg1, set mask if <=;
```

Например, пусть имеем такое содержимое регистров:

vreg0 == {7, 6, 5, 4, 3, 2, 1, 0}

vreg1 == {0, 1, 2, 3, 4, 5, 6, 7}

```
fp0_dmask == 0
```

тогда после выполнения команды

```
fpu 0  . double vreg7= vreg0 - vreg1, set mask if >;
```

регистр маски будет содержать

```
fp0_dmask == 11110000B
```

Если надо только сравнить два числа, а результат вычитания или сложения не нужен, приёмник операции опускается:

```
fpu 3  .double vreg0 - vreg1, set mask if =0;
```

3. Можно посчитать модуль числа для типов float и double, обратить знак, или сделать и то и другое:

```
fpu 0  .float    vreg7= /vreg0/;
fpu 0  .double   vreg7= -vreg0;
fpu 0  .double   vreg7= -/vreg0/;
```

1.4.1.7 Использование масок

Маску, (обычно) полученную при сравнении векторов можно использовать в арифметической операции. При этом если значение замаскировано, соответствующий элемент приёмника сохранит прежнее значение:

```
//   vreg1 <- max( vreg1,0 )
fpu 1  .double vreg1 + vreg1, set mask if <;
fpu 1  .double vreg1= mask ? vreg1-vreg1 : vreg1;
```

“: vreg1” в приведённой комманде используется исключительно для придания сходства с тернарным оператором языка C «a ? в : с», его можно опустить, смысл не изменится:

```
fpu 1  .double vreg1= mask ? vreg1-vreg1;
```

Немного по-другому маска используется с операцией пересылки между регистрами, здесь элемент для замаскированного значения берётся из третьего регистра, кроме того регистр результата может принадлежать другой ячейке сопроцессора:

```
// cell_2:vreg7 <- max(cell_1:vreg1, cell_1:vreg0)
fpu 1  .double vreg0 - vreg1, set mask if >;
fpu 2  vreg7= fpu 1 mask ? vreg0 : vreg1;
```

1.4.1.8 Устройство упаковки и распаковки

Позволяет преобразовывать данные между форматами:

1. Вектор double-ов, ключевое слово `.double`
2. Вектор float-ов, ключевое слово `.float`
3. Разреженный вектор float-ов: `.float .in_high` или `.float .in_low` . Это означает, что в каждом 64-разрядном слове вектора лежит одно значение одинарной точности, в старшей(`.in_high`) или младшей(`.in_low`) половине.
4. Вектор 32 или 64-битных чисел в формате с фиксированной точкой. Позиция точки задаётся в специальном регистре: `.fixed_64, .fixed_32`
5. Разреженный вектор 32-битных чисел в формате с фиксированной точкой: `.fixed_32 .in_high` или `.fixed_32 .in_low`

Устройство может работать по схеме память-память или память-ячейка FPU. Переупаковка одного вектора размером не более 32 64-х разрядных слов делается двумя инструкциями. Первая инструкция отвечает за загрузку исходных данных и тип переупаковки, вторая – за выгрузку переупакованных данных.

1.4.2 Специальные регистры FPU

1.4.2.1 Скалярные чтение и запись специальных регистров FPU

Работа со специальными регистрами FPU аналогична работе со скалярными регистрами ядра, с некоторыми ограничениями, - нельзя напрямую пересылать данные из специального регистра FPU в скалярный регистр ядра и наоборот, за исключением пересылок через регистр «**sir**», который для таких пересылок и предназначен, нельзя также напрямую записать константу в специальный регистр.

```
fp_pack_exp= [ar0+gr0] with gr1= gr1 +1 noflags;
fp_pack_exp= gr0;   //    ошибка
sir= gr0;
fp_pack_exp= sir;
```

Специальные регистры, привязанные к конкретному FPU имеют в своём имени номер своей ячейки:

`fp0_flags, fp0_dmask, fp0_lmask, fp0_hmask`

1.4.2.2 Список специальных регистров FPU

Далее приведены мнемоники специальных регистров с кратким пояснением их функции. За подробной информацией обращайтесь к описанию архитектуры процессора. Имена регистров из описания архитектуры приведены в скобках, они отличаются от мнемоник ассемблера.

`fp_pack_exp` (RIER – имя регистра в описании архитектуры) – используется при переупаковке 1.1.3.8, задаёт позицию точки в числе с фиксированной точкой.

Регистры флагов и масок устанавливаются операцией сравнения векторов (1.1.3.6). Регистры флагов по функциям аналогичны флагам psw. Регистры масок позволяют заполнять вектора по применяемому поэлементно условию.

Обратите внимание, что маска для двойной точности с одной стороны и для одинарной с другой, это всего лишь разные интерфейсы к одному и тому же регистру данной процессорной ячейки! То есть, к примеру запись в `fp0_dmask` меняет значение `fp0_lmask` и `fp0_hmask`.

`fp0_flags ... fp3_flags` (FPFRi) – регистры флагов процессорной ячейки.

`fp0_dmask ... fp3_dmask` (DPMRi) – маски операций двойной точности.

`fp0_lmask ... fp3_lmask` (SPMRLi), `fp0_hmask ... fp3_hmask` (SPMRHi) – маски операций одинарной точности.

Ниже следуют регистры прерываний сопроцессора.

`fp_ir_bad_data` (FPIOIR) - информация о прерывании по некорректным данным

`fp_ir_overflow` (FPOIR) - информация о прерывании по переполнению

`fp_ir_underflow` (FPUIR) - информация о прерывании по потере значимости

`fp_ir_inexact` (FPIIR) - информация о прерывании по потере точности

`fp_ir_triple` (FPDLIR) - информация о прерывании по специфическому случаю потери точности при умножении матриц с накоплением, см. FPDLIR в описании архитектуры

`fp_ir_bad_code` (FPIEIR) - информация о прерывании по неправильной команде

`fp_irr` (FPIRR) - регистр запросов на прерывание. Данный регистр можно использовать только в команде clear:

```
fp_irr clear sir;
```

1.4.2.2.1 Режимы округления и разрешение параллельной работы fpu

`fp_ccr` (FPIRR) - конфигурационный регистр, содержащий бит разрешения параллельной работы, режимы округления и маски

прерываний сопроцессора. Он используется только для чтения, для изменения используйте команды, описанные ниже.

Следующий набор регистров реагирует не на записанное в них значение, а на сам факт записи. Для работы с такими регистрами в ассемблере предусмотрен специальный синтаксис. Далее расписаны соответствующие команды.

`set fp_round_to_nearest;` (RND_00) – установка режима округления к ближайшему;

`set fp_round_down;` (RND_01) – установка режима округления к меньшему;

`set fp_round_up;` (RND_10) – установка режима округления к большему;

`set fp_round_to_zero;` (RND_11) – установка режима округления к меньшему по модулю;

`set fp_wait;` (P_reset) – запрещение параллельной работы для сопроцессора плавающей точки;

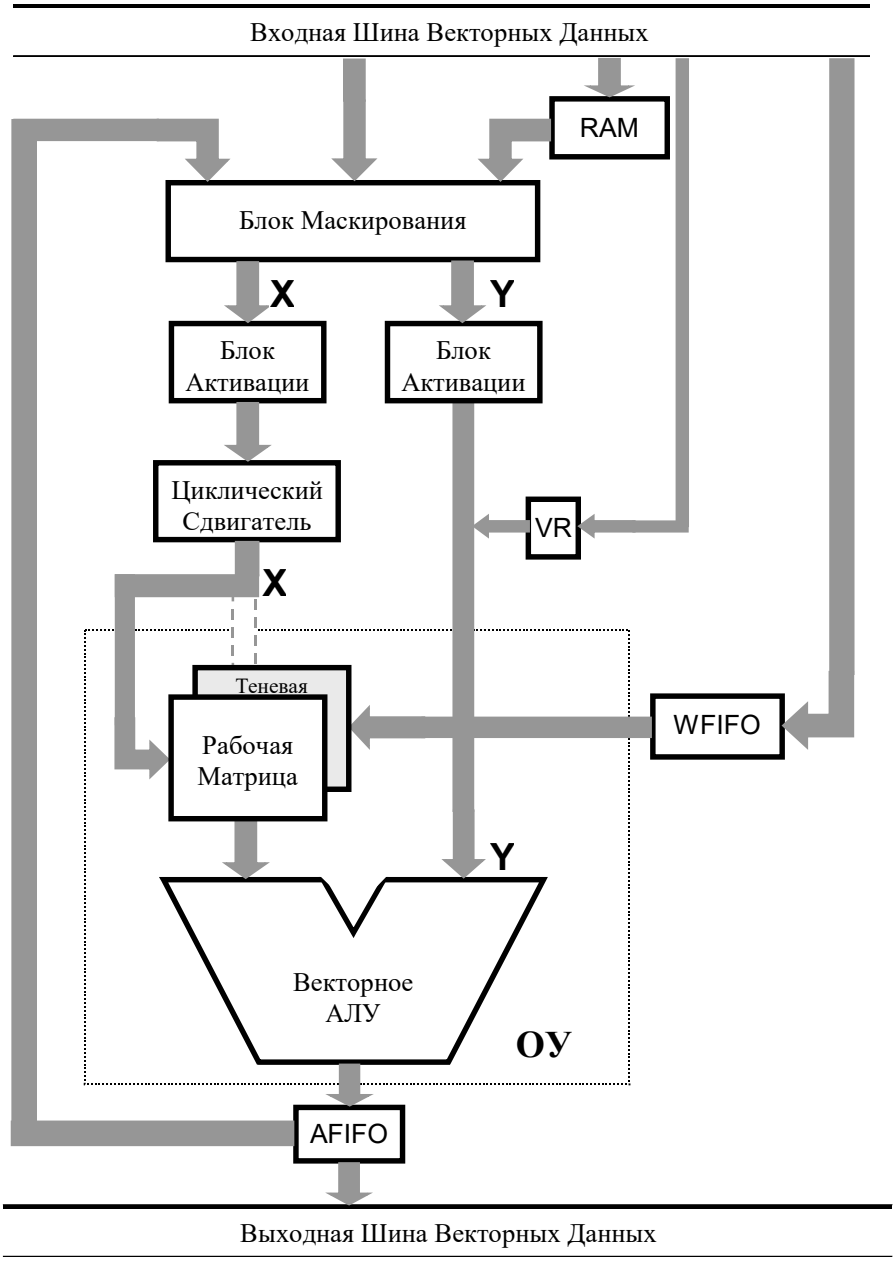
`set fp_branch;` (P_set) – разрешение параллельной работы для сопроцессора плавающей точки;

1.5 Векторный сопроцессор чисел с фиксированной точкой

1.5.1 Описание основных элементов векторного процессора

Архитектура векторного процессора фиксированной точки представлена на рисунке 1-4.

Рис. 1-4 Схема векторного процессора фиксированной точки



Входная Шина Векторных Данных, изображенная на Рис. 1-5, соответствует на Рис. 1-2 Входной шине #1 и Входной Шине #2, Выходная Шина Векторных Данных отражена как Выходная Шина.

Центральным узлом Векторного Процессора является операционное устройство (ОУ). Оно состоит из Рабочего/Теневого Матричного блока и Векторного АЛУ.

Операционное Устройство (ОУ) используется для выполнения операций умножения с накоплением, арифметических и логических операций, произвольной перестановки битов в векторах упакованных данных. Дополнительные вычислительные узлы Векторного Процессора позволяют выполнять операции

маскирования, осуществлять преобразования над векторами и матрицами кусочно-линейными пороговой функцией и функцией насыщения.

В состав ВП входят следующие компоненты:

- **Рабочая матрица** - операционный узел, в котором осуществляются операции умножения с накоплением. С рабочей матрицей связана пара регистров, которые определяют ее разбиение на столбцы и строки. Описание функционирования рабочей матрицы приведено в пункте 1.5.2.;
- **Теневая матрица** - устройство, используемое для ускорения закачки весовых коэффициентов в рабочую матрицу. В то время как рабочая матрица участвует в операции умножении с накоплением, в теневую матрицу может параллельно подкачиваться новая порция весовых коэффициентов. После того, как теневая матрица загружена, она в течение одного процессорного такта может быть перегружена в рабочую. С теневой матрицей связана своя пара регистров, определяющая ее разбиение на столбцы и строки. Это разбиение может быть отличным от того, которое использовалось в рабочей матрице на предыдущем этапе;
- **Векторное АЛУ** - устройство, позволяющее совершать стандартный набор арифметических и логических операций над парами 64-х разрядных слов, каждое из которых разделено на малоразрядные элементы. При арифметических операциях в случае переполнения внутри диапазона, отведенного под один малоразрядный элемент, блокируется перенос битов в соседний элемент. Более подробное описание работы векторного АЛУ приведено в пункте 1.5.3. Выполнение операций на векторном АЛУ:
- **Буфер весовых коэффициентов (**`wfifo`**)** - очередь глубиной в 32 64-х разрядных слова, организованная по принципу FIFO. В нее подгружаются весовые коэффициенты, и в ней они хранятся, прежде чем происходит их загрузка в теневую матрицу. Загрузка данных и их выгрузка из `wfifo` может осуществляться по частям, то есть, например, в `wfifo` можно загрузить сначала 8 слов, а затем еще 24, но так, чтобы не произошло переполнения. Более подробную информацию о правилах использования `wfifo` содержит пункт 3.4.8. Регистр-контейнер wfifo;
- **Буфер внутренней памяти** (`ram`)- очередь глубиной в 32 64-х разрядных слова, организованная по принципу FIFO. Используется, как один из аргументов в операциях умножения с накоплением, а также в операциях на векторном АЛУ. В `ram` может быть загружено от 1 до 32 слов. Буфер может использоваться многократно, однако в операциях должны

участвовать все данные, хранящиеся в `ram`. Не допускается использование только части хранящихся там данных. Более подробную информацию о правилах использования `ram` содержит пункт 3.4.7.Регистр-контейнер ram.

- **Псевдобуфер шины данных** (`data`) используется для обозначения данных, находящихся на шине данных непосредственно в процессе их загрузки из памяти в ВП. Позволяет обрабатывать данные на проходе. Псевдобуфер имеет глубину в 32 64-х разрядных слов и организован по принципу FIFO. Используется, как один из аргументов в операциях умножения с накоплением, а также в операциях на векторном АЛУ. Более подробную информацию о правилах использования `data` содержит пункт 3.4.6. Логический регистр-контейнер data;
- **Буфер накопления результатов** (`afifo`) - очередь глубиной в 32 64-х разрядных слова, организованная по принципу FIFO. Результат любой операции на ВП сохраняется в `afifo`. Может также использоваться, как один из аргументов в операциях умножения с накоплением и в операциях на векторном АЛУ. Для того, чтобы получить доступ к результатам вычислений на векторном процессоре, хранящимся в `afifo`, необходимо выгрузить их в память. Более подробную информацию о правилах использования `afifo` содержит пункт 3.4.5. Регистр-контейнер afifo;
- **Векторный регистр** (`vr`) - 64-х разрядный регистр, используемый в качестве определенного операнда в операции умножения с накоплением. Можно представить его как буфер, состоящий из заданного количество одинаковых слов. Может быть загружен из СП. Доступен только на запись. Более подробное описание использования `vr` дано в пункте 3.4.4. Регистр vr;
- Устройства, обеспечивающие выполнение **функции активации** над входными векторами. В ВП содержится два устройства, работающих независимо. Они позволяют активировать входные данные перед выполнением операции умножения с накоплением или перед подачей их на векторное АЛУ. Более подробное писание работы функций активации дано в пункте 3.4.1. Регистры f1cr и f2cr;
- Устройства, обеспечивающие выполнение **операции маскирования** над входными векторами. Более подробное писание выполнения операции маскирования дано в пункте 1.5.4. Операция маскирования;
- Устройство, **обеспечивающее циклический сдвиг вправо на один бит** слова данных, подаваемого на вход **X** рабочей матрицы в операции взвешенного суммирования. Более подробное

писание работы устройства циклического сдвига дано в пункте 1.5.6. Циклический сдвиг вправо операнда **X** при взвешенном суммировании.

1.5.2 Представление данных в векторном процессоре.

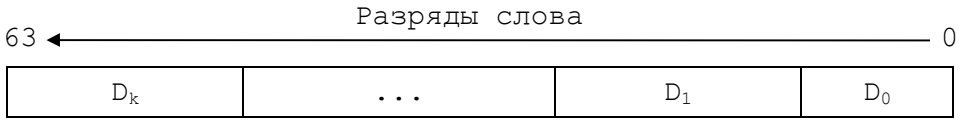
Под векторами понимаются одномерные массивы однородных данных, расположенные в памяти в виде непрерывного блока. Матрица - это массив векторов.

Разрядность всех узлов ВП составляет 64 бита. ВП осуществляет обработку целочисленных данных, которые упакованы в 64-х разрядные слова с помощью простой конкатенации (см. Рис. 1-5). В общем случае слово упакованных данных представляет собой вектор

$\mathbf{D} = \{D_k \ldots D_1\}$,

содержащий k элементов, суммарная разрядность которых составляет 64 бита. Причем в одном слове **D** могут быть упакованы данные, имеющие разную разрядность. Количество элементов k, упакованных в одном слове, зависит от их разрядностей и может принимать целочисленное значение в диапазоне от 1 до 64.

Рис. 1-5 Формат слова упакованных векторных данных.



Далее приводится описание работы основных блоков ВП, определяющих возможности процессора в выполнении следующих операций:

- умножение с накоплением, называемое также взвешенным суммированием;
- арифметические и логические операции на векторном АЛУ;
- маскирование данных;
- функции активации;
- сдвиг операнда **X** при выполнении взвешенного суммирования.

Помимо этого, приведен порядок выполнения преобразований над данными, если эти преобразования заданы в одной векторной команде.

1.5.3 Источники и пути данных.

Данные в процессор могут поступать из локальной внешней памяти посредством использования локальной шины данных и из глобальной внешней памяти через глобальную шину данных. В

состав Векторного Процессора входят три внутренних FIFO буфера `ram`, `afifo`, и `wfifo`, которые также принимают участие в обработке данных. Для управления поступлением данных непосредственно из внешней памяти, используется псевдобуфер `data`(см. пункт 3.4.6).

FIFO буфер весовых коэффициентов (`wfifo`) предназначен для хранения весовых коэффициентов для Рабочей Матрицы, пока другие буферы (`ram`, `data`, `afifo`) используются в качестве исходных буферов для обработки данных. Когда процессор начинает вычисления, `ram` и `afifo` пусты. Прежде чем использовать `ram`, необходимо загрузить в него данные. После этого данные могут использоваться многократно, до тех пор, пока не будут заменены новым блоком данных.

Результаты любых вычислений на Векторном Процессоре сохраняются в `afifo`. Они могут быть выгружены во внешнюю память или/и могут принимать участие в последующих шагах вычислений. Буфер накопления результатов представляет собой двухсторонний порт FIFO, он позволяет, например, выгружать данные во внешнюю память и возвращать новые данные из Векторного АЛУ одновременно.

Передача данных в Операционный Узел может осуществляться через два входа **X** и/или **Y**. Каждый из буферов (`data`, `ram` , `afifo`) может быть связан с одним или обоими каналами данных.

Например, вектор 64-х разрядных слов, хранящийся в `ram`, может быть передан на обработку в операционный узел через вход **X** и/или **Y**.

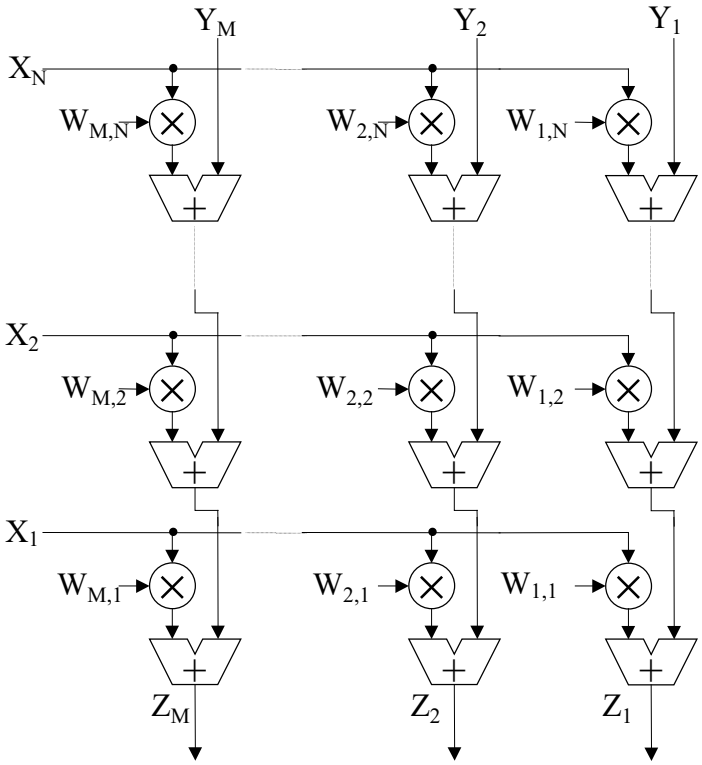
В качестве входа **Y** может также быть использован векторный регистр `vr`.(см. пункт 3.4.4.).

Кроме этого в качестве входов могут выступать так называемые "нулевые" устройства, что означает, что данные на вход не поступают.

1.5.4 Взвешенное суммирование

Операция умножения с накоплением выполняется рабочей матрице, входящей в состав операционного узла ВП. Схематично она представлена на Рис. 1-6.

Рис. 1-6 Схематичное представление операции взвешенного суммирования.



Математически операция взвешенного суммирования, выполняемая на операционном узле ВП, записывается следующим образом:

$$Z_i = Y_i + \sum_{j=1}^{N} X_j W_{ij}, (i = 1,...,M; j = 1,...,N),$$

где Z_I - элемент выходного вектора

X_j - элемент данных, поступающих на вход **X** операционного узла ВП.

Y_I - частичная сумма, накопленная на предыдущем шаге взвешенного суммирования.

W_{ij} - весовой коэффициент, расположенный в соответствующей ячейке рабочей матрицы процессора.

M - количество столбцов рабочей матрицы

N - количество строк рабочей матрицы.

Рабочая Матрица имеет вход, связанный с каналом **X** (см. Рис. 1-7). Он используется для загрузки 64-разрядных слов входных данных из внешней памяти (`data`) или из внутренних буферов FIFO (`ram` или/и `afifo`) в Рабочую Матрицу для выполнения операции

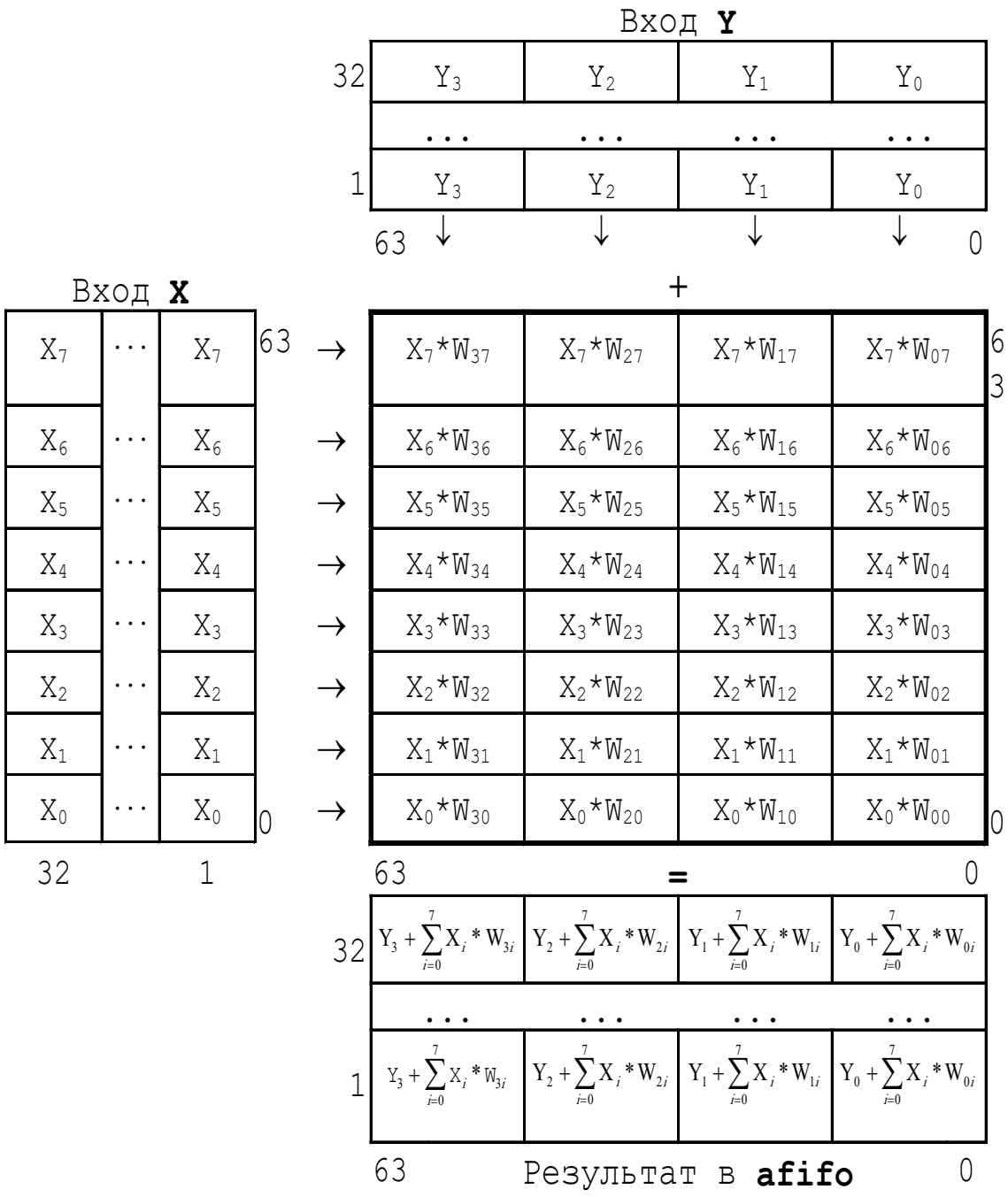
взвешенного суммирования (Σ). Результат вычислений передается в Векторное АЛУ для выполнения окончательного сложения с результатами предыдущих вычислений, поступивших по каналу **Y**(Y + Σ).

Входы **X** и **Y** имеют различное предназначение при вычислениях. Данные, поступающие на вход **X**, умножаются на ячейки Рабочей Матрицы, и суммируются в пределах столбца. Данные с входа **Y** в Векторном АЛУ добавляются к результату умножения со сложением, выполненного над данными входа **X**.

Перед началом вычислений в Рабочей Матрице, должна быть определена конфигурация Теневой/Рабочей матрицы, а также требуется предварительно загрузить весовые коэффициенты. Более подробно см. 3.4.8 Регистр-контейнер wfifo.

Разбиение матрицы на строки определяется регистром `sb2`. Этот же регистр определяет разбиение 64-х разрядных слов входных данных, поступающих на вход **X**. В `sb2` предварительно записывается слово, определяющее границы разбиения. Более подробное описание использования регистра `sb2` приведено в пункте 3.4.1. Регистры f1cr и f2cr.

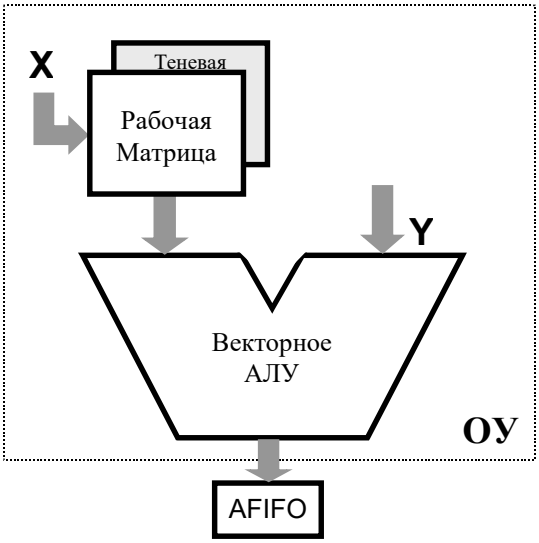
Рис. 1-7 Схема выполнения операции взвешенного суммирования на ВП.



Разбиение рабочей матрицы на столбцы задается регистром `nb2`. Он же определяет разбиение 64-х разрядных данных на входе **Y**, и разрядности результатов вычислений, содержащихся в буфере `afifo`. Более подробное описание использования регистра `nb2` приведено в пункте 3.4.2 Регистр nb1(nb2).

Рисунок Рис. 1-7 Схема выполнения операции взвешенного суммирования на ВП. показывает, какие действия может выполнить ВП при помощи одной процессорной инструкции. Допустим, в буфер данных `ram` было предварительно загружено из памяти 32 длинных слова. Другой источник данных находится во внешней памяти. При операции взвешенного суммирования из памяти по очереди подчитываются слова входных данных, каждое из которых направляется на вход **X** Рабочей Матрицы. Параллельно из буфера `ram` подчитывается очередное слово и направляется на вход **Y** Векторного АЛУ.

Рис. 1-8 Прохождение данных через Операционный Узел при выполнении операции взвешенного суммирования.



Каждый элемент, составляющий слово на входе **X**, умножается на весовой коэффициент, находящийся в соответствующей ячейке Рабочей Матрицы, результаты умножения складываются в пределах столбца. Вычисленные данные направляются на вход **X** Векторного АЛУ, а затем к ним добавляется значение элемента, находящегося на соответствующей позиции в слове, поступившем на вход **Y**. Все эти операции занимают один такт на одно упакованное слово данных. За следующий такт процессор выполняет те же операции над следующей парой входных слов данных и так далее до 32-х раз за

векторную инструкцию. Результат операции записывается в буфер `afifo`.

Данные, находящиеся в буферах FIFO векторного процессора, хранятся в 64-х разрядных словах. Для них на этом этапе разбиение на элементы **не определено**. Такое разбиение появляется только тогда, когда они поступают на вход **X** или **Y** Операционного Узла.

В зависимости от того, на какой вход, **X** или **Y**, поступают данные, они делятся на элементы тем или иным образом.

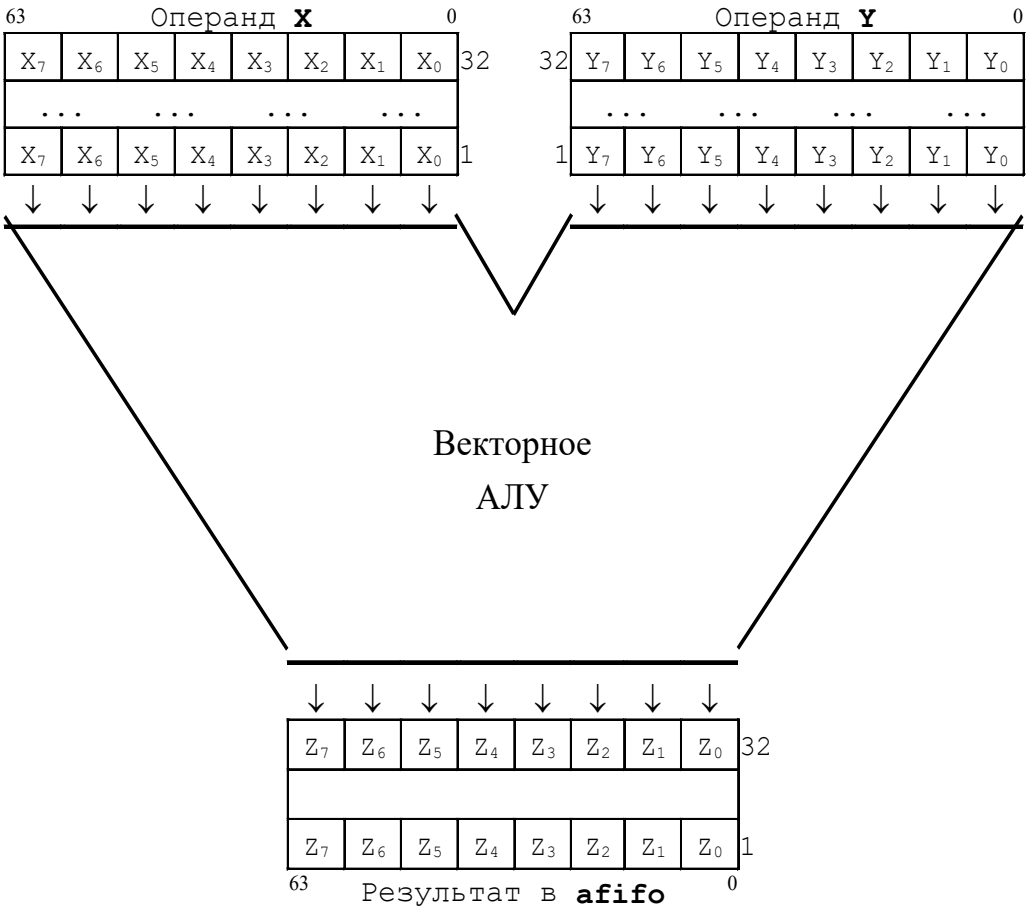
Например, если разбиение входа **X** - 8 битовое, то есть каждое слово представляется как 8 элементов по 8 битов, а разбиение **Y** - 16 битовое, то в зависимости от того, куда будут направлены данные, хранящиеся в `ram`, на вход **X** или **Y**, они будут трактоваться либо как массив 8-ми, либо 16-ти битовых элементов.

1.5.5 Выполнение операций на векторном АЛУ

Арифметические и логические операции на векторном АЛУ выполняются над наборами 64-х разрядных слов, подаваемых на входы **X** и **Y** операционного узла ВП. Эти наборы подаются на входы из буферов `ram`, `afifo`, либо из памяти (`data`). Данные, хранящиеся, например, в `ram`, могут быть переданы на обработку в операционное устройство как через вход **X**, так и через **Y (**Рис. 1-8**)**. Кроме этого в качестве входов могут выступать так называемые "нулевые" устройства, что означает, что данные на вход не поступают. Результат вычислений всегда попадает в `afifo`.

Входы **X** и **Y** векторного АЛУ являются равноправными. Разбиение данных, поступающих на эти входы, на **элементы определяется регистром `nb2`**. Регистр `sb2` не оказывает никакого влияния на вычисления в векторном АЛУ. В этом состоит особенность работы векторного АЛУ по сравнению с рабочей матрицей.

Рис. 1-9 Выполнение вычислений на векторном АЛУ.



Операции на векторном АЛУ выполняются с учетом разбиения входных данных на элементы (см. Рис. 1-9). Это означает, что в местах разбиения 64-х разрядных слов на элементы ставятся "перегородки", которые в случае переполнения блокируют перенос старшего бита в соседнее поле, занимаемое другим элементом, а также препятствуют распространению знака за пределы границ элемента. При блокировках переноса переносимый бит теряется.

На Рис. 1-10 приведен пример сложения двух 64-х разрядных слов, разбитых на 8 элементов по 8 битов каждый, на векторном АЛУ фиксированной точки.

Рис. 1-10 Сложение двух 64-х разрядных слов на векторном АЛУ.

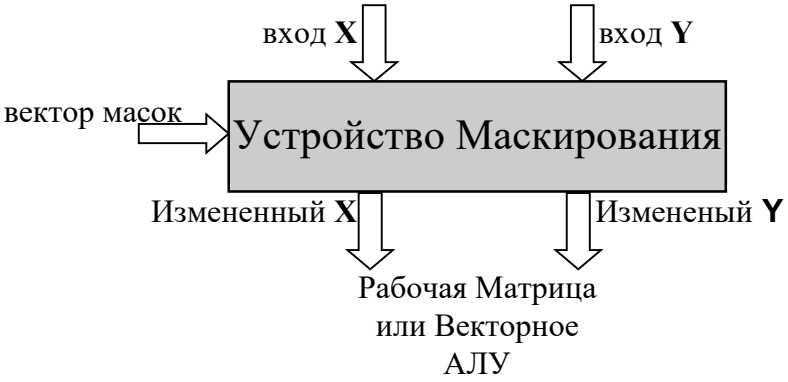
01	80	80	F0	02	FF	FF	01
+							
FE	20	80	1F	02	01	FF	01
=							
FF	A0	00	0F	04	00	FE	02

Серым цветом на рисунке отображены ячейки, где "перегородки" повлияли на результат вычислений.

1.5.6 Операция маскирования

Для выполнения маскирования в операционном узле ВП существует специальное устройство. Оно имеет три входа и два выхода (см. Рис. 1-11). Данные, подаваемые на входы **X** и **Y** рабочей матрицы или векторного АЛУ, сначала проходят через это устройство, и только после этого попадают в рабочую матрицу или на векторное АЛУ. Если код векторной команды не включает операцию маскирования, то данные, поступающие на входы **X** и **Y** устройства, пропускаются на выход без изменений. Если в коде команды присутствует запрос на маскирование, то используется также третий вход в устройство, на который подается вектор масок. Он может подчитываться из памяти (`data`), либо из буферов `ram` или `afifo`.

Рис. 1-11 Входные и выходные потоки данных в Устройстве Маскирования.



Векторные команды маскирования, как и все остальные векторные команды процессора, выполняются от 1 до 32 тактов. На каждом такте на входы X, Y и на вход вектора масок подаётся по одному 64-х разрядному слову.

Данные, прошедшие через Устройство Маскирования, подаются на Рабочую Матрицу и/или Векторное АЛУ для дальнейшей обработки.

Маскирование с Векторным Умножением

В системе команд процессора существует векторная команда, например:

маска **X** **Y**

```
rep 32 data = [ar0++] with vsum ram, data, afifo;
```

которая совмещает выполнение операции маскирования с обработкой результатов на рабочей матрице. В этом случае над данными выполняются следующие преобразования:

- побитовая операция AND слова с входа **X** и слова маски: `X and MASK`. Эта операция оставляет в результирующем векторе **X** только те биты, которые в маске были равны единице;
- побитовая операция AND слова с входа **Y** и инвертированного слова маски: `Y and not MASK`. Эта операция оставляет в результирующем векторе **Y** только те биты, которые в маске были равны нулю;
- выполнение операции взвешенного суммирования над маскированными данными, подаваемыми на вход **X** (Σ);
- сложение маскированного вектора **Y** с результатом взвешенного суммирования (Y+Σ).

Логическое Маскирование

В случае совмещения операции маскирования с обработкой на векторном АЛУ, задаваемой командой:

маска **X** **Y**

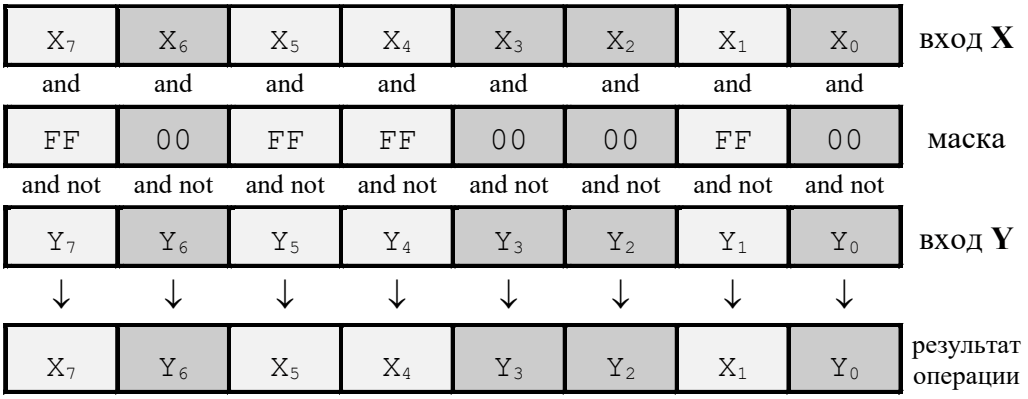
```
rep 32 data = [ar0++] with mask ram, data, afifo;
```

над данными на каждом шаге выполняется следующее преобразование:

```
(X and MASK) or (Y and not MASK)
```

Сложно записанная формула вероятно скрывает простоту и пользу данного преобразования. На Рис. 1-12 дается пояснение в графической форме:

Рис. 1-12 Выполнение операции маскирования.



В тех позициях, на которых в маске стоят единицы, в слово результата будут записаны биты слова с входа **X**, на остальные места попадут биты слова с входа **Y**. Таким образом, операция маскирования позволяет за один такт из двух длинных слов собрать одно, взяв нужные биты из одного слова и из другого.

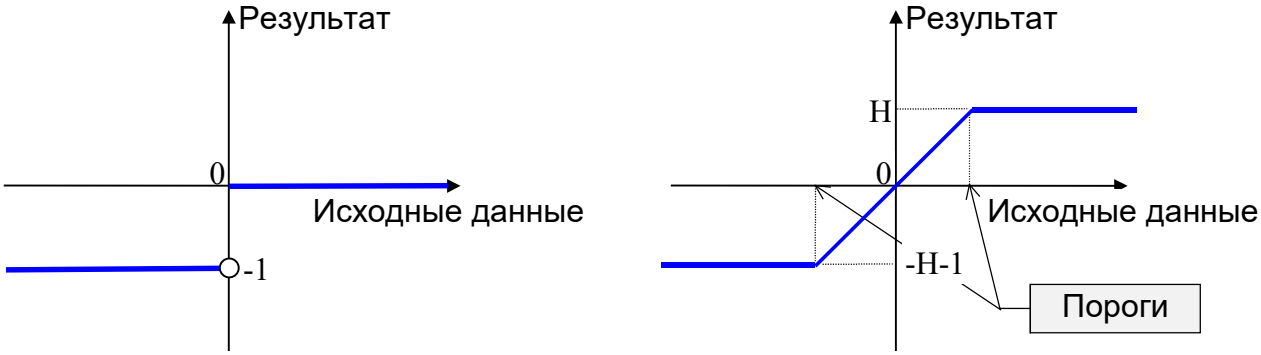
1.5.7 Обработка данных функцией активации

Векторный Узел содержит два блока активации для обработки пороговой функцией и функцией насыщения входных данных. Один из них связан с входом **X**, другой с входом **Y**.

Всего существует два типа функций активации:

- пороговая функция (см. Рис. 1-13а);
- функция насыщения (см. Рис. 1-13б).

Рис. 1-13. Типы встроенных функций активации процессора NM6403



a) Пороговая функция *b) Функция насыщения*

В блоках активации осуществляются вычисления над упакованными словами данных. Блоки активации позволяют применять функции активации ко всем элементам упакованного слова одновременно. Основную роль в управлении функциями активации играют регистры `f1cr` и `f2cr`. Более подробную информацию об их роли в управлении обработкой потоков данных см. пункт 3.4.1 Регистры f1cr и f2cr.

Блоки активации размещаются между устройством маскирования и Рабочей Матрицей или Векторным АЛУ. Функции активации могут быть подвергнуты либо данные, поступающие на вход **X**, либо на **Y**, либо на оба входа сразу.

Выбор типа активации зависит от того, в какой команде эта активация задается. Существуют два типа векторных инструкций: арифметические и логические. Термины “арифметическая векторная инструкция ” и ”логическая векторная инструкция” описаны ниже в этом подпункте.

Пороговая функция может быть применена к упакованным словам данных, только если они обрабатываются логической векторной инструкцией. Функция насыщения может использоваться только в паре с арифметической векторной инструкцией. Таким образом, пороговая функция называется также “логической

активацией", тогда как функция насыщения указана как "арифметическая активация".

Арифметическая Активация

Функция насыщения может быть использована только в составе арифметической векторной инструкции. Термин "арифметическая векторная инструкция" включает в себя следующий список векторных инструкций:

- Все типы инструкций, содержащие операцию взвешенного суммирования. Например:

  вход **X** вход **Y**

  ```
  rep 12 with vsum , activate ram, afifo;
  ```

  Зарезервированное слово "activate" указывает, что функция насыщения применяется к данным, поступающим на вход **X** Рабочей Матрицы;

- Все типы инструкций, содержащие арифметические операции. Например:

  вход **X** вход **Y**

  ```
  rep 32 data =[ar0++] with data + activate ram;
  ```

  В этом случае термин "activate" означает применение функции насыщения к данным, поступающим на вход **Y** Векторного АЛУ.

Все векторные инструкции, указанные выше, имеют некоторые характерные черты. Все они содержат арифметические операции. Итак, функция насыщения называется "арифметической активацией", так как используется вместе с арифметическими инструкциями.

Логическая Активация

Пороговая функция, в отличие от функции насыщения, используется только совместно с логическими векторными инструкциями. Термин "логическая векторная инструкция" связан с множеством векторных инструкций, которые содержат только логические операции, выполняемые на Векторном АЛУ. Например:

вход **X** вход **Y**

```
rep 32 data = [ar0++] with data or activate ram;
```

В этом случае термин "активация" означает, что пороговая функция применяется к данным, поступающим на вход **Y** Векторного АЛУ.

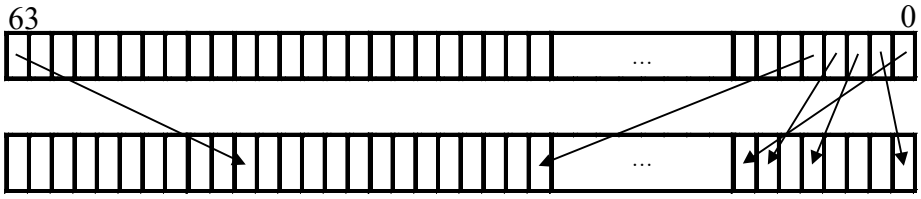
Пороговая функция используется вместе с логическими векторными инструкциями, поэтому называется также "логическая активация".

Более подробная информация о выполнении активации приведены в подпункте 3.4.1 Регистры f1cr и f2cr.

1.5.8 Циклический сдвиг вправо операнда **X** при взвешенном суммировании

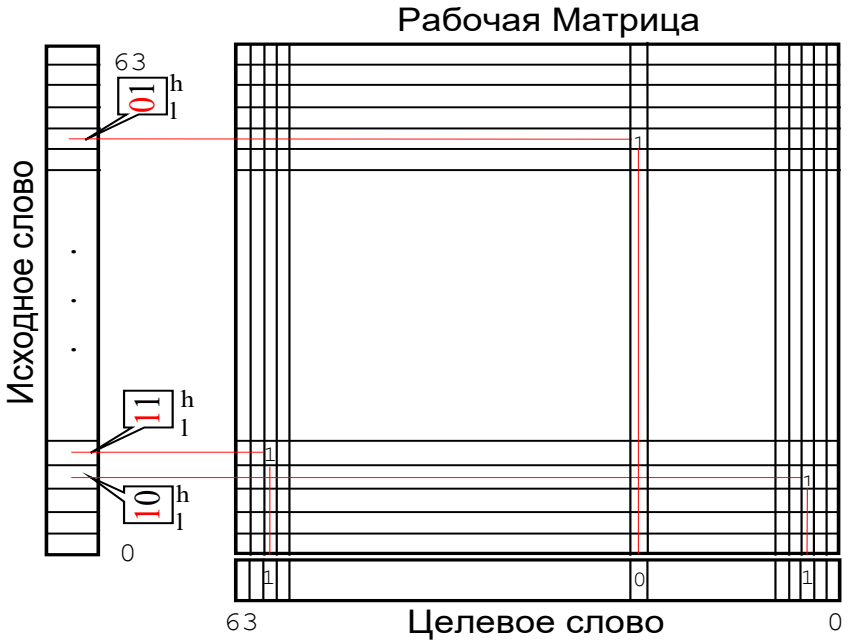
Данные, поступающие на вход **X** при операции взвешенного суммирования или маскирования, могут быть подвергнуты циклическому смещению на один бит вправо (см. Рис. 1-4). Это смещение необходимо для того, чтобы компенсировать неудобства, которые возникают из-за невозможности разбиения данных, подаваемых на вход **X** рабочей матрицы, на элементы нечетной разрядности, в частности на элементы разрядности 1 бит. Минимально возможная разрядность элементов 2 бита. Так как регистр `sb2` не позволяет задать побитовое разбиение данных на входе **X**, необходим дополнительный преобразователь для того, чтобы обеспечить обработку старших битов при двухбитовом разбиении. Это преобразование обеспечивает циклический сдвиг вправо на один бит. При этом разбиение на элементы не учитывается, то есть сдвигается целиком все слово. В этом случае старшие биты двухбитовых слов становятся младшими и могут теперь принимать участие в вычислениях. Младшие биты становятся старшими в соседнем элементе справа, но они могут быть маскированы в той же векторной инструкции. Рассмотрим следующий пример:

Нам необходимо произвести полную перестановку внутри 64-разрядного слова



Рабочая Матрица разбита на 32 строки и 64 столбца. Заполнив отдельные ячейки Рабочей Матрицы 0 или 1, мы можем осуществить перестановку каждого младшего бита 2-х разрядных элементов входного слова для получения необходимых позиций в выходном слове.

Рис. 1-14. Побитовая перестановка на Рабочей Матрице



Для перестановки старших битов элементов исходного слова мы должны сдвинуть их на один бит вправо. В этом случае старшие биты становятся младшими и могут быть обработаны тем же способом, как показано на Рис. 1-14.

Циклический сдвиг может быть применен к исходным данным в векторной инструкции вместе с операцией взвешенного суммирования или маскированием. Активизируется устройство циклического сдвига при использовании в векторной команде ключевого слова “shift”.

Например,

Вход **X** вход **Y**

```
rep 12 data = [ar0++] with vsum , shift data, 0;
```

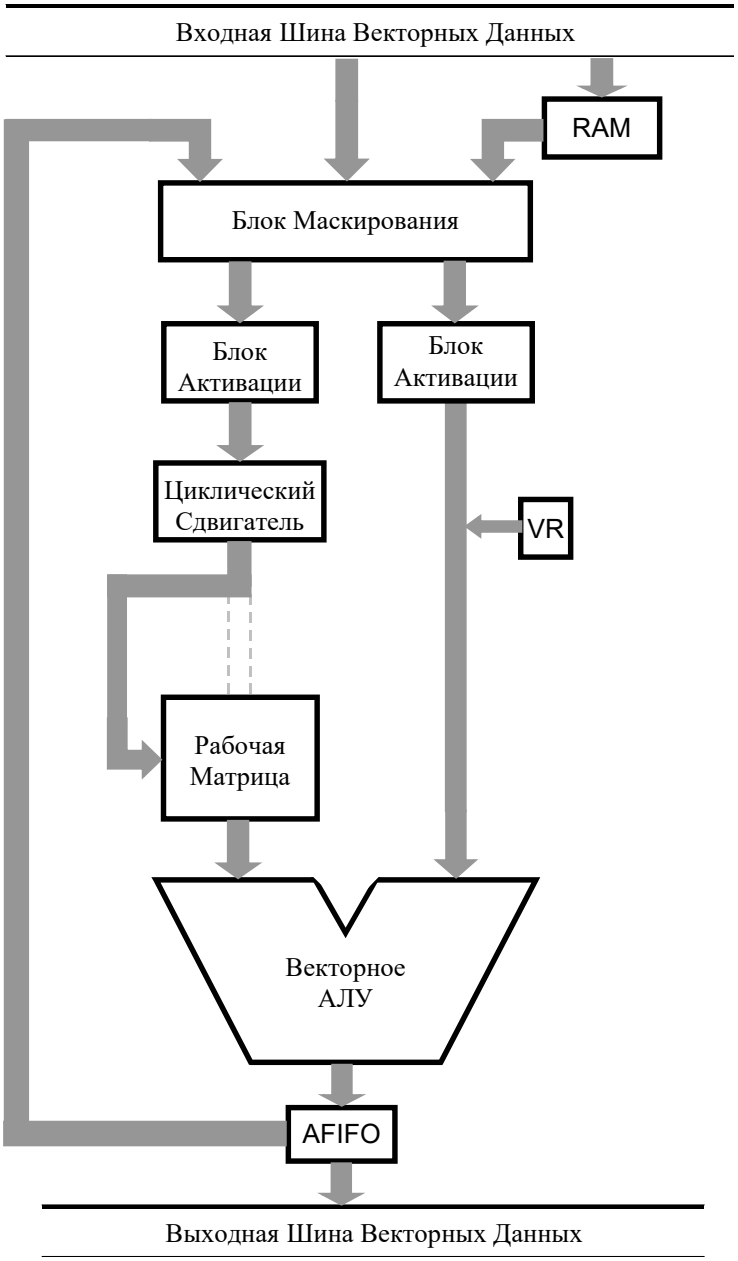
Для ядра nmc3 сдвиг не совместим с активацией.

Более подробно см. подраздел 5.2 Векторные инструкции.

1.5.9 Порядок выполнения преобразований над данными на ВП

В векторном блоке расположены шесть вычислительных узлов. Многие из них могут быть включены в вычисления одной векторной инструкцией, в этом случае преобразования выполняются в определенном порядке. На Рис. 1-15. Порядок выполнения преобразований на Векторном Процессоре показана последовательность преобразований, которым подвергаются данные в процессе вычислений на Векторном Процессоре.

Рис. 1-15. Порядок выполнения преобразований на Векторном Процессоре



Каждый блок обработки может пропускать через себя данные без изменений, либо модифицировать их. Это определяется ассемблерной инструкцией. Активизируются определенные вычислительные узлы с помощью использования в инструкции специальных ключевых слов.

Например, “shift” используется для активизации устройства циклического сдвига, “mask” используется для активизации операции маскирования, и так далее. В одной инструкции можно активизировать до шести узлов, однако, выполнение

преобразований над данными будет производиться в порядке, показанном на Рис. 1-15.

1.6 Отличия структуры процессора NM6405 от процессора NM6403

- По системе команд процессоры совместимы снизу вверх, т.е. все команды ассемблера процессора NM6403 имеются на NM6405 и должны работать в точности также, за считанными исключениями. В то же время, изменился набор и функции управляющих регистров.
- Периферийные устройства вынесены за пределы ядра процессора.
    - Наличие коммуникационных портов и регистры управления ими могут варьироваться в разных вариантах комплектации ядра. Подробнее см. описание процессора NM6405.
    - Могут присутствовать несколько DMA-каналов для обмена с внешней памятью без участия процессора. Описание регистров управления см. в описании процессора NM6405.
    - Наличие таймеров и регистры управления ими могут варьироваться в разных вариантах комплектации ядра. Подробнее см. описание процессора NM6405.
    - Формат регистров управления внешней памятью см. описание процессора NM6405.

2 Обзор Основных Элементов Языка Ассемблера

Программы, написанные на языке ассемблера для NM6403, состоят из различных синтаксических конструкций, которые могут включать в себя директивы ассемблера, инструкции, макросы, псевдокоманды, комментарии. Длина строки синтаксической конструкции ограничена только требованиями текстового файла и удобством просмотра в различных текстовых редакторах.

Следующие строки демонстрируют несколько примеров корректных синтаксических конструкций:

```
MySym: word = 80h;          // Определение переменной.
<L1>                        // Определение метки
     ar0 = ar2 + gr2;       // Модификация адресного регистра
rep 32 [ar0++] = afifo;     // Векторная команда
```

Синтаксическая конструкция не содержит жестких требований к структуре строки, то есть нет предопределенных позиций, в которых должны располагаться те или иные поля команд или директив.

Каждая ассемблерная инструкция должна заканчиваться знаком ‘;’. Если на текущей строке ассемблер не находит знак окончания инструкции, он полагает, что инструкция продолжается на следующей строке.

Общий вид синтаксической конструкции, соответствующей ассемблерным инструкциям, может выглядеть следующим образом:

[<метка>] ассемблерная инструкция; [//комментарии]

Метки и комментарии могут быть опущены.

С-подобные многострочные (/**/) комментарии в языке также допускаются.

2.1 Служебные слова и идентификаторы

В расположенной ниже таблице приведены зарезервированные слова, которые нельзя использовать как имена меток, переменных и т.д. В описании использованы регулярные выражения, то есть «ar[0-7]» означает, что зарезервированы слова ar0, ar1, … ar7, а fp_.* означает, что зарезервированы слова, начинающиеся на «fp_». Резервирование осуществляется всегда, в том смысле, что к примеру имя специфического регистра будет резервироваться даже при трансляции для устройства, у которого этот регистр отсутствует.

Табл. 2-1 Основные служебные слова языка ассемблера.

.align	.branch	.debug_.*	.endif	.fixed_32	.fixed_64
.in_high	.in_low	.nm6403	.nm6405	.nm64revision	.nm64revision2
.packer	.soc	.wait	activate	addr	afifo
and	ar[0-7]	begin	block	call	callrel
carry	cfalse	clear	code	common	const
ctrue	data	delayed	dir0	dir1	dmac0
dmac1	dor0	dor1	dup	end	extern
f1cr	f1crh	f1crl	f2cr	f2crh	f2crl
false	flag	fp[0-3]"_".*	fp_.*	fpu	from
ftw	gema	gima	global	gmcr0	gmcr1
gmicr	goto	gpa	gr[0-7]	halt	hiword
ica0	ica1	icc0	icc1	if	import
imr	intr	intr_mask	intr_nmc3	intr_req	iop
iopcr	ireturn	irr	label	lema	lima
lmcr0	lmcr1	lmicr	local	locdesc	long
loword	macro	mask	nb1	nb1h	nb1l
nmcscr	nmscu	nobits	noflags	nop	not
nul	oca0	oca1	occ0	occ1	offset
or	own	pc	pcr	pcr_6405	pop
pr[0-9]	pr1[0-5]	pswr	pswr_nmc3	punit[0-9]	push
ram	ref	rep	return	sb	sbh
sbl	sconst	set	shift	sir	sizeof
skip	sp	store	string	struct	t0
t1	tmr_count0	tmr_count1	tmr_mode0	tmr_mode1	true
uconst	var	vfalse	vlen	vnul	vr
vreg[0-7]	vregs	vrh	vrl	vsum	vtrue
weak	wfifo	with	word	wtw	xor
zero					

2.1.1 Идентификаторы

Идентификаторы используются для обозначения программных объектов, таких как метки, переменные, константы и макросы.

Лексически идентификатором может быть любая последовательность символов, удовлетворяющая следующим условиям:

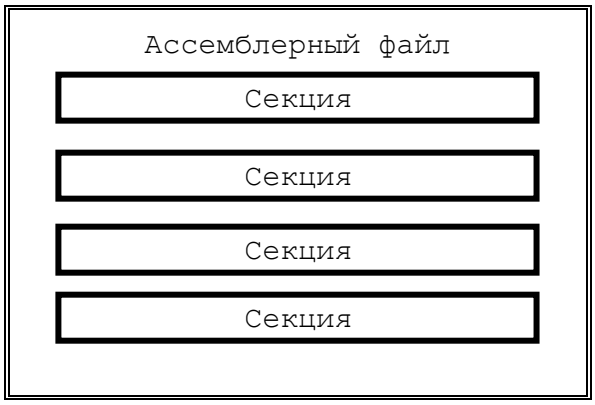
- первый символ последовательности – латинская буква или символ подчеркивания «_»;
- остальные символы последовательности – латинские буквы, цифры, символы подчеркивания «_» и точки «.»;
- последовательность не совпадает ни с одним служебным словом

Идентификаторы, как и служебные слова, чувствительны к регистру.

2.2 Структура ассемблерного Файла

Файл исходного текста программы, разрабатываемой на языке ассемблера, имеет определенную структуру, приведенную на Рис. 2-1:

Рис. 2-1 Структура ассемблерного файла.



Условно пространство ассемблерного файла можно разбить на подпространство секций и подпространство между секциями.

2.3 Секции

Секции в языке ассемблера бывают трех типов:

- секции кода;
- секции инициализированных данных;
- секции неинициализированных данных.

Секции всех типов, встречаемых в языке, подчиняются одинаковым правилам оформления, которые будут перечислены ниже.

Правила оформления секций

Секция начинается со служебного слова: `begin`, `data` или `nobits`, являющегося открывающей скобкой и заканчивается словом `end` (закрывающая скобка). По открывающей скобке ассемблер определяет, каков тип данной секции. Информация о соответствии ключевых слов открывающей скобки типам секций будет приведена ниже в этом пункте. Сразу за открывающей/закрывающей скобкой в той же строке должно располагаться имя секции. Например, секция кода `mysect` должна быть оформлена следующим образом:

```
begin mysect // начало секции mysect

    тело секции

end mysect;   // конец секции mysect
```

После открывающей скобки **точка с запятой не ставится**, после закрывающей **ставится обязательно**. Имена секции при открывающей и закрывающей скобках должны совпадать.

Имя секции может быть произвольной длины в пределах 255 символов. Оно представляет собой набор заглавных и прописных латинских букв, цифр и знака ‘_’. Имя секции не должно начинаться с цифры, только с буквы или ‘_’.

Имя секции **может** быть заключено в двойные кавычки, либо использоваться без кавычек. Если оно не заключено в кавычки, оно не должно совпадать ни с одним из служебных слов языка ассемблера. При этом, к названию секции, не заключенной в кавычки, при записи ее в объектный файл спереди добавляется точка, то есть, например, секция `mysect`, сохраненная в объектном файле, получает новое имя `.mysect`.

Если имя секции заключено в двойные кавычки, например: `begin "mysect"`, то в этом случае это имя попадет в объектный файл без изменения. Имя секции, заключенной в кавычки может совпадать со служебным словом.

2.3.1 Секции кода

В секциях данного типа содержатся последовательности инструкций, определяющие порядок выполнения программы.

Секция кода начинается с открывающей скобки `begin`.

Для удобства дальнейшей работы с секциями кода рекомендуется формировать имя секции путем прибавления к придуманному имени префикса `text`, например: `begin textmycode`. Наличие данного префикса является сигналом для декодера объектных и исполняемых файлов (`dump.exe`)

осуществлять автоматическое декодирование секции, как секции кода.

В секции кода возможно помещать не только коды инструкций, но и строки объявления переменных. Если переменная инициализирована, то есть ей присвоено начальное значение, то оно располагается в секции по соответствующему адресу. Если переменная не инициализирована, то в секции выделяется место, которое заполняется нулевым значением.

Например, секция кода может быть оформлена следующим образом:

```
begin ".textmycode"
// блок объявления переменных.
    A: long = 0123456789ABCDEFhl;
    B: word;
// блок инструкций процессора.
<Label>
    ar0 = A;
    ar2,gr2 = [ar0];
end ".textmycode";
```

Возможность объявления переменных в секции кода введена скорее как удобная возможность, нежели как постоянно используемый прием. Для объявления инициализированных и неинициализированных переменных существуют свои типы секций, работать с которыми более удобно, потому что можно управлять их положением в памяти процессора.

2.3.2 Секции инициализированных данных

В секциях данного типа содержатся объявления и инициализация переменных, используемых программой.

Секция инициализированных данных начинается с открывающей скобки `data`.

Пример секции инициализированных данных:

```
data ".my_init_data"
    Val1: word = 12;
    Val2: word = 0A5h;
    Arr:  long[4] = ( 0FF00FF00FF00FF00hl dup 4 );
end ".my_init_data";
```

Ассемблер **позволяет** объявлять неинициализированные переменные в секции инициализированных данных, например, не вызовет ошибки следующая конструкция:

```
data ".my_init_data"
    Val1: long = 12;
    Arr:  long[4]; // неинициализированная переменная.
    Val2: word = 0A5h;
end ".my_init_data";
```

Однако в процессе ассемблирования разделяются инициализированные и неинициализированные данные. Для неинициализированных переменных создается дополнительная секция, куда помещаются все обнаруженные переменные. Существует правило выбора имени секции неинициализированных данных, автоматически порождаемой ассемблером. К имени соответствующей секции инициализированных данных добавляется префикс ".bss", например, для секции ".my_init_data" будет создана дополнительно секция ".bss.my_init_data".

Таким образом, если в секцию инициализированных данных попадают неинициализированные переменные, ассемблер во время компиляции создает дополнительную секцию неинициализированных данных, куда помещает все найденные неинициализированные переменные, что эквивалентно следующей конструкции:

```
data ".my_init_data"
    Val1: long = 12;
    Val2: word = 0A5h;
end ".my_init_data";


nobits ".bss.my_init_data"
    Arr:  long[4]; //неинициализированная переменная.
end ".bss.my_init_data";
```

Если в секции инициализированных данных объявлена структура, поля которой частично инициализированы, то она **не разрывается** на инициализированные и неинициализированные части. Неинициализированные поля заполняются нулями. Если все поля не инициализированы, то структура попадает в соответствующую секцию неинициализированных данных. Для того, чтобы структура осталась на месте, достаточно инициализировать хотя бы одно ее поле.

Примечание

Если имя секции инициализированных данных не заключено в кавычки, то в процессе компиляции при создании соответствующей ей секции неинициализированных данных, имя последней будет порождено путем добавления префикса ".bss.". К примеру, по секции AAAA из исходного текста программы, в объектном файле будут созданы секции .AAAA и .bss.AAAA.

2.3.3 Секции неинициализированных данных

В секциях данного типа содержатся только объявления переменных, используемых программой, без их инициализации.

Секция неинициализированных данных начинается с открывающей скобки nobits.

Пример секции неинициализированных данных:

```
nobits ".my_bss_data"
```

```
    Val1: word;
    Val2: word;
    Arr:  long[4];
end ".my_bss_data";
```

Если в секцию неинициализированных данных попадает инициализированная переменная, то ассемблер игнорирует ее инициализацию, рассматривая переменную, как неинициализированную.

2.3.4 Пространство между секциями

Пространство между секциями в ассемблерном файле также может использоваться для размещения следующих синтаксических конструкций:

- задания констант и константных выражений, включая использование псевдофункций;
- объявления меток всех возможных типов;
- объявления переменных типа `common`, `extern` и `local`;
- описания структурных типов данных;

2.4 Константы

В данном разделе будет приведена информация о том, как в языке ассемблера определяются константы и константные выражения, а также то, в каком формате они могут быть представлены.

Целочисленные константы могут быть 32-х или 64-х разрядными. По умолчанию константа считается 32-х разрядной (короткая константа). Для записи 64-х разрядных целых констант, на которые в дальнейшем будем ссылаться, как на длинные, используется знак **l(L)**, добавляемый в конец. Например, константа `0x123h` является 32-х разрядной, а `123hl` 64-х разрядной. Целочисленным константам можно назначать символьные имена, то есть, создавать именованные константы(п.2.4.3): `const A = 0x123h`.

Константы с плавающей точкой также бывают 32-х и 64-х разрядными. Для их представления в пользовательской программе на языке ассемблера используются специальные псевдофункции периода компиляции: `float`() и `double`().Более подробную информацию см. ниже в этом разделе.

2.4.1 Форматы представления констант

Ассемблер поддерживает шесть форматов представления констант:

- Двоичные целые константы;
- Восьмеричные целые константы;
- Десятичные целые константы;
- Шестнадцатеричные целые константы;

В теле вышеперечисленных констант можно использовать символ подчеркивания (_) для визуальной группировки разрядов. От таких вставок интерпретация констант не изменяется

- Константы с плавающей точкой;
- Символьные константы.

2.4.1.1 Двоичные целые константы

Двоичные целые константы представляют собой строку нулей и единиц длиной до 64 элементов. За двоичной константой следует буква **b(B)**, которая соответствует данному формату числа в представлении ассемблера. Если количество элементов в строке двоичной константы меньше 32-х или 64-х, неопределенные биты слева (старшая часть слова) автоматически заполняются нулями при ассемблировании. Примеры правильно записанных двоичных констант:

```
00000000b          //    Константа, равная нулю.
001b1              //    Длинная константа, равная 1.
10000B             //    Константа, равная 16₁₀.
10101010b          //    Константа, равная 170₁₀ или AA₁₆.
10001000_10001000b //    Группировка разрядов
```

2.4.1.2 Восьмеричные целые константы

Восьмеричные целые константы представляют собой строки, состоящие из цифр от `0` до `7`, в конце которых следует символ **o(O)**. Примеры правильно записанных восьмеричных констант:

```
000o               //    Константа, равная нулю.
001o               //    Константа, равная единице.
20OL               //    Длинная константа, равная 16₁₀.
252o               //    Константа, равная 170₁₀ или AA₁₆.
7777_00_00o1       //    Группировка разрядов
```

2.4.1.3 Десятичные целые константы

Десятичные целые константы представляют собой строки, состоящие из цифр от 0 до 9. Диапазон изменения составляет от -2.147.483.648 до 2.147.483.647 для знаковых и от 0 до 4.294.967.295 для беззнаковых коротких констант, и от -9.223.372.036.854.775.808 до 9.223.372.036.854.775.807 для знаковых и от 0 до 18.446.744.073.709.551.615 для беззнаковых длинных констант. Примеры правильно записанных десятичных констант:

```
7000               //    Константа, равная 7000₁₀.
-1                 //    Константа, равная минус единице.
```

```
-20l            //      Длинная константа, равная -20.
170             //      Константа, равная 17010 или AA16.
1_234_567_890l  //      Группировка разрядов
```

Знак "-" может использоваться для обозначения отрицательных чисел только в десятеричном формате.

2.4.1.4 Шестнадцатеричные целые константы

Шестнадцатеричные целые константы представляют собой строки, состоящие из цифр от `0` до `9`, а также букв `A`, `B`, `C`, `D`, `E` и `F`, заглавных или строчных в конце которых следует символ **h(H)**. Запись константы должна начинаться с цифры. Если первым значимым символом является буква, то перед ней необходимо поставить цифру `0`. Символ "`0`" в начале шестнадцатеричной константы ставится с целью отличить ее от идентификатора. Примеры правильно записанных шестнадцатеричных констант:

```
000h            //      Константа, равная нулю.
01H             //      Константа, равная единице.
20hl            //      Длинная константа, равная 3210.
0ffffffffh      //      Константа, равная -1.
0FFFFFFFF_FFFFFFFFhl        //      Группировка разрядов
```

2.4.1.5 Константы с плавающей точкой

Операции над данным типом констант не поддерживаются аппаратно. Для их записи в языке ассемблера используются специальные псевдофункции. Для 32-х разрядных чисел типа `float` используется псевдофункция `float()`. Для 64-х разрядных чисел типа double используется псевдофункция `double()`. Внутри скобок псевдофункции может располагаться вещественное число в привычной форме, например:

```
float(123.456)      // Число с плавающей точкой (32 бита).
double(-1.02E-3)    // Отрицательное число.
```

Формат вещественного числа:

`[+|-]num[[.num]E[+|-]num]`, *где* `num` *- десятичные числа.*

Или

```
[+|-]num.num
```

Библиотеки функций, эмулирующие работу с плавающей точкой на процессоре NM6403, используют ее представление в формате IEEE 754.

2.4.1.6 Строковые константы

Строковая константа представляет собой произвольную (возможно, пустую) последовательность символов ASCII, заключенную в двойные либо одинарные кавычки. Если в тексте строки необходимо задать символ кавычки, то он должен быть удвоен.

Примеры строковых констант:

"Строковая константа",
'Упакованная строковая константа'

Строковая константа занимает различное число слов, в зависимости от символа ограничителя. Строка, заключённая в двойные кавычки, в памяти располагается в соответствии с соглашением, принятым в языке Си++ – по минимально адресуемому элементу памяти (32-х разрядное слово в процессоре NM6403) на каждый символ. Таким образом, возможен произвольный доступ к символам строки.

В случае использования символа-ограничителя “одинарная кавычка” символы “упаковываются” по четыре в короткое слово. Упакованная строковая константа занимает в памяти количество слов, достаточное для хранения всех символов константы.

Строковая константа не содержит никакой иной информации (специального завершающего символа или значения длины строки), кроме той, которая явно указана внутри кавычек.

Допускается использование эскейп-кодов в следующей форме:

"Строковая\032константа"

Что эквивалентно первой строке из предыдущего примера.

Чтобы напечатать обратный слэш (\) в теле строки его следует удвоить, то же самое касается символа ограничителя строки (″ если строка неупакованная или′ в противном случае).

Допускается также конкатенация строк в стиле C:
"Строковая"
" константа"

2.4.2 Константные выражения

Помимо использования констант в языке ассемблера для NM6403 допускается использование константных выражений, которые могут быть вычислены на этапе компиляции программы, так что в объектный код попадает только результат вычислений.

Константные выражения разделяются на числовые и адресные. В первом случае результат вычисления константного выражения трактуется как обычная числовая константа, во втором, как адрес в памяти.

2.4.2.1 Числовые константные выражения

Числовое константное выражение строится по правилам, традиционным для построения выражений. Тип констант – 32 или 64-х битные знаковые числа. В языке имеется набор из обычных арифметических операций, операций побитового сложения,

умножения, сдвига и логических операций сравнения. Интерпретация константных выражений, в частности, приоритет операций, в целом соответствует правилам языка C++, но вместо связок “&&”, “&”, “||”, “|”, “^”, “!” следует использовать “and”, “or”, “xor”, “not”. Операции сравнения возвращают 1 (true) или 0 (false). Для частичного изменения порядка выполнения операций используются круглые скобки.

Табл. 2-2 Сводка операций численных константных выражений.

ОПЕРАЦИИ	ОПИСАНИЕ
not	Побитовое НЕ
-	Унарный минус
*	Умножение
/	Деление
+	Сложение (плюс)
-	Вычитание (минус)
<<	Сдвиг влево
>>	Сдвиг вправо
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно
and	Побитовое И
xor	Побитовое исключающее ИЛИ
or	Побитовое ИЛИ

При создании именованных констант тип указывать не нужно, тип выводится. Пример числовых константных выражений:

```
const A = 117;
const B = 23;
const C = ((A + B)/2 + A >>2)*2;          //    C= 92
const D = ((A > B)*5 + (A < B)*3);        //    D= 5
//   64-х битная константа
const E = 1l << 63;
```

2.4.2.2 Адресные константные выражения

Адресное константное выражение более ограничено по выразительным возможностям, чем числовое. При вычислении

адресных значений в константных выражениях используются адреса меток и переменных.

В адресной арифметике для константных выражений разрешены лишь операции сложения и вычитания со следующими замечаниями:

- результат сложения адреса с числом (вычитания из адреса числа) – адресный; запрещено складывать два адреса;
- разность двух адресов из одной секции является числом; запрещено вычитать адреса из разных секций и адрес из числа.

Нарушение указанных условий на построение адресных константных выражений является ошибкой.

В константных выражениях не допускается использование имен регистров или конструкций косвенного доступа к памяти.

Пример адресного константного выражения:

```
ar2 = ARRAY + 2;
```

2.4.3 Определение и использование именованных констант времени компиляции

Определение именованной константы времени компиляции задаёт отношение эквивалентности между символьным обозначением (именем) константы и значением константного выражения. Константа определяется с помощью ключевого слова `const` и знака равенства, например:

```
const MyConst = 0FA5Fh;
```

В правой части конструкции (после знака ‘=’) записывается константное выражение (которое может быть и числовым и адресным), слева – ее символьное обозначение (имя), которое в дальнейшем может использоваться везде, где допустимо использование константного выражения такого типа.

Таким образом создаваемые программные сущности называются константами времени компиляции (или символьными константами), так как время их существования ограничено периодом компиляции программы. Константы времени компиляции сами по себе места в целевой памяти программы не занимают и не обладают такой характеристикой, как адрес. Использование имени константы полностью эквивалентно использованию значения, заданного выражением при её определении - при инициализации какой-либо переменной символьной константой, по адресу переменной будет помещено значение константного выражения из правой части определения константы.

Конструкция определения константы времени компиляции может быть расположена как внутри какой-либо секции, так вне секций. Часто, для того чтобы показать, что константа не зависит от наличия какой-либо секции, её определение выносят в пространство между секций.

Константа должна быть определена до момента её первого использования, например:

```
const C = 12;  // Определение константы времени компиляции
...
begin text
      ...
      ar0 = C;     // Ее первое использование;
      ...
end text;
```

Константы являются доступными только внутри того файла, в котором они определены.

2.4.4 Определение и использование переменных времени компиляции

Переменные времени компиляции (не путать с настоящими переменными описанными в 2.6) аналогичны константам времени компиляции за исключением того, что в течении компиляции их значение может изменяться. Переменная времени компиляции определяется с помощью ключевого слова `var` и знака равенства, например:

```
var MyConst = 0FA5Fh;
```

В правой части конструкции (после знака ‘=’) записывается константное выражение (которое может быть и числовым и адресным), слева – ее символьное обозначение (имя), которое в дальнейшем может использоваться везде, где допустимо использование константного выражения такого типа.

Переменная времени компиляции должна быть определена до момента её первого использования, например:

```
var C = 12;  // Определение переменной времени компиляции
...
begin text
      ...
      ar0 = C;    // Ее первое использование;
      C = 13;     // Присвоение нового значения;
      ar0 = C;    // Здесь в регистр записывается 13;
      ...
end text;
```

Переменные времени компиляции являются доступными только внутри того файла, в котором они определены.

2.4.5 Средства описания специальных констант для программирования целочисленного векторного устройства

При программировании векторного ядра фиксированной точки NeuroMatrix встает задача разбиения 64-битных векторов на элементы. Надо во-первых, сконфигурировать узел векторного АЛУ, для чего служат регистры семейств nb, sb, fcr. Во-вторых, часто нужно формировать 64-векторы констант.

В данном параграфе описаны вспомогательные средства для наглядного и единообразного задания таких констант.

2.4.5.1 Константы для задания разбиения строк (регистр nb)

Синтаксис

`.NM_a_b_c_..._xN` где a, b, c, N – числа

Примеры

Новая форма	Традиционная форма
`nb1= .NM_8_x4;`	`nb1= 80808080h;`
`nb1= .NM_1_x32;`	`nb1= 0ffffffffh;`
`nb1= .NM_3_5_x4;`	`nb1= 84848484h;`
`nb1= .NB_1_2_3_4_5_6_7_4;`	`nb1= 88104225h;`
`nb1l= loword(.NM_10_20_20_14);` `nb1h= hiword(.NM_10_20_20_14);`	`nb1l=loword(8002000020000200h11);` `nb1h=hiword(8002000020000200h11);`

Примечание

Архитектура ntmc4 не позволяет писать значения напрямую в специальные регистры векторного сопроцессора.

Это значит, что вместо

```
nb1= .NM_8_x4;
```

на самом деле надо писать

```
sir= .NM_8_x4;
nb1= sir;
```

Пояснение

В новой форме наглядно отображаются размеры полей и их последовательность. Следует обратить внимание на то, что в последовательности сначала идут поля лежащие в младших адресах, а не наоборот, как при обычном разбиении с помощью прямого задания константы. Это сделано для того, чтобы порядок разбиения совпадал с расположением полей в памяти. Если все таки нужен обратный порядок, можно использовать префикс .rNM вместо .NM

Примечания

Допускается разбиение 32 и 64 разрядных слов. При попытке задать разбиение с другой суммой размеров полей ассемблер выдает ошибку.

Следует помнить, что не существует инструкции, присваивающей регистровой паре длинную константу покомпонентно. Поэтому, вместо

```
nb1=.NM_10_20_20_14;  //   ошибка, nb1l, nb1h будут
проинициализированы младшим словом константы,
ассемблер выдаст предупреждение «Initializer too big,
truncated»
```

Следует писать

```
nb1l= loword(.NM_10_20_20_14);
nb1h= hiword(.NM_10_20_20_14);
```

Примечание

Архитектура ntmc4 не позволяет писать значения напрямую в специальные регистры векторного сопроцессора.

Это значит, что вместо

```
nb1l= loword(.NM_10_20_20_14);
```

на самом деле надо писать

```
sir= loword(.NM_10_20_20_14);
nb1l= sir;
```

Данное замечание касается и всех остальных регистров из данного раздела.

2.4.5.2 Задание разбиения регистра sb

Ввиду того, что на разбиение множителя для операции взвешенного суммирования наложены ограничения, регистр sb имеет особый формат. Чтобы разбиение всех регистров задавалось единообразно, для разбиения sb создан специальный литерал.

Синтаксис

`.SB_a_b_c_..._xN` где a, b, c, N – числа

Примеры

Новая форма	Традиционная форма
`sb= .SB_8_x4;`	`sb= 02020202h;`
`sb= .SB_2_2_4_8_16;`	`sb= 0002022Ah;`

Примечания

Все, что было сказано о .NM также относится и к .SB, например, существует форма .rSB. В разбиении могут участвовать только четные числа, что продиктовано свойствами матрицы.

2.4.5.3 Задание разбиения регистра f1cr

Разбиение регистра f1cr как правило, соответствует разбиению nb, но несет дополнительную информацию. Соответственно усложнена структура литерала. Каждому полю соответствует два числа, одно указывает длину поля, другое – количество значащих битов после округления (количество нулей в соответствующем поле регистра f1cr). Эти два числа в литерале разделены точкой.

Синтаксис

`.FCR_a.k_b.l_..._xN` где a, b, k, l, N – числа

Примеры

Новая форма –
Традиционная форма

Новая форма	Традиционная форма
`f1cr= .FCR_4.2_5.1_7.6_x2;`	`f1cr= 81EC81ECh;`
`f1cr= .FCR_4.1_4.2_4.3_4.2_x2;`	`f1cr= C8CEC8CEh;`

Примечания

Все, что было сказано о .NM также относится и к .FCR, например, существует форма .rFCR. Первые компоненты пар чисел в литерале долны в сумме давать 32 или 64. Второе число в описании поля должно быть больше нуля и меньше первого числа.

2.4.5.4 Конструирование слова по разбиению и значениям полей.

.NM - литерал можно использовать в функциональной форме для того, чтобы «собрать» из значений элементов 64-битного вектора значение константы, в соответствии с заданным разбиением.

Синтаксис

`.NM_a_b_..._xN( a1,b1,…,aN,bN )` где a, b, N, ai – числа

Примеры

Новая форма

```
    matr00: long[8] = (
.NM_4_x8(1,2,3,4,5,6,7,8),
.NM_8_2_4_16_2(1,1,0,-1,1),
.NM_16_x4(-1,0,0,0),
.NM_16_x4(0,-1,0,0),
.NM_16_x4(0,0,-1,0),
.NM_16_x4(0,0,0,-1),
.NM_16_32_16(0,-1,0),
.NM_10_5_10_5_15_4_5_4_3_2_1( -1,0,-1,0,-1,0,-1,0,-
1,0,-1 ),
.NM_10_5_10_5_15_4_5_4_3_2_1( 0,-1,0,-1,0,-1,0,-1,0,-
1,0 ) );
```

Традиционная форма

```
    matr00: long[8] = (
87654321,
7FFFC101,
0000000000000FFFFhl,
00000000FFFF0000hl,
0000FFFF00000000hl,
FFFF000000000000hl,
0000FFFFFFFF0000hl,
9C3E1FFFC1FF83FFhl,
63C1E0003E007C00hl );
```

Примечания

Аргументами могут быть числовые константные выражения. Контролируется размер аргументов, они должны умещаться в своих полях. Допускаются отрицательные аргументы. Литерал в функциональной форме может начинаться только с .NM. Дальнейшее использование сформированного значения никак не контролируется, это просто константа, длинная или короткая.

2.5 Метки

Любая команда в секции кода может быть снабжена меткой (возможно, более чем одной). В этом случае можно организовать передачу управления на помеченную команду посредством одной из команд переходов. Метка - это адрес в памяти той команды, с которой она ассоциирована.

Метка представляет собой идентификатор(п.2.1.1). Приведем примеры корректных имен меток:

```
Loop_0,
__main,
Z987654321A.
```

2.5.1 Объявление метки

Объявление метки - это просто сообщение компилятору, что такая метка должна в дальнейшем встретиться в данном файле.

Пример объявления:

```
MyLabel: label;
```

Конструкция объявления метки может быть расположена как внутри какой-либо секции, так и вне секций.

2.5.2 Определение метки

Помимо объявления метки существует ее определение. Пользователь определяет, что меткой помечается некоторая ассемблерная инструкция. Пример определения метки:

```
    gr0 = 123;
<loop>      // место определения метки.
    [ar0++] = gr0;
     ...
    goto loop;
```

Метка определяется путем задания ее имени в угловых скобках, например, `<loop>`. При этом она помечает ту команду, которая следует после нее до ближайшей "`;`". В приведенном выше примере меткой помечается команда `[ar0++] = gr0;`.

2.5.3 Ссылки на метку

Кроме объявления метки и ее определения в программе встречаются ссылки на метку. Их может быть несколько. В основном ссылки встречаются в командах перехода. Приведем примеры ссылок на метку:

```
goto loop;   // ссылка на метку loop.
call Func;   // вызов функции, Func - метка ее начала.
gr3 = Func;  // присвоение регистру адреса метки Func.
```

Ссылка на метку содержит только ее имя так, как оно было объявлено, без каких либо дополнительных скобок.

Приведем пример использования метки:

```
L: label;   // объявление метки.
...
begin text
...
<L>             // определение метки.
    gr0 = gr2 or gr3;
    ...
    goto L;   // ссылка на метку.
...
end text;
```

2.5.4 Типы связывания и область действия меток

В языке ассемблера поддерживается четыре различных типа связывания меток:

- `local` - локальные метки;
- `global` - глобальные метки;
- `extern` - внешние метки;
- `weak` - глобальные метки со слабым связыванием.

Служебное слово, определяющее тип метки, ставится перед ее именем при объявлении, например:

```
global MyFunc: label;
local  MyFunc: label;
extern MyFunc: label;
weak   MyFunc: label;
```

Тип метки определяет область ее действия.

Простейшим типом является тип **`local`**. Область действия метки такого типа ограничена рамками файла, в котором она определена.

Внутри одного файла локальная метка может быть определена, то есть ассоциирована с определенным адресом в программе, только один раз. В то же время внутри одного файла может находиться произвольное количество ссылок на метку. Ссылки из другого файла на локальную метку данного файла запрещены.

В разных файлах могут использоваться локальные метки с одинаковыми именами, каждая из которых имеет свою область действия и свое место определения.

Для удобства написания программ, можно опускать слово `local` в объявлении локальной метки, например, можно записать:

```
MyFunc: label;
```

и ассемблер сам определит, что данная метка имеет локальный тип. Более того, можно опускать и само объявление локальных меток. Если ассемблер встретит определение метки без предварительного объявления, он полагает, что это локальная метка.

Объявление локальных меток в основном используется для повышения читаемости, документированности программ.

Тип **`global`** в объявлении метки сообщает компилятору, что она будет определена в данном файле, а область ее действия, то есть возможность ссылаться на нее, простирается на все файлы программы.

Обязательные условия использования глобальных меток:

- не допускается использование в программе двух глобальных меток с одинаковым именем;
- глобальная метка должна быть определена в том же файле, в котором она была объявлена.

Во всех остальных файлах программы могут содержаться ссылки на данную метку, то есть на адрес, с которым она была ассоциирована. Для того чтобы, глобальная метка, определенная в другом файле, стала доступна из данного файла, она должна быть объявлена как внешняя (`extern`).

Тип **`extern`** используется, когда в данном файле содержится хотя бы одна ссылка на глобальную метку, определенную в другом файле. Только после того, как была объявлена внешняя метка, она становится доступной из данного файла.

Соответствующая глобальная метка метка может располагаться в другом файле, но может быть и в текущем файле

Итак, глобальная метка распространяет область действия на всю программу, состоящую из произвольного количества файлов. Она может быть определена только однажды. В том файле, где она определена, она должна быть объявлена, как `global`. Во всех остальных файлах программы при необходимости доступа к данной метке она объявляется, как `extern`. Возможно многократное объявление в одном файле одной и той же метки.

Пример:

```
файл F1.ASM:
  global MyFunc: label;  // объявление глобальной метки
  ...          // метка будет определена в этом же файле.

  begin text
  ...
  <MyFunc>                // определение метки.
      push ar0, gr0;
      ...
      return;
      ...
      call MyFunc;    // ссылка на метку.
  end text;

файл F2.ASM:
  extern MyFunc: label;  // объявление внешней метки
  ...                    // метка определена в другом файле.

  begin text1
      ...
      call MyFunc;  // ссылка на внешнюю метку.
  end text1;
```

Кроме меток с типами `global` и `extern` к глобальным относятся метки с типом связывания **`weak`**. Областью действия меток типа `weak` являются все файлы программы. Отличие метки `weak` от `global` состоит в том, что она обладает меньшим приоритетом.

Если в программе встречается определение метки с типом связывания `weak` и одноименной с ней метки с типом связывания `global`, то `weak` метка игнорируется редактором связей, а все ссылки настраиваются на место определения глобальной метки. В отсутствие одноименной метки `global` она сама воспринимается редактором связей, как глобальная метка.

Особенности меток с типом связывания `weak`:

- в одном файле не может быть двух одноименных меток;
- метка должна быть определена в том же файле, в котором она была объявлена;
- допускается использование в программе двух и более меток с одинаковым именем. При этом, если в разных файлах встретились метки с одинаковым именем, то редактор связей в качестве места определения выбирает первое из встреченных мест;
- если в программе встречена глобальная метка, то все одноименные с ней `weak` метки игнорируются.
- в отсутствие одноименной глобальной метки метка со слабым типом связывания становится глобальной, то есть к ней осуществляется доступ из других файлов путем объявления ее внешней.

Пример использования меток со слабым типом связывания:

файл `F1.asm`, *в котором* `AB` - *слабое связывание*:

```
weak AB: label;
...
begin text
  <AB>
       ...
       gr0 = gr1 + gr2;
       ...
       return;
end text;
```

файл `F2.asm`, *в котором* `AB` - *глобальное связывание:*

```
global AB: label;
...
begin text
  <AB>
       ...
       gr0 = gr1 - gr2;
```

```
        ...
        return;
end text;
```

файл `F3.asm` *- вызов функции* `AB`*:*

```
extern AB: label;
global __main: label;
...
begin text
  <__main>
        ...
        call AB;
        ...
        return;
end text;
```

Если при помощи редактора связей в единую программу будут собраны файлы `F1.ELF` и `F3.ELF`, то будет вызвана функция из файла `F1.ELF`, содержащая операцию `gr0 = gr1 + gr2`.

Если в воедино будут собраны файлы `F1.ELF`, `F2.ELF` и `F3.ELF`, то будет вызвана функция из файла `F2.ELF`, содержащая операцию `gr0 = gr1 - gr2`.

Таким образом, в отсутствие глобальной метки слабая воспринимается, как глобальная, а при наличии глобальной игнорируется.

2.6 Переменные

Под переменной в языке ассемблера NeuroMatrix понимается адрес области памяти процессора, в которой хранятся данные, используемые программой в процессе счета.

Имя переменной представляет собой идентификатор(п.2.1.1). Приведем примеры корректных имен переменных:

```
Array_0,
__Long_Value,
Z987654321A.
```

Каждое имя переменной в языке ассемблера ассоциировано с адресом конкретной области памяти. Переменные доступны для чтения или записи.

В языке ассемблера существует два способа использования переменных. Первый - это получение адреса переменной, второй - получение содержимого ячейки памяти, с которой ассоциирована данная переменная, то есть значения переменной.

2.6.1 Получение адреса переменной

В языке ассемблера имеется два простых типа данных и несколько видов агрегатности. Таким образом, переменные делятся на простые и составные (массивы и структуры).

Для доступа к элементам составных переменных используются модификаторы имени переменной, записываемые непосредственно за именем переменной.

Для элемента массива модификатор имеет вид: '`[`*`индекс`*`]`', где *`индекс`* должен быть числовым константным выражением, имеющим значение порядкового номера элемента массива (нумерация элементов начинается с 0).

Для элемента (поля) структуры модификатор имеет вид: '`.`*`имя_поля`*'.

Для того чтобы получить адрес переменной, достаточно просто использовать ее имя. Приведем примеры корректных записей получения **адресов** переменных:

```
ar0 = Value;
ar0 = Array[4];
ar0 = Struct.Field;
```

2.6.2 Получение значения переменной

Для того, чтобы получить значение переменной, необходимо ее имя заключить в квадратные скобки. Приведем примеры корректных записей получения **значений** переменных:

```
gr0 = [Value];
gr0 = [Array[4]];
gr0 = [Struct.Field];
```

2.6.3 Простые переменные

К простым типам переменных относятся минимальные аппаратно поддерживаемые типы значений, рассматриваемые как единое целое.

Так как процессор NeuroMatrix поддерживает два базовых формата – 32-разрядное и 64-разрядное слово, то в языке ассемблера имеется два простых типа данных, которые называются словным и двухсловным. В программах они обозначаются служебными словами `'word'` для 32-х разрядных и `'long'` для 64-х разрядных переменных:

Примеры описаний данных простых форматов:

```
Var1 : word;
Var2 : word = 0abch;
Var3 : long;
```

Примечание

Переменные типа `long` *всегда располагаются в памяти процессора по четному адресу. За этим следит ассемблер в процессе компиляции. Если двойное слово приходится на нечетный адрес, перед ним вставляется пустое короткое неиспользуемое слово, то есть осуществляется его выравнивание по четному адресу.*

2.6.4 Составные переменные

Составные типы переменных формируются на основе простых. В языке имеются два таких способа: массивы и структуры.

2.6.4.1 Массивы

Массив представляет собой конечное упорядоченное множество элементов одинакового формата. Размер массива (количество элементов) задается статически, то есть определяется во время компиляции программы. Доступ к отдельным элементам массивов производится посредством задания имени массива и индекса элемента в этом массиве. Считается, что первый элемент массива имеет нулевой номер.

Описание массива должно содержать указание его размера в квадратных скобках, приписываемое к обозначению формата отдельных элементов (который в этом случае называется базовым форматом массива).

Примеры описаний массивов:

```
Word : word[32]; // массив из 32-х 32-разрядных слов.
Long : long[10]; // массив из 10-ти 64-разрядных слов.
```

Примеры получения адресов элементов массива:

```
ar0 = Word[4]; // адрес 4-ого слова массива.
ar0 = Long[8]; // адрес 8-ого двойного слова массива.
```

Примеры получения значений элементов массива:

```
ar0 = [Word[4]]; // значение 4-ого слова массива.
ar0,gr0 = [Long[8]]; // значение 8-ого двойного слова.
```

Примечание

При индексации по элементам массива учитывается тип элементов. Если элементами массива являются двойные слова, то и расстояние между двумя соседними элементами массива равно двум словам, например:

```
ar0 = Long[0]; // адрес 0x00000010;
ar1 = Long[1]; // адрес 0x00000012;
ar2 = Long[2]; // адрес 0x00000014;
```

2.6.4.2 Структуры

Структура представляет собой сложную переменную, состоящую из конечного множества элементов произвольного типа. Доступ к элементам структуры осуществляется заданием имени структурного значения и имени элемента (пример см. ниже).

Чтобы задать описание структурной переменной, необходимо сначала описать общий вид ("шаблон") ее структуры. Приведем пример описания шаблона структуры:

```
struct MyStructName // открывающая скобка структуры
    F1 : word;      // поля структуры
    F2 : long;      //  -//-
    F3 : long;      //  -//-
end MyStructName;   // закрывающая скобка структуры
```

Примечание

Описание шаблона структуры является абстрактным понятием до тех пор, пока в программе не появилась переменная данного типа. Само по себе описание шаблона не занимает места в памяти, поэтому для улучшения читаемости программы рекомендуется выносить его за пределы секций.

Имея описание шаблона структуры, можно объявить конкретную переменную, обладающую такой структурой. Для этого следует использовать имя введенного ранее шаблона, например:

```
nobits ".data"
    ...
    Var5 : MyStructName;
    ...
end ".data";
```

В качестве элементов составных форматов, кроме простых, можно использовать также другие составные форматы. Иными словами, допускаются такие конструкции, как массив структур, структуры, содержащие в качестве своих элементов массивы или другие структуры.

Для того чтобы получить адрес элемента структуры, используется точечная нотация, то есть сначала идет имя структуры, а затем через точку имя ее поля, например:

```
nobits ".data"
    ...
    Struct : MyStructName;
    ...
end ".data";
...
begin text
    ...
    ar0 = Struct.F2;
    gr0 = [Struct.F3];
    ...
end;
```

Как уже было замечено выше, переменные типа `long` всегда располагаются в памяти процессора по четному адресу. Это же относится к переменным, если они входят в состав структуры. Если на этот факт не обращать внимания, то в структуре могут появиться неиспользуемые ячейки памяти - пустоты между полями, как в нижеприведенном примере:

```
// Переменная структурного типа
Var5: MyStructName;
// ======= Размещение переменной Var5 в памяти ==========
Var5.F1 (short)    | 0x00000000
 Empty field       | 0x00000001
Var5.F2 (long)     | 0x00000002
Var5.F3 (long)     | 0x00000004
```

Примечание

Если структура содержит поле типа `long`*, то данные этого поля при объявлении структуры выравниваются по четному адресу.*

2.6.5 Начальные значения

При описании переменных можно задавать их начальные значения. Начальное значение или список начальных значений следует за объявлением типа переменной и предваряется знаком "=". Приведем примеры начальной инициализации переменных:

```
data ".data"
    Var1: word = 01234567h;
    Var2: long = 0123456789abcdefhl;
    Var3: word[4] = ( 0, 1, 2, 3 );
end ".data"
```

Для данных простых форматов начальное значение задается в виде константы (или константного выражения), например:

```
Var1 : word = 123;
```

При инициализации переменных типа long после значения константы следует добавить латинскую букву 'L' или 'l' для указания того, что данное число считается длинным. Например:

```
Var2 : long = 0ffffffffffffffHL;
```

В случае инициализации данных составных форматов в качестве начального значения используется список начальных значений, разделяемых запятыми и заключенный в круглые скобки. Количество значений в списке должно соответствовать составному типу, а вид каждого значения – типу соответствующего элемента составного типа. Ассемблером не производится проверки соответствия инициализатора и инициализирующегося объекта по базовым типам. При инициализации структуры из элементов типа long необходимо явно приписывать 'L' к использующимся числовым литералам. В противном случае ассемблер самостоятельно приведет его к двухсловному типу, взяв в качестве младшего присваиваемую переменной константу инициализации, а в качестве старшего нулевое слово. Об этом пользователь может узнать из сообщения, выдаваемого в процессе ассемблирования (см. документ: ЮФКВ.30047-01 95 01. Кросс-средства разработки программ. Руководство пользователя.).

Примеры инициализаций переменных составных типов:

```
struct MyStruct
    First : word;
    Second: long;
    Third : word[2];
end MyStruct;
...
data ".data"
    Var1 : word[3] = ( 1, 1, 1 );
    Var2 : MyStruct[2] = ( ( 3, -1L, (12, 345) ),
                           ( 2, -2L, (67, 89)  ) );
end ".data";
```

Если в списке инициализирующих констант задается последовательность одинаковых значений, то запись можно сократить, указав повторитель `dup`. Пример:

```
Var1 : word[3] = ( 1 dup 3 );
Var2 : MyStruct[2] = ( (3, -1L, (12, 345)) dup 2 );
```

2.6.6 Область действия данных

Переменные, описываемые в программе на языке ассемблера, можно классифицировать по их области действия. Допускаются следующие области действия данных:

- `local` - локальные переменные, которые используются только в пределах данного файла и недоступны (неизвестны) вне его, в других файлах , образующих программу;

- `extern` - внешние переменные, описанные в некотором другом файле и используемые в данном;
- `global` - глобальные данные, описанные в данном файле и доступные для других файлов программы;
- `weak` - глобальные данные со слабым типом связывания, описанные в данном файле и доступные для других файлов программы. Если в ней встретилось определение глобальной переменной с тем же именем, то переменные данного типа игнорируются;
- `common` - общие данные, не описанные ни в одном файле и доступные из любого файла (место под общие данные резервируется редактором связей).

Служебное слово `local` в спецификациях локальных переменных можно опускать; считается, что если в описании переменной явно не указана ее область действия, то переменная локальна в данном модуле.

Чтобы указать, что некоторое описание вводит внешнюю переменную, необходимо задать перед ее именем служебное слово `extern`, например:

```
extern Var3 : word;
```

Для внешних данных, описанных в модуле, память не отводится; все ссылки на такие переменные в программе относятся к описанию глобальной переменной с таким же именем в некотором другом модуле. В описаниях внешних данных инициализация не допускается.

Глобальные данные вводятся описаниями, в которых перед именем переменной указано служебное слово `global`, например:

```
global Var1 : long;
weak Var3 : word = 1234;
```

Разновидностью глобальных данных являются данные типа `weak`, имеющие слабый тип связывания.

На все перечисленные выше типы связывания переменных распространяются те же правила, что и на метки, за исключением того, что нельзя объявлять переменные global и weak за пределами секций(см. подраздел 2.5 Метки). Чтобы описать общие данные, необходимо задать `common` перед именем переменной:

```
common Var : word;
```

Инициализация общих данных не допускается. Особенности переменных с типом связывания `common`:

- в одном файле не может быть двух одноименных переменных;
- переменные с одинаковыми именами, объявленные в различных модулях программы объединяются редактором связей воедино;

- все ссылки на одноименные переменные из разных файлов в результате настраиваются на один и тот же адрес памяти;
- метки с одинаковыми именами могут иметь различные типы, например, если в одном файле объявлена переменная `common MyCommon: word;`, а в другом `common MyCommon: long[4]`, то после объединения общий размер выделенной памяти будет равен `sizeof(long)*4`;
- место под переменные выделяется редактором связей в специальной секции `.common`. Эта секция создается автоматически при наличии переменных данного типа связывания.

2.6.7 Места для объявлений и начальной инициализации переменных

При объявлении переменной происходит резервирование области памяти, необходимой для размещения ее значения (за исключением `extern` и `common`). Если переменная инициализируются начальным значением, то оно записывается в эту область памяти.

Переменные типов `global`, `weak` должны объявляться **только в пределах секций**. Для этого специально отведены секции инициализированных и неинициализированных данных. Более подробно о секциях данных см. подраздел 2.3 Секции.

При объявлении переменных типов `extern` и `common` не происходит резервирование памяти. Поэтому для большей наглядности рекомендуется объявлять переменные данных типов **вне секций**.

Пример правильного и неправильного объявления переменных:

```
data ".data"
    // Область файла внутри секции.
    global Aglob: word; // правильное объявление;
           Bloc : long; // правильное объявление;
    weak   Cweak: word; // правильное объявление;
end ".data";


// Область файла вне секций.
global Dglob: word; // неправильное объявление;
common Ecomm: long; // правильное объявление;
       Floc : long; // правильное объявление;
weak   Gweak: word; // неправильное объявление;
extern Hextr: word; // правильное объявление;


data ".data1"
    // Область файла внутри секции.
```

```
    common Icomm: long; // нерекомендуемое объявление;
    extern Jextr: word; // нерекомендуемое объявление;
end ".data1";
```

2.7 Директивы языка ассемблера

В данном разделе перечислены все директивы ассемблера, описано их назначение, приведены правила работы с ними. В отличие от инструкций процессора директивы не транслируются в какой-либо специальный код, а лишь оказывают влияние на ход компиляции программы. Директивы ассемблера позволяют:

- осуществлять условную компиляцию;
- задавать выравнивание элементов программы, а именно инструкций и данных;
- определять количество повторений блоков данных;
- вводить в текст на языке ассемблера отладочную информацию;
- разрешать и запрещать параллельное выполнение команд процессора.

Далее будут приведены две таблицы директив. Сначала сводная таблица без списка директив отладочной информации и отдельно таблица директив отладочной информации. В каждой таблице директивы расположены в алфавитном порядке. А затем к каждой директиве будут даны комментарии по использованию.

Табл. 2-3 Сводная таблица директив языка ассемблера (Часть 1).

МНЕМОНИКА	ОПИСАНИЕ
.align	Выравнивание по четному адресу.
.branch(*nmc3 и ранее)	Включение режима параллельного выполнения инструкций процессора.
.endif	Конец блока условной компиляции.
.endrepeat	Конец блока повторения инструкций.
.if условие	Начало блока условной компиляции.
.repeat кол-во повторений	Начало блока повторения инструкций.
.wait(*nmc3 и ранее)	Отключение режима параллельного выполнения.
.nm6403	Переключение в режим компиляции для NM6403.
.nm6405	Переключение в режим компиляции для NM6405.

Примечание

Инструкции отмеченные звёздочкой работают только для версий архитектуры nmc3 *и более ранних. Начиная с* nmc4 *переключение режима параллельного выполнения осуществляется динамически специальными инструкциями процессора.*

В языке ассемблера имеется набор директив для задания информации, необходимой для дальнейшей отладки.

Ассемблер поддерживает генерацию отладочной информации в формате DWARF версии 2.0. Описание формата можно найти в документе: **TIS Committee "DWARF Debugging Information Format. v2.0".**

Примечание

Директивы отладочной информации генерируются компилятором с языка C++, для использования при программировании на языке ассемблера не предназначены.

Все директивы отладочной информации начинаются с префикса '.debug_'.

После директив языка ассемблера всегда должен ставиться завершитель строки ";".

2.7.1 Директива .align

Директива `.align` в процессе компиляции сообщает ассемблеру, что следующая за ней процессорная инструкция или элемент данных должны начинаться с четного адреса. Если текущий адрес был нечетным, то будет выбран ближайший четный адрес, следующий за ним. Если текущий адрес был четным, то он не изменится.

Директива `.align` не требует никаких дополнительных параметров.

Директива `.align` не может использоваться вне секций. Если она используется внутри секции кода, при выравнивании в программу вставляется инструкция `nul`; если в секции инициализированных данных, пропуск заполняется нулем; если в секции неинициализированных данных, текущий адрес увеличивается на 1.

Пример использования директивы `.align`:

- При работе с данными:

```
data "Init"          // секция всегда начинается
                     // с четного адреса.
    Var1: word[5] = (-1 dup 5);
                     // нечетное кол-во элементов.
                     // следующая переменная должна быть
                     // размещена по нечетному адресу.
    .align;          // директива выравнивания;
                     // ассемблер пропускает слово.
```

```
                    // переменная начинается с четного адреса.
    Var2: word[2] = (5A5A5A5Ah dup 2);
end "Init";
```

Тогда в памяти данные разместятся следующим образом:

```
00: FFFFFFFF FFFFFFFF   // Var1 5 элементов.
02: FFFFFFFF FFFFFFFF
04: FFFFFFFF 00000000   // вставленное нулевое слово.
06: 5A5A5A5A 5A5A5A5A   // Var2 с четного адреса.
```

- При работе с инструкциями:

```
begin "textFunc"    // секция всегда начинается
                    // с четного адреса.
   ...
   gr0 = [Var1];    // длинная команда расположена
                    // по четному адресу.
   gr1 = gr0 << 1;  // короткая команда.
                    // следующая команда должна быть
                    // размещена по нечетному адресу.
   .align;          // директива выравнивания;
                    // ассемблер вставляет nul.
   gr2 = not gr1;   // короткая команда,
                    // расположена по четному адресу.
end "textFunc";
```

2.7.2 Директива .branch (только nmc3)

Директива `.branch` устанавливает бит параллельности равным единице во всех следующих за ней инструкциях процессора до тех пор, пока не будет встречена директива `.wait` или не будет достигнут конец файла. Работает только для версии архитектуры nmc3 и более ранних.

Директива `.branch` не требует никаких дополнительных параметров. Она может быть использована только внутри секции кода.

По умолчанию в программе бит параллельности сброшен в 0. Использование директивы `.branch` позволяет включить режим параллельного исполнения инструкций процессора.

Пример использования директивы `.branch`:

```
begin "textFunc"
<MyFunc>            // метка начала функции;
                    // бит параллельности p = 0.
    ar0 = Vector;
.branch;            // бит параллельности p = 1 здесь
                    // и далее во всех командах.
    rep 16 ram = [ar0++]; // векторная команда
                          // выполняется 16 тактов.
```

```
        gr0 =  gr1 << 4;          // скалярная команда
                                  // выполняется параллельно.
.wait;                  // бит параллельности p = 0 здесь
                        // и далее во всех командах.
        gr0 = gr1 << 4;           // скалярная команда
                                  // ожидает окончания работы
                                  // векторной команды.
end "textFunc";
```

2.7.3 Директива .endif

Директива `.endif` является закрывающей скобкой блока условной компиляции, используется только в паре с директивой открывающей скобки блока условной компиляции `.if`, и самостоятельного значения не имеет.

Пример использования `.endif` приведен в пункте 2.7.5 Директива .if .

Директива `.endif` не требует никаких дополнительных параметров.

2.7.4 Директива .endrepeat

Директива `.endrepeat` сообщает ассемблеру, что достигнут конец блока инструкций, который должен быть продублирован в программе то число раз, которое указано при директиве `.repeat`.

Директива `.endrepeat` используется только в паре с `.repeat`, и никакого самостоятельного значения не имеет. Пример использования `.endrepeat` приведен в пункте 2.7.6 Директива .repeat.

Директива `.endrepeat` не требует никаких дополнительных параметров.

2.7.5 Директива .if

Директива `.if` *условие* определяет начало блока условной компиляции. Она используется в паре с `.endif` и задает условия, при которых блок инструкций процессора, заключенный в кавычки `.if ... .endif`, будет скомпилирован ассемблером.

Результат выражения, стоящего после `.if`, рассматривается как булевское число. Если условие истинно, ассемблер компилирует данный блок, если ложно, блок пропускается.

Блок условной компиляции может использоваться как в секциях данных, так и в секциях кода. Он не может располагаться вне секций.

Пример программы с использованием блока условной компиляции:

```
const DEBUG = 1;  // константа используется для
...               // отладки программы.
nobits ".data"    // секция данных.
...
.if DEBUG;        // начало блока условной компиляции.
    GenRegs: word[8];  // место под 8 регистров.
.endif;           // конец блока условной компиляции.
end ".data";
...
begin ".text"     // секция кода.
...
.if DEBUG;    // начало блока условной компиляции.
    [GenRegs[0]] = gr0;  // в отладочном режиме
    [GenRegs[1]] = gr1;  // все регистры общего
    [GenRegs[2]] = gr2;  // назначения сбрасываются
    [GenRegs[3]] = gr3;  // в выделенную область
    [GenRegs[4]] = gr4;  // памяти.
    [GenRegs[5]] = gr5;  // В реальном режиме работы,
    [GenRegs[6]] = gr6;  // когда DEBUG = 0, данный
    [GenRegs[7]] = gr7;  // блок будет пропущен.
.endif;       // конец блока условной компиляции.
...
end ".text";
```

2.7.6 Директива .repeat

Директива `.repeat` *кол-во повторений* определяет начало блока, который будет размножен то число раз, которое указано в качестве ее параметра. Она используется в паре с `.endrepeat`.

Количество повторений задается положительной константой. Скобки могут использоваться, как альтернатива цикла, однако следует помнить, что большое количество повторений может неоправданно увеличить объем кода.

Приведем пример использования `.repeat ... .endrepeat`:

```
begin ".text"
    ...
    gr2 = [Mask];
    gr0 = [ar0++];
.repeat 9;   // повторить блок из двух команд 9 раз.
    gr0 = [ar0++] with gr1 = gr0 and gr2;
    [ar1++] = gr1;
.endrepeat; // конец блока.
    [ar1++] = gr1;
    ...
end ".text";
```

Внутри блока повторения не должно встречаться определение меток и переменных, так как подобные определения

дублируются текстуально, что приведет к потоку синтаксических ошибок.

2.7.7 Директива .wait (только nmc3)

Директива `.wait` устанавливает бит параллельности равным нулю во всех следующих за ней инструкциях процессора до тех пор, пока не будет встречена директива `.branch` или не будет достигнут конец текущей секции. Работает только для версии архитектуры nmc3 и более ранних.

Директива `.wait` не требует никаких дополнительных параметров. Она может быть использована только внутри секции кода.

По умолчанию в программе бит параллельности сброшен в 0. Если при помощи директивы `.branch` был включен режим параллельного исполнения инструкций процессора, то `.wait` сбрасывает его. В результате каждая инструкция процессора, прежде чем выполниться, будет дожидаться выполнения предыдущей.

Пример использования директивы см. 2.7.2 Директива .branch.

2.7.8 Директивы .nm6403, .nm6405.nm64revision и.nm64revision2

Директивы `.nm6403`, `.nm6405` заставляют транслятор убедиться, что исходный текст транслируется для подходящей версии процессора. Используйте как меру предосторожности, чтобы гарантировать, что программу не будут компилировать для той версии процессора, с которой она несовместима.

Текущий режим можно узнать, проверив значение предопределенного символа `.nm64revision`. Это значение – число от 3 до 5.

Пример:

```
// Запустить таймер
.if (.nm64revision == 3) or (.nm64revision == 4);
        pswr set 0001_0000h;
.endif;
.if .nm64revision == 5;
        pcr set 0_D000_0000h;
.endif;
```

Отметим, что данные дерективы, как и все прочие, должны располагаться внутри секций.

Для версий архитектуры начиная с nmc4 следует использовать .nm64revision2

2.8 Псевдофункции

В язык ассемблера введено несколько псевдокоманд, облегчающих запись и вычисление константных выражений программы и повышающих ее наглядность.

Псевдофункции обрабатываются на этапе ассемблирования. В качестве входных данных они используют константу или константное выражение. Результатом их работы также является константа, которая затем используется для инициализации переменных, модификации регистров, адресации.

Псевдофункции могут рассматриваться как часть константных выражений, поэтому они могут использоваться в выражениях как внутри секций, так и вне их.

В языке ассемблера могут использоваться следующие псевдофункции:

Табл. 2-4 Сводная таблица псевдофункций языка ассемблера.

ПСЕВДОФУНКЦИИ	ОПИСАНИЕ
`double`	Преобразование 64-х разрядного числа с плавающей точкой во внутреннее представление (IEEE-754).
`float`	Преобразование 32-х разрядного числа с плавающей точкой во внутреннее представление (IEEE-754).
`hiword`	Получение старшей части 64-х разрядного слова.
`loword`	Получение младшей части 64-х разрядного слова.
`offset`	Получение смещения поля структуры относительно ее начала.
`sizeof`	Получение размера объекта данных.

2.8.1 Функция loword

Функция `loword` служит для получения младшей части 64-разрядного слова, выраженного при помощи константы или константного выражения.

Следующий пример демонстрирует способ ее использования:

```
begin ".text"
    ...
    gr0 = loword(0f0f0f0ff0f0hl * 5);
    ...
end ".text";
```

Данная функция возвращает 32-х разрядное значение.

С логической точки зрения функция `loword` не может быть использована при вычислении адресного выражения, поскольку по

своей природе адресные выражения не могут давать результат, количество значимых битов в котором превышает 32.

2.8.2 Функция hiword

Функция `hiword` служит для получения старшей части 64-разрядного слова, выраженного при помощи константы или константного выражения.

Следующий пример демонстрирует способ ее использования:

```
begin ".text"
    ...
    gr0 = hiword(0f0f0f0ff0f0hl * 5);
    ...
end ".text";
```

Данная функция возвращает 32-х разрядное значение.

Функция `hiword` не может быть использована при вычислении адресного выражения, поскольку по своей природе адресные выражения не могут давать результат, количество значимых битов в котором превышает 32.

2.8.3 Функция sizeof

Функция `sizeof` служит для получения реального размера переменной указанного типа с учетом всевозможных выравниваний и внутренней структуры.

Результатом работы данной функции является размер указанного типа данных в 32-х разрядных словах.

Приведем пример использования `sizeof()`:

```
struct S          // объявление типа структуры.
    Var1: word;   // короткое слово, после него перед
    Var2: long;   // длинным образуется промежуток.
    Var3: word[4];
end S;
begin ".text"
    gr0 = sizeof(S);  // полученный результат: 8 слов.
end ".text";
```

В качестве входного параметра функции `sizeof` подается имя типа, а не имя переменной данного типа. Приведем примеры правильного и неправильного использования `sizeof`:

- правильное использование, в качестве аргумента подано имя типа:

```
gr0 = sizeof(S) + 10;
```

- неправильное использование, потому что в качестве аргумента вместо имени типа подано имя переменной:

```
gr0 = sizeof(Dest) + 10;
```

2.8.4 Функция offset

Функция `offset` служит для получения смещения указанного поля в указанной структуре с учетом возможного выравнивания.

Результатом работы функции является размер смещения поля от адреса начала структуры в 32-х разрядных словах.

Данная функция полезна для доступа к полям структуры адресуемой через регистр.

Приведем пример использования функции `offset`:

```
struct S
    Var1: word;
    Var2: long;
end S;
...
data ".data"
    VarStruct: S = (1, -1l,);
end ".data";
...
begin ".text"
    // заносим адрес структурной переменной в регистр.
    ar0 = VarStruct;
    // читаем поле структуры.
    ar1, gr1 = [ ar0 += offset(S, Var2) ];
end ".text";
```

В качестве входного параметра функции `offset` подается имя типа, а не имя переменной данного типа. Приведем примеры правильного и неправильного использования `offset`:

- правильное использование, в качестве аргумента подано имя типа:

```
gr0 = offset(S, Var2) + 10;
```

- неправильное использование, потому что в качестве аргумента вместо имени типа подано имя переменной:

```
gr0 = sizeof(VarStruct, Var2) + 10;
```

2.8.5 Функция float

Функция `float` переводит переменную, записанную в формате числа с плавающей точкой во внутреннее 32-х разрядное представление, выбранное в соответствии с IEEE-754.

Операции с плавающей точкой аппаратно не поддерживаются процессором, поэтому вся арифметика чисел с плавающей точкой реализована в виде библиотеки функций, входящей в состав `libc.lib`.

Ассемблер использует формат IEEE-754 для внутреннего представления чисел с плавающей точкой.

Приведем пример использования функции `float`:

```
Float1: word = float( 1.57 ) - float(-4.32E2);
Float2: word = float( -4.98 ) + float(5.51E2);
Float2: word = float( -2.23E-3 ) + float(5.51E-2);
```

В приведенных примерах использованы все возможные форматы представления чисел с плавающей точкой, поддерживаемые процессором.

Формат вещественного числа:

`[+|-]num[[.num]E[+|-]num]`, *где* `num` - *десятичные числа*.

Или

`[+|-]num.num`

2.8.6 Функция double

Функция `double` переводит переменную, записанную в формате числа с плавающей точкой во внутреннее 64-х разрядное представление, выбранное в соответствии с IEEE-754.

Функция `double` аналогична функции `float`, с той поправкой, что в отличие от `float` функция `double` оперирует с 64-х разрядными числами.

2.9 Использование макросов в языке ассемблера

2.9.1 Синтаксис

Определение макроса:

```
macro имя_макроса ( [параметр1 [, параметр2 ...]] )
    последовательность_элементов
end имя_макроса;
```

Вызов макроса:

```
имя_макроса( [параметр1 [, параметр2 ...]] );
```

Объявление внешних макросов, импорт из макробиблиотек:

```
import [имя_макроса1 [, имя_макроса2 ...]] from
имя_библиотеки;
```

2.9.2 Описание

Здесь `имя_макроса` - произвольный идентификатор. Формальные параметры макроса (если макрос имеет параметры) также должны быть идентификаторами.

Полная синтаксическая и семантическая проверка тела макроса производится только при подстановке, с учётом контекста подстановки и фактических параметров вызова.

В качестве подстановочных параметров могут быть использованы практически любые конструкции, за исключением

запятых (допускаются запятые внутри круглых скобок) и непарных круглых скобок.

Примечание: *передать в макрос известную программную сущность не по значению, а по имени, невозможно. Нельзя, к примеру, передать в макрос имя переменной времени компиляции для её изменения внутри тела макроса.*

Примечание: *передача ещё не определённого идентификатора даёт возможность для создания внутри макроса новых программных сущностей, именованных заданным образом:*

```
macro entry_point( name )
<name>
    nul 10;
    call subroutine;
end entry_point;
```

Определения макросов могут быть расположены в любом месте ассемблерного текста, однако предпочтительнее помещать их вне секций, в начале файла.

Подстановка макроса возможна в любом месте программы, где разрешено использование всех содержащихся в макросе конструкций. Например, макрос, содержащий инструкции, не может быть вызван вне секций.

Внутри тела макроса возможно использование блоков условной компиляции.

Допускается рекурсивный вызов макроса.

2.9.3 Использование меток в макросах

Использование меток в макросах имеет следующее ограничение - макрос, в теле которого имеется определение метки, может быть инстанцирован (подставлен) только один раз, так как при последующих подстановках будет возникать ошибка повторного определения метки с тем же именем.

Для преодоления данного ограничения в языке существует механизм собственных меток макросов - метка, объявленная внутри макроса со спецификатором `own`, обрабатывается особым образом так, чтобы в каждой подстановке макроса данная метка имела уникальное среди всех подстановок данного макроса имя. Например:

```
macro FEQ ( Res, Arg1, Arg2 )
extern FCmp :label;
own Cont :label;

  ar5 = sp;
```

```
    sp += 2;
    [ ar5++ ] = Arg1;
    [ ar5 ] = Arg2;
    call FCmp;
    if carry delayed skip Cont
       with Res = false noflags;
    sp -= 2;
    // cond. skip Cont
    if <>0 skip Cont;
    Res++;
<Cont>
end FEQ;
```

Макрос `FEQ()`, использующий собственную метку `Cont` для организации внутреннего ветвления, может инстанцироваться более одного раза.

Использование спецификатора `own` при объявлении метки, возможно, только внутри тела макросов.

Вообще, “own” позволяет создавать и другие локальные сущности, например константы, с уникальными (в пределах исходного файла) именами, т.е. при каждой подстановке макроса создается новый объект.

2.9.4 Импорт макросов из макробиблиотек.

Возможность использования макросов из внешних макробиблиотечных файлов на уровне языка реализует директива `import from`.

Объявление внешних макросов с помощью директивы импорта предписывает ассемблеру включить определения указанных макросов из некоторой внешней макробиблиотеки. *`имя_библиотеки`* считается чистым именем файла макробиблиотеки; оно должно быть задано без пути и может быть задано без расширения, если расширение файла библиотеки не отличается от стандартного (`.mlb`).

Список включаемых макросов может быть опущен, в этом случае включаются все макросы указанной библиотеки.

Примеры объявлений внешних макросов:

```
import mode_constants  from com_decl.mlb;

import mode_constants, irqtab_layout  from
com_decl.mlb;

import  from com_decl.mlb;
```

Здесь первая директива включает из библиотеки `com_decl.mlb` один, указанный, макрос, вторая - два макроса, третья - все макросы библиотеки.

Поиск макробиблиотек производится сначала в текущем каталоге, а затем, в порядке следования, в каталогах, указанных ключами командной строки ассемблера `-I`.

Примечание: *Информация о порядке создания макробиблиотек с помощью специального режима работы ассемблера, а также о параметрах вызова, дана в документе: ЮФКВ.30047-01 95 01.Кросс-средства разработки программ. Руководство пользователя.*

Примечание: *Возможна ситуация, когда в новой версии ассемблера макросы из созданной в предыдущей версии макробиблиотеки будут интерпретироваться некорректно, в этом случае необходимо перекомпилировать макробиблиотеку.*

3 Регистры

В данном разделе рассматриваются три группы регистров:

- адресные регистры;
- регистры общего назначения;
- специальные регистры;
- векторные регистры.

Адресные и регистры общего назначения образуют группу основных регистров.

Специальные регистры могут относиться к определённой версии архитектуры

3.1 Основные регистры

К основным регистрам процессора относятся адресные регистры и регистры общего назначения, то есть те, которые используются в большинстве вычислительных операций процессора.

Всего имеется 8 адресных регистров и 8 регистров общего назначения (см. Табл. 3-1). Все они 32-х разрядные, доступны как по чтению, так и по записи.

Табл. 3-1 Основные регистры

АДРЕСНЫЕ РЕГИСТРЫ	РЕГИСТРЫ ОБЩЕГО НАЗНАЧЕНИЯ
ar0	gr0
ar1	gr1
ar2	gr2
ar3	gr3
ar4	gr4
ar5	gr5
ar6	gr6
ar7(sp)	gr7

3.1.1 Адресные регистры

Адресные регистры могут использоваться только в левой части ассемблерной инструкции (см. подразделы 5.1 Скалярные инструкции и 5.2 Векторные инструкции).

Примеры использования адресных регистров:

```
ar0 = ar5;                          // копирование.
ar2 = ar3 + gr3;                    // модификация.
[ar4++] = gr7 with gr7 -= gr4;      // запись в память.
```

Адресный регистр `ar7` используется процессором в качестве указателя стека адресов возврата `sp(Stack Pointer)`. Это обозначает, что `ar7` модифицируется автоматически, когда происходит вызов функции или прерывания, возврат из функции, из прерывания.

3.1.2 Регистры общего назначения

Регистры общего назначения в отличие от адресных, могут использоваться как в левой, так и в правой частях ассемблерной инструкции. С их помощью можно выполнять арифметические и логические преобразования, адресоваться по памяти.

Хотя регистры общего назначения могут использоваться для адресации по памяти, например:

```
[gr0] = gr4;  // запись значения регистра gr4 в память
              // по адресу, хранящемуся в gr0.
```

однако адресные регистры обладают в этом смысле значительно более широкими возможностями.

Примеры использования регистров общего назначения:

```
gr0 = gr5;                          // копирование.
gr2 = gr1 + gr3;                    // модификация.
[ar4++] = gr7 with gr7 -= gr4;      // запись в память.
```

Примечание

В процессоре не предусмотрены специальные регистры для организации программных циклов.

3.1.3 Регистровые пары

Каждому адресному регистру поставлен в соответствие регистр общего назначения с тем же номером. Таким образом образуются пары, которые в дальнейшем будут называться регистровыми парами.

Регистровые пары часто используются в инструкциях процессора, осуществляющих копирование, чтение/запись в память 64-х разрядных данных, некоторые виды адресных операций.

Приведем пример загрузки из памяти в регистровую пару 64-х разрядного значения:

```
ar0, gr0 = [ar1++];  // чтение 64-х разрядного слова.
```

В операциях чтения или записи регистровой пары порядок записи регистров несущественен:

```
gr0, ar0 = [ar1++]; // тоже допустимо
```

Еще один пример использования регистровых пар встречается в некоторых методах адресации, например:

```
[ar0+=gr0] = gr4; // запись в память с модификацией.
```

Непарные регистры не могут использоваться в подобных операциях. Иными словами:

```
ar0, gr1 = [ar1++]; // неправильная инструкция.
[ar0+=gr1] = gr4;   // неправильная инструкция.
```

3.2 Специальные регистры

pc - программный счетчик, в отличие от nm6403, дает точно предсказуемое значение, в частности, по нему можно адресоваться.

pswr - регистр слова состояния процессора. Кроме присваивания, к регистру можно применять операции установки(set) и сброса(clear) битов. Примеры: pswr set 0000_0200h; pswr clear 0000_0200h;

intr – регистр запросов на прерывание. Регистр можно читать или применять операцию сброса битов. Пример: intr clear 0000_0100h;

3.2.1 Специальные регистры NMC4

В данной архитектуре управление периферийными устройствами осуществляется не через специальные регистры, а командами псевдо- записи\чтения из памяти. В связи с этим специальных регистров много меньше, по сравнению с предыдущими версиями архитектуры.

3.2.1.1 Регистр sir

В nmc4 нельзя выполнять прямые пересылки констант и значений регистров в регистры сопроцессоров. (Допускаются пересылки между регистрами данного сопроцессора, но регистры сопроцессора фиксированной точки недоступны для чтения.)

Единственный регистр свободный от этого ограничения – регистр системного интегратора sir, в него можно писать константы и пересылать содержимое (доступных по чтению) регистров, а его значение пересылать в любой регистр (доступный по записи).

При портировании кода замена может быть выполна так:

```
//  код nmc3
nb1= 1000h;
sb= gr7;
//  соответствующий код nmc4
sir= 1000h;
```

```
nb1= sir;
sir= gr7;
sb= sir;
```

3.3 Специальные регистры NM6403

Примечание

Информация в этой секции как правило относится исключительно к NM6403. Описание специальных регистров более современных устройств NeuroMatrix можно найти в #3.3 и в описании архитектуры.

Процессор NM6403 содержит большое количество специальных регистров, которые позволяют осуществлять управление отдельными вычислительными, коммуникационными, коммутационными блоками, входящими в его состав.

Далее приводится список специальных регистров процессора в форме таблицы (см. Табл. 3-2), а затем даются комментарии по их использованию и правила работы с ними.

Табл. 3-2 Сводная таблица специальных регистров процессора NM6403.

НАИМЕНОВАНИЕ	ОПИСАНИЕ	ПРИМЕЧАНИЕ
gmicr	Регистр управления интерфейсом с глобальной шины.	32 бита.
ica0, icc0	Регистр адреса и счетчик данных канала приема через нулевой коммуникационный порт.	Парные регистры, 32 бита каждый.
ica1, icc1	Регистр адреса и счетчик данных канала приема через первый коммуникационный порт.	Парные регистры, 32 бита каждый.
intr	Регистр запросов на прерывание и прямой доступ к памяти (ПДП).	32 бита.
lmicr	Регистр управления интерфейсом с локальной шины.	32 бита.
oca0, occ0	Регистр адреса и счетчик данных канала передачи через нулевой коммуникационный порт.	Парные регистры, 32 бита каждый.
oca1, occ1	Регистр адреса и счетчик данных канала передачи через первый коммуникационный порт.	Парные регистры, 32 бита каждый.
pc	Счётчик команд.	32 бита.
pswr	Регистр слова состояния процессора.	32 бита.
t0, t1	Регистры управления таймерами.	32 бита.

dorj, dirj	Используются как буферные регистры при работе комм. портов. (Программно доступны, но возможность их практического применения в пользовательской программе весьма сомнительна.)	64 бита.

3.4 Специальные регистры NMC3

3.4.1 Регистры процессорного ядра NMC3

pcr - регистр управления периферией. Кроме присваивания, к регистру можно применять операции установки и сброса битов. Следует помнить о том, что для разных устройств, построенных на базе процессорного ядра NMC3, семантика использования регистра различается, поэтому используйте псевдонимы pcr_6405 или pcr_soc.

f1cr, f1crh, f1crl - регистры управления функцией насыщения операнда X (целый, старшая, младшая часть)

f2cr, f2crh, f2crl - регистры управления функцией насыщения операнда Y (целый, старшая, младшая часть)

vr, vrh, vrl - регистр порога (целый, старшая, младшая часть)

nb1, nb1h, nb1l - регистр границ нейронов (целый, старшая, младшая часть). Единица в имени – артефакт

sb, sbh, sbl - регистр границ синапсов (целый, старшая, младшая часть)

3.4.2 О переферийных регистрах NMC3

Начиная с NM6405 подсистема управления переферией вынесена за пределы процессорного ядра, что делает возможным появление чипов на базе одного ядра но с различной переферийной частью. С точки зрения языка ассемблера, различие выражается в разных наборах регистров управления переферией.

Другая важная особенность – появление регистровых окон. Разработчики аппаратуры столкнулись с нехваткой кодов для переферийных регистров, в результате один и тот же код соответствует двум и более регистрам, в зависимости от контекста. Переключение контекста осуществляется манипуляциями с 9 и 10 битами pswr.

К сожалению, нет универсального способа статически определить текущее регистровое окно, поэтому контроль за соответствием регистра текущему окну возложен на программиста. Если допущена ошибка, и регистр используется в чужом окне, то ассемблер этого не заметит и в реальности будет задействован регистр-двойник из текущего окна.

Настоятельно рекомендум придерживаться следующей конвенции: считать нулевое окно окном по-умолчанию, а при работе с первым

переключать контекст туда и обратно, причем как можно ближе к месту использования переферийного регистра.

Есть способ обращаться к коду периферийного регистра безотносительно регистрового окна и ревизии процессора, для этого выделены специальные имена pr0 – pr15. При трансляции такого имени ассемблер не производит никаких проверок.

3.4.3 Периферийные регистры NM6405

Нулевое регистровое окно

t0, t1 – значение таймера с соответствующим номером

irr – регистр запросов на внешние прерывания. Регистр можно читать или применять операцию сброса битов.

imr – регистр масок на внешние прерывания. Кроме присваивания, к регистру можно применять операции установки и сброса битов.

lema - регистр адреса локальной внешней памяти каналов ПДП

gema - регистр адреса глобальной внешней памяти каналов ПДП

lima - регистр адреса локальной внутренней памяти каналов ПДП

gima - регистр адреса глобальной внутренней памяти каналов ПДП

dmac0, dmac1 – счетчики каналов ПДП

iop – порт общего назначения. Кроме присваивания, к регистру можно применять операции установки и сброса битов.

iopcr - регистр управления портом общего назначения

Первое регистровое окно

oca0, oca1 – регистры адреса канала вывода коммуникационных портов

ica0, ica1 – регистры адреса канала ввода коммуникационных портов

occ0, occ1 – счетчики канала вывода коммуникационных портов

icc0, icc1 – счетчики канала ввода коммуникационных портов

gpa – регистр адреса гиперстраницы кеш команд

lmcr0, lmcr1 - регистры управления локальной внешней шиной, доступны только на запись

gmcr0,gmcr1 - регистры управления глобальной внешней шиной, доступны только на запись

Подробное описание специальных регистров NM6405 см. в описании архитектуры процессора NM6405.

3.4.4 Мнемоники периферийных регистров СБИС ЦУПП

nmc3 - имя	Регистр - источник	Регистр - приемник	Регистр - источник	Регистр - приемник

pr0	**tmr_count0**	**tmr_count0**		**punit0**
pr1	**tmr_count1**	**tmr_count0**		**punit1**
pr2	**tmr_mode0**	**tmr_mode0**		
pr3	**tmr_mode1**	**tmr_mode1**		
pr4	**intr_mask**	**intr_mask**	-	-
pr5	**intr_req**	**clear intr_req**	-	-
pr6	**intr_req**	**set intr_req**	-	-
pr7	**nmscu(nmcscr)**	**nmscu(nmcscr)**		
pr8				**punit2**
pr9				**punit3**
pr10				**punit4**
pr11				**punit5**
pr12				**punit6**
pr13				**punit7**
pr14				**punit8**
pr15				**punit9**

Данные имена являются синонимами pr0-pr15, но более информативны и позволяют частично проконтролировать контекст использования регистра.

3.5 Векторные регистры сопроцессора фиксированной точки

Векторный сопроцессор для работы с числами с фиксированной точкой содержит векторные регистры, которые позволяют осуществлять управление ВП, хранить промежуточные данные, получаемые в процессе счета.

Векторные регистры делятся на две группы. В первую входят регистры управления ВП, определяющие конфигурацию рабочей и теневой матриц, векторного АЛУ, функций активации. Во вторую входят так называемые "регистры-контейнеры", внутренние блоки памяти ВП, работающие по принципу FIFO. Все регистры-контейнеры имеют одинаковую глубину. Они рассчитаны максимально на 32 слова по 64 бита каждое.

Далее приводится список векторных регистров процессора в форме таблицы Табл. 3-15, а затем даются комментарии по их использованию и правила работы с ними.

Табл. 3-3 Сводная таблица векторных регистров процессора NM6403.

НАИМЕНОВАНИЕ	ОПИСАНИЕ	ПРИМЕЧАНИЕ
f1cr,f2cr	Регистры управления функцией активации. Имеют разрядность 64 бита каждый, разрешена запись значений отдельно в младшую и старшую части регистров.	64 бита. Доступны только по записи.
nb1	Регистр границ нейронов. Имеет разрядность 64 бита, разрешена запись	64 бита. Доступен только

	значений отдельно в младшую и старшую части регистра.	по записи.
sb	Регистр границ синапсов. Имеет разрядность 64 бита, разрешена запись значений отдельно в младшую и старшую части регистра.	64 бита. Доступен только по записи.
vr	Регистр порога. Имеет разрядность 64 бита, разрешена запись значений отдельно в младшую и старшую части регистра.	64 бита. Доступен только по записи.
afifo	Регистр-контейнер для хранения результата выполнения любой операции на векторном узле процессора NM6403.	32x64 бита.
data	Логический регистр-контейнер, описывает данные, проходящие в данный момент по шине (глобальной или локальной) данных в процессе их загрузки из памяти в ВП.	Отображается на шину данных, поэтому имеет такой объем, какое кол-во слов загружается из памяти командой, которая его использует.
ram	Регистр-контейнер для хранения данных, которые могут повторно использоваться в процессе вычислений.	32x64 бита.
wfifo	Регистр-контейнер для хранения весовых коэффициентов, которые затем загружаются в теневую матрицу ВП.	32x64 бита.

3.5.1 Регистры f1cr и f2cr

Регистры `f1cr` и `f2cr` имеют разрядность 64 бита каждый. Они доступны для записи и используются для управления конфигурацией операндов при выполнении кусочно-линейных преобразований над входными данными векторного процессора.

Регистры `f1cr` и `f2cr` недоступны по чтению. Попытка чтения из `f1cr` или `f2cr` воспринимается ассемблером, как синтаксическая ошибка.

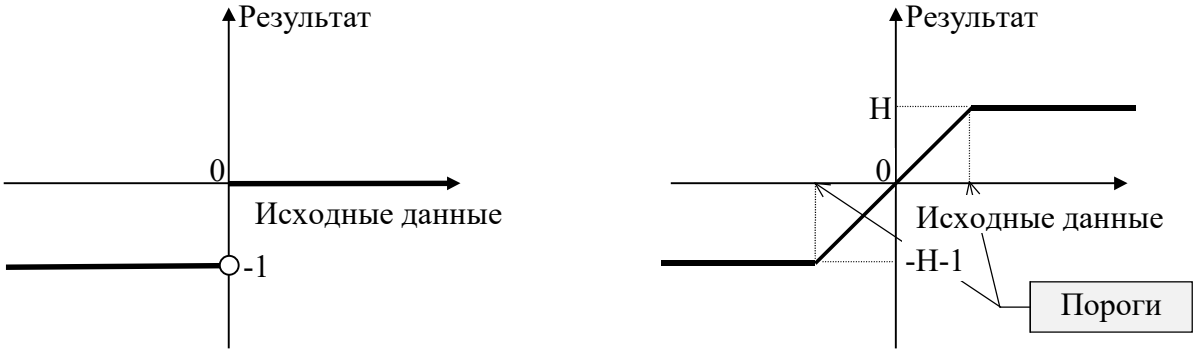
Данные, поступающие на входы **X** и **Y** ВП, предварительно на проходе могут быть подвергнуты преобразованию кусочно-линейными функциями, называемыми функциями активации. Это происходит, если в коде векторной команды использовано ключевое слово

activate. Правила вызова той или иной функции активации описано в подпункте 1.5.5 Обработка данных функцией активации.

Всего существует два типа функций активации:

- пороговая функция (см. Рис. 3-1а);
- функция насыщения (см. Рис. 3-1б).

Рис. 3-1 Типы встроенных функций активации процессора NM6403.



а) Пороговая функция *б) Функция насыщения*

Регистры f1cr и f2cr играют основную роль в определении порогов функций активации.

Регистр f1cr задает пороги функций активации для данных, поступающих на вход **X** ВП, а f2cr для данных входа **Y**.

Регистры f1cr и f2cr делят 64-х разрядные слова входных данных на элементы. Над каждым элементом выполняется функция активации, то есть активации подвергаются одновременно и независимо все элементы, составляющие длинное слово.

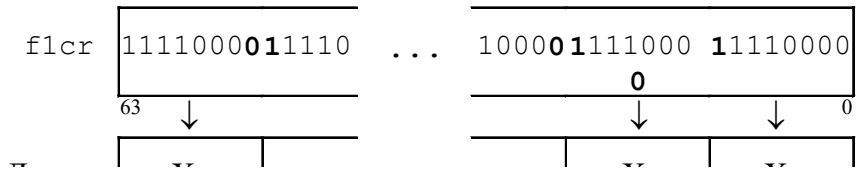
Разбиение на элементы, записанное в f1cr и f2cr, может не совпадать с тем, которое задается на входах **X** и **Y** регистрами sb2 и nb2 соответственно при операциях умножения с накоплением, или регистром nb2 при выполнении вычислений на векторном АЛУ. Однако в большинстве случаев программист устанавливает разбиение в f1cr и f2cr таковым, чтобы оно совпадало с разбиением, заданным регистрами nb2, sb2.

Разбиение данных на элементы при помощи f1cr (f2cr)

Далее в описании будет использоваться регистр f1cr, однако все сказанное ниже, если это специально не оговорено, справедливо и для f2cr.

Разбиение на элементы задается в регистре f1cr путем перепада значений соседних битов с 1 в 0 при движении от младших битов к старшим (см. Рис. 3-2).

Рис. 3-2 Разбиение 64-х битного слова на элементы при помощи f1cr (f2cr).



Поскольку деление на элементы происходит только при переходе от 1 к 0, наименьший размер элемента составляет 2 бита (у элемента самый младший бит должен быть равен 0, а самый старший 1).

Последовательность нулей и единиц в `f1cr` произвольна, а это значит, что длинное слово данных может быть разбито на произвольное количество элементов (в пределах от 1 до 32) произвольной разрядности с суммарным количеством битов, равным 64.

Использование f1cr(f2cr) в функции насыщения (арифметическая активация)

В режиме арифметической активации, когда данные подвергаются преобразованию при помощи функции насыщения (см. Рис. 3-1б), важную роль играют значения битов регистра `f1cr`, расположенных в пределах элемента данных.

Имеется в виду, что есть регистр `f1cr` и есть входные данные, поступающие на вход **X** ВП. Слова входных данных, также как и `f1cr` имеют разрядность 64 бита, то есть каждому биту слова данных можно сопоставить бит регистра `f1cr`.

Как уже было сказано, переход от 1 к 0 в `f1cr` ведет к разделению слова входных данных на элементы. Дальнейшие рассуждения относятся к битам регистра `f1cr`, соответствующим полю элемента данных.

Самому старшему биту поля элемента всегда соответствует единица в `f1cr` (см. Рис. 3-2). Количество единиц в `f1cr` вправо от него определяет порог насыщения, что иллюстрирует Табл. 3-16.

Все поведение функции насыщения определяется тем, какие значения имеют биты поля элемента данных в тех позициях, которые соответствуют единичным битам регистра `f1cr`. Если все биты элемента данных, находящиеся в этих позициях, равны нулю, то значение элемента данных положительно и меньше установленного порога.

Табл. 3-4 Пороги для 8-битных данных.

БИТЫ F1CR	ПОРОГ
10000000_2	127
11000000_2	63
11100000_2	31
11110000_2	15

11111000_2	7
11111100_2	3
11111110_2	1

Возможные значения порогов определяются формулой: $H = 2^n - 1$.

В качестве примера рассматривается 8-ми разрядный элемент данных. В случае, когда три старших бита регистра f1cr, соответствующие полю этого элемента, равны единице, остальные нулю, можно наблюдать следующее:

f1cr: ...**111**00000... ← *верхний порог равен* 31(0x1F)
поле данных: ...**000**10110... ← *значение элемента* 22(0x16)
после активации: ...00010110... ← *значение элемента* 22(0x16)

Значение элемента данных равно 22, что меньше, чем 31. Поэтому данные будут переданы на выход функции активации без изменений.

Если все три старших бита значения элемента данных равны единице, то:

f1cr: ...**111**00000... ← *нижний порог равен* -32(0xE0)
поле данных: ...**111**10110... ← *значение элемента* -10(0xF6)
после активации: ...11110110... ← *значение элемента* -10(0xF6)

Значение элемента данных равно -10, что больше, чем -32. В этом случае данные также будут переданы на выход функции активации без изменений.

В случае, когда биты поля элемента, соответствующие единичным битам f1cr, имеют не одинаковые значения, различаются два случая. Первый - старший бит поля элемента равен 0 (положительное значение), второй - старший бит поля элемента равен 1 (отрицательное значение).

В первом случае арифметическая функция активации осуществляет следующее преобразование:

f1cr: ...**111**00000... ← *верхний порог равен* 31(0x1F)
поле данных: ...**010**10110... ← *значение элемента* 86(0x56)
после активации: ...00011111... ← *значение элемента* 31(0x1F)

Значение элемента данных равно 86, что больше, чем 31. При этом функция активации заменит значение элемента на положительное пороговое.

Во втором случае арифметическая функция активации осуществляет следующее преобразование:

f1cr: ...**111**00000... ← *нижний порог равен* -32(0xE0)
поле данных: ...**110**10110... ← *значение элемента* -42(0xD6)
после активации: ...11100000... ← *значение элемента* -32(0xE0)

Значение элемента данных равно -42, что меньше, чем -32. В этом случае на выходе функции активации значение будет заменено на отрицательное пороговое.

Итак, если значения битов поля данных, соответствующих единичным битам регистра `f1cr`:

одинаковы, то поле данных не изменяется при обработке арифметической функцией активации (функцией насыщения);

не одинаковы и старший бит равен 0, то значение заменяется на положительное пороговое;

не одинаковы и старший бит равен 1, то значение заменяется на отрицательное пороговое.

Использование f1cr(f2cr) в пороговой функции (логическая активация)

В режиме логической активации, когда данные подвергаются преобразованию при помощи пороговой функции (см. Рис. 3-1а), важную роль играет значение старшего бита регистра `f1cr`, расположенного в пределах элемента данных.

Имеется в виду, что есть регистр `f1cr` и есть входные данные, поступающие на вход **X** ВП. Слова входных данных, также как и `f1cr` имеют разрядность 64 бита, то есть каждому биту слова данных можно сопоставить бит регистра `f1cr`. Переход от 1 к 0 в `f1cr` ведет к разделению слова входных данных на элементы.

Самому старшему (знаковому) биту поля элемента всегда соответствует единица в `f1cr` (см. Рис. 3-2). Значение этого поля является определяющим при обработке входных данных логической функцией активации (пороговой функцией). Таким образом, для логической активации важно только то, как слова входных данных делятся на элементы при помощи `f1cr`. Значения битов `f1cr`, не участвующих в разбиении роли не играют.

В качестве примера рассматривается 8-ми разрядный элемент данных. Результат обработки логической функцией активации зависит от знака обрабатываемого числа. Для положительных чисел:

`f1cr:` `...0|10000000|1...`
поле данных: `.....00010110...` ← *значение* `22(0x16)`
после активации: `.....00000000...` ← *результат* `0`.

Значение элемента данных равно `22` (положительное). Поэтому результат обработки логической функцией активации равен `0`.

Для отрицательных чисел:

`f1cr:` `...0|10000000|1...`
поле данных: `.....10010110...` ← *значение* `-106(0x96)`
после активации: `.....11111111...` ← *результат* `-1(0xFF)`.

Значение элемента данных равно `-106` (отрицательное). Поэтому результат обработки логической функцией активации равен `-1`.

Итак, логическая функция активации (пороговая функция) переводит неотрицательные значения элементов полей данных в `0`, отрицательные в `-1`.

Загрузка значений в регистр f1cr (f2cr)

Все сказанное в данном пункте относится в одинаковой степени как к регистру `f1cr`, так и к `f2cr`.

Прежде всего, необходимо помнить, что `f1cr` - это 64-х разрядный регистр. Система команд процессора NM6403 не содержит кода команды, который бы позволил напрямую разгрузить в регистр 64-х разрядную константу. Однако существует несколько косвенных способов загрузки значений в регистр `f1cr`.

Способ 1. Загрузка по частям.

Процессор NM6403 позволяет осуществлять доступ к регистру `f1cr` по частям. То есть отдельно может быть загружена старшая часть регистра, отдельно младшая. Для этого в язык ассемблера введены специальные символы:

- `f1crh` - старшая часть регистра `f1cr`;
- `f1crl` - младшая часть регистра `f1cr`.

Приведем пример загрузки регистра `f1cr` по частям(код для nmc4_fixed):

```
sir   = 80808080h;   // загрузка младшей части f1cr.
f1crl = sir;
sir   = 40404040h;   // загрузка старшей части f1cr.
f1crh = sir;
```

В результате в регистр `f1cr` загружено число `40404040808080hl`.

Способ 2. Загрузка одинаковой младшей и старшей частей.

Если в младшую и старшую половины регистра необходимо загрузить одно и то же число, то можно использовать следующую команду:

```
sir  = 80808080h; // загрузка одинаковых констант
                  // в старшую и младшую части f1cr.
f1cr = sir;
```

При этом, процессор автоматически запишет эту константу в обе половины регистра, то есть, после выполнения команды в регистре `f1cr` будет находиться число: `8080808080808080hl`.

Способ 3. Загрузка в регистр содержимого памяти.

Регистр `f1cr` можно инициализировать 64-х разрядной константой, расположенной в памяти. Приведем пример:

```
data ".data"
    MyF1CR: long = 0123456789ABCDEFhl;
    ...
end ".data";
...
begin ".text"
    ...
```

```
    sir  = [MyF1CR]; // загрузка в f1cr константы из памяти.
    f1cr = sir;
...
end ".text";
```

В результате в регистр `f1cr` загружено `0123456789ABCDEFh1`.

Загрузку константы из памяти возможно осуществить при помощи одной команды, поскольку процессор автоматически определяет, что данные загружаются в 64-х разрядный регистр, поэтому будет подкачено сразу две соседних ячейки памяти, начиная с четного адреса.

В таблице Табл. 3-17 приведены наиболее часто встречающиеся значения, записываемые в регистр `f1cr` при использовании пороговой функции активации. Предполагается, что 32-х разрядная константа заносится в `f1cr` способом 2, приведенным выше, а 64-х разрядная считывается из памяти.

Табл. 3-5 Часто используемые при логической активации константы разбиения для регистра f1cr(f2cr).

РАЗРЯДНОСТЬ ЭЛЕМЕНТА	КОЛИЧЕСТВО ЭЛЕМЕНТОВ В СЛОВЕ	ЗНАЧЕНИЕ КОНСТАНТЫ В F1CR (F2CR)
64 бита	1	`f1crh = 80000000h`
32 бита	2	`80000000h`
21 бит	3	`4000020000100000h1`
16 бит	4	`80008000h`
10 бит	6	`0802008020080200h1`
8 бит	8	`80808080h`
4 бита	16	`88888888h`
2 бита	32	`0AAAAAAAAh`

3.5.2 Конфигурационные регистры векторного сопроцессора

Результат вычислений на векторном процессоре зависит не только от последовательности команд векторного сопроцессора и векторных данных, загружаемых из памяти в векторные регистры, но и от дополнительной конфигурации, осуществляемой через специальные 32 или 64-разрядные регистры

В случае векторного сопроцессора фиксированной точки так задаются например разбиения битовых строк на числа-элементы.

Ранее, для версий архитектуры NeuroMatrix nmc3 и более ранних, конфигурационные регистры можно было читать и писать напрямую командами копирования из любых регистров, из памяти или установкой константы:

```
nb1  = 80808080h;
```

```
nb1l = gr1;
```

К сожалению, в версии nmc4 писать напрямую нельзя, приходится использовать регистр-посредник.

```
sir  = 80808080h;
nb1l = sir;
sir  = gr1;
nb1l = sir;
```

3.5.3 Регистр nb1(nb2)

Регистр `nb1` имеет разрядность 64 бита. Он доступен по записи из ассемблера и играет следующую роль в работе векторного процессора:

- определяет разбиение матрицы векторного процессора на столбцы;
- определяет разбиение на независимые элементы 64-х разрядных слов, участвующих в арифметических и логических операциях на АЛУ векторного процессора.

Регистр `nb1` не доступен по чтению. Попытка прочитать состояние регистра путем прямого обращения к нему приведет к синтаксической ошибке.

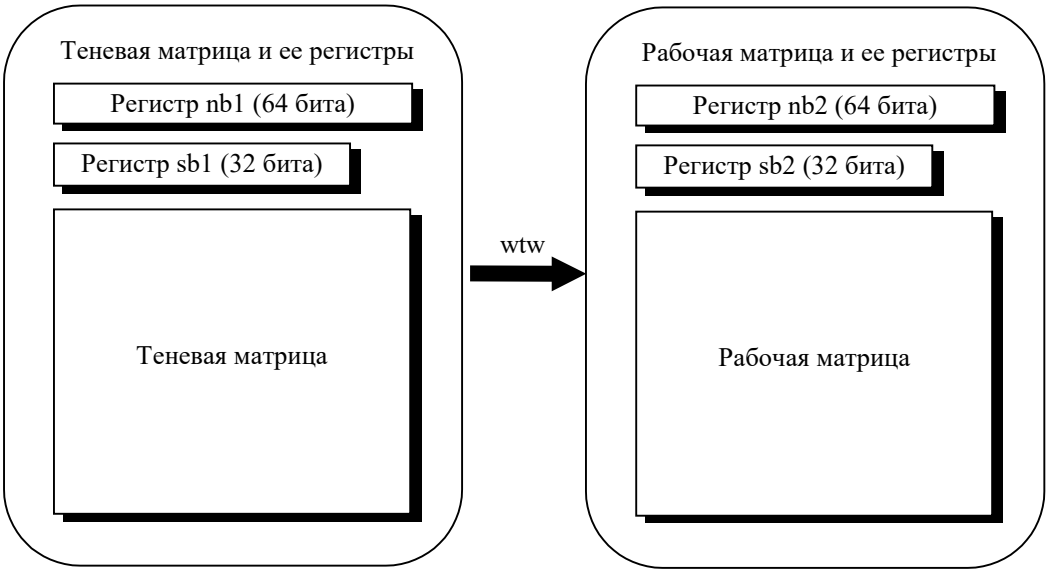
Прежде чем рассматривать его структуру и варианты использования, необходимо пояснить то, какое место занимает он в структуре векторного процессора NM6403.

Векторный процессор имеет в своем составе две операционных матрицы: рабочую и теневую. Рабочая матрица используется непосредственно для вычислений, а теневая для подготовки очередной порции весовых коэффициентов, которая будет использована на последующем шаге вычислений. Информация из теневой матрицы в рабочую переписывается за один процессорный такт (см. описание команды `wtw`). Такая организация векторного процессора позволяет одновременно вести вычисления и заниматься загрузкой очередной порции входных данных.

Каждая матрица сопровождается своей парой регистров:

- `nb` - определяет разбиение матрицы на столбцы;
- `sb` - определяет разбиение на строки.

Рис. 3-3 Теневая и рабочая матрицы векторного процессора.



Для того, чтобы отличать пары регистров теневой и рабочей матрицы, они помечены цифрами. Так регистры теневой матрицы отмечены как `nb1` и `sb1`, а регистры рабочей - `nb2` и `sb2`.

Описание и правила использования регистров `sb1` и `sb2` приводятся в п. 3.4.3, здесь же далее будет дано описание регистров `nb1` и `nb2`.

Из программы на языке ассемблера доступен только регистр `nb1`. Он доступен только по записи. Чтение из `nb1` невозможно. Попытки чтения воспринимаются ассемблером, как синтаксические ошибки.

Регистр `nb1`, как уже было отмечено, является принадлежностью теневой матрицы. Для того, чтобы изменить значение регистра рабочей матрицы `nb2`, необходимо:

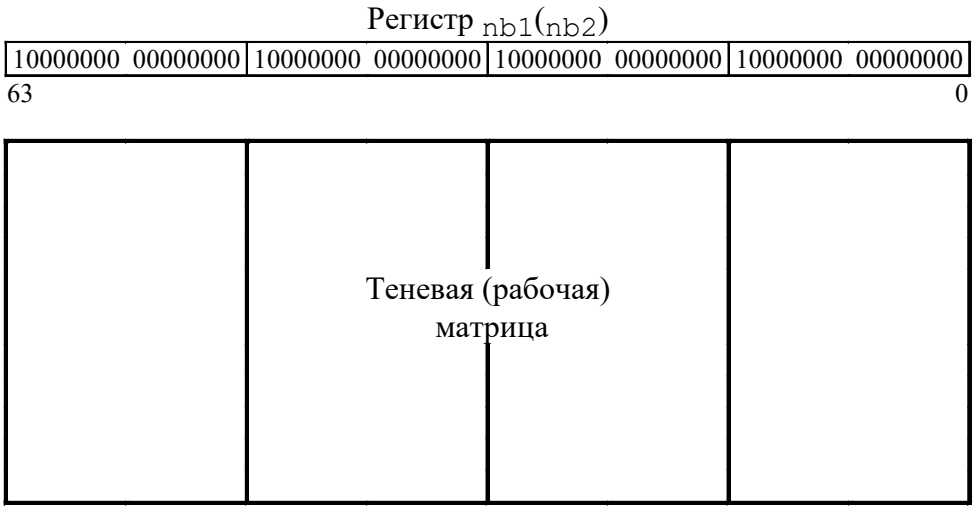
- сначала загрузить его в регистр `nb1`,
- загрузить теневую матрицу командой `ftw`
- при помощи команды `wtw`, осуществляющей копирование содержимого теневой матрицы в рабочую, переписать в `nb2`.

Использование регистра nb1(nb2)

Регистр `nb1`(`nb2`) управляет разбиением теневой (рабочей) матрицы ВП на независимые вычислительные элементы и определяет конфигурацию входа **`Y`**. Он также задает границы элементов при выполнении операций на векторном АЛУ. Его разрядность 64 бита.

Границы между элементами внутри слова в общем случае задаются перепадом значения соответствующих граничных битов с единицы в ноль, если двигаться в направлении увеличения разрядности. Единицей помечается самый старший разряд элемента (см. Рис. 3-4).

Рис. 3-4 Разбиение матрицы на столбцы регистром nb1(nb2).



Крайний из возможных случаев - все биты регистра равны единице. Это означает, что каждый бит является старшим битом элемента разбиения, то есть разрядность всех элементов равна единице.

Диапазон возможных размерностей элементов, определяемых регистром `nb1(nb2)`, составляет от 1 бита до 64.

Возможно также разбиение слова на элементы нерегулярным образом, то есть элементы, составляющие слово, могут иметь различную разрядность. Например, запись в регистр `nb1` следующего значения(код для nmc4_fixed):

```
data "NB"
    NB1: long = 8000000080008000hl;
end "NB";

begin "text"
    sir = [NB1]; // присвоение значения регистру nb1.
    nbl = sir;
     ...
    wtw;
    ...
end "text";
```

определяет разбиение 64-х разрядных слов, поступающих на вход **x** векторного узла, на один 32-х разрядный элемент и два 16-ти разрядных.

При отсутствии необходимости разбиения матрицы на столбцы в регистр `nb1` может быть записан 0. Процессор в этом случае автоматически выставит границу элемента в 63-ем разряде.

Загрузка значений в регистр nb1

Прежде всего, необходимо помнить, что `nb1` - это 64-х разрядный регистр. Система команд процессора NM6403 не содержит кода команды, который бы позволил напрямую разгрузить в регистр

64-х разрядную константу. Однако существует несколько косвенных способов загрузки значений в регистр nb1.

Способ 1. Загрузка по частям.

Процессор NM6403 позволяет осуществлять доступ к регистру nb1 по частям. То есть отдельно может быть загружена старшая часть регистра, отдельно младшая. Для этого в язык ассемблера введены специальные символы:

- nb1h - старшая часть регистра nb1;
- nb1l - младшая часть регистра nb1.

Приведем пример загрузки регистра nb1 по частям:

```
sir  = 80808080h;   // загрузка младшей части nb1.
nb1l = sir;
sir  = 40404040h;   // загрузка старшей части nb1.
nb1h = sir;
```

В результате в регистр nb1 загружено число 40404040808080hl.

Способ 2. Загрузка одинаковой младшей и старшей частей.

Если в младшую и старшую половины регистра необходимо загрузить одно и то же число, то можно использовать следующую команду:

```
sir = 80808080h;  // загрузка одинаковых констант
                  // в старшую и младшую части nb1.
nb1 = sir;
```

При этом процессор автоматически запишет эту константу в обе половины регистра, то есть, после выполнения команды в регистре nb1 будет находиться число: 8080808080808080hl.

Способ 3. Загрузка в регистр содержимого памяти.

Регистр nb1 можно инициализировать 64-х разрядной константой, расположенной в памяти. Приведем пример:

```
data ".data"
    MyNB1: long = 0123456789ABCDEFhl;
    ...
end ".data";
...
begin ".text"
    ...
    sir = [MyNB1]; // загрузка в nb1 константы из памяти.
    nb1 = sir;
    ...
end ".text";
```

В результате в регистр nb1 загружено число 0123456789ABCDEFhl.

Загрузку константы из памяти возможно осуществить при помощи одной команды, поскольку процессор автоматически определяет, что данные загружаются в 64-х разрядный регистр,

поэтому будет подкачено сразу две соседних ячейки памяти, начиная с четного адреса.

В таблице Табл. 3-18 приведены наиболее часто встречающиеся значения, записываемые в регистр nb1. Предполагается, что 32-х разрядная константа заносится в nb1 способом 2, приведенным выше, а 64-х разрядная считывается из памяти.

Табл. 3-6 Часто используемые константы разбиения для регистра nb1.

РАЗРЯДНОСТЬ ЭЛЕМЕНТА	КОЛИЧЕСТВО ЭЛЕМЕНТОВ В СЛОВЕ	ЗНАЧЕНИЕ КОНСТАНТЫ В NB1
64 бита	1	0
32 бита	2	80000000h
21 бит	3	4000020000100000hl
16 бит	4	80008000h
10 бит	6	0802008020080200hl
8 бит	8	80808080h
4 бита	16	88888888h
2 бита	32	0AAAAAAAAh
1 бит	64	0FFFFFFFFh

3.5.4 Регистр sb (sb1 и sb2)

Регистр sb имеет разрядность 64 бита. Он доступен по записи из ассемблера и используется для управления конфигурацией операционной матрицы ВП. С его помощью задается разбиение матрицы на строки.

Регистр sb недоступен по чтению. Попытка чтения из sb воспринимается ассемблером, как синтаксическая ошибка.

Структура регистра sb

Регистр sb, являясь скорее логическим, чем реальным, представляет собой совокупность регистров sb1 и sb2. Регистры sb1 и sb2 имеют разрядность 32 бита каждый.

Назначение sb1 и sb2 становится ясным из Рис. 3-3. Регистр sb1 является атрибутом теневой матрицы ВП и управляет ее разбиением на строки, а sb2 определяет разбиение на строки рабочей матрицы.

Однако из ассемблера напрямую недоступен ни sb1, ни sb2. Для того, чтобы записать в них значения, необходимо использовать sb.

Каждый из битов регистра sb принадлежит либо sb1, либо sb2. Биты, принадлежащие одному из регистров, расположены в sb не по порядку, а **перемешаны** с битами другого (см. Рис. 3-5.).

Рис. 3-5 Регистр sb и составляющие его регистры sb1 и sb2.

Нечетные биты регистра sb принадлежат регистру sb1 (на рисунке - белые), четные - регистру sb2(на рисунке - серые).

При записи значений в регистр sb, изменяются только биты регистра sb1(нечетные). Ассемблер не запрещает записывать новые значения в четные биты sb, однако это никак не влияет на состояние sb2.

Использование регистра sb (sb1, sb2)

Регистр sb является логической формой представления регистров sb1 и sb2. Он доступен из языка ассемблера и позволяет задать значение регистра sb1. Задание напрямую значения регистра sb1 невозможно в силу архитектурных ограничений процессора NM6403.

Как уже отмечалось, регистр sb1 управляет разбиением на строки теневой матрицы ВП. Установка sb должна предшествовать команде ftw, которая формирует теневую матрицу. sb2 определяет то же для рабочей матрицы и описывает конфигурацию ее входа **X**.

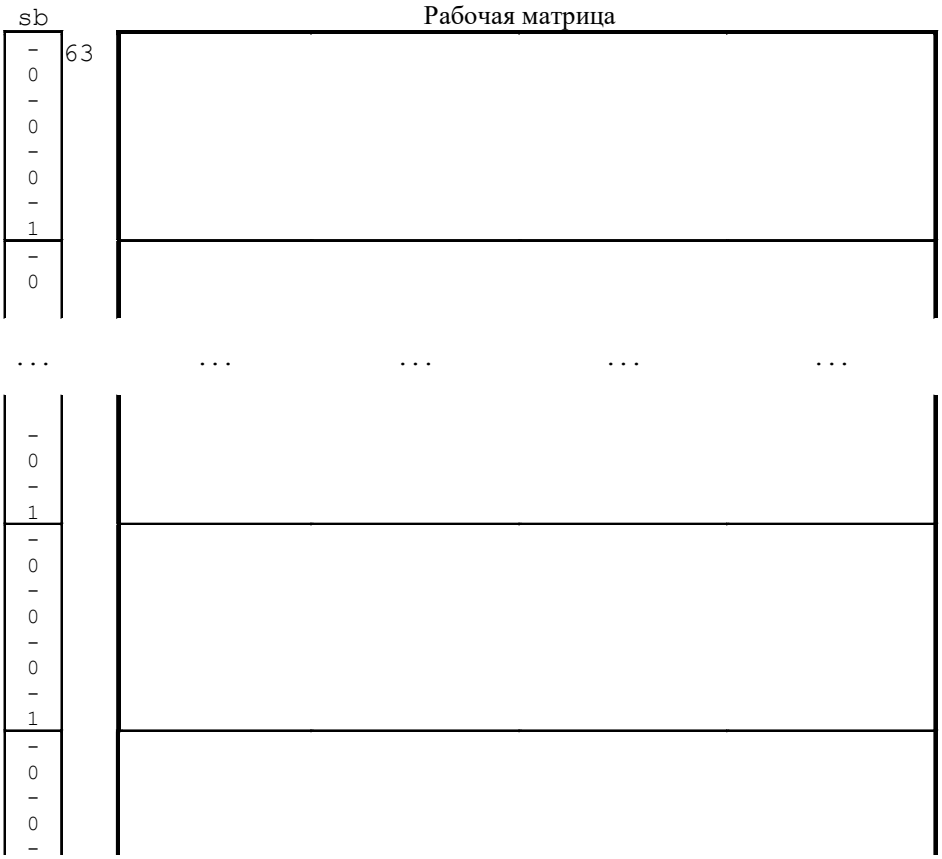
Значение в sb1 заносится путем записи в нечетные биты регистра sb. В sb2 содержимое sb1 копируется при выполнении команды wtw.

Регистр sb1(sb2) имеет разрядность 32 бита. Это означает, что каждый его бит управляет двумя битами данных, поступающих на вход **X** рабочей матрицы. В этом его отличие от регистра nb1(nb2), позволяющего управлять каждым битом входа **Y**.

Границы между элементами внутри слова в общем случае задаются перепадом значения соответствующих граничных битов регистра sb1(sb2) с нуля в единицу, если двигаться в направлении увеличения разрядности. Единицей помечается самый младший разряд элемента.

На Рис. 3-6 изображено, как именно происходит разделение рабочей матрицы на строки при помощи регистра sb(sb2). Биты, замененные знаком "-", принадлежат регистру sb1 и не влияют на разбиение рабочей матрицы.

Рис. 3-6 Разбиение рабочей матрицы на строки регистром sb(sb2).



Из Рис. 3-6 видно, как появилась логическая модель регистра `sb`, напоминающая "расческу", "зубцами" которой являются биты регистров `sb1` и `sb2`. Эта модель позволяет "растянуть" 32-х разрядные регистры на 64 бита.

Минимальная разрядность элементов, входящих в состав длинного слова, равна 2-м битам. Такое разбиение возникает при `sb2=0FFFFFFFFh`. Максимальная - 64 бита, при этом `sb2 = 1`. Если в регистр записан ноль, то процессор автоматически устанавливает младший бит в единицу.

Загрузка значений в регистр sb

Прежде всего, необходимо помнить, что `sb` - это 64-х разрядный регистр. Система команд процессора NM6403 не содержит кода команды, который бы позволил напрямую разгрузить в регистр 64-х разрядную константу. Однако существует несколько косвенных способов загрузки значений в регистр `sb`.

Способ 1. Загрузка по частям.

Процессор NM6403 позволяет осуществлять доступ к регистру `sb` по частям. То есть отдельно может быть загружена старшая часть регистра, отдельно младшая. Для этого в язык ассемблера введены специальные символы:

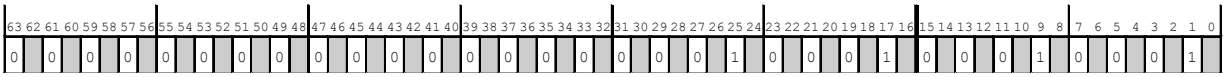
- `sbh` - старшая часть регистра `sb`;

- `sbl` - младшая часть регистра `sb`.

Приведем пример загрузки регистра `sb` по частям:

```
sir = 02020202h;    // загрузка младшей части sb.
sbl = sir;
sir = 05050505h;    // загрузка старшей части sb.
sbh = sir;
```

В результате в регистр `sb` загружается число `0505050502020202hl` – и получается значение вида:



Напомним, что до выполнения команды wtw нечетные биты регистра sb определяются текущим значением регистра sb2, непосредственно после выполнения команды wtw в регистре sb окажется значение 0000000003030303hl.

Способ 2. Загрузка одинаковой младшей и старшей частей.

Если в младшую и старшую половины регистра необходимо загрузить одно и то же число, то можно использовать следующую команду:

```
sir = 03030303h;    // загрузка одинаковых констант
                    // в старшую и младшую части sb.
sb  = sir;
```

При этом процессор автоматически запишет эту константу в обе половины регистра, то есть, после выполнения команды в регистре `sb` будет (после выполнения команды wtw) находиться число: `0303030303030303hl`.

Способ 3. Загрузка в регистр содержимого памяти.

Регистр `sb` можно инициализировать 64-х разрядной константой, расположенной в памяти. Приведем пример:

```
data "Секция_данных"
    MySB: long = 0123456789ABCDEFhl;
    ...
end "Секция_данных";
...
begin "Секция_кода"
    ...
    sir = [MySB];
    sb  = sir;    // загрузка в sb константы из
                  // из памяти.
    ...
end "Секция_кода";
```

В результате в регистр sb (после выполнения команды wtw) загружено число 00330033CCFFCCFFh1.

Загрузку константы из памяти возможно осуществить при помощи одной команды, поскольку процессор автоматически определяет, что данные загружаются в 64-х разрядный регистр, поэтому будет подкачено сразу две соседних ячейки памяти, начиная с четного адреса.

В Табл. 3-19 приведены наиболее часто встречающиеся значения, записываемые в регистр sb. Предполагается, что 32-х разрядная константа заносится в sb способом 2, приведенным выше, а 64-х разрядная считывается из памяти.

Табл. 3-7 Пример констант разбиения для регистра sb.

РАЗРЯДНОСТЬ ЭЛЕМЕНТА (БИТОВ)	КОЛИЧЕСТВО ЭЛЕМЕНТОВ В СЛОВЕ	ЗНАЧЕНИЕ КОНСТАНТЫ, ЗАПИСЫВАЕМОЙ В SB
64	1	0
32	2	2
20	3	2000020000200002h1
16	4	00020002h
10	6	0208020080200802h1
8	8	02020202h
4	16	22222222h
2	32	0AAAAAAAAh

Чтобы не возникала путаница, какие биты регистра sb нужно заполнять, а какие нет, можно одними и теми же значениями прописывать пары битов. Это не приведет к каким-либо последствиям, поскольку четные биты, соответствующие sb2, будут проигнорированы процессором. Таким образом, например, запись в sb значения 02020202h эквивалентна sb = 03030303h.

Следует отметить, что при работе с матрицей весов нельзя изменять регистры nb1 и sb между командами ftw и wtw.

3.5.5 Регистр vr

Регистр vr используется только в операции взвешенного суммирования на векторном процессоре, выступая в качестве операнда **Y** (см. пункт 1.5.2). Он имеет разрядность 64 бита. Регистр vr доступен для записи. Содержимое vr не подается на Рабочую Матрицу, а приходит прямо на Векторное АЛУ (см. Рис. 1-5). Главной функцией vr является добавление одинакового смещения ко всем элементам вектора данных.

Использование регистра vr

Как уже было сказано, регистр `vr` используется только в операции взвешенного суммирования только в качестве операнда Y. При выполнении команды `vsum` из регистра `vr` на каждом такте считывается хранящееся там значение и суммируется с результатом умножения соответствующего значения из операнда X на рабочую матрицу.

Пример использования регистра `vr` в операции взвешенного суммирования:

```
begin "text"
    ...
    rep 20 data = [ar0++] with vsum, data, vr;
    ...
end "text";
```

В результате выполнения данной команды из памяти по адресу `ar0` будет прочитано 20 длинных слов, каждое из которых будет умножено на рабочую матрицу, а к результату добавлено значение из `vr`.

Загрузка значений в регистр vr

Прежде всего, необходимо помнить, что `vr` - это 64-х разрядный регистр. Система команд процессора NM6403 не содержит кода команды, который бы позволил напрямую загрузить в регистр 64-х разрядную константу. Однако существует несколько косвенных способов загрузки значений в регистр `vr`.

Загрузка значений в регистр может осуществляться либо из памяти, либо из пары регистров - адресного и парного с ним регистра общего назначения.

Способ 1. Загрузка по частям.

Процессор NM6403 позволяет осуществлять доступ к регистру `vr` по частям. То есть отдельно может быть загружена старшая часть регистра, отдельно младшая. Для этого в язык ассемблера введены специальные символы:

- `vrh` - старшая часть регистра `vr`;
- `vrl` - младшая часть регистра `vr`.

Приведем пример загрузки регистра `vr` по частям:

```
sir = 02020202h;   // загрузка младшей части vr.
vrl = sir;
sir = 05050505h;   // загрузка старшей части vr.
vrh = sir;
```

В результате в регистр `vr` загружено число `0505050502020202hl`.

Способ 2. Загрузка одинаковой младшей и старшей частей.

Если в младшую и старшую половины регистра необходимо загрузить одно и то же число, то можно использовать следующую команду:

```
sir = 03030303h;    // загрузка одинаковых констант
                    // в старшую и младшую части vr.
vr = sir;
```

При этом процессор автоматически запишет эту константу в обе половины регистра, то есть, после выполнения команды в регистре `vr` будет находиться число: `0303030303030303hl`.

Способ 3. Загрузка в регистр содержимого памяти.

Пример загрузки `vr` из памяти:

```
data "data"
    InitVR : long = 0123456789ABCDEFhl;
end "data";
begin "text"
    ...
    sir = [InitVR]; // длинная команда
                    // (содержит константу).
    vr = sir;       // короткая команда
    ...
    ar0 = InitVR;
    sir = [ar0];      // короткая команда.
    vr = sir;     // короткая команда.
    ...
end "text";
```

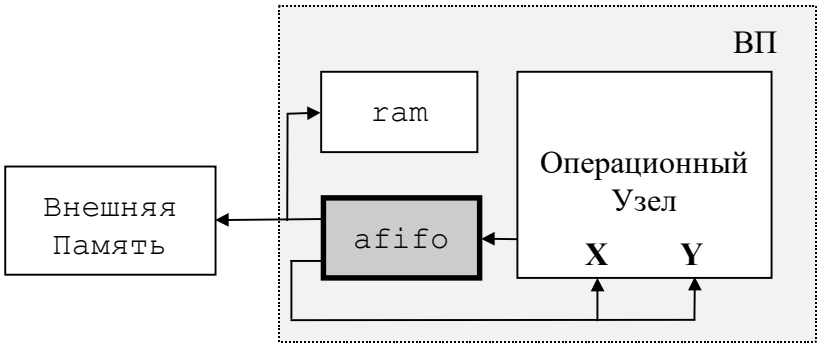
Пример загрузки `vr` из пары регистров:

```
begin "text"
    ...
    ar0, gr0 = [InitVR];
    sir = ar0, gr0;      // короткая команда.
    vr = sir;     // короткая команда.
    ...
end "text";
```

3.5.6 Регистр-контейнер afifo

Векторный регистр `afifo` представляет собой двух портовый буфер, организованный по принципу FIFO и имеющий объем 32 64-х разрядных слова. Его основное назначение - аккумулятор для хранения результата выполнения последней векторной команды.

Рис. 3-7 Взаимодействие afifo с другими устройствами процессора NM6403.



Регистр `afifo` доступен из ассемблера как по чтению, так и по записи. Однако он может использоваться только в векторных командах процессора. Данные из памяти не могут напрямую быть записаны в `afifo`, минуя операционный узел ВП. То есть имеется только один вход - это выход операционного узла (см. Рис. 3-7). Ниже будет показано, как можно загрузить данные из памяти в `afifo`.

Буфер `afifo` участвует непосредственно или косвенно в каждой векторной инструкции, исключая загрузку весов в рабочую матрицу. Буфер может использоваться в обеих частях векторной команды. Если он используется в левой части команды, то в нем сохраняется результат вычислений, если `afifo` используется в правой части команды, то он участвует в вычислениях как операнд **X** или **Y**.

Буфер `afifo` может освобождаться и/или заполняться только в течение одной векторной команды. В нем **не могут** накапливаться результаты нескольких векторных команд.

Содержимое буфера `afifo` изменяется в каждой векторной инструкции, за исключением загрузки весов. Если буфер содержит данные (т.е. он заполнен), следующая векторная команда должна или сохранить данный в памяти или повторно использовать их как входные данные в последующих вычислениях. Выполнение команды, которая производит вычисления и не сохраняет или не использует содержимое `afifo`, вызывает сбой.

Для того чтобы избежать некорректного использования `afifo`, необходимо придерживаться следующих правил:

- не загружать данные из пустого `afifo` (пустое `afifo` не может быть использовано как источник-операнд в векторных командах);
- не загружать данные в непустое `afifo`, если текущая инструкция не сохраняет предыдущее содержимое `afifo` в память или не использует его как источник-операнд.
- все регистры-контейнеры, использующиеся в векторных командах, должны содержать одинаковое количество данных.

Очистка содержимого `afifo`

Контроль за содержимым afifo может осуществляться программно путем анализа битов EMPTA (бит 12: "afifo пустое"/"afifo непустое") и FULLA (бит 11: "afifo заполнено"/"afifo не заполнено") в поле VPF регистра intr (см. 3.2.3 Регистр intr). Они отражают динамическое состояние afifo (то, как меняется его состояние в процессе выполнения векторной команды).

Биты поля AFIFO_VAL регистра pswr хранят информацию о количестве слов в afifo до и после выполнения команды или группы команд. Эта информация более статична (меняется один раз в течение выполнения команды и описывает результат, а не процесс).

Содержимое afifo может быть программно очищено путем установки в 1 бита AFCL (бит 14) поля FCL регистра pswr (см. 3.2.6 Регистр pswr). После того, как бит AFCL в pswr выставлен в 1, он должен быть сброшен в 0, иначе будет блокирована работа ВП. Команды установки и снятия бита могут следовать друг за другом. Следующий пример показывает фрагмент программы, который очищает содержимое afifo:

```
begin ".text"
    ...
    pswr set   4000h; // бит AFCL устанавливается в '1'
    pswr clear 4000h; // бит AFCL очищается, afifo пусто
    ...
end ".text";
```

Загрузка данных в afifo

Регистр afifo представляет собой очередь FIFO. Глубина её заполнения зависит от выполняемой векторной команды, и может изменяться в пределах от одного до тридцати двух длинных слов, например:

```
rep 24 data = [ar0++] with vsum , data, ram;
```

Вследствие выполнения операции взвешенного суммирования, приведенной выше, в afifo попало 24 слова результата.

Хотя данные попадают в afifo только в результате выполнения вычислений на ВП, существует возможность загрузить их из памяти без изменений. Например, следующая команда направляет 10 длинных слов входных данных в afifo:

```
rep 10 data = [ar0++] with data; // данные попадают
                                 // напрямую в afifo.
```

При выполнении приведенной выше инструкции в ВП выполняется операция логического OR с нулевым вектором, которая не приводит к изменению данных. Теперь в afifo хранится 10 длинных слов, загруженных из памяти.

Как уже было сказано, невозможно заполнять afifo по частям, то есть первой командой загрузить туда 5 слов, а второй еще пять. Загрузка данных в afifo возможна только в том случае, если к этому моменту afifo пусто, или оно используется в качестве входного

буфера в данной операции. Более подробно об ошибках при работе с `afifo` см. ниже.

Выгрузка данных из `afifo` в память и в `ram`

Данные, хранящиеся в `afifo`, могут быть выгружены в память. Для этого используется, например, следующая команда:

```
rep 1 [ar0++] = afifo;
```

Приведенная выше команда выгружает из `afifo` в память одно слово. Необходимо отметить, что невозможно выгружать данные из `afifo` в память по частям, то есть одной командой выгрузить 10 слов из находящихся там 16-ти, а другой оставшиеся шесть. Возможна только выгрузка содержимого целиком, задаваемая одной командой.

Параллельно с выгрузкой в память содержимое `afifo` может быть скопировано в `ram`. Эта операция осуществляется обычно при помощи следующей инструкции:

```
rep 8 [ar0++], ram = afifo;
```

При такой записи старое содержимое `ram` будет заменено на новое, пришедшее из `afifo`. Невозможно скопировать `afifo` только в `ram`, без записи в память.

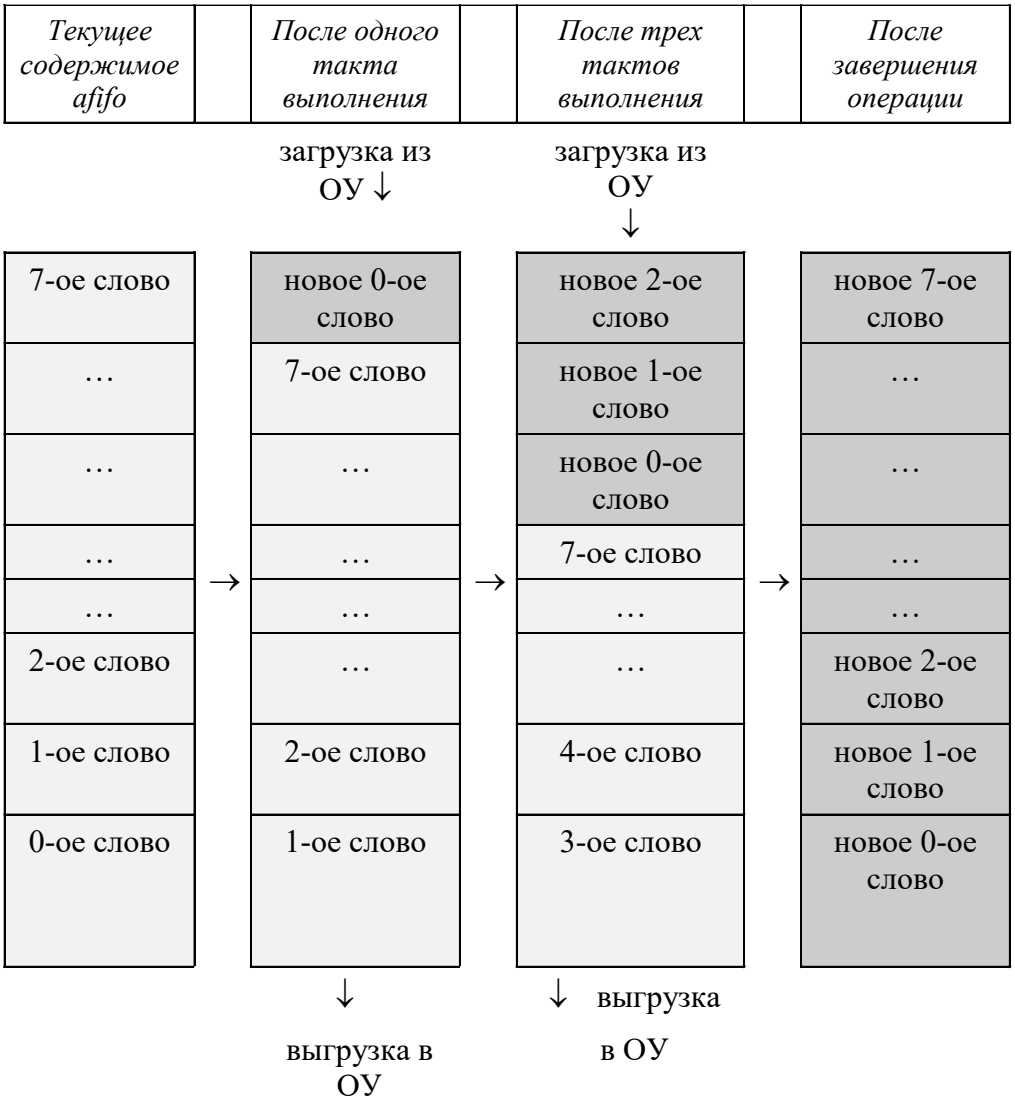
Примечание

Содержимое `afifo` не может быть выгружено в регистры процессора или регистровые пары, только в память.

Использование `afifo` в качестве входного буфера в векторных операциях

Содержимое `afifo` может быть использовано на следующем шаге вычислений, в качестве входного буфера, содержимое которого подается на вход **X** или/и **Y** ВП (см. Рис. 3-7). Такое возможно благодаря тому, что `afifo` является двух портовым. Одновременно его старое содержимое подается на вход ВП, очередь внутри `afifo` сдвигается, а в освободившуюся ячейку заносится результат новых вычислений.

Рис. 3-8 Содержимое afifo на различных стадиях выполнения векторных инструкций



Содержимое `afifo` может подаваться на вход **X** операционного узла, например:

```
                                        вход X      вход Y
rep 32 data = [ar0++] with afifo and data;
```

Приведенная выше инструкция выполняет побитовую операцию AND содержимого памяти с тем, что хранилось к этому моменту в `afifo`. Операция выполняется на векторном АЛУ, входящем в состав операционного узла, первый их двух операндов в правой части команды попадает на вход **X**, второй на вход **Y**.

Также содержимое `afifo` может быть подано на вход **Y**:

```
                                        вход X  вход Y
rep 32 data = [ar0++] with vsum , data, afifo;
```

Приведенная выше инструкция выполняет операцию взвешенного суммирования на рабочей матрице ВП, входящей в состав операционного узла.

Содержимое `afifo` может быть подано и на вход **X** и на **Y**:

```
                вход X     вход Y
rep 32 with afifo + afifo;
```

Приведенная выше инструкция удваивает значения элементов, расположенных в `afifo`.

Результаты всех трех приведенных выше операций попадают в `afifo`.

Одновременная выгрузка из `afifo` в память и передача его на вход ОУ

Регистр `afifo` позволяет одновременную выгрузку данных в память и передачу его содержимого на входы **X** и/или **Y** операционного узла ВП, например:

```
rep 32 [ar0++] = afifo with afifo - ram;
```

В приведенной выше команде содержимое `afifo` сохраняется в памяти и одновременно используется для операции вычитания в правой части команды. В результате этого действия через 32 такта старое содержимое `afifo` окажется во внешней памяти, а в `afifo` будет храниться разность его старого содержимого и данных из `ram`.

Регистр `afifo` позволяет в одной команде выполнять даже такую многоходовую комбинацию пересылок и преобразований, как одновременное сохранение содержимого в памяти, в `ram` и передача его на вход операционного узла, например:

```
rep 20 [ar0++],ram = afifo with not afifo;
```

В приведенной выше команде содержимое `afifo` сохраняется в памяти, копируется в `ram` и одновременно используется для операции отрицания в правой части команды. В результате этого действия через 20 тактов старое содержимое `afifo` окажется во внешней памяти, те же данные попадут в `ram`, а в `afifo` будет храниться отрицание его старого содержимого.

Использование `afifo` в операциях маскирования

В операциях маскирования, выполняемых на операционном узле ВП, помимо входов **X** и **Y** существует вход для маски. Более подробно об операциях маскирования см. 1.5.4 Операция маскирования. Содержимое `afifo` может использоваться в качестве маски, например:

```
                                  вход маски
rep 32 data = [ar0++] with mask afifo, ram, data;
```

Приведенная выше команда выполняет операцию маскирования на векторном АЛУ.

Другой пример демонстрирует использование `afifo` в операции маскирования, совмещенной с выполнением взвешенного суммирования:

вход маски

```
rep 32 data = [ar0++] with vsum afifo, ram, data;
```

Ошибки при работе с `afifo`

Следующие причины лежат в основе ошибок при работе с `afifo`:

- попытка читать данные из пустого `afifo`;
- попытка читать больше данных, чем содержится в `afifo`;
- попытка читать меньше данных, чем содержится в `afifo`;
- попытка записывать результаты в непустое `afifo`.

В случае возникновения ошибки при работе с `afifo` по указанным выше причинам команда, в которой содержится ошибка, не отрабатывается. Процессор заменяет ее пустой командой `vnul` и порождает прерывание по неправильной векторной команде.

3.5.7 Логический регистр-контейнер data

Векторный регистр `data` является логическим регистром, используемым для описания данных, проходящих по шине данных в направлении из внешней памяти в операционный узел ВП во время выполнения векторной команды.

На каждом такте выполнения векторной команды значение регистра `data` равно значению слова данных, считанного из памяти и проходящего в данный момент по шине данных (глобальной или локальной). Регистр `data` введен для того, чтобы управлять потоком данных, считываемых из памяти и направлять его на входы операционного узла ВП.

Регистр `data` может для общности рассматриваться как псевдобуфер шины данных. Псевдобуфер имеет глубину 32x64 бита и организован по принципу FIFO. Поскольку одна векторная команда может считывать содержимое произвольного числа ячеек внешней памяти в пределах от 1 до 32, глубина псевдобуфера может изменяться в этих пределах.

Использование `data` для обработки данных на проходе

Регистр `data` используется для обработки данных на проходе во время считывания их из внешней памяти, например:

вход **Y**

```
rep 32 data = [ar0++] with ram or data;
```

В приведенной выше команде 32 слова данных, считываемые из памяти, поступают на вход **Y** операционного узла, где выполняется побитовая операция OR с содержимым `ram`. Точно также данные из памяти могут быть поданы на вход **X**:

Вход **X**

```
rep 32 data = [ar0++] with data or ram;
```

В приведенной выше команде 32 слова данных, считываемые из памяти, поступают на вход **X** операционного узла, где выполняется побитовая операция OR с содержимым `ram`. Результат выполнения двух приведенных выше команд будет одинаковым.

С помощью регистра `data` данные из памяти могут быть направлены как на вход **X** операционного узла ВП, так и на вход **Y**, в том числе одновременно, например:

Вход **X** вход **Y**

```
rep 32 data = [ar0++] with data + data;
```

В приведенной выше команде значения элементов вектора, считываемого из памяти, удваиваются на проходе.

Результаты выполнения всех приведенных выше команд попадают в `afifo`.

Использование `data` в операциях маскирования

В операциях маскирования, выполняемых на операционном узле ВП, помимо входов **X** и **Y** существует вход для маски. Более подробно об операциях маскирования см. 1.5.4 Операция маскирования. Содержимое `data` может использоваться в качестве маски, например:

вход маски

```
rep 32 data = [ar0++] with mask data, afifo, ram;
```

Приведенная выше команда выполняет операцию маскирования на векторном АЛУ.

Другой пример демонстрирует использование `data` в операции маскирования, совмещенной с выполнением взвешенного суммирования:

вход маски

```
rep 32 data = [ar0++] with vsum data, ram, afifo;
```

Ошибки при работе с регистром data

Ошибки при работе с регистром `data` происходят при попытке использовать регистр `data` в правой части инструкции без загрузки данных в регистр из памяти в левой части инструкции.

Инструкции, содержащие подобную ошибку, не выполняются, процессор заменяет их пустой командой `vnul` и порождает прерывание по неправильной векторной команде.

3.5.8 Регистр-контейнер ram

Векторный регистр `ram` представляет собой очередь глубиной в 32 64-х разрядных слова, организованная по принципу FIFO. В `ram` может быть загружено от 1 до 32 слов. Основным его отличием от обычного FIFO является то, что после считывания из `ram` его содержимое сохраняется. То есть, записав один раз вектор данных в `ram` его затем можно повторно использовать в векторных операциях.

Регистр `ram` доступен из ассемблера как по чтению, так и по записи. Однако он может использоваться только в векторных командах процессора.

Данные в `ram` могут быть записаны напрямую из памяти или из `afifo`(см. выше).

Данные, хранящиеся в `ram`, используются в качестве входных для векторного АЛУ и для операций взвешенного суммирования на рабочей матрице ВП. Они могут быть поданы как на вход X операционного узла ВП, так и на вход Y.

Содержимое `ram` не может быть напрямую сохранено во внешней памяти, только через операционный узел ВП.

Векторный регистр `ram` предназначен для хранения только одного вектора упакованных данных, длиной от 1 до 32 длинных слов. Любая запись в `ram` приводит к потере его прежнего содержимого. Содержимое `ram` может многократно использоваться в векторных операциях, однако в вычислениях должны участвовать все данные, хранящиеся в `ram`. Не допускается использование только части хранящихся там данных.

Контроль над содержимым `ram` может осуществляться программно, путем анализа поля `RAM_VAL` регистра `intr` (см. 3.2.3 Регистр intr). Это поле сообщает, какое количество длинных слов в данный момент находится в `ram`.

Загрузка данных в `ram`

Данные загружаются непосредственно из внешней памяти, например:

```
rep 16 ram = [ar0++];
```

Приведенная выше команда, описывает загрузку 16-ти длинных слов упакованных данных из внешней памяти в `ram`. При этом старое содержимое буфера теряется, даже если там хранилось 32 слова.

Важное свойство процесса загрузки `ram` состоит в том, что данные, считываемые из памяти, проходят по шине данных, а значит параллельно с загрузкой в `ram` могут быть направлены на входы операционного узла ВП, например:

```
rep 32 ram = [ar0++] with data + afifo;
```

В приведенной выше команде данные из памяти копируются в `ram` и параллельно передаются на вход векторного АЛУ для выполнения операции сложения с содержимым буфера `afifo`.

Использование `ram` в операциях на ОУ ВП

Как уже говорилось, содержимое `ram` может многократно использоваться в вычислениях, выполняемых на операционном узле ВП. Содержимое `ram` может подаваться на вход **X** операционного узла, например:

вход **X**

```
rep 32 data = [ar0++] with ram and data;
```

Приведенная выше инструкция выполняет побитовую операцию AND содержимого памяти с тем, что хранится в `ram`. Операция выполняется на векторном АЛУ, входящем в состав операционного узла, первый их двух операндов в правой части команды попадает на вход **X**, второй на вход **Y**.

Также содержимое `ram` может быть подано на вход **Y**:

вход **Y**

```
rep 32 data = [ar0++] with vsum , data, ram;
```

Приведенная выше инструкция выполняет операцию взвешенного суммирования на рабочей матрице ВП, входящей в состав операционного узла.

Содержимое `afifo` может быть подано и на вход **X** и на **Y**:

вход **X** вход **Y**

```
rep 32 with ram + ram;
```

Приведенная выше инструкция удваивает значения элементов, расположенных в `ram`.

Результаты выполнения всех приведенных выше команд попадают в `afifo`.

Использование `ram` в операциях маскирования

В операциях маскирования, выполняемых на операционном узле ВП, помимо входов **X** и **Y** существует вход для маски. Более подробно об операциях маскирования см. 1.5.4 Операция маскирования. Содержимое `ram` может использоваться в качестве маски, например:

вход маски

```
rep 32 data = [ar0++] with mask ram, afifo, data;
```

Приведенная выше команда выполняет операцию маскирования на векторном АЛУ.

Другой пример демонстрирует использование `ram` в операции маскирования, совмещенной с выполнением взвешенного суммирования:

вход маски

```
rep 32 data = [ar0++] with vsum ram, data, afifo;
```

Ошибки при работе с `ram`

Следующие причины лежат в основе ошибок при работе с `ram`:

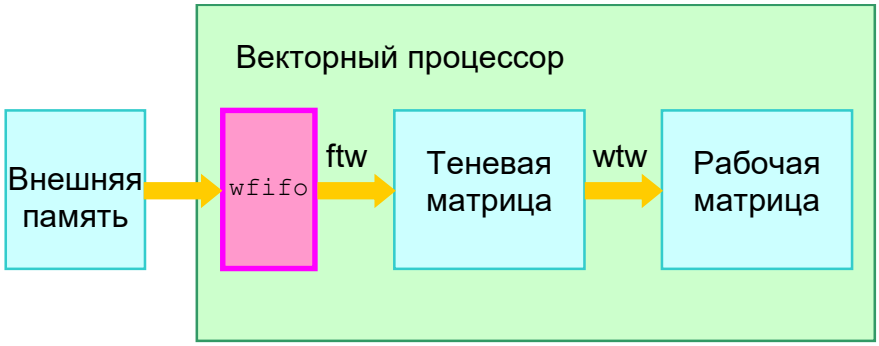
- попытка читать данные из пустого `ram`;
- попытка читать больше данных, чем содержится в `ram`;
- попытка читать меньше данных, чем содержится в `ram`;
- попытка одновременно вести запись в `ram` и использовать его старое содержимое является ошибкой для nm6403, для позднейших версий ядра это вполне допустимо.

В случае возникновения ошибки при работе с `ram` по указанным выше причинам команда, в которой содержится ошибка, не отрабатывается. Процессор заменяет ее пустой командой `vnul` и порождает прерывание по неправильной векторной команде.

3.5.9 Регистр-контейнер wfifo

Векторный регистр `wfifo` представляет собой очередь глубиной в 32 64-х разрядных слова, организованную по принципу двух портового FIFO. Он используется как буфер для накопления и хранения весовых коэффициентов, которые затем загружаются в теневую матрицу ВП, а из нее в рабочую.

Рис. 3-9 Загрузка весов из внешней памяти



Как видно из рисунка буфер `wfifo` может использоваться для быстрой загрузки данных из внешней памяти, данный процесс позволяет разгрузить внешнюю шину данных.

Регистр `wfifo` доступен как по чтению, так и по записи. Однако он может использоваться только в векторных командах процессора. Загрузка данных из внешней памяти в `wfifo` осуществляется напрямую 64-х разрядными словами (см. ниже). Данные, хранящиеся в `wfifo`, могут быть выгружены оттуда только в теневую матрицу.

Регистр `wfifo` не может быть использован в качестве источника для входов **X** и **Y** операционного узла ВП, а также для входа

маски при операциях маскирования. Единственное его предназначение состоит в хранении весовых коэффициентов.

В отличие от других векторных регистров-контейнеров загрузка данных и их выгрузка из wfifo может осуществляться по частям, то есть, например, в wfifo можно загрузить сначала 8 слов, а затем еще 24, но так, чтобы не произошло переполнения. То же происходит при чтении из wfifo. Процессор NM6403 в зависимости от конфигурации теневой матрицы считывает из wfifo разное количество слов. При этом в буфере могут оставаться данные, которые будут считаны следующей командой обращения к wfifo.

Примечание

Загрузка весов из wfifo в теневую матрицу всегда происходит за 32 такта, что является основным ограничением ВП. Все использующиеся алгоритмы должны принимать это во внимание. Но данная операция может выполняться в фоновом режиме во время выполнения других скалярных и векторных команд.

Примечание

Начиная с версии архитектуры ntc3 загрузка весов из wfifo в теневую матрицу значительно ускорена и происходит за число тактов равное числу строк в разбиении матрицы, но не менее 2-х тактов. Данная операция может выполняться в фоновом режиме во время выполнения других скалярных и векторных команд.

Очистка содержимого wfifo

Контроль за содержимым wfifo может осуществляться программно путем анализа битов EMPTW (бит 10: "wfifo пустое"/"wfifo непустое") и FULLW (бит 9: "wfifo заполнено"/"wfifo не заполнено") в поле VPF регистра intr (см. 3.2.3 Регистр intr). Они отражают динамическое состояние wfifo (то, как меняется его состояние в процессе выполнения векторной команды).

Поле WFIFO_VAL регистра intr содержит информацию о количестве слов в wfifo после выполнения инструкции или группы инструкций. Данное поле изменяется медленней, чем поле EMPTW и FULLW, т.к. оно описывает результат, а не процесс.

Содержимое wfifo может быть программно очищено путем установки в 1 бита WFCL (бит 15) поля FCL регистра pswr (см. 3.2.6 Регистр pswr). После того, как бит WFCL в pswr выставлен в 1, он должен быть сброшен в 0, иначе будет блокирована работа ВП. Команды установки и снятия бита могут следовать друг за другом, как в нижеприведенном примере:

```
begin ".text"
    ...
    pswr set   8000h; // бит WFCL установлен в '1'
    pswr clear 8000h; // бит WFCL сброшен, wfifo пустое
    ...
end ".text";
```

Загрузка весовых коэффициентов в wfifo

Загрузка весовых коэффициентов в wfifo происходит напрямую из памяти. wfifo может заполняться за одну или за несколько команд, например:

```
rep 16 wfifo = [ar0++]; // загрузка первых 16-ти слов
rep 16 wfifo = [ar1++]; // загрузка вторых 16-ти слов
```

Первой из приведенных выше команд в wfifo загружаются 16 длинных слов весовых коэффициентов, во второй подгружаются еще 16 слов с другого адреса памяти. При этом общее заполнение wfifo составляет 32 слова.

Выгрузка весовых коэффициентов из wfifo

Из wfifo весовые коэффициенты могут быть выгружены только в теневую матрицу. Для этого используется инструкция ftw, которая может использоваться как отдельная команда, например:

```
ftw;
```

и может также входить в состав другой векторной команды, например:

```
rep 32 data = [ar0++], ftw with not data;
```

Одновременная загрузка и выгрузка данных из wfifo

Буфер wfifo является двух портовым, возможна одновременная загрузка в него набора весовых коэффициентов и выгрузка в теневую матрицу, например:

```
rep 32 wfifo = [ar0++], ftw;
```

Приведенная выше команда осуществляет одновременно загрузку весов в wfifo и выгрузку из wfifo в теневую матрицу ВП. Если к моменту начала выполнения команды wfifo было пусто, то операция выгрузки весов в теневую матрицу будет аппаратно блокирована, пока в wfifo не появится первый элемент данных, подгруженный из памяти.

После того, как первое слово данных появится в буфере, оно следующим же тактом будет отправлено в теневую матрицу. Этим же тактом в wfifo будет подгружено следующее слово из памяти. Таким образом, в теневую матрицу 32 слова будут загружены за 33 такта. При этом wfifo останется в результате пустым.

Если в wfifo к моменту выполнения команды уже находилось какое-то количество весовых коэффициентов, то они первыми будут направлены в теневую матрицу. В то же время в wfifo будут подгружаться новые данные. Если тех весов, которые находились в wfifo к моменту начала выполнения команды, недостаточно для заполнения теневой матрицы, то в нее попадет и часть вновь загружаемых данных.

Загрузка в wfifo нескольких матриц весовых коэффициентов

Количество слов, загружаемых в теневую матрицу из wfifo, определяется разбиением ее на строки. Каждой строке матрицы соответствует 64-х разрядное слово данных (см. 3.4.3 Регистр sb (sb1 и sb2)).

Если количество слов, загружаемых в теневую матрицу из wfifo значительно меньше его емкости, то целесообразно загрузить в wfifo сразу несколько матриц весовых коэффициентов, а затем командой ftw подчитывать порции данных для обновления содержимого теневой матрицы. Это позволит разгрузить шины данных и использовать их более эффективно, например:

```
data "dataMatrix"
    Matrix: long[32] = (...);
end "dataMatrix";

begin "text"
...
sir = 02020202h; // в матрице 8 строк.
sb = sir;
ar0 = Matrix;    // загрузка в регистр адреса матрицы.
rep 32 wfifo = [ar0++], ftw, wtw; //загрузка весов с
...              // одновременной записью 8-ми из них
                 // в теневую матрицу.
                 // В wfifo осталось 24 слова.
ftw, wtw;        // Загрузка в теневую матрицу
                 // следующих 8-ми слов.
...              // В wfifo осталось 16 слов.
ftw, wtw;        // Загрузка в теневую матрицу
                 // следующих 8-ми слов.
...              // В wfifo осталось 8 слов.
ftw, wtw;        // Загрузка в теневую матрицу
                 // следующих 8-ми слов.
...              // В wfifo осталось пустым.
end "text";
```

Загрузка весовых коэффициентов из wfifo в теневую матрицу не занимает внешнюю шину, поэтому может протекать на фоне выполнения других векторных и скалярных команд. Часто описанный выше подход к работе с wfifo упрощает разработку программ.

Ошибки при работе с wfifo

Следующие причины лежат в основе ошибок при работе с wfifo:

- попытка читать данные из пустого wfifo;
- попытка записывать данные в заполненное wfifo без одновременной их выгрузки в теневую матрицу;

В случае возникновения ошибки при работе с `ram` по указанным выше причинам команда, в которой содержится ошибка, не отрабатывается и процессор блокируется.

Пример загрузки весов в `wfifo`

Процедура загрузки весов в рабочую матрицу рассматривается на примере, в котором матрица разбита на 8 строк и 8 столбцов. На Рис. 3-10 представлена схема обработки данных в векторной части процессора:

Рис. 3-10 Загрузка коэффициентов в рабочую матрицу.

Данные в рабочую матрицу заносятся из внешней памяти, доступной процессору. В памяти они хранятся в виде массива 64-х разрядных слов. При загрузке матрицы в нее попадет столько слов, каково ее разбиение по строкам, заданное в регистре `sb2`. Каждая строка задается отдельным 64-х разрядным словом. Слово массива с нулевым индексом попадает в нулевую строку матрицы. Как видно из Рис. 3-10, строки рабочей матрицы нумеруются снизу вверх. Такой способ нумерации обусловлен нумерацией битов в слове (нумерация ведется справа налево, младший бит находится справа). В том же направлении ведется увеличение адресов памяти.

На языке ассемблера запись массива данных, соответствующую блоку коэффициентов, загружаемых в матрицу, имеет вид:

```
data "data"
    Matrx: long[8] = (00101010101010101hl, // нулевая строка
                      0FF01FF01FF01FF01hl,//младший байт слова
                      0FFFF0101FFFF0101hl, // вторая строка
                      001FFFF0101FFFF01hl,
                      0FFFFFFFF01010101hl,
                      001FF01FFFF01FF01hl,
                      00101FFFFFFFF0101hl,
                      0FF0101FF01FFFF01hl); // седьмая строка
```

Поскольку рабочая матрица процессора NM6403 разбивается регистром `nb2` на 8 столбцов, младший байт нулевого слова массива попадет в нулевой столбец, первый байт в первый столбец, и т.д.

Примечание

Стоит обратить внимание на то, что хотя строки рабочей матрицы нумеруются снизу вверх, в нулевую строку попадет нулевой элемент массива, который при записи массива в столбик (в ассемблерном файле) оказывается самым верхним.

Весовые коэффициенты читаются из памяти `wfifo`. Загрузка данных из памяти в рабочую матрицу описывается следующими командами на языке ассемблера:

```
begin "text"
<_Func>
    sir = 80808080h; // разбиение матрицы на столбцы.
    nb1 = sir
    sir = 03030303h; // разбиение матрицы на строки.
    sb  = sir

    ar0 = Matrix;
    rep 8 wfifo = [ar0++], ftw, wtw;
    ...
end "text";
```

Сначала заполняются регистры `nb1` и `sb`, определяющие будущее разбиение матрицы. Реально разбиение, задаваемое ими, вступит в силу не сразу после присваивания им новых значений, а только после выполнения команд `ftw` и `wtw`.

Примечание

Значения регистров nb1 и sb между командами ftw и wtw менять нельзя. Для процессора nm6403 это правило можно было не соблюдать, более того, конструкция вида

```
.wait;
 nb1=***;   // (1)
 wtw;
```

широко использовалась для фиксации аппаратной ошибки. Часто конструкция оборачивалась в макрос. Для переноса таких программ на ядро nmc3 придётся вручную изменить все соответствующие места:

-проблема есть только если выше по коду лежит команда ftw, если раньше попадется wtw (или ftw, wtw), ничего делать не требуется.

-если значение nb1 ниже не используется, или записываемое значение равно прежнему, то просто уберите команду(1).

-в противном случае, команду (1) нужно как-то перенести в другое место.

Регистры `nb1` и `sb` являются 64-х битовыми. Если они инициализируются 32-х разрядной константой, то процессор копирует это значение в старшие 32 бита, то есть получается, что регистр nb1 инициализирован длинной константой `8080808080808080hl`. То же верно и для регистра `sb`. Более подробно работа с регистрами `nb1` и `sb` описана в пунктах 3.4.2 Регистр nb1(nb2) и 3.4.3 Регистр sb (sb1 и sb2).

Итак, определено будущее разбиение рабочей матрицы.

В адресный регистр заносится адрес массива весовых коэффициентов, а затем данные загружаются из памяти в `wfifo`.

По команде `ftw` данные из `wfifo` попадают в теневую матрицу. Эта передача сопровождается формированием структуры матрицы, она занимает некоторое время, и для большей эффективности её можно выполнять параллельно с предыдущей векторной командой. Если требуется изменить разбиение матрицы (предыдущая команда использовала другое), то это надо сделать до `ftw`.

Для ядра nmc3 загрузка весов из *`wfifo`* в теневую матрицу происходит за число тактов равное числу строк в разбиении матрицы, но не менее 2-х тактов.

После того, как закачка весов в теневую матрицу завершена, выполняется команда `wtw`. Она за один такт переписывает содержимое теневой матрицы в рабочую. При этом значения регистров `nb1` и `sb1` копируются в `nb2` и `sb2`. Загрузка рабочей матрицы завершена.

3.5.10 Сценарии типичного использования векторных регистров

Данный пункт содержит практические примеры работы с описанными регистрами для настройки векторного узла.

Арифметические и логические операции

Более простой случай, поскольку не используется матрица. Порядок действий:

- Установить `nb1`, `f1cr` (если не устраивают прежние значения);
- `wtw;`
- Векторная арифметическая команда.

Пример:

```
sir = 80008000h;
nb1 = sir;
wtw;
rep 16 data= [ar0++] with data +1;
```

Взвешенное суммирование

Здесь дополнительно требуется загрузить и сконфигурировать матричный умножитель. Порядок действий:

- Установить `nb1`, `sb1`, `f2cr`, `f1cr` (если не устраивают прежние значения); Загрузить веса в векторный регистр wfifo
- `ftw;`
- `wtw;`
- Команда взвешенного суммирования.

Пример:

```
rep 4 wfifo= [ar4++];
sir = 80008000h;
nb1 = sir;
sir = 80008000h;
sb1 = sir;
ftw;
wtw;
rep 16 data= [ar0++] with vsum ,data, 0;
```

Надо отметить, что инструкции приведенного кода будут выполняться последовательно, то есть, не слишком эффективно. Но при выполнении нескольких подобных блоков подряд, загрузка весов будет выполняться параллельно с предыдущей операцией умножения. Максимальная же эффективность кода достигается только тщательным изучением архитектуры устройства и экспериментированием.

4 Формат Ассемблерных Инструкций

Процессор NeuroMatrix работает с машинными командами 32-х и 64-х разрядного формата. В одной машинной команде содержится две операции процессора. В этом смысле NeuroMatrix представляет собой скалярный микропроцессор со статической LIW-архитектурой.

Разрядность инструкций процессора

Короткие инструкции не содержат константы и имеют разрядность 32 бита.

Длинные инструкции содержат в коде команды 32-х разрядную константу, поэтому их разрядность составляет 64 бита.

Процессор адресуется к 32-х разрядным словам. На хранение коротких инструкций отводится одна ячейка памяти, для длинных - две.

Примечание

Длинные инструкции всегда располагаются по четным адресам. Если начало длинной инструкции при ассемблировании приходится на нечетный адрес, перед ней автоматически вставляется пустая команда `nul`.

Процессор NeuroMatrix за одно обращение к памяти считывает либо две коротких инструкции, либо одну длинную, поэтому регистр `pc`, определяющий адрес следующей считываемой инструкции, всегда имеет четное значение. Подробнее о регистре `pc` см. 3.2.5 Регистр pc.

Типы инструкций процессора

Все инструкции процессора делятся на:

- **скалярные инструкции**, которые управляют работой скалярного RISC-ядра, таймеров, осуществляют загрузку/чтение всех регистров (доступных по чтению/записи) за исключением векторных регистров, образующих очереди FIFO;
- **векторные инструкции**, которые управляют работой векторного процессора.

Структура инструкций процессора

Каждая инструкция процессора состоит из **двух частей**, называемых условно "левой" и "правой". Обе части инструкции выполняются процессором одновременно.

В **левой** части инструкции записываются только **адресные операции**, в **правой** все **арифметическо-логические**, не связанные с вычислением адресов и обращением к памяти.

Левая и правая части инструкции соединяются воедино при помощи ключевого слова `with`. Пример инструкции процессора:

```
gr0 = [ar0++] with gr1 = gr3 << 4;
```

Левая часть инструкции, адресная операция

Правая часть инструкции, арифметическая операция

В языке ассемблера левая или правая часть инструкции может быть опущена, однако поскольку процессор не может выполнить только левую или только правую часть команды, вместо опускаемой части при компиляции автоматически добавляется пустая операция `nul`. То есть ассемблерная инструкция записанная как:

```
gr0 = [ar0++];
```

трактуется ассемблером как:

```
gr0 = [ar0++] with nul;
```

Или то же самое с левой частью:

```
gr1 = gr3 << 4;
```

рассматривается как:

```
nul with gr1 = gr3 << 4;
```

Для улучшения читаемости программы, в случае если левая или правая часть инструкции не используется, связка `with` может опускаться.

Особенности структуры векторных инструкций процессора

Векторные инструкции процессора также как и скалярные разделены на левую и правую части. Однако помимо этого они имеют дополнительное поле, которое присутствует во всех векторных инструкциях за исключением одиночных инструкций `ftw` и `wtw`. Поле, о котором идет речь, называется полем количества повторений. Вот пример того, как выглядит векторная инструкция:

```
rep 32 data = [ar1++] with vsum , data, afifo;
```

Левая и правая часть векторной инструкции разделены ключевым словом `with`, поле количества повторений (подчеркнуто) определяет, сколько длинных слов будет обработано данной командой. В большинстве случаев векторная команда будет выполняться столько тактов, каково значение счетчика, поскольку операция над длинным словом в векторном процессоре выполняется за один такт.

В случае, если левая часть инструкции опущена, поле повторения и слово-связка `with` остаются при написании инструкции, например:

```
rep 16 with ram - 1; // правильная инструкция
```

Любые другие формы записи инструкции, как то

```
rep 16 ram - 1; или with ram - 1; // содержат ошибки
```

являются ошибочными, о чем сообщит ассемблер.

Примечание

Векторные и скалярные инструкции не могут смешиваться в одной команде, даже если у одной из них опущена левая часть, а у другой правая.

4.1 Типы скалярных команд

Скалярная инструкция процессора состоит из двух частей, в каждой из которых могут встречаться только определенные типы скалярных команд. Далее в таблице указано, какие группы скалярных команд могут быть записаны в левой, а какие в правой части скалярной инструкции.

Табл. 4-1 Положение различных типов команд в скалярной инструкции.

ЛЕВАЯ ЧАСТЬ СКАЛЯРНОЙ ИНСТРУКЦИИ	ПРАВАЯ ЧАСТЬ СКАЛЯРНОЙ ИНСТРУКЦИИ
• Команды загрузки/записи регистров; • Команды пересылки значений регистров; • Команды адресной арифметики; • Специальные скалярные команды; • Команды безусловного и условного перехода; • Команды безусловного и условного обращения к подпрограмме; • Команды возврата из подпрограммы или прерывания; • Пустая команда.	• Арифметические операции; • логические операции; • сдвиговые операции; • пустая операция.

В скалярной команде любой тип операции из левой колонки таблицы может быть совмещен с любым типом из правой. Однако не могут встретиться два типа операций из одной колонки.

В качестве примера скалярной инструкции, содержащей левую и правую части, может быть рассмотрена инструкция:

```
gr4 = [ar0++] with gr0 = gr1 and not gr2;
```

В её левой части происходит инициализация регистра `gr4` содержимым ячейки памяти, на которую указывает регистр `ar0` с одновременной инкрементацией адреса.

В правой части выполняется трехоперандная логическая операция. Содержимое регистров `gr1` и `gr2` подвергается побитовой логической операции `and not`, а результат записывается в регистр `gr0`.

4.2 Типы векторных команд

Векторная инструкция, также как и скалярная, состоит из левой и правой частей. В левой части используются операции адресации, в правой большинство векторных команд. Для большей наглядности ниже приведена таблица, в которой содержится информация о том, какие типы векторных команд могут располагаться в левой, а какие в правой части векторной инструкции.

ЛЕВАЯ ЧАСТЬ ВЕКТОРНОЙ ИНСТРУКЦИИ	ПРАВАЯ ЧАСТЬ ВЕКТОРНОЙ ИНСТРУКЦИИ
• Команды загрузки данных в векторный процессор; • Команды выгрузки данных из векторного процессора; • Специальные векторные команды; • пустая векторная операция.	• взвешенное суммирование (матричное умножение); • маскирование; • арифметические операции; • логические операции; • операция циклического сдвига; • операции активации операндов; • выгрузка управляющих векторных регистров; • пустая операция.

Пример векторной инструкции с левой и правой частями:

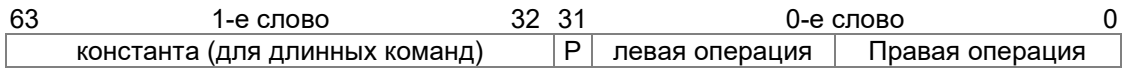
```
rep 32 ram = [ar0++gr0] with vsum , data, afifo;
```

Ключевое слово `with` разделяет левую и правую части. Счетчик повторений `rep` *`число`* (числовое константное выражение) присутствует в каждой векторной команде за исключением одиночных `ftw` и `wtw`. В левой части векторной команды выполняется операция загрузки данных во внутренний буфер `ram` векторного процессора, в правой данные, проходящие в `ram` по шине данных, дублируются и направляются в рабочую матрицу для выполнения взвешенного суммирования.

4.3 Машинные коды команд процессора

В одной машинной инструкции совмещаются коды двух команд, флаг параллельного исполнения и, возможно, константа, которую использует данная инструкция:

Рис. 4-1 Структура машинной инструкции процессора.



Бит `P` указывает, должен ли процессор ожидать завершения уже работающих векторных команд прежде чем исполнить данную или пытаться выполнить её сразу.

В Табл. 4-2 приведён список скалярных и векторных команд с указанием длины в 32-х разрядных словах.

В первой колонке таблицы указан условный номер типа машинной команды. Эти индексы введены для удобства ссылки на машинные коды и будут в дальнейшем использоваться.

Табл. 4-2 Список типов команд скалярного процессора NeuroMatrix.

№	КОМАНДА	ДЛИНА	СОДЕРЖИМОЕ 1-ГО СЛОВА КОМАНДЫ
1.1	Загрузка/выгрузка регистра в память	1	–
1.2	Загрузка/выгрузка регистра в память	2	Адресное смещение
2.1	Пересылка “регистр-регистр”	1	–
2.2	Загрузка константы в регистр	2	Константа
3.1	Модификация адресного регистра	1	–
3.2	Модификация адресного регистра	2	Константа модификации
3.3	Пустая команда	1	–
3.4	Пустая команда	2	Любая константа
4.1	Переход к подпрограмме	1	–
4.2	Переход к подпрограмме	2	Константа-смещение
4.3	Возврат из прерывания/подпрограммы	1	–

Подобное описание форматов и типов команд приведено в главе 7 и приложении документа «**Архитектура.** NeuroMatrix: Руководство пользователя»

5 Набор Инструкций Языка Ассемблера

В данном разделе приводится структурированный обзор набора инструкций языка ассемблера процессоров NeuroMatrix®. Обзор состоит из двух подразделов: обзор набора скалярных и набора векторных инструкций. Инструкции группируются по типам. Для каждой команды или операции дается ее положение в ассемблерной инструкции (в какой части инструкции левой или правой она может появляться).

Также перечисляются дополнительные инструкции процессоров NeuroMatrix® разных семейств.

5.1 Скалярные инструкции

В данном разделе приводится полный список скалярных инструкций процессора с кратким пояснением.

Все скалярные команды процессора сведены в таблицу. Первый столбец дает пояснение назначения команды, второй описывает форму записи команды, третий и четвертый столбцы содержат размер и тип кодировки команды.

Размер ассемблерной инструкции определяется левой частью инструкции. Если инструкция содержит константу (32 бита), то размер инструкции – 64 бита. Инструкции, использующие константу (32 бита), могут быть только в левой части. Величина сдвига в арифметико-логической операции не считается константой, она записывается в правой части и её поле занимает 6 разрядов.

Если синтаксис определяет только правую часть, в этом случае размер инструкции не определен и обозначается «-».

Поскольку инструкции процессора содержат левую и правую части, та команда или операция, о которой идет речь в конкретной строке таблицы, подчеркнута. Не подчеркнутые части инструкции представляют собой реально встречающиеся операции, приведенные для сохранения представления о её структуре.

Если не оговорено специально, то на место адресного регистра, используемого в описании синтаксических конструкций, может быть поставлен любой адресный регистр. То же относится к регистрам общего назначения.

5.1.1 Пустая команда

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП
Пустая команда	**`nul;`**	1	3.3

Пустая команда	`nul Const;`	2	3.4
Пустая команда в левой части инструкции	`nul with gr0 += gr1;`	-	3.3, 3.4
Пустая команда в левой части инструкции. (*)	`gr0 += gr1;`	-	3.3, 3.4
Пустая команда в левой части инструкции.	`nul Const with gr0 += gr1;`	2	3.4
Пустая команда в правой части инструкции. (*)	`[ar0++] = gr0;`	1	3.3
Пустая команда в правой части инструкции. (**)	`with gr0 = gr1 >> 0;`	-	3.3, 3.4

Примечание

Инструкции, помеченные знаком (*), показывают пример того, что в тех случаях, когда в левой или правой части стоит пустая операция и при этом другая часть инструкции не пуста, пустая операция может быть опущена.

Примечание

В инструкции, помеченной знаком (**), любой тип сдвига на 0 воспринимается как пустая операция. При этом необходимо отметить, что копирования значения регистра не происходит, то есть регистр `gr0` *не будет равен* `gr1`*, его значение останется тем же, что и до операции.*

5.1.2 Команды чтения из памяти

Команда чтения из памяти может располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП

Прямое чтение из памяти в регистр (или в регистровую пару).	`ar0 = [Const];`
	`ar0 = [Const] with gr1 = gr2 and gr3;`
	`gr0 = [Const] with gr1 = gr2 A>> 1;`
	`ar0,gr0 = [Const] with gr1 += gr2;` (*)
		2	1.2

Косвенное чтение из памяти в регистр общего назначения по адресу равному сумме адресного регистра и константного выражения.
Адресный регистр не изменяется. (**)(*)
	`gr0 = [ar1+Const];`
	`ar2 = [ar1-Const];`
	`gr0 = [ar1+Const] with gr1 -= gr2;`
	`ar4 = [ar1+Const];`
		1	1.1

Косвенное чтение из памяти в регистр.	`ar0 = [ar1];` `ar0 = [ar1] with gr1 -= gr2;`	1	1.1

	pc = [ar1] with gr1 = not gr2; ar0,gr0 = [ar1] with gr1 += gr2;(*)		

Косвенное чтение из памяти в регистр (адресация по регистру общего назначения).	**ar0 = [gr4];**
	ar0 = [gr4] with gr1 = -gr2;
	gr0 = [gr1] with gr1 = gr2 << 10;
	ar0,gr0 = [gr1] with gr1 = gr2 + gr3;(*)
		1	1.1

Косвенное чтение из памяти в регистр с пост инкрементацией адреса	**ar0 = [ar1++];**
	ar0 = [ar1++] with gr1 -= gr2;
	gr0 = [ar1++];
	ar0,gr0 = [ar1++] with gr1 += gr2;(*)
		1	1.1

Косвенное чтение из памяти в регистр с пре декрементацией адреса	**ar0 = [--ar1];**
	ar0 = [--ar1] with gr1 = gr2 and gr3;
	gr0 = [--ar1] with gr1 -= gr2;
	ar0,gr0 = [--ar1] with gr1 += gr2;(*)
		1	1.1

Косвенное чтение из памяти в регистр с адреса равного сумме адресного регистра и регистра общего назначения.
Адресный регистр не меняется. (**)(*)
	gr0 = [ar1+gr1];
	ar0 = [ar1+gr1];
	gr0 = [ar3+gr3] with gr1;
	ar1,gr1 = [ar1+gr1] with gr2 = -gr2;(*)
		1	1.1

Косвенное чтение из памяти в регистр с пост инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**ar0 = [ar1++gr1];**
	ar0 = [ar1++gr1] with gr1 -= gr2;
	gr0 = [ar1++gr1] with gr1 -= gr2;
	ar0,gr0 = [ar1++gr1] with gr1 += gr2;(*)
		1	1.1

Косвенное чтение из памяти в регистр с пре инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**ar0 = [ar1+=gr1];**
	ar0 = [ar1+=gr1] with gr1 -= gr2;
	gr0 = [ar1+=gr1] with gr1 -= gr2;
	ar0,gr0 = [ar1+=gr1];(*)
		1	1.1

Косвенное чтение из памяти в регистр с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**ar0 = [ar1=gr1];**
	ar0 = [ar1=gr1] with gr1 -= gr2;
	gr0 = [ar1=gr1] with gr1 -= gr2;
	ar0,gr0 = [ar1=gr1] with gr3;(*)
		1	1.1

Косвенное чтение из памяти в регистр с предварительной записью в адресный регистр адреса, заданного константным выражением.	**ar0 = [ar1=Const];**
	ar0 = [ar1=Const] with gr1 -= gr2;
	gr0 = [ar1=Const] with gr1 -= gr2;
	ar0,gr0 = [ar1=Const] with gr7 + gr2;(*)
		2	1.1

Косвенное чтение из памяти в регистр с пре инкрементацией адреса на величину константного выражения.	**ar0 = [ar1+=Const];**
	sir = [ar1-=Const];
	ar0 = [ar1+=Const] with gr1 -= gr2;
	gr0 = [ar1+=Const] with gr1 -= gr2;
	ar0,gr0 = [ar1+=Const] with gr1;(*)
		2	1.1

Примечание

В инструкциях, помеченных знаком (*), используются регистровые пары. Они образуются адресным регистрами и регистрами общего назначения с одинаковым номерами, например, (`ar3, gr3`*) или (*`ar5, gr5`*). Регистры с разными номерами не могут образовывать регистровую пару. Регистровая пара содержит 64-х разрядное слово, младшая часть содержится в адресном регистре, старшая в регистре общего назначения. Более подробную информацию о регистровых парах см. в 3.1.3 . В инструкциях чтения или записи роль пар двоякая: во-первых, пара может быть источником\приёмником, во-вторых, есть режимы адресации, задействующие пару в адресном выражении.*

Примечание

Тип команды чтения: чтение 64 бит или 32 бита, определяется регистром приемником.

Примечание

Инструкции, помеченные (**) отсутствовали в NM6403, первом процессоре семейства NeuroMatrix .

5.1.3 Команды записи в память

Команда записи в память может располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП

Прямая запись в память из регистра (или пары).	**`[Const] = ar0;`**
	`[Const] = ar0 with gr1 = gr2 and gr3;`
	`[Const] = gr0 with gr1 = gr2 A>> 1;`
	`[Const] = ar0,gr0 with gr1 += gr2;(*)`
		2	1.2

Косвенная запись в память из регистра по адресу равному сумме адресного регистра и константного выражения.
Адресный регистр не меняется. (**)(*)
	`[ar1+Const] = gr2;`
	`[ar1-Const] = ar0,gr0;`
	`[ar1+Const] = gr0 with gr1 -= gr2;`
	`[ar0-Const] = ar1 with gr1 -= gr2;`
		2	1.2

Косвенная запись в память из адресного регистра.	**`[ar0] = ar1;`**
	`[ar0] = ar1 with gr1 -= gr2;`
	`[ar1] = gr0 with gr1 = not gr2;`
	`[ar1] = ar0,gr0 with gr1 += gr2;(*)`
		1	1.1

Косвенная запись в память из регистра (адресация по регистру общего назначения).	**[gr0] = ar4;** [gr0] = ar4 with gr1 = -gr2; [gr0] = gr1 with gr1 = gr2 << 10; [gr1] = ar0,gr0 with gr1 = gr2 or gr3;	1	1.1
Косвенная запись в память из регистра с пост инкрементацией адреса.	**[ar0++] = ar1;** [ar0++] = ar1 with gr1 -= gr2; [ar1++] = gr0 with gr1 -= gr2; [ar1++] = ar0,gr0 with gr1 += gr2;(*)	1	1.1
Косвенная запись в память из регистра с пре декрементацией адреса.	**[--ar1] = ar0;** [--ar1] = ar0 with gr1 -= gr2; [--ar1] = gr0 with gr1 -= gr2; [--ar1] = ar0,gr0 with gr1 += gr2; (*)	1	1.1
Косвенная запись в память из регистра по адресу равному сумме адресного регистра и регистра общего назначения. Адресный регистр не меняется. (**)(*)	**[ar1+gr1] = gr0;** [ar1+gr1] = gr0 with gr1 -= gr2; [ar1+gr1] = ar0,gr0;(*)	1	1.1
Косвенная запись в память из регистра с пост инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1++gr1] = ar0;** [ar1++gr1] = ar0 with gr1 -= gr2; [ar1++gr1] = gr0 with gr1 -= gr2; [ar1++gr1] = ar0,gr0 with gr1 += gr2;	1	1.1
Косвенная запись в память из регистра с пре инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1+=gr1] = ar0;** [ar1+=gr1] = ar0 with gr1 -= gr2; [ar1+=gr1] = gr0 with gr1 = gr2 >> 4; [ar1+=gr1] = ar0,gr0 with gr1 += gr2;(*)	1	1.1
Косвенная запись в память из регистра с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**[ar1=gr1] = ar0;** [ar1=gr1] = ar0 with gr1 -= gr2; [ar1=gr1] = gr0 with gr1 -= gr2; [ar1=gr1] = ar0,gr0 with gr1 += gr2;(*)	1	1.1
Косвенная запись в память из регистра с предварительной записью в адресный регистр адреса, заданного константным выражением.	**[ar1=Const] = ar0;** ar0 = [ar1=Const] with gr1 -= gr2; [ar1=Const] = gr0 with gr1 -= gr2; [ar1=Const] = ar0,gr0 with gr7;(*)	2	1.2
Косвенная запись в память из регистра с пре инкрементацией адреса на величину константного выражения.	**[ar1+=Const] = ar0;** **[ar1-=Const] = ar0;** [ar1+=Const] = ar0 with gr1 -= gr2; [ar1+=Const] = gr0 with gr1 -= gr2; [ar1+=Const] = ar0,gr0 with gr1;(*)	2	1.2

Примечание *В инструкциях, помеченных знаком (*), используются регистровые*

пары. Они образуются адресным регистрами и регистрами общего назначения с одинаковым номерами, например, (`ar3, gr3`*) или (*`ar5, gr5`*). Регистры с разными номерами не могут образовывать регистровую пару. Регистровая пара содержит 64-х разрядное слово, младшая часть содержится в адресном регистре, старшая в регистре общего назначения. Более подробную информацию о регистровых парах см. в 3.1.3 . В инструкциях чтения или записи роль пар двоякая: во-первых, пара может быть источником\приёмником, во-вторых, есть режимы адресации, задействующие пару в адресном выражении.*

Примечание *Тип команды чтения: чтение 64 бит или 32 бита, определяется регистром источником.*

Примечание *Инструкции, помеченные (**) отсутствовали в NM6403, первом процессоре семейства NeuroMatrix .*

5.1.4 Команды работы со стеком

Команды работы со стеком могут располагаться только в левой части ассемблерной инструкции. В стеке могут быть сохранены и восстановлены только значения регистров доступных как по чтению, так и по записи (см. Табл. 5-1).

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Запись адресного регистра в вершину стека.	**`push ar0;`** `push ar0 with gr7 = gr1 + gr2;`	1	1.1
Запись регистра общего назначения в вершину стека.	**`push gr0;`** `push gr0 with gr7 = gr0;`	1	1.1
Запись регистровой пары в вершину стека.(*)	**`push ar0,gr0;`** `push ar0,gr0 with gr1 = not gr1;`	1	1.1
Чтение адресного регистра из вершины стека.	**`pop ar0;`** `pop ar0 with gr7 = gr0 << 2;`	1	1.1
Чтение регистра общего назначения из вершины стека.	**`pop gr0;`** `pop gr0 with gr7 = gr0;`	1	1.1
Чтение регистровой пары из вершины стека.(*)	**`pop ar0,gr0;`** `pop ar0,gr0 with gr1 += gr2;`	1	1.1

Примечание *В инструкции, помеченные знаком (*), сохраняют в стеке 64-х*

разрядные слова, всегда оставляя указатель на вершину стека четным. Требование четности вершины стека возникает из того, что в случае прерывания и возврата из него программа может вернуться в правильное место только при условии, что указатель стека в момент прерывания был четным. Поскольку прерывание может случиться в любой момент, стек всегда необходимо держать четным. Отсюда предпочтительнее при работе со стеком использовать инструкции, сохраняющие и восстанавливающие регистровые пары.

5.1.5 Команды копирования регистров

Команды копирования производят простое копирование значения из регистра-источника в регистр-приемник. В качестве регистра-источника может выступать любой регистр процессора, доступный по чтению, в качестве регистра-приемника любой регистр, доступный по записи.

Примечание

Пересылка «память-память» не поддерживается процессором. Копирование константы в регистр называется: «инициализация регистра константой» (см. 5.1.6).

В Табл. 5-1 приводится список регистров и регистровых пар процессора NM6403, доступных как по чтению, так и по записи:

Табл. 5-1 Регистры и регистровые пары, доступные по чтению/записи.

КОПИРОВАНИЕ 32-РАЗРЯДНЫХ СЛОВ		КОПИРОВАНИЕ 64-РАЗРЯДНЫХ СЛОВ
ar0, ... ar7(sp)	gr0, ... gr7	(ar0, gr0), ... (ar7, gr7)
icc0	ica0	(icc0, ica0)
icc1	ica1	(icc1, ica1)
occ0	oca0	(occ0, oca0)
occ1	oca1	(occ1, oca1)
t0	t1	(t0, t1)
pswr	pc	(pswr, pc)
lmicr	gmicr	-

Любой из регистров, приведенных в колонке "Копирование 32-разрядных слов" может быть как регистром-источником, так и регистром-приемником, то же самое можно сказать о регистровых парах.

В Табл. 5-2. приводится список регистров процессора NM6403, доступных только по записи:

Табл. 5-2 Регистры, доступные только по записи.

ЗАПИСЬ 32-РАЗРЯДНЫХ СЛОВ		ЗАПИСЬ 64-РАЗРЯДНЫХ СЛОВ
nb1l	nb1h	nb1
sbl	sbh	sb
f1crl	f1crh	f1cr
f2crl	f2crh	f2cr
vrl	vrh	vr

В разряд доступных только по записи попадают специальные регистры управления векторным процессором. Все они 64-х разрядные, поэтому для доступа к ним предусмотрен отдельный доступ к их младшим и старшим частям. Все младшие части помечены индексом `l`, старшие индексом `h`.

Единственный регистр, доступный только по чтению - 32-х разрядный `intr`.

Команды копирования регистров могут располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Копирование значений адресных регистров в адресный регистр. (*)	**ar2 = ar0;** **ar2 = ar0 set;** ar2 = ar0 with gr1 = gr2 and gr3;	1	2.1
Копирование значения адресного регистра в регистр общего назначения.	**gr0 = ar3;** gr0 = ar3 with gr1 = gr0 A>> 1;	1	2.1
Копирование значения регистра общего назначения в адресный регистр. (**)	**ar0 = gr5;** **ar0 = gr5 set;** ar0 = gr5 with gr1 = gr0 and gr5;	1	2.1
Копирование значения регистра общего назначения в регистр общего назначения.	**gr0 = gr5;** gr0 = gr5 with gr1 = gr0 or gr1;	1	2.1
Копирование значения регистровой пары в регистровую пару.	**ar0,gr0 = ar4,gr4;** ar0,gr0 = ar4,gr4 with not gr1;	1	2.1
Копирование значения адресного регистра в регистровую пару. Одно и тоже значение копируется в оба регистра. (**)	**ar0,gr0 = ar5;** ar0,gr0 = ar5 with gr1++;	1	2.1

Копирование значения регистра общего назначения в регистровую пару. Одно и тоже значение копируется в оба регистра. (**)	**ar0,gr0 = gr4;** ar0,gr0 = gr4 with gr2 = gr1 - 1;	1	2.1
Копирование значения адресного регистра в регистр-посредник (32 бита) для последующей пересылки в специальный векторный регистр управления.	**sir = ar5;** nbil = ar5 with gr0 += gr1;	1	2.1
Копирование значения регистра-посредника в младшую/старшую часть (32 бита) специального векторного регистра управления.	**vrh = sir;** vrh = gr4 with gr1 = gr1 << 3;	1	2.1

Примечание

В инструкциях копирования регистром-приемником и регистром-источником могут быть любые адресные регистры независимо от того, принадлежат они одной или разным регистровым группам (см. 3.1.1. Адресные регистры*).*

Примечание

В инструкциях, помеченных знаком (*), дополнительно к обычной форме присваивания, записываемой выражением ar0 = ar2, может быть использован специальный суффикс set. Он используется для того, чтобы показать, что данная команда является командой копирования в отличие от команды модификации адресного регистра, которая соответствует другой машинной команде, хотя по сути выполняет те же действия. Суффикс set может быть опущен, поскольку ассемблер, встретив такую строку, транслирует ее в команду копирования.

Примечание

В инструкциях, помеченных знаком (**), происходит копирование 32-х разрядного регистра в 64-х разрядный или в регистровую пару. Процессор, встретив такую инструкцию, копирует содержимое 32-х разрядного регистра-источника как в младшую, так и в старшую часть 64-х разрядного регистра-приемника, или в одновременно в оба регистра регистровой пары.

5.1.6 Команды инициализации регистров константами

Команды инициализации регистров константами или константными выражениями пересылают 32-х битную константу в регистр-приемник. В качестве регистра-приемника может выступать любой регистр, доступный по записи (см. Табл. 5-1 и Табл. 5-2 в предыдущем пункте).

Команды инициализации регистров константами могут располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Инициализация константой адресного регистра (*)	**ar2 = Const;** **ar2 = Const set;** `ar2 = Const with gr1--;`	2	2.2
Инициализация константой регистра общего назначения.	**gr0 = Const;** `gr0 = Const with gr1 = gr0 A>> 1;`	2	2.2
Инициализация константой регистровой пары. (**)	**ar2,gr2 = Const;** `ar2,gr2 = Const with gr1++;`	2	2.2
Инициализация константой специального регистра.	**sir = Const;** `occ1 = Const with gr1 = - gr2;`	2	2.2

Примечание

В инструкции, помеченной знаком (*), дополнительно к обычной форме присваивания, записываемой выражением ar0 = Const, может быть использован специальный суффикс set. Он используется для того, чтобы показать, что данная команда является командой копирования в отличие от команды модификации адресного регистра, которая соответствует другой машинной команде, хотя по сути выполняет те же действия. Суффикс set может быть опущен, поскольку ассемблер, встретив такую строку, транслирует ее в команду копирования.

Примечание

В инструкциях, помеченных знаком (**), происходит копирование 32-х разрядной константы в 64-х разрядный регистр или в регистровую пару. Процессор, встретив такую инструкцию, копирует константу как в младшую, так и в старшую часть 64-х разрядного регистра-приемника или в оба регистра регистровой пары.

5.1.7 Команды модификации адресных регистров

Команда модификации адресного регистра может располагаться только в левой части ассемблерной инструкции.

В командах модификации адресного регистра важную роль имеет разбиение адресных регистров на группы (см. 3.1.1). Все адресные регистры разбиты на две группы, т.к. процессор имеет два генератора адресов (DAG). К первой группе относятся регистры `ar0,...,ar3`, они соответствуют DAG1, а ко второй `ar4,...,ar7`, они соответствуют DAG2 см. Рис. 1-3 Блочная структура скалярного процессора NM6403.

В команде модификации могут встречаться только адресные регистры из одной группы, например, команда

```
ar0 = ar2+gr2; // правильная команда
```

является корректной, тогда как команда

```
ar0 = ar4+gr4; // ошибочная команда
```

не будет пропущена ассемблером и будет рассматриваться, как ошибочная.

Разделение на группы не касается регистров общего назначения, то есть любой адресный регистр может быть модифицирован любым регистром общего назначения.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Модификация адресного регистра значением другого адресного регистра.(*)	**ar3 = ar0 addr;** ar3 = ar0 addr with gr1 = gr2 and gr3;	1	3.1
Модификация адресного регистра значением регистра общего назначения.(*)	**ar0 = gr7 addr;** ar0 = gr7 addr with gr1 = gr2 xor gr3;	1	3.1
Модификация адресного регистра значением суммы регистров регистровой пары.	**ar5 = ar6+gr6;** ar5 = ar6+gr6 with gr1 = gr2 - gr3;	1	3.1
Модификация адресного регистра значением суммы адресного регистра и константы.	**ar4 = ar6+Const;** ar4 = ar6+Const with gr1 = gr2 C>> 1;	2	3.2
Модификация адресного регистра значением константы.(*)	**ar1 = Const addr;** ar5 = ar6+Const with gr1 = gr2;	2	3.2
Инкрементация адресного регистра.(**)	**ar4++;** ar4++ with gr1 += gr2;	2	3.1
Декрементация адресного регистра. (**)	**ar4--;** ar4-- with gr1 = - gr2;	2	3.1
Инкрементация адресного регистра значением парного ему регистра общего назначения.	**ar2+=gr2;** ar2+=gr2 with gr1 = not gr2;	1	3.1
Инкрементация адресного регистра значением константы.	**ar4+=Const;** ar4+=Const with gr1 -= gr2;	2	3.2
Декрементация адресного регистра значением константы.	**ar4-=Const;** ar2-=Const with gr1;	2	3.2

Примечание

В инструкциях, помеченных знаком (*), команда модификации адресного регистра сопровождается суффиксом addr. Он используется для того, чтобы показать, что данная команда является командой модификации адресного регистра в отличие от команды копирования (см. 5.1.5), которая соответствует другой машинной команде, хотя по сути выполняет те же действия. По умолчанию при отсутствии суффиксов в данной команде ассемблер предполагает использование суффикса set и транслирует ее в машинную инструкцию 3.1/3.2 (см. 4.3).

(**) *Это просто синонимы для* **`ari+=1;`** *и* **`ari-=1;`** *. Использовать не рекомендуется, обратите внимание, что размер равен 2.*

5.1.8 Команды модификации регистра pswr

Регистр `pswr` имеет большое значение, он описывает состояние процессора, входит в блок регистров управления процессором.

Хотя регистр `pswr` доступен как по чтению, так и по записи, его модификацию рекомендуется производить при помощи специальных ассемблерных инструкций.

В систему команд входят две функции модификации `pswr`. Обе они располагаются в левой части скалярной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Установка в единицу определенных битов регистра `pswr`.	**`pswr set Const;`** `pswr set Const with gr1++;`	2	2.3
Сброс битов регистра `pswr`, определяемых константой, т.к. выполняется логическая операция побитового И НЕ содержимого `pswr` с константой.	**`pswr clear Const;`** `pswr clear Const with gr1--;`	2	2.3

5.1.9 Команды перехода

Под собирательным понятием команд перехода понимаются команды, нарушающие выполнение последовательно расположенных инструкций с целью перейти к другому набору, находящемуся в каком-либо другом месте памяти.

К командам перехода относятся:

- команды перехода на другой адрес;
- команды вызова подпрограммы;
- команды возврата из подпрограммы;

- команды возврата из прерывания.

Команды перехода разделяются на два типа. К первому относятся команды обычного перехода, ко второму команды отложенного. Любая команда перехода может быть как обычной, так и отложенной.

Все команды перехода делятся на безусловные и условные. Под безусловным понимается переход, который будет выполнен всегда, когда данная инструкция будет прочитана процессором. Условный переход будет выполнен только в том случае, если флаги, установленные в регистре `pswr`, свидетельствуют о необходимости такого перехода.

Только арифметические и логические операции в правой части ассемблерной инструкции изменяют состояние флагов.

5.1.9.1 Об отложенных (delayed) командах

Для отличия отложенного перехода от обычного перед командой перехода пишется служебное слово `delayed`. Разделение на обычный и отложенный переход введено искусственно, для удобства программирования. От момента выбора команды перехода и до того, как состоится реальный переход, проходит от одного до трёх тактов. За это время процессор успевает выбрать дополнительно одну три инструкции, следующих непосредственно за инструкцией перехода. Назовём такие инструкции отложенными.

При использовании обычного, не отложенного перехода, ассемблер автоматически вставит nul-ы в качестве отложенных команд.

Отложенные инструкции, помещенные после команды перехода, выполняются до того, как произойдет переход. Они будут исполнены в любом случае, независимо от того, выполнилось условие перехода или нет.

Определение точного количества отложенных инструкций в зависимости от: длины, расположения в памяти и типа команды перехода:

Если команда перехода длинная, или короткая, оказавшаяся по нечетному адресу, то слотов два, и там помещается одна длинная команда или две коротких. Если же команда перехода короткая и размещена она по четному адресу, то слотов три и там либо три короткие команды, либо короткая а за ней длинная. (Длинная не может быть первой, поскольку длинные команды выравниваются по четному адресу).

5.1.9.2 Команды безусловного перехода

Команды безусловного перехода могут располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Безусловный переход по абсолютному адресу, задаваемому константой, значение которой вычисляется на этапе компиляции.	**goto Const_addr;** goto Const_addr with gr1++;	2	4.2
Безусловный переход по абсолютному адресу, задаваемому адресным регистром.	**goto ar0;** goto ar0 with gr1--;	1	4.1
Безусловный переход по абсолютному адресу, задаваемому регистром общего назначения.	**goto gr0;** goto gr0 with gr1 = -gr1;	1	4.1
Безусловный переход по абсолютному адресу, задаваемому суммой значений пары регистров.	**goto ar0+gr0;** goto ar0+gr0 with gr1 = not gr1;	1	4.1
Безусловный переход по абсолютному адресу, задаваемому суммой значений адресного регистра и константы.	**goto ar0+Const;** **goto ar0-Const;** goto ar0+Const with gr1 = gr1 << 2;	2	4.2
Безусловный переход по относительному смещению от текущего адреса, задаваемому константой. Значение смещения вычисляется на этапе компиляции.	**skip Const_addr;** skip Const_addr with gr1 = gr2 and gr3;	2	4.2
Безусловный переход по относительному смещению от текущего адреса, задаваемому регистром общего назначения.(*)	**skip gr0;** skip gr0 with gr1--;	1	4.1
Отложенный безусловный переход.(**)	**delayed goto gr0;** delayed goto gr0 with gr1++;	1 или 2	4.1, 4.2

Примечание

Инструкция, помеченной знаком (**), показывает пример записи отложенного безусловного перехода. Ключевое слово delayed может быть использовано с любым из описанных выше типов безусловного перехода, превратив его в отложенный.

5.1.9.3 Команды перехода к подпрограмме

В данном разделе рассматриваются только команды безусловного перехода к подпрограмме. Для того, чтобы сконструировать инструкцию условного перехода к подпрограмме, необходимо условие, описываемое в подпункте 5.1.9.5 Набор условий перехода, добавить к команде безусловного перехода. Например, если команда безусловного перехода к подпрограмме выглядит так:

```
call MyFunc;
```

то после добавления условия команда условного перехода к подпрограмме будет иметь вид:

```
if =0 call MyFunc;
```

Команды перехода к подпрограмме могут располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому константой (меткой).	**call Const_addr;** call Const_addr with gr1++;	2	4.2
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому адресным регистром.	**call ar0;** call ar0 with gr1--;	1	4.1
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому регистром общего назначения.	**call gr0;** call gr0 with gr1 = -gr1;	1	4.1
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому суммой значений пары регистров.	**call ar0+gr0;** call ar0+gr0 with gr1 = not gr1;	1	4.1
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому суммой значений адресного регистра и константы.	**call ar0+Const;** **call ar0-Const;** goto ar0+Const with gr1 = gr1 << 2;	2	4.2
Безусловный переход к подпрограмме по относительному смещению от текущего адреса, задаваемому константой. Значение смещения вычисляется на этапе компиляции.	**callrel Const_addr;** callrel Const_addr with gr1++;	2	4.2
Безусловный переход к подпрограмме по относительному смещению от текущего адреса, задаваемому регистром общего назначения.	**callrel gr0;** callrel gr0 with gr1--;	1	4.1
Отложенный безусловный переход к подпрограмме.(**)	**delayed call gr0;** delayed call gr0 with gr1++;	1 or 2	4.1, 4.2

Примечание

Инструкция, помеченной знаком (**), показывает пример записи отложенного безусловного перехода к подпрограмме. Подробнее см. примечание из предыдущего подпункта 5.1.9.1.

5.1.9.4 Команды возврата из подпрограммы/прерывания

Команды возврата из подпрограммы/прерывания могут располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Возврат из подпрограммы.	**return;** return with gr7 = gr1 + gr2;	1	4.3
Возврат из прерывания.	**ireturn;** ireturn with gr7 = gr0;	1	4.3
Отложенный возврат из подпрограммы.	**delayed return;** delayed return with gr7 = gr1 + gr2;	1	4.3
Отложенный возврат из прерывания.	**delayed ireturn;** delayed ireturn with gr7 = gr0;	1	4.3

5.1.9.5 Набор условий перехода

Команды условного перехода, вызова функции или возврата из функции/прерывания выполняются или не выполняются в зависимости от того, какие флаги в регистре pswr были установлены одной из предыдущих команд. Установка флагов происходит только путем выполнения арифметическо-логической операции в правой части скалярной команды.

Таким образом, для того, чтобы условный переход стал возможен, сначала выполняется скалярная арифметическая или логическая команда или операция сдвига, устанавливающая значение флагов в pswr, а затем уже сам условный переход. Например:

```
begin "text"
...
gr2 = [ar0++] with gr0--; // Команда, устанавливающая
                         // флаги в регистре pswr.
if > goto Label;         // Условный переход на
                         // метку Label.
...
end "text";
```

Команды условного перехода могут располагаться только в левой части ассемблерной инструкции.

В графе "Синтаксис команды" подчеркнутая часть инструкции показывает то, как записывается условие. Мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой правой частью. В графе «Условие перехода» при записи флагов Z,N,V,C эквивалентна записи (Z=1),(N=1),(V=1),(C=1)

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	УСЛОВИЕ ПЕРЕХОДА, ФЛАГ

Равно нулю.	**if =0 goto ...;** if =0 goto Label with gr1++;	Z
Не равно нулю.	**if <>0 goto ...;** if <>0 goto gr0 with gr1--;	~Z
Знаковое больше.	**if > delayed goto ...;** if > delayed goto ar0 with gr1--;	~Z AND ~N
Знаковое меньше.	**if < skip ...;** if < skip Label with gr1--;	N
Знаковое больше или равно.	**if >= call ...;** if >= call Label with gr1--;	~N
Знаковое меньше или равно.	**if <= callrel ...;** if <= callrel Label with gr1--;	N OR Z
Беззнаковое больше или равно.	**if u>= goto ...;** if >= goto Label with gr7 -= gr1;	C
Беззнаковое меньше.	**if u< return ...;** if u< return with gr7 = gr1 noflags;	~C
Произошел перенос (бит C = 1).	**if carry call ...;** if carry call ar0 with gr7 -= gr1;	C
Не произошел перенос (бит C = 0).	**if not carry return ...;** if not carry return with gr7 = gr1;	~C
Возникло переполнение (бит V = 1).	**if vtrue call ...;** if vtrue call ar0 with gr7 -= gr1;	V
Не возникло переполнение (бит V = 0).	**if vfalse return ...;** if vfalse return with gr7 = gr1;	~V
Знаковое больше с проверкой бита переполнения.	**if v> goto ...;** if v> goto ar0 with gr7 -= gr1;	~ ((N XOR V) OR Z)
Знаковое меньше с проверкой бита переполнения.	**if v< delayed goto ...;** if v< delayed goto gr1 with gr7 = gr1;	N XOR V
Знаковое больше или равно с проверкой бита переполнения.	**if v>= callrel ...;** if v>= callrel gr0 with gr7 -= gr1;	~ (N XOR V)
Знаковое меньше или равно с проверкой бита	**if v<= ireturn ...;** if v<= ireturn with gr7 = gr1;	(N XOR V) OR Z

переполнения.		

5.1.10 Основные скалярные операции процессора

Все скалярные вычислительные операции процессора располагаются в правой части ассемблерной инструкции, то есть после ключевого слова `with`. В выполнении скалярных операций могут использоваться только регистры общего назначения.

Скалярные операции являются трехоперандными. Любой регистр общего назначения может располагаться как слева, так и справа от знака равенства, например:

- `gr0 = gr1 + gr2;` - слева от знака равенства;
- `gr1 = gr0 + gr2;` - справа от знака равенства;
- `gr0 = gr0 + gr0;` - слева и справа от знака равенства.

Структура любой скалярной операции допускает следующие варианты действий:

- Если скалярная операция имеет форму присваивания, например:
  `gr1 = gr2 + gr3;`

  то результат вычислений, заданных в правой части, помещается в регистр, указанный в левой части.

- Если справа от операции приписано служебное слово 'noflags', например:

  ```
  gr1 = gr2 + gr3 noflags;
  ```

  то это означает запрет модификации флагов из регистра состояния процессора по результатам выполнения операции. Иными словами, состояние поля флагов до выполнения команды остается без изменений. В операциях сдвига служебное слово 'noflags' использоваться не может. Любая (кроме nul) скалярная операция без 'noflags' изменит значение флагов.

- Наконец, скалярная операция может иметь форму простого выражения, например:

  ```
  gr2 + gr3;
  ```

  или просто

  ```
  gr2;
  ```

  В этом случае результат вычисления выражения нигде не запоминается и влияет только на установку флагов. Для операций сдвига использование формы простого выражения (без присваивания) не допускается.

5.1.11 Арифметические операции

Арифметические операции могут располагаться только в правой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой левой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Сумма значений двух регистров.	**gr0 = gr1 + gr2;** ar0 = gr0 with gr0 = gr1 + gr2;	6.3
Сумма значений двух регистров (сокращённая форма записи).	**gr0 += gr1;** эквивалентно gr0 = gr0 + gr1; ar0 = 100h with gr0 += gr1;	6.3
Сумма значения регистра с единицей.	**gr1 = gr2 + 1;** ar1+=gr1 with gr1 = gr2 + 1;	6.3
Инкрементация значения регистра.	**gr1++;** эквивалентно gr1 = gr1 + 1; ar1++ with gr1++;	6.3
Разность значений двух регистров.	**gr1 = gr0 – gr7;** [ar1++] = ar0 with gr1 = gr0 – gr7;	6.3
Разность значений двух регистров (сокращённая форма записи).	**gr1 -= gr7;** эквивалентно gr1 = gr1 - gr7; [--ar1] = gr0 with gr1 -= gr7;	6.3
Вычитание единицы из значения регистра.	**gr1 = gr2 - 1;** call gr4 with gr1 = gr2 - 1;	6.3
Декрементация значения регистра.	**gr1--;** эквивалентно gr1 = gr1 - 1; ar1-- with gr1--;	6.3
Добавление значения флага переноса к значению регистра	**gr1 = gr2 + carry;** ar4 += gr4 with gr1 = gr2 + carry;	6.3
Сложение значений двух регистров с добавлением значения флага переноса.	**gr1 = gr2 + gr6 + carry;** ar4++ with gr1 = gr2 + gr6 + carry;	6.3
Вычитание значения флага переноса из значения регистра.	**gr1 = gr2 - 1 + carry;** ar6 -= gr6 with gr1 = gr2 - 1 + carry;	6.3
Вычитание значений двух регистров с учетом значения флага переноса.	**gr1 = gr2 - gr6 - 1 + carry;** ar4–with gr1 = gr2 - gr6 - 1 + carry;	6.3

Изменение знака регистра.	**gr1 = - gr5;** goto gr4 with gr1 = - gr5;	6.3
Первый шаг многошагового умножения(*), вторым источником-операндом должен быть регистр gr7.	**gr1 = gr2 *: gr7;** ar4,gr4 = gr0 with gr1 = gr2 *: gr7;	6.3
Последующие шаги многошагового умножения (*), вторым источником-операндом должен быть регистр gr7.	**gr1 = gr2 * gr7;** [ar6] = gr6 with gr1 = gr2 * gr7;	6.3
Пример выполнения арифметической операции без изменения значения флагов в регистре pswr.	**gr1 = gr2 + gr2 noflags;** ar5++ with gr1 = gr2 + gr2 noflags;	6.3
Пример выполнения арифметической операции с установкой желаемых флагов без изменения значения регистров.	**gr1 + gr2;** [ar1++] = AAA with gr1 + gr2;	6.3

Примечание

Чтобы перемножить два 32-х битных числа с помощью инструкций скалярного умножения(*), надо выполнить одну инструкцию «первого шага» и пятнадцать «последующих шагов» подряд, с одинаковыми аргументами, например, так:

```
    with gr1 = gr0 *: gr7;
.repeat 15;
    with gr1 = gr0 * gr7;
.endrepeat;
```

После выполнения приведённого кода в регистре приёмнике (gr1) окажутся старшие 32 бита произведения, младшие же – в **gr7**

5.1.12 Логические операции

Логические операции могут располагаться только в правой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой левой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Побитовая операция OR двух операндов.	**gr0 = gr1 or gr2;** ar0 = gr0 with gr0 = gr1 or gr2;	6.2
Побитовая операция AND операндов.	**gr1 = gr2 and gr3;** ar0 = 100h with gr1 = gr2 and gr3;	6.2
Побитовая операция XOR двух операндов.	**gr2 = gr3 xor gr4;** ar0 = gr0 with gr2 = gr3 or gr4;	6.2
Операция отрицания.	**gr1 = not gr2;**	6.2

	`ar0 = 100h with gr1 = not gr2;`	
Комбинация операции отрицания первого операнда с побитовой операцией OR двух операндов.	**`gr0 = not gr1 or gr2;`** `ar0++ with gr0 = not gr1 or gr2;`	6.2
Комбинация операции отрицания второго операнда с побитовой операцией OR двух операндов.	**`gr1 = gr2 or not gr3;`** `ar6-- with gr1 = gr2 or not gr3;`	6.2
Комбинация операции отрицания обоих операндов с побитовой операцией OR.	**`gr1 = not gr2 or not gr3;`** `return with gr1 = not gr2 or not gr3;`	6.2
Комбинация операции отрицания первого операнда с побитовой операцией AND двух операндов.	**`gr2 = not gr3 and gr4;`** `ar4+=gr4 with gr0 = not gr1 and gr2;`	6.2
Комбинация операции отрицания второго операнда с побитовой операцией AND двух операндов.	**`gr3 = gr4 and not gr5;`** `[ar6]=gr6 with gr3 = gr4 and not gr5;`	6.2
Комбинация операции отрицания обоих операндов с побитовой операцией AND.	**`gr3 = not gr4 and not gr5;`** `[ar6]=gr6 with gr3 = gr4 and not gr5;`	6.2
Комбинация операции отрицания первого операнда с побитовой операцией XOR двух операндов.	**`gr2 = not gr3 xor gr4;`** `[addr]=gr0 with gr2 = not gr3 xor gr4;`	6.2
Запись в регистр общего назначения логического нуля. Обнуляет регистр без использования константы.	**`gr6 = false;`** `[ar0] = ar5,gr5 with gr6 = false;`	6.2
Запись в регистр общего назначения логической -1. Устанавливает все биты регистра в единицу.	**`gr0 = true;`** `ar0=[ar2=10h] with gr0 = true;`	6.2
Копирование значения одного регистра общего назначения в другой.(*)	**`with gr2 = gr4;`** `gr0 = gr5 with gr2 = gr4;`	6.2

Примечание

Операция, помеченная знаком (*), использует ключевое слово `with`, в противном случае компилятор воспримет её как команду копирования и поместит в левую часть ассемблерной инструкции. При этом исчезнет возможность по результату операции установить флаги, так как команды, выполняемые в левой части ассемблерной инструкции не воздействуют на флаги. Сравните:

```
gr2 = gr3;          // команда копирования в левой части
                    // инструкции (например:
                    // gr2 = gr3  with gr4 += gr5;)

with gr2 = gr3;  // логическая операция в правой части
                 // инструкции (например:
                 // [ar0+=5] = ar7 with gr2 = gr3;)
```

5.1.13 Операции Установки Флагов без Изменения Значений Регистров

В данном разделе приводятся арифметические и логические выражения, которые влияют на флаги, использующиеся в командах

условного перехода, однако не меняют значений регистров, поскольку не содержат операции присваивания.

Все приводимые ниже выражения могут располагаться только в правой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой левой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Установка флагов по значению регистра.	**gr0;** if > goto gr2 with gr0;	6.2
Установка флагов по значению регистра с измененным знаком.	**- gr1;** [ar4++gr4] = ar2,gr2 with - gr1;	6.3
Установка флагов по значению суммы регистров.	**gr2 + gr3;** ar0 = 100h with gr2 + gr3;	6.3
Установка флагов по сумме значения регистра с единицей.	**gr4 + 1;** gr2++ with gr4 + 1;	6.3
Установка флагов по значению разности регистров.	**gr0 – gr7;** ar0 += gr0 with gr0 + gr7;	6.3
Установка флагов по разности значения регистра и единицы.	**gr2 – 1;** gr7 = [ar0++gr0] with gr2 - 1;	6.3
Установка флагов по сумме значения регистра с флагом переноса.	**gr1 + carry;** ar4 -= gr4 with gr1 + carry;	6.3
Установка флагов по сумме значений двух регистров с добавлением значения флага переноса.	**gr2 + gr6 + carry;** ar4++ with gr2 + gr6 + carry;	6.3
Установка флагов по результату вычитания значения флага переноса из значения регистра.	**gr2 - 1 + carry;** goto addr with gr2 - 1 + carry;	6.3
Установка флагов по результату вычитания значения двух регистров с учетом значения флага переноса.	**gr2 – gr6 - 1 + carry;** ar4-- with gr2 - gr6 - 1 + carry;	6.3
Установка флагов по результату побитовой операции OR над двумя операндами.	**gr1 or gr2;** Ar0 = gr0 with gr1 or gr2;	6.2
Установка флагов по результату побитовой операции AND над двумя операндами.	**gr2 and gr3;** ar0 = 100h with gr2 and gr3;	6.2
Установка флагов по результату побитовой операции XOR над двумя операндами.	**gr3 xor gr4;** ar0 = gr0 with gr3 or gr4;	6.2
Установка флагов по результату операции отрицания.	**not gr2;** Ar0 = 100h with not gr2;	6.2
Установка флагов по результату выполнения комбинации операции отрицания первого операнда	**not gr1 or gr2;** ar0++ with not gr1 or gr2;	6.2

с побитовой операцией OR двух операндов.		
Установка флагов по результату выполнения комбинации операции отрицания второго операнда с побитовой операцией OR двух операндов.	**gr2 or not gr3;** ar6—with gr2 or not gr3;	6.2
Установка флагов по результату выполнения комбинации операции отрицания обоих операндов с побитовой операцией OR.	**not gr2 or not gr3;** return with not gr2 or not gr3;	6.2
Установка флагов по результату выполнения комбинации операции отрицания первого операнда с побитовой операцией AND двух операндов.	**not gr3 and gr4;** ar4+=gr4 with not gr1 and gr2;	6.2
Установка флагов по результату выполнения комбинации операции отрицания второго операнда с побитовой операцией AND двух операндов.	**gr4 and not gr5;** [ar6]=gr6 with gr4 and not gr5;	6.2
Установка флагов по результату выполнения комбинации операции отрицания обоих операндов с побитовой операцией AND.	**not gr4 and not gr5;** [--ar6]=gr2 with gr4 and not gr5;	6.2
Установка флагов по результату выполнения комбинации операции отрицания первого операнда с побитовой операцией XOR двух операндов.	**not gr3 xor gr4;** [addr]=gr0 with not gr3 xor gr4;	6.2
Установка флагов по результату выполнения комбинации операции отрицания второго операнда с побитовой операцией XOR двух операндов.	**gr2 xor not gr7;** [ar0++]=gr0 with gr2 xor not gr7;	6.2
Установка флага Z в единицу, а остальных в ноль.	**false;** [ar0] = ar5,gr5 with false;	6.2
Установка флага N в единицу, а остальных в ноль.	**true;** ar0=[ar2=10h] with true;	6.2

5.1.14 Операции сдвига

Операции сдвига могут располагаться только в правой части ассемблерной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе скалярной команды с непустой левой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Сдвиг влево на произвольное число битов влево. Константа сдвига в пределах от 1 до 31. Младшие биты заполняются нулями, старшие биты сдвигаются через бит переноса.	**gr0 = gr1 << 10;** ar0 = gr0 with gr0 = gr1 << 10;	6.1
Сдвиг влево на произвольное число битов влево (сокращенная запись). Константа сдвига в пределах от 1 до 31. Младшие биты заполняются нулями, старшие биты сдвигаются через бит переноса.	**gr0 <<= 2;** эквивалентно gr0 = gr0 << 2; [ar0] = gr0 with gr0 <<= 2;	6.1
Беззнаковый сдвиг на произвольное число битов вправо. Константа сдвига в пределах от 1 до 31. Старшие биты заполняются нулями, младшие биты сдвигаются через бит переноса.	**gr1 = gr2 >> 24;** ar0 = 100h with gr1 = gr2 >> 24;	6.1

Беззнаковый сдвиг на произвольное число битов вправо (сокращенная запись). Константа сдвига в пределах от 1 до 31. Старшие биты заполняются нулями, младшие биты сдвигаются через бит переноса.	**gr1 >>= 3;** эквивалентно gr1 = gr1 >> 3; [ar0++] = gr3 with gr1 >>= 3;	6.1
Арифметический сдвиг на произвольное число битов вправо. Константа сдвига в пределах от 1 до 31. Старшие биты заполняются знаком, младшие биты сдвигаются через бит переноса.	**gr2 = gr3 A>> 5;** ar0 += gr0 with gr2 = gr3 A>> 5;	6.1
Арифметический сдвиг на произвольное число битов вправо (сокращенная запись). Константа сдвига в пределах от 1 до 31. Старшие биты заполняются знаком, младшие биты сдвигаются через бит переноса.	**gr2 A>>= 9;** эквивалентно gr2 = gr2 A>> 9; skip addr with gr2 A>>= 9;	6.1
Циклический сдвиг на произвольное число битов влево. Константа сдвига в пределах от 1 до 31.	**gr3 = gr4 R<< 6;** ar0 -= gr0 with gr3 = gr4 R<< 6;	6.1
Циклический сдвиг на произвольное число битов влево (сокращенная запись). Константа сдвига в пределах от 1 до 31.	**gr3 R<<= 12;** эквивалентно gr3 = gr3 R<< 12; goto gr0 with gr3 R<<= 12;	6.1
Циклический сдвиг на произвольное число битов вправо. Константа сдвига в пределах от 1 до 31.	**gr3 = gr4 R>> 7;** [--ar0] = gr0 with gr3 = gr4 R>> 7;	6.1
Циклический сдвиг на произвольное число битов вправо (сокращенная запись). Константа сдвига в пределах от 1 до 31.	**gr3 R>>= 14;** эквивалентно gr3 = gr3 R>> 14; ar5—with gr3 R>>= 14;	6.1
Циклический сдвиг влево через бит переноса. Константа сдвига 1.	**gr4 = gr5 C<< 1;** ar0 -= gr0 with gr4 = gr5 C<< 1;	6.1
Циклический сдвиг влево через бит переноса (сокращенная запись). Константа сдвига 1.	**gr4 C<<= 1;** эквивалентно gr4 = gr4 C<< 1; gr0 = gr7 with gr4 C<<= 1;	6.1
Циклический сдвиг вправо через бит переноса. Константа сдвига 1.	**gr5 = gr6 C>> 1;** [ar0+=2] = gr0 with gr5 = gr6 C>> 1;	6.1
Циклический сдвиг вправо через бит переноса (сокращенная запись). Константа сдвига 1	**gr5 C>>= 1;** эквивалентно gr5 = gr5 C>> 1; ireturn with gr5 C>>= 1;	6.1

5.2 Векторные инструкции

В данном разделе приводится полный список векторных команд процессора с кратким пояснением.

Все векторные инструкции процессора сведены в таблицу. Первый столбец дает пояснение назначения команды, второй

описывает форму записи команды, третий и четвертый столбцы содержат размер и тип кодировки команды.

Если инструкции процессора содержат левую и правую части, та команда или операция, о которой идет речь в конкретной строке таблицы, подчеркнута. Неподчеркнутые части инструкции представляют собой реально встречающиеся операции, приведенные для сохранения представления о её структуре.

5.3 Инструкции векторного сопроцессора чисел с плавающей точкой

В данном разделе приводится список типов инструкций векторного сопроцессора для работы с плавающей точкой. В списке перечислены не все возможные сочетания свойств.

5.3.1 Унарные операции

Унарые операции включают пересылку, обращение знака и нахождение модуля, возможно применение маски. Поддерживаются только типы `.float и .double`.

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП

Копирование. Тип можно не указывать.	**vreg0= vreg1;**
	`fpu 0 .float vreg0= vreg1;`
	`fpu 0 vreg0= vreg1;`
		всегда 1	3.1F

Нахождение модуля.	**vreg0= /vreg1/;** `fpu 0 .float vreg0= /vreg0/;`		3.1F
Нахождение противоположного числа.	**vreg7= -vreg6;** `fpu 0 .float vreg0= -vreg0;`		3.1F
Нахождение числа, противоположного модулю.	**vreg0= -/vreg1/;** `fpu 0 .float vreg0= -/vreg0/;`		3.1F
Копирование по маске.	**vreg0= not mask ? vreg1 : vreg0;** `fpu 0 .float vreg0= not mask ? vreg1 : vreg0;`		3.1F

Унарная операция по маске.	**vreg0= mask ? -/vreg1/ : vreg0;**
	`fpu 0 .float vreg0= mask ? -/vreg1/ : vreg0;`
	`fpu 0 .float vreg0= mask ? -/vreg1/;`
			3.1F

5.3.2 Команда умножения, поэлементное перемножение векторов

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП
Умножение. Почленное перемножение содержимого двух векторных	**vreg7= vreg0 * vreg1;** `fpu 0 .float vreg7= vreg0*vreg1;`	всегда 1	**3.2F**

регистров указанной ячейки	`fpu 1 .double vreg4= vreg4*vreg4;`	
Нахождение противоположного числа от произведения	**`vreg0= - vreg1 * vreg2;`** `fpu 0 .float vreg0= - vreg1*vreg2;`	3.2F
Умножение вектора на скаляр.	**`vreg7= vreg4 * .retrive(vreg0);`** `fpu 0 .complex vreg7= vreg4*.retrive(vreg0);`	3.2F
Умножение с накоплением	**`vreg7= vreg0 * vreg1 +vreg2;`** `fpu 0 .float vreg7= vreg0*vreg1 +vreg2;`	**3.4F**
Умножение вектора на скаляр с вычитанием скаляра	**`vreg7= vreg0 * .retrive(vreg1) - .retrive(vreg2);`** `fpu 0 .complex vreg7= vreg0*.retrive(vreg1) - .retrive(vreg2);`	3.4F
Умножение с записью результата по маске	**`vreg7= mask ? vreg0 * vreg0 :vreg7;`** `fpu 3 .float vreg7= mask ? vreg0*vreg1: vreg7;`	**3.2F**

Умножение вектора на скаляр, прибавление вектора, обращение знака результата, запись по маске	**`vreg3= not mask ? – vreg4 * .retrive (vreg5) – vreg6 : vreg3;`**
	`fpu 0 .float vreg3= not mask ? – vreg4 * .retrive (vreg5) – vreg6 : vreg3;`
	`fpu 0 .float vreg3= not mask ? – vreg4 * .retrive (vreg5) – vreg6;`
		3.4F

5.3.3 Команда умножения, тип `.matrix`

Команда умножения типа `.matrix` работает так: первый сомножитель рассматривается как вектор пар значений типа float. Другой сомножитель занимает два регистра и хранит вектор четвёрок значений типа float, то есть матриц 2x2. Ко второму аргументу можно применить транспонирование – модификатор "**`.trans`**".

Если применить ко второму сомножителю модификатор «**`.retrive`**», то в регистр результата попадёт произведение матриц, - Nx2 из первого аргумента и 2x2 из второго, с получением матрицы Nx2 в регистре результата.

3Схема работы инструкции перемножения матриц Nx2 и 2x2

`vreg2= vreg4 * .retrive(vreg0, vreg1);`

vreg4			**vreg0**	**vreg1**					**vreg2**	
1	0		1	2	3	4	=>	1	2	1	2
0	1		?	?	?	?		3	4	3	4
0	0		?	?	?	?				0	0
10	1							10*1 + 1*3 =		13	24
2	-1									-1	0

3	7
-1	1

3*1 + 7*3 =	24	34
	2	2

4Схема работы инструкции перемножения матриц Nx2 и 2x2 с транспонированием второго сомножителя (2x2)

vreg2= vreg4 * .trans .retrive(vreg0, vreg1);

vreg4		vreg0					vreg1			vreg2	
1	0	1	2	3	4	=>	1	3		1	3
0	1	?	?	?	?		2	4		2	4
0	0	?	?	?	?					0	0
10	1						10*1 + 1*2 =			12	34
2	-1									0	2
3	7						3*1 + 7*2 =			17	37
-1	1									1	1

Операция без «**.retrive**» поэлементно перемножает вектора, то есть, i-ая пара значений из первого сомножителя умножается как строка на i-ую матрицу из второго сомножителя. Полученная пара чисел станет i-ым элементом вектора произведений.

5Схема работы инструкции множественного перемножения матриц 1x2 и 2x2 (что будет, если не использовать «.retrive»)

vreg2= vreg4 * (vreg0, vreg1);

vreg4		vreg0				vreg1				vreg2	
1	0	1	2	3	4					1	2
1	0	-1	2	-3	4					-1	2
1	1	0	0	0	0					0	0
10	1	5	6	7	8					57	68
A_0	A_1	B_{00}	B_{01}	B_{10}	B_{11}					C_0	C_1

$C_0 = A_0 * B_{00} + A_1 * B_{10}$ $C_1 = A_0 * B_{01} + A_1 * B_{11}$

Предполагается, что обычно команда будет использоваться с «**.retrive**», этот модификатор можно сочетать с “**.trans**” (**.trans .retrive**).

Синтаксически команда с типом.matrix отличается от обычного умножения тем, что одним из сомножителей указывается регистровая пара.

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП
Умножение. Почленное перемножение вектора строк 1x2 на вектор матриц 2x2	**vreg7= vreg0 * (vreg4,vreg5);** fpu 0 .matrix vreg7= vreg0*(vreg4,vreg5);	всегда 1	**3.2F**

Умножение матрицы Nx2 на матрицу 2x2.	**vreg1= vreg4 * .retrive(vreg0, vreg1);** fpu 0 .matrix vreg7= vreg4*.retrive(vreg0, vreg1);	3.2F
Умножение матрицы Nx2 на транспонированную матрицу 2x2	**vreg0= vreg0 * .trans .retrive (vreg2,vreg3);** fpu 0 .matrix vreg0= vreg0 * .trans .retrive (vreg2,vreg3);	3.2F
Умножение с обращением знака	**vreg0= - vreg1 * (vreg2,vreg3);** fpu 0 .matrix vreg0= - vreg1*(vreg2,vreg3);	3.2F
Перемножение матриц, результат складывается с третьей матрицей.	**vreg7= vreg0 * .retrive (vreg4,vreg5) +vreg3;** fpu 0 . matrix vreg7= vreg0*.retrive (vreg4,vreg5) +vreg3;	**3.4F**
Перемножение матриц, из результата вычитается константная пара. (если элементы пары равны, то каждый элемент матрицы результата будет уменьшен на эту константу)	**vreg7= vreg0 * .retrive(vreg4, vreg5) -.retrive(vreg3);** fpu 0 . matrix vreg7= vreg0*.retrive(vreg4, vreg5) -.retrive(vreg3);	3.4F
Почленное перемножение с записью результата по маске	**vreg7= mask ? vreg0 * (vreg2, vreg3) :vreg7;** fpu 3 .float vreg7= mask ? vreg0*(vreg2, vreg3): vreg7;	**3.2F**

Умножение вектора строк на вектор матриц, прибавление вектора, обращение знака результата, запись по маске	**vreg3= not mask ? - vreg4 * .retrive (vreg0, vreg1) - vreg6 : vreg3;**
	fpu 0 .float vreg3= not mask ? - vreg4 * .retrive (vreg0, vreg1) - vreg6 : vreg3;
	fpu 0 .float vreg3= not mask ? - vreg4 * .retrive (vreg0, vreg1) - vreg6;
		3.4F

5.3.4 Аддитивные команды и сравнение

Только команды данной группы позволяют сформировать маску для дальнейшего использования в командах условного действия над векторами.

Чтобы сформировать маску, аддитивную команду надо дополнить указанием сформировать маску и предикатом, определяющим, по какому условию в маску будут попадать «1

Кроме маски, данное подмножество команд устанавливает регистры флагов. Флаги формируются по-умолчанию, если их формирование нежелательно, используйте суффикс **noflags**. Нельзя отменить установку флагов, если команда формирует маску.

Если нужны только результаты сравнения, вместо присваивания следует использовать только правую часть.

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП

Поэлементное сложение векторов	**vreg7= vreg0 + vreg1;**
	fpu 0 .float vreg7= vreg0+vreg1;
	fpu 1 .double vreg4= vreg4+vreg4;
		всегда 1	**3.3F**

Поэлементное вычитание векторов	**vreg6= vreg1 - vreg0;** fpu 0 .float vreg6= vreg1-vreg0;		3.3F
Сложение и обращение знака	**vreg6= - vreg1 – vreg0;** fpu 0 .float vreg6= -vreg1-vreg0;		3.3F
Поэлементное вычитание векторов с запрещением формирования флагов	**vreg7= vreg7 – vreg5 noflags;** fpu 0 .double vreg7= vreg7-vreg5 noflags;		3.3F
Формирование флагов по сравнению двух чисел без записи разности в регистр	**vreg1 – vreg2;** fpu 0 .double vreg1-vreg2;		3.3F
Сравнение векторов, установка маски по равенству.	**vreg1 – vreg2, set mask if =0;** fpu 2 .double vreg1-vreg2, set mask if =0;		3.3F
Поэлементное прибавление константы к вектору	**vreg7= vreg0 + .retrive(vreg1);** fpu 0 .float vreg7= vreg0+.retrive (vreg1);		3.3F
Поэлементное сложение векторов, запись результата по маске	**vreg7= mask ? vreg0 + vreg1 : vreg7;** fpu 0 .float vreg7= mask ? vreg0 + vreg1 : vreg7;		3.3F
Поэлементное вычитание векторов, маска используется и формируется новая	**vreg6= mask ? vreg1 - vreg0, set mask if >;** fpu 0 .float vreg6= mask ? vreg1 - vreg0, set mask if >;		3.3F
Поэлементное вычитание вектора из константы	**vreg7= - vreg0 + .retrive(vreg1);** fpu 0 .float vreg7= - vreg0+.retrive (vreg1);		3.3F

5.3.5 Команды чтения и записи в память векторных регистров

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП
Чтение из памяти в векторный регистр тридцати двух 64-разрядных слов	**rep 32 vreg0= [ar0++];** fpu 0 rep 32 vreg0= [ar0++];	всегда 1	**4.1F**
Чтение элементов вектора с режимами адресации: - преинкремент на регистр - постинкремент на регистр - предекремент	**rep 10 vreg1= [ar1+=gr1];** **rep 10 vreg1= [ar1++gr1];** **rep 10 vreg1= [--ar1];** **rep 10 vreg1= [ar1++];**		4.1F

- постинкремент	`fpu 0 rep 10 vreg1= [ar1+=gr1];`	
Адресации, использование которых приводит к заполнению вектора копиями одного 64-х разрядного слова: - адрес из адресного регистра - адрес из РОН - адрес – сумма регистров - адрес из РОН с модификацией адресного	**rep 1 vreg1= [ar3];** **rep 10 vreg1= [gr3];** **rep 20 vreg1= [ar3+gr3];** **rep 30 vreg1= [ar3=gr3];** `fpu 3 rep 10 vreg1= [gr3];`	4.1F
Чтение, количество читаемых слов берётся из vlen.	**rep vlen vreg0= [ar1++];** `fpu 0 vlen vreg0= [ar1++];`	4.1F
Запись в память шестнадцати 64-разрядных слов из векторного регистра	**rep 16 [ar0++]= vreg0;** `fpu 0 rep 16 [ar0++]= vreg0;`	**4.2F**
Запись с использованием всевозможных адресаций	**rep 10 [ar1+=gr1]= vreg0;** **rep 10 [ar1++gr1]= vreg0;** **rep 10 [--ar1]= vreg0;** **rep 10 [ar1++]= vreg0;** **rep 10 [ar1]= vreg0;** **rep 10 [gr1]= vreg0;** **rep 10 [ar1+gr1]= vreg0;** **rep 10 [ar1=gr1]= vreg0;** `fpu 2 rep 10 [ar1++]= vreg0;`	4.2F
Запись, количество читаемых слов берётся из vlen.	**rep vlen [ar1++]= vreg0;** `fpu 0 vlen [ar1++]= vreg0;`	4.2F

5.3.6 Пересылки между процессорными ячейками

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП
Пересылка значения вектоного регистра в регистр другой процессорной ячейки	**fpu 1 vreg2= fpu 3 vreg4;**	всегда 1	**5.1F**
Пересылка в векторный регистр вектора, скомпонованного по маске из значений двух векторных регистров.	**fpu 1 vreg2= fpu 3 mask ? vreg7 : vreg3;**		**5.2F**

5.3.7 Работа с блоком переупаковки

Переупаковщик работает с восемью различными типами векторов, с его помощью можно преобразовать любой из них к любому другому. Имена типов:

- **.float**
- **.double**
- **.float .in_high**
- **.float .in_low**
- **.fixed_32**
- **.fixed_64**
- **.fixed_32 .in_high**
- **.fixed_32 .in_low**

Цикл использования переупаковки состоит из двух команд, первая задаёт загрузку данных в узел и типы для преобразования; Вторая команда выгружает преобразованные данные.

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП
Переупаковка. Загрузка вектора из памяти в переупаковщик и запуск переупаковки	**.packer= [ar0++] with .float <= .fixed_32;** `fpu rep 32 .packer= [ar0++] with .float <= .fixed_32;`	всегда 1	**4.3F**
Переупаковка. Загрузка вектора из векторного регистра и запуск переупаковки	**fpu 1 .packer= vreg4 with .double <= .fixed_32 .in_high;**		**5.3F**
Загрузка из регистровой пары	**fpu 1 .packer= (vreg6,vreg7) with .float <= .fixed_64;**		5.3F
Выгрузка переупакованных данных в память	**fpu rep 32 [ar0++]= .packer;**		**4.4F**
Выгрузка переупакованных данных в векторный регистр	**fpu 1 rep 32 vreg4= .packer;**		**5.4F**
Использование регистра vlen для задания размера массива данных, запись в регистровую пару	**fpu 1 rep vlen (vreg4,vreg3)= .packer;**		5.4F

5.4 Инструкции векторного сопроцессора чисел с фиксированной точкой

5.4.1 Чтение и запись данных в векторных инструкциях

Команды доступа к памяти при чтении/записи данных могут располагаться только в левой части векторной инструкции.

При обращении к памяти считывается/записывается число длинных слов, равное количеству повторений, заданному в коде векторной инструкции (`rep` *`кол-во повторений`*).

Т.к. векторные инструкции обрабатывают только 64-х разрядные данные, все адреса, использующиеся в командах адресации, должны быть четными.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой правой частью.

Чтение из памяти

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Чтение по адресу, хранящемуся в адресном регистре. Данные загружаются в ВП для обработки в рабочей матрице или в векторном АЛУ. Адрес, с которого происходит считывание, не изменяется, и все данные считываются по одному и тому же адресу. Количество загружаемых длинных слов соответствует счетчику повторений.	**data = [ar0];** `rep 4 data = [ar0] with not data;`	1	5.1
Чтение по адресу, хранящемуся в регистре общего назначения. Данные загружаются в ВП для обработки в рабочей матрице или в векторном АЛУ. Адрес, с которого происходит считывание, не изменяется, и все данные считываются по одному и тому же адресу. Количество загружаемых длинных слов соответствует счетчику повторений. (*)	**ram, data = [gr2];** `rep 12 ram, data = [gr2] with data + 0;`	1	5.1
Чтение по адресу, хранящемуся в регистре общего назначения с модификацией адресного регистра. Адрес, с которого происходит считывание, не изменяется, и все данные считываются по одному и тому же адресу. Количество загружаемых длинных слов соответствует счетчику повторений. (*)	**ram = [ar4=gr4];** `rep 1 ram = [ar4=gr4] with data + 1;`	1	5.1
Чтение по адресу, хранящемуся в адресном регистре с пост инкрементацией на 2. Количество загружаемых длинных слов соответствует счетчику повторений.	**data = [ar0++];** `rep 32 data = [ar0++] with data;`	1	5.1
Чтение по адресу, хранящемуся в адресном регистре с пре декрементацией на 2. Количество загружаемых длинных слов соответствует счетчику повторений.	**data = [--ar4];** `rep 16 data = [--ar4] with data or ram;`	1	5.1

Загрузка в wfifo данных, хранящихся по адресу, хранящемуся в адресном регистре с пост инкрементацией адреса на значение регистра общего назначения. Количество загружаемых длинных слов соответствует счетчику повторений.	**wfifo = [ar0++gr0];** rep 8 wfifo = [ar0++gr0], ftw;	1	5.1
Загрузка в ram данных, хранящихся по адресу с пре инкрементацией на значение регистра общего назначения. Количество загружаемых длинных слов соответствует счетчику повторений. (*)	**ram = [ar0+=gr0];** rep 12 ram = [ar0+=gr0] with afifo - 1;	1	5.1

Запись в память

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Сохранение данных из afifo в память по адресу, хранящемуся в адресном регистре. Адрес, по которому происходит запись, не изменяется, и все данные записываются по одному и тому же адресу.	**[ar0] = afifo;** rep 4 [ar0] = afifo with not afifo;	1	5.1
Сохранение данных из afifo в память по адресу, хранящемуся в регистре общего назначения. Адрес, по которому происходит запись, не изменяется, и все данные записываются по одному и тому же адресу.	**[gr2] = afifo;** rep 2 [gr2],ram = afifo with afifo + 0;	1	5.1
Сохранение данных из afifo в память по адресу, хранящемуся в регистре общего назначения с модификацией адресного регистра. Адрес, по которому происходит запись, не изменяется, и все данные записываются по одному и тому же адресу.	**[ar4=gr4] = afifo;** rep 1 [ar4=gr4] = afifo with ram + 1;	1	5.1
Сохранение данных из afifo в память по адресу, хранящемуся в адресном регистре с пост инкрементацией на 2.	**[ar0++] = afifo;** rep 4 [ar0++] = afifo with vsum ,ram,0;	1	5.1
Сохранение данных из afifo в память по адресу, хранящемуся в адресном регистре с пре декрементацией на 2.	**[--ar4] = afifo;** rep 16 [--ar4] = afifo with ram - 1;	1	5.1
Сохранение данных из afifo в память по адресу, хранящемуся в адресном регистре с пост инкрементацией адреса на значение	**[ar0++gr0] = afifo;** rep 8 [ar0++gr0] = afifo with not ram;	1	5.1

регистра общего назначения.			
Сохранение данных из afifo в память по адресу с пре инкрементацией на значение регистра общего назначения.	**[ar0+=gr0] = afifo;** rep 12 [ar0+=gr0] = afifo with not ram;	1	5.1
Запись в память с одновременным копированием содержимого afifo в ram.(*)	**[ar0++],ram = afifo;** rep 5 [ar0++],ram = afifo with 0-1;	1	5.1

Примечание

В правой части операции, помеченной знаком (*), не может использоваться буфер ram*, поскольку он может либо только принимать данные, либо только передавать их.*

5.4.2 Пустая команда и отсутствие адресных операций

Система команд процессора предусматривает возможность отсутствия адресной команды левой части в векторной инструкции. Более того, существует пустая векторная команда, которая может быть использована в программе. Основное свойство пустой векторной команды состоит в том, что ей соответствует нулевой машинный код.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Пустая векторная команда	**vnul;**	1	5.3
Отсутствие адресной операции в левой части векторной инструкции.	**rep 32 with not ram;**	1	5.3

5.4.3 Логические операции над операндами X и Y

Логические операции над операндами **X** и **Y** могут располагаться только в правой части векторной инструкции. В действительности в качестве операндов **X** и **Y** используются внутренние регистры-контейнеры векторного процессора: ram, data и afifo, и нулевой вектор.

Логические операции над векторами выполняются векторным АЛУ. Заметим, что результат логических операций не зависит от разбиения данных.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Если векторная инструкция не содержит команды доступа в память, то левая часть инструкции может быть опущена (см. подраздел 4.2).

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Побитовая операция OR, результат сохраняется в afifo. (*)	**with X or Y;** rep 4 data = [ar0+=gr0] with data or ram;	7.1
Побитовая операция AND, результат сохраняется в afifo.	**with X and Y;** rep 32 wfifo = [ar2++] with ram and afifo;	7.1
Побитовая операция XOR, результат сохраняется в afifo.	**with X xor Y;** rep 10 [ar0++] = afifo with afifo xor ram;	7.1
Операция отрицания, результат сохраняется в afifo.	**with not X;** rep 16 data = [ar4++],ftw with not data;	7.1
Комбинация операции отрицания первого операнда с побитовой операцией OR двух операндов, результат сохраняется в afifo.	**with not X or Y;** rep 8 [ar2++] = afifo with not afifo or ram;	7.1
Комбинация операции отрицания второго операнда с побитовой операцией OR двух операндов, результат сохраняется в afifo.	**with X or not Y;** rep 1 [gr4] = afifo with ram or not afifo;	7.1
Комбинация операции отрицания обоих операндов с побитовой операцией OR, результат сохраняется в afifo.	**with not X or not Y;** rep 3 data = [ar6+=gr6] with not data or not ram;	7.1
Комбинация операции отрицания первого операнда с побитовой операцией AND двух операндов, результат сохраняется в afifo.	**with not X and Y;** rep 2 with not afifo and ram;	7.1
Комбинация операции отрицания второго операнда с побитовой операцией AND двух операндов, результат сохраняется в afifo.	**with X and not Y;** rep 9 [ar2++] = afifo with afifo and not ram;	7.1
Комбинация операции отрицания обоих операндов с побитовой операцией AND, результат сохраняется в afifo.	**with not X and not Y;** rep 24 ftw with ram and not afifo;	7.1
Комбинация операции отрицания первого операнда с побитовой операцией XOR двух операндов, результат сохраняется в afifo.	**with not X xor Y;** rep 21 data = [ar0++] with not data xor ram;	7.1

Нулевая логическая операция (заполняет afifo длинными словами: 0000000000000000hl).	**with vfalse;** rep 16 [ar3++] = afifo with vfalse;	7.1
Логическая операция, заполняющая afifo длинными словами: 0FFFFFFFFFFFFFFFhl.	**with vtrue;** rep 32 wfifo = [ar5++gr5] with vtrue;	7.1

Примечание

Если в инструкции, помеченной знаком (*), участвует нулевой вектор как операнд, то он может быть опущен, например:

```
rep 32 data = [ar0++] with data;
```

В данном случае данные, загружаемые из памяти, попадут в afifo без изменений.

5.4.4 Арифметические операции над операндами X и Y

Арифметические операции над операндами **X** и **Y** могут располагаться только в правой части векторной инструкции. В действительности в качестве операндов **X** и **Y** используются внутренние регистры-контейнеры векторного процессора: ram, data и afifo.

Арифметические операции над векторами выполняются векторным АЛУ, поэтому перед выполнением АЛУ арифметической операции необходимо определить значение векторного регистра nb1 (см. пункт 3.4.2).

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Если векторная инструкция не содержит команды доступа в память, то левая часть инструкции может быть опущена (см. подраздел 4.2).

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Сумма двух векторов, результат сохраняется в afifo..	**with X + Y;** rep 4 data = [ar0+=gr0] with data + ram;	7.2
Разность двух векторов, результат сохраняется в afifo..	**with X – Y;** rep 32 wfifo = [ar2++] with ram - afifo;	7.2
Изменение знака элементов вектора Y, результат сохраняется в afifo..	**with 0 – Y;** rep 16 wfifo = [ar2++] with 0 - ram;	7.2
Сумма вектора с единичным вектором, результат сохраняется в afifo..	**with X + 1;** rep 8 ftw with afifo + 1;	7.2

Инициализация элементов вектора единичными значениями, результат сохраняется в `afifo`.	**`with 0 + 1;`** `rep 4 [ar4+=gr4] = afifo with 0 + 1;`	7.2
Вычитание единичного вектора из вектора данных, подаваемого на вход векторного АЛУ, результат сохраняется в `afifo`.	**`with X – 1;`** `rep 8 ram = [--ar2] with data - 1;`	7.2
Инициализация элементов вектора значением –1 (все биты равны 1) , результат сохраняется в `afifo`.	**`with 0 – 1;`** `rep 8 with 0 - 1;`	7.2

5.4.5 Операция маскирования на векторном АЛУ

Операция маскирования на векторном АЛУ располагается в правой части векторной инструкции. В действительности в качестве операндов **X** и **Y** используются внутренние регистры-контейнеры векторного процессора: `ram`, `data` и `afifo`.

Операции маскирования выполняются блоком маскирования, перед его работой необходимо определить значения векторных регистров `nb1` и `sb` (см. пункт 3.4.2 и пункт 3.4.3).

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Если векторная инструкция не содержит команды доступа в память, то левая часть инструкции может быть опущена (см. подраздел 4.2).

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Маскирование векторов, подаваемых на входы **X** и **Y** векторного АЛУ, маской **M**, результат операции: (**X** AND **M**) OR (**Y** AND ~**M**) сохраняется в `afifo`.	**`with mask M, X, Y;`** `rep 4 data = [ar0+=gr0] with mask afifo, data, ram;`	7.3
Маскирование векторов, подаваемых на входы X и Y векторного АЛУ, маской M (**Y** AND ~**M**), (**X** AND **M**) и плюс циклический сдвиг вправо на 1 бит операнда X до выполнения логического ИЛИ двух операндов, результат сохраняется в `afifo`.	**`with mask M, shift X, Y;`** `rep 8 data = [--ar0] with mask afifo, shift data, ram;`	7.3

Описание операции маскирования приведено в пункте 1.5.4 «Операция маскирования». Операция циклического сдвига описана в пункте 1.5.6 «Циклический сдвиг вправо операнда **X** при взвешенном суммировании».

5.4.6 Операция взвешенного суммирования

Операция взвешенного суммирования располагается в правой части векторной инструкции. В действительности в качестве операндов **X** и **Y** используются внутренние регистры-контейнеры векторного процессора: `ram`, `data` и `afifo`, кроме того, в качестве операнда **Y** может использоваться регистр `vr`.

Операции взвешенного суммирования выполняются на рабочей матрице, перед выполнением операции необходимо определить значения векторных регистров `nb1` и `sb` (см. пункт 3.4.2 и пункт 3.4.3).

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Если векторная инструкция не содержит команды доступа в память, то левая часть инструкции может быть опущена (см. подраздел 4.2).

Если в операции взвешенного суммирования маска не используется, то первый операнд (маска) может быть опущен, но запятая перед операндом **X** сохраняется

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Простая операция взвешенного суммирования, результат сохраняется в `afifo`.	**`with vsum , X, 0;`** `rep 12 data = [ar2++gr2] with vsum , data, 0;`	7.4
Операция взвешенного суммирования с добавлением вектора частичных сумм, результат сохраняется в `afifo`.	**`with vsum , X, Y;`** `rep 32 ram = [--ar2] with vsum , data, vr;`	7.4
Операция взвешенного суммирования, совмещенная с предварительным маскированием аргументов ((**M** AND **X**) OR (**Y** AND ~**M**)), результат сохраняется в `afifo`.	**`with vsum M, X, Y;`** `rep 8 data = [ar2++] with vsum ram, data, afifo;`	7.4
Операция взвешенного суммирования с циклическим сдвигом вправо на 1 бит операнда **X**, результат сохраняется в `afifo`.	**`with vsum , shift X, Y;`** `rep 1 [ar2=gr2] with vsum , shift ram, ram;`	7.4

5.4.7 Операции активации

Операции активации выполняются над операндами в правой части векторной инструкции путем добавления к имени операнда ключевого слова `activate`. Операция активации выполняется над каждым операндом независимо.

В зависимости от того, совместно с каким типом команды выполняется активация, она может быть логической (пороговая функция) или арифметической (функция насыщения).

Операции активации выполняются в двух блоках активации, перед выполнением операции необходимо определить функции активации для каждого операнда, которые определяются значениями

векторных регистров `f1cr` (для операнда **X**) и `f2cr` (для операнда **Y**) (см. пункт 3.4.1).

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Для процессора nm6405 есть ограничение в использовании активации. Не допускается активировать операнд X и одновременно использовать сдвиг, к примеру команда, содержащая **`shift activate data`** более не допускается, ассемблер в таком случае сообщит об ошибке.

Логическая активация

Логическая активация значений векторов может проводиться при любой логической операции из приведенных в 5.2.3. Логические операции над операндами X и Y. Ниже в таблице приводятся примеры активации операндов, стоящих на различных позициях в правой части векторной инструкции.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Пример логической операции с активацией операнда **X**, активация операнда **X** с последующим выполнением логического ИЛИ с **Y**.	**`with activate X or Y;`** `rep 32 data = [ar0++] with activate data or ram;`	7.1
Пример логической операции с активацией операнда **X**, когда над **X** дополнительно совершается операция отрицания после активации и перед выполнением логического И.	**`with not activate X and Y;`** `rep 16 data = [--ar0] with not activate data and ram;`	7.1
Пример логической операции с активацией операнда **Y**, активация операнда **Y** с последующим выполнением логического ИЛИ с **X**.	**`with X or activate Y;`** `rep 8 data = [ar0++gr0] with data or activate ram;`	7.1
Пример логической операции с активацией операнда **Y**, когда над **Y** дополнительно совершается операция отрицания после активации и перед выполнением логического И.	**`with X and not activate Y;`** `rep 12 [--ar0] = afifo with afifo and not activate ram;`	7.1
Пример логической операции с активацией обоих операндов перед её выполнением.	**`with activate X xor activate Y;`** `rep 8 ftw with activate afifo xor activate ram;`	7.1
Пример логической операции с активацией обоих операндов, когда над ними дополнительно совершается операция отрицания, первым выполняется активация операндов, затем отрицание операндов, затем и сама логическая операция (в данном случае **AND**).	**`with not activate X and not activate Y;`** `rep 2 with not activate afifo and not activate ram;`	7.1
Маскирование векторов перед активацией операнда **X**, с	**`with mask M, activate X, Y;`**	7.1

последующим выполнением логического ИЛИ операндов.	`rep 8 data = [ar0++] with mask afifo, activate data, ram;`	
Маскирование векторов перед логической активацией операнда **Y**, с последующим выполнением логического ИЛИ операндов.	**`with mask M, X, activate Y;`** `rep 1 data = [--ar0] with mask afifo, data, activate ram;`	7.1
Маскирование векторов перед логической активацией обоих операндов, с последующим выполнением логического ИЛИ операндов.	**`with mask M, activate X, activate Y;`** `rep 4 with mask afifo, activate ram, activate ram;`	7.1
Маскирование векторов перед логической активацией операнда X, после чего выполняется логическое ИЛИ операндов.	**`with mask M, shift X, Y;`** `rep 12 with mask afifo, shift data, ram;`	7.1

Арифметическая активация

Ниже в таблице приводятся примеры активации операндов, стоящих на различных позициях в правой части векторной инструкции.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Пример арифметической операции с активацией операнда **X** до выполнения операции, к операнду **X** до выполнения сложения с **Y** применяется функция насыщения.	**`with activate X + Y;`** `rep 32 data = [ar0++] with activate data + ram;`	7.2
Пример арифметической операции с активацией операнда **Y** до выполнения операции, к операнду **Y** до выполнения вычитания из **X** применяется функция насыщения..	**`with X - activate Y;`** `rep 8 data = [ar0++gr0] with data - activate ram;`	7.2
Пример арифметической операции с применением функции насыщения к обоим операндам до выполнения операции.	**`with activate X + activate Y;`** `rep 16 data = [--ar0] with activate data + activate ram;`	7.2
Пример операции взвешенного суммирования с применением функции насыщения к операнду **X** перед выполнением взвешенного суммирования.	**`with vsum , activate X, Y;`** `rep 2 data = [ar4++] with vsum , activate data, ram;`	7.2
Пример последовательного выполнения маскирования обоих операндов, затем применением функции насыщения к операнду **Y** и выполнения взвешенного суммирования.	**`with vsum , X, activate Y;`** `rep 2 [ar6++] = afifo with vsum , ram, activate afifo;`	7.2

Пример последовательного выполнения операции маскирования операндов, затем применением функции насыщения и выполнения взвешенного суммирования.	**with vsum M, activate X, activate Y;** rep 5 with vsum ram, activate ram, activate afifo;	7.2
Пример операции взвешенного суммирования операнда **Y** с результатом циклического сдвига на 1 бит вправо операнда **X**.	**with vsum , shift X, 0;** rep 30 ftw with vsum , shift ram, 0;	7.2

5.4.8 Загрузка весов в матричный узел

В данном разделе собраны все векторные команды, необходимые для загрузки весовых коэффициентов в теневую и рабочую матрицы ВП.

В векторной инструкции все операции загрузки весов располагаются слева от ключевого слова `with`.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой правой частью.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Загрузка весовых коэффициентов из памяти в буфер `wfifo`.	**rep 24 wfifo = [ar0++];** rep 24 wfifo = [ar0++] with ram + 1;	1	5.2
Загрузка весовых коэффициентов из памяти в буфер `wfifo` с одновременной передачей их в теневую матрицу.	**Rep 32 wfifo = [ar1++], ftw;** Rep 32 wfifo = [ar1++],ftw with 0 - 1;	1	5.2
Загрузка весовых коэффициентов из памяти в буфер `wfifo` с одновременной передачей их в теневую, а затем и в рабочую матрицу.	**rep 32 wfifo = [ar1++], ftw, wtw;** rep 32 wfifo = [ar1++],ftw, wtw with not ram;	1	5.2
Передача весовых коэффициентов из `wfifo` в теневую матрицу.(**)	**ftw;** rep 16 ftw with not ram and afifo;	1	5.3
Копирование весовых коэффициентов из теневой матрицы в рабочую.	**wtw;** rep 12 wtw with activate afifo;	1	5.3

Примечание

Операция `ftw`*, помеченная знаком (**), может использоваться не только изолированно, но и быть записана в левой части любой векторной команды, например,*

rep 32 data = [ar0++], ftw with vsum , data, 0;.

5.4.9 Сохранение в памяти значений векторных регистров

Для сохранения в памяти значений регистров `f2cr`, `f1cr`, `nb2`, `sb`, `vr` используется специальная команда. Перечисленные регистры напрямую недоступны по чтению, однако, их значения можно получить косвенным путем.

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	ТИП
Сохранение векторных регистров в буфер `afifo`.	**`rep 5 with store vregs;`**	7.5

Все сохраняемые регистры являются 64-х разрядными. Счетчик повторений в векторной инструкции должен равняться 5, по количеству сохраняемых регистров. Приведенная выше команда сохраняет регистры в `afifo`, откуда они следующей инструкцией могут быть сохранены в памяти см. пункт 5.2.1.

5.5 Дополнительные операции nmc4

В этом разделе перечисляются дополнительные команды, отсутствующие в более ранних версиях архитектуры.

5.5.1 Команда останова nmc4

Архитектура и система команд векторного сопроцессора фиксированной точки nmc4 в целом повторяет nmc3, но не полностью. Некоторые инструкции nmc3 не будут работать на nmc4 (см. 3.2.1.1), кроме того изменился способ разрешения параллельного выполнения инструкций.

Специальная команда `halt` останавливает выполнение потока команд. Подобно командам перехода эта команда имеет слоты отложенных инструкций, которые перед остановом успеют выполниться. Как и для команд перехода, эти слоты по умолчанию заполняются нулями; чтобы поместить туда произвольные инструкции, используйте ключевое слово `delayed`.

При останове энергопотребление процессора уменьшается.

Для возобновления выполнения процессор должен получить прерывание; после возврата из обработчика прерывания, начнут выполняться инструкции, следующие за отложенными командами `halt`.

Команда впервые появилась в версии архитектуры nmc4

ОПИСАНИЕ	СИНТАКСИС	РАЗМЕР	ТИП
Останов.	**`halt;`** `halt with gr0++;`	1	2.1(?)

5.5.2 Команды переключения режимов nmc4

В nmc4 имеется ряд режимов, переключение которых осуществляется записью в специальные регистры, причём записываемое значение игнорируется, важен сам факт записи. Во избежание недоразумений, для таких переключающих команд созданы специальные мнемоники.

Из этих режимов часть относятся к ядру, а часть только к сопроцессору плавающей точки

ОПИСАНИЕ	СИНТАКСИС КОМАНДЫ	РАЗМЕР	ТИП
Установка и отмена режима параллельного выполнения на уровне ядра. (*)	**set .branch;** **set .wait;** set .branch with gr1++;	1	2.1
Команды переключения режимов сопроцессора. См. 1.4.2.2.1	**set fp_branch;** set fp_branch with gr1--;	1	1.1F

Примечание *Отключаемый режим параллельного выполнения имеется у всех устройств nmc, но раньше он устанавливался статически для каждой инструкции, (см. 2.7.2) а начиная с nmc4 включается и выключается динамически описываемой здесь командой*

НАУЧНО-ТЕХНИЧЕСКИЙ ЦЕНТР

Научно-технический центр Модуль
АЯ 166, Москва, 125190, Россия
Тел: +7 (495) 152-9335
Факс: +7 (495) 152-4661
E-Mail: info@module.ru
WWW: http://www.module.ru

Напечатано в России Дата издания: 20.07.2023