Базовое программное обеспечение
процессора NM6403

NeuroMatrix® NM6403
Описание языка ассемблера
Предварительная версия

АКЦИОНЕРНОЕ ОБЩЕСТВО
НАУЧНО-ТЕХНИЧЕСКИЙ ЦЕНТР

Оглавление

ПРЕДИСЛОВИЕ 1
О СПРАВОЧНОМ РУКОВОДСТВЕ 1
КАК ОРГАНИЗОВАН ДАННЫЙ ДОКУМЕНТ 1
СОГЛАШЕНИЯ О НОТАЦИЯХ 2
КАК ОФОРМЛЯТЬ ЗАМЕЧАНИЯ И ПРЕДЛОЖЕНИЯ 3
По документации 3
По работе компонент БПО 3
1 КРАТКИЙ ОБЗОР СТРУКТУРЫ ПРОЦЕССОРА 1-1
1.1 ВВЕДЕНИЕ 1-3
1.2 ВНЕШНИЙ ИНТЕРФЕЙС ПРОЦЕССОРА 1-3
1.3 ОБЩЕЕ ОПИСАНИЕ ВНУТРЕННЕЙ СТРУКТУРЫ ПРОЦЕССОРА 1-4
1.3.1 Описание основных элементов скалярного процессора 1-5
1.3.2 Описание основных элементов векторного процессора 1-6
1.4 ОСНОВНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ БЛОКИ ВЕКТОРНОГО ПРОЦЕССОРА 1-8
1.4.1 Взвешенное суммирование 1-9
1.4.2 Выполнение операций на векторном АЛУ 1-12
1.4.3 Операция маскирования 1-14
1.4.4 Обработка данных функцией активации 1-15
1.4.5 Циклический сдвиг вправо операнда **X** при взвешенном суммировании 1-17
1.4.6 Порядок выполнения преобразований над данными на ВП 1-17
2 ОБЗОР ОСНОВНЫХ ЭЛЕМЕНТОВ ЯЗЫКА АССЕМБЛЕРА 2-1
2.1 СЛУЖЕБНЫЕ СЛОВА 2-3
2.2 СТРУКТУРА АССЕМБЛЕРНОГО ФАЙЛА 2-5
2.3 СЕКЦИИ 2-5
2.3.1 Секции кода 2-6
2.3.2 Секции инициализированных данных 2-7
2.3.3 Секции неинициализированных данных 2-8
2.3.4 Пространство между секциями 2-8
2.4 КОНСТАНТЫ 2-9
2.4.1 Форматы представления констант 2-9
2.4.1.1 Двоичные целые константы 2-9
2.4.1.2 Восьмеричные целые константы 2-10
2.4.1.3 Десятичные целые константы 2-10
2.4.1.4 Шестнадцатеричные целые константы 2-10
2.4.1.5 Константы с плавающей точкой 2-10
2.4.1.6 Строковые константы 2-11
2.4.2 Константные выражения 2-11
2.4.2.1 Числовые константные выражения 2-12
2.4.2.2 Адресные константные выражения 2-12

2.4.3 Определение и использование константы 2-13
2.5 Метки 2-14
2.5.1 Объявление метки 2-14
2.5.2 Определение метки 2-14
2.5.3 Ссылки на метку 2-15
2.5.4 Типы связывания и область действия меток 2-15
2.6 Переменные 2-18
2.6.1 Получение адреса переменной 2-19
2.6.2 Получение значения переменной 2-19
2.6.3 Простые переменные 2-19
2.6.4 Составные переменные 2-20
2.6.4.1 Массивы 2-20
2.6.4.2 Структуры 2-20
2.6.5 Начальные значения 2-22
2.6.6 Область действия данных 2-23
2.6.7 Места для объявлений и начальной инициализации переменных 2-24
2.7 Директивы языка ассемблера 2-25
2.7.1 Директива .align 2-27
2.7.2 Директива .branch 2-28
2.7.3 Директива .endif 2-28
2.7.4 Директива .endrepeat 2-29
2.7.5 Директива .if 2-29
2.7.6 Директива .repeat 2-30
2.7.7 Директива .wait 2-30
2.7.8 Директивы отладочной информации 2-30
2.7.8.1 Директива .debug_arange 2-30
2.7.8.2 Директивы .debug_die и .debug_die_child 2-31
2.7.8.3 Директива .debug_die_endchild 2-33
2.7.8.4 Директивы .debug_start_sequence и .debug_end_sequence 2-33
2.7.8.5 Директива .debug_frame_cie 2-34
2.7.8.6 Директива .debug_frame_fde 2-34
2.7.8.7 Директива .debug_line 2-34
2.7.8.8 Директива .debug_pubname 2-35
2.7.8.9 Директива .debug_root_die 2-35
2.7.8.10 Директива .debug_source_directory 2-35
2.7.8.11 Директива .debug_source_file 2-35
2.8 Псевдофункции 2-36
2.8.1 Функция loword 2-37
2.8.2 Функция hiword 2-37
2.8.3 Функция sizeof 2-37
2.8.4 Функция offset 2-38
2.8.5 Функция float 2-39
2.8.6 Функция double 2-39
3 РЕГИСТРЫ 3-1

3.1 Основные регистры 3-3
3.1.1 Адресные регистры 3-3
3.1.2 Регистры общего назначения 3-4
3.1.3 Регистровые пары 3-4
3.2 Специальные регистры 3-5
3.2.1 Регистр gmicr 3-6
3.2.2 Регистры управления коммуникационными портами: (ica, icc), (oca, occ) 3-13
3.2.3 Регистр intr 3-19
3.2.4 Регистр lmicr 3-24
3.2.5 Регистр pc 3-25
3.2.6 Регистр pswr 3-25
3.2.7 Регистры таймеров t0, t1 3-33
3.3 Векторные регистры 3-34
3.3.1 Регистры f1cr и f2cr 3-36
3.3.2 Регистр nb1(nb2) 3-41
3.3.3 Регистр sb(sb1 и sb2) 3-45
3.3.4 Регистр vr 3-49
3.3.5 Регистр-контейнер afifo 3-50
3.3.6 Логический регистр-контейнер data 3-54
3.3.7 Регистр-контейнер ram 3-56
3.3.8 Регистр-контейнер wfifo 3-58
4 ФОРМАТ АССЕМБЛЕРНЫХ ИНСТРУКЦИЙ 4-1
4.1 Типы скалярных команд 4-4
4.2 Типы векторных команд 4-6
4.3 Машинные коды команд процессора 4-6
5 НАБОР ИНСТРУКЦИЙ ЯЗЫКА АССЕМБЛЕРА 5-1
5.1 Скалярные инструкции NM6403 5-3
5.1.1 Пустая команда 5-3
5.1.2 Команды чтения из памяти 5-4
5.1.3 Команды записи в память 5-6
5.1.4 Команды работы со стеком 5-9
5.1.5 Команды копирования регистров 5-10
5.1.6 Команды инициализации регистров константами 5-12
5.1.7 Команды модификации адресных регистров 5-14
5.1.8 Команды модификации регистра pswr 5-15
5.1.9 Команды перехода 5-15
5.1.9.1 Команды безусловного перехода 5-16
5.1.9.2 Команды перехода к подпрограмме 5-17
5.1.9.3 Команды возврата из подпрограммы/прерывания 5-18
5.1.9.4 Набор условий перехода 5-19
5.1.10 Основные скалярные операции процессора 5-20
5.1.11 Арифметические операции 5-21
5.1.12 Логические операции 5-22

5.1.13 Операции установки флагов без изменения значений регистров.....5-24
5.1.14 Операции сдвига 5-26
5.2 Векторные инструкции NM6403 5-27
5.2.1 Методы адресации при чтении/записи данных в ВП 5-27
5.2.2 Пустая команда и отсутствие адресных операций.....5-29
5.2.3 Логические операции над операндами X и Y 5-29
5.2.4 Арифметические операции над операндами X и Y.....5-30
5.2.5 Операция маскирования на векторном АЛУ.....5-31
5.2.6 Операция взвешенного суммирования.....5-31
5.2.7 Операции активации 5-32
5.2.8 Загрузка весов в матричный узел 5-34
5.2.9 Сохранение в памяти значений векторных регистров 5-34

Предисловие

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

Примечание

В данном справочном руководстве содержится информация об устройстве процессора NeuroMatrix® NM6403, достаточная для того, чтобы разрабатывать программы на языке ассемблера.

О справочном руководстве

Данное руководство содержит следующую информацию:

- структура и функционирование основных узлов процессора с точки зрения программиста;
- описание всех регистров;
- 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 каждой команды языка ассемблера с примером ее использования.

Как организован данный документ

Данный документ разбит на шесть глав, каждая из которых освещает свою область языка ассемблера. Разбиение на части осуществляется следующим образом:

Глава 1 **Обзор структуры процессора**

Содержит справочную информацию о структуре процессора с точки зрения программиста. Включает описание функционирования основных вычислительных узлов.

Глава 2 **Обзор основных элементов языка ассемблера**

Содержит полную информацию о структуре и правилах построения программ на языке

ассемблера, приводит примеры использования различных синтаксических конструкций.

Глава 3 **Описание регистров процессора**

Содержит подробную информацию о составе и различных типах регистров, описывает регистровые поля регистров управления, правила использования и назначение векторных регистров, приводит примеры их использования в различных ассемблерных инструкциях.

Глава 4 **Формат ассемблерных инструкций**

Содержит информацию о формате скалярных и векторных инструкций ассемблера, приводит состав и структуру машинных команд.

Глава 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.

Глава 6 **Подробное описание инструкций**

Содержит подробное описание каждой команды или операции, выполняемой процессором, приводит пример её использования, дает рекомендации по применению.

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 языках Си++ и ассемблера.

`Courier` отмечает текст, который должен быть заменен пользовательской информацией, например, реальным значением константы.

Текст Так помечается текст, на который необходимо

или Текст	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 базовое ПО, у Вас возникли сложности, например: Вы считаете, что материал изложен недостаточно подробно для понимания, пожалуйста, присылайте свои замечания в следующем виде:

- название документа;
- персональный номер документа и номер версии документации;
- номера страниц, на которые приводится ссылка;
- краткое описание замечания.

По работе компонент БПО

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

- номер версии базового ПО процессора NM6403, которое Вы используете;
- небольшой фрагмент исходного кода, на котором проявляется ошибка;
- краткое описание ее проявления и возможные предположения о причинах ее возникновения, если таковые имеются.

1 Краткий обзор структуры процессора

1.1 Введение 1-3
1.2 Внешний интерфейс процессора 1-3
1.3 Общее описание внутренней структуры процессора 1-4
1.3.1 Описание основных элементов скалярного процессора 1-5
1.3.2 Описание основных элементов векторного процессора 1-6
1.4 Основные вычислительные блоки векторного процессора 1-8
1.4.1 Взвешенное суммирование 1-9
1.4.2 Выполнение операций на векторном АЛУ 1-12
1.4.3 Операция маскирования 1-14
1.4.4 Обработка данных функцией активации 1-15
1.4.5 Циклический сдвиг вправо операнда **X** при взвешенном суммировании 1-17
1.4.6 Порядок выполнения преобразований над данными на ВП 1-17

1.1 Введение

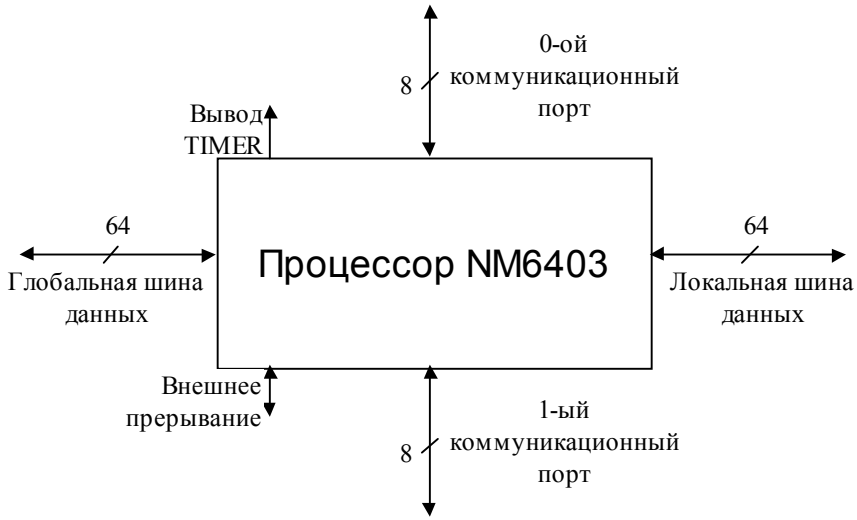
Прежде чем приступить к описанию языка ассемблера, необходимо ввести некоторые понятия, связанные со структурой процессора NM6403, на которые в дальнейшем можно будет ссылаться при описании тех или иных конструкций языка.

1.2 Внешний интерфейс процессора

Структура процессора, описанная в данном разделе, отражает взгляд программиста, поэтому некоторые понятия, которые не используются при программировании, опущены. Более детальный обзор процессора приведен в документе:
Процессор NeuroMatrix®NM6403. Руководство пользователя.

Процессор NM6403 имеет четыре канала, по которым он может обмениваться данными с внешними устройствами (см. Рис. 1-1).

Рис. 1-1 Схема каналов доступа к данным со стороны процессора NM6403.



Глобальная и локальная шина используются для доступа к внешней памяти. Память, которая доступна через глобальную шину, называется **глобальной памятью**. Память, доступная через локальную шину, называется **локальной**.

Помимо работы с внешней памятью процессор может принимать и передавать данные через коммуникационные порты. Коммуникационные порты связывают данный процессор с другими такими же, или с процессорами TMS320C4x, которые имеют аналогичный интерфейс обмена данными. Более подробное описание работы с коммуникационными портами дано в параграфе 3.2.2. Регистры управления коммуникационными портами (стр. 3-13).

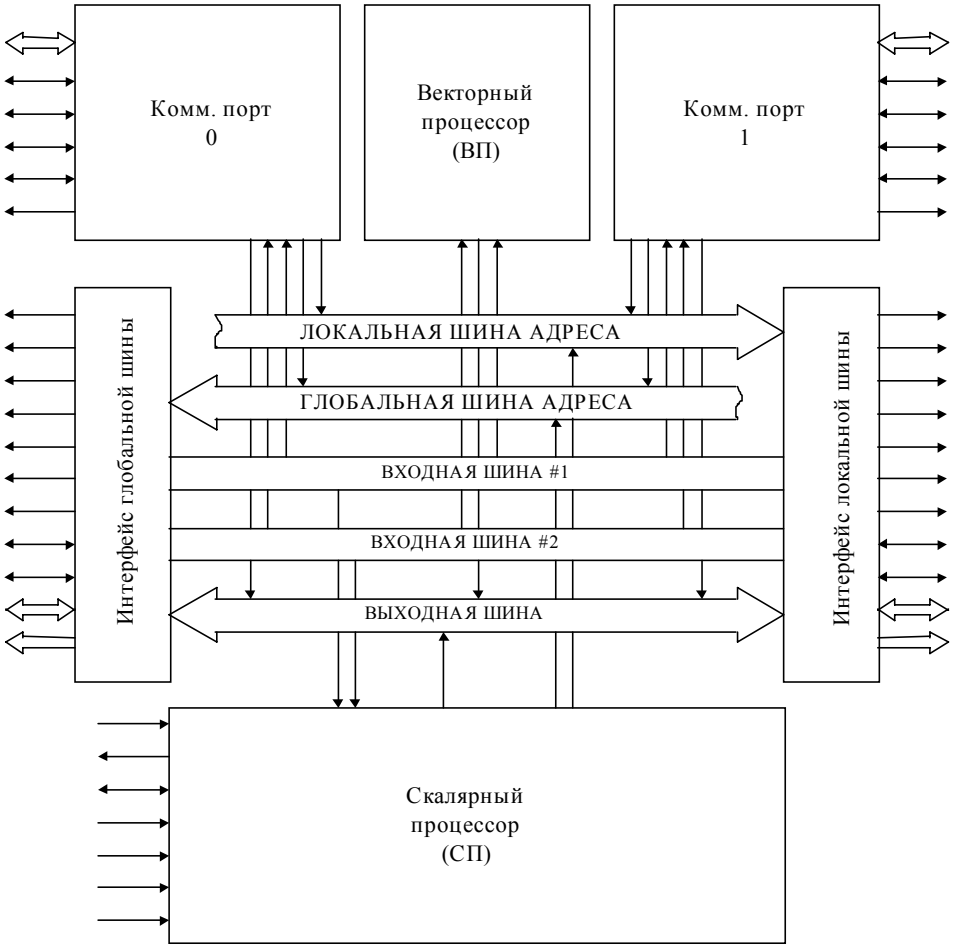
Внутренней памяти в обычном понимании в процессоре нет.

1.3 Общее описание внутренней структуры процессора

Процессор NM6403 имеет следующие внутренние блоки (Рис. 1-2):

- скалярный процессор (СП);
- векторный процессор (ВП);
- два DMA-сопроцессора, управляющие работой коммуникационных портов (см. 3.2.2);
- два таймера (см. 3.2.7);
- регистры управления интерфейсом доступа к внешней памяти (см. 3.2.1 и 3.2.4).

Рис. 1-2 Блочная структура процессора NeuroMatrix NM6403.



Процессор NM6403 имеет 64-х разрядный интерфейс работы с внешней памятью. За одно обращение к памяти он позволяет записать или прочитать одно 64-х разрядное число по каждой из шин.

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 процессора NM6403 можно найти в разделе на стр. 5-1.

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 разделе 3.1.Основные регистры на стр.3-3.

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;
- вызовы функций с записью с стек адреса возврата, в том числе и отложенные вызовы функций;
- пошаговое умножение;
- управление таймерами;
- настройка регистров управления доступом к внешней памяти на тип, используемый в конкретном устройстве;
- управление векторным процессором путем задания его конфигурации.

Полный набор скалярных команд процессора приведен в разделе 5.1.Скалярные инструкции NM6403 на стр. 5-3.

1.3.2 Описание основных элементов векторного процессора

ВП представляет собой специализированный матричный узел - операционное устройство (ОУ) для выполнения операций умножения с накоплением, арифметических и логических операций, маскирования, функций активации над векторами и матрицами.

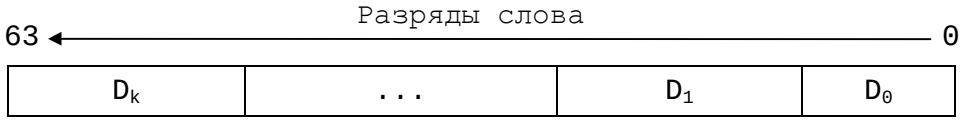
Под векторами понимаются одномерные массивы однородных данных, расположенные в памяти в виде непрерывного блока. Матрица - это массив векторов.

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. 1-3). В общем случае слово упакованных данных представляет собой вектор

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

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. 1-3 Формат слова упакованных векторных данных.



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 параграфе 1.4.1. Взвешенное суммирование на стр. 1-9;
- **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 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 параграфе 1.4.2. Выполнение операций на векторном АЛУ на стр. 1-12.

- **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 параграф 3.3.8. Регистр-контейнер (см. стр. 3-58);
- **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 параграф 3.3.7. Регистр-контейнер (см. стр. 3-56);
- **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 параграф 3.3.6. Логический регистр-контейнер (см. стр. 3-54);
- **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 параграф 3.3.5. Регистр-контейнер (см. стр. 3-50);

- **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 разделе 3.3.4. Регистр (см. стр. 3-49);
- 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 разделе 3.3.1. Регистры f1cr и (см. стр. 3-36);
- Устройства, обеспечивающие выполнение **операции маскирования** над входными векторами. Более подробное писание выполнения операции маскирования дано в разделе 1.4.3. Операция маскирования (см. стр. 1-14);
- 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 разделе 1.4.5. Циклический сдвиг вправо операнда X при взвешенном суммировании (см. стр. 1-17).

1.4 Основные вычислительные блоки векторного процессора

Далее приводится описание работы основных блоков ВП, определяющих возможности процессора NM6403 в выполнении следующих операций:

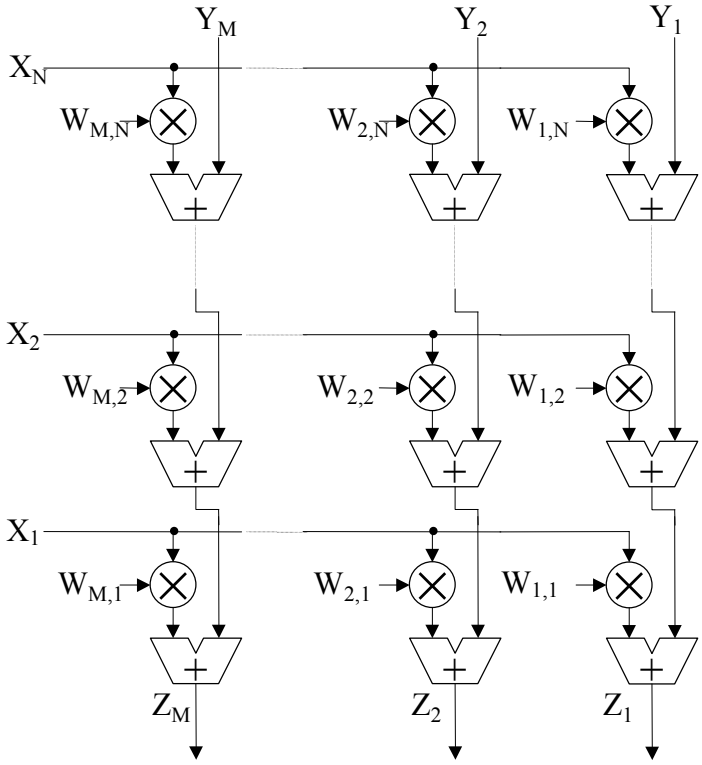
- умножение с накоплением, называемое также взвешенным суммированием;
- арифметические и логические операции на векторном АЛУ;
- маскирование данных;
- функции активации;
- сдвиг операнда **X** при выполнении взвешенного суммирования.

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

1.4.1 Взвешенное суммирование

Операция умножения с накоплением выполняется рабочей матрице, входящей в состав операционного узла ВП. Схематично она представлена на (см. Рис. 1-4).

Рис. 1-4 Схематичное представление операции взвешенного суммирования.



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

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

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 два входа **X** и **Y** (см. Рис. 1-5). На эти входы подаются данные, расположенные во внешней памяти, либо во

внутренних буферах `ram` (см. п. 3.3.7. на стр. 3-56) и `afifo` (см. п. 3.3.5. на стр. 3-50), работающих по принципу FIFO. Данные из буферов или из памяти могут быть поданы как на вход **X**, так и на **Y**. То есть, например, вектор 64-х разрядных слов, хранящийся в `ram`, может быть передан на обработку в операционный узел через вход **X** и/или **Y**. Для управления потоком данных из внешней памяти используется логический буфер `data` (см. п. 3.3.6. на стр. 3-54).

В качестве входа **Y** может также быть использован векторный регистр `vr`.(см. 3.3.4. на стр. 3-49).

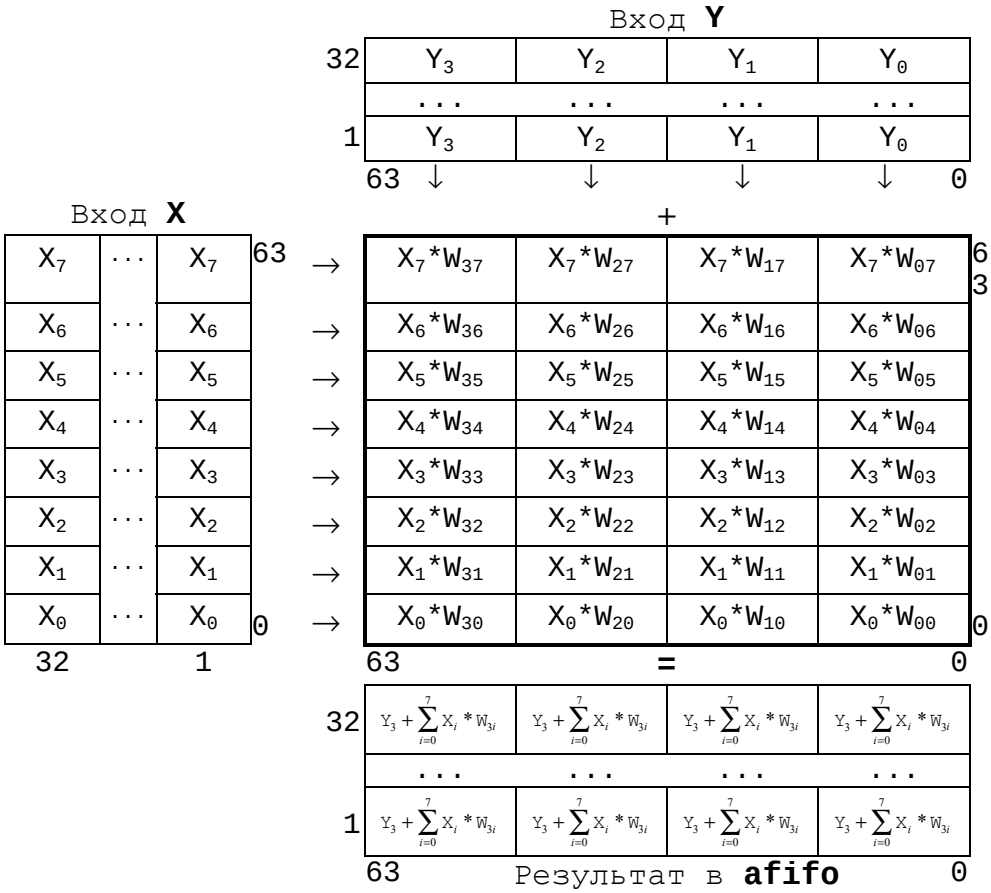
Кроме этого в качестве входов могут выступать так называемые "нулевые" устройства, что означает, что данные на вход не поступают.

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

В рабочую матрицу предварительно загружаются весовые коэффициенты. Более подробно см. 3.3.8. Регистр-контейнер на стр. 3-58.

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 параграфе 3.3.1. Регистр (см. стр. 3-36)

Рис. 1-5 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 параграфе 3.3.2. Регистр nb1 (см. стр. 3-41)

Рисунок Рис. 1-5 показывает, какие действия может выполнить ВП при помощи одной процессорной инструкции. Допустим, в буфер данных `ram` было предварительно загружено из памяти 32 длинных слова. При операции взвешенного суммирования из памяти по очереди подчитываются слова входных данных, каждое из которых направляется на вход **X** операционного узла ВП. Параллельно из буфера `ram` подчитывается очередное слово и направляется на вход **Y**. Каждый элемент, составляющий слово на входе **X**, умножается на весовой коэффициент, находящийся в соответствующей ячейке рабочей матрицы, результаты умножения складываются в пределах столбца, а затем к ним добавляется значение элемента, находящегося на соответствующей позиции в слове, поступившем на вход **Y**. 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** рабочей матрицы, или

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

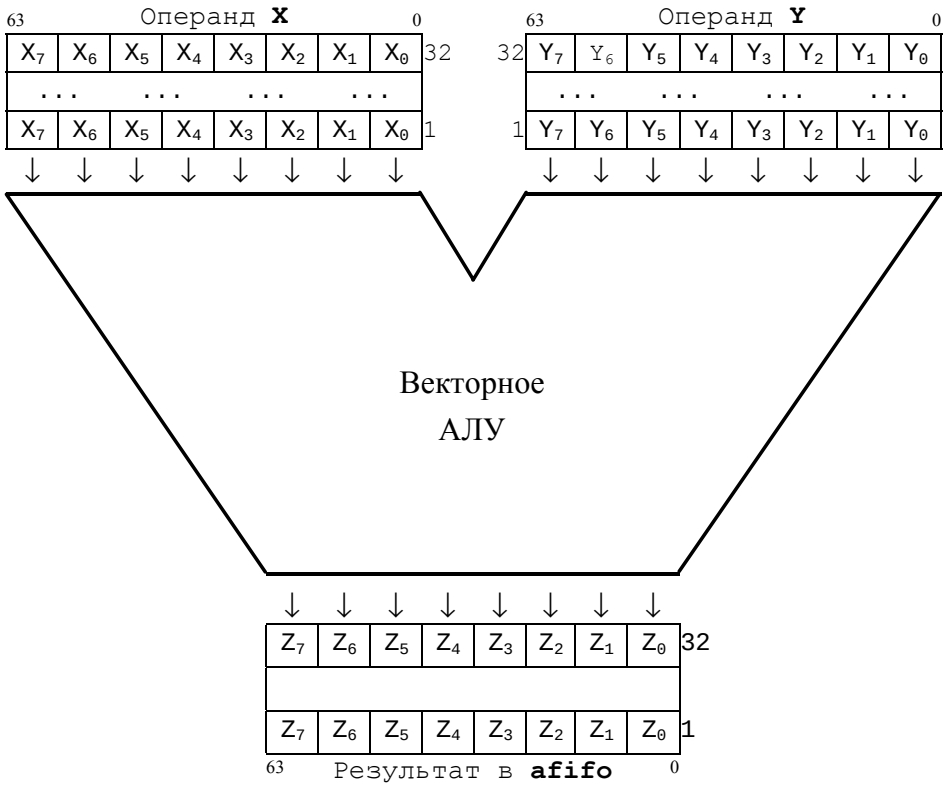
Например, если разбиение входа **X**- 8 бит, то есть каждое слово представляется как 8 элементов по 8 бит, а разбиение **Y** - 16 бит, то в зависимости от того, куда будут направлены данные, хранящиеся в `ram`, на вход **X** или **Y**, они будут трактоваться либо как массив 8-ми, либо 16-ти битных элементов.

1.4.2 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**. 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. 1-6 Выполнение вычислений на векторном АЛУ.



Операции на векторном АЛУ выполняются с учетом разбиения входных данных на элементы (см. Рис. 1-6). 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. 1-7 приведен пример сложения двух 64-х разрядных слов, разбитых на 8 элементов по 8 бит каждый, на векторном АЛУ процессора NM6403.

Рис. 1-7 Сложение двух 64-х разрядных слов на векторном АЛУ.

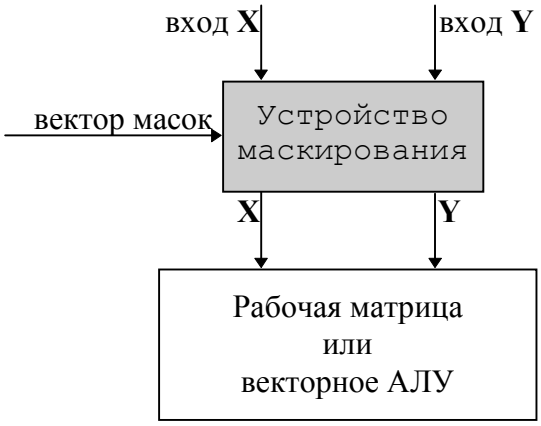
01	80	80	F0	02	FF	FF	01
+							
FE	20	80	1F	02	01	FF	01
=							
FF	A0	00	0F	04	00	FE	02

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

1.4.3 Операция маскирования

Для выполнения маскирования в операционном узле ВП существует специальное устройство. Оно имеет три входа и два выхода (см. Рис. 1-8). 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. 1-8 Положение устройство маскирования в ОУ ВП.



Векторные команды маскирования, как и все остальные векторные команды процессора NM6403, 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.

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 со 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 со 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.

Логическое маскирование

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

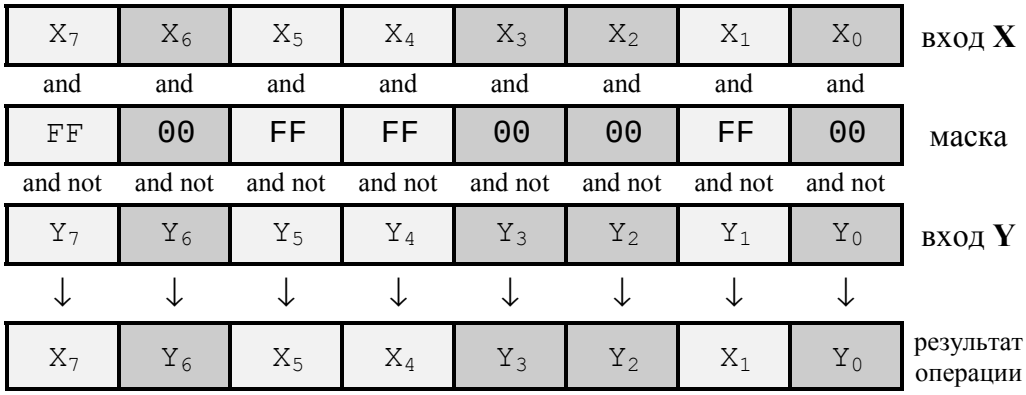
```
                                        маска    X       Y
rep 32 data = [ar0++] with mask ram, data, afifo;
```

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. 1-9 дается пояснение в графической форме:

Рис. 1-9 Выполнение операции маскирования.



В тех позициях, на которых в маске стоят единицы, в слово результата будут записаны биты слова со входа **X**, на остальные места попадут биты слова со 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.

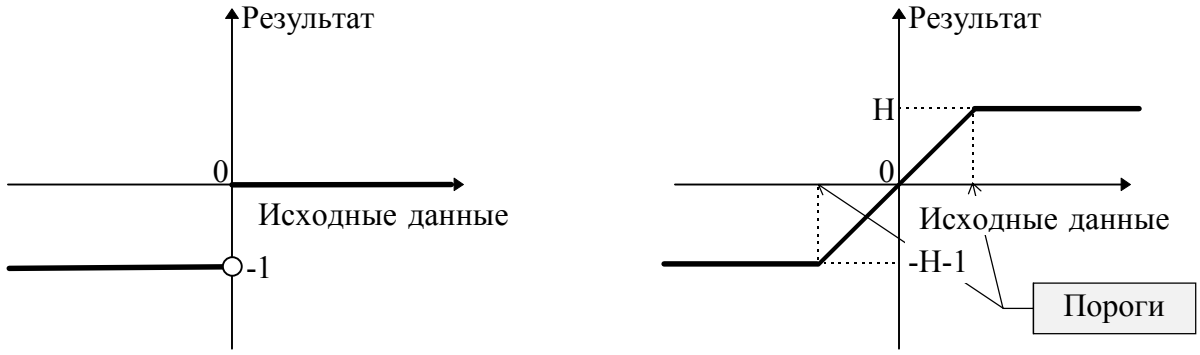
1.4.4 Обработка данных функцией активации

Данные, поступающие на входы **X** и **Y** ВП, предварительно на проходе могут быть подвергнуты преобразованию кусочно-линейными функциями, называемыми функциями активации.

Всего существует два типа функций активации:

- пороговая функция (см. Рис. 1-10а);
- функция насыщения (см. Рис. 1-10б).

Рис. 1-10 Типы встроенных функций активации процессора NM6403.



а) Пороговая функция *б) Функция насыщения*

Основную роль в управлении функциями активации играют регистры `f1cr` и `f2cr`. Более подробную информацию об их роли в управлении обработкой потоков данных см. параграф 3.3.1. Регистры f1cr и на стр. 3-36.

Активация выполняется перед тем, как данные попадут в рабочую матрицу или на векторное АЛУ. Ей 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.

Арифметическая активация

Функция насыщения выполняется только в составе команды взвешенного суммирования (в том числе и с маскированием), и в арифметических командах на векторном АЛУ. Действие, которое она выполняет, называется также арифметической активацией. Второе название удобно, поскольку позволяет очертить круг векторных команд, совместно с которыми может выполняться функция насыщения. Это векторные команды процессора, содержащие арифметические действия.

Логическая активация

Другой набор команд включает все логические операции на векторном АЛУ, в том числе и операцию маскирования. Пороговая функция может быть использована только совместно с этим типом векторных команд, поэтому те преобразования, которые она совершает над данными, называются логической активацией.

Более подробная информация о выполнении активации приведены в параграфе 3.3.1. Регистры f1cr и на стр. 3-36.

1.4.5 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. 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. 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. В этом случае старшие биты двухбитовых слов становятся младшими.

Другими словами устройство циклического сдвига позволяет обеспечить побитовую коммутацию входных и выходных битов, то есть любой бит входного слова путем преобразования на рабочей матрице поставить в произвольную позицию в выходном.

Циклический сдвиг вправо осуществляется на один бит. При этом разбиение на элементы не учитывается, то есть сдвигается целиком все слово. Его младший бит становится самым старшим (63-им), первый нулевым, второй первым и т.д.

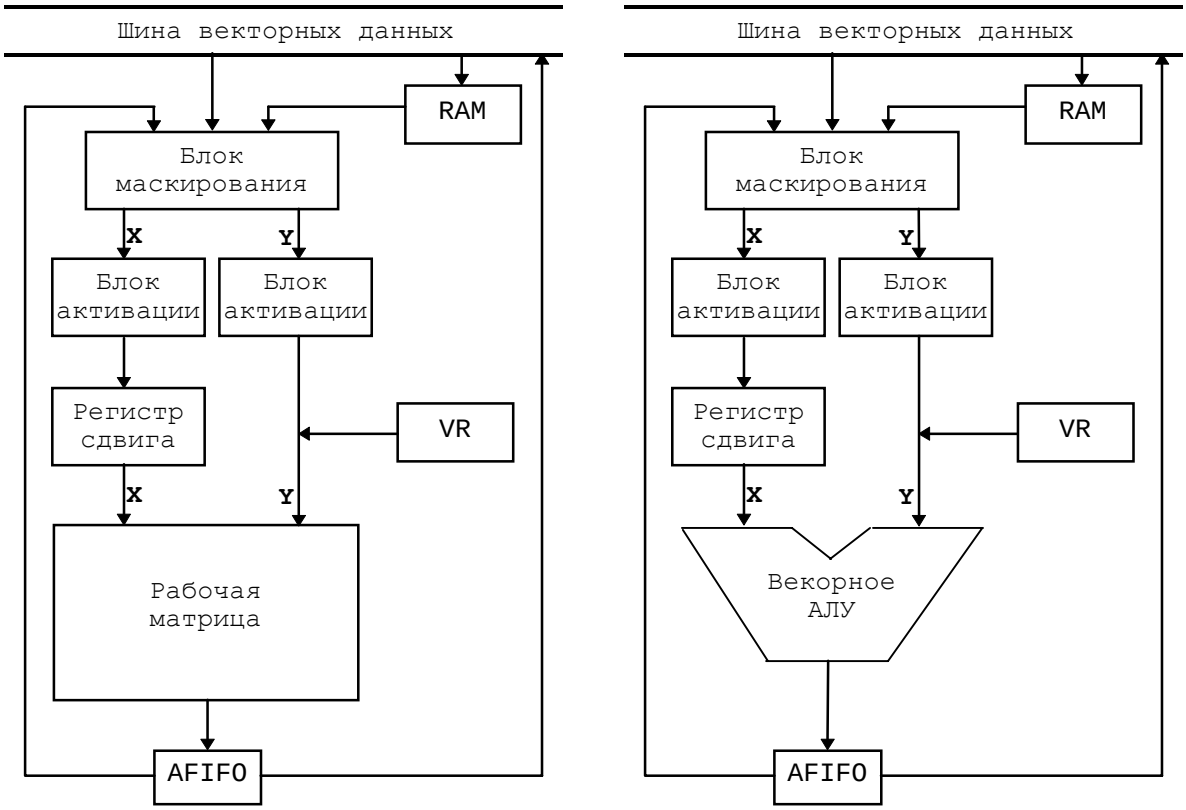
Циклический сдвиг работает при любом разбиении матрицы на строки, влияет только на данные, поступающие на вход **X**, встречается только в командах взвешенного суммирования и логического маскирования. Активизируется устройство циклического сдвига при использовании в векторной команде ключевого слова `shift`, расположенного перед операндом **X**. Более подробно см. раздел 5.2 Векторные инструкции NM6403 на стр. 5-27.

1.4.6 Порядок выполнения преобразований над данными на ВП

При выполнении вычислений на векторном процессоре помимо рабочей матрицы и векторного АЛУ могут использоваться также блоки, выполняющие маскирование, функции активации, циклический сдвиг данных. Если несколько устройств задействованы в одной векторной инструкции, то необходимо знать, в какой последовательности выполняются преобразования над данными.

В зависимости от того, направляются ли данные на обработку в векторное АЛУ или в рабочую матрицу, они проходят через разные вычислительные блоки:

Рис. 1-11 Порядок выполнения преобразований над данными в ВП.



a) Порядок прохождения данных через блоки обработки при **выполнении операции взвешенного суммирования**.

b) Порядок прохождения данных через блоки обработки при **выполнении операций на векторном АЛУ**.

Примечание

На Рис. 1-11b регистр сдвига может использоваться только в операции логического маскирования (см. 1.4.3.Операция маскирования *на стр. 1-14).*

Каждый блок обработки может пропускать через себя данные без изменений, либо модифицировать их. Это определяется ассемблерной инструкцией. Если в инструкции указано, например, что необходимо выполнить циклический сдвиг вправо операнда **X**, то регистр сдвига включается в работу, иначе он пропускает данные через себя не модифицируя их.

На Рис. 1-11 показана последовательность преобразований, которым подвергаются данные в процессе вычислений на векторном процессоре. В качестве небольшого пояснения к приведенным выше схемам можно заметить, что если в ассемблерной инструкции встречаются, например, маскирование и активация одного из операндов, то сначала будет выполнена операция маскирования, а затем активация.

С точки зрения реализации на процессоре NM6403 нейронных сетей необходимо отметить, что активация значений операндов

выполняется до операции взвешенного суммирования. Поэтому, если возникает необходимость выполнить активацию после взвешенного суммирования, требуется дополнительный проход через ВП, например, по сценарию Рис. 1-11b с выполнением на векторном АЛУ какого-либо тождественного преобразования.

2 Обзор основных элементов языка ассемблера

2.1 Служебные слова ... 2-3
2.2 Структура ассемблерного файла ... 2-5
2.3 Секции ... 2-5
2.3.1 Секции кода ... 2-6
2.3.2 Секции инициализированных данных ... 2-7
2.3.3 Секции неинициализированных данных ... 2-8
2.3.4 Пространство между секциями ... 2-8
2.4 Константы ... 2-9
2.4.1 Форматы представления констант ... 2-9
2.4.1.1 Двоичные целые константы ... 2-9
2.4.1.2 Восьмеричные целые константы ... 2-10
2.4.1.3 Десятичные целые константы ... 2-10
2.4.1.4 Шестнадцатеричные целые константы ... 2-10
2.4.1.5 Константы с плавающей точкой ... 2-10
2.4.1.6 Строковые константы ... 2-11
2.4.2 Константные выражения ... 2-11
2.4.2.1 Числовые константные выражения ... 2-12
2.4.2.2 Адресные константные выражения ... 2-12
2.4.3 Определение и использование константы ... 2-13
2.5 Метки ... 2-14
2.5.1 Объявление метки ... 2-14
2.5.2 Определение метки ... 2-14
2.5.3 Ссылки на метку ... 2-15
2.5.4 Типы связывания и область действия меток ... 2-15
2.6 Переменные ... 2-18
2.6.1 Получение адреса переменной ... 2-19
2.6.2 Получение значения переменной ... 2-19
2.6.3 Простые переменные ... 2-19
2.6.4 Составные переменные ... 2-20
2.6.4.1 Массивы ... 2-20
2.6.4.2 Структуры ... 2-20
2.6.5 Начальные значения ... 2-22
2.6.6 Область действия данных ... 2-23
2.6.7 Места для объявлений и начальной инициализации переменных ... 2-24
2.7 Директивы языка ассемблера ... 2-25
2.7.1 Директива .align ... 2-27
2.7.2 Директива .branch ... 2-28
2.7.3 Директива .endif ... 2-28
2.7.4 Директива .endrepeat ... 2-29
2.7.5 Директива .if ... 2-29

2.7.6 Директива .repeat 2-30
2.7.7 Директива .wait 2-30
2.7.8 Директивы отладочной информации 2-30
2.7.8.1 Директива .debug_arange 2-30
2.7.8.2 Директивы .debug_die и .debug_die_child 2-31
2.7.8.3 Директива .debug_die_endchild 2-33
2.7.8.4 Директивы .debug_start_sequence и .debug_end_sequence 2-33
2.7.8.5 Директива .debug_frame_cie 2-34
2.7.8.6 Директива .debug_frame_fde 2-34
2.7.8.7 Директива .debug_line 2-34
2.7.8.8 Директива .debug_pubname 2-35
2.7.8.9 Директива .debug_root_die 2-35
2.7.8.10 Директива .debug_source_directory 2-35
2.7.8.11 Директива .debug_source_file 2-35
2.8 Псевдофункции 2-36
2.8.1 Функция loword 2-37
2.8.2 Функция hiword 2-37
2.8.3 Функция sizeof 2-37
2.8.4 Функция offset 2-38
2.8.5 Функция float 2-39
2.8.6 Функция double 2-39

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> ar0 = ar2 + gr2;       // Модификация адресного рег.
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 .

2.1 Служебные слова

В расположенных ниже таблицах приведены отдельно основная группа служебных слов, и служебные слова, используемые для хранения отладочной информации.

Табл. 2-1 Основные служебные слова языка ассемблера.

Activate	addr	afifo	align	and	begin	branch
call	callrel	carry	cfalse	clear	code	common
const	ctrue	data	delayed	double	dup	end
endif	endrepeat	extern	false	flag	float	from
ftw	global	goto	hiword	if	import	ireturn
label	local	locdesc	long	loword	macro	mask
nobits	noflags	not	nul	offset	own	push
pop	ram	ref	rep	repeat	return	sconst
set	shift	sizeof	skip	store	string	struct
true	uconst	vfalse	vnul	vregs	vsum	vtrue
wait	weak	wfifo	with	word	wtw	xor

Табл. 2-2 Служебные слова языка ассемблера для целей отладки.

debug_arange	debug_die	debug_die_child
debug_die_endchild	debug_end_sequence	debug_frame_cie
debug_frame_fde	debug_line	debug_macro_def
debug_macro_end_file	debug_macro_start_file	debug_macro_undef
debug_pubname	debug_root_die	debug_source_directory
debug_source_file	debug_start_sequence	

Примечание

Ассемблер учитывает регистр символов (различает прописные и строчные), поэтому служебные слова должны записываться строчными буквами, в противном случае слово будет воспринято как идентификатор.

Все регистры процессора также являются ключевыми словами. Они **всегда** записываются строчными буквами. Например, запись "gr0" обозначает регистр процессора, а "Gr0" или "GR0" - это уже идентификаторы переменных.

Табл. 2-3 Регистры процессора NM6403.

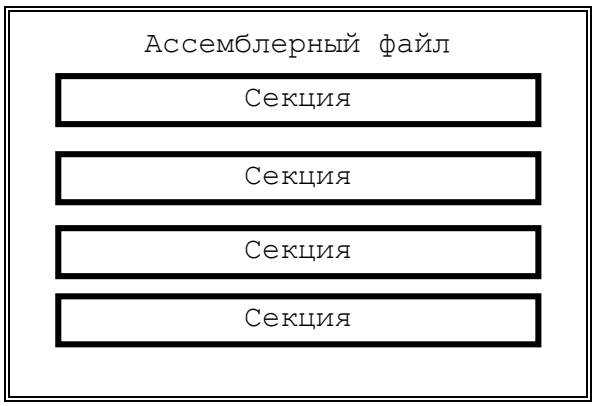
Обозначение	Назначение	Разрядность
grj	Регистр общего назначения j (j=0,…,7)	32
arj	Адресный регистр j (j=0,…,7)	32
pc	Счётчик команд	32
pswr	Регистр слова состояния процессора	32
intr	Регистр запросов на прерывание и прямой доступ к памяти (ПДП)	32
fjcr(h,l)	Регистр управления функцией активации j (j=1,2)	64(32+32)
vr(h,l)	Векторный регистр	64(32+32)
nb1(h,l)	Регистр границ нейронов	64(32+32)
sb(h,l)	Регистр границ синапсов	64(32+32)
gmicr	Регистр управления интерфейсом с глобальной шины	32
lmicr	Регистр управления интерфейсом с локальной шины	32
ocaj	Регистр адреса канала вывода j (j=0,1)	32
icaj	Регистр адреса канала ввода j (j=0,1)	32
occj	Счетчик канала вывода j (j=0,1)	32

iccj	Счетчик канала ввода j (j=0,1)	32
dorj	Регистр данных канала вывода j (j=0,1)	64
dirj	Регистр данных канала ввода j (j=0,1)	64
Tj	Регистр управления таймером j (j=0,1)	32

2.2 Структура ассемблерного файла

Файл, разрабатываемый 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 ассемблера для процессора NM6403, 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 **запятая 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 неинициализирована, 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 компиляции компилятор разделяет инициализированные и неинициализированные данные. Для неинициализированных переменных создается дополнительная секция, куда помещаются все обнаруженные переменные. Существует правило выбора имени секции неинициализированных данных, автоматически порождаемой компилятором. 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."`*. То есть, например, разбирая секцию* AAAA *в ассемблерном файле, компилятор создаст в объектном файле 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` и `extern`;
- 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.

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**, добавляемый в конец. Например, константа `A = 0x123h` является 32-х разрядной, а `A=123hl` 64-х разрядной.

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;
- 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**, которая соответствует данному формату числа в представлении компилятора. Если количество элементов в строке двоичной константы меньше 32-х или 64-х, неопределенные биты слева (старшая часть слова) автоматически заполняются нулями при компиляции. Примеры правильно записанных двоичных констант:

`00000000b` // *Константа, равная нулю.*

```
001bl           //      Длинная константа, равная 1.
10000b          //      Константа, равная 16(10).
10101010b       //      Константа, равная 170(10) или AA(16).
```

2.4.1.2 Восьмеричные целые константы

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

```
000o            //      Константа, равная нулю.
001o            //      Константа, равная единице.
20ol            //      Длинная константа, равная 16(10).
252o            //      Константа, равная 170(10) или AA(16).
```

2.4.1.3 Десятичные целые константы

Десятичные целые константы представляют собой строки, состоящие из чисел от 0 до 9. Диапазон изменения составляет от -2.147.483.647 до 4.294.967.295 для коротких констант и от -9.223.372.036.854.775.808 до 184.467.440.737.709.551.616 для длинных констант. Примеры правильно записанных десятичных констант:

```
7000            //      Константа, равная 7000(10).
-1              //      Константа, равная минус единице.
-20l            //      Длинная константа, равная -20.
170             //      Константа, равная 170(10) или AA(16).
```

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 чисел от `0` до `9`, а также букв `A`, `B`, `C`, `D`, `E` и `F`, в конце которых следует символ **h**. Запись константы должна начинаться с цифры. Если первым значимым символом является буква, то перед ней необходимо поставить `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 32(10).
0FFFFFFFFh//    Константа, равная -1.
```

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[.numE][+|-]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 битное 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.

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. В языке имеется набор из обычных арифметических операций, операций побитового сложения, умножения, сдвига и логических операций сравнения. Приоритеты операций совпадают с приоритетами идентичных операций языка Си++. Для частичного изменения порядка выполнения операций используются круглые скобки.

Табл. 2-4 Сводка операций численных констант ных 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

Пример числового константного выражения:

```
const A = 117;
const B = 23;
const C = ((A + B)/2 + A>>2)*2;
```

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 – адресный; запрещено складывать два адреса;
- разность двух адресов из одной секции является численной константой; 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 константы

В языке ассемблера константа определяется с помощью ключевого слова `const`, например:

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

В правой части данной синтаксической конструкции записана числовая константа или константное выражение, слева ее символьный эквивалент, который может в дальнейшем использоваться для инициализации переменных и регистров.

Такая константа называется константой времени компиляции ассемблера, потому что она существует только в период компиляции программы. Сама по себе она не занимает место в памяти, то есть не обладает такой характеристикой, как адрес. Однако, если какая-то переменная инициализирована данной константой, то во время компиляции значение константы записывается по адресу переменной.

Определение константы может располагаться в произвольном месте файла; как внутри секций, так и за их пределами. Часто, для того чтобы показать что константа не зависит от наличия какой-либо секции, ее выносят в пространство между секций.

Символьная константа должна быть определена до момента ее первого использования, например:

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

```
end text;
```

Символьные константы являются полным эквивалентом численных констант и могут использоваться на равных с ними условиях, то есть могут участвовать в константных выражениях, использоваться для инициализации переменных и регистров.

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

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 представляется в виде символьной строки, которая должна начинаться с латинской буквы или символа "_", и может содержать латинские буквы, как строчные, так и прописные, цифры и символ "_". Приведем примеры корректных имен меток:

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

2.5.1 Объявление метки

Прежде чем использовать метку, она должна быть объявлена, например:

```
MyLabel: label;
```

Объявление метки может происходить в любом месте ассемблерного файла; как внутри секций, так и вне их. Объявление - это просто сообщение компилятору, что такая метка должна встретиться в данном файле.

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 для NM6403 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, 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 объявлена только один раз, только один раз она может быть определена, то есть ассоциирована с определенным адресом в программе. 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.

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`.

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 для NM6403 понимается адрес ячейки в памяти процессора, в котором хранятся данные, используемые программой в процессе счета.

Переменная представляется в виде символьной строки, которая должна начинаться с латинской буквы или символа "_", и может содержать латинские буквы, как заглавные, так и прописные, цифры и символ "_". Приведем примеры корректных имен переменных:

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

Каждая переменная в языке ассемблера ассоциирована с адресом конкретной ячейки памяти. Переменные доступны для чтения или записи.

В языке ассемблера для NM6403 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

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];
```

Имеется несколько типов переменных, которые делятся на две группы: простые и составные переменные.

2.6.3 Простые переменные

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

Так как процессор NM6403 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 битных и `'long'` для 64-х битных 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 неиспользуемое короткое 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 = [Sruct.F3];
    ...
end;
```

Примечание

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.

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 ассемблером в процессе компиляции (см. документ: Базовое программное обеспечение процессора NM6403. Руководство программиста.).

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 модуля и недоступны (неизвестны) вне его, в других модулях, образующих программу;
- `extern` - внешние переменные, описанные в некотором другом модуле и используемые в данном;
- `global` - глобальные данные, описанные в данном модуле и доступные для других модулей программы;
- `weak` - глобальные данные со слабым типом связывания, описанные в данном модуле и доступные для других модулей 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 модуле и доступные из любого модуля (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;
waek 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 (см. раздел 2.5. Метки на стр. 2-14)

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 `local`, `global`, `weak` должны объявляться **только в пределах секций**. Для этого специально отведены секции инициализированных и неинициализированных данных. Более подробно о секциях данных см. раздел 2.3. Секции на стр. 2-5.

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;
commom Ecomm: long; // правильное объявление;
       Floc : long; // неправильное объявление;
weak   Gweak: word; // неправильное объявление;
extern Hextr: word; // правильное объявление;

data ".data1"
     // Область файла внутри секции.
     commom Icomm: long; // неправильное объявление;
     extern Jextr: word; // неправильное объявление;
end ".data1";
```

2.7 Директивы языка ассемблера

В данном разделе перечислены все директивы ассемблера, описано их назначение, приведены правила работы с ними. В отличии 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. 2-5 Сводная таблица директив языка ассемблера (Часть 1).

Мнемоника	Описание
.align	Выравнивание по четному адресу.
.branch	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	Отключение режима параллельного выполнения.

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

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

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

Табл. 2-6 Сводная таблица директив языка ассемблера (Часть 2).

Мнемоника	Описание
.debug_arange	информация о адресных диапазонах программы;
.debug_die	описывает новую DIE(Debug Information Entry);
.debug_die_child	описывает новую DIE, дочернюю по отношению к .debug_die;
.debug_die_endchild	указывает на окончание цепочки текущего уровня;
.debug_end_sequence	маркер конца куска непрерывного кода;
.debug_frame_cie	информация о стеке вызовов;
.debug_frame_fde	информация о стеке вызовов;
.debug_line	информация о строках исходного текста;
.debug_pubname	информация о глобальных символах;
.debug_root_die	описывает корневую DIE компиляционной единицы (CU);
.debug_source_directory	информация о каталогах, в которых находятся исходные тексты;
.debug_source_file	Информация о файлах исходных текстов;
.debug_start_sequence	маркер начала куска непрерывного кода;

Каждая из директив отладочной информации имеет несколько аргументов, перечисленных через запятую, и оканчивается точкой с запятой. Например: `.debug_line 2, 3, 1;`

Для того, чтобы правильно понимать назначение директив отладочной компиляции, их структуру и принципы использования, необходимо предварительно ознакомиться с описанием стандарта представления отладочной информации в формате DWARF. (документ: **TIS Committee "DWARF Debugging Information Format. v2.0"**).

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

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 текущей секции.

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` используется в программах на языке ассемблера для NM6403 для того, чтобы завершить блок условной компиляции.

Директива `.endif` используется только в паре с `.if`, и самостоятельного значения не имеет. Пример использования `.endif` приведен в параграфе 2.7.5 Директива .if на стр. 2-29.

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. на стр. 2-30.

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

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.

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 на стр. 2-28.

2.7.8 Директивы отладочной информации

2.7.8.1 Директива .debug_arange

Директива .debug_arange добавляет информацию о диапазонах адресов, описанных в текущей единицы компиляции CU(Compilation Unit) для быстрого поиска по адресу. У директивы

два параметра: адрес и длина. Адрес – перемещаемый в целевом пространстве. Соответственно, значением этого параметра может быть только выражение, вычисляемое как перемещаемое.

Пример:

```
.debug_arange   function, fend - fbegin;
// диапазон адресов:
//    [function .. function + fend - fbegin]
// принадлежит коду описанному в данной CU.
```

2.7.8.2 Директивы .debug_die и .debug_die_child

Создают новый элемент хранения отладочной информации DIE (Debug Information Entry) в текущей CU. Директива .debug_die создает DIE как брата предыдущего DIE. Директива .debug_die_child создает DIE как сына предыдущего DIE.

Первый параметр директивы содержит идентификатор, используемый ассемблером в качестве метки DIE для ссылок на текущий DIE из других отладочных директив. Эта метка никак не связана с именем DIE, которое является значением атрибута DW_AT_name.

Второй параметр директивы содержит целочисленный тег типа DIE. Значения различных тегов заданы в описании формата DWARF. В макробиблиотеке dwarf_ct.mlb, поставляемой вместе с библиотекой, для каждой величины стандарта DWARF определены константы, имена которых совпадают с используемыми в стандарте. Как и в стандарте, имена тегов начинаются с DW_TAG_. В дальнейшем описании вместо численных значений будут использованы имена.

Остальные параметры директивы представляют собой список описаний атрибутов DIE:

имя, форма, значение,

имя, форма, значение, ...

Имя атрибута является константным выражением. В качестве символьного имени можно использовать имена, определённые в стандарте DWARF (в dwarf_ct.mlb). Стандартные имена атрибутов начинаются с DW_AT_.

Форма - служебное слово, задающее форму, в которой представлено значение атрибута. Возможные формы атрибутов DIE представлены в Табл. 2-7.

Табл. 2-7 Формы атрибутов DIE.

Форма	Описание
`ref`	Ссылка на другую DIE.
`Addr`	Адрес в целевом пространстве.

`Flag`	Флаг.
`Uconst`	Константа без знака.
`Sconst`	Константа со знаком.
`Block`	Блок данных.
`Locdesc`	Описатель местоположения.
`String`	Символьная строка.

Значение - значение атрибута, которое зависимости от формы атрибута DIE, может иметь различный смысл (см. Табл. 2-8.)

Табл. 2-8 Связь значений атрибутов DIE с их формой.

Форма	Значение	Примечание
`ref`	Метка DIE (первый параметр какой-либо DIE).	Ассемблер формирует значение этого атрибута как смещение DIE, на которую была ссылка, относительно CU в секции объектного файла с именем ".debug_info".
`addr`	Перемещаемый адрес в целевом пространстве.	Ассемблер обрабатывает его как и все перемещаемые адреса в программе. Значением атрибута может служить либо метка , либо переменная, либо адресное выражение.
`Flag`	Константа размером в один байт.	
`uconst, sconst`	Константа, понимаемая, как число без знака или со знаком, соответственно.	
`block`	Блок данных, т.е. не интерпретируемая последовательность байтов, перечисленных через запятую в фигурных скобках.	Значения в фигурных скобках могут быть либо константами, понимаемыми как байты, либо адресными выражениями.
`locdesc`	Последовательность команд абстрактной стековой машины, определяющая местоположение объекта, описываемого DIE.	Вся последовательность должна быть заключена в фигурные скобки, кроме того, каждая команда, состоящая из трёх констант (возможно адресных), также должна быть заключена в фигурные скобки:

		{{12,23,24},{24,24,0},...}
string	Символьная строка.	Любые символьные строки, используемые в отладочных командах, должны заключаться в одинарные кавычки, иначе, в соответствии с правилами ассемблера, каждый символ строки в отладочной информации будет представляться четырьмя байтами.

Примеры псевдокоманды .debug_*die*:

```
.debug_root_die die1856, DW_TAG_compile_unit,
   DW_AT_name,string,'myfile.c',
   DW_AT_producer,string,'Compiler: version 13',
   DW_AT_compdir,string,'/home/mydir/src',
   DW_AT_language,flag,1,        // DW_LANG_C89
   DW_AT_low_pc,addr,start // start – это метка
   DW_AT_high_pc,addr,start + 138;


.debug_die_child  die78235, DW_TAG_base_type,
   DW_AT_name,string,'char',
   DW_AT_encoding,flag,8,  // DW_ATE_unsigned_char
   DW_AT_byte_size,flag,1;


.debug_die  die1234, TAG_pointer_type
   DW_AT_type,ref,die78235;
.debuf_die_endchild;
.debug_die  die21, TAG_typedef
   AT_name,string,'POINTER',
   AT_type,ref,die1234;
```

2.7.8.3 Директива .debug_die_endchild

Завершает текущий уровень дерева DIE, происходит переход на уровень выше (к отцу последних DIE), следующий DIE будет братом отца последнего DIE.

Команда не имеет параметров.

2.7.8.4 Директивы .debug_start_sequence и .debug_end_sequence

Директивы .debug_start_sequence и .debug_end_sequence ограничивают куски непрерывного кода программы. Команды .debug_line могут встречаться только между этих ограничителей.

Команды не имеют параметров.

2.7.8.5 Директива .debug_frame_cie

Добавляет новую запись (Call Information Entry) к информации о стеке вызова.

Параметры:

- строка-информация о функции, порождающей CIE,
- выравнивание кода,
- выравнивание данных,
- номер регистра адреса возврата,
- список команд абстрактной машины – инструкции, содержащиеся в CIE (записываются в фигурных скобках).

Пример:

```
.debug_frame_cie 'func()',2,2,7, {{1,2,3},{4,5,0}};
```

2.7.8.6 Директива .debug_frame_fde

Добавляет новую запись (Frame Description Entry) к информации о стеке вызова.

Параметры:

- порядковый номер соответствующей CIE,
- адрес в целевом пространстве начала данной функции,
- длина кода функции,
- список команд абстрактной машины – инструкции, содержащиеся в FDE (в фигурных скобках).

Пример:

```
.debug_frame_fde 1, addr, 2056, {{1,2,6},{45,7,4}};
```

2.7.8.7 Директива .debug_line

Директива .debug_line добавляет информацию о номере строки, соответствующей данному адресу.

Формат записи:

```
.debug_line     <номер строки>
              [, <индекс файла> ]
              [, <номер колонки>]
```

Индекс файла должен быть назначен ранее командой .debug_source_file. Индекс файла может отсутствовать, означая, что он тот же, что и у предыдущей команды .debug_line. Самая первая команда .debug_line должна иметь индекс файла. В качестве адреса строки ассемблер использует текущий адрес. Вместе со вторым параметром (индексом файла) может быть опущен и третий параметр (номер столбца), тогда номер столбца в данной строке считается неопределенным.

Пример:

```
.debug_line     10,5,6;   // 10-я строка 5-го файла
.debug_line     11;       // 11-я строка того же файла.
```

2.7.8.8 Директива .debug_pubname

Директива `.debug_pubname` добавляет имя глобального символа и ссылку на DIE, содержащую информацию об этом символе, в секцию .debug_pubname для ускорения поиска по символу.

У команды два параметра:

- имя символа в виде символьной строки;
- ссылка на DIE в виде метки DIE.

Пример:

```
.debug_pubname  'TBigArray::find',die123;
```

2.7.8.9 Директива .debug_root_die

Создает компиляционную единицу CU (Compilation Unit), и добавляет в нее корневую DIE, содержащую в себе атрибуты CU в целом. Все последующие директивы .debug_die создают новые DIE в CU. Параметры у данной директивы такие же, как и у .debug_die, т.к. тоже описывают DIE. Эта директива отделена от .debug_die чтобы логически выделить корневую DIE.

Компилятором ассемблера поддерживается существование только одной компиляционной единицы, поэтому повторное вхождение директивы .debug_root_die не допускается.

2.7.8.10 Директива .debug_source_directory

Директива .debug_source_directory имеет два параметра:

- индекс каталога;
- строку-путь к каталогу.

Индекс файла имеет лишь информативное значение и должен совпадать с порядковым номером команды .debug_source_directory (среди всех таких команд), нумерация начинается с единицы.

Все исходные файлы должны быть доступны по одному из каталогов заданных командой .debug_source_directory.

Пример:

```
.debug_source_directory    3,'/user/myfiles/';
```

2.7.8.11 Директива .debug_source_file

Директива .debug_source_file имеет пять параметров:

- индекс файла,
- индекс каталога,

- строка-имя,
- размер,
- дата файла исходного текста.

Размер и дата, хотя и являются обязательными, однако их правильность никак не проверяется, эта информация может использоваться отладчиком. Все исходные файлы должны быть описаны командами .debug_source_file. Индекс файла используется псевдокомандами для указания на файлы. Индекс каталога должен быть задан ранее командой .debug_source_directory.

Пример:

```
.debug_source_file   6,3,'myfile.c',1352,19071996;
```

2.8 Псевдофункции

В язык ассемблера для NM6403 введено несколько псевдокоманд, облегчающих запись и вычисление константных выражений программы и повышающих ее наглядность.

Псевдофункции обрабатываются ассемблером на этапе трансляции. 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. 2-9 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`, в котором осуществляется пересылка данных из одной области памяти в другую при помощи векторного процессора:

```
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[.numE][+|-]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 отличии от `float` функция `double` оперирует с 64-х разрядными числами.

3 Регистры

3.1 Основные регистры 3-3
3.1.1 Адресные регистры 3-3
3.1.2 Регистры общего назначения 3-4
3.1.3 Регистровые пары 3-4
3.2 Специальные регистры 3-5
3.2.1 Регистр gmicr 3-6
3.2.2 Регистры управления коммуникационными портами: (ica, icc), (oca, occ) 3-13
3.2.3 Регистр intr 3-19
3.2.4 Регистр lmicr 3-24
3.2.5 Регистр pc 3-25
3.2.6 Регистр pswr 3-25
3.2.7 Регистры таймеров t0, t1 3-33
3.3 Векторные регистры 3-34
3.3.1 Регистры f1cr и f2cr 3-36
3.3.2 Регистр nb1(nb2) 3-41
3.3.3 Регистр sb(sb1 и sb2) 3-45
3.3.4 Регистр vr 3-49
3.3.5 Регистр-контейнер afifo 3-50
3.3.6 Логический регистр-контейнер data 3-54
3.3.7 Регистр-контейнер ram 3-56
3.3.8 Регистр-контейнер wfifo 3-58

В данном пункте рассматриваются три группы регистров процессора NM6403:

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

Адресные и регистры общего назначения образуют группу основных регистров.

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 процессора NM6403.

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 делятся на две равноправные группы. В первую входят `ar0 .. ar3`, а во вторую `ar4..ar7`. Это связано с наличием двух адресных устройств в процессоре.

Существуют ограничения на возможность использовать адресные регистры из разных групп в одной процессорной инструкции. Более подробные замечания на этот счет будут даны в разделе, описывающем операции модификации адресных регистров (см. XXXXXXX).

Адресные регистры могут использоваться только в левой части ассемблерной инструкции (см. разделы 5.1 Скалярные инструкции NM6403 на стр. 5-3 и 5.2 Векторные инструкции NM6403 на стр. 5-27).

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. О том, как именно это происходит, см. XXXXXXXX.

3.1.2 Регистры общего назначения

Регистры общего назначения в отличии от адресных не имеют разделения на группы, 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 NM6403 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.
```

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

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.

3.2.1 Регистр gmicr

Поле BOUND 3-6
Поле PAGE1 3-8
Поле PAGE0 3-8
Поле TYPE1 3-9
Поле TYPE0 3-9
Поле TIME1 3-9
Поле TIME0 3-11
Поле TRAS 3-11
Поле RDY1 3-12
Поле RDY0 3-12
Поле SHMEM 3-13

Регистр `gmicr` имеет разрядность 32 бита. С его помощью осуществляется управление доступом к внешней памяти через глобальную шину. Путем конфигурирования данного регистра программист может открыть доступ по глобальной шине к одному или двум банкам памяти, различающимся типом, страничной организацией и динамическими параметрами.

Регистр `gmicr` доступен как для чтения, так и для записи.

Он в обязательном порядке должен быть установлен при загрузке прикладной исполняемой программы в процессор. Обычно его установкой занимается загрузчик, входящий в состав библиотеки загрузки и обмена.

Примечание

Пользователь вправе самостоятельно изменять содержимое регистра `gmicr`. *Однако при этом необходимо соблюдать предосторожность, поскольку неправильно сконфигурированный регистр не позволит процессору корректно осуществлять доступ к глобальной памяти, что может привести к зависанию программы.*

В приведенной ниже таблице Табл. 3-3 содержится информация о полях регистра `gmicr`.

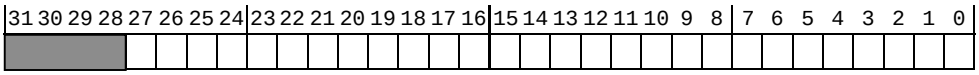
Табл. 3-3 Формат регистра управления интерфейсом с глобальной шиной GMICR.

31 30 29 28	27 26 25 24	23 22 21 20	19 18	17 16	15 14 13 12 11	10 9 8 7 6	5 4	3	2	1 0
BOUND	PAGE1	PAGE0	TYPE1	TYPE0	TIME1	TIME0	TRAS	RDY1	RDY0	SHMEM

Далее следует более подробное описание полей регистра `gmicr`.

Поле BOUND

Поле BOUND задает разбиение адресного пространства глобальной шины на банки 0 и 1. Оно определяет наличие банка 1 (банк 0 существует всегда), адрес начала банка 1, его размер. Биты, которые оно занимает в регистре gmicr, отмечены серым:



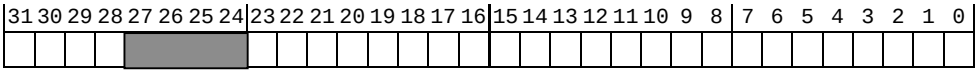
В приведенной ниже таблице Табл. 3-4. содержится информация о зависимости размеров банков и начальных адресов их расположения в глобальной памяти от комбинации битов в поле BOUND регистра gmicr.

Табл. 3-4 Разбиение адресного пространства глобальной шины на банки 0 и 1, задаваемое полем BOUND регистра gmicr.

BOUND	Размер банка 0 (64 бита)	Адресное пространство банка 0	Размер банка 1 (64 бита)	Адресное пространство банка 1
0000	2^{15} = 32K	80000000 - 8000FFFF	32K	80010000 - FFFFFFFF
0001	2^{16} = 64K	80000000 - 8001FFFF	64K	80020000 - FFFFFFFF
0010	2^{17} = 128K	80000000 - 8003FFFF	128K	80040000 - FFFFFFFF
0011	2^{18} = 256K	80000000 - 8007FFFF	256K	80080000 - FFFFFFFF
0100	2^{19} = 512K	80000000 - 800FFFFF	512K	80100000 - FFFFFFFF
0101	2^{20} = 1M	80000000 - 801FFFFF	1M	80200000 - FFFFFFFF
0110	2^{21} = 2M	80000000 - 803FFFFF	2M	80400000 - FFFFFFFF
0111	2^{22} = 4M	80000000 - 807FFFFF	4M	80800000 - FFFFFFFF
1000	2^{23} = 8M	80000000 - 80FFFFFF	8M	81000000 - FFFFFFFF
1001	2^{24} = 16M	80000000 - 81FFFFFF	16M	82000000 - FFFFFFFF
1010	2^{25} = 32M	80000000 - 83FFFFFF	32M	84000000 - FFFFFFFF
1011	2^{26} = 64M	80000000 - 87FFFFFF	64M	88000000 - FFFFFFFF
1100	2^{27} = 128M	80000000 - 8FFFFFFF	128M	90000000 - FFFFFFFF
1101	2^{28} = 256M	80000000 - 9FFFFFFF	256M	A0000000 - FFFFFFFF
1110	2^{29} = 512M	80000000 - BFFFFFFF	512M	C0000000 - FFFFFFFF
1111	2^{30} = 1G	80000000 - FFFFFFFF	0	-

Поле PAGE1

Поле `PAGE1` задает размер страницы памяти для банка 1 глобальной памяти. Биты, которые оно занимает в регистре `gmicr`, отмечены серым:



В приведенной ниже таблице Табл. 3-5 содержится информация о зависимости размеров страницы памяти банка 1 глобальной памяти от комбинации битов в поле `PAGE1` регистра `gmicr`.

Табл. 3-5 Размеры страниц памяти, задаваемые полями PAGE(0,1).

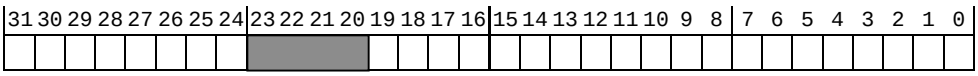
Поле PAGE(0,1)	Размер страницы памяти (в 64-х битных словах)
0000	2^8 = 256 слов
0001	2^9 = 512 слов
0010	2^{10} = 1К слов
0011	2^{11} = 2К слов
0100	2^{12} = 4К слов
0101	2^{13} = 8К слов
0110	2^{14} = 16К слов
0111	2^{15} = 32К слов
1000	2^{16} = 64К слов
1001	2^{17} = 128К слов
1010	2^{18} = 256К слов
1011	2^{19} = 512К слов
11xx	Резервная комбинация

Примечание

Комбинации битов 1000 и старше (последние 5 строк таблицы) могут использоваться только при работе со статической памятью (SRAM).

Поле PAGE0

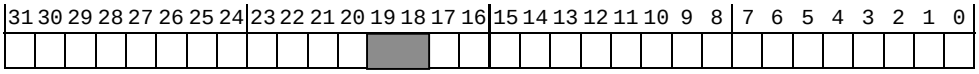
Поле `PAGE0` задает размер страницы памяти для банка 0 глобальной памяти. Биты, которые оно занимает в регистре `gmicr`, отмечены серым:



В таблице Табл. 3-5 содержится информация о зависимости размеров страницы памяти банка 0 глобальной памяти от комбинации битов в поле PAGE0 регистра gmicr.

Поле TYPE1

Поле TYPE1 определяет тип памяти, используемой для банка 1 глобальной памяти. Биты, которые оно занимает в регистре gmicr, отмечены серым:



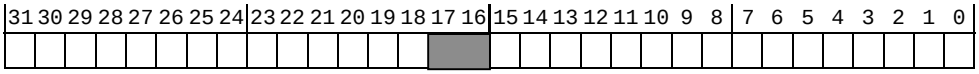
Поле может принимать 4 различных значения.

00	используется для доступа к статической памяти SRAM.
01,10,11	используются для доступа к динамической памяти DRAM.

Более подробная информация о различиях в типах динамической памяти содержится в документе: **Процессор NeuroMatrix®NM6403. Руководство пользователя.**

Поле TYPE0

Поле TYPE0 определяет тип памяти, используемой для банка 0 глобальной памяти. Биты, которые оно занимает в регистре gmicr, отмечены серым:



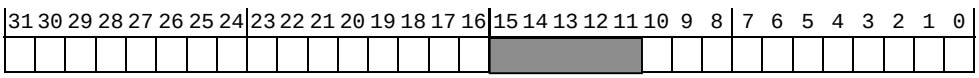
Поле может принимать 4 различных значения.

00	используется для доступа к статической памяти SRAM.
01,10,11	используются для доступа к динамической памяти DRAM.

Более подробная информация о различиях в типах динамической памяти содержится в документе**: Процессор NeuroMatrix®NM6403. Руководство пользователя.**

Поле TIME1

Поле TIME1 определяет временные параметры циклов обращения к банку 1 глобальной памяти. Биты, которые оно занимает в регистре gmicr, отмечены серым:



Циклы адресации памяти включают в себя от 1 до 5 фаз. Типы этих фаз и их программируемые длительности приведены в .

Табл. 3-6 Длительности фаз цикла обращения к памяти.

Фаза цикла обращения к памяти		Длительность фазы		
Обозначение	**Наименование**	**Обозначение**	**DRAM**	**SRAM**
RP	Фаза сброса сигналов $\overline{RAS0}$ / $\overline{CS0}$ и $\overline{RAS1}$ / $\overline{CS1}$.	T_{RP}	(1-2)T	1T
PAGE	Фаза адресации страницы памяти.	T_{PAGE}	(1-2)T	1T
CP	Пассивная фаза адресации ячейки памяти.	T_{CP}	(0-1)T	(0-3)T
CA	Активная фаза адресации ячейки памяти.	T_{CA}	(1-2)T	(1-4)T
BE	Фаза перехода выходов памяти в высокоимпедансное состояние.	T_{BE}	(1-2)T	(1-2)T

Далее приводится некоторая справочная информация, комментирующая отдельные положения, приведенные в таблице:

- Длительность `T` соответствует одному процессорному такту.
- Фаза `CP` является опциональной. Если ее длительность T_{CP} задана равной 0, то данная фаза не будет встречаться ни в одном из циклов адресации памяти.
- Длительность фаз `RP` и `PAGE` в циклах адресации SRAM не программируется пользователем и всегда равна одному процессорному такту.

Формат поля `TIME1` зависит от того, какой тип памяти используется, SRAM или DRAM.

Далее в таблице Табл. 3-7 приводится формат поля `TIME1` в случае использования статической памяти SRAM, а в таблице Табл. 3-8 его формат в случае использования динамической памяти DRAM.

Табл. 3-7 Форматы полей TIME0 и TIME1 регистра gmicr(lmicr) для SRAM.

Биты поля TIME1	Биты поля TIME0	Обозначение	Длительность фазы	
15	10	T_{BE}	0 1	2T 1T
14 13	9 8	T_{CP}	00 01 10 11	3T 2T 1T 0T
12 11	7 6	T_{CA}	00 01 10 11	4T 3T 2T 1T

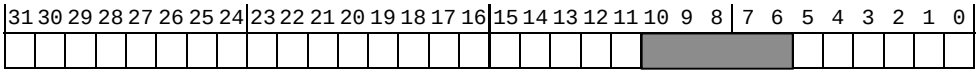
Табл. 3-8 Форматы полей TIME0 и TIME1 регистра gmicr(lmicr) для DRAM.

Биты поля TIME1	Биты поля TIME0	Обозначение	Длительность фазы	
15	10	T_{BE}	0 1	2T 1T
14	9	T_{PAGE}	0 1	2T 1T
13	8	T_{CP}	0 1	1T 0T
12	7	T_{RP}	0 1	2T 1T
11	6	T_{CA}	0 1	2T 1T

Более подробная информация о различиях временных параметрах доступа к различным типам памяти содержится в документе:
Процессор NeuroMatrix®NM6403. Руководство пользователя.

Поле TIME0

Поле `TIME0` определяет временные параметры циклов обращения к банку 0 глобальной памяти. Биты, которые оно занимает в регистре `gmicr`, отмечены серым:

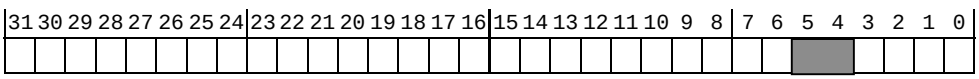


Структура поля `TIME0` идентична структуре поля `TIME1`, содержит аналогичное число циклов адресации, определяет те же фазы (см. Табл. 3-6) и времена доступа к различным типам памяти (см. Табл. 3-7 и Табл. 3-8).

Более подробная информация о различиях временных параметрах доступа к различным типам памяти содержится в документе:
Процессор NeuroMatrix®NM6403. Руководство пользователя.

Поле TRAS

Поле TRAS определяет длительность активного уровня сигнала $\overline{RAS}$. Биты, которые оно занимает в регистре `gmicr`, отмечены серым:



Ниже приведена Табл. 3-9, в которой отражается зависимость длительности активного уровня сигнала $\overline{RAS}$ от значения поля TRAS.

Табл. 3-9 Длительность активного уровня сигнала RAS.

Значение поля TRAS	Длительность в тактах (T_{RAS})
00	4 такта
01	3 такта
10	2 такта
11	1 такт

Примечание

Если в обоих полях TYPE0 и TYPE1указан тип памяти SRAM, то поле TRAS может принимать любое значение, поскольку оно не повлияет на доступ к памяти.

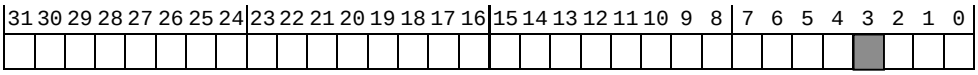
Более подробная информация о регенерации динамической памяти содержится в документе: **Процессор NeuroMatrix®NM6403. Руководство пользователя.**

Поле RDY1

Поле RDY1 определяет условие окончания цикла обращения к памяти банка 1:

- 0 - только по внутреннему счетчику;
- 1 - по внутреннему счетчику с учетом внешнего сигнала готовности.

Биты, которые оно занимает в регистре gmicr, отмечены серым:

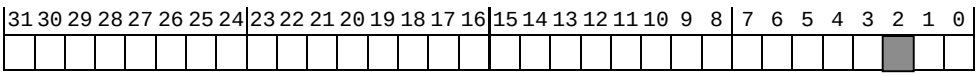


Поле RDY0

Поле RDY0 определяет условие окончания цикла обращения к памяти банка 0:

- 0 - только по внутреннему счетчику;
- 1 - по внутреннему счетчику с учетом внешнего сигнала готовности.

Биты, которые оно занимает в регистре gmicr, отмечены серым:



Примечание

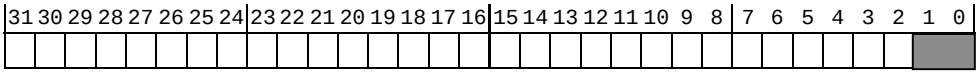
Если в поле SHMEM (см. ниже) указан один из типов многопроцессорной конфигурации, то значения полей RDY1 и RDY0 не влияют на доступ к памяти и могут иметь произвольное значение.

Поле SHMEM

Поле SHMEM определяет конфигурацию глобальной шины при работе с разделяемой памятью:

- 00 - многопроцессорная конфигурация первого типа (банк 0 - "общий", банк 1 - "общий");
- 01 - многопроцессорная конфигурация второго типа (банк 0 - "свой", банк 1 - "общий");
- 10 - многопроцессорная конфигурация второго типа (банк 0 - "свой", банк 1 - "чужой");
- 11 - однопроцессорная конфигурация.

Биты, которые оно занимает в регистре gmicr, отмечены серым:



Более подробная информация о конфигурациях разделяемой памяти содержится в документе: **Процессор NeuroMatrix®NM6403. Руководство пользователя.**

3.2.2 Регистры управления коммуникационными портами (ica, icc) (oca, occ)

Общие сведения о коммуникационных портах процессора 3-13
Состояние комм. портов после сброса 3-14
Формат принимаемых/передаваемых данных 3-14
Приём данных через коммуникационный порт 3-15
Передача данных через коммуникационный порт 3-16
Ожидание окончания приема/передачи по коммуникационному порту 3-17
Одновременная передача/прием данных через коммуникационный порт 3-18
Блокировка приёма/передачи через коммуникационный порт 3-18

Общие сведения о коммуникационных портах процессора

Процессор NM6403 имеет два коммуникационных порта, совместимых по сигналам с коммуникационными портами процессора TMS320C40. Это означает, что посредством этих портов процессор может обмениваться информацией с другим процессором NM6403, либо с TMS320C40.

Для обслуживания коммуникационных портов в процессоре NM6403 имеется два DMA-сопроцессора. Каждый сопроцессор обслуживает свой порт как на прием, так и на передачу. Сопроцессоры работают независимо как друг от друга, так и от самого процессора, то есть каждый может выполнять свою работу параллельно с остальными. Например, возможно так организовать пользовательскую программу, чтобы процессор одновременно принимал одну порцию данных через нулевой коммуникационный порт, производил вычисления над второй порцией, а третью

посылал бы через первый коммуникационный порт в другой процессор.

Оба коммуникационных порта являются абсолютно равноправными. Единственное их отличие в том начальном состоянии, в которое они устанавливаются после сброса. Комм. порт 0 по сигналу RESET переходит в состояние передачи данных, а первый комм. порт в состояние приема.

В дальнейшем каждый из комм. портов может быть переведен в любое из двух состояний. Перевод комм. порта в состояние приема/передачи осуществляется при помощи регистров адреса `ica0`, `ica1`, `oca0`, `oca1` и регистров счетчиков `icc0`, `icc1`, `occ0`, `occ1` каналов ввода/вывода.

Регистры с 0 на конце управляют нулевым комм. портом, регистры с 1 первым. Регистры, названия которых начинаются с `i`, используются для управления приемом данных, а начинающиеся с `o` для передачи.

Состояние комм. портов после сброса

После сброса комм. порт 0 переходит в состояние передачи, а порт 1 в состояние приема. Данная информация используется при соединении процессора с другими процессорами или иными источниками/приемниками данных. При соединении двух процессоров через комм. порты следует убедиться, что в момент подачи питания комм. порт одного из процессоров находился в состоянии приема, а порт другого в режиме передачи.

Например, при соединении двух процессоров NM6403 через комм. порты необходимо проследить, чтобы нулевой порт одного был соединен с первым портом второго.

Внимание!

*Необходимо помнить о том, что **соединение одноименных портов процессоров** (оба на прием или на передачу) в худшем случае **может привести к выходу системы из строя в момент подачи питания**.*

Формат принимаемых/передаваемых данных

Процессор NM6403 позволяет принимать/передавать через комм. порты **только 64-х разрядные** данные. Для того, чтобы добиться совместимости с TMS320C40, необходимо, чтобы работающая там программа обменивалась с нейропроцессором блоками, состоящими из четного количества 32-х разрядных слов.

Передача/прием слова данных в процессоре NM6403 ведется в порядке от младших битов к старшим, то есть сначала посылаются/принимаются младшие биты слова, а затем старшие.

Таким образом при приеме данных в TMS320C40, посланных из

NM6403, сначала приходит младшее 32-х разрядное слово, а затем старшее, и наоборот, первое 32-х разрядное слово, пришедшее из TMS320C40 становится младшей частью 64-х разрядного слова в NM6403.

Приём данных через коммуникационный порт

DMA-сопроцессор в режиме приема данных управляется регистрами `ica0` и `icc0` (здесь и далее все рассуждения приведены для регистров нулевого комм. порта, однако всё сказанное справедливо также для регистров первого).

В регистр `ica0` заносится адрес в памяти процессора, начиная с которого будет располагаться массив принимаемых данных.

В регистр `icc0` заносится количество принимаемых 64-х разрядных слов (со знаком '-'). Запись в регистр `icc0` инициирует перевод комм. порта в режим приема данных. Например, инициализация комм. порта на прием массива данных может выглядеть следующим образом:

```
nobits "data"
    InputBuff: long[100];
end "data";
begin "text"
    ...
    ica0 = InputBuff;  // указатель на буфер приема.
    icc0 = -100;       // начало приема 100 длинных слов.
    ...
end "text";
```

Счётчик `icc0` после приема каждого слова увеличивается на 1. Приём данных будет завершён, когда значение `icc0` достигнет нуля. При этом не имеет значения, что той стороны канала ещё могли остаться не переданные данные, DMA-сопроцессор прекратит приём, как только счётчик достигнет нуля.

Существует два способа инициализации комм. порта на прием:

- программная инициализация. Программист пишет программу инициализации, в которой напрямую задается значение регистров `ica0` и `icc0` (см. пример выше);
- внешняя инициализация. В этом случае DMA-сопроцессор, приняв первое 64-х разрядное слово, младшие 32 бита записывает в `ica0`, рассматривая их как адрес в памяти, а старшие в `icc0`, трактуя их как количество принимаемых слов.

Выбор режима инициализации нулевого комм. порта на прием осуществляется при помощи бита `CP0I` поля `CP0 control` регистра `pswr` (бит 22). Выбор режима инициализации первого комм. порта на прием осуществляется при помощи бита `CP1I` поля `CP1 control` (бит 25). См. параграф 3.2.6. Регистр стр. 3-25.

После системного сброса все поля регистра `pswr` равны нулю. Если используемый загрузчик не изменяет значение этих битов, то

инициализация комм. портов на прием должна осуществляться программно.

Пример внешней инициализации комм. порта на прием данных:

```
begin "text"
    ...
    pswr set 400000h; // установка 22-го бита в 1.
    ...
end "text";
```

После того, как бит, определяющий тип инициализации комм. порта установлен в единицу, DMA-сопроцессор готов к приему данных. Как только первые данные с того конца канала посланы, начнется их прием, и первое слово используется для инициализации DMA-сопроцессора, сообщая ему адрес и размер принимаемого массива.

Передача данных через коммуникационный порт

DMA-сопроцессор в режиме передачи данных управляется регистрами `oca0` и `occ0` (здесь и далее все рассуждения приведены для регистров нулевого комм. порта, однако всё сказанное справедливо также для регистров первого).

В регистр `oca0` заносится адрес в памяти процессора, начиная с которого располагается массив передаваемых данных.

В регистр `icc0` заносится количество передаваемых 64-х разрядных слов (со знаком '-'). Запись в регистр `icc0` инициирует перевод комм. порта в режим передачи данных. Например, инициализация комм. порта на передачу массива данных может выглядеть следующим образом:

```
nobits "data"
    OutputBuff: long[100];
end "data";
begin "text"
    ...
    oca0 = OutputBuff; // указатель на передаваемый
                       // массив.
    occ0 = -100;       // момент начала передачи
    ...                // 100 длинных слов.
end "text";
```

Счётчик `occ0` после передачи каждого слова увеличивается на 1. Передача данных будет завершёна, когда значение `occ0` достигнет нуля. При этом не имеет значения, что той стороны канала комм. порт готов принимать ещё, DMA-сопроцессор прекратит передачу, как только счётчик достигнет нуля.

В случае, если планируется передавать данные в другой процессор, соответствующий DMA-сопроцессор которого установлен в режим внешней инициализации, необходимо предусмотреть, чтобы первое слово передаваемого массива содержало его адрес и размер. Под адресом понимается адрес в адресной сетке принимающего процессора. Размер массива дается без учета первого слова.

Пример:

```
nobits "data"
    OutputBuff: long[101];
end "data";
begin "text"
    ...
    ar0 = OutputBuff;
    ar4 = 80000000h; // адрес массива в
                     // принимающем процессоре.
    gr4 = -100;      // размер массива.
    [ar0] = ar4, gr4; // запись в нулевое слово
                      // передаваемого массива.
    oca0 = ar0;  // указатель на передаваемый массив.
    occ0 = -101; // момент начала передачи
    ...          // 101 длинных слов.
end "text";
```

Ожидание окончания приема/передачи по коммуникационному порту

Существуют два критерия, свидетельствующих об окончании приема/передачи данных через комм. порт.

Первый критерий - прерывание, порождаемое по окончании приема/передачи. Дает стопроцентную гарантию, что процесс обмена данными завершен. Однако использование прерывания кажется громоздким в большинстве случаев.

Второй критерий - нулевое значение в регистре счетчика. Удобное средство проверки окончания передачи. Используется в большинстве случаев.

Пример:

```
begin "text"
    oca0 = OutputBuff;
    occ0 = -100;
    ...
<Loop>              // метка начала цикла ожидания.
    gr0 = occ0;     // считывается значение счетчика.
    gr0;            // устанавливаются флаги
                    // условного перехода.
    if < goto Loop; // цикл повторяется пока счетчик
                    // occ0 не достигнет нуля.
    ...
end "text";
```

Примечание

В случае внешней инициализации опрос регистра счетчика не может использоваться как способ ожидания окончания приема. Внешняя инициализация - процесс асинхронный, поэтому принимающий процессор не знает, когда она начинается. Проверка счетчика на ноль/не ноль может привести к ошибке, ведь в момент проверки неизвестно, закончился прием данных или ещё не начинался. И в том и в другом случае значение счетчика равно нулю. При внешней инициализации единственным способом проверки

окончания приема является прерывание.

Одновременная передача/прием данных через коммуникационный порт

DMA-сопроцессор имеет две пары регистров. Одна используется для управления приемом, другая передачей данных. Эти пары никак не связаны меду собой, поэтому ничто не мешает одновременно инициализировать порт на прием и на передачу.

Одновременная инициализация комм. порта на прием и передачу возможна, однако никакого выигрыша в скорости обмена данными она не принесет. Такая возможность введена для сохранения функциональности при работе с TMS320C40.

В случае одновременной инициализации комм. порта на прием и передачу выбор останется за другой стороной канала. Если тот комм. порт готов принимать данные, то начнется передача и наоборот. Пока передача не завершится, прием данных не начнется.

Блокировка приёма/передачи через коммуникационный порт

Процессор позволяет заблокировать/разблокировать прием/передачу данных через комм. порт. Блокирование может быть установлено как до момента обмена данными, так и в процессе обмена. При этом соответствующий комм. порт будет находиться в "замороженном" состоянии до тех пор, пока бит блокировки в регистре `pswr` не будет сброшен.

Блокировка приема по комм. порту 0 осуществляется путем записи единицы в бит `ICH0` поля `CP0 control` регистра `pswr` (бит 21).

Блокировка приема по комм. порту 1 осуществляется путем записи единицы в бит `ICH1` поля `CP1 control` регистра `pswr` (бит 24).

Блокировка передачи по комм. порту 0 осуществляется путем записи единицы в бит `OCH0` поля `CP0 control` регистра `pswr` (бит 20).

Блокировка передачи по комм. порту 1 осуществляется путем записи единицы в бит `OCH1` поля `CP1 control` регистра `pswr` (бит 23).

Пример установки режима блокировки приема данных через комм. порт 1:

```
begin "text"
    ...
    pswr set 1000000h; // установка 24-го бита в 1.
    ...
end "text";
```

Разблокирование комм. порта осуществляется путем сброса соответствующего бита в 0. Пример снятия режима блокировки приема данных через комм. порт 1:

```
begin "text"
    ...
    pswr clear 1000000h; // сброс 24-го бита в 0.
```

```
    ...
end "text";
```

3.2.3 Регистр intr

Поле BS 3-20
Поле DMAR 3-20
Поле PS 3-21
Поле AFIFO_VAL 3-21
Поле RAM_VAL 3-22
Поле VPF 3-22
Поле INTEREQ 3-23

Регистр `intr` имеет разрядность 32 бита. Он доступен из ассемблера только по чтению и используется как регистр запросов на прерывание и операции прямого доступа к памяти.

Он недоступен по записи, то есть не может быть изменен. Попытка записи в регистр `intr` из ассемблера приведет к синтаксической ошибке.

Текущее значение `intr` описывает состояние процессора. Ниже в таблице Табл. 3-10 приведено наименование его полей, а также их начальные значения, отражающие состояние регистра после системного сброса. Знаком 'X' показаны неопределенные значения, которые при сбросе могут быть как нулем, так и единицей.

Табл. 3-10 Регистр запросов на прерывание и прямой доступ к памяти INTR.

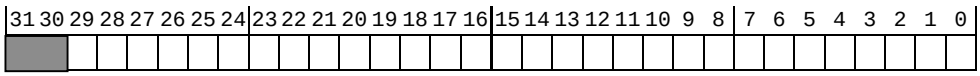
Бит	Поля	Имена битов	Начальное Значение
31 30	BS	GBS LBS	0 1
29 28 27 26	DMAR	IC1DR IC0DR OC1DR OC0DR	0 0 1 1
25 24	PS	CP1S CP0S	1 0
23 24 21 20 19 18	AFIFO_VAL	EMPTY - - - - -	0 X X X X X X
17 16 15 14 13	RAM_VAL	- - - - -	X X X X X
12 11 10 9	VPF	EMPTA FULLA EMPTW FULLW	1 0 1 0

8	INTREQ	T0R	0
7		SPR	0
6		VPR	0
5		INR	0
4		IC1R	0
3		IC0R	0
2		OC1R	0
1		OC0R	0
0		T1R	0

Далее приводится более подробное разъяснение значения каждого поля регистра `intr`.

Поле BS

Поле `BS` содержит информацию о том, принадлежат ли в данный момент процессору нулевые банки памяти на локальной и глобальной шинах. Биты, которые оно занимает в регистре `intr`, отмечены серым:

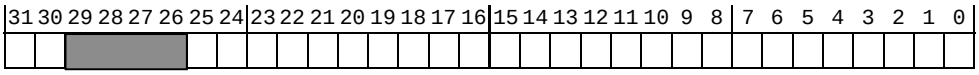


Далее следует описание значений битов поля:

Бит	Имя	Описание
31	`GBS`	состояние глобальной шины: • 0 - шина принадлежит процессору при необходимости обращения по ней в нулевой банк памяти; • 1 - шина процессору в данный момент не принадлежит.
30	`LBS`	состояние локальной шины: • 0 - шина принадлежит процессору при необходимости обращения по ней в нулевой банк памяти; • 1 - шина процессору в данный момент не принадлежит.

Поле DMAR

Поле `DMAR` содержит запросы на прямой доступ к памяти от коммуникационных портов процессора. При появлении запроса на доступ соответствующий бит взводится в единичное положение, при снятии запроса, а также в его отсутствие бит равен нулю. Биты, которые оно занимает в регистре `intr`, отмечены серым:



Далее следует описание значений битов поля:

Бит	Имя	Описание
29	IC1DR	запрос на прямой доступ к памяти (ПДП) от канала приема данных порта 1.
28	IC0DR	запрос на ПДП от канала приема данных порта 0.
27	OC1DR	запрос на ПДП от канала передачи данных порта 1.
26	OC0DR	запрос на ПДП от канала передачи данных порта 0.

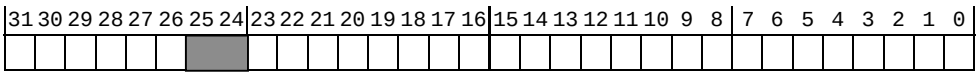
Примечание

При возникновении запроса на ПДП от каналов приема данных в соответствующих разрядах 29 и 28 регистра `intr` *автоматически устанавливается единица, которая сбрасывается, когда доступ к памяти разрешен.*

При возникновении запроса на ПДП от каналов выдачи данных в соответствующих разрядах 27 и 26 регистра `intr` *автоматически устанавливается ноль, который меняется на единицу, когда доступ к памяти разрешен.*

Поле PS

Поле PS содержит информацию о текущем направлении передачи данных через коммуникационные порты процессора NM6403. Биты, которые оно занимает в регистре `intr`, отмечены серым:



Далее следует описание значений битов поля:

Бит	Имя	Описание
25	CP1S	текущее направление передачи данных через порт 1: • 0 - порт находится в режиме выдачи данных; • 1 - порт находится в режиме приема данных.
24	CP0S	текущее направление передачи данных через порт 0: • 0 - порт находится в режиме выдачи данных; • 1 - порт находится в режиме приема данных.

Поле AFIFO_VAL

Поле AFIFO_VAL содержит информацию о том, сколько 64-х разрядных слов должно быть в AFIFO после того, как закончится выполнение всех векторных команд, на попавших на данный

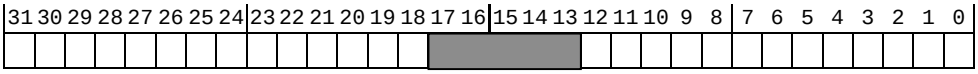
момент в конвейер процессора. Биты, которые оно занимает в регистре `intr`, отмечены серым:

Далее следует описание значений битов поля:

Бит	Имя	Описание
23	EMPTY	признак того, что AFIFO не пусто: • 0 - AFIFO пусто; • 1 - AFIFO не пусто, и число записанных в нем слов определяется разрядами 22 ... 18.
22 21 20 19 18		число записанных в AFIFO слов в диапазоне от 1 до 32. При этом: 00000 соответствует одному 64-х разрядному слову; 00001 соответствует двум 64-х разрядным словам; ..... 11111 соответствует тридцати двум словам.

Поле RAM_VAL

Поле RAM_VAL содержит информацию о том, сколько 64-х разрядных слов должно быть в RAM (внутреннее ОЗУ процессора) после того, как закончится выполнение всех векторных команд, на попавших на данный момент в конвейер процессора. Биты, которые оно занимает в регистре `intr`, отмечены серым:



Далее следует описание значений битов поля:

Бит	Описание
17 16 15 14 13	число слов, находящихся в RAM, в диапазоне от 1 до 32. При этом: 00000 соответствует одному 64-х разрядному слову; 00001 соответствует двум 64-х разрядным словам; ..... 11111 соответствует тридцати двум словам.

Поле VPF

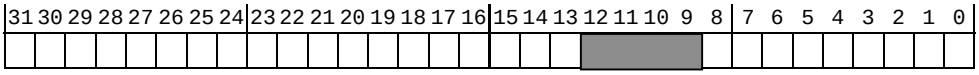
Поле VPF содержит информацию о степени наполнения буферов WFIFO и AFIFO векторного процессора. Биты, которые оно занимает в регистре `intr`, отмечены серым:

Далее следует описание значений битов поля:

Бит	Имя	Описание
12	EMPTA	флаг наличия информации в буфере AFIFO: • 0 - AFIFO содержит данные; • 1 - AFIFO пусто.
11	FULLA	флаг степени наполнения буфера AFIFO: • 0 - AFIFO заполнено не полностью; • 1 - AFIFO заполнено полностью.
10	EMPTW	флаг наличия информации в буфере WFIFO: • 0 - WFIFO содержит данные; • 1 - WFIFO пусто.
9	FULLW	флаг степени наполнения буфера WFIFO: • 0 - WFIFO заполнено не полностью; • 1 - WFIFO заполнено полностью.

Поле INTEREQ

Поле INTEREQ содержит запросы на прерывание. Биты, которые оно занимает в регистре intr, отмечены серым:



Далее следует описание значений битов поля:

Бит	Имя	Описание
8	T0R	запрос на прерывание от таймера t0.
7	SPR	запрос на прерывание по переполнению при выполнении арифметической операции, заданной скалярной командой.
6	VPR	запрос на прерывание по запрещенной векторной команде.
5	INR	запрос на внешнее прерывание.
4	IC1R	запрос на прерывание по завершению приема данных по коммуникационному порту 1.
3	IC0R	запрос на прерывание по завершению приема данных по коммуникационному порту 0.
2	OC1R	запрос на прерывание по завершению передачи данных по коммуникационному порту 1.
1	OC0R	запрос на прерывание по завершению передачи данных по коммуникационному порту 0.

0	T1R	запрос на прерывание от таймера t1.

Примечание

При возникновении запроса на прерывание в соответствующем разряде регистра intr автоматически устанавливается единица, которая сбрасывается, когда данный запрос начинает обслуживаться.

3.2.4 Регистр lmicr

Регистр lmicr имеет разрядность 32 бита. С его помощью осуществляется управление доступом к внешней памяти через локальную шину. Путем конфигурирования данного регистра программист может открыть доступ по локальной шине к одному или двум банкам памяти, различающимся типом, страничной организацией и динамическими параметрами.

Регистр lmicr доступен как для чтения, так и для записи.

Он в обязательном порядке должен быть установлен при загрузке прикладной исполняемой программы в процессор. Обычно его установкой занимается загрузчик, входящий в состав библиотеки загрузки и обмена.

Примечание

Пользователь вправе самостоятельно изменять содержимое регистра lmicr. Однако при этом необходимо соблюдать предосторожность, поскольку неправильно сконфигурированный регистр не позволит процессору корректно осуществлять доступ к локальной памяти, что может привести к зависанию программы.

В Табл. 3-3 содержится информация о полях регистра gmicr. Регистр lmicr имеет абсолютно такую же структуру. Все его поля соответствуют тем, что приведены в данной таблице.

Поскольку локальная шина имеет свой диапазон адресов, ее разбиение на банки определяется Табл. 3-11.

Табл. 3-11 Разбиение адресного пространства локальной шины на банки 0 и 1, задаваемое полем BOUND регистра lmicr.

BOUND	Размер банка 0 (64 бита)	Адресное пространство банка 0	Размер банка 1 (64 бита)	Адресное пространство банка 1
0000	2^{15} = 32K	00000000 - 0000FFFF	32K	00010000 - 7FFFFFFF
0001	2^{16} = 64K	00000000 - 0001FFFF	64K	00020000 - 7FFFFFFF
0010	2^{17} = 128K	00000000 - 0003FFFF	128K	00040000 - 7FFFFFFF
0011	2^{18} = 256K	00000000 - 0007FFFF	256K	00080000 - 7FFFFFFF
0100	2^{19} = 512K	00000000 - 000FFFFF	512K	00100000 - 7FFFFFFF

0101	2^{20} = 1M	00000000 - 001FFFFF	1M	00200000 - 7FFFFFFF
0110	2^{21} = 2M	00000000 - 003FFFFF	2M	00400000 - 7FFFFFFF
0111	2^{22} = 4M	00000000 - 007FFFFF	4M	00800000 - 7FFFFFFF
1000	2^{23} = 8M	00000000 - 00FFFFFF	8M	01000000 - 7FFFFFFF
1001	2^{24} = 16M	00000000 - 01FFFFFF	16M	02000000 - 7FFFFFFF
1010	2^{25} = 32M	00000000 - 03FFFFFF	32M	04000000 - 7FFFFFFF
1011	2^{26} = 64M	00000000 - 07FFFFFF	64M	08000000 - 7FFFFFFF
1100	2^{27} = 128M	00000000 - 0FFFFFFF	128M	10000000 - 7FFFFFFF
1101	2^{28} = 256M	00000000 - 1FFFFFFF	256M	20000000 - 7FFFFFFF
1110	2^{29} = 512M	00000000 - 3FFFFFFF	512M	40000000 - 7FFFFFFF
1111	2^{30} = 1G	00000000 - 7FFFFFFF	0	-

3.2.5 Регистр pc

Регистр pc используется процессором как указатель на адрес в памяти процессора, из которого будет подгружена следующая команда выполняемой программы.

Процессор всегда подгружает из памяти 64-х разрядное слово команд. В нем может содержаться одна длинная команда или две короткие. Длинная команда не может начинаться с нечетного адреса.

На изменение значения регистра pc оказывают серьезное влияние буфер подчитываемых команд и буфер исполняемых команд. Это влияние выражается в том, что не всегда известно, каково же в данный момент быть значение регистра, равно ли оно текущему адресу плюс два или плюс четыре.

Из-за плохой предсказуемости значения регистра pc в каждый конкретный момент времени адресация по счетчику команд ассемблером не поддерживается.

3.2.6 Регистр pswr

Поле BC 3-26
Поле TIMER pin control 3-27
Поле CP1 control 3-28
Поле CP0 control 3-29
Поле T0C 3-30
Поле T1C 3-31
Поле FCL 3-31
Поле INTERRUPT MASKS 3-32
Поле FLAGS 3-33

Регистр pswr (слово состояния) входит в блок регистров управления состоянием процессора. После системного сброса он переходит в

нулевое состояние. Он состоит из нескольких полей, каждое из которых доступно по чтению и записи.

В таблице Табл. 3-12 представлено расположение полей в регистре `pswr`.

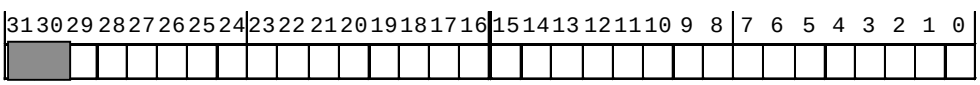
Табл. 3-12 Регистр слова состояния процессора pswr (32 бита).

Бит	Поля	Имена битов
31 30	BC	GBRE LBRE
29 28 27 26	TIMER pin control	TEN TM2 TM1 TM0
25 24 23	CP1 control	CP1I ICH1 OCH1
22 21 20	CP0 control	CP0I ICH0 OCH0
19 18	T0C	TM0 TE0
17 16	T1C	TM1 TE1
15 14	FCL	WFCL AFCL
13 12 11 10 9 8 7 6 5 4	INTERRUPT MASKS	T0M SPM VPM INTM IC1M IC0M OC1M OC0M T1M STM
3 2 1 0	FLAGS	N Z V C

Далее приводится подробное разъяснение значения каждого поля регистра `pswr`.

Поле BC

Поле `BC` задает режим доступа процессора к памяти по внешним шинам при работе с общей памятью. Общей называется память, доступ к которой имеют два и более процессоров. Биты, которые поле `BC` занимает в регистре `pswr`, отмечены серым:

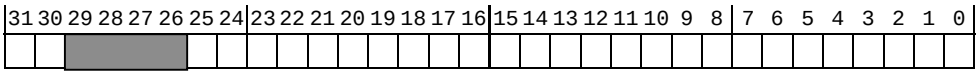


Далее следует описание значений битов поля:

Бит	Имя	Описание
31	GBRE	разрешение/запрещение передачи контроля над глобальной шиной внешнему устройству при запросе от него: • 0 - контроль не передается; • 1 - контроль передается.
30	LBRE	разрешение/запрещение передачи контроля над локальной шиной внешнему устройству при запросе от него: • 0 - контроль не передается; • 1 - контроль передается.

Поле TIMER pin control

Поле `TIMER pin control` определяет режим работы внешнего вывода TIMER. Биты, которые оно занимает в регистре `pswr`, отмечены серым:



Далее следует описание значений битов поля:

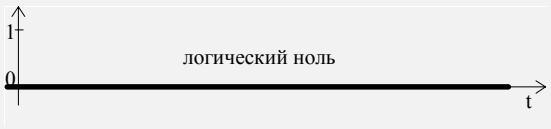
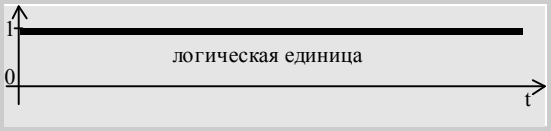
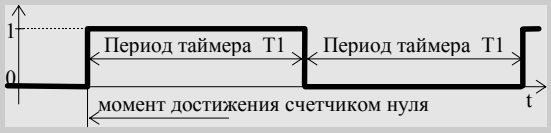
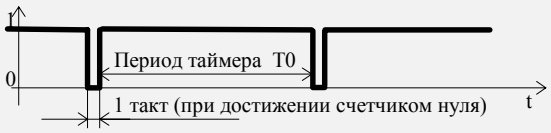
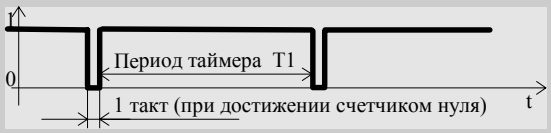
Бит	Имя	Описание
29	TEN	разрешение/запрещение выдачи сигнала на внешний вывод (ножку) TIMER процессора. • 0 - выдача не разрешается; • 1 - выдача разрешается.
28	TM2	поле управления внешним выводом TIMER
27	TM1	поле управления внешним выводом TIMER
26	TM0	поле управления внешним выводом TIMER

Если бит `TEN = 0`, то никакие комбинации битов `TM2`, `TM1` и `TM0` не приведут к изменению состояния ножки `TIMER`.

В случае `TEN = 1` поля `TM2`, `TM1` и `TM0` используются для определения того, какой именно сигнал выдается на ножку `TIMER`. Зависимость типа сигнала от состояния данных полей приведена в Табл. 3-13.

Кроме случаев "логический ноль" и "логическая единица", когда ножка `TIMER` установлена в соответствующее положение, и не меняется с течением времени, все остальные наборы сигналов определяются работой внутренних таймеров T0 и T1 процессора NM6403.

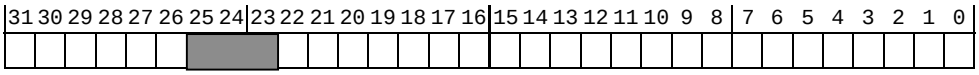
Табл. 3-13 Управление внешним выводом TIMER.

TM2	TM1	TM0	Состояние вывода TIMER
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

В случае, если таймер T0 или T1 работает в циклическом режиме, то приведенные временные диаграммы также будут иметь периодический характер. Если же таймер запрограммирован на однократный отсчет, то по достижению счетчиком нуля в зависимости от установленных `TM2`, `TM1` и `TM0` возникнет либо однократный пик с последующим возвращением в основное состояние (как в последних четырех случаях), либо смена состояния на противоположное (как в третьем и четвертом случаях).

Поле CP1 control

Поле `CP1 control` определяет режимы работы первого коммуникационного порта процессора. Биты, которые оно занимает в регистре `pswr`, отмечены серым:

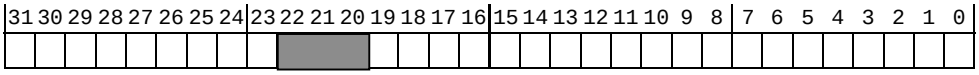


Далее следует описание значений битов поля:

Бит	Имя	Описание
25	CP1I	бит внешней инициализации канала ввода порта 1: • 0 - инициализация канала ввода порта 1 осуществляется программно. Это значит, что адрес и размер принимаемого массива задается перед тем, как начать прием данных. При этом, DMA сопроцессор уже до начала приема знает, сколько слов принять и куда разместить; • 1 - инициализация канала ввода порта 1 осуществляется по первому принятому слову. То есть, DMA сопроцессор сначала принимает первое слово (64 бита) и рассматривает его, как пару 32-разрядных слов, одно из которых содержит адрес, а второе размер массива.
24	ICH1	бит разрешения/запрещения приема данных через порт 1: • 0 - разрешение приема данных; • 1 - запрещение приема данных.
23	OCH1	бит разрешения/запрещения передачи данных через порт 1: • 0 - разрешение передачи данных; • 1 - запрещение передачи данных.

Поле CP0 control

Поле `CP0 control` определяет режимы работы первого коммуникационного порта процессора. Биты, которые оно занимает в регистре `pswr`, отмечены серым:



Далее следует описание значений битов поля:

Бит	Имя	Описание
22	CP0I	бит внешней инициализации канала ввода порта 0: • 0 - инициализация канала ввода порта 0 осуществляется программно. Это значит, что адрес и размер принимаемого массива задается перед тем, как начать прием данных. При этом, DMA сопроцессор уже до начала приема знает,

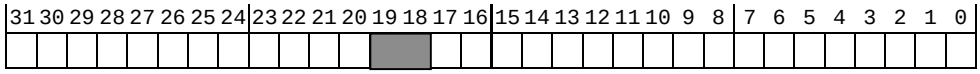
		сколько слов принять и куда разместить; • 1 - инициализация канала ввода порта 0 осуществляется по первому принятому слову. То есть, DMA сопроцессор сначала принимает первое слово (64 бита) и рассматривает его, как пару 32-разрядных слов, одно из которых содержит адрес, а второе размер массива.
21	ICH0	бит разрешения/запрещения приема данных через порт 0: • 0 - разрешение приема данных; • 1 - запрещение приема данных.
20	OCH0	бит разрешения/запрещения передачи данных через порт 0: • 0 - разрешение передачи данных; • 1 - запрещение передачи данных.

Примечание

Изменение полей CP0 и CP1 control напрямую в регистре `pswr` *возможно только путем использования команд* `pswr set` *или* `pswr clear` *(см. параграф* 5.1.8 Команды модификации регистра pswr *на стр. 5-15).*

Поле T0C

Поле `T0C` управляет работой таймера T0. Биты, которые оно занимает в регистре `pswr`, отмечены серым:



Далее следует описание значений битов поля:

Бит	Имя	Описание
19	TM0	определяет режим работы таймера T0.
18	TE0	запрещает/разрешает работу таймера T0.

Различные комбинации битов данного поля позволяют задавать следующие варианты работы таймера:

Табл. 3-14 Режимы работы таймера T0.

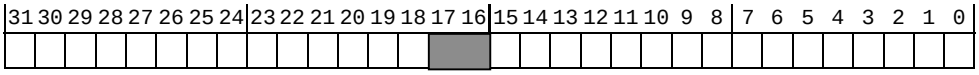
TM0	TE0	Описание режима работы таймера
0	0	Работа таймера T0 остановлена независимо от значения счетчика тактов в регистре `t0`.

0	1	Таймер, отсчитав положенное число тактов, заданных в регистре t0, останавливается. При обнулении счетчика возникает запрос на прерывание.
1	x	Таймер работает в циклическом режиме. При обнулении счетчика таймера возникает запрос на прерывание, счетчик восстанавливается в исходное положение и начинается новый отсчет тактов. Знак 'x' обозначает, что значение бита не имеет значения.

Более подробное описание работы таймеров T0 и T1 см. 3.2.7. Регистры таймеров t0, t1 на стр. 3-33.

Поле T1C

Поле T1C управляет работой таймера T1. Биты, которые оно занимает в регистре pswr, отмечены серым:



Далее следует описание значений битов поля:

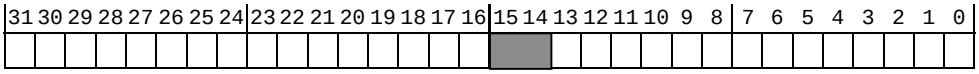
Бит	Имя	Описание
17	TM1	определяет режим работы таймера T1.
16	TE1	запрещает/разрешает работу таймера T1.

Комбинация битов TM1 и TE1 определяет работу таймера T1 в соответствии с Табл. 3-14.

Более подробное описание работы таймеров T0 и T1 см. 3.2.7. Регистры таймеров t0, t1 на стр. 3-33.

Поле FCL

Поле FCL управляет очисткой буферов AFIFO и WFIFO. Биты, которые оно занимает в регистре pswr, отмечены серым:



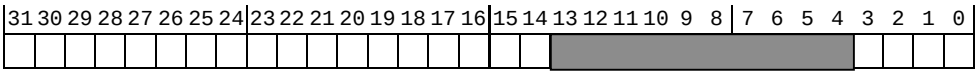
Далее следует описание значений битов поля:

Бит	Имя	Описание
15	WFCL	бит разрешения/запрещения очистки WFIFO: • 0 - очистка запрещена; • 1 - очистка разрешена, при этом на каждом такте процессор будет посылать сигнал очистки

		буфера, пока значение поля вновь не будет установлено в 0.
14	AFCL	бит разрешения/запрещения очистки AFIFO: • 0 - очистка запрещена; • 1 - очистка разрешена, при этом на каждом такте процессор будет посылать сигнал очистки буфера, пока значение поля вновь не будет установлено в 0.

Поле INTERRUPT MASKS

Поле INTERRUPT MASKS используется для маскирования или снятия маски с прерываний процессора. Бит, равный 1, соответствует установленной маске, бит 0 - снятой маске. Биты, которые оно занимает в регистре pswr, отмечены серым:



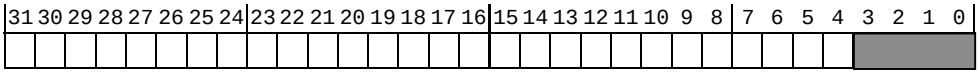
Далее следует описание значений битов поля:

Бит	Имя	Описание
13	TOM	маска прерывания от таймера t0. Запрос на прерывание выставляется, когда счетчик таймера достигает нулевого значения.
12	SPM	маска прерывания по переполнению при выполнении арифметической операции, заданной скалярной командой.
11	VPM	маска прерывания по запрещенной векторной команде.
10	INTM	маска прерывания по внешнему прерыванию (пользовательское прерывание).
9	IC1M	маска прерывания по завершению приема данных по коммуникационному порту 1. Запрос на прерывание выставляется, когда счетчик порта, отсчитывающий принятые слова, достигает нулевого значения.
8	IC0M	маска прерывания по завершению приема данных по коммуникационному порту 0. Запрос на прерывание выставляется, когда счетчик порта, отсчитывающий принятые слова, достигает нулевого значения.

7	OC1M	маска прерывания по завершению передачи данных по коммуникационному порту 1. Запрос на прерывание выставляется, когда счетчик порта, отсчитывающий посылаемые слова, достигает нулевого значения.
6	OC0M	маска прерывания по завершению передачи данных по коммуникационному порту 0. Запрос на прерывание выставляется, когда счетчик порта, отсчитывающий посылаемые слова, достигает нулевого значения.
5	T1M	маска прерывания от таймера t1. Запрос на прерывание выставляется, когда счетчик таймера достигает нулевого значения.
4	ST	маска пошагового прерывания.

Поле FLAGS

Поле FLAGS представляет собой набор флагов, выставляемых при различных арифметических операциях на скалярном процессоре. Биты, которые оно занимает в регистре pswr, отмечены серым:



Далее следует описание значений битов поля:

Бит	Имя	Описание
3	N	признак знака. Устанавливается в 1 всякий раз, когда результатом арифметической операции на скалярном процессоре является отрицательное число.
2	Z	признак нуля. Устанавливается в 1 всякий раз, когда результатом арифметической операции на скалярном процессоре является нулевое значение.
1	V	признак переполнения. Устанавливается в 1 всякий раз, когда в результате арифметической операции на скалярном процессоре возникло переполнение.
0	C	признак переноса. Устанавливается в 1 всякий раз, когда в результате арифметической операции на скалярном процессоре возник перенос.

3.2.7 Регистры таймеров t0, t1

Регистры таймеров t0 и t1 имеют разрядность 32 бита. Они доступны из ассемблера как по записи, так и по чтению.

Регистр `t0` считается системным и на него обычно возложены функции счетчика тактов для регенерации динамической памяти. В этом случае он не может использоваться пользовательской программой. Если же процессор работает только со статической памятью, пользовательская программа может использовать оба регистра.

Режим работы таймеров определяется полями регистра `pswr`. (см. параграф 3.2.6. Регистр на стр. 3-25). Регистры `t0` и `t1` управляются независимо, каждый посредством своих полей регистра `pswr`. В зависимости от значений, прописанных в этих полях, таймер может выполнять следующее:

- при проходе через 0 порождать прерывание;
- дойдя до нуля прекратить счет;
- дойдя до нуля инициализироваться значением переменной цикла и перейти отсчету нового цикла.

Таймер делает полный проход по разрядной сетке (от `0` до `0xFFFFFFFF`) за:

- 107,37 сек при частоте 40 МГц;
- 85,9 сек при частоте 50 МГц.

Ниже следует пример работы с регистром таймера:

```
begin "text"
     t1 = 0;   // установка счетчика таймера 1
               // в нулевое положение
     ...
     // выполняются вычисления.
     ...
     gr7 = t1; // в регистр gr7 записывается время
               // выполнения вычислений, измеренное в
               // процессорных тактах.
end "text";
```

Примечание

После записи значения в регистр t0(t1) проходит еще 6 тактов, прежде чем в таймер будет записано новое значение счетчика. Это объясняется тем, что прежде чем инструкция будет выполнена, она должна пройти через все стадии конвейера команд.

3.3 Векторные регистры

Процессор NM6403 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. 3-15 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 значений отдельно в младшую и старшую части регистра.	64 бита. Доступен только 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	32x64 бита.

	в процессе вычислений.	
`wfifo`	Регистр-контейнер для хранения весовых коэффициентов, которые затем загружаются в теневую матрицу ВП.	32x64 бита.

3.3.1 Регистры f1cr и f2cr

Разбиение данных на элементы при помощи f1cr (f2cr)..... 3-37
Использование f1cr(f2cr) в функции насыщения (арифметическая активация)... 3-37
Использование f1cr(f2cr) в пороговой функции (логическая активация) 3-39
Загрузка значений в регистр f1cr (f2cr) 3-40

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

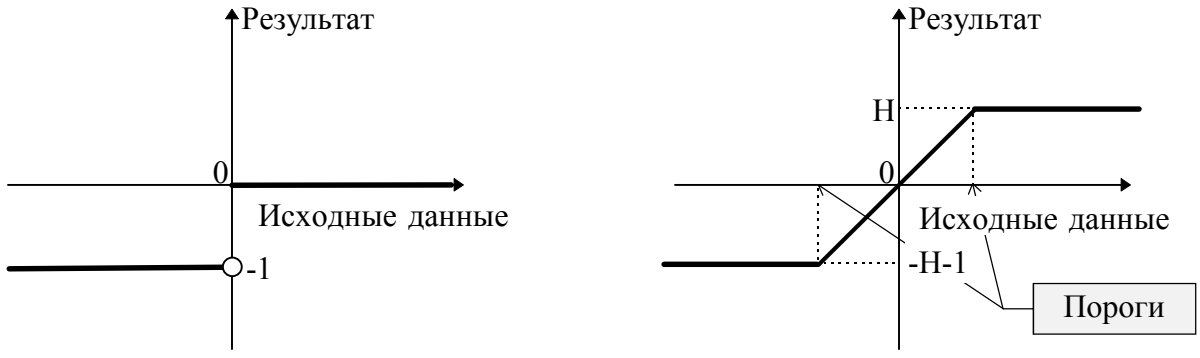
Регистры `f1cr` и `f2cr` недоступны по чтению. Попытка чтения из `f1cr` или `f2cr` воспринимается ассемблером, как синтаксическая ошибка.

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 параграфе 1.4.4 Обработка данных функцией активации на стр. 1-15.

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.

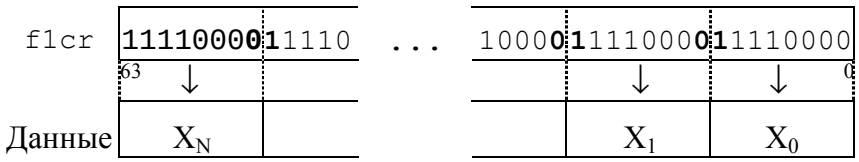
Разбиение на элементы, записанное в `f1cr` и `f2cr`, может не совпадать с тем, которое задается на входах **X** и **Y** регистрами `sb2` и `nb2` соответственно при операциях умножения с накоплением, или регистром `nb2` при выполнении вычислений на векторном АЛУ. Однако в большинстве случаев программист 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 бит, 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.

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.

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). Если значения битов не одинаковы, и старший бит равен 0, то значение заменяется на положительное пороговое, если старший бит равен 1, то 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|1`**`0000000`**`|1...`
поле данных: `.....0`**`0010110`**`...` ← *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|1`**`0000000`**`|1...`
поле данных: `.....1`**`0010110`**`...` ← *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:

```
f1crl = 80808080h;   // загрузка младшей части nb1.
f1crh = 40404040h;   // загрузка старшей части nb1.
```

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:

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

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"
    ...
```

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

В результате в регистр `f1cr` загружено `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. 3-16 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. 3-16 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	`4000020000100000hl`
16 бит	4	`80008000h`
10 бит	6	`0802008020080200hl`
8 бит	8	`80808080h`
4 бита	16	`88888888h`
2 бита	32	`AAAAAAAAh`

3.3.2 Регистр nb1(nb2)

Использование регистра nb1(nb2) 3-43
Загрузка значений в регистр nb1 3-44

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.

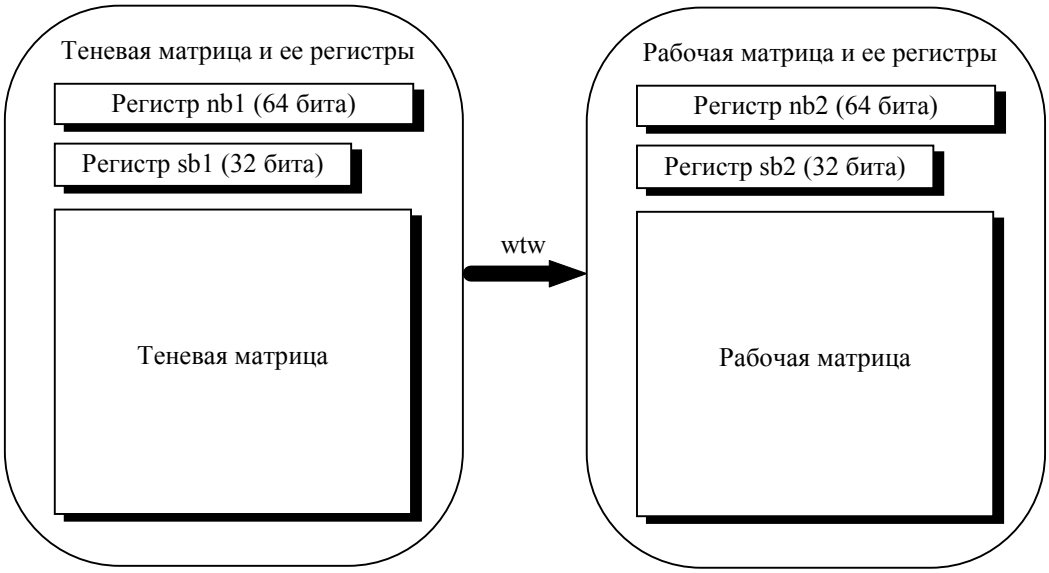
Прежде чем рассматривать его структуру и варианты использования, необходимо пояснить то, какое место занимает он в структуре векторного процессора NM6403.

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. 3.3.1, 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:

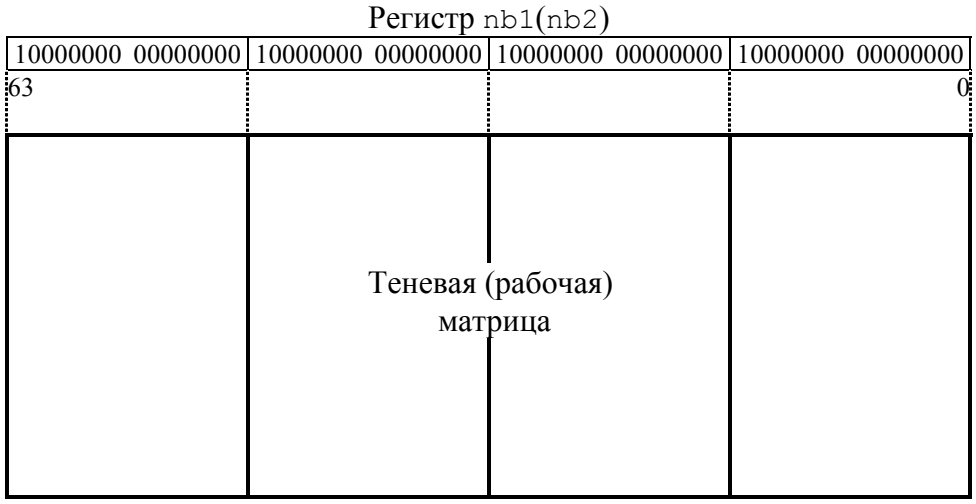
- сначала загрузить его в регистр nb1,
- при помощи команды wtw, осуществляющей копирование содержимого теневой матрицы в рабочую, переписать в nb2.

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:

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

begin "text"
```

```
    nb1 = [NB1];  // присвоение значения регистру nb1.
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:

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

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:

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

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"
```

```
    ...
    nb1 = [MyNB1]; // загрузка в nb1 константы из  памяти.
    ...
.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. 3-17 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. 3-17 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	`AAAAAAAAh`
1 бит	64	`FFFFFFFFh`

3.3.3 Регистр sb (sb1 и sb2)

Структура регистра sb 3-45
Использование регистра sb (sb1, sb2) 3-46
Загрузка значений в регистр sb 3-47

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

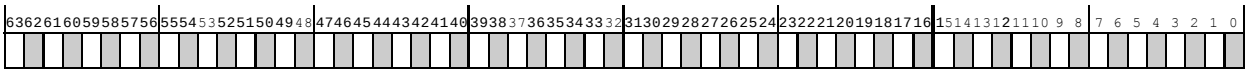
Регистр sb, являясь скорее логическим, чем реальным, представляет собой совокупность регистров sb1 и sb2. Регистры sb1 и sb2 имеют разрядность 32 бита каждый.

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. 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.

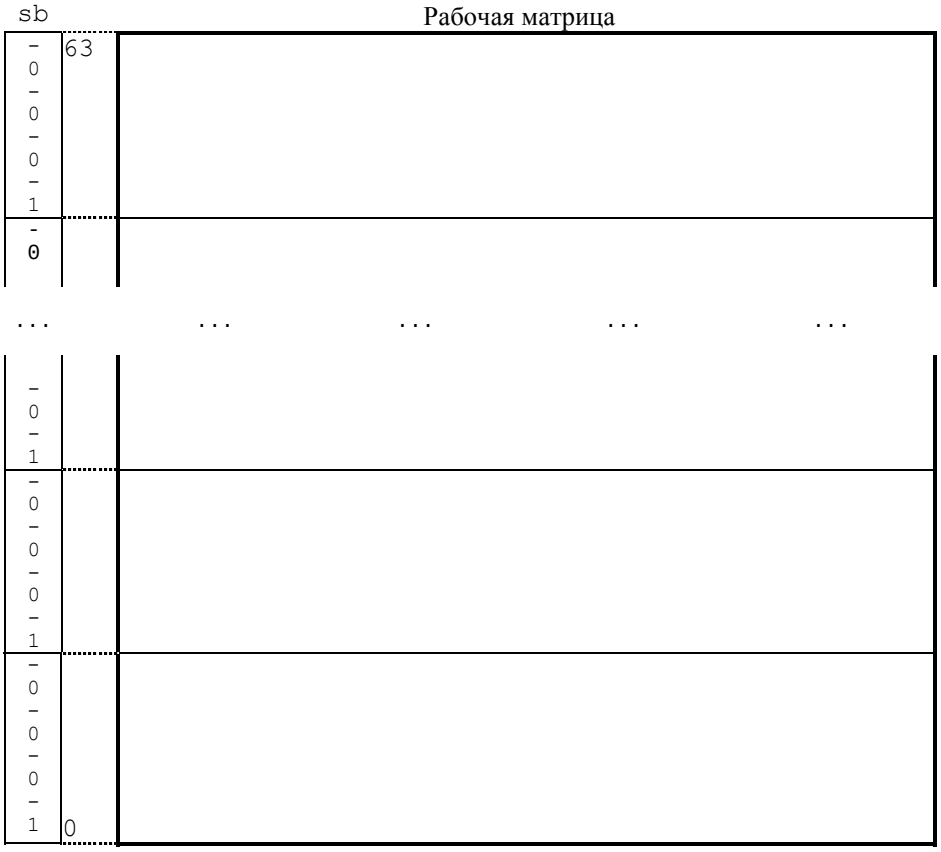
Регистр sb1(sb2) имеет разрядность 32 бита. Это означает, что каждый его бит управляет двумя битами данных, поступающих на вход **X** рабочей матрицы. В этом его отличие от регистра nb1(nb2), позволяющего управлять каждым битом входа **Y**.

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 битные 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 `sb1`;
- `sbl` - младшая часть регистра `sb1`.

Приведем пример загрузки регистра `nb1` по частям:

```
nb1l = 02020202h;   // загрузка младшей части sb.
nb1h = 05050505h;   // загрузка старшей части sb.
```

В результате в регистр `sb` 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:

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

При этом, процессор автоматически запишет эту константу в обе половины регистра, то есть, после выполнения команды в регистре `sb` будет 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"
    ...
    sb = [MySB];    // загрузка в sb константы из
                    // из памяти.
    ...
.end "Секция_кода";
```

В результате в регистр `sb` 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 Табл. 3-18 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. 3-18 Часто используемые константы разбиения для регистра sb.

Разрядность элемента	Количество элементов в слове	Значение константы в sb

64 бита	1	0
32 бита	2	2
20 бит	3	2000020000200002hl
16 бит	4	00020002h
10 бит	6	0208020080200802hl
8 бит	8	02020202h
4 бита	16	22222222h
2 бита	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 нечетные биты, соответствующие sb2, будут проигнорированы процессором. Таким образом, например, запись в sb значения 02020202h эквивалентна sb = 03030303h.

3.3.4 Регистр vr

Использование регистра vr 3-49
Загрузка значений в регистр vr 3-50

Регистр vr используется только в операции взвешенного суммирования на векторном процессоре, выступая в качестве операнда **Y** (см. Рис. 1-5). Он имеет разрядность 64 бита. Регистр vr доступен по записи из ассемблера.

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 разгрузить 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.

Пример загрузки `vr` из памяти:

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

Пример загрузки `vr` из пары регистров:

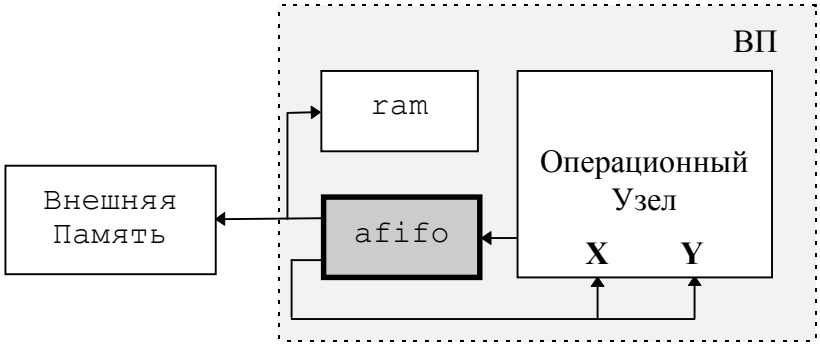
```
begin "text"
    ...
    ar0, gr0 = [InitVR];
    vr = ar0, gr0;     // короткая команда.
    ...
end "text";
```

3.3.5 Регистр-контейнер afifo

Загрузка данных в `afifo` 3-51
Выгрузка данных из `afifo` в память и в `ram` 3-52
Использование `afifo` в качестве входного буфера в векторных операциях 3-52
Одновременная выгрузка из `afifo` в память и передача его на вход ОУ 3-53
Использование `afifo` в операциях маскирования 3-54
Ошибки при работе с `afifo` 3-54

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

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 на стр. 3-19). 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 на стр. 3-25). 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.

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 напрямую. Например, следующая команда направляет 10 длинных слов входных данных в `afifo`:

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

В результате выполнения приведенной выше инструкции данные из памяти попадают в `afifo` напрямую. В ВП выполняется операция логического OR с нулевым вектором, которая не приводит к изменению данных. Теперь в `afifo` хранится 10 длинных слов, загруженных напрямую из памяти.

Как уже было сказано, невозможно заполнять `afifo` по частям, то есть первой командой загрузить туда 5 слов, в 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 `afifo`, в 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.

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.

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. параграф 1.4.3 Операция маскирования на стр. 1-14. 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 `nul` и порождает прерывание по неправильной векторной команде.

3.3.6 Логический регистр-контейнер data

Использование `data` для обработки данных на проходе 3-55
Использование `data` в операциях маскирования 3-55

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. параграф 1.4.3 Операция маскирования на стр. 1-14. 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;
```

3.3.7 Регистр-контейнер ram

Загрузка данных в ram 3-56
Использование ram в операциях на ОУ ВП 3-57
Использование ram в операциях маскирования 3-57
Ошибки при работе с ram 3-58

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 за содержимым ram может осуществляться программно путем анализа поля RAM_VAL регистра intr (см. 3.2.3. Регистр на стр. 3-19). Это поле сообщает, какое количество длинных слов в данный момент находится в ram.

Загрузка данных в ram

Данные загружаются в 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. параграф 1.4.3 Операция

маскирования на стр. 1-14. 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.

В случае возникновения ошибки при работе с ram по указанным выше причинам команда, в которой содержится ошибка, не отрабатывается. Процессор заменяет ее пустой командой nul и порождает прерывание по неправильной векторной команде.

3.3.8 Регистр-контейнер wfifo

Загрузка весовых коэффициентов в wfifo 3-59
Выгрузка весовых коэффициентов из wfifo 3-59
Одновременная загрузка и выгрузка данных из wfifo 3-59
Загрузка в wfifo нескольких матриц весовых коэффициентов 3-60
Ошибки при работе с wfifo 3-61

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.

Регистр wfifo доступен из ассемблера 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 отличии 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.

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 на стр. 3-19). Они отражают динамическое состояние afifo (то, как меняется его состояние в процессе выполнения векторной команды).

Содержимое wfifo может быть программно очищено путем установки в 1 бита WFCL (бит 15) поля FCL регистра pswr (см. 3.2.6. Регистр на стр. 3-25). 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.

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, в 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. 3.3.3. Регистр на стр. 3-45).

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"
...
sb = 02020202h; // в матрице 8 строк.
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`:

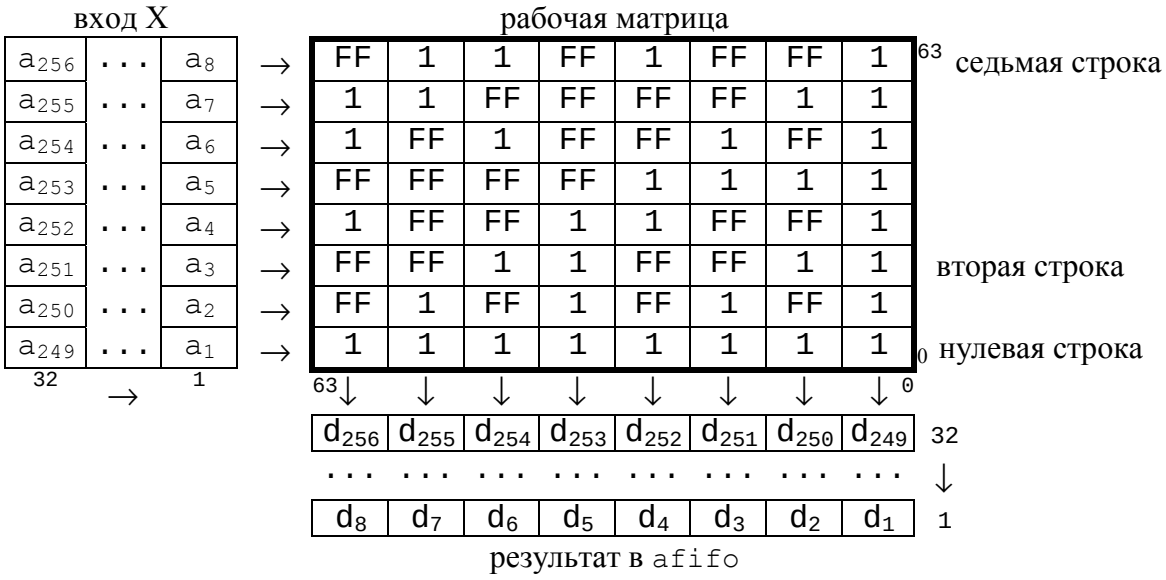
- попытка читать данные из пустого `wfifo`;
- попытка записывать данные в заполненное `wfifo` без одновременной их выгрузки в теневую матрицу;

В случае возникновения ошибки при работе с `ram` по указанным выше причинам команда, в которой содержится ошибка, не отрабатывается. Процессор заменяет ее пустой командой `nul` и порождает прерывание по неправильной векторной команде.

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. 3-8 представлена схема обработки данных в векторной части процессора:

Рис. 3-8 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. 3-8, 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] = (0010101010101010hl,  // 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 тд.

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>
    nb1 = 80808080h;  // разбиение матрицы на столбцы.
    sb  = 03030303h;  // разбиение матрицы на строки.

    ar0 = Matrx;
    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 команды `wtw`.

Регистры `nb1` и `sb` являются 64-х битными. 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 параграфах 3.3.2. Регистр nb1 на стр. 3-41 и 3.3.3. Регистр на стр. 3-45.

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 занимает всегда 32 такта, независимо от того, сколько слов загружается в матрицу. Например, в случае загрузки матрицы из восьми слов, их перекодировка во внутреннее представление по прежнему длится 32 такта. Однако, перекодировка ведется параллельно с закачкой весов и начинается с момента появления первого слова в `wfifo`.

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.

4 Формат ассемблерных инструкций

4.1 Типы скалярных команд 4-4
4.2 Типы векторных команд 4-6
4.3 Машинные коды команд процессора 4-6

Процессор NM6403 работает с машинными командами 32-х и 64-х разрядного формата. В одной машинной команде содержится две операции процессора. В этом смысле NM6403 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 из 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 компиляции ассемблером приходится на нечетный адрес, перед ней автоматически вставляется пустая команда `nul`.

Процессор NM6403 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 на стр. 3-25.

Типы инструкций процессора

Все инструкции процессора NM6403 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 NM6403 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 инструкций 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.

Примечание

Векторные и скалярные инструкции не могут смешиваться в одной команде, даже если у одной из них опущена левая часть, а у другой правая.

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 компилятор.

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`.

Использование одних и тех же регистров будет обсуждаться в разделе XXXX.

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:

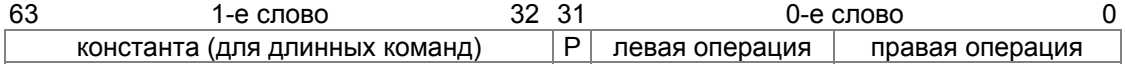
```
rep 32 ram = [ar0++gr0] with vsum , data, afifo;
```

Ключевое слово `with` разделяет левую и правую части. Счетчик повторений `rep` *`число`* присутствует в каждой векторной команде за исключением стоящих отдельно `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 приведён список скалярных и векторных команд с указанием длины в 32-х разрядных словах.

В первой колонке таблицы указан условный номер машинной команды. Эти индексы введены для удобства ссылки на машинные коды и будут в дальнейшем использоваться.

Табл. 4-2 Полный список машинных команд процессора NM6403.

№	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	–
5.1	Векторная загрузка/выгрузка	1	–
5.2	Загрузка весов из памяти	1	–
5.3	Пустая векторная команда	1	–

5 Набор инструкций языка ассемблера

5.1 Скалярные инструкции NM6403 5-3
5.1.1 Пустая команда 5-3
5.1.2 Команды чтения из памяти 5-4
5.1.3 Команды записи в память 5-6
5.1.4 Команды работы со стеком 5-9
5.1.5 Команды копирования регистров 5-10
5.1.6 Команды инициализации регистров константами 5-12
5.1.7 Команды модификации адресных регистров 5-14
5.1.8 Команды модификации регистра pswr 5-15
5.1.9 Команды перехода 5-15
5.1.9.1 Команды безусловного перехода 5-16
5.1.9.2 Команды перехода к подпрограмме 5-17
5.1.9.3 Команды возврата из подпрограммы/прерывания 5-18
5.1.9.4 Набор условий перехода 5-19
5.1.10 Основные скалярные операции процессора 5-20
5.1.11 Арифметические операции 5-21
5.1.12 Логические операции 5-22
5.1.13 Операции установки флагов без изменения значений регистров 5-24
5.1.14 Операции сдвига 5-26
5.2 Векторные инструкции NM6403 5-27
5.2.1 Методы адресации при чтении/записи данных в ВП 5-27
5.2.2 Пустая команда и отсутствие адресных операций 5-29
5.2.3 Логические операции над операндами X и Y 5-29
5.2.4 Арифметические операции над операндами X и Y 5-30
5.2.5 Операция маскирования на векторном АЛУ 5-31
5.2.6 Операция взвешенного суммирования 5-31
5.2.7 Операции активации 5-32
5.2.8 Загрузка весов в матричный узел 5-34
5.2.9 Сохранение в памяти значений векторных регистров 5-34

В данном разделе приводится структурированный обзор набора инструкций языка ассемблера процессора NeuroMatrix® NM6403. 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).

5.1 Скалярные инструкции NM6403

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 содержит ссылку на страницу, на которой дается подробное описание команды.

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

Функция	Синтаксис	Стр.
Пустая команда	`nul;`	
Пустая команда	`nul Const;`	
Пустая команда в левой части инструкции	`nul with gr0 += gr1;`	
Пустая команда в левой части инструкции (*)	`gr0 += gr1;`	
Пустая команда в левой части инструкции	`nul Const with gr0 += gr1;`	
Пустая команда в правой части инструкции (*)	`[ar0++] = gr0;`	
Пустая команда в правой части инструкции.(**)	`with gr0 = gr1 >> 0;`	

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. Подробнее см. XXXX.*

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.

Функция	Синтаксис команды	Маш. Код
Прямое чтение из памяти в адресный регистр.	**`ar0 = [Const];`** `ar0 = [Const] with gr1 = gr2 and gr3;`	1.2
Прямое чтение из памяти в регистр общего назначения.	**`gr0 = [Const];`** `gr0 = [Const] with gr1 = gr2 A>> 1;`	1.2
Прямое чтение из памяти в регистровую пару. (*)	**`ar0,gr0 = [Const];`** `ar0,gr0 = [Const] with gr1 += gr2;`	1.2
Косвенное чтение из памяти в адресный регистр.	**`ar0 = [ar1];`** `ar0 = [ar1] with gr1 -= gr2;`	1.1
Косвенное чтение из памяти в регистр общего назначения.	**`gr0 = [ar1];`** `gr0 = [ar1] with gr1 = not gr2;`	1.1
Косвенное чтение из памяти в регистровую пару. (*)	**`ar0,gr0 = [ar1];`** `ar0,gr0 = [ar1] with gr1 += gr2;`	1.1
Косвенное чтение из памяти в адресный регистр (адресация по регистру общего назначения).	**`ar0 = [gr4];`** `ar0 = [gr4] with gr1 = -gr2;`	1.1
Косвенное чтение из памяти в регистр общего назначения (адресация по регистру общего назначения).	**`gr0 = [gr1];`** `gr0 = [gr1] with gr1 = gr2 << 10;`	1.1
Косвенное чтение из памяти в регистровую пару (адресация по регистру общего назначения). (*)	**`ar0,gr0 = [gr1];`** `ar0,gr0 = [gr1] with gr1 = gr2 or gr3;`	1.1
Косвенное чтение из памяти в адресный регистр с пост-инкрементацией адреса	**`ar0 = [ar1++];`** `ar0 = [ar1++] with gr1 -= gr2;`	1.1

Косвенное чтение из памяти в регистр общего назначения с пост-инкрементацией адреса	**gr0 = [ar1++];** gr0 = [ar1++] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистровую пару с пост-инкрементацией адреса. Адрес увеличивается на 2. (*)	**ar0,gr0 = [ar1++];** ar0,gr0 = [ar1++] with gr1 += gr2;	1.1
Косвенное чтение из памяти в адресный регистр с пре-декрементацией адреса	**ar0 = [--ar1];** ar0 = [--ar1] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистр общего назначения с пре-декрементацией адреса	**gr0 = [--ar1];** gr0 = [--ar1] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистровую пару с пре-декрементацией адреса. Адрес уменьшается на 2. (*)	**ar0,gr0 = [--ar1];** ar0,gr0 = [--ar1] with gr1 += gr2;	1.1
Косвенное чтение из памяти в адресный регистр с пост-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**ar0 = [ar1++gr1];** ar0 = [ar1++gr1] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистр общего назначения с пост-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**gr0 = [ar1++gr1];** gr0 = [ar1++gr1] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистровую пару с пост-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**ar0,gr0 = [ar1++gr1];** ar0,gr0 = [ar1++gr1] with gr1 += gr2;	1.1
Косвенное чтение из памяти в адресный регистр с пре-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**ar0 = [ar1+=gr1];** ar0 = [ar1+=gr1] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистр общего назначения с пре-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**gr0 = [ar1+=gr1];** gr0 = [ar1+=gr1] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистровую пару с пре-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**ar0,gr0 = [ar1+=gr1];** ar0,gr0 = [ar1+=gr1] with gr1 += gr2;	1.1
Косвенное чтение из памяти в адресный регистр с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**ar0 = [ar1=gr1];** ar0 = [ar1=gr1] with gr1 -= gr2;	1.1
Косвенное чтение из памяти в регистр общего назначения с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**gr0 = [ar1=gr1];** gr0 = [ar1=gr1] with gr1 -= gr2;	1.1

Косвенное чтение из памяти в регистровую пару с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**ar0,gr0 = [ar1=gr1];** ar0,gr0 = [ar1=gr1] with gr1 += gr2;	1.1
Косвенное чтение из памяти в адресный регистр с предварительной записью в адресный регистр адреса, заданного константным выражением.	**ar0 = [ar1=*Const*];** ar0 = [ar1=*Const*] with gr1 -= gr2;	1.2
Косвенное чтение из памяти в регистр общего назначения с предварительной записью в адресный регистр адреса, заданного константным выражением.	**gr0 = [ar1=*Const*];** gr0 = [ar1=*Const*] with gr1 -= gr2;	1.2
Косвенное чтение из памяти в регистровую пару с предварительной записью в адресный регистр адреса, заданного константным выражением. (*)	**ar0,gr0 = [ar1=*Const*];** ar0,gr0 = [ar1=*Const*] with gr1 += gr2;	1.2
Косвенное чтение из памяти в адресный регистр с пре-инкрементацией адреса на величину константного выражения.	**ar0 = [ar1+=*Const*];** **ar0 = [ar1-=*Const*];** ar0 = [ar1+=*Const*] with gr1 -= gr2;	1.2
Косвенное чтение из памяти в регистр общего назначения с пре-инкрементацией адреса на величину константного выражения.	**gr0 = [ar1+=*Const*];** **gr0 = [ar1-=*Const*];** gr0 = [ar1+=*Const*] with gr1 -= gr2;	1.2
Косвенное чтение из памяти в регистровую пару с пре-инкрементацией адреса на величину константного выражения.(*)	**ar0,gr0 = [ar1+=*Const*];** **ar0,gr0 = [ar1-=*Const*];** ar0,gr0 = [ar1+=*Const*] with gr1;	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.*

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.

Функция	Синтаксис команды	Маш. Код
Прямая запись в память из адресного регистра.	**[*Const*] = ar0;** [*Const*] = ar0 with gr1 = gr2 and gr3;	1.2

Прямая запись в память из регистра общего назначения.	**[*Const*] = gr0;** [*Const*] = gr0 with gr1 = gr2 A>> 1;	1.2
Прямая запись в память из регистровой пары. (*)	**[*Const*] = ar0,gr0;** [*Const*] = ar0,gr0 with gr1 += gr2;	1.2
Косвенная запись в память из адресного регистра.	**[ar0] = ar1;** [ar0] = ar1 with gr1 -= gr2;	1.1
Косвенная запись в память из регистра общего назначения.	**[ar1] = gr0;** [ar1] = gr0 with gr1 = not gr2;	1.1
Косвенная запись в память из регистровой пары. (*)	**[ar1] = ar0,gr0;** [ar1] = ar0,gr0 with gr1 += gr2;	1.1
Косвенная запись в память из адресного регистра. (адресация по регистру общего назначения).	**[gr0] = ar4;** [gr0] = ar4 with gr1 = -gr2;	1.1
Косвенная запись в память из регистра общего назначения (адресация по регистру общего назначения)	**[gr0] = gr1;** [gr0] = gr1 with gr1 = gr2 << 10;	1.1
Косвенная запись в память из регистровой пары (адресация по регистру общего назначения). (*)	**[gr1] = ar0,gr0;** [gr1] = ar0,gr0 with gr1 = gr2 or gr3;	1.1
Косвенная запись в память из адресного регистра с пост-инкрементацией адреса.	**[ar0++] = ar1;** [ar0++] = ar1 with gr1 -= gr2;	1.1
Косвенная запись в память из регистра общего назначения с пост-инкрементацией адреса.	**[ar1++] = gr0;** [ar1++] = gr0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистровой пары с пост-инкрементацией адреса. Адрес увеличивается на 2. (*)	**[ar1++] = ar0,gr0;** [ar1++] = ar0,gr0 with gr1 += gr2;	1.1
Косвенная запись в память из адресного регистра с пре-декрементацией адреса.	**[--ar1] = ar0;** [--ar1] = ar0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистра общего назначения с пре-декрементацией адреса.	**[--ar1] = gr0;** [--ar1] = gr0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистровой пары с пре-декрементацией адреса. Адрес уменьшается на 2. (*)	**[--ar1] = ar0,gr0;** [--ar1] = ar0,gr0 with gr1 += gr2;	1.1
Косвенная запись в память из адресного регистра с пост-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1++gr1] = ar0;** [ar1++gr1] = ar0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистра общего назначения с пост-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1++gr1] = gr0;** [ar1++gr1] = gr0 with gr1 -= gr2;	1.1

Косвенная запись в память из регистровой пары с пост-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1++gr1] = ar0,gr0;** [ar1++gr1] = ar0,gr0 with gr1 += gr2;	1.1
Косвенная запись в память из адресного регистра с пре-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1+=gr1] = ar0;** [ar1+=gr1] = ar0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистра общего назначения с пре-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1+=gr1] = gr0;** [ar1+=gr1] = gr0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистровой пары с пре-инкрементацией адреса на величину, записанную в регистр общего назначения. (*)	**[ar1+=gr1] = ar0,gr0;** [ar1+=gr1] = ar0,gr0 with gr1 += gr2;	1.1
Косвенная запись в память из адресного регистра с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**[ar1=gr1] = ar0;** [ar1=gr1] = ar0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистра общего назначения с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**[ar1=gr1] = gr0;** [ar1=gr1] = gr0 with gr1 -= gr2;	1.1
Косвенная запись в память из регистровой пары с предварительной записью в адресный регистр адреса, хранящегося в парном регистре общего назначения. (*)	**[ar1=gr1] = ar0,gr0;** [ar1=gr1] = ar0,gr0 with gr1 += gr2;	1.1
Косвенная запись в память из адресного регистра с предварительной записью в адресный регистр адреса, заданного константным выражением.	**[ar1=*Const*] = ar0;** ar0 = [ar1=*Const*] with gr1 -= gr2;	1.2
Косвенная запись в память из регистра общего назначения с предварительной записью в адресный регистр адреса, заданного константным выражением.	**[ar1=*Const*] = gr0;** [ar1=*Const*] = gr0 with gr1 -= gr2;	1.2
Косвенная запись в память из регистровой пары с предварительной записью в адресный регистр адреса, заданного константным выражением. (*)	**[ar1=*Const*] = ar0,gr0;** [ar1=*Const*] = ar0,gr0 with gr1 += gr2;	1.2
Косвенная запись в память из адресного регистра с пре-инкрементацией адреса на величину константного выражения.	**[ar1+=*Const*] = ar0;** **[ar1-=*Const*] = ar0;** [ar1+=*Const*] = ar0 with gr1 -= gr2;	1.2
Косвенная запись в память из регистра общего назначения с пре-инкрементацией адреса на величину константного выражения.	**[ar1+=*Const*] = gr0;** **[ar1-=*Const*] = gr0;** [ar1+=*Const*] = gr0 with gr1 -= gr2;	1.2

Косвенная запись в память из регистровой пары с пре-инкрементацией адреса на величину константного выражения.(*)	**`[ar1+=Const] = ar0,gr0;`** **`[ar1-=Const] = ar0,gr0;`** `[ar1+=Const] = ar0,gr0 with gr1;`	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.*

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.

Функция	Синтаксис команды	Стр.
Запись адресного регистра в вершину стека.	**`push ar0;`** `push ar0 with gr7 = gr1 + gr2;`	
Запись регистра общего назначения в вершину стека.	**`push gr0;`** `push gr0 with gr7 = gr0;`	
Запись регистровой пары в вершину стека.(*)	**`push ar0,gr0;`** `push ar0,gr0 with gr1 = not gr1;`	
Чтение адресного регистра из вершины стека.	**`pop ar0;`** `pop ar0 with gr7 = gr0 << 2;`	
Чтение регистра общего назначения из вершины стека.	**`pop gr0;`** `pop gr0 with gr7 = gr0;`	
Чтение регистровой пары из вершины стека.(*)	**`pop ar0,gr0;`** `pop ar0,gr0 with gr1 += gr2;`	
Удаление значения из вершины стека.	**`pop;`** `pop with gr7 -= gr0;`	

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.

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-х бит		Копирование 64-х бит
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-х бит" 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-х бит		Запись 64-х бит
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.

Функция	Синтаксис команды	Стр.
Копирование значений адресных регистров. (*)	**ar2 = ar0;** **ar2 = ar0 set;** ar2 = ar0 with gr1 = gr2 and gr3;	
Копирование значения адресного регистра в регистр общего назначения.	**gr0 = ar3;** gr0 = ar3 with gr1 = gr0 A>> 1;	
Копирование значения регистра общего назначения в адресный регистр.(*)	**ar0 = gr5;** **ar0 = gr5 set;** ar0 = gr5 with gr1 = gr0 and gr5;	
Копирование значения регистра общего назначения в регистр общего назначения.	**gr0 = gr5;** gr0 = gr5 with gr1 = gr0 or gr1;	
Копирование значения регистровой пары в регистровую пару.	**ar0,gr0 = ar4,gr4;** ar0,gr0 = ar4,gr4 with not gr1;	
Копирование значения адресного регистра в регистровую пару.	**ar0,gr0 = ar5;** ar0,gr0 = ar5 with gr1++;	
Копирование значения регистра общего назначения в регистровую пару.	**ar0,gr0 = gr4;** ar0,gr0 = gr4 with gr2 = gr1 - 1;	
Копирование значения адресного регистра в младшую/старшую часть (32 бита) специального векторного регистра управления.	**nb1l = ar5;** nb1l = ar5 with gr0 += gr1;	
Копирование значения регистра общего назначения в младшую/старшую часть (32 бита) специального векторного регистра управления.	**vrh = gr4;** vrh = gr4 with gr1 = gr1 << 3;	
Копирование значения адресного регистра в специальный векторный регистр управления (64 бита).(**)	**sb = ar5;** sb = ar5 with gr0 -= gr1;	
Копирование значения регистра общего	**f1cr = gr4;**	

назначения в специальный векторный регистр управления (64 бита).(*)	`f1cr = gr4 with gr1 = not gr1;`	
Копирование значения регистровой пары в специальный векторный регистр управления (64 бита).	**`nb1 = ar5,gr5;`** `nb1 = ar5,gr5 with gr0 -= gr1;`	
Копирование значения специального регистра в адресный регистр.	**`ar5 = lmicr;`** `ar5 = lmicr with gr0 = - gr1;`	
Копирование значения специального регистра в регистр общего назначения.	**`gr7 = t1;`** `gr7 = t1 with gr1 = not gr1;`	
Копирование значения специального регистра в регистровую пару.(**)	**`ar5,gr5 = ica1;`** `ar5,gr5 = ica1 with gr0 = - gr1;`	

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 *на стр. 3-3).*

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 отличии от команды модификации адресного регистра, которая соответствует другой машинной команде, хотя по сути выполняет те же действия. Суффикс set может быть опущен, поскольку компилятор, встретив такую строку транслирует ее в команду копирования. Разницу в использовании копирования и модификации адресного регистра см. в XXXX.

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.

Функция	Синтаксис команды	Стр.
Инициализация константой адресного регистра (*)	**ar2 = *Const*;** **ar2 = *Const* set;** ar2 = *Const* with gr1--;	
Инициализация константой регистра общего назначения.	**gr0 = *Const*;** gr0 = *Const* with gr1 = gr0 A>> 1;	
Инициализация константой регистровой пары.	**ar2,gr2 = *Const*;** ar2,gr2 = *Const* with gr1++;	
Инициализация константой специального регистра.	**occ1 = *Const*;** occ1 = *Const* with gr1 = - gr2;	
Инициализация константой младшей/старшей части (32 бита) специального векторного регистра управления.	**sbl = *Const*;** gr0 = *Const* with gr1 = gr0 >> 12;	
Инициализация 32-х битной константой специального векторного регистра управления. (64 бита)(**)	**nb1 = *Const*;** nb1 = *Const* with gr1 = gr2 - 1;	

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 отличии от команды модификации адресного регистра, которая соответствует другой машинной команде, хотя по сути выполняет те же действия. Суффикс set может быть опущен, поскольку компилятор, встретив такую строку транслирует ее в команду копирования. Разницу в использовании копирования и модификации адресного регистра см. в XXXX.

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. К первой группе относятся регистры `ar0,...,ar3`, а ко второй `ar4,...,ar7`.

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 компилятором 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.

Функция	Синтаксис команды	Стр.
Модификация адресного регистра значением другого адресного регистра.(*)	**`ar3 = ar0 addr;`** `ar3 = ar0 addr with gr1 = gr2 and gr3;`	
Модификация адресного регистра значением регистра общего назначения.(*)	**`ar0 = gr7 addr;`** `ar0 = gr7 addr with gr1 = gr2 xor gr3;`	
Модификация адресного регистра значением суммы регистров регистровой пары.	**`ar5 = ar6+gr6;`** `ar5 = ar6+gr6 with gr1 = gr2 - gr3;`	
Модификация адресного регистра значением суммы адресного регистра и константы.	**`ar4 = ar6+Const;`** `ar4 = ar6+Const with gr1 = gr2 C>> 1;`	
Модификация адресного регистра значением константы.(*)	**`ar1 = Const addr;`** `ar5 = ar6+Const with gr1 = gr2;`	
Инкрементация адресного регистра	**`ar4++;`** `ar4++ with gr1 += gr2;`	
Декрементация адресного регистра.	**`ar4--;`** `ar4-- with gr1 = - gr2;`	
Инкрементация адресного регистра значением парного ему регистра общего назначения.	**`ar2+=gr2;`** `ar2+=gr2 with gr1 = not gr2;`	
Инкрементация адресного регистра значением константы.	**`ar4+=Const;`** `ar4+=Const with gr1 -= gr2;`	

Декрементация адресного регистра значением константы.	**ar4-=*Const*;** ar2-=*Const* with gr1;	

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 отличии от команды копирования, которая соответствует другой машинной команде, хотя по сути выполняет те же действия. По умолчанию при отсутствии суффиксов в данной команде компилятор предполагает использование суффикса set. Разницу в использовании копирования и модификации адресного регистра см. в XXXX.

5.1.8 Команды модификации регистра pswr

Регистр `pswr` имеет важное 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.

Функция	Синтаксис команды	Стр.
Установка в единицу определенных битов регистра `pswr`.	**pswr set *Const*;** pswr set *Const* with gr1++;	
Безусловный переход по абсолютному адресу, задаваемому адресным регистром.	**pswr clear *Const*;** pswr clear *Const* with gr1--;	

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.

Процессор выполнит отложенный переход, когда в ассемблерной инструкции он встретит слово `delayed`. Более подробно о механизме простого и отложенного переходов см. XXXX.

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.

5.1.9.1 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.

Функция	Синтаксис команды	Стр.
Безусловный переход по абсолютному адресу, задаваемому константой, значение которой вычисляется на этапе компиляции.	**goto *Const_Addr*;** `goto Const_Addr with gr1++;`	
Безусловный переход по абсолютному адресу, задаваемому адресным регистром.	**goto ar0;** `goto ar0 with gr1--;`	
Безусловный переход по абсолютному адресу, задаваемому регистром общего назначения.	**goto gr0;** `goto gr0 with gr1 = -gr1;`	
Безусловный переход по абсолютному адресу, задаваемому суммой значений пары регистров.	**goto ar0+gr0;** `goto ar0+gr0 with gr1 = not gr1;`	
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;`	
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;`	

Безусловный переход по относительному смещению от текущего адреса, задаваемому регистром общего назначения.(*)	**skip gr0;** skip gr0 with gr1--;	
Отложенный безусловный переход.(**)	**delayed goto gr0;** delayed goto gr0 with gr1++;	

Примечание

В использовании инструкции, помеченной знаком (*), необходимо быть предельно осторожным, поскольку необходимо знать механизм вычисления смещения относительно данной команды. Лучше без надобности не использовать данную команду, поскольку вероятность неправильного вычисления относительного смещения достаточно велика. Подробнее см. XXXXXX.

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. Подробнее см. XXXX.

5.1.9.2 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 параграфе 5.1.9.4 Набор условий перехода на стр. 5-19, 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.

Функция	Синтаксис команды	Стр.
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому константой(меткой).	**call *Const_Addr*;** call *Const_Addr* with gr1++;	
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому адресным регистром.	**call ar0;** call ar0 with gr1--;	

Безусловный переход к подпрограмме по абсолютному адресу, задаваемому регистром общего назначения.	**call gr0;** call gr0 with gr1 = -gr1;	
Безусловный переход к подпрограмме по абсолютному адресу, задаваемому суммой значений пары регистров.	**call ar0+gr0;** call ar0+gr0 with gr1 = not gr1;	
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;	
Безусловный переход к подпрограмме по относительному смещению от текущего адреса, задаваемому константой. Значение смещения вычисляется на этапе компиляции.	**relcall *Const_Addr*;** relcall *Const_Addr* with gr1++;	
Безусловный переход к подпрограмме по относительному смещению от текущего адреса, задаваемому регистром общего назначения.(*)	**relcall gr0;** relcall gr0 with gr1--;	
Отложенный безусловный переход к подпрограмме.(**)	**delayed call gr0;** delayed call gr0 with gr1++;	

Примечание

В использовании инструкции, помеченной знаком (*), необходимо быть предельно осторожным, поскольку необходимо знать механизм вычисления смещения относительно данной команды. Лучше без надобности не использовать данную команду, поскольку вероятность неправильного вычисления относительного смещения достаточно велика. Подробнее см. XXXX.

Примечание

Инструкция, помеченной знаком (**), показывает пример записи отложенного безусловного перехода к подпрограмме. Ключевое слово delayed может быть использовано с любым из описанных выше типов безусловного перехода, превратив его в отложенный. Подробнее см. XXXX.

5.1.9.3 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.

Функция	Синтаксис команды	Стр.

Возврат из подпрограммы.	**return;** return with gr7 = gr1 + gr2;	
Возврат из прерывания.	**ireturn;** ireturn with gr7 = gr0;	
Отложенный возврат из подпрограммы.	**delayed return;** delayed return with gr7 = gr1 + gr2;	
Отложенный возврат из прерывания.	**delayed ireturn;** delayed ireturn with gr7 = gr0;	

5.1.9.4 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.

Функция	Синтаксис команды	Стр.
Равно нулю.	**if =0 goto ...;** if =0 goto *Label* with gr1++;	
Не равно нулю.	**if <>0 goto ...;** if <>0 goto gr0 with gr1--;	
Знаковое больше.	**if > delayed goto ...;** if > delayed goto ar0 with gr1--;	

Знаковое меньше.	**if < skip ...;** `if < skip Label with gr1--;`	
Знаковое больше или равно.	**if >= call ...;** `if >= call Label with gr1--;`	
Знаковое меньше или равно.	**if <= relcall ...;** `if <= relcall Label with gr1--;`	
Беззнаковое больше или равно.	**if u>= goto ...;** `if >= goto Label with gr7 -= gr1;`	
Беззнаковое меньше.	**if u< return ...;** `if u< return with gr7 = gr1 noflags;`	
Произошел перенос (бит C = 1).	**if carry call ...;** `if carry call ar0 with gr7 -= gr1;`	
Не произошел перенос (бит C = 0).	**if not carry return ...;** `if not carry return with gr7 = gr1;`	
Возникло переполнение (бит V = 1).	**if vtrue call ...;** `if vtrue call ar0 with gr7 -= gr1;`	
Не возникло переполнение (бит V = 0).	**if vfalse return ...;** `if vfalse return with gr7 = gr1;`	
Знаковое больше с проверкой бита переполнения.	**if v> goto ...;** `if v> goto ar0 with gr7 -= gr1;`	
Знаковое меньше с проверкой бита переполнения.	**if v< delayed goto ...;** `if v< delayed goto gr1 with gr7 = gr1;`	
Знаковое больше или равно с проверкой бита переполнения.	**if v>= relcall ...;** `if v>= relcall gr0 with gr7 -= gr1;`	
Знаковое меньше или равно с проверкой бита переполнения.	**if v<= ireturn ...;** `if v<= ireturn with gr7 = gr1;`	

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 трех-операндными. 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.
- 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.

Функция	Синтаксис команды	Стр.
Сумма значений двух регистров.	**`gr0 = gr1 + gr2;`** `ar0 = gr0 with gr0 = gr1 + gr2;`	

Сумма значений двух регистров (сокращённая форма записи).	**`gr0 += gr1;`**
	`эквивалентно gr0 = gr0 + gr1;`
	`ar0 = 100h with gr0 += gr1;`	

Сумма значения регистра с единицей.	**`gr1 = gr2 + 1;`** `ar1+=gr1 with gr1 = gr2 + 1;`	
Инкрементация значения регистра.	**`gr1++;`**	

	эквивалентно `gr1 = gr1 + 1;` `ar1++ with gr1++;`	
Разность значении двух регистров.	**`gr1 = gr0 - gr7;`** `[ar1++] = ar0 with gr1 = gr0 - gr7;`	
Разность значений двух регистров (сокращённая форма записи).	**`gr1 -= gr7;`** эквивалентно `gr1 = gr1 - gr7;` `[--ar1] = gr0 with gr1 -= gr7;`	
Вычитание единицы из значения регистра.	**`gr1 = gr2 - 1;`** `call gr4 with gr1 = gr2 - 1;`	
Декрементация значения регистра.	**`gr1--;`** эквивалентно `gr1 = gr1 - 1;` `ar1-- with gr1--;`	
Добавление значения флага переноса к значению регистра.	**`gr1 = gr2 + carry;`** `ar4 += gr4 with gr1 = gr2 + carry;`	
Сложение значений двух регистров с добавлением значения флага переноса.	**`gr1 = gr2 + gr6 + carry;`** `ar4++ with gr1 = gr2 + gr6 + carry;`	
Вычитание значения флага переноса из значения регистра.	**`gr1 = gr2 - 1 + carry;`** `ar6 -= gr6 with gr1 = gr2 - 1 + carry;`	
Вычитание значений двух регистров с учетом значения флага переноса.	**`gr1 = gr2 - gr6 - 1 + carry;`** `ar4-- with gr1 = gr2 - gr6 - 1 + carry;`	
Изменение знака регистра.	**`gr1 = - gr5;`** `goto gr4 with gr1 = - gr5;`	
Первый шаг многошагового умножения.	**`gr1 = gr2 *: gr7;`** `ar4,gr4 = gr0 with gr1 = gr2 *: gr7;`	
Последующие шаги многошагового умножения.	**`gr1 = gr2 * gr7;`** `[ar6] = gr6 with gr1 = gr2 * gr7;`	
Пример выполнения арифметической операции без изменения значения флагов в регистре `pswr`.	**`gr1 = gr2 + gr2 noflags;`** `ar5++ with gr1 = gr2 + gr2 noflags;`	

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.

Функция	Синтаксис команды	Стр.

Побитовая операция OR двух операндов.	**gr0 = gr1 or gr2;** ar0 = gr0 with gr0 = gr1 or gr2;	
Побитовая операция AND операндов.	**gr1 = gr2 and gr3;** ar0 = 100h with gr1 = gr2 and gr3;	
Побитовая операция XOR двух операндов.	**gr2 = gr3 xor gr4;** ar0 = gr0 with gr2 = gr3 or gr4;	
Операция отрицания.	**gr1 = not gr2;** 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;	
Комбинация операции отрицания второго операнда с побитовой операцией OR двух операндов.	**gr1 = gr2 or not gr3;** ar6-- with gr1 = gr2 or not gr3;	
Комбинация операции отрицания обоих операндов с побитовой операцией OR.	**gr1 = not gr2 or not gr3;** return with gr1 = not gr2 or not gr3;	
Комбинация операции отрицания первого операнда с побитовой операцией AND двух операндов.	**gr2 = not gr3 and gr4;** ar4+=gr4 with gr0 = not gr1 and gr2;	
Комбинация операции отрицания второго операнда с побитовой операцией AND двух операндов.	**gr3 = gr4 and not gr5;** [ar6]=gr6 with gr3 = gr4 and not gr5;	
Комбинация операции отрицания обоих операндов с побитовой операцией AND.	**gr3 = not gr4 and not gr5;** [ar6]=gr6 with gr3 = gr4 and not gr5;	
Комбинация операции отрицания первого операнда с побитовой операцией XOR двух операндов.	**gr2 = not gr3 xor gr4;** [Addr]=gr0 with gr2 = not gr3 xor gr4;	
Комбинация операции отрицания второго операнда с побитовой операцией XOR двух операндов.	**gr3 = gr4 xor not gr5;** ar0 = ar2 with gr2 = gr4 xor not gr5;	
Запись в регистр общего назначения логического нуля. Обнуляет регистр без использования константы.	**gr6 = false;** [ar0] = ar5,gr5 with gr6 = false;	
Запись в регистр общего назначения логической -1. Устанавливает все биты регистра в единицу.	**gr0 = true;** ar0=[ar2=10h] with gr0 = true;	
Копирование значения одного регистра общего назначения в другой.(*)	**with gr2 = gr4;** gr0 = gr5 with gr2 = gr4;	

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.

Функция	Синтаксис команды	Стр.
Установка флагов по значению регистра.	**gr0;** if > goto gr2 with gr0;	
Установка флагов по значению регистра с измененным знаком.	**- gr1;** [ar4++gr4] = ar2,gr2 with - gr1;	
Установка флагов по значению суммы регистров.	**gr2 + gr3;** ar0 = 100h with gr2 + gr3;	
Установка флагов по сумме значения регистра с единицей.	**gr4 + 1;** gr2++ with gr4 + 1;	
Установка флагов по значению разности регистров.	**gr0 - gr7;** ar0 += gr0 with gr0 + gr7;	
Установка флагов по разности значения регистра и единицы.	**gr2 - 1;** gr7 = [ar0++gr0] with gr2 - 1;	
Установка флагов по сумме значения регистра с флагом переноса.	**gr1 + carry;** ar4 -= gr4 with gr1 + carry;	
Установка флагов по сумме значений двух регистров с добавлением значения флага переноса.	**gr2 + gr6 + carry;** ar4++ with gr2 + gr6 + carry;	
Установка флагов по результату вычитания значения флага переноса из значения регистра.	**gr2 - 1 + carry;** goto Addr with gr2 - 1 + carry;	

Установка флагов по результату вычитания значения двух регистров с учетом значения флага переноса.	**gr2 - gr6 - 1 + carry;** ar4-- with gr2 - gr6 - 1 + carry;	
Установка флагов по результату побитовой операции OR над двумя операндами.	**gr1 or gr2;** ar0 = gr0 with gr1 or gr2;	
Установка флагов по результату побитовой операции AND над двумя операндами.	**gr2 and gr3;** ar0 = 100h with gr2 and gr3;	
Установка флагов по результату побитовой операции XOR над двумя операндами.	**gr3 xor gr4;** ar0 = gr0 with gr3 or gr4;	
Установка флагов по результату операции отрицания.	**not gr2;** ar0 = 100h with not gr2;	
Установка флагов по результату выполнения комбинации операции отрицания первого операнда с побитовой операцией OR двух операндов.	**not gr1 or gr2;** ar0++ with not gr1 or gr2;	
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;	
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;	
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;	
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;	
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;	
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;	
Установка флага Z в единицу, а остальных в ноль.	**false;** [ar0] = ar5,gr5 with false;	
Установка флага N в единицу, а остальных в ноль.	**true;** ar0=[ar2=10h] with true;	

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.

Функция	Синтаксис команды	Стр.
Сдвиг на произвольное число битов влево. Константа сдвига в пределах от 1 до 31.	**gr0 = gr1 << 10;** ar0 = gr0 with gr0 = gr1 << 10;	
Сдвиг на произвольное число битов влево (сокращенная запись).	**gr0 <<= 2;** эквивалентно gr0 = gr0 << 2; [ar0] = gr0 with gr0 <<= 2;	
Беззнаковый сдвиг на произвольное число битов вправо. Константа сдвига в пределах от 1 до 31.	**gr1 = gr2 >> 24;** ar0 = 100h with gr1 = gr2 >> 24;	
Беззнаковый сдвиг на произвольное число битов вправо (сокращенная запись).	**gr1 >>= 3;** эквивалентно gr1 = gr1 >> 3; [ar0++] = gr3 with gr1 >>= 3;	
Арифметический сдвиг на произвольное число битов вправо. Константа сдвига в пределах от 1 до 31.	**gr2 = gr3 A>> 5;** ar0 += gr0 with gr2 = gr3 A>> 5;	
Арифметический сдвиг на произвольное число битов вправо (сокращенная запись).	**gr2 A>>= 9;** эквивалентно gr2 = gr2 A>> 9; skip Addr with gr2 A>>= 9;	
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;	
Циклический сдвиг на произвольное число битов влево (сокращенная запись).	**gr3 R<<= 12;** эквивалентно gr3 = gr3 R<< 12; goto gr0 with gr3 R<<= 12;	
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;	
Циклический сдвиг на произвольное число битов вправо (сокращенная запись).	**gr3 R>>= 14;** эквивалентно gr3 = gr3 R>> 14; ar5-- with gr3 R>>= 14;	
Циклический сдвиг влево через бит переноса. Константа сдвига 1.	**gr4 = gr5 C<< 1;** ar0 -= gr0 with gr4 = gr5 C<< 1;	
Циклический сдвиг влево через бит переноса (сокращенная запись).	**gr4 C<<= 1;** эквивалентно gr4 = gr4 C<< 1; gr0 = gr7 with gr4 C<<= 1;	

Циклический сдвиг вправо через бит переноса. Константа сдвига 1.	**gr5 = gr6 C>> 1;** [ar0+=2] = gr0 with gr5 = gr6 C>> 1;	
Циклический сдвиг вправо через бит переноса (сокращенная запись).	**gr5 C>>= 1;** эквивалентно gr5 = gr5 C>> 1; ireturn with gr5 C>>= 1;	

5.2 *Векторные инструкции NM6403*

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 содержит ссылку на страницу, на которой дается подробное описание команды.

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.

5.2.1 Методы адресации при чтении/записи данных в ВП

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`*).

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой правой частью.

Чтение из памяти

Функция	Синтаксис команды	Стр.
Чтение по адресу, хранящемуся в адресном регистре. Адрес, с которого происходит считывание, не изменяется.	**data = [ar0];** rep 4 data = [ar0] with not data;	
Чтение по адресу, хранящемуся в регистре общего назначения. Адрес, с которого происходит считывание, не изменяется.	**ram = [gr2];** rep 12 ram = [gr2] with data + 0;	
Чтение по адресу, хранящемуся в регистре общего назначения с модификацией адресного регистра. Адрес, с которого происходит считывание, не изменяется.	**ram = [ar4=gr4];** rep 1 ram = [ar4=gr4] with data + 1;	
Чтение по адресу, хранящемуся в адресном регистре с пост-	**data = [ar0++];**	

инкрементацией.(*)	`rep 32 data = [ar0++] with data;`	
Чтение по адресу, хранящемуся в адресном регистре с пре-декрементацией.(*)	**`data = [--ar4];`** `rep 16 data = [--ar4] with data or ram;`	
Чтение по адресу, хранящемуся в адресном регистре с пост-инкрементацией адреса на значение регистра общего назначения.	**`wfifo = [ar0++gr0];`** `rep 8 wfifo = [ar0++gr0], ftw;`	
Чтение по адресу с пре-инкрементацией на значение регистра общего назначения.	**`ram = [ar0+=gr0];`** `rep 12 ram = [ar0+=gr0] with afifo - 1;`	

Запись в память

Функция	Синтаксис команды	Стр.
Запись по адресу, хранящемуся в адресном регистре. Адрес, по которому происходит запись, не изменяется.	**`[ar0] = afifo;`** `rep 4 [ar0] = afifo with not afifo;`	
Запись по адресу, хранящемуся в регистре общего назначения. Адрес, по которому происходит запись, не изменяется.	**`[gr2] = afifo;`** `rep 2 [gr2],ram = afifo with afifo + 0;`	
Запись по адресу, хранящемуся в регистре общего назначения с модификацией адресного регистра. Адрес, по которому происходит запись, не изменяется.	**`[ar4=gr4] = afifo;`** `rep 1 [ar4=gr4] = afifo with ram + 1;`	
Запись по адресу, хранящемуся в адресном регистре с пост-инкрементацией.(*)	**`[ar0++] = afifo;`** `rep 4 [ar0++] = afifo with vsum ,ram,0;`	
Запись по адресу, хранящемуся в адресном регистре с пре-декрементацией.(*)	**`[--ar4] = afifo;`** `rep 16 [--ar4] = afifo with ram - 1;`	
Запись по адресу, хранящемуся в адресном регистре с пост-инкрементацией адреса на значение регистра общего назначения.	**`[ar0++gr0] = afifo;`** `rep 8 [ar0++gr0] = afifo with not ram;`	
Запись по адресу с пре-инкрементацией на значение регистра общего назначения.	**`[ar0+=gr0] = afifo;`** `rep 12 [ar0+=gr0] = afifo with not ram;`	
Запись в память с одновременным копированием содержимого `afifo` в `ram`.(**)	**`[ar0++],ram = afifo;`** `rep 5 [ar0++],ram = afifo with 0-1;`	

Примечание

В операциях инкрементации/декрементации, помеченных знаком (*), адресный регистр при каждом обращении к памяти увеличивается/уменьшается на 2, поскольку обмен с памятью ведется 64-х разрядными словами.

Примечание

В правой части операции, помеченной знаком (**), не может использоваться буфер `ram`*, поскольку он может либо только принимать данные, либо только передавать их.*

5.2.2 Пустая команда и отсутствие адресных операций

Система команд процессора NM6403 предусматривает возможность отсутствия адресной команды левой части в векторной инструкции. Более того, существует пустая векторная команда, которая может использована в программе. Основное свойство пустой векторной команды состоит в том, что ей соответствует нулевой машинный код.

Функция	Синтаксис команды	Стр.
Пустая векторная команда	**`vnul;`**	
Отсутствие адресной операции в левой части векторной инструкции.	**`rep 32 with not ram;`**	

5.2.3 Логические операции над операндами X и Y

Логические операции над операндами **X** и **Y** могут располагаться только в правой части векторной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Функция	Синтаксис команды	Стр.
Побитовая операция OR.	**`with  X or Y;`** `rep 4 data = [ar0+=gr0] with data or ram;`	
Побитовая операция AND.	**`with X and Y;`** `rep 32 wfifo = [ar2++] with ram and afifo;`	
Побитовая операция XOR.	**`with X xor Y;`** `rep 10 [ar0++] = afifo with afifo xor ram;`	
Операция отрицания.	**`with not X;`** `rep 16 data = [ar4++],ftw with not data;`	
Комбинация операции отрицания первого операнда с побитовой операцией OR двух операндов.	**`with not X or Y;`** `rep 8 [ar2++] = afifo with not afifo or ram;`	
Комбинация операции отрицания второго операнда с побитовой операцией OR двух операндов.	**`with X or not Y;`** `rep 1 [gr4] = afifo with ram or not afifo;`	
Комбинация операции	**`with not X or not Y;`**	

отрицания обоих операндов с побитовой операцией OR.	`rep 3 data = [ar6+=gr6] with not data or not ram;`	
Комбинация операции отрицания первого операнда с побитовой операцией AND двух операндов.	**`with not X and Y;`** `rep 2 with not afifo and ram;`	
Комбинация операции отрицания второго операнда с побитовой операцией AND двух операндов.	**`with X and not Y;`** `rep 9 [ar2++] = afifo with afifo and not ram;`	
Комбинация операции отрицания обоих операндов с побитовой операцией AND.	**`with not X and not Y;`** `rep 24 ftw with ram and not afifo;`	
Комбинация операции отрицания первого операнда с побитовой операцией XOR двух операндов.	**`with not X xor Y;`** `rep 21 data = [ar0++] with not data xor ram;`	
Нулевая логическая операция (заполняет нулями afifo).	**`vfalse;`** `rep 16 [ar3++] = afifo with vfalse;`	
Логическая операция, заполняющая afifo значением -1.	**`vtrue;`** `rep 32 wfifo = [ar5++gr5] with vtrue;`	

5.2.4 Арифметические операции над операндами X и Y

Арифметические операции над операндами **X** и **Y** могут располагаться только в правой части векторной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Функция	Синтаксис команды	Стр.
Сумма двух векторов.	**`with  X + Y;`** `rep 4 data = [ar0+=gr0] with data + ram;`	
Разность двух векторов.	**`with X - Y;`** `rep 32 wfifo = [ar2++] with ram - afifo;`	
Изменение знака элементов вектора **Y**.	**`with 0 - Y;`** `rep 16 wfifo = [ar2++] with 0 - ram;`	
Сумма вектора с единичным вектором.	**`with  X + 1;`** `rep 8 ftw with afifo + 1;`	
Инициализация элементов вектора единичными значениями.	**`with  0 + 1;`** `rep 4 [ar4+=gr4] = afifo with 0 + 1;`	

Вычитание единичного вектора из вектора данных, подаваемого на вход векторного АЛУ.	**with X - 1;** `rep 8 ram = [--ar2] with data - 1;`	
Инициализация элементов вектора значением -1 (все биты равны 1).	**with 0 - 1;** `rep 8 with 0 - 1;`	

5.2.5 Операция маскирования на векторном АЛУ

Операция маскирования на векторном АЛУ располагается в правой части векторной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Функция	Синтаксис команды	Стр.
Маскирование векторов, подаваемых на входы X и Y векторного АЛУ, маской M.	**with mask M, X, Y;** `rep 4 data = [ar0+=gr0] with mask afifo, data, ram;`	
Маскирование векторов, подаваемых на входы X и Y векторного АЛУ, маской M плюс циклический сдвиг вправо на 1 бит операнда X.	**with mask M, shift X, Y;** `rep 8 data = [--ar0] with mask afifo, shift data, ram;`	

5.2.6 Операция взвешенного суммирования

Операция взвешенного суммирования располагается в правой части векторной инструкции.

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

Функция	Синтаксис команды	Стр.
Простая операция взвешенного суммирования.	**with vsum , X, 0;** `rep 12 data = [ar2++gr2] with vsum , data, 0;`	
Операция взвешенного суммирования с добавлением вектора частичных сумм.	**with vsum , X, Y;** `rep 32 ram = [--ar2] with vsum , data, vr;`	
Операция взвешенного суммирования, совмещенная с маскированием аргументов.	**with vsum M, X, Y;** `rep 8 data = [ar2++] with vsum ram, data, afifo;`	
Операция взвешенного суммирования с циклическим сдвигом вправо	**with vsum , shift X, Y;** `rep 1 [ar2=gr2] with vsum , shift ram, ram;`	

на 1 бит операнда X.		

5.2.7 Операции активации

Операции активации выполняются над операндами в правой части векторной инструкции путем добавления к имени операнда ключевого слова `activate`. В зависимости от того, совместно с каким типом команды выполняется активация она может быть логической (пороговая функция) или арифметической (функция насыщения).

В графе "Синтаксис команды" мелким шрифтом показан пример использования рассматриваемой операции в составе векторной команды с непустой левой частью.

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 на стр. 5-29. Ниже в таблице приводятся примеры активации операндов, стоящих на различных позициях в правой части векторной инструкции.

Функция	Синтаксис команды	Стр.
Пример логической операции с активацией операнда X.	**`with activate X or Y;`** `rep 32 data = [ar0++] with activate data or ram;`	
Пример логической операции с активацией операнда X, когда над X дополнительно совершается операция отрицания.	**`with not activate X and Y;`** `rep 16 data = [--ar0] with not activate data and ram;`	
Пример логической операции с активацией операнда Y.	**`with X or activate Y;`** `rep 8 data = [ar0++gr0] with data or activate ram;`	
Пример логической операции с активацией операнда Y, когда над Y дополнительно совершается операция отрицания.	**`with X and not activate Y;`** `rep 12 [--ar0] = afifo with afifo and not activate ram;`	
Пример логической операции с активацией операнда обоих операндов.	**`with activate X xor activate Y;`** `rep 8 ftw with activate afifo xor activate ram;`	
Пример логической операции с активацией обоих операндов, когда над ними дополнительно совершается операция отрицания.	**`with not activate X and not activate Y;`** `rep 2 with not activate afifo and not activate ram;`	

Маскирование векторов, с активацией операнда X.	**with mask M, activate X, Y;** `rep 8 data = [ar0++] with mask afifo, activate data, ram;`	
Маскирование векторов, с логической активацией операнда Y.	**with mask M, X, activate Y;** `rep 1 data = [--ar0] with mask afifo, data, activate ram;`	
Маскирование векторов, с логической активацией обоих операндов.	**with mask M, activate X, activate Y;** `rep 4 with mask afifo, activate ram, activate ram;`	
Маскирование векторов, с логической активацией операнда X, который затем подвергается операции циклического сдвига на 1 бит вправо.	**with mask M, shift activate X, Y;** `rep 12 with mask afifo, shift activate data, ram;`	

Арифметическая активация

Ниже в таблице приводятся примеры активации операндов, стоящих на различных позициях в правой части векторной инструкции.

Функция	Синтаксис команды	Стр.
Пример арифметической операции с активацией операнда X.	**with activate X + Y;** `rep 32 data = [ar0++] with activate data + ram;`	
Пример арифметической операции с активацией операнда Y.	**with X - activate Y;** `rep 8 data = [ar0++gr0] with data - activate ram;`	
Пример арифметической операции с активацией обоих операндов.	**with activate X + activate Y;** `rep 16 data = [--ar0] with activate data + activate ram;`	
Пример операции взвешенного суммирования с активацией операнда X.	**with vsum , activate X, Y;** `rep 2 data = [ar4++] with vsum , activate data, ram;`	
Пример операции взвешенного суммирования с активацией операнда Y.	**with vsum , X, activate Y;** `rep 2 [ar6++] = afifo with vsum , ram, activate afifo;`	
Пример операции взвешенного суммирования с активацией обоих операндов.	**with vsum M, activate X, activate Y;** `rep 5 with vsum ram, activate ram, activate afifo;`	
Пример операции взвешенного суммирования с активацией операнда X, который затем подвергается операции циклического сдвига на 1 бит вправо.	**with vsum , shift activate X, 0;** `rep 30 ftw with vsum , shift activate ram, 0;`	

5.2.8 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.

Функция	Синтаксис команды	Стр.
Загрузка весовых коэффициентов из памяти в буфер `wfifo`.	**`rep 24 wfifo = [ar0++];`** `rep 24 wfifo = [ar0++] with ram + 1;`	
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;`	
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;`	
Передача весовых коэффициентов из `wfifo` в теневую матрицу.(**)	**`ftw;`** `rep 16 ftw with not ram and afifo;`	
Копирование весовых коэффициентов из теневой матрицы в рабочую.(*)	**`wtw;`** `rep 12 wtw with activate afifo;`	

Примечание

В инструкциях, помеченных знаком (*), используется операция `wtw` *копирования содержимого теневой матрицы в рабочую. Из-за наличия в процессоре аппаратной ошибки необходимо при использовании операции* `wtw` *придерживаться правил, описанных в XXXX.*

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;.
```

5.2.9 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.

Функция	Синтаксис команды	Стр.

Сохранение векторных регистров в буфер `afifo`.	`rep 5 with store vregs;`	

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. Более подробно см. XXXX.

АКЦИОНЕРНОЕ ОБЩЕСТВО
НАУЧНО-ТЕХНИЧЕСКИЙ ЦЕНТР

Научно-технический центр Модуль
АЯ 166, Москва, 125190, Россия
Тел: +7 (095) 152-9335
Факс: +7 (095) 152-4661
E-Mail: postmast@module.ru
WWW: http://www.module.ru

Напечатано в России Дата издания: 02.06.99