Программное Обеспечение
NeuroMatrix

Первые Шаги в Разработке Программ для NeuroMatrixĐ

Версия 2.0

Đ
АКЦИОНЕРНОЕ ОБЩЕСТВО
НАУЧНО-ТЕХНИЧЕСКИЙ ЦЕНТР

Оглавление

Рисунки

Предисловие

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

О справочном руководстве

Данное руководство описывает шаг за шагом основные принципы создания программ на языке ассемблера, показывает, как разработать программу, задействуя все основные узлы процессоров семейства NeuroMatrix, в том числе, векторный процессор, и затем, вызвать её из программы на Си++.

Кроме того, приводятся основные принципы оптимизации программ с учётом архитектуры процессора, структуры и состава окружающей его периферии.

Прежде, чем начать изучение материала, необходимо установить Системное ПО для процессора NeuroMatrix на Вашем компьютере.

Все пути к исходным текстам программ, указанные в данном руководстве, отсчитываются от того каталога, на который указывает переменная окружения NEURO, задаваемая автоматически при инсталляции базового ПО для процессора NeuroMatrix.

См. раздел Переменная Окружения NEURO в Справочном Руководстве по SDK (страница 5).

Как организован данный документ

Документ разделен на две главы, каждая из которых описывает следующие вопросы:

Глава 1 Как писать программы на ассемблере для NeuroMatrix

Знакомит с основами программирования процессора на языке ассемблера:

- ♦ Описывает работу со скалярными командами, организацию циклов, совмещение в одной инструкции адресной команды и арифметической операции, отложенные переходы.
- ♦ приводит примеры использования векторных команд, объясняет, как задействовать различные вычислительные узлы векторного процессора.

Глава 2 **Методы оптимизации программ**

Содержит описание основных подходов, используемых при оптимизации ассемблерного кода, размещению различных фрагментов кода и данных по различным областям внешней памяти процессора, распределению ресурсов между параллельно выполняемыми инструкциями.

Глава 3 **Приложение A**

В данном приложении содержится дополнительная информация о количестве отложенных команд, выполняемых при переходах, о макросах и об использовании одного и того же регистра общего назначения в левой и правой частях скалярной инструкции.

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
Текст
Так помечается текст, на который необходимо обратить особое внимание.

//Текст так помечаются комментарии к программам.

Примечание *Это пример того, как оформлены все важные замечания и комментарии, возникающие по ходу описания.*

1 Как Писать Программы на Ассемблере для NeuroMatrix

Данная глава состоит из уроков, в которых на основе примеров излагаются основные принципы разработки программ на языке ассемблера для процессора NM6403.

Каждый урок состоит из программы, рекомендаций по компиляции, а также пояснений к основным фрагментам кода.

Для лучшего усвоения материала рекомендуется последовательно ознакомиться со всеми приведёнными здесь уроками.

1.1 Урок 1: Простейшая Программа на Языке Ассемблера

Исходный текст примера, используемого в данном уроке, содержится в файле `step1.asm` в каталоге: `..\Tutorial\Step1`.

Пример загружает в регистры общего назначения пару констант, затем складывает содержимое регистров и передаёт сумму в качестве возвращаемого значения.

```
global __main: label;   // объявление глобальной метки

begin ".textAAA"        // начало секции кода
<__main>                // определение глобальной метки
    gr0 = 1;            // загрузка константы в первый общий регистр
    gr1 = 2;            // загрузка константы во второй общий регистр
    gr7 = gr0 + gr1;    // нахождение суммы
    return;             // возврат из функции, возвращаемое значение хранится в gr7
end ".textAAA";         // признак окончания секции кода
```

Комментарии к Примеру

Пример начинается с объявления глобальной (`global`) метки `__main`, которая будет в данном случае определять адрес в памяти той команды, с которой начинается тело основной программы.

Объявление метки может происходить в любом месте ассемблерного файла, однако для лучшей читаемости кода рекомендуется выносить его за пределы секций.

Метка `__main` особенная (два подчёркивания перед словом `main` обязательны), так как она является меткой начала пользовательской программы. Функция с этим именем вызывается из кода начальной инициализации, автоматически добавляемого к любой пользовательской программе при компиляции (об этом см. ниже).

За объявлением глобальной метки следует секция кода. Секция кода начинается с открывающей скобки `begin` и заканчивается закрывающей скобкой `end`. Имена секций при открывающей и закрывающей скобках должны совпадать.

```
begin ".textAAA"

...

end ".textAAA";
```

Примечание

Рекомендуется имя секции кода начинать с префикса text, например: «textMyCodeSection». Дизассемблер (программа dump.exe), разбирая первые символы имени секции, поймёт, что это код программы и представит её содержимое в виде дизассемблированных инструкций. В противном случае он оставит содержимое секции в виде бинарного кода.

За открывающей скобкой `begin “.textAAA”` следует определение метки:

```
<__main>
```

Метка помечает ту команду, которая следует после нее до ближайшей `";"`. В приведённом выше примере меткой помечается инструкция `gr0 = 1;`.

Инструкции:

```
gr0 = 1;
gr1 = 2;
```

представляют собой команды инициализации константой регистров общего назначения `gr0` и `gr1`.

Инструкция:

```
gr7 = gr0 + gr1;
```

выполняет арифметическую операцию суммирования содержимого регистров `gr1` и `gr2`, а результат заносит в регистр `gr7`.

Регистр `gr7` используется для хранения возвращаемого значения при выходе из функции.

Тело программы заканчивается командой возврата из подпрограммы:

```
return;
```

Последней строкой примера стоит закрывающая скобка секции кода.

Компиляция Примера

Для компиляции данного примера необходимо перейти в папку `make_emu6403` и в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step1.asm libc.lib -cemu6403.cfg
-omain.abs
```

Программа `nmcc` представляет собой специальную оболочку (shell), упрощающую процесс запуска компилятора. Она автоматически определяет, какой набор системных компонент необходимо вызвать для сборки исполняемого файла.

Параметры `nmcc` могут располагаться в произвольном порядке. Параметр `-g` включает отладочную информацию в выходной файл. Параметр `-m` включает порождение файла-карты памяти, где показано, какой объём памяти отведён под те или иные секции кода и данных, каковы их адреса, приводит список глобальных переменных и т.д. Ключ –с задает файл конфигрурации памяти. Ключ –о задает имя выходного исполняемого файла.

Для того чтобы компиляция примера `step1.asm` прошла успешно, в командную строку добавлен файл `libc.lib`. Это библиотека времени выполнения Си. В библиотеке содержится стартовый код программы, определена точка входа `start`. Стартовый код позволяет выполнять и отлаживать программу при помощи набора утилит, входящих в состав SDK. В частности, там содержится код останова, куда программа приходит по окончании выполнения. Именно по выполнению этого кода утилиты, запустившие программу на исполнение, понимают, что программа завершена.

Внимание

Библиотека `libc.lib` добавляется автоматически, если при компиляции в командной строке встречается файл с расширением `*.cpp`. Если же программа собрана только из ассемблерных файлов, библиотека `libc.lib` должна быть добавлена вручную.

Если в командной строке не указано имя выходного файла, то его имя будет сформировано по имени первого встреченного файла.

Запуск Программы на Симуляторе

Полученный файл `main.abs` может быть выполнен на симуляторе (программа `emurun.exe`). Эта программа представляет собой программный эмулятор на уровне инструкций. С помощью команды `emurun main.abs` она выполняет программу, и в строке

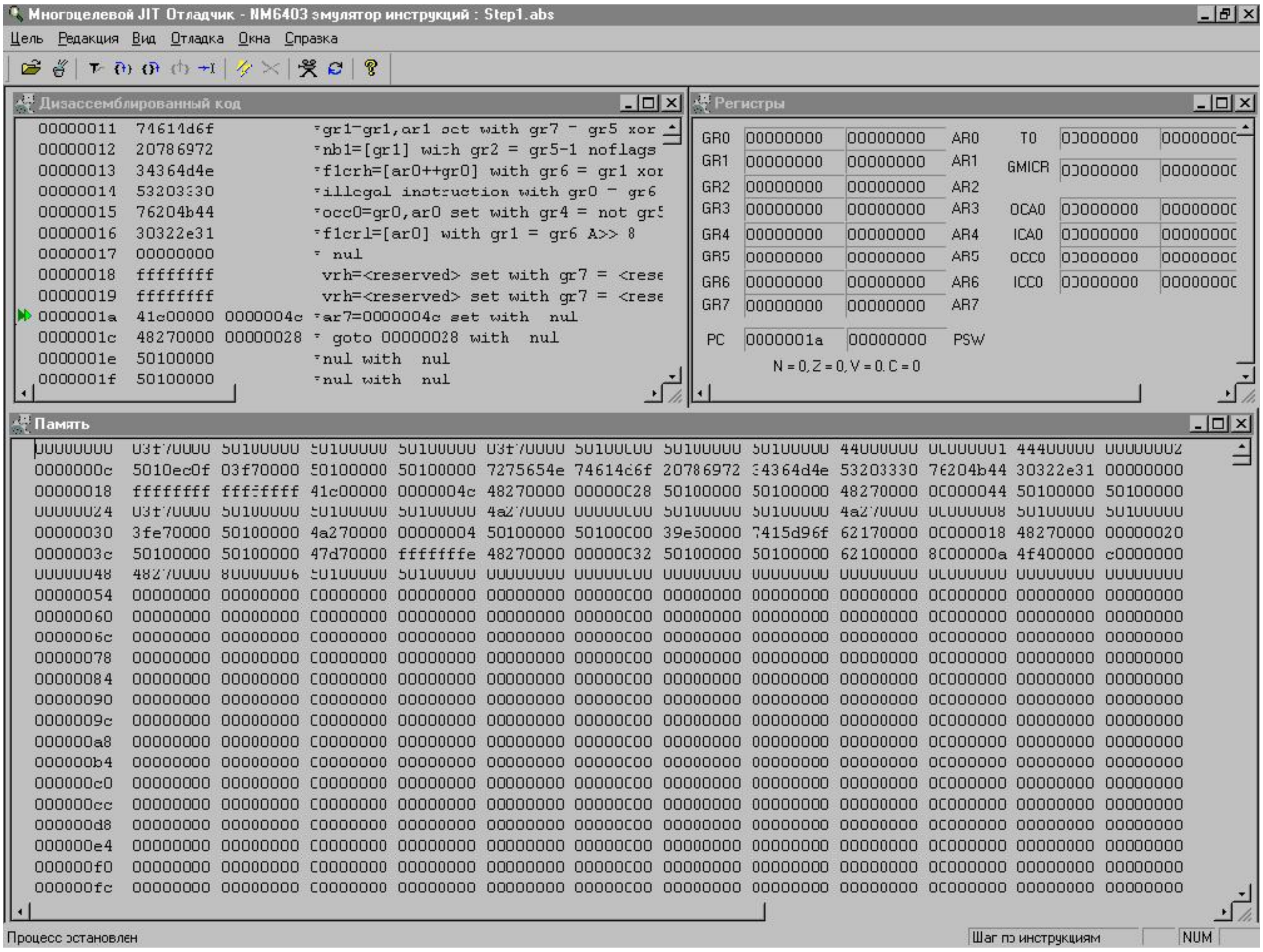
```
main.abs:: WARNING: return 3 = 0x3
```

отображает значение, возвращаемое пользовательской программой. Такая строка появляется всегда, когда функция `main()` возвращает ненулевое значение.

Отладка Программы на Символьном Отладчике

Для загрузки программы в символьный отладчик надо дважды нажать мышкой на иконку с именем пользовательской программы. Программа автоматически будет загружена в `emudbg` и остановится на точке входа в готовности выполнить первую инструкцию стартового кода.

Рис. 1-1. Внешний Вид Символьного Отладчика EMUDBG



```
00000011  74614d6f           gr1-gr1,ar1 oct with gr7 - gr5 xor
00000012  20786972           nb1=[gr1] with gr2 = gr5-1 noflags
00000013  34364d4e           f1crh=[ar0++gr0] with gr6 = gr1 xor
00000014  53203330           illegal instruction with gr0 - gr6
00000015  76204b44           occ0=gr0,ar0 set with gr4 = not gr5
00000016  30322e31           f1crl=[ar0] with gr1 = gr6 A>> 8
00000017  00000000            nul
00000018  ffffffff            vrh=<reserved> set with gr7 = <rese
00000019  ffffffff            vrh=<reserved> set with gr7 = <rese
0000001a  41c00000 0000004c  ar7=0000004c set with  nul
0000001c  48270000 00000028   goto 00000028 with  nul
0000001e  50100000           nul with  nul
0000001f  50100000           nul with  nul
```

```
GR0  00000000  00000000  AR0     T0   00000000  0000000C
GR1  00000000  00000000  AR1  GMICR   00000000  0000000C
GR2  00000000  00000000  AR2
GR3  00000000  00000000  AR3   OCA0   00000000  0000000C
GR4  00000000  00000000  AR4   ICA0   00000000  0000000C
GR5  00000000  00000000  AR5   OCC0   00000000  0000000C
GR6  00000000  00000000  AR6   ICC0   00000000  0000000C
GR7  00000000  00000000  AR7

 PC  0000001a  00000000  PSW
       N = 0, Z = 0, V = 0. C = 0
```

```
00000000 03f70000 50100000 50100000 50100000 03f70000 50100C00 50100000 50100000 44000000 0C000001 44400000 00000002
0000000c 5010ec0f 03f70000 50100000 50100000 7275654e 74614c6f 20786972 34364d4e 53203330 76204b44 30322e31 00000000
00000018 ffffffff ffffffff 41c00000 0000004c 48270000 00000C28 50100000 50100000 48270000 0C000044 50100000 50100000
00000024 03f70000 50100000 50100000 50100000 4a270000 00000C00 50100000 50100000 4a270000 0C000008 50100000 50100000
00000030 3fe70000 50100000 4a270000 00000004 50100000 50100C00 39e50000 7415d96f 62170000 0C000018 48270000 00000020
0000003c 50100000 50100000 47d70000 fffffffe 48270000 00000C32 50100000 50100000 62100000 8C00000a 4f400000 c0000000
00000048 48270000 80000006 50100000 50100000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
00000054 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
00000060 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
0000006c 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
00000078 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
00000084 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
00000090 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
0000009c 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000a8 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000b4 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000c0 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000cc 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000d8 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000e4 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000f0 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
000000fc 00000000 00000000 C0000000 00000000 00000000 00000C00 00000000 00000000 00000000 0C000000 00000000 00000000
```

Процесс эстановлен

Внимание

При первом запуске отладчик не обнаружит целевых библиотек и предложит добавить отладочную цель. В дальнейшем при запуске отладчика необходимо подключать соотвествующую отладочную цель, которая позволит осуществить запуск и отладку программы.

В данном руководстве примеры рассматриваются для случая эмулятора процессора NM6403 и поэтому в качестве отладочной цели необходимо выбрать файл `nm6403emu.dll` *из папки* `bin` *установленного SDK .*

Для того чтобы перейти к отладке строк пользовательской программы, необходимо в меню ВИД | ИСХОДНЫЕ ТЕКСТЫ выбрать файл с исходным текстом, установить курсор на строку, начиная с которой вы собираетесь вести отладку и нажать на панели инструментов кнопку →I (шаг к курсору).

Далее отладка ведётся привычным способом. Вся информация о процессе выполнения (состояние регистров, содержимое областей памяти и т.д.) может быть получена путём открытия соответствующих диалоговых окон отладчика.

1.2 Урок 2: Доступ к Памяти

Исходный текст примера, используемого в данном уроке, содержится в файле `step2.asm` в каталоге: `..\Tutorial\Step2`.

Пример демонстрирует описание различных типов секций данных, а также методы доступа к памяти.

```
global __main: label;     // объявление глобальной метки.

data ".MyData"            // секция инициализированных данных.
   A: word = 1;
   B: word = 2;
end ".MyData";

nobits ".MyData1"         // секция неинициализированных данных.
   global C: word[2];
end ".MyData1";

begin ".textAAA"          // начало секции кода.
<__main>
   ar0 = A;               // в ar0 загрузили адрес A.
   gr0 = [ar0];           // в gr0 загрузили значение ячейки памяти по адресу A.
   gr1 = [B];             // в gr1загрузили значение ячейки памяти по адресу B.
   gr2 = gr0 + gr1;       // gr2 = A + B.
   ar0 = C;               // в ar0 загрузили адрес C.
   [ar0++] = gr2;         // в память по адресу C[0] записываем содержимое gr2, а
                          // затем  увеличиваем на 1 адрес (пост-инкрементация).
   gr2 = gr0 – gr1;       // gr2 = A – B.
```

```
    [ar0++] = gr2;          // в память по адресу C[1] записываем содержимое gr2, а
                            // затем  увеличиваем на 1 адрес (пост-инкрементация).

    gr7 = [--ar0];          // gr7 = C[1]. Сначала уменьшаем адрес на единицу, а затем
                            // считываем из памяти содержимое ячейки C[1].

    return;
end ".textAAA";             // признак окончания секции кода.
```

Комментарии к Примеру

В примере описаны секции инициализированных и неинициализированных данных.

В секциях инициализированных данных содержатся объявления и инициализация переменных, используемых программой. Секция данного типа начинается с открывающей скобки `data` и заканчивается словом `end` (закрывающая скобка), например:

```
data ".MyData"
     A: word = 1;
     B: word = 2;
end ".MyData";
```

Объявление переменной имеет вид “`имя_переменной : тип`”, например “`A: word`”. Начальное значение или список начальных значений следует за объявлением типа переменной и предваряется знаком “=”.

```
A: word = 1;// (Подробнее см. раздел 2.3.2 Описания Языка
            // Ассемблера для NM6403 [AsmOver.pdf])
```

В секциях неинициализированных данных содержатся только объявления переменных, используемых программой, без их инициализации. Секция данного типа начинается с открывающей скобки `nobits` и заканчивается словом `end`. Пример:

```
nobits ".MyData1"
  global C: word[2]; // массив из 2-х 32-разрядных слов
end ".MyData1";
```

Секция кода демонстрирует команды чтения из памяти:

- Получение адреса переменной. Для этого достаточно просто использовать ее имя:
  `ar0 = A;`
  (в адресный регистр `ar0` загружаем адрес переменной `A`)

- Косвенное чтение из памяти в регистр общего назначения `gr0`
  `gr0 = [ar0];`

- Получение значения переменной. Для этого необходимо ее имя заключить в квадратные скобки.

`gr1 =[B];` *// прямое чтение из памяти в регистр общего назначения* (в регистр общего назначения `gr1` загрузили значение переменной `B`).

Команда `[ar0++] = gr2;` выполняет косвенную запись в память из регистра общего назначения с пост-инкрементацией адреса. Это означает, что по адресу, указанному в `ar0` записывается значение, содержащееся в `gr2`, после чего ar0 увеличивается на 1.

Команда `gr7 = [--ar0];` выполняет косвенное чтение из памяти в регистр общего назначения с пре-декрементацией адреса, то есть перед тем, как считать значение из памяти, адрес уменьшается на 1.

В целом, программа помещает в `C[0]` значение `A+B`, в `C[1]` значение `A-B`, а в регистр `gr7` попадает значение `C[1]`.

Компиляция Примера

Для компиляции примера `step2.asm` необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step2.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.3 Урок 3: Организация Циклов

Исходный текст примера, используемого в данном уроке, содержится в файле `step3.asm` в каталоге: `..\Tutorial\Step3`.

Пример демонстрирует простейший метод организации цикла при заполнении массива данных возрастающими значениями.

```
global __main: label;      // объявление глобальной метки.


nobits ".MyData1"          // секция неинициализированных данных.
   global C:word[16];      // объявили массив из 16 32-разрядных слов
end ".MyData1";


begin ".textAAA"           // начало секции кода.
<__main>
   ar0 = C;               // в ar0 загрузили адрес массива C.
   gr0 = 0;                // в gr0 загрузили значение 0.
   gr1 = 16;               // в gr1 загрузили значение 16, равное количеству итераций в
                           цикле.
<Loop>
```

```
    [ar0++] = gr0;      // в память по адресу ar0 записываем содержимое gr0, а
                        // затем  увеличиваем адрес на 1 (пост-инкрементация).
    gr0 ++;             // увеличили значение gr0 на 1
    gr1--;              //уменьшили значение gr1 на 1, таким образом установили
                        флаг в регистре pswr для дальнейшей проверки
    if > goto Loop;     // если условие выполнено, осуществляется переход на метку
                        Loop.

    return;
 end ".textAAA";        // признак окончания секции кода.
```

Комментарии к Примеру

В примере массив `C` в цикле последовательно заполняется возрастающими значениями.

Цикл организован путем перехода на заданную метку при выполнении определенных условий (с помощью команд условного перехода). Команда

```
if > goto Loop;
```

осуществляет переход на метку `Loop`, в случае если условие > (больше) выполнено (все сравнения осуществляются с нулём). Эта команда проверяет значение флагов, выставленных предшествующей операцией, в данном случае такой операцией является `gr1--` (Для конкретного примера `gr1` является счетчиком цикла, в процессоре NM6403 нет специального регистра - счётчика циклов). Установка флагов происходит только при выполнении арифметическо-логической операции в правой части скалярной команды.

Подробнее об условиях перехода см. раздел 5.1.9.4 Набор условий перехода в документе **NeuroMatrix NM6403 Описание Языка Ассемблера**.

Компиляция Примера

Для компиляции примера `step3.asm` необходимо в командной строке ввести команду:

```
nmcc -g –m -6403 ../Step3.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.4 Урок 3а: Оптимизация Выполнения Цикла

Исходный текст примера, используемого в данном уроке, содержится в файле `step3a.asm` в каталоге: `..\Tutorial\Step3a`.

Пример демонстрирует, как может быть оптимизирован цикл, описанный в предыдущем уроке.

```
global __main: label;      // объявление глобальной метки.

nobits ".MyData1"          // секция неинициализированных данных.
   global C:word[16];      // объявление массива из 16 32-разрядных слов
end ".MyData1";

begin ".textAAA"           // начало секции кода.
<__main>
   ar0 = C;               // в ar0 загружается адрес массива C.
   gr0 = 0;                // в gr0 загружается значение 0.
   gr1 = 16;               // в gr1 загружается значение 16, равное количеству
                           итераций в цикле.
   gr1--;                  // переменная цикла уменьшается на 1 для входа в цикл с
                           // правильно выставленными условными флагами.
<Loop>
   // если условие выполнено, осуществляется отложенный переход на метку Loop
   if > delayed goto Loop  with gr1--;
      // две следующих инструкции выполняются до того, как произойдёт переход
      [ar0++] = gr0 with gr0++ noflags;
      nul;
      // ------------- здесь произойдёт переход на метку Loop ------------------
   return;                 // сюда перейдёт программа, когда условие не выполнится
end ".textAAA";            // признак окончания секции кода.
```

Комментарии к Примеру

В языке ассемблера введено два типа команд перехода. К первому относятся команды обычного перехода, ко второму отложенного. Такое разделение введено искусственно, для удобства программирования.

От момента выбора команды перехода и до того, как состоится реальный переход проходит от одного до трёх тактов. За это время процессор успевает выбрать дополнительно одну-три инструкции, следующих непосредственно за инструкцией перехода. Назовём такие инструкции отложенными. Формализованный подход к определению точного количества отложенных инструкций, описан в приложении A.1.Типы Команд Перехода данного документа.

Упрощённая схема выполнения перехода подразумевает, что компилятор сам рассчитывает количество отложенных инструкций и заполняет их пустыми командами (`nul`). Если программист для выполнения перехода использует инструкцию без ключевого слова `delayed`, например:

```
if > goto Loop;
```

то две инструкции `nul` будут автоматически добавлены компилятором.

Если же программист захочет осмысленно использовать отложенные инструкции, то в команду перехода должно быть добавлено ключевое слово `delayed`. В конкретном примере после инструкции

```
if > delayed goto Loop with gr1--;
```

будут выполнены две отложенные инструкции:

```
[ar0++] = gr0 with gr0++ noflags;
null;
```

Отложенные инструкции выполняются в любом случае, независимо от того, выполнилось условие перехода или нет.

Рассмотрим подробнее сам цикл. Как уже отмечалось, он состоит из инструкции условного перехода и следующих за ней двух отложенных инструкций. При первом вхождении в цикл инструкция

```
if > delayed goto Loop with gr1--;
```

производит проверку флагов, выставленных предыдущей арифметической операцией, а именно: `gr1--`. При этом в правой части инструкции выполняется вычитание, которое выставляет флаги для проверки на следующем цикле.

Для того чтобы предотвратить модификацию флагов в отложенных командах, после арифметической операции используется служебное слово `‘noflags’`. Оно запрещает процессору менять флаги.

Компиляция Примера

Для компиляции примера `step3a.asm` необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step3a.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.5 Урок 4: Копирование Массива Данных на Скалярном Процессоре

Исходный текст примера, используемого в данном уроке, содержится в файле `step4.asm` в каталоге: `..\Tutorial\Step4`.

Пример демонстрирует два способа копирования массива 64-разрядных слов на скалярном процессоре. Первый способ – простое копирование, второй – копирование при помощи регистровых пар.

```
global __main: label;        // объявление глобальной метки.

data ".MyData"               // секция инициализированных данных
    // массив A из 16 64-разрядных слов заполняется начальными значениями
    global A: long[16] = ( 0l, 1l, 2l, 3l, 4l, 5hl, 6l, 7l, 8l, 9l,
                           10l, 0Bhl, 0Chl, 13l, 14l, 15l);
end ".MyData";

nobits ".MyData1"            // секция неинициализированных данных.
    global B:long[16];       // объявляется массив B из 16 64-разрядных слов
    global C:long[16];       // объявляется массив C из 16 64-разрядных слов
end ".MyData1";

begin ".textAAA"             // начало секции кода.
<__main>
    // простое копирование массива данных на скалярном процессоре
    ar0 = A;
    ar1 = B;
    gr1 = 32;                // счётчик цикла (32 цикла для копирования 16 64-bit слов)
    gr1--;                   // устанавливается флаг для первого вхождения в цикл
<Loop>
    // если условие выполнено, осуществляется отложенный переход на метку Loop
    if > delayed goto Loop  with gr1--;
        // чтение из памяти  32-разрядного слова
        gr2 = [ar0++];
        // запись в память 32-разрядного слова
        [ar1++] = gr2;

    // копирование массива данных при помощи регистровых пар
    ar0 = A;
    ar1 = B;
    gr1 = 16;                    // счётчик цикла (16 циклов для копирования 16 64-bit слов)
    gr1--;                       // устанавливается флаг для первого вхождения в цикл
<Loop1>
    // если условие выполнено, осуществляется отложенный переход на метку Loop1
    if > delayed goto Loop1 with gr1--;
        // чтение из памяти  64-разрядного слова
        gr2,ar2 = [ar0++];
        // запись в память 64-разрядного слова
        [ar1++] = ar2,gr2;

    return;
end ".textAAA";                  // признак окончания секции кода.
```

Комментарии к Примеру

В первой части примера копирование данных осуществляется через один 32-х разрядный регистр. На первом шаге в регистр заносится слово из памяти, на втором оно копируется из регистра в память по другому адресу. В данном случае значения адресных регистров каждый раз увеличиваются на единицу. Поскольку необходимо скопировать массив из шестнадцати 64-х разрядных слов, а за один цикл копирования через регистр переносится одно 32-х разрядное число (младшая или старшая половина 64-х разрядного слова), то для того, чтобы скопировать весь массив необходимо выполнить **тридцать два** цикла.

Во второй части примера копирование происходит через регистровую пару `ar2`, `gr2` (в регистровой паре каждому адресному регистру поставлен в соответствие регистр общего назначения с тем же номером). За один цикл чтения/записи переносится целиком 64-разрядное слово, поэтому количество циклов копирования равно **шестнадцати**.

При чтении из памяти в регистровую пару `ar2,gr2 = [ar0++];` ВСЕГДА младшая часть 64-разрядного слова попадает в `arX`, старшая – в `grX` независимо от того, в каком порядке перечислены регистры в паре. Те же правила действуют при записи содержимого регистровой пары в память. По младшему адресу всегда записывается содержимое регистра `arX`, по старшему - `grX`. Таким образом, команда `[ar1++] = gr2,ar2;` запишет данные в память в том же порядке, в каком они были считаны, независимо от того, в какой последовательности перечислены регистры регистровой пары.

Другим важным моментом, на который стоит обратить внимание, является то, как изменяются значения адресных регистров, используемых для доступа к памяти. И в первой, и во второй части примера используется одна и та же форма записи для инкрементации регистров `ar0` и `ar1`. Однако в первой части, когда выполняется 32-х разрядный доступ к памяти, значения адресных регистров увеличиваются на единицу, а во второй на двойку.

Процессор автоматически распознаёт, какой тип доступа к памяти используется в заданной инструкции - 32-х или 64-х разрядный. Наличие в инструкции регистровой пары или 64-х разрядного регистра управления приводит к тому, что доступ к памяти ведётся 64-х разрядными словами. Но поскольку единица адресации - 32-х разрядное слово, то при 64-х разрядном доступе простая инкрементация адресного регистра приводит к увеличению его значения на два, например:

```
gr2 = [ar0++];        // ar0 увеличивается на 1
ar2,gr2 = [ar0++];    // ar0 увеличивается на 2
```

Компиляция Примера

Для компиляции примера `step4.asm` необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step4.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.6 Урок 4а: Копирование Массива Данных на Векторном Процессоре

Исходный текст примера, используемого в данном уроке, содержится в файле `step4a.asm` в каталоге: `..\Tutorial\Step4a`.

Пример демонстрирует копирование массива 32-разрядных слов с помощью скалярного процессора и векторного процессора.

```
global __main: label;    // объявление глобальной метки.


data ".MyData"           // секция инициализированных данных

    // массив A из 16 32-разрядных слов заполняется начальными значениями
    global A: word[16] = ( 0, 1, 2, 3, 4, 5h, 6, 7, 8, 9, 10, 0Bh,
                           0Ch, 13, 14, 15);
end ".MyData";


nobits ".MyData1"        // секция неинициализированных данных.
```

```
   global B:word[16];      // объявляется массив B из 16 32-разрядных слов
   global C:word[16];      // объявляется массив C из 16 32-разрядных слов
end ".MyData1";

begin ".textAAA"           // начало секции кода.
<__main>
   // копирование массива данных с помощью скалярного процессора
   ar0 = A;
   ar1 = B;
   gr1 = 16;              // счётчик цикла (16 циклов для копирования 16-ти 32-bit слов)
   gr1--;                 // устанавливается флаг для первого вхождения в цикл

<Loop>
   // если условие выполнено, осуществляется отложенный переход на метку Loop
   if > delayed goto Loop with gr1--;
      // чтение из памяти  32-разрядного слова
      gr2  = [ar0++];
      // запись в память 32-разрядного слова
      [ar1++] = gr2;



   // копирование массивов данных с помощью векторного процессора
   ar0 = A;
   ar1 = C;
   // массив A подаётся на векторное АЛУ и попадает в afifo без изменений
   rep 8 data = [ar0++] with data;
   // сохранение во внешней памяти содержимого afifo, заполненного предыдущей
   // векторной инструкцией.
   rep 8 [ar1++] = afifo;

   return;
end ".textAAA";            // признак окончания секции кода.
```

Комментарии к Примеру

Копирование с помощью скалярного процессора подробно комментировалось в предыдущем уроке, поэтому здесь особое внимание будет уделено работе с векторным процессором.

Копирование с помощью векторного процессора:

Аналогично скалярной векторная инструкция состоит из левой и правой частей. В левой части содержится команда обращения к памяти на чтение/запись, а в правой операции на векторном процессоре.

Левая часть инструкции

```
rep 8 data = [ar0++] with data;
```

осуществляет чтение значений из внешней памяти по адресу, хранящемуся в адресном регистре, в логический регистр-контейнер `data` с пост-инкрементацией адресного регистра. В правой части инструкции данные, проходящие по шине данных, поступают на вход **X** операционного узла векторного процессора и, в данном случае, остаются без изменений. Правую часть инструкции можно представить как краткую запись выражения
`‘with data or 0’`.
Результаты выполнения векторной инструкции попадают в регистр-контейнер `afifo`.

Обязательным атрибутом векторной инструкции является количество повторений, определяющее, какое количество 64-х разрядных векторов данных обрабатывается данной инструкцией. В этом смысле векторные инструкции являются SIMD (Single Instruction Multiple Data) инструкциями, выполняя одно и то же действие над несколькими векторами данных.

При выполнении арифметических и логических операций, необходимо определить разбиение 64-разрядных векторов, поступающих на вход векторного АЛУ, на элементы. Это действие осуществляется с помощью регистра `nb1`. В данной программе перед векторными инструкциями следовало бы поместить команды

```
nb1 = 0;
wtw;// переписывает информацию из теневого регистра nb1
    // в рабочий nb2,
```

но поскольку выполняемая логическая операция не предполагает переноса битов, то эти команды можно опустить.

Команда

```
rep 8 [ar1++] = afifo;
```

осуществляет выгрузку данных из `afifo` в память с пост-инкрементацией адресного регистра. (`rep` кол-во выгружаемых слов). Нельзя выгружать данные по частям (например, сначала 4, а потом еще 4 слова), только целиком все содержимое `afifo`.

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

Следует обратить внимание на несовпадении количества повторений для скалярного и векторного процессора: количество итераций в цикле скалярного процессора равно 16, а количество повторений команд чтения/записи в векторной инструкции равно 8. Это различие возникает в связи с тем, что при обращении к памяти на скалярном процессоре считываются/записываются 32-разрядные слова, тогда как на векторном процессоре осуществляется чтение/запись 64-разрядных слов. Таким образом, при инкрементации `[ar1++]` адресный процессор каждый раз увеличивается на два.

Компиляция Примера

Для компиляции примера `step4a.asm` необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step4a.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.7 Урок 5: Арифметические Операции на Векторном АЛУ

Исходный текст примера, используемого в данном уроке, содержится в файле `step5.asm` в каталоге: `..\Tutorial\Step5`.

Пример демонстрирует выполнение арифметических операций над массивом векторов с помощью векторного АЛУ процессора NM6403. Массив из 256-ти 32-х разрядных элементов заполняется возрастающими значениями.

```
global __main: label;     // объявление глобальной метки.


data ".MyData"            // секция инициализированных данных
    // начальное значение для заполнения массива
    AA: long = 100000000hl;
    // инкремент для первого цикла
    BB: long = 200000002hl;
    // инкремент для второго цикла
    CC: long = 4000000040hl;
end ".MyData";
```

```
nobits ".MyData1"       // секция неинициализированных данных.
   .align;              // псевдокоманда для выравнивания начала массива по
                        // чётному адресу.
   // массив из 256-ти 32-х разрядных элементов, который будет заполнен возрастающими
   // значениями от 0 до 255
   global A: word[256];
end ".MyData1";

begin ".textAAA"        // начало секции кода.
<__main>
   ar0     = AA;        // в ar0 загружается адрес AA (AA = 100000000hl)
   ar4     = BB;        // в ar4 загружается адрес BB (BB = 200000002hl)
   ar1,gr1 = A;         // в ar1 и в gr1 загружается адрес буфера A
   gr2     = 31;        // счетчик цикла

   nb1 = 80000000h;     // разбиение по два 32-х разрядных элемента в векторе
   wtw;                 // копирование содержимого теневого регистра nb1 в
                        // рабочий nb2

   // в ram записывается инкремент, который будет добавляться в цикле к текущему
   // значению afifo для получения новых значений заполнителя.
   rep 1 ram = [ar4];
   // в векторный процессор заносится первое значение заполнителя.
   rep 1 data = [ar0] with data;

   gr2--;               // установка флагов для первого вхождения в цикл.
<Loop>
   if > delayed goto Loop with gr2--;
      // заполняются первые 64 элемента выходного массива
      rep 1 [ar1++] = afifo with afifo + ram;
      nul;

   gr2 = 2;                 // счётчик для второго цикла
   rep 1 [ar1++] = afifo;   // выгрузка в память последнего значения из afifo
   ar1 = gr1 with gr2--;    // возвращение в начало массива и установка флагов

   ar0 = CC;                // в ar0 загружается адрес CC (CC = 4000000040hl)
   // в ram записывается инкремент, который будет добавляться в цикле к текущему
   // значению afifo для получения новых значений заполнителя.
   rep 32 ram = [ar0];
   // в векторный процессор заносятся первые значения заполнителя.
   rep 32 data = [ar1++] with data;

<Loop1>
   if > delayed goto Loop1 with gr2--;
      // массив A заполняется возрастающими значениями
      // за один цикл обрабатываются 64 элемента.
      rep 32 [ar1++] = afifo with afifo + ram;
      nul;

// последние 64 элемента массива сохраняются в памяти
   rep 32 [ar1++] = afifo;

   return;
end ".textAAA";         // признак окончания секции кода.
```

Комментарии к Примеру

Программа состоит из двух частей, выполняющих аналогичные действия по заполнению массива A набором возрастающих значений. В первой части заполняются первые 64 элемента данных (каждый из которых является 32-х разрядным числом), а во второй части на базе результата, полученного в первой, происходит заполнение оставшейся части массива.

Программа выполняется на векторном процессоре и использует операцию суммирования на векторном АЛУ.

Инструкция

```
rep 1 ram = [ar4];
```

помещает в `ram` 64-х разрядный вектор `0000000200000002hl`. Этот вектор будет использоваться для инкрементации значений заполнения массива.

В результате выполнения инструкции

```
rep 1 data = [ar0] with data;
```

в `afifo` попадёт число `0000000100000000hl`, с которого начинается заполнение массива.

Далее в цикле происходит заполнение массива. После команды перехода выполняются две отложенные инструкции

```
rep 1 [ar1++] = afifo with afifo + ram;
nul;
```

Старое значение `afifo` сохраняется во внешней памяти, и одновременно с этим к нему прибавляется инкремент - вектор, расположенный в `ram`. Результат операции снова помещается в `afifo`. Таким образом, в массиве заполняются 64 значения (по два в каждом цикле).

Команда

```
rep 1 [ar1++] = afifo;
```

заполняет 62-й и 63-й элементы массива данными из `afifo`.

После того, как первые 64 элемента массива заполнены, они используются для заполнения остальной части массива. Для этого в `ram` заносится дублированное 32 раза число `0000004000000040hl`. Оно служит инкрементом для модификации заполнителей во втором цикле. Принцип работы процессора в обоих циклах одинаков, разница только в том, что во втором случае одна процессорная инструкция задаёт заполнение сразу 32 векторов данных.

Компиляция Примера

Для компиляции примера `step5.asm` необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step5.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.8 Урок 6: Операция Взвешенного Суммирования

Исходный текст примера, используемого в данном уроке, содержится в файле `step6.asm` в каталоге: `..\Tutorial\Step6`.

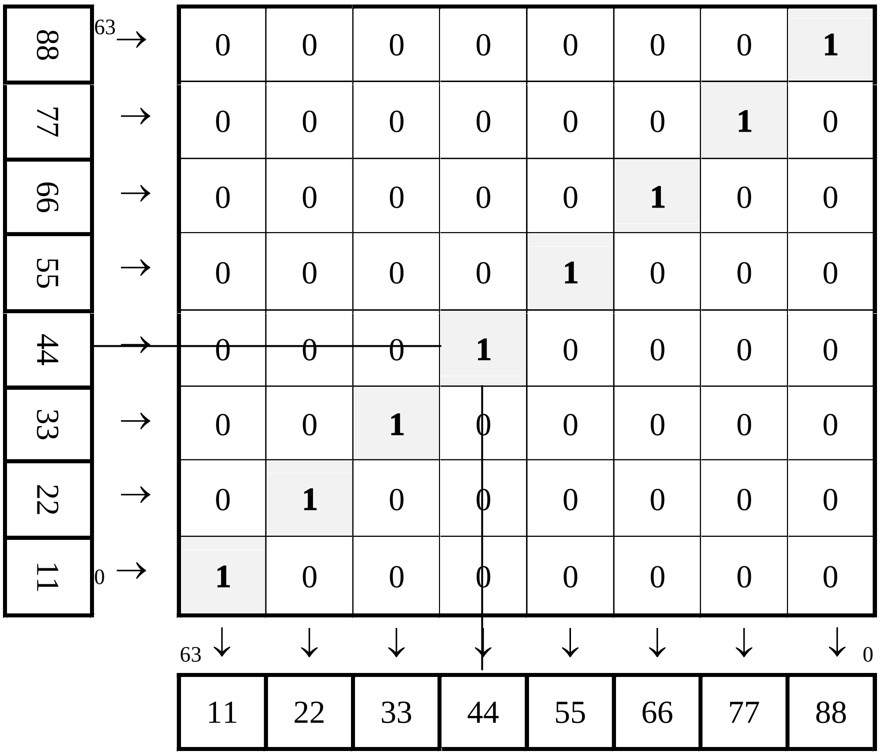
Демонстрируется пример использования устройства умножения, теневой и рабочей матрицы, входящих в состав векторного процессора NeuroMatrix. В примере рассмотрено использование операции взвешенного суммирования для перестановки байтов внутри 64-х разрядного вектора данных.

```
global __main: label;    // объявление глобальной метки.


data ".MyData"           //секция инициализированных данных
    // исходный вектор
    A: long = 8877665544332211hl;
    // место для хранения результата вычислений
    B: long = 0l;
    // массив Matr содержит значения для заполнения матрицы весовых коэффициентов
    Matr: long[8] = (0100000000000000hl,
                     0001000000000000hl,
                     0000010000000000hl,
                     0000000100000000hl,
                     0000000001000000hl,
                     0000000000010000hl,
                     0000000000000100hl,
                     0000000000000001hl);
end ".MyData";

begin ".textAAA"             // начало секции кода.
<__main>
    ar1 = Matr;
    nb1 = 80808080h;         // матрица делится на 8 столбцов по 8 бит
    sb  = 03030303h;         // матрица делится на 8 строк
    // весовые коэффициенты загружаются в буфер wfifo
    rep 8 wfifo = [ar1++];
    ftw;                     // весовые коэффициенты пересылаются в теневую матрицу
                             // с перекодировкой. Эта инструкция всегда выполняется 32
                             // такта.
    wtw;                     // весовые коэф.копируются  из теневой матрицы в рабочую
    ar2 = A;
    ar4 = B;
    // операция взвешенного суммирования, переставляющая местами байты вектора.
    rep 1 data = [ar2] with vsum , data, 0;
    // результат операции выгружается из afifo в память
    rep 1 [ar4] = afifo;
    return;
end ".textAAA";              // признак окончания секции кода.
```

Комментарии к Примеру

Задачей данного примера является перестановка порядка элементов в 64-разрядном векторе из состояния `A = 8877665544332211hl` в состояние `B = 1122334455667788hl`. Эта перестановка выполняется на устройстве умножения векторного процессора при помощи операции взвешенного суммирования. Основная идея этого преобразования поясняется на Рис. 1-2:

Рис. 1-2. Перестановка Элементов Вектора на Матричном Умножителе



0	0	0	0	0	0	0	**1**
0	0	0	0	0	0	**1**	0
0	0	0	0	0	**1**	0	0
0	0	0	0	**1**	0	0	0
0	0	0	**1**	0	0	0	0
0	0	**1**	0	0	0	0	0
0	**1**	0	0	0	0	0	0
1	0	0	0	0	0	0	0
11	22	33	44	55	66	77	88

Для выполнения операции взвешенного суммирования необходимо заполнить матрицу весовых коэффициентов значениями. Но прежде требуется разбить матрицу на строки и столбцы.

Команда `nb1 = 80808080h`; разбивает матрицу на 8 столбцов. При этом в обе части регистра попадают одинаковые константы. Таким образом, в `nb1` содержится константа `8080808080808080hl`. Регистр `nb1` является 64-разрядным регистром. Он отвечает за разбиение теневой матрицы на столбцы.

Команда `sb = 03030303h`; разбивает матрицу на 8 строк. При этом в обе части регистра попадают одинаковые константы. Таким образом, в `sb` (64 разряда) содержится константа `0303030303030303hl`.

Команда `rep 8 wfifo = [ar1++]`; осуществляет загрузку весовых коэффициентов из памяти в регистр-контейнер `wfifo`. Загрузку можно осуществлять и по частям, но так, чтобы не произошло переполнения. Контейнер `wfifo` имеет глубину в тридцать два 64-х разрядных слова.

Команда `ftw`; выполняет перекодировку весовых коэффициентов, расположенных в `wfifo`, в специальный вид, в котором они хранятся в теневой матрице. Эта операция всегда выполняется за 32 такта, однако, она может выполняться параллельно с другими векторными инструкциями.

Команда `wtw`; копирует весовые коэффициенты из теневой матрицы в рабочую.

Инструкция

```
rep 1 data = [ar2] with vsum , data, 0;
```

выполняет взвешенное суммирование с коэффициентами, которые прежде были загружены в рабочую матрицу. Вычисление производится по схеме приведённой на Рис. 1-2. Результат операции попадает в регистр-контейнер `afifo`.

Инструкция

```
rep 1 [ar4] = afifo;
```

выгружает результат из `afifo` во внешнюю память.

Компиляция Примера

Для компиляции примера `step6.asm` необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step6.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.9 Урок 6а: Операция Взвешенного Суммирования (Продолжение 1)

Исходный текст примера, используемого в данном уроке, содержится в файле `step6a.asm` в каталоге: `..\Tutorial\Step6a`.

Пример демонстрирует использование операции взвешенного суммирования для одновременного вычисления суммы и разности двух 32-х разрядных элементов вектора.

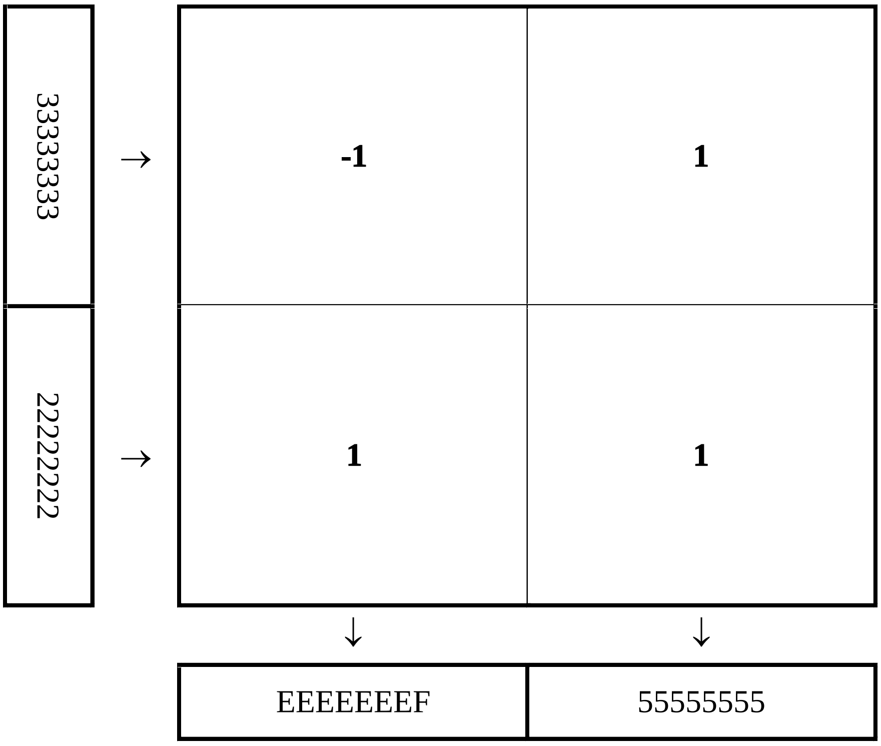
```
global __main: label;       // объявление глобальной метки.


data ".MyData"              //секция инициализированных данных
    // исходный вектор
    A: long = 3333333322222222hl;
    // место для хранения результата вычислений
    B: long = 0l;
    // массив Matr содержит значения для заполнения матрицы весовых коэффициентов
    Matr: long[2] = ( 0000000100000001hl,
                     0FFFFFFFF00000001hl);
end ".MyData";
```

```
begin ".textAAA"              // начало секции кода.
<__main>
    ar1 = Matr;
    nb1 = 80000000h;          // разбиение матрицы на два столбца по 32 бита
    sb  = 03h;                // разбиение матрицы на две строки по 32 бита
    // весовые коэффициенты загружаются в буфер wfifo, одновременно с этим они
    // транслируются в теневую матрицу, а затем и в рабочую.
    rep 2 wfifo = [ar1++], ftw, wtw;

    ar2 = A;
    ar4 = B;
    // операция взвешенного суммирования
    rep 1 data = [ar2] with vsum , data, 0;
    // результат операции выгружается из afifo в память
    rep 1 [ar4] = afifo;
    return;
end ".textAAA";               // признак окончания секции кода.
```

Комментарии к Примеру

Задачей данного примера является одновременное вычисление суммы и разности элементов 64-х разрядного вектора. Это вычисление выполняется на устройстве умножения векторного процессора при помощи операции взвешенного суммирования. Основная идея этого преобразования поясняется на Рис. 1-3:

Рис. 1-3. Вычисление Суммы и Разности Элементов Вектора



Для выполнения операции взвешенного суммирования необходимо заполнить матрицу весовых коэффициентов значениями. Но прежде требуется разбить матрицу на строки и столбцы.

Команда `nb1 = 80000000h`; разбивает матрицу на 2 столбца. При этом в обе части регистра попадают одинаковые константы. Таким образом, в `nb1` содержится константа `8000000080000000hl`.

Команда `sb = 03h`; разбивает матрицу на 2 строки. При этом в обе части регистра попадают одинаковые константы. Таким образом, в `sb` (64 разряда) содержится константа `0000000300000003hl`.

Команда `rep 2 wfifo = [ar1++], ftw, wtw`; осуществляет загрузку весовых коэффициентов из памяти в регистр-контейнер `wfifo`. Как только первое слово весов попадёт в `wfifo`, сразу же начинается его перекодирование в формат теневой матрицы (эта операция задаётся командой `ftw`). По завершению перекодирования весов в формат теневой матрицы произойдёт копирование её содержимого в рабочую матрицу. Эта операция занимает один процессорный такт.

Инструкция

```
rep 1 data = [ar2] with vsum , data, 0;
```

выполняет взвешенное суммирование с коэффициентами, которые прежде были загружены в рабочую матрицу. Вычисление производится по схеме приведённой на Рис. 1-3. Результат операции попадает в регистр-контейнер `afifo`.

Инструкция

```
rep 1 [ar4] = afifo;
```

выгружает результат из `afifo` во внешнюю память.

Компиляция Примера

Для компиляции примера `step6a.asm` необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step6a.asm libc.lib -cemu6403.cfg
-omain.abs
```

1.10 Урок 6b: Операция Взвешенного Суммирования (Продолжение 2)

Исходный текст примера, используемого в данном уроке, содержится в файле `step6b.asm` в каталоге: `..\Tutorial\Step6b`.

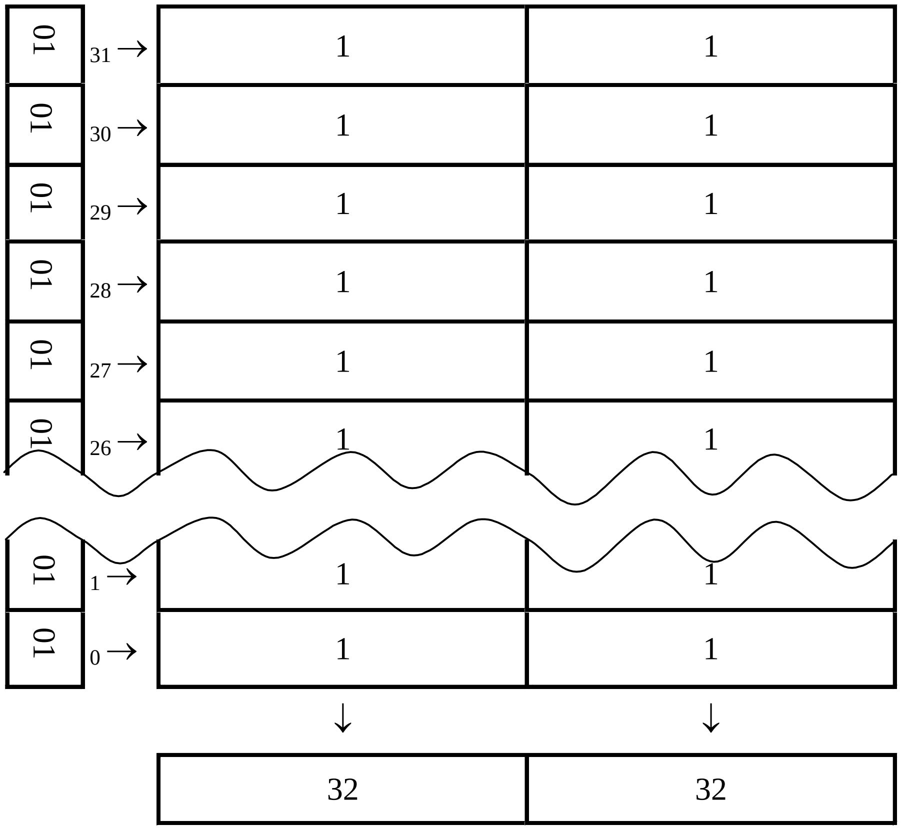
Пример демонстрирует использование операции взвешенного суммирования для вычисления суммы шестидесяти четырёх 32-разрядных слов. Предполагается, что переполнения при суммировании не происходит.

```
global __main: label;       // объявление глобальной метки.

data ".MyData"              // секция инициализированных данных
    // массив А заполняется единицами
    A: word[64] = (1 dup 64);
    // вектор, используемый для суммирования строк матрицы весов
    Vect: long = 5555555555555555hl;
end ".MyData";

nobits ".MyData1"
    Temp: long;             // временный буфер для хранения промежуточных
                            // результатов
end ".MyData1";

begin ".textAAA"            // начало секции кода
<__main>
    ar1 = A;
    nb1 = 80000000h;        // матрица делится на два столбца
    sb  = 0AAAAAAAAh;       // и на 32 строки

    // элементы массива A загружаются в буфер wfifo
    rep 32 wfifo = [ar1++], ftw, wtw;

    ar2 = Vect;
    ar4 = Temp;
    //на вход X рабочей матрицы подается Vect
    rep 1 data = [ar2] with vsum , data, 0;
    // результат операции выгружается из afifo в Temp
    rep 1 [ar4] = afifo;

    ar0,gr0 = [ar4];        // из памяти считывается пара значений
    gr1 = ar0;              // над адресными регистрами нельзя совершать
                            // арифметические операции, поэтому содержимое ar0
                            // копируется в gr1.
    // возвращаемое программой значение записывается в регистр gr7
    return with gr7 = gr1 + gr0;
end ".textAAA";             // признак окончания секции кода.
```

Комментарии к Примеру

Задачей данного примера является одновременное вычисление суммы элементов 64-х разрядного вектора. Это вычисление выполняется на устройстве умножения векторного процессора при помощи операции взвешенного суммирования. Основная идея этого преобразования поясняется на Рис. 1-4:

Рис. 1-4. Вычисление Суммы Элементов Массива



Особенность данного примера состоит в том, что в рабочую матрицу загружаются не весовые коэффициенты, а данные. Записанное в `sb` значение `AAAAAAAAh` позволяет загрузить в матрицу 32 вектора, а затем за один такт сложить элементы вдоль каждого из столбцов.

Команда `rep 32 wfifo = [ar1++], ftw, wtw;` осуществляет загрузку векторов из памяти в регистр-контейнер `wfifo`. Как только первый вектор попадёт в `wfifo`, начинается его перекодирование в формат теневой матрицы (эта операция задаётся командой `ftw`). По завершению перекодирования её содержимое будет скопировано в рабочую матрицу.

Инструкция

```
rep 1 data = [ar2] with vsum , data, 0;
```

выполняет суммирование всех тридцати двух строк. Вычисление производится по схеме приведённой на Рис. 1-4. Результат операции попадает в регистр-контейнер `afifo`.

Инструкция

```
rep 1 [ar4] = afifo;
```

выгружает результат из `afifo` во внешнюю память. После этого на скалярном процессоре две частичных суммы суммируются между собой, и результат возвращается в регистре `gr7`.

Компиляция Примера

Для компиляции примера `step6b.asm` необходимо в командной строке ввести команду:

```
nmcc -g ../Step6b.asm libc.lib -m
```

1.11 Урок 7: Вызов Ассемблерных Функций из Си++.

Начиная с данного урока все примеры оформлены в виде ассемблерных функций, вызываемых из основной программы, написанной на языке Си++.

Исходный текст примера, используемого в данном уроке, содержится в файлах `main.cpp` и `step7.asm` в каталоге: `..\Tutorial\Step7`.

Пример демонстрирует вызов функции, написанной на языке ассемблера для NM6403, из программы на языке Си++.

Файл “main.cpp”

```
extern "C" int Neg ( int value );
int main()
{
   int a = 16;
   return Neg(a);          // вызов функции, осуществляющей замену знака значения
                           //входного параметра.
}
```

Файл “step7.asm”

```
global _Neg: label;        // объявление метки с именем подпрограммы.

begin ".text"
<_Neg>
   ar5 = ar7 - 2;          // адреса в стеке для доступа к входным параметрам.
   push ar0, gr0;          // сохранение используемых в подпрограмме регистров.

   gr0 = [--ar5];          // получение значения входного параметра.
   gr7 = - gr0;

   pop ar0, gr0;           // восстановление значений регистров при выходе
   return;
```

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

Комментарии к Примеру

В программе `int main()` (из файла `main.cpp`) осуществляется вызов функции `Neg()`, написанной на языке ассемблера.

Для осуществления доступа к ассемблерной функции необходимо в заголовочном файле или непосредственно в `*.cpp` объявить эту функцию как внешнюю с Си-связыванием, например:

```
extern "C" int Neg ( int value );
```

При этом она станет доступной для вызова и передачи параметров.

Файл `step7.asm` содержит реализацию функции `Neg`. Функция возвращает значение передаваемого ей параметра с обратным знаком.

`global _Neg: label;` - метка объявляется как глобальная. Для того чтобы из файла на Си++ можно было вызвать функцию с именем `Neg,` необходимо при объявлении метки в ассемблерном файле добавлять “_” перед её именем.

Далее следует секция кода

```
 begin ".text"
<_Neg>
…
return;
end ".text";
```

Действия, описанные внутри секции кода, будут выполнены при вызове функции `Neg()`.

Ассемблерные функции, которые разрабатываются для последующих вызовов из программ на Си++, должны удовлетворять определённым требованиям по организации. Основное требование состоит в том, чтобы сохранять при входе и восстанавливать при выходе все общие регистры процессора за исключением `ar5` и `gr7`. В отдельных случаях, которые будут описаны ниже, регистр `gr6` также может быть изменён.

Первой инструкцией любой ассемблерной функции, в которую передаются входные параметры, должна быть инструкция

```
ar5 = ar7 – 2;
```

В регистр `ar5` помещается указатель на место в стеке, ниже которого находятся параметры вызова функции.

Адресный регистр `ar7` используется процессором в качестве указателя стека. Это означает, что `ar7` модифицируется автоматически, когда происходит вызов функции или прерывания, возврат из функции или прерывания. При вызове функции в стек заносятся ее входные параметры, а также регистры `- pc` и `pswr` (более подробно см. документ “Конвенция о вызовах функций”). Регистр ar7 будет указывать на свободное место в стеке.

Для корректной работы программы требуется сохранять в стеке те регистры, которые будут использованы в теле функции. Команда

```
push ar0,gr0;
```

записывает регистровую пару в вершину стека. Настоятельно рекомендуется записывать в стек 64-разрядные слова (например, регистровые пары), чтобы оставлять указатель на вершину стека четным.

Подробнее о работе со стеком см. раздел 5.1.4 Команды работы со стеком **Описания Языка Ассемблера для NM6403.**

`gr0 = [--ar5];` - в регистр `gr0` помещается значение параметра функции.

`gr7 = - gr0;` - значение, возвращаемое функцией, должно быть записано в регистр `gr7`.

`pop ar0, gr0;` - команда чтения регистровой пары из вершины стека.

`return;` - команда возврата из функции.

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step7.asm ../main.cpp libc.lib -
cemu6403.cfg -omain.abs
```

Среди входных параметров компилятора `nmcc` встречается файл с расширением *.cpp*. Компилятор воспринимает его, как текст на языке Си++ и в этом случае при редактировании связей автоматически добавляет библиотеку времени выполнения Си (`libc.lib`), поэтому её имя в командной строке может быть опущено.

1.12 Урок 8: Передаваемые и Возвращаемые Значения Типа LONG

Исходный текст примера, используемого в данном уроке, содержится в файле `step8.asm` в каталоге: `..\Tutorial\Step8`.

Пример демонстрирует возможность использования 64-разрядных переменных для передачи в качестве параметров в ассемблерную функцию и получения в качестве возвращаемого значения.

Файл “main.cpp”

```
extern "C" {
  // функции Neg_Scal и Neg_Vect объявлены как внешние с Си-связыванием
  long Neg_Scal ( long value );
  long Neg_Vect ( long value );
}

int main()
{
  long a = 0x2222222211111111;
  long b = Neg_Scal(a);
  long c = Neg_Vect(a);
  return int(b-c);
}
```

Файл “step8.asm”

```
global _Neg_Scal: label;
global _Neg_Vect: label;

nobits ".my_data"
  A: long;                   // объявляется 64-разрядная переменная A
end ".my_data";

begin ".text"
// функция Neg_Scal на скалярном процессоре выполняет обработку 64-разрядного числа,
// заменяя его знак на противоположный
<_Neg_Scal>
   ar5 = ar7 - 2;            // сохраняется указатель стека
   push ar0, gr0;            // сохраняются регистровые пары в стеке
   push ar1, gr1;
   ar0,gr0 = [--ar5];        // из стека считывается входной параметр функции
   gr1 = ar0;                // в gr1 помещается младшее слово параметра

   // в gr1записывается 0, и одновременно в gr7 помещается младшая часть параметра
   // функции с обратным знаком
   gr1 = 0 with gr7 = - gr1;
   // вычитание значений двух регистров с учетом значения флага переноса
   gr6 = gr1 – gr0 - 1 + carry;
```

```
    pop ar1, gr1;           // восстановление регистровых пар из стека
    pop ar0, gr0;
    return;                 // возвращаемое значение передаётся в регистрах:
                            // gr6 – старшая часть, gr7 – младшая чать



// функция Neg_Vect на векторном процессоре выполняет обработку 64-разрядного числа,
// заменяя его знак на противоположный
<_Neg_Vect>
    ar5 = ar7 - 2;          // сохраняется указатель стека
    push ar0, gr0;          // сохраняются регистровые пары в стеке
    push ar1, gr1;
    ar1 = A;                // в ar1загружается адрес A
    nb1 = 0;                // nb1 определяет разбиение на элементы 64-разрядного
                            // вектора, участвующего  в арифметических операциях на
                            // векторном АЛУ (nb1 = 0 – разбиения нет).
    wtw;                    // копирование содержимого теневого регистра nb1 в
                            // рабочий nb2
    // изменение знака 64-разрядного числа
    rep 1 data = [--ar5] with 0-data;
    // результат помещается в память по адресу,  хранящемуся в регистре ar1
    rep 1 [ar1] = afifo;
    gr7 = [ar1++];          // младшая часть результата копируется из памяти в
                            // регистр gr7
    gr6 = [ar1++];          // старшая часть результата копируется из памяти в
                            // регистр gr6
    pop ar1, gr1;           // восстановление регистровых пар из стека
    pop ar0, gr0;
    return;

end ".text";
```

Комментарии к Примеру

Пример состоит из двух частей, выполняющих одно и то же действие – изменение знака 64-разрядного числа.

Замена Знака Числа на Скалярнм Процессоре

Функция начинается с сохранения в `ar5` адреса входных параметров и сохранения используемых в теле функции регистров. Далее загружается входной параметр. Так как он имеет тип `long` (64-бита), то необходимо использовать регистровую пару: младшее слово параметра попадает в `ar0`, старшее – в `gr0`:

```
ar0,gr0 = [--ar5];
```

Поскольку адресный регистр не может использоваться в арифметических операциях, его значение необходимо скопировать в регистр общего назначения:

```
gr1 = ar0;
```

Инструкция

```
gr1 = 0 with gr7 = - gr1;
```

в левой части обнуляет регистр `gr1`, а в правой выполняется операция изменения знака регистра `gr1` и результат загружается в `gr7`. Регистр `gr1` используется в левой и в правой частях инструкции. В этом случае, для понимания того, какие же значения и в какое время принимает этот регистр, необходимо руководствоваться следующим правилом:

Правило

Левая и правая части инструкции выполняются одновременно, а в качестве исходных используются значения регистров, которые хранились в них до выполнения данной инструкции.

Таким образом, в правой части инструкции используется старое значение регистра `gr1`, а в результате выполнения инструкции ему присваивается новое значение. Более подробно проблема использования одного и того же регистра в обеих частях инструкции описывается в приложении А.3.Использование Регистров в Обеих Частях Скалярной Инструкции.

Операция `gr6 = gr1 - gr0 - 1 + carry`; осуществляет вычитание значений двух регистров с учетом значения флага переноса (Подробнее см. раздел 5.1.11 Арифметические операции **Описания Языка Ассемблера для NM6403**). Из значения регистра `gr1(gr1 = 0)` вычитается значение `gr0`(старшее слово входного параметра функции), при этом учитывается состояние бита переноса, выставленное предыдущей операцией.

Старшее слово результата помещается в `gr6`. В случае, если функция возвращает 64-разрядное значение, младшая часть результата должна загружаться в `gr7`, старшая – в `gr6`.

Инструкции `nb1 = 0; wtw;` определяют разбиение 64-разрядного слова на элементы. В данном случае, слово, поступающее на вход векторного АЛУ, будет представлено, как один элемент, и перенос битов будет осуществляться в пределах 64 разрядов.

`rep 1 data = [--ar5] with 0-data;` - векторная инструкция, осуществляющая арифметическую операцию на векторном АЛУ (разность операндов **X** и **Y**, где на операнд **X** подан нулевой вектор). Операция изменяет знак входного параметра функции.

`rep 1 [ar1] = afifo;` - результат из `afifo` помещается в переменную `A`, адрес которой находится в `ar1`.

Далее старшая и младшая части `A` копируются в регистры `gr6` и `gr7` соответственно.

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step8.asm ../main.cpp libc.lib -cemu6403.cfg -omain.abs
```

1.13 Урок 9: Операции Логической Активации и Маскирования на ВП

Исходный текст примера, используемого в данном уроке, содержится в файле `step9.asm` в каталоге: `..\Tutorial\Step9`.

Пример демонстрирует выполнение операции логической активации и операции маскирования на векторном АЛУ.

Файл “main.cpp”

```
// функция Mask объявлена как внешняя с Си-связыванием
extern "C" void Mask ( long *A, int msk );

long A[32];                    // объявление массива из 32 64-разрядных векторов

int main()
{
   for ( int i=0; i< 32; i++)
   {
      // массив заполняется начальными значениями
      A[i] = 0x0102030405060708*i;
   }
   Mask(A, 0x44);          // вызов функции Mask, первый параметр – адрес массива
                           // значений, второй – маска.

   return 0;
}
```

Файл “step9.asm”

```
global _Mask: label;       // объявляется глобальная метка _Mask

data ".my_data"
   Temp: long = 0l;
end ".my_data";
begin ".text"
<_Mask>
   ar5 = ar7 - 2;          // сохраняется указатель стека
   push ar0, gr0;          // в стеке сохраняются регистровые пары
   push ar1, gr1;
   push ar2, gr2;

   ar0 = [--ar5];          // в ar0 загружается адрес массива
   gr0 = [--ar5];          // в gr0 загружается маска:  00000044h

   ar2 = ar0;              // адрес входного массива копируется в ar2
   gr1 = gr0 << 8;         // gr1 = 00004400
   gr0 = gr0 or gr1;       // gr0 = 00004444
   gr1 = gr0 << 16;        // gr1 = 44440000
   gr1 = gr0 or gr1;       // gr0 = 44444444
   ar1 = gr1;
   // в переменную Temp записывается значение маски: 4444444444444444h,
   // при этом в регистр ar1 заносится адрес этой переменной.
   [ar1 = Temp] = ar1, gr1;

   nb1 = 80808080h;
   wtw;
   // регистр управления функцией активации, задаёт обработку вектора, подаваемого
   // на операнд X векторного процессора.
   f1cr = 80808080h;
```

```
    rep 32 ram = [ar0++];
    rep 32 data = [ar1] with ram - data;

    // применение логической функции активации к содержимому afifo.
    rep 32 with not activate afifo;
    // выполнение операции маскирования, маска хранится в afifo, в операнд X попадают
    // данные с шины данных, в операнд Y данные из ram.
    rep 32 data = [ar1] with mask afifo, data, ram;
    // результат опреации сохраняется во внешней памяти.
    rep 32 [ar2++] = afifo;

    pop ar2, gr2;           // восстановление регистровых пар из стека
    pop ar1, gr1;
    pop ar0, gr0;
    return;

end ".text";
```

Комментарии к Примеру

Функция `Mask` сравнивает значение каждого 8-разрядного элемента вектора со значением порога и при превышении заменяет его значением порога. Например, если в массиве был вектор `1122334455667788hl`, то после применения порога `44h` к каждому его элементу, получится `1122334444444444hl`.

На вход функции подаётся байтовый порог. Функция `Mask` содержит код, позволяющий по этим данным сформировать полноценный 64-разрядный вектор порога.

```
gr1 = gr0 << 8;    // gr1= 00004400h
gr0 = gr0 or gr1;// gr0 = 00004444h
gr1 = gr0 << 16; // gr1 = 44440000h
gr0 = gr0 or gr1; // gr0 = 44444444h
ar1 = gr1;  // копирование содержимого gr1 в парный регистр ar1
[ar1 = Temp] = ar1,gr1;
```

Последняя команда копирует содержимое регистровой пары в память по адресу `Temp`, а затем заносит этот адрес в регистр `ar1`.

Новое понятие, вводимое в данном примере - регистр управления функцией активации `f1cr` (`f2cr`). Здесь рассматривается только логическая активация, арифметическая будет рассмотрена в следующем уроке.

Регистр `f1cr` используется для управления блоком активации, расположенным на пути данных, подаваемых на операнд **X** векторного процессора, регистр `f2cr` связан с операндом **Y**.

`f1cr = 80808080h;` определяет разбиение 64-разрядного слова, подаваемого на вход **X** ВП для обработки функцией активации, на 8 элементов по 8 бит. В общем случае это разбиение может не совпадать с тем, которое задается `nb1`. (Подробнее см. раздел 3.3.1 Регистры f1cr и f2cr **Описания Языка Ассемблера для NM6403**).

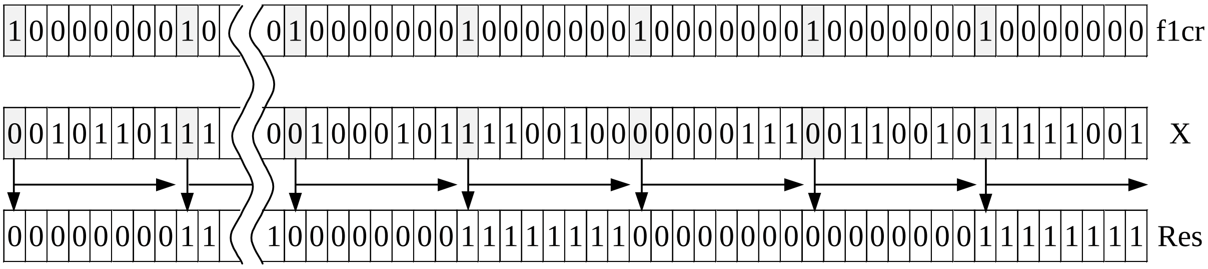
`rep 32 ram = [ar0++];` - вектора входных данных загружаются в регистр-контейнер `ram`.

`rep 32 data = [ar1] with ram - data;` - на вход векторного АЛУ из памяти поступают тридцать два одинаковых вектора порогов и выполняется операция разности векторов данных и векторов порогов.

`rep 32 with not activate afifo;` - операция логической активации данных, попавших в `afifo` в результате вычитания.

При выполнении логической активации процессор анализирует биты вектора входных данных расположенные в местах, где у регистра `f1cr` стоят ненулевые биты (см. Рис. 1-5).

Рис. 1-5. Распространение Знаковых Битов при Создании Маски



1 0 0 0 0 0 0 0 1 0 … 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 f1cr

0 0 1 0 1 1 0 1 1 1 … 0 0 1 0 0 0 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 0 1 0 1 1 1 1 1 0 0 1 X

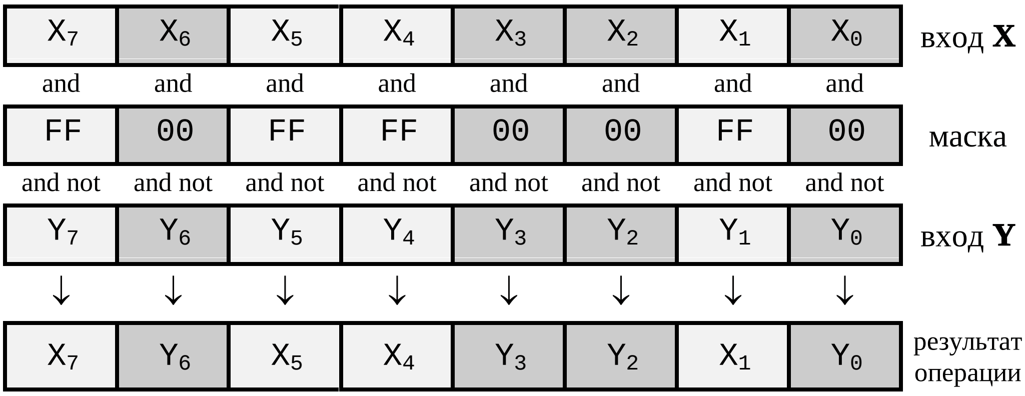
0 0 0 0 0 0 0 0 1 1 … 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 Res

Если анализируемый бит равен единице, то вправо от него до следующего ненулевого бита `f1cr` все биты вектора результата заполняются единицей, если нуль, то биты справа заполняются нулём.

В данном случае над операндом дополнительно совершается операция отрицания. В результате байты, значение которых было меньше порога, превратились в 0, остальные в –1. Результат операции помещается в `afifo`. Операция отрицания выполняется на векторном АЛУ, поэтому она применяется к уже активированному вектору. Более подробно порядок выполнения преобразований на векторном процессоре описан в пункте 1.4.6 Порядок выполнения преобразований над данными на ВП документа **NeuroMatrixÐ NM6403. Описание Языка Ассемблера.**

Результат активации, лежащий в `afifo`, затем используется для выполнения маскирования, по итогам которого все значения байтов исходного массива, превышающие порог заменяются значением порога. Суть операции маскирования приведена на Рис. 1-6:

Рис. 1-6. Операция Логического Маскирования



X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	вход **X**
and	and	and	and	and	and	and	and	
FF	00	FF	FF	00	00	FF	00	маска
and not	and not	and not	and not	and not	and not	and not	and not	
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	вход **Y**
↓	↓	↓	↓	↓	↓	↓	↓	
X_7	Y_6	X_5	X_4	Y_3	Y_2	X_1	Y_0	результат операции

Операция логического маскирования выполняется при вызове следующей инструкции:

```
rep 32 data = [ar1] with mask afifo, data, ram;
```

В данном случае вектора из `afifo` используются в качестве маски, данные, считываемые из внешней памяти – операнд **X**, данные из `ram` – операнд **Y**. Над всеми наборами векторов, принимающих участие в вычислениях, выполняется преобразование:

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

Как видно из Рис. 1-6, в тех позициях, на которых в маске стоят 1, в слово результата будут записаны элементы вектора с входа **X**, на остальные места попадут элементы вектора со входа **Y**. Таким образом, в те позиции, на которых в `afifo` стоят 1, будут записаны биты из `Temp` (регистр `ar1`), в тех позициях, на которых стоят 0, будут записаны биты из массива данных (`ram`).

`rep 32 [ar2++] = afifo;` - результат сохраняется во внешней памяти.

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step9.asm ../main.cpp libc.lib -
cemu6403.cfg -omain.abs
```

1.14 Урок 10: Операция Арифметической Активации

Исходный текст примера, используемого в данном уроке, содержится в файле `step10.asm` в каталоге: `..\Tutorial\Step10`.

Пример демонстрирует выполнение операции арифметической активации на векторном процессоре. Функция `AddSaturate()` выполняет поэлементное суммирование двух байтовых массивов, состоящих из элементов, значения которых лежат в диапазоне от `-128` до `127`. В случае переполнения функция заменяет значение на минимально возможное `0xFF(-128)` в случае отрицательного переполнения или на максимально возможное `0x7F(127)` в случае положительного переполнения.

Файл “main.cpp”

```
// функция AddSaturate объявлена как внешняя с Си связыванием
extern "C" void AddSaturate( long* Src1, long* Src2, long* Dst);

long SRC1[32];             // первый входной массив
long SRC2[32];             // второй входной массив
long DST[32];              // массив результатов

int main()
{ // заполнение массивов начальными значениями
  for (int i = 0; i < 32; i++)
  {
    SRC1[i] = 0x0203040504030201*i;
    SRC2[i] = 0x0807060804050607*i;
  }
  // вызов ассемблерной функции с передачей трёх параметров
  AddSaturate( SRC1, SRC2, DST );
  return 0;
}
```

Файл “step10.asm”

```
global _AddSaturate: label;
data ".data"
  Masks: long[24] = ( 0000000000000001hl,  // матрица для первого прохода
                      0000000000010000hl,
                     0000000100000000hl,
                     0001000000000000hl,
                     0000000000000000hl dup 4,

                     0000000000000001hl,    // матрица для второго прохода
                     0000000000000100hl,
                     0000000000010000hl,
                     0000000001000000hl,

                     0000000000000000hl dup 4,    // матрица для третьего
                     0000000000000001hl,    // прохода
                     0000000000010000hl,
                     0000000100000000hl,
                     0001000000000000hl,

                     0000000100000000hl,    // матрица для четвёртого
                     0000010000000000hl,    // прохода
                     0001000000000000hl,
                     0100000000000000hl);
end ".data";
begin ".text"
<_AddSaturate>
   ar5 = sp - 2;
   push ar0, gr0;
   push ar1, gr1;
   push ar4, gr4;
   push ar6, gr6;

   gr0 = [--ar5];         // первый входной параметр (SRC1)
   gr1 = [--ar5];         // второй входной параметр (SRC2)
   ar4 = [--ar5];         // третий входной параметр (DST)

   ar0 = gr0;
   ar1 = gr1;

   ar6 = Masks;           // адрес буфера, хранящего весовые коэффициенты
```

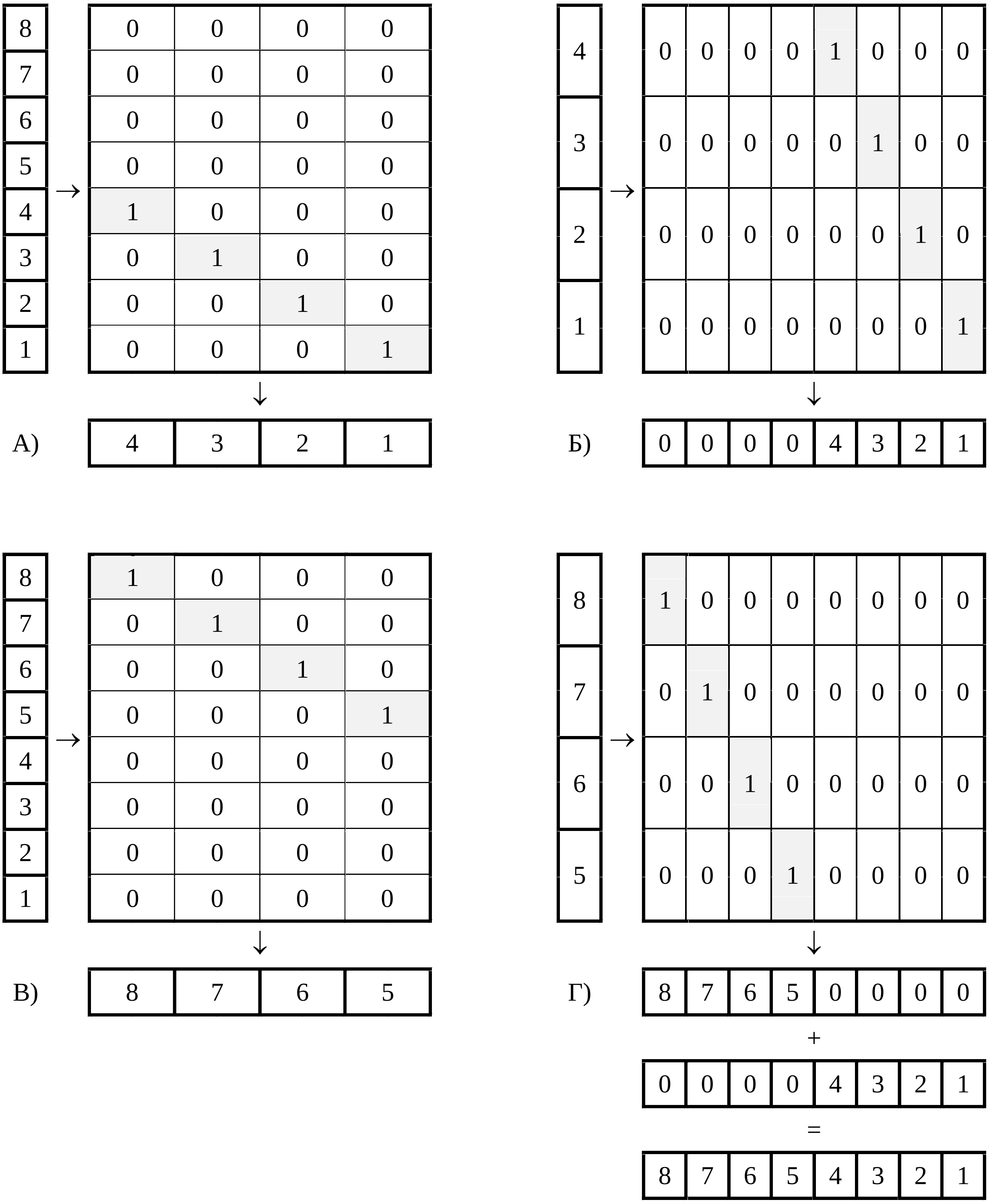
```
f1cr = 0FF80FF80h;  // конфигурация арифметической функции активации

// определение конфигурации рабочей матрицы для первого шага вычислений
nb1 = 80008000h;    // 4 столбца
sb  = 03030303h;    // 8 строк

// сразу все весовые коэффициенты (для четырёх матриц) загружаются в wfifo,
// а в теневую матрицу передаётся только 8 слов в соответствии со значениями sb и nb1
rep 24 wfifo = [ar6++],ftw, wtw;

// поскольку рабочая матрица уже загружена, можно приступить к загрузке теневой
// матрицы новой порцией весовых коэффициентов и определить новую конфигурацию.
nb1 = 80808080h;
sb  = 00030003h;
// вычисления на рабочей матрице выполняются параллельно с загрузкой теневой,
// следующие две инструкции выполняют преобразование разрядностей и поэлементное
// сложение входных векторов.
rep 32 data = [ar0++], ftw with vsum , data, 0;
rep 32 data = [ar1++] with vsum , data, afifo;

wtw;                // копирование теневой матрицы в рабочую
nb1 = 80008000h;
sb  = 03030303h;
// выполнение арифметической активации с последующим преобразованием разрядности
rep 32 ftw with vsum , activate afifo, 0;

// возвращение к началу исходных массивов для обработки вторых половин векторов
ar0 = gr0;
ar1 = gr1;
// сохранение результатов первого шага преобразования в ram
rep 32 [ar4],ram = afifo;
wtw;

// второй шаг вычислений полностью повторяет первый, отличие в весах матрицы.
nb1 = 80808080h;
sb  = 00030003h;
```

```
    rep 32 data = [ar0++], ftw with vsum , data, 0;
    rep 32 data = [ar1++] with vsum , data, afifo;

    wtw;
    // инструкция активирует данные, преобразует размерность и складывает с
    // результатом первого прохода.
    rep 32 with vsum , activate afifo, ram;

    // результат вычислений сохраняется в памяти.
    rep 32 [ar4++] = afifo;

    pop ar6, gr6;           // восстановление регистровых пар из стека
    pop ar4, gr4;
    pop ar1, gr1;
    pop ar0, gr0;
    return;

end ".text";
```

Комментарии к Примеру

Вычисления в примере выполняются в два этапа, что связано с преобразованием разрядностей входных векторов. Сначала обрабатываются четыре младших элемента входных векторов, а затем четыре старших (см. Рис. 1-7).

Рис. 1-7. Матричные Операции при Поэлементном Сложении Векторов



$$\begin{array}{|c|}\hline 8\\ \hline 7\\ \hline 6\\ \hline 5\\ \hline 4\\ \hline 3\\ \hline 2\\ \hline 1\\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|c|}\hline 0&0&0&0\\ \hline 0&0&0&0\\ \hline 0&0&0&0\\ \hline 0&0&0&0\\ \hline 1&0&0&0\\ \hline 0&1&0&0\\ \hline 0&0&1&0\\ \hline 0&0&0&1\\ \hline\end{array} \downarrow \quad \text{А)} \begin{array}{|c|c|c|c|}\hline 4&3&2&1\\ \hline\end{array}$$

$$\begin{array}{|c|}\hline 4\\ \hline 3\\ \hline 2\\ \hline 1\\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&1&0&0&0\\ \hline 0&0&0&0&0&1&0&0\\ \hline 0&0&0&0&0&0&1&0\\ \hline 0&0&0&0&0&0&0&1\\ \hline\end{array} \downarrow \quad \text{Б)} \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&4&3&2&1\\ \hline\end{array}$$

$$\begin{array}{|c|}\hline 8\\ \hline 7\\ \hline 6\\ \hline 5\\ \hline 4\\ \hline 3\\ \hline 2\\ \hline 1\\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|c|}\hline 1&0&0&0\\ \hline 0&1&0&0\\ \hline 0&0&1&0\\ \hline 0&0&0&1\\ \hline 0&0&0&0\\ \hline 0&0&0&0\\ \hline 0&0&0&0\\ \hline 0&0&0&0\\ \hline\end{array} \downarrow \quad \text{В)} \begin{array}{|c|c|c|c|}\hline 8&7&6&5\\ \hline\end{array}$$

$$\begin{array}{|c|}\hline 8\\ \hline 7\\ \hline 6\\ \hline 5\\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|c|c|c|c|c|}\hline 1&0&0&0&0&0&0&0\\ \hline 0&1&0&0&0&0&0&0\\ \hline 0&0&1&0&0&0&0&0\\ \hline 0&0&0&1&0&0&0&0\\ \hline\end{array} \downarrow \quad \text{Г)} \begin{array}{|c|c|c|c|c|c|c|c|}\hline 8&7&6&5&0&0&0&0\\ \hline\end{array} + \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&4&3&2&1\\ \hline\end{array} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 8&7&6&5&4&3&2&1\\ \hline\end{array}$$

Функция выполняет следующую последовательность действий:

- Загружает веса в матрицу для выполнения преобразования разрядности, как изображено на Рис. 1-7А;

- Выполняет преобразование младших половин двух входных векторов из 8-ми в 16 разрядов и поэлементно складывает преобразованные вектора;
- Применяет арифметическую функцию активации (насыщения) для замены сумм, превысивших диапазон `–128 .. 127`, на пограничные значения диапазона;
- Выполняет обратное преобразование из 16-ти в 8 бит, как изображено на Рис. 1-7Б;
- Сохраняет результат первого прохода во внутреннем буфере `ram`;
- Выполняет аналогичные преобразования над старшими половинами входных векторов, как изображено на Рис. 1-7В и Г, а затем складывает результаты первого и второго проходов.

Переходя к особенностям реализации алгоритма необходимо обратить внимание на то, что весовые коэффициенты, хотя и принадлежат четырём разным матрицам, загружаются в `wfifo` одной командой:

```
rep 24 wfifo = [ar6++], ftw, wtw;
```

Это одно из свойств `wfifo`, позволяющее добавлять и выбирать коэффициенты порциями. После того, как определена конфигурация теневой матрицы, процессор в соответствии с этой конфигурацией выбирает необходимое количество весов, преобразуя их в формат теневой матрицы. Оставшиеся в `wfifo` веса дожидаются своей очереди.

Инструкция

```
rep 32 data = [ar0++], ftw with vsum , data, 0;
```

совмещает в себе выполнение операции взвешенного суммирования с загрузкой новой порции весовых коэффициентов (`ftw`) в теневую матрицу.

Инструкция

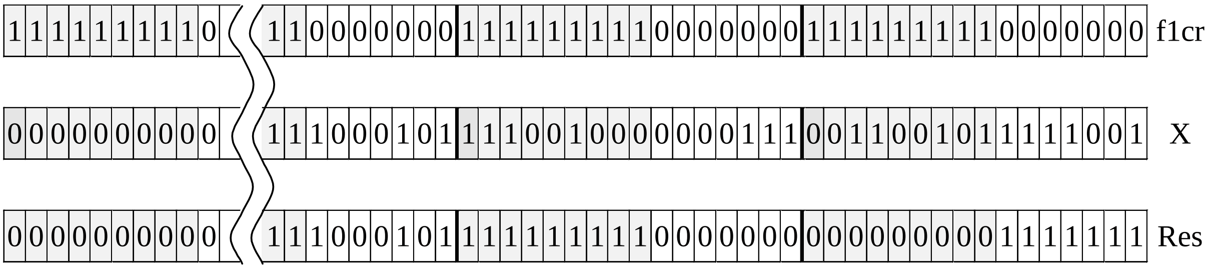
```
rep 32 ftw with vsum , activate afifo, 0;
```

выполняет арифметическую активацию (насыщение) набора векторов, хранящихся в `afifo`, а затем активированные данные поступают на матричное устройство умножения, где происходит возврат от 16-ти к 8-ми битам.

Несколько слов необходимо сказать об арифметической активации. Она называется арифметической, поскольку применяется только в том случае, когда на векторном АЛУ выполняется арифметическая операция, например: `activate ram + data`, или `0 – activate afifo`. Операция взвешенного суммирования также является арифметической, поэтому в паре с ней выполняется именно функция насыщения.

Рис. 1-8 приводит пример того, как регистр `f1cr` (`f2cr`) управляет арифметической функцией активации:

Рис. 1-8. Управление Арифметической Функцией Активации



1111111110 110000000 1111111110000000 1111111110000000 f1cr

0000000000 111000101 1110010000000111 0011001011111001 X

0000000000 111000101 1111111110000000 0000000001111111 Res

Выход за пределы диапазона `–128..127` в 16-разрядном числе может быть обнаружен по следующему признаку: несовпадение значений знаковых битов. Если 16-разрядное число принадлежит определённому выше диапазону, то биты с 7-ого по 15-ый являются знаковыми (на Рис. 1-8 помечены серым). В случае если значение хотя бы одного знакового бита отличается от значения самого старшего бита (на рисунке затемнён больше остальных), можно утверждать, что это число лежит вне заданного диапазона. Этот анализ является основой функции насыщения.

В регистре `f1cr` все знаковые биты элементов данных помечаются единицами. При обработке входного вектора процессор сравнивает все его биты, расположенные под единичными битами `f1cr` с самым старшим битом элемента. Если все знаковые биты имеют одинаковое значение, то превышения диапазона не обнаружено, число пропускается через фильтр без изменений.

В случае если в данном элементе какие-то из знаковых бит отличаются от самого старшего, детектируется выход за диапазон и число заменяется максимальным или минимальным числом диапазона (опять таки в зависимости от значения старшего бита).

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step10.asm ../main.cpp libc.lib -
cemu6403.cfg -omain.abs
```

1.15 Урок 11: Использование Циклического Сдвигателя на ВП

Исходный текст примера, используемого в данном уроке, содержится в файле `step11.asm` в каталоге: `..\Tutorial\Step11`.

Пример демонстрирует использование операции циклического сдвига на векторном процессоре при работе с бинарными элементами. Программа изменяет порядок битов во входном 64-разрядном слове данных на противоположный. Меняются местами 0-ой и 63-ий биты, 1-ый и 62-ой, и т.д.

Файл “main.cpp”

```
// функция ReverseBits объявлена внешней с Си-связыванием
extern "C" long ReverseBits(long Src);

int main()
{
  long A = 0x5555EEEEAAAA7777;  // исходное число
  long B = 0xEEEE55557777AAAA;  // ожидаемый результат перестановки битов
  long C = ReverseBits(A);      // C содержит переставленные биты

  // сравнение результата вычислений с ожидаемым значением
  if (B == C) return 1;         // результат совпал с ожидаемым
  else return –1;               // результат не совпал с ожидаемым
return 0;
}
```

Файл “step11.asm”

```
global _ReverseBits: label;
data ".data"              // весовые коэффициенты для перестановки битов
  Weights: long[64] = (   1l<<63, 1l<<61, 1l<<59, 1l<<57,
                          1l<<55, 1l<<53, 1l<<51, 1l<<49,
                          1l<<47, 1l<<45, 1l<<43, 1l<<41,
                          1l<<39, 1l<<37, 1l<<35, 1l<<33,
                          1l<<31, 1l<<29, 1l<<27, 1l<<25,
                          1l<<23, 1l<<21, 1l<<19, 1l<<17,
                          1l<<15, 1l<<13, 1l<<11, 1l<< 9,
                          1l<< 7, 1l<< 5, 1l<< 3, 1l<< 1,
                          1l<<62, 1l<<60, 1l<<58, 1l<<56,
                          1l<<54, 1l<<52, 1l<<50, 1l<<48,
                          1l<<46, 1l<<44, 1l<<42, 1l<<40,
                          1l<<38, 1l<<36, 1l<<34, 1l<<32,
                          1l<<30, 1l<<28, 1l<<26, 1l<<24,
                          1l<<22, 1l<<20, 1l<<18, 1l<<16,
                          1l<<14, 1l<<12, 1l<<10, 1l<< 8,
                          1l<< 6, 1l<< 4, 1l<< 2, 1l<< 0);
end ".data";
begin ".text"
<_ReverseBits>
    ar5 = sp - 2;
    push ar0, gr0;

    ar0 = Weights;        //в ar0 загружается адрес массива весов для матрицы ВП

    nb1 = 0FFFFFFFFh;     // 64 столбца
    sb  = 0FFFFFFFFh;     // 32 строки

    // загрузка первого набора весовых коэффициентов в рабочую матрицу
    rep 32 wfifo = [ar0++],ftw, wtw;
    // загрузка второго набора весовых коэффициентов в теневую матрицу
    rep 32 wfifo = [ar0++],ftw;

    // слово входных данных напрямую из стека загружается в ram и одновременно подаётся
    // на вход X матричного устройства умножения
    rep 1 ram = [--ar5] with vsum , data, 0;
    wtw;
    // после обновления коэффициентов тоже слово входных данных подаётся из ram
    // на вход X, а по пути циклически сдвигается на 1 бит вправо
    rep 1 with vsum , shift ram, afifo;
    rep 1 [ar5] = afifo;

    gr7 = [ar5++];        // младшее слово вектора результата – в gr7
    gr6 = [ar5++];        // старшее слово вектора результата – в gr6

    pop ar0, gr0;
    return;

end ".text";
```

Комментарии к Примеру

Для заполнения массива весовых коэффициентов в примере используются константные выражения, вычисляемые на этапе компиляции. Компилятор способен вычислять константные выражения, результатом которых является 64-разрядная константа для заполнения ячеек памяти.

В примере используется максимально возможное разбиение матриц: 64 столбца и 32 строки. При этом вектора входных данных разбиваются на тридцать два элемента разрядностью по 2 бита, а результирующий вектор состоит из шестидесяти четырёх бинарных элементов.

Инструкция

```
rep 1 ram = [--ar5] with vsum , data, 0;
```

обращается непосредственно в стек, считывает входной вектор, загружает его в `ram`, а пока он проходит по шине данных дублирует его и отправляет на вход **X** матричного устройства умножения.

Инструкция

```
rep 1 with vsum , shift ram, afifo;
```

при помощи ключевого слова `shift` активирует циклический сдвигатель, и вектор данных из `ram` при проходе через него сдвигается на один бит вправо, так что нулевой бит становится шестьдесят третьим, первый нулевым, второй первым, и т.д.

Циклический сдвиг может применяться только к векторам, подаваемым на вход **X** матричного устройства умножения. Разбиение вектора на элементы никак не учитывается, так что младший бит одного элемента после сдвига становится старшим битом его соседа справа. Если такое поведение сдвигателя мешает правильной обработке данных, можно замаскировать отдельные биты. В случае рассматриваемого примера в дополнительном маскировании нет необходимости, поскольку при таком разбиении матрицы (32 строки и 64 столбца) обрабатываются только чётные биты элементов входного вектора, нечётные игнорируются.

Таким образом, входной вектор дважды проходит через матрицу, сначала обрабатываются его чётные биты, а затем, путём сдвига нечётные биты становятся чётными, и также подвергаются обработке.

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step11.asm ../main.cpp libc.lib -
cemu6403.cfg -omain.abs
```

1.16 Урок 12: Использование Векторного Регистра VR

Исходный текст примера, используемого в данном уроке, содержится в файле `step12.asm` в каталоге: `..\Tutorial\Step12`

Пример демонстрирует использование регистра `vr` для добавления одинаковой константы ко всем элементам массива 16-разрядных чисел.

Файл “main.cpp”

```
// функция AddBias объявлена как внешняя с Си-связыванием
extern "C" void AddBias( short* buff, int size, long bias );

long Data[1024];          // массив из 1024 64-разрядных векторов (4096 элементов)

int main()
{
    // цикл начального заполнения массива данных
    Data[0] = 0x0001000100010001;
    for ( int i = 1; i < 1024; i++ )
        Data[i] = Data[i-1] + 0x0001000100010001;

    // вызов функции AddBias
    AddBias( (short*)Data, 4096, 0x0012001200120012 );

    return 0;
}
```

Файл “step12.asm”

```
global _AddBias :label;

data ".data"
  // веса для матрицы, единичная диагональ, данные проходят без изменений
  Weights: long[4] = ( 1l, 1l<<16, 1l<<32, 1l<<48 );
end ".data";

begin ".textAAA"
<_AddBias>
    ar5 = sp - 2;
    push ar0, gr0;
    push ar4, gr4;

    gr4 = [--ar5];          // адрес массива
    gr0 = [--ar5];          // количество 16-разрядных слов в массиве

    nb1 = 80008000h;        // 4 столбца
    sb  = 00030003h;        // 4 строки

    // gr0 преобразуется из кол-ва 16-разрядных слов в кол-во 64-разрядных векторов
    ar4 = Weights with gr0 >>= 2;

    // веса загружаются в рабочую матрицу
    rep 4 wfifo = [ar4++], ftw, wtw;

    vr = [--ar5];           // в регистр vr загружается константный вектор

    // gr0 будет определять количество циклов, где каждый цикл обрабатывает
    // по 32 вектора
    ar0 = gr4 with gr0 >>= 5;
    ar4 = gr4 with gr0--;
<Loop>
    if > delayed goto Loop with gr0--;
       // входные вектора, проходя через матричный умножитель, суммируются с vr
       rep 32 data = [ar0++] with vsum , data, vr;
       rep 32 [ar4++] = afifo;

    pop ar4, gr4;
    pop ar0, gr0;
    return;
end ".textAAA";
```

Комментарии к Примеру

Функция `AddBias` добавляет к каждому 16-разрядному элементу векторов входных данных константу. Операция выполняется на векторном процессоре с использованием регистра `vr`.

Регистр `vr` обладает тем преимуществом, что не требует долгой загрузки и не зависит от количества обрабатываемых векторов. Он задумывался, как регистр, хранящий константу, добавляемую ко всем элементам вектора, полученного в результате взвешенного суммирования. Это, в некотором смысле, альтернатива буферу ram, в котором хранятся одинаковые константные вектора. В отдельных случаях регистр vr позволяет пользователю высвободить основные буфера ВП (ram, data, afifo), заменив их в качестве операнда **Y** в операции взвешенного суммирования. Более подробная информация о регистре vr содержится в пункте 3.3.4 Регистр vr документа **NeuroMatrixÐ NM6403. Описание Языка Ассемблера.**

Инструкция

```
vr = [--ar5];
```

загружает вектор констант из стека в 64-разрядный регистр vr.

Инструкция

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

добавляет вектор констант, хранящийся в vr, к результату взвешенного суммирования.

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step12.asm ../main.cpp libc.lib -
cemu6403.cfg -omain.abs
```

1.17 Урок 13: Работа с Макросами

Исходный текст примера, используемого в данном уроке, содержится в файле step13.asm в каталоге: ..\Tutorial\Step13

Пример демонстрирует описание и подстановку макросов с параметрами, а также использование меток внутри тела макроса.

Функция Copy копирует содержимое одного буфера в другой. В теле функции вызывается макрос, описанный в том же файле и выполняющий непосредственное копирование.

Файл “main.cpp”

```
// функция Copy объявлена внешней с Си-связыванием
extern "C" void Copy( long *Src, long *Dst );
long A[16];                 // массив исходных данных
long B[16];                 // массив результатов

int main()
{
   for (int i=0; i<16; i++)
```

```
      A[i] = 0x0807060504030201*i;

   Copy( A, B );          // вызов функции Copy
   return 0;
}
```

Файл “step13.asm”

```
global _Copy: label;      // объявление глобальной метки _Copy


// описание макроса, предназначенного для копирования одного массива 64-разрядных слов в
// другой. Первый параметр – Исходный массив, второй – массив результата, третий –
количество элементов массива.
macro AAA (Arg1, Arg2, Arg3)
  own Loop: label;        // объявление метки внутри макроса

  gr1 = Arg3;             // в gr1 загружается количество итераций
  gr1--;                  // установка флага для вхождения в цикл
<Loop>
  // инструкция, содержащая команду отложенного перехода
  if > delayed goto Loop with gr1--;
    gr2, ar2  = [Arg1++];
    [Arg2++] =  ar2, gr2;
end  AAA;

begin ".textAAA"
<_Copy>
   ar5 = ar7 – 2;
   push ar0, gr0;
   push ar1, gr1;
   push ar2, gr2;

   ar0 = [--ar5];         // в ar0 адрес исходного массива A
   ar1 = [--ar5];         // в ar1 адрес массива результата B
   AAA(ar0, ar1, 16);     // подстановка макроса AAA

   pop ar2, gr2;
   pop ar1, gr1;
```

```
    pop ar0, gr0;
    return;
end ".textAAA";
```

Комментарии к Примеру

Функция Copy использует макрос AAA.

```
AAA(ar0, ar1, 16);
```

В качестве первого параметра передается массив A, второй параметр – массив B, третий – число 16.

До использования макроса следует описать тело макроса, описание, как правило, располагают между секциями.

```
macro AAA (Arg1, Arg2, Arg3)
…
end  AAA;
```

Описание тела макроса начинается с ключевого слова

`macro`

затем следует имя макроса, далее в круглых скобках задаются параметры.

Заканчивается описание закрывающей скобкой

`end`

после которой указано имя макроса.

При подстановке макроса в тело программы формальные параметры заменяются фактическими значениями.

Невозможно в качестве параметра передать фрагмент программного кода, так как при использовании макроса не осуществляется текстуальная подстановка. Макрос подставляется в программу как законченный фрагмент кода.

Подробнее об использовании макросов см. в приложении A.2.Использование Макросов в Языке Ассемблера.

Если внутри тела макроса требуется использовать метку, то при ее объявлении требуется указать ключевое слово own.

```
own Loop: label;
```

В этом случае при подстановке макроса ассемблер заменит имя метки уникальным. Таким образом, в программе макрос с внутренней меткой можно вызывать более одного раза, при этом ассемблер не выдаст ошибку в связи с повторным определением метки с одним и тем же именем.

В случае если требуется использовать макрос, описанный в библиотеке макросов, необходимо в программе указать перед подстановкой

```
import from M.mlb;
```

где `M.mlb` – библиотека макросов, в которой находится необходимый макрос.

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step13.asm ../main.cpp libc.lib -
cemu6403.cfg -omain.abs
```

1.18 Урок 13а: Создание Библиотеки Макросов

Исходный текст примера, используемого в данном уроке, содержится в файле `step13a.asm` в каталоге: `..\Tutorial\Step13a`

На примере предыдущего урока демонстрируются возможности использования макросов, описанных в макробиблиотеках, способы создания библиотек. Кроме того, в одном из макросов описаны возможности условной компиляции.

Файл “macros1.asm” *// Макрос осуществляет копирование одного массива данных // в другой.*

```
macro AAA (Arg1, Arg2, Arg3)
own Loop: label;
    gr1 = Arg3;
    gr1--;
<Loop>
    if > delayed goto Loop with gr1--;
    gr2, ar2=[Arg1++];
    [Arg2++]=ar2, gr2;
end  AAA;
```

Файл “macros2.asm”

```
macro Push_Pop (Arg1)
.if Arg1 xor 1;              // начало блока услованой компиляции
    push ar0, gr0;
```

```
   push ar1, gr1;
   push ar2, gr2;
.endif;                     // конец блока условной компиляции
.if Arg1;                   // начало блока условной компиляции
   pop ar2, gr2;
   pop ar1, gr1;
   pop ar0, gr0;
.endif;                     // конец блока условной компиляции
end  Push_Pop;
```

Комментарии к Примеру

В макросе `Push_Pop` используются возможности условной компиляции. Если в качестве фактического параметра передается ноль, то регистровые пары сохраняются в стеке; если же передана единица, то происходит восстановление регистровых пар из стека. На этапе компиляции в каждом отдельном случае будет подставлен тот или иной фрагмент программного кода.

Блоки условной компиляции формируются с помощью директивы `.if`. Подробнее см. раздел 2.7.5 Директива .if. документа **NeuroMatrixÐ NM6403. Описание Языка Ассемблера.**

Создание библиотеки макросов

Для создания библиотеки, содержащей описания макросов AAA и `Push_Pop` в командной строке требуется ввести сначала:

```
asm -mmacros.mlb macros1.asm
```

Таким образом, создается библиотека `macros.mlb`, содержащая макрос AAA.

Для добавления в макробиблиотеку макроса `Push_Pop` в командной строке используется следующий вызов:

```
asm -mmacros.mlb -a macros2.asm
```

Более подробное описание макросов и примеров работы с ними приведено в приложении А.2.Использование Макросов в Языке Ассемблера.

После того, как сформирована макробиблиотека, можно осуществлять подстановку макросов в программу.

Файл “main.cpp”

```
// функция Copy объявлена внешней с Си-связыванием
extern "C" void Copy( long *Src, long *Dst );
```

```
long A[16];             // массив исходных данных
long B[16];             // массив результатов

int main()
{
   for (int i=0; i<16; i++)
      A[i] = 0x0807060504030201*i;

   Copy( A, B );        // вызов функции Copy
   return 0;
}
```

Файл “step13a.asm”

```
global _Copy: label;    // объявление глобальной метки _Copy

import from macros.mlb;

begin ".textAAA"
<_Copy>
   ar5 = ar7 – 2;
   Push_Pop(0);         // подстановка макроса (сохранение регистров)

   ar0 = [--ar5];       // в ar0 адрес исходного массива A
   ar1 = [--ar5];       // в ar1 адрес массива результата B
   AAA(ar0, ar1, 16);   // подстановка макроса AAA

   Push_Pop(1);         // подстановка макроса (восстановление регистров)
   return;
end ".textAAA";
```

Компиляция Примера

Для компиляции примера необходимо в командной строке ввести команду:

```
nmcc -g -m -6403 ../Step15a.asm ../main.cpp libc.lib
-cemu6403.cfg -omain.abs –I../Include
```

где `Include` – каталог, в котором содержатся необходимые макробиблиотеки.

2 Методы Оптимизации Программ

В данной главе содержится описание основных подходов, используемых при оптимизации ассемблерного кода, оптимальном размещении различных фрагментов кода и данных по различным областям внешней памяти процессора, распределении ресурсов между параллельно выполняемыми инструкциями.

Система команд процессора NM6403 содержит набор векторных инструкций, выполняющихся в течение нескольких тактов (от одного до тридцати двух). Кроме того, в процессоре существует ряд вычислительных блоков, которые могут работать независимо. Это обстоятельство позволяет процессору выполнять параллельно несколько векторных и скалярных инструкций. Однако для того, чтобы добиться их параллельного выполнения, должны быть созданы определённые условия. Эти условия создаются разработчиком при проектировании программы. Главное правило, которому необходимо следовать:

Важно

Несколько процессорных инструкций выполняются параллельно только в том случае, если они используют непересекающиеся ресурсы процессора.

Это правило относится *только* к последовательности векторных инструкций или к векторной инструкции и выполняемым на её фоне скалярным инструкциям. Сами по себе скалярные инструкции выполняются строго последовательно.

Далее в разделах данной главы приводятся примеры, показывающие, какие факторы необходимо учитывать при написании программы, для того, чтобы заставить векторные инструкции выполняться параллельно.

2.1 Урок 14: Оптимизация Ассемблерного Кода

Исходный текст примера, используемого в данном уроке, содержится в файле `step14.asm` в каталоге: `..\Tutorial\Step14`

Пример содержит описание основных подходов, используемых для оптимизации ассемблерного кода. Оптимизация программы разбирается на примере копирования блока данных при помощи векторного процессора.

Файл “step14.asm”

```
global __main: label;

data ".MyData"
  global A: long[16] = (   0l, 1l, 2l, 3l, 4l, 5hl, 6l, 7l,
                           8l, 9l, 10l, 0Bhl, 0Chl, 13l, 14l, 15l );
end ".MyData";

nobits ".MyData1"
  global C: long[16];
end ".MyData1";

begin ".text"
<__main>
.branch;                // псевдокоманда разрешения параллельного выполнения
                        // векторных инструкций.
   ar0 = A;
   ar4 = C;

   rep 16 data = [ar0++] with data;
   rep 16 [ar4++] = afifo;

   return;
.wait;                  // запрет параллельного выполнения векторных инструкций

end ".text";
```

Комментарии к Примеру

Для того чтобы ассемблерная программа выполнялась более эффективно, при её написании необходимо придерживаться правил, перечисленных ниже:

Размещение Обрабатываемых Массивов на Разных Шинах Данных

Если функция обрабатывает входной массив, а результат обработки записывает в выходной массив, то во многих случаях удобно определять входной и выходной массивы в разных секциях данных. В этом случае появляется возможность расположить входной и выходной буфера в разных банках памяти, и тем самым обеспечить параллельное чтение исходных данных и сохранение результатов вычислений.

В примере к данному уроку два массива A и C объявлены в разных секциях, что позволит разместить их на разных шинах данных (имеются в виду шины, соединяющие процессор с внешней памятью). Информация о том, как определить размещение секций по банкам памяти, содержится ниже в разделе 2.2.Использование Файла Конфигурации на стр. 2-4.

Использование Псевдокоманд .branch и .wait

В язык ассемблера введены две директивы, с помощью которых можно разрешить или запретить параллельное выполнение инструкций.

Директива `.branch` позволяет включить режим параллельного исполнения инструкций процессора. Бит параллельности устанавливается равным единице во всех следующих за псевдокомандой инструкциях процессора до тех пор, пока не будет достигнут конец файла или не будет встречена директива `.wait`. По умолчанию бит параллельности сброшен в 0, то есть параллельное выполнение векторных команд запрещено.

Директива `.wait` устанавливает бит параллельности в 0. В результате каждая векторная инструкция процессора, прежде чем выполниться, будет дожидаться выполнения предыдущей.

Две Группы Адресных Регистров

Процессор NM6403 имеет два устройства генерации адреса, что позволяет одновременно адресоваться по двум адресам на внешней памяти. Каждый из генераторов содержит по четыре адресных регистра, первый – регистры `ar0..ar3`, второй – `ar4..ar7`.

Для того чтобы две векторных инструкции, содержащие обращение к внешней памяти, выполнялись параллельно, необходимо, чтобы они использовали разные адресные генераторы. На программном уровне это означает, что если одна из инструкций для адресации использует регистр `ar2`, то вторая должна использовать какой-либо регистр из диапазона `ar4..ar6` (регистр `ar7` хранит адрес вершины стека). Только в этом случае две векторных инструкции могут выполняться параллельно (с учетом выше названных правил).

Если вернуться к примеру, то для параллельного выполнения инструкций

```
rep 16 data = [ar0++] with data;
rep 16 [ar4++] = afifo;
```

требуется удовлетворить 3 условия:

1) В начале тела функции указать директиву `.branch`

2) Использовать адресные регистры из разных регистровых групп (`ar0` из одной группы, `ar4` из другой).

3) Расположить входной и выходной массивы в разных банках памяти.

Косвенно обе инструкции используют регистр-контейнер `afifo`. Однако он представляет собой двухпортовый буфер, поэтому векторные инструкции могут параллельно записывать и считывать информацию из него.

Примечание

Возможны случаи, когда векторные инструкции не могут выполняться параллельно, например инструкции

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

```
rep 32 [ar4++] = afifo;
```

не выполняются параллельно ни при каких условиях, так как первая инструкция использует afifo на чтение и запись одновременно.

Компиляция Примера

Для компиляции примера под процессор NM6403 необходимо ввести в командной строке:

```
nmcc ../Step14.asm Libc.lib -m -cstep14.cfg
```

2.2 Использование Файла Конфигурации

Файл конфигурации используется редактором связей для того, чтобы разместить коды и данные исполняемой программы по физическим адресам, определённым в конкретной конфигурации вычислительного устройства.

Подробное описание назначения файла конфигурации, его структура и синтаксис описаны в разделе 3.11 Файл Конфигурации документа **Программное Обеспечение NeuroMatrixÐ NM6403. Справочное руководство**.

Ниже приводится файл конфигурации для данного примера, и даются краткие пояснения по его структуре.

Файл Step14.cfg

```
// секция, описывающая доступные адреса физической памяти
MEMORY
{
  local0 : at 0x00000080, len = 0x01ff80;
  global0: at 0xC0000100, len = 0x01ff00;
}

// секция, описывающая сегменты программы и их расположение
SEGMENTS
```

```
{
  local: in local0;
  global: in global0;
}
// секция, описывающая распределение секций программы по сегментам
SECTIONS
{
  // служебные секции
  .init:     in global;
  .fini:     in global;
  .text:     in local;
  .bss :     in local;
  .stack:    in local;
  // пользовательские секции
  .MyData1:  in local;
  .MyData:   in global;
}
```

Любой файл конфигурации состоит из трёх секций:

- секция `MEMORY`, где задаётся конфигурация физической памяти устройства, а именно, её адреса и объём. Здесь необязательно указывать всю имеющуюся физическую память, а только ту его часть, которая будет доступна пользовательской программе. Например, в банках памяти `local0` и `global0` начальные области выделены под служебную информацию, поэтому они исключены из доступной памяти, и редактор связей никогда не разместит там код пользовательской программы.

- секция `SEGMENTS`, в которой определяется принадлежность программных сегментов к тем или иным банкам физической памяти.

- секция `SECTIONS`, где перечислены секции кода и данных программы, и определено, какому сегменту они принадлежат. Секции располагаются в сегменте в том порядке, в котором они перечислены.

Понятие сегмента введено на этапе сборки программы редактором связей. Сегмент используется для упрощения загрузки программы. Секции, располагаясь в сегменте, составляют целый блок данных и кода, который за один раз может быть загружен в память устройства по заданному адресу.

Существует несколько служебных секций, в основном обусловленных требованиями стандарта Си++. Это следующие секции:

`.init` – содержит коды инициализации глобальных статических объектов, её код исполняется до вызова функции `main()`.

`.fini` – деструкторы глобальных статических объектов, её код выполняется сразу после выполнения пользовательской программы.

`.text` – основная секция, содержащая коды программы.

`.data` – секция глобальных инициализированных данных.

`.bss` – секция глобальных неинициализированных данных.

`.stack` – секция, содержащая стек программы.

`.heap` – секция кучи для динамически выделяемой памяти, расположена где указано в файле конфигурации.

`.heap1` – секция кучи для динамически выделяемой памяти, расположена где указано в файле конфигурации.

Если какие-то из пользовательских или служебных секций не упомянуты в файле конфигурации, но в то же время присутствуют в программе, редактор связей автоматически разместит их в первом сегменте, указанном в секции `SEGMENTS`.

2.3 Использование Точного Эмулятора

Для того чтобы определить, насколько эффективно выполняется программа, необходимо выполнить ее на точном эмуляторе `temu`. Для этого в командной строке требуется ввести:

```
temu –S[имя файла1] –B[имя файла2] –L[имя файла3] a.abs
```

где `a.abs` – имя абсолютного файла отлаживаемой программы

Точный эмулятор породит три файла:

- *имя файла1* - файл статистики, где приведены общие сведения о количестве инструкций в программе (в том числе пустых), количестве тактов, за которые происходит выполнение программы, процент пустых инструкций и т.п. информация;
- *имя файла2* - файл, где приведены сведения о том, что происходит на каждом такте на внешних шинах процессора;
- *имя файла3* - листинг выполнения программы, где расписано, какая инструкция выполняется на каждом такте.

Если в команде не указан ни один из ключей (`-s`, `-l` или `–b` с указанием имени файла), то информация будет выведена на экран. Если указан ключ, но без имени файла, информация данного типа порождена не будет.

Таким образом, с помощью этих файлов можно наблюдать за выполнением программы при внесении в нее изменений.

В случае примера `Step14.abs` точный эмулятор может быть запущен со следующими ключами:

```
temu -lstep14.lll -sstep14.sss -b step14.abs
```

Примечание

Рекомендуется опускать имя файла при ключе `–b`*, поскольку порождаемая информация об активности на шинах носит скорее служебный характер, и может помочь в отладке программ только продвинутым пользователям.*

Использовние точного эмулятора temu возможно только для программ, собранных для процессора NM6403 (с ключом -6403 или без ключей). Для программ собранных с ключом -6403 или др. (для более новых процессорв) запусе temu невозможен.

Основной упор в анализе эффективности выполнения программы должен быть сделан на файл листинга, порождаемый при ключе `–l`, поскольку он содержит потактовое описание работы процессора, включая описание всех возможных блокировок по использованию общих ресурсов.

Приложение А

В данном приложении содержится дополнительная информация о количестве отложенных команд, выполняемых при переходах, о макросах и об использовании одного и того же регистра общего назначения в левой и правой частях скалярной инструкции.

А.1 Типы Команд Перехода

Под командами перехода подразумеваются команды, нарушающие естественную последовательность выборки инструкций из памяти.

К командам перехода относятся:

- команды перехода на другой адрес (`goto`, `skip`),
- команды вызова подпрограммы (`call`, `callrel`),
- команды возврата из подпрограммы (`return`),
- команды возврата из прерывания (`ireturn`).

Подобно всем инструкциям процессора, команды перехода делятся на короткие:

```
goto gr0;  //команда не содержит константу и является 32-разрядной
```

и длинные:

```
goto A;    //переход по адресу – содержит 32-разрядную константу.
```

В языке ассемблера введено два типа команд перехода. К первому относятся команды обычного перехода, ко второму - отложенного. Такое разделение введено искусственно, для удобства программирования. От момента выбора команды перехода и до того, как состоится реальный переход, проходит от одного до трёх тактов. За это время процессор успевает выбрать дополнительно одну-три инструкции, следующих непосредственно за инструкцией перехода. Назовём такие инструкции отложенными.

Если программист для выполнения перехода использует инструкцию без ключевого слова `delayed`, например:

```
goto A;
```

то в качестве отложенных инструкций будут автоматически добавлены компилятором пустые инструкции `nul.`

Если же программист захочет осмысленно использовать отложенные инструкции, то в команду перехода должно быть добавлено ключевое слово `delayed`, например:

```
delayed goto A;
```

Отложенные инструкции, помещенные после команды перехода, выполняются до того, как произойдет переход. Они будут исполнены в любом случае, независимо от того, выполнилось условие перехода или нет.

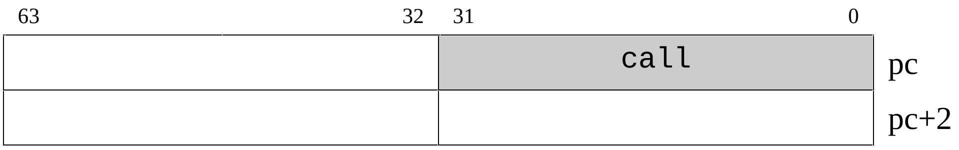
Определение точного количества отложенных инструкций в зависимости от:

- длины,
- расположения в памяти,
- типа команды перехода.

A.1.1. Короткий Переход

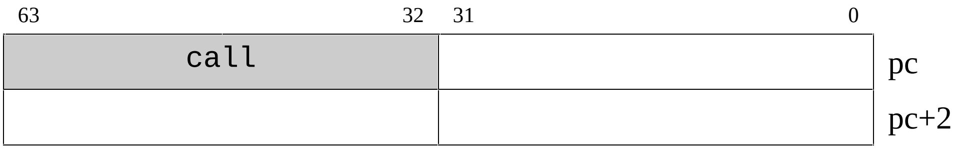
Команда короткого перехода может располагаться по чётному и по нечётному адресу.

Команда Перехода Располагается по Четному Адресу



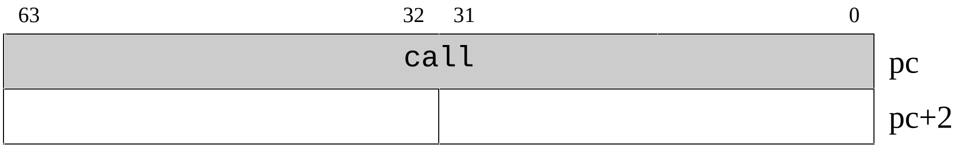
- Команда обычного перехода `call`. В этом случае после этой команды будут добавлены три пустых инструкции `nul`.
- Команда отложенного перехода `delayed call`. Эта команда предполагает, что программист должен поместить вместо трёх `nul` свои отложенные инструкции: либо три коротких, либо одну короткую и одну длинную.

Команда Перехода Располагается по Нечетному Адресу



- `call`. После команды перехода будут добавлены две пустые инструкции `nul`.
- `delayed call`. Требуется поместить после команды либо 2 коротких, либо 1 длинную отложенную инструкцию.

А.1.2. Длинный переход.



Длинные команды всегда располагаются в памяти по четному адресу.

- `call`. После команды перехода будут добавлены <u>две</u> пустые инструкции `nul`.
- `delayed call`. Требуется поместить после команды перехода либо две коротких, либо одну длинную инструкцию.

А.2 Использование Макросов в Языке Ассемблера

Макросы в языке ассемблера для процессора NM6403 – это определяемые разработчиком фрагменты текста программы, которые содержат законченные синтаксически правильные языковые элементы. Макросы могут содержать определения констант и переменных времени компиляции, определения данных, команды и пр. (кроме секций). При их использовании проверка синтаксической правильности происходит дважды: один раз при обработке тела макроса, второй раз - при подстановке, с учётом текущего контекста. Подстановка макроса возможна в любом месте программы, где разрешено использование всех содержащихся в макросе конструкций.
Для описания макроса используется ключевое слово `macro`.

Простейшие Макросы

Пример описания простейшего макроса (описания желательно располагать вне секций):

```
macro AAA()
      gr0 = gr1;
      ar0 = gr1;
end AAA;
```

После описания макроса можно осуществлять его подстановку в программу, для этого достаточно указать его имя с круглыми скобками, означающими, что макрос не имеет параметров:

```
begin ".text"
<__main>
      …
      AAA();
      …
end ".text";
```

Макросы с Параметрами

Ассемблер позволяет описывать макросы с параметрами: после имени макроса в круглых скобках через запятую указываются формальные параметры

```
macro AAA (Arg1, Arg2)
      Arg1 = gr1;
      Arg2 = gr1;
end AAA;
```

Подстановка макроса:

```
AAA(gr0, ar0);
```

При подстановке макроса параметры будут заменены указанными фактическими значениями. В качестве фактических параметров могут передаваться регистры, переменные, метки и константы.

Использование Меток в Макросах

В описании макроса можно объявить метку, используя ключевое слово own. Эта метка будет принадлежать данному макросу:

```
macro AAA (Arg1, Arg2, Arg3)
own A: label:
<A>
      Arg1 = Arg2 + Arg3;
      if > goto A;
end AAA;
```

В этом случае, при компиляции макроса ассемблер заменит имя метки на уникальное, таким образом, в программе подстановку макроса можно осуществлять неоднократно. При этом ассемблер не выдаст сообщение об ошибке в связи с повторным определением метки с одним и тем же именем.

Использование Макросов из Других Файлов

Если требуемый макрос находится в библиотеке макросов (файл, с расширением .mlb), то перед подстановкой макроса компилятору необходимо указать соответствующую библиотеку, например

```
import from A.mlb; // желательно расположить в начале файла,
                   // вне секций.
```

Если точно известно, подстановка какого макроса из макробиблиотеки потребуется, то возможен более конкретный вариант:

```
import AAA from A.mlb;
```

При этом другие макросы из макробиблиотеки A.mlb будут недоступны.

```
begin ".text"
<__main>
…
AAA();
…
end ".text";
```

Возможно, что библиотека макросов находится не в текущем и не в базовом каталоге, тогда перед компиляцией программы в командной строке следует ввести

`–I`*`Path`*, где *Path* – путь к библиотеке макросов.

Создание Библиотек Макросов

Допустим из ассемблерного файла с именем `mac1.asm`, в котором описаны макросы, требуется создать библиотеку макросов. В командной строке нужно ввести:

```
asm -m mac1.asm
```

При этом будет создана библиотека с тем же именем `mac1.mlb`

или

```
asm -mmac.mlb mac1.asm
```

будет создана библиотека с именем `mac.mlb`.

Добавление Макросов в Макробиблиотеку

Для добавления макроса в уже существующую библиотеку наряду с `-m` используется также опция `–a`:

```
asm -mmac.mlb -a mac2.asm
```

Использование ключей `-m –a` означает, что макросы будут добавлены в уже существующую макробиблиотеку.

А.3 Использование Регистров в Обеих Частях Скалярной Инструкции

По принятому соглашению о названиях машинное слово, выполняемое процессором, называется инструкцией. Инструкция состоит из правой и левой частей. В левой части инструкции выполняются адресные команды (поэтому будем называть её командой), в правой – арифметические и логические операции (будем ссылаться на неё, как на операцию).

В левой и правой части скалярной инструкции могут использоваться только регистры общего назначения. В случае если один и тот же регистр используется и в команде, и в операции, возникает вопрос:

какое значение этого регистра используется и какое значение будет иметь регистр по окончании выполнения инструкции?

Основное правило, применяемое при выполнении подобных инструкций:

Правило

Всегда в левой и правой части инструкции используются старые значения регистров - те, которые они имели до начала выполнения инструкции. А по результатам выполнения инструкции регистры получают новые значения.

Далее рассматривается несколько характерных примеров использования одного регистра в обеих частях инструкции.

Инструкция:

```
gr0 = [const] with gr0 = gr1 and gr2;
```

Регистр `gr0` модифицируется как в левой, так и в правой частях инструкции.

Пример:

```
global __main: label;

data ".MyData"
     A: word = 2;
     B: word = 0;
end ".MyData";

begin ".textAAA"
<__main>
   gr1 = [B];
   gr2 = 11111111h;
   gr0 = [A] with gr0 = gr1 and gr2;
   return;
end ".textAAA";
```

В результате выполнения инструкции

```
gr0 = [A] with gr0 = gr1 and gr2;
```

в регистр `gr0` попадет значение 2, то есть приоритет будет отдан команде `gr0 = [A]`.

Такая инструкция является допустимой и компилятор ошибки не выдаст, хотя в программе инструкция может появиться, скорее всего, ошибочно.

Инструкция:

```
gr0 = [const] with gr2 = gr0<<1;
```

В операции используется значение регистра gr0, а в команде происходит его модификация.

Пример:

```
global __main: label;

data ".MyData"
     A: word = 2;
end ".MyData";

begin ".textAAA"
<__main>
   gr0 = 4;
   gr0 = [A] with gr7 = gr0<<1;
   return;
end ".textAAA";
```

После выполнения инструкции в регистре `gr0` будет содержаться число 2, а при вычислении значения регистра `gr7` будет использовано старое значение регистра 4, соответственно в `gr7` окажется число 8.

Инструкция:

```
ar0 = [gr4] with gr4++;
```

В команде происходит чтение из памяти по адресу, хранящемуся в `gr4`, а в операции он изменяет своё значение.

Пример:

```
global __main: label;

data ".MyData"
     A: word[2] = (1,2);
end ".MyData";

begin ".textAAA"
<__main>
   gr4 = A;
   ar0 = [gr4] with gr4++;
   ar1 = [gr4];
   return;
```

```
end ".textAAA";
```

До того, как была выполнена инструкция

```
ar0 = [gr4] with gr4++;
```

первый элемент массива A содержит 1, второй 2. Регистр gr4 указывал на первый элемент массива A. После выполнения инструкции в ar0 загружено значение 1 (первый элемент массива), а gr4 указывает на второй элемент.

Соответственно команда ar1 = [gr4]; загружает в ar1 число 2.

Инструкция:

```
ar0,gr0 = [ar1 = gr1] with gr1 = gr0 + 1;
```

В команде происходит загрузка регистровой пары с изменением значения регистра gr0, а в операции регистр gr0 используется для модификации значения другого регистра.

Пример:

```
global __main: label;

data ".MyData"
     A: long = 3l;
end ".MyData";

begin ".textAAA"
<__main>
   gr1 = A;
   gr0 = 1;
   ar0,gr0 = [ar1 = gr1] with gr1 = gr0 + 1;
   return;
end ".textAAA";
```

Команда ar0,gr0 = [ar1 = gr1] – косвенное чтение из памяти в регистровую пару с предварительной записью в адресный регистр адреса, хранящегося в регистре общего назначения. В ar1 загрузится адрес A, в ar0 попадет младшая часть значения из памяти, находящегося по адресу A (число 3), в gr0 попадет старшая часть (0).

В правой части инструкции

```
ar0,gr0 = [ar1 = gr1] with gr1 = gr0 + 1;
```

для вычисления значения gr1 используется старое значение gr0, равное 1. Таким образом, после выполнения инструкции:

ar0 содержит 3, gr0 содержит 0, gr1 содержит 2.

Инструкция:

```
[ar0] = gr1 with gr1++;
```

Запись значения регистра gr1 с его одновременной модификацией.

Пример:

```
global __main: label;

nobits ".MyData"
     A: word;
end ".MyData";

begin ".textAAA"
<__main>
   ar0 = A;
   gr1 = 1;
   [ar0] = gr1 with gr1++;
   return;
end ".textAAA";
```

Команда [ar0] = gr1 – косвенная запись в память из регистра общего назначения. До выполнения инструкции

```
[ar0] = gr1 with gr1++;
```

ar0 содержал адрес A, gr1 содержал 1. После выполнения в правой части инструкции gr1 увеличится на 1 (т.е. gr1 будет содержать 2), а в левой по адресу A запишется прежнее значение gr1 (т.е. 1).

Инструкция:

```
[ar0++gr0] = gr1 with gr0++;
```

Регистр gr0 используется для модификации адреса доступа к памяти, а в правой части инструкции модифицируется.

Пример:

```
global __main: label;

nobits ".MyData"
     A: word[16];
end ".MyData";

begin ".textAAA"
<__main>
```

```
   ar0 = A;
   gr0 = 1;
   gr1 = 0aah;
   gr2 = 5;
<Loop>
   [ar0++gr0] = gr1 with gr0++;
   gr2--;
   if > goto Loop;
 return;
end ".textAAA";
```

Команда `[ar0++gr0] = gr1` – косвенная запись в память из регистра общего назначения `gr1` с пост-инкрементацией адреса на величину, записанную в регистр общего назначения `gr0`. В правой части инструкции

```
[ar0++gr0] = gr1 with gr0++;
```

значение `gr0` инкрементируется.

После выполнения программы в регистр `gr0` будет загружено значение 6, а массив А будет иметь вид:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
aa	0	0	0	0	aa	0	0	0	aa	0	0	aa	0	aa	aa

Инструкция:

```
push ar0,gr0 with gr0 = not gr0;
```

Значения регистровой пары сохраняются в стеке, и одновременно модифицируется регистр `gr0`.

Пример:

```
global __main: label;

begin ".textAAA"
<__main>
    gr0 = 0;
     push ar0,gr0 with gr0 = not gr0;
     gr1 = gr0;
     pop ar0, gr0;
     gr2 = gr0;
 return;
end ".textAAA";
```

При выполнении инструкции

```
push ar0,gr0 with gr0 = not gr0;
```

в правой части в регистр gr0 загружается значение 0FFFFFFFFh, в левой сохраняется в стек регистровая пара, при этом gr0 содержит прежнее значение, равное 0.

Таким образом, после выполнения рассматриваемой инструкции

регистр gr0 содержит 0,

регистр gr1 содержит 0FFFFFFFFh,

регистр gr2 содержит 0.

Инструкция:

```
pop ar0, gr0 with gr1 = gr0;
```

Значение регистровой пары восстанавливается из стека, а в правой части значение gr0 используется для модификации другого регистра.

Пример:

```
global __main: label;

begin ".textAAA"
<__main>
    gr0 = 2;
     push ar0,gr0;
     gr0 = 1;
     pop ar0, gr0 with gr1 = gr0;
     gr2 = gr0;
 return;
end ".textAAA";
```

При выполнении инструкции

```
pop ar0, gr0 with gr1 = gr0;
```

в левой части восстанавливается регистровая пара из стека (gr0 содержит 2), в правой при вычислении gr1 используется прежнее значение gr0, равное 1.

Соответственно, после выполнения рассматриваемой инструкции

регистр gr0 содержит 2,

регистр gr1 содержит 1,

регистр gr2 содержит 2.

Инструкция:

```
ar0,gr0 = ar2 with gr2 = -gr0;
```

Значение адресного регистра дублируется в регистровую пару, а в правой части `gr0` используется для модификации другого регистра.

Пример:

```
global __main: label;

begin ".textAAA"
<__main>
     gr0 = 2;
     ar2 = 11111111h;
    ar0,gr0 = ar2 with gr2 = -gr0;
 return;
end ".textAAA";
```

При выполнении инструкции

```
ar0,gr0 = ar2 with gr2 = -gr0;
```

в ее левой части в регистровую пару загружается значение из `ar2` (в `ar0` попадает `11111111h`, и в `gr0` попадает `11111111h`), в правой части при вычислении значения `gr2` используется прежнее значение регистра `gr0`, равное 2.

Таким образом, после выполнения рассматриваемой инструкции

регистр `gr0` содержит `11111111h`,

регистр `gr2` содержит `0FFFFFFFEh` (т.е. –2).

Инструкция:

```
goto gr0 with gr0 = gr0 + =gr1;
```

Пример:

```
global __main: label;
global Loop: label;
global L1: label;
begin ".textAAA"
<__main>
    gr2 = Loop;
    gr1 = L1;
    gr0 = gr2 with gr1-= gr2;// gr1 содержит разницу
                             // адресов  команд, на которые
                             // указывают Loop и L1
    goto gr0 with gr0 += gr1;
<Loop>
    gr7 = 1;
    return;
<L1>
    gr7 = 2;
    return;
end ".textAAA";
```

В регистр gr0 загружается адрес команды, на которую указывает метка Loop.

При выполнении программы в правой части инструкции

```
goto gr0 with gr0 += gr1;
```

модифицируется регистр gr0, в нем будет содержаться адрес команды в памяти, на который указывает метка L1.

В левой части содержится переход по абсолютному адресу, задаваемому регистром gr0.

Поскольку по общему для всех инструкций правилу для команды перехода будет использовано прежнее значение gr0, то в результате выполнения рассматриваемой инструкции будет осуществлен переход на метку Loop.

В результате функция main вернет значение 1, а не 2.(Возвращаемое значение передается в регистре gr7).

Модуль Ð

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: info@module.ru
WWW: http://www.module.ru

НТЦ Модуль оставляет за собой право производить изменения как в описании, так и в самом продукте без дополнительных уведомлений. НТЦ Модуль не несет ответственности за любой ущерб, причиненный использованием информации в данном описании, ошибками или недосказанностью в описании, а также путем неправильного использования продукта.

Напечатано в России Дата издания: 12/27/2017