

# Первые шаги в разработке программ для NeuroMatrix

# Содержание

Назначение .....	1
Требования и зависимости к ПО .....	1
1. Работа со скалярным RISC процессором .....	1
1.1. Сборка, запуск и отладка программы на C .....	1
1.2. Печать и отладка через printf .....	14
1.3. Простейшая Программа на Языке Ассемблера .....	19
1.4. Доступ к Памяти .....	21
1.5. Организация Циклов .....	23
1.6. Оптимизация Выполнения Цикла .....	24
1.7. Копирование Массива Данных на Скалярном Процессоре .....	26
2. Работа с целочисленным векторным сопроцессором .....	28
2.1. Копирование Массива Данных на Векторном Процессоре .....	28
2.2. Арифметические Операции на Векторном АЛУ .....	31
2.3. Операция Взвешенного Суммирования .....	33
2.4. Вызов Ассемблерных Функций из Си++ .....	36
2.5. Передаваемые и Возвращаемые Значения Типа LONG .....	37
2.6. Операции Логической Активации и Маскирования на ВП .....	40
2.7. Операция Арифметической Активации .....	44
2.8. Использование Циклического Сдвигателя на ВП .....	49
2.9. Использование Векторного Регистра VR .....	52
2.10. Создание Библиотеки Макросов .....	54
2.11. Методы Оптимизации Программ .....	56
3. Работа с векторным процессором на плавающей точке .....	59
3.1. Простейшая программа .....	60
3.2. Краткое введение в архитектуру .....	63
3.3. Поэлементная операция $a*b+c$ над массивом данных .....	65
3.4. Поэлементная операция $a*b+c$ над массивом данных. SIMD - оптимизация .....	68
3.5. Поэлементная операция $a*b+c$ над массивом данных. Оптимизация по памяти .....	71
3.6. Поэлементная операция $a*b+c$ над массивом данных. Анализ производительности .....	74
3.7. Поэлементная операция $ a+b $ над массивом данных. Оптимизация по памяти .....	77
3.8. Поэлементная операция $ a+b $ над массивом данных. Оптимизация по вычислительным ячейкам .....	80
3.9. Поэлементная операция $ a+b $ над массивом данных. Оптимизация по потокам .....	83

# Назначение

Данное руководство пошагово описывает основные принципы создания программ на языке ассемблера и С для процессоров семейства NeuroMatrix. Показывается как разработать программу, задействуя скалярный и векторный процессор. Приводятся основные принципы оптимизации программ с учётом архитектуры процессора, структуры и состава окружающей его периферии.

Каждый шаг представляет собой проект с исходными текстами и сопровождается подробным описанием. Проекты могут быть собраны и запущены как на отладочной плате, так и на эмуляторе под ОС Windows/Linux.

Руководство является практическим и состоит из наборов уроков для целочисленного сопроцессора, сопроцессора с плавающей точкой и RISC ядра. К каждый уроку прилагается исходные коды, которые можно собрать и запустить в режиме пошаговой отладки на имеющейся отладочной плате. Поддерживаются следующие платы : МЦ121.01 МЦ51.03 MB77.07 MB127.05 . Запуск примеров также можно осуществить с помощью эмулятора QEMU.

## Требования и зависимости к ПО

- Для сборки примеров должен быть установлен [Neuro Matrix SDK]
- Для работы с платами должны быть установлены соответствующие Библиотеки Загрузки и обмена (БЗИО);
- Для сборки/запуска примеров используется утилита `make` Под ОС Windows рекомендуется использовать [make v3.81](<http://gnuwin32.sourceforge.net/packages/make.htm>).

## 1. Работа со скалярным RISC процессором

### 1.1. Сборка, запуск и отладка программы на С

В данном разделе описывается порядок сборки, запуска и отладки простейшего примера на языке С. Приводится два варианта сборки программы: из командной строки и в среде разработки VS CODE. Рассматривается сборка и запуск для двух целей: для симулятора QEMU и платы МЦ121.01.

Управление сборкой и запуском программы осуществляется с помощью прилагаемых сборочных файлов `Makefile` из соответствующих целям подпапок: `qemu` и `mc12101`.

Исходный текст примера, используемого в данном уроке, содержится в файле `step1.S` в каталоге: `\steps-risc\step00`.

Файл “`main.cpp`”

```

#include <time.h>
#include <stdio.h>
#include <string.h>
#define      SIZE     400
unsigned int in[SIZE] __attribute__ ((section (".data imu1")));
unsigned int out[SIZE] __attribute__ ((section (".data imu2")));
extern "C" {
    int clock_gettime(clockid_t, struct timespec *);
};

int main()
{
    clock_t t1, t2;
    //struct timespec tms1, tms2;
    volatile int a=3;
    volatile int b=4;
    t1=clock();
    volatile int c=b-a;
    memcpy(in,out,SIZE);
    t2=clock();
    printf("NMC Hello %d %d\n", c, int(t2-t1));
    return 777;
}

```

### 1.1.1. Сборка и запуск примера на QEMU-эммуляторе

Для сборки примера под QEMU необходимо перейти в папку **qemu**

```
cd steps-risc/step00/qemu
```

#### Сборка программы для QEMU-эммулятора из командной строки

Для компиляции данного примера под эмулятор необходимо в командной строке из папки **qemu** ввести команду:

```
nmc-gcc -o test.abs -O2 -Wall -mnmvc4 -g -Wl,-Map=test.map -Wl,-Tqemu.lds ../main.cpp -lc
```

Расшифровка строки сборки:

- nmc-gcc - компилятор gcc из состава SDK для NeuroMatrix;
- -otest.abs - задает имя выходного исполняемого файла **test.abs**. Если в командной строке не указано имя выходного файла, то его имя будет сформировано по имени первого встреченного файла;
- -O2 - максимальный уровень оптимизации;

- -Wall -включение всех предупреждений;
- -mnmc4 - выбора платформы **nmc4**;
- -g -включает отладочную информацию в выходной файл;
- -Wl,-Map=test.map - ключ линкеру, который 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.;
- -Wl,-Tqemu.lds - ключ линкеру использовать файл конфигурации;
- ./main.cpp - исходный файл ;
- -lc - подключение библиотеки **libc.a**. 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. Библиотека **libc.a** добавляется автоматически, если при компиляции в командной строке встречается файл с расширением **\*.cpp**. Если же программа собрана только из ассемблерных файлов, библиотека **libc.a** должна быть добавлена вручную.

В дальнейшем для удобства компиляции и запуска будут использоваться сборочные Makefile-скрипты для трансляции которых будет вызываться утилита **make**.

Так сборка примера сразу в двух версиях: Release и Debug осуществляется командой **make** из папки с примером. Результат:

```
c:\Module\MC12101\examples\first-steps\steps-risc\step00\qemu>make

----- build release -----
nmc-gcc -o test.abs -O2 -Wall -mnmc4-fixed -g -Wl,-Map=test.map -Wl,-Tqemu.lds
-I"/include" -I../../include -I.. -L"/lib"    ./main.cpp -lc
nmc-objdump -D test.abs > .S.txt
----- build debug -----
nmc-gcc -o testd.abs -O0 -Wall -mnmc4-fixed -g -Wl,-Map=testd.map -Wl,-Tqemu.lds
-I"/include" -I../../include -I.. -L"/lib"   ./main.cpp -lc
nmc-objdump -D testd.abs > .Sd.txt
```

В папке появятся Release и Debug исполняемые файлы: **main.abs** и **maind.abs**.

По окончании сборки с помощью **nmc-objdump** формируется дизассемблированные дампы программ.

### **Запуск Release-программы на QEMU-эмулаторе из командной строки**

Прямой (без отладки) запуск примера (Release версии) осуществляется командой **make run**. Результат:

```
c:\Module\MC12101\examples\first-steps\steps-risc\step00\qemu>make run  
nmc-qemu test.abs
```

```
NMC Hello 1 0  
Simulation error! Return value 777  
make: *** [run] Error 1
```

### Запуск Debug-программы на QEMU-эммуляторе через GDB-отладчик из командной строки

В первом окне консоли запустить эмулятор QEMU на порте :10000 командой `make run`  
Результат:

```
c:\Module\MC12101\examples\first-steps\steps-risc\step00\qemu>make run  
nmc-qemu -g 10000 testd.abs
```

nmc-qemu переходит в режим ожидания подключения GDB-отладчика через 10000.

Во втором окне консоли запускаем nmc-gdb отладчик (с авто подключением через порт :10000) командой `make gdb` Результат:

```
C:\git\nmc-first-steps\build_mc12101\first-steps\steps-risc\step00\qemu>make gdb  
nmc-gdb -iex "set tcp connect-timeout unlimited" -ex "target remote :10000" testd.abs  
  
GNU gdb (NMC SDK Binutils ) 7.8  
Copyright (C) 2014 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "--host=x86_64-w64-mingw32 --target=nm-unknown-elf".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"..."  
Reading symbols from testd.abs...done.  
Remote debugging using :10000  
0x0000020c in loadStackAddr ()  
(gdb)
```

Далее отлаживаемся по шагам, например:

```
(gdb) b main
Breakpoint 1 at 0x27e: file ../main.cpp, line 15.
(gdb) c
Continuing.
```

```
Breakpoint 1, main () at ../main.cpp:15
15          volatile int a=3;
(gdb) n
16          volatile int b=4;
(gdb) n
17          t1=clock();
(gdb) n
18          volatile int c=b-a;
(gdb) n
19          memcpy(in,out,SIZE);
(gdb) p c
$1 = 1
(gdb) c
Continuing.
```

```
Program received signal SIG_NM_SUSPEND, Suspend of NMC simulation.
0x00000212 in int_stop ()
(gdb) k
Kill the program being debugged? (y or n) y
(gdb)
```

## Сборка Release и Debug программы под QEMU-эмулятор из VS CODE

Открываем проект с примером в vscode из папки с примером

```
cd \steps-risc\step00
code .
```

Через меню **\Terminal\Run Build Task...** либо через комбинацию клавиш **Ctrl+Shift+B** в выпадающем списке выбрать сборку под плату: **build qemu** Результат: в окне Terminal появится лог сборки Debug и Release версии:

```
Executing task: make qemu

c:/Module/tools/make -C qemu
make[1]: Entering directory `C:/Module/MC12101/examples/first-steps/steps-
risc/step00/qemu'
----- build release -----
nmc-gcc -otest.abs -O2 -Wall -mnmvc4-fixed -g -Wl,-Map=test.map -Wl,-Tqemu.lds
-I"/include" -I../../include -I.. -L"/lib" ../main.cpp -lc
nmc-objdump -D test.abs > .S.txt
----- build debug -----
nmc-gcc -otestd.abs -O0 -Wall -mnmvc4-fixed -g -Wl,-Map=testd.map -Wl,-Tqemu.lds
-I"/include" -I../../include -I.. -L"/lib" ../main.cpp -lc
nmc-objdump -D test.abs > .Sd.txt
make[1]: Leaving directory `C:/Module/MC12101/examples/first-steps/steps-
risc/step00/qemu'
```

### Запуск Release-программы на QEMU-эмуляторе из VS CODE

Через меню `\Terminal\Run Task...` в выпадающем списке выбрать запуск на плате : `qemu run`, а затем `Continue without scanning the task output`. Будет вызван `make qrun` с целью `qrun` файла `step00/Makefile`, который перевызовет `make -C qemu run` с целью `run` файла `step00/qemu/Makefile`, который уже вызовет `nmc-qemu` с Release-версий программы `test.abs`. Результат: в окне `Terminal` появится лог запуска:

```
Executing task: make qrun

c:/Module/tools/make -C qemu run
make[1]: Entering directory `C:/Module/MC12101/examples/first-steps/steps-
risc/step00/qemu'
nmc-qemu test.abs
NMC Hello 1 0

Simulation error! Return value 777
make[1]: *** [run] Error 1
make[1]: Leaving directory `C:/Module/MC12101/examples/first-steps/steps-
risc/step00/qemu'
make: *** [qrun] Error 2
```

### Запуск Debug-программы на QEMU-эмуляторе в режиме отладки из VS CODE

Слева на вертикальной панели инструментов нажать на значок отладки "Run and Debug (Ctrl+Shift+D)" (в форме треугольника с жучком). В появившейся панели отладки "RUN AND DEBUG" открыть выпадающей список и выбрать цель для gdb-отладчика `nmc-qemu`. Далее нажать на значок "Start Debugging (F5)" (в форме зеленого треугольника) или клавишу F5. Появится приглашение для выбора порта подключения gdb (по умолчанию 10000). Подтвердить порт 10000 клавишей "Enter". Результат: откроется отдельное консольное окно с запуском примера в эмуляторе:

```
C:/Module/tools/make -C qemu rund
make[1]: Entering directory `C:/module/MC12101/examples/first-steps/steps-
risc/step00/qemu'
nmc-qemu -g 10000 testd.abs
```

Дождаться (~ 10-20 секунд) появления панели управления отладкой.

В редакторе vscode перейти к файлу `main.cpp` и установить точку останова в функции `main`, например на `return` через `Menu\Run\Toggle Breakpoint (F9)` или клавишу `F9`. В панели управления отладкой нажать значок `Continue (F5)` или клавишу `F5`. Программа начнет выполняться, а курсор отладки остановится в точке останова. В окне запуска программы появится результат работы программы.

```
C:/Module/tools/make -C qemu rund
make[1]: Entering directory `C:/module/MC12101/examples/first-steps/steps-
risc/step00/qemu'
----- build debug -----
nmc-gcc -otestd.abs -O2 -Wall -mnmvc4-fixed -g -Wl,-Map=testd.map -Wl,-Tqemu.lds
-I"/include" -I../../include -I.. -L"/lib" ../main.cpp -lc
nmc-qemu -g 10000 testd.abs
NMC Hello 1 0
```

Продолжить и завершить выполнение программы клавишей `Continue (F5)` Эмулятор вернет код ошибки 777 в соответствии с кодом возврата из функции `main`:

```
Simulation error! Return value 777
```

## 1.1.2. Сборка и запуск примера на плате

Ниже рассматривается порядок сборки примера для платы МЦ121.01 в Release и Debug режимах в консольном окне и в среде разработке VS CODE

### Сборка примера для платы из командной строки

В консольном окне перейти в папку с примером

```
C:\Module\MC12101\examples\first-steps> cd steps-risc\step00\mc12101
C:\Module\MC12101\examples\first-steps\steps-risc\step00\mc12101>
```

Сборка программы (Release и Debug версии) для платы МЦ121.01 осуществляется командой `make` из папки `mc12101` примера. Результат:

```
c:\Module\MC12101\examples\first-steps\steps-risc\step00\mc12101>make

----- build release -----
nmc-gcc -otest.abs -O2 -Wall -mnmc4-float -g -Wl,-Map=test.map -Wl,-Tmc12101
-nmpu0.lds -I"C:\Module\MC12101/include" -I../../include -I..
-L"C:\Module\MC12101/lib" ../main.cpp -lnm6407int -Wl,--whole-archive
-lnm6407_io_nmc -lmc12101load_nm -Wl,--no-whole-archive
nmc-objdump -D test.abs > .S.txt
----- build debug -----
nmc-gcc -otestd.abs -O0 -Wall -mnmc4-float -g -Wl,-Map=testd.map -Wl,-Tmc12101
-nmpu0.lds -I"C:\Module\MC12101/include" -I../../include -I..
-L"C:\Module\MC12101/lib" ../main.cpp -lnm6407int -Wl,--whole-archive -lnm6407_io_nmc
-lmc12101load_nm -lmc12101_stub_nmc4_float -Wl,--no-whole-archive
nmc-objdump -D test.abs > .Sd.txt
```

В папке **mc12101** появятся Release и Debug исполняемые файлы: **main.abs** и **maind.abs**.

Release версия предназначена для непосредственного исполнения **test.abs** на плате. Debug версия предназначена для исполнения **testd.abs** на плате в режиме отладки через GDB (программу **nmc-gdb**).

### Запуск Release-программы на плате из командной строки

Запуск примера (Release версии) осуществляется командой **make run**. Результат:

```
c:\Module\MC12101\examples\first-steps\steps-risc\step00\mc12101>make run
mc12101run -p -R -a0 -v test.abs

Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
NMC Hello 1 10250
test.abs :: Core 0 return 777 = 0x309.
make: *** [run] Error 777
```

### Запуск Debug-программы на плате через GDB-отладчик из командной строки

В первом окне консоли запустить пример в режиме отладки командой **make rund**. Результат:

```
c:\Module\MC12101\examples\first-steps\steps-risc\step00\mc12101>make run  
mc12101run -p -R -a0 -v testd.abs
```

```
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.  
Performing reset...  
Done.  
Firmware v6.1  
Start user program on core 0...
```

Во втором окне консоли запустить монитор командой `make monitor`. Результат:

```
C:\Module\MC12101\examples\first-steps\steps-risc\step00\mc12101>make monitor  
mc12101_gdb_monitor 0 0 5555  
  
Logger name pattern: dbg-obj_[board-num_core-num]  
Enter 'q', 'x' or 'exit' to quit.  
15:56:00.140 [tcp_0_0] server listening on port 5555
```

Формат [tcp\_X\_Y] означает подключение монитора к ядру Y кластера X . Например

```
21:40:25.373 [tcp_0_0] server listening on port 5555  
21:40:25.373 [tcp_0_1] server listening on port 5556
```

[tcp\_0\_0] - означает, что для подключения gdb-отладчика к ядру 0 (float) открыт порт 5555.  
[tcp\_0\_1] - означает, что для подключения gdb-отладчика к ядру 1 (fixed) открыт порт 5556.

В третьей консоли запустить GDB-отладчик командой `make gdb` Результат:

```
C:\git\nmc-first-steps\build_mc12101\first-steps\steps-risc\step00\mc12101>make gdb  
nmc-gdb -ex "target remote :5555" testd.abs  
  
GNU gdb (NMC SDK Binutils ) 7.8  
Copyright (C) 2014 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "--host=x86_64-w64-mingw32 --target=nm-unknown-elf".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"..."  
Reading symbols from testd.abs...done.  
Remote debugging using :5555  
0x2000018e in gdb_bp_template ()  
(gdb)
```

Далее устанавливаем точку останову на функции `main` и выполняем программу в пошаговом режиме с помощью GDB-команд, например:

```
(gdb) b main  
Breakpoint 1 at 0x2b4: file main.cpp, line 19.  
(gdb) c  
Continuing.  
  
Breakpoint 1, main () at main.cpp:19  
19      {  
(gdb) n  
22          for(int i = 0; i < SIZE; i++) {  
(gdb) n  
23              src[i].re = sinf(i);  
(gdb) display src  
1: src = (nm8s *) 0x8180  
(gdb) display ((int*)src)[0]  
9: ((int*)src)[0] = 0  
(gdb) p $gr7  
$1 = 2864434397  
(gdb) c  
Continuing.  
[Inferior 1 (Remote target) exited normally]  
(gdb) q
```

## Сборка Debug и Release программы под плату из VS CODE

Открываем проект с примером в vscode

```
cd \steps-risc\step00  
code .
```

Через меню **\Terminal\Run Build Task...** либо через комбинацию клавиш **Ctrl+Shift+B** в выпадающем списке выбрать сборку под плату: **build board** Результат: в окне Terminal появится лог сборки:

```
Executing task: make board  
  
c:/Module/tools/make -C mc12101  
make[1]: Entering directory 'C:/module/MC12101/examples/first-steps/steps-  
risc/step00/mc12101'  
----- build release -----  
nmc-gcc -otest.abs -O2 -Wall -mnm4-float -g -Wl,-Map=test.map -Wl,-Tmc12101  
-nmpu0.lds -I"C:/Module/MC12101/include" -I../../include -I..  
-L"C:/Module/MC12101/lib" ../main.cpp -lnm6407int -Wl,--whole-archive  
-lnm6407_io_nmc -lmc12101load_nm -Wl,--no-whole-archive  
nmc-objdump -D test.abs > .S.txt  
----- build debug -----  
nmc-gcc -otestd.abs -O2 -Wall -mnm4-float -g -Wl,-Map=testd.map -Wl,-Tmc12101  
-nmpu0.lds -I"C:/Module/MC12101/include" -I../../include -I..  
-L"C:/Module/MC12101/lib" ../main.cpp -lnm6407int -Wl,--whole-archive -lnm6407_io_nmc  
-lmc12101load_nm -lmc12101_stub_nmc4_float -Wl,--no-whole-archive  
make[1]: Leaving directory 'C:/module/MC12101/examples/first-steps/steps-  
risc/step00/mc12101'
```

## Запуск Release-программы на плате из VS CODE

Через меню **\Terminal\Run Task...** в выпадающем списке выбрать запуск на плате : **board run**, а затем **Continue without scanning the task output** Результат: в окне Terminal появится лог запуска:

```
Executing task: make run
```

```
c:/Module/tools/make -C mc12101 run
make[1]: Entering directory `C:/module/MC12101/examples/first-steps/steps-
risc/step00/mc12101'
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
NMC Hello 1 10232
test.abs :: Core 0 return 777 = 0x309.
make[1]: *** [run] Error 777
make[1]: Leaving directory `C:/module/MC12101/examples/first-steps/steps-
risc/step00/mc12101'
make: *** [run] Error 2
```

### Запуск Debug-программы на плате в режиме отладки из VS CODE

В отдельном окне консоли запустить монитор командой `make monitor` Результат:

```
C:/Module/MC12101/first-steps/steps-risc/step00/mc12101> make monitor

make[1]: Entering directory `C:/Module/MC12101/first-steps/steps-risc/step00/mc12101'
nm_gdb_monitor_mc12101 0 0 5555

Logger name pattern: dbg-obj_[board-num_core-num]
Enter 'q', 'x' or 'exit' to quit.
16:40:48.029 [tcp_0_0] server listening on port 5555
```

[tcp\_0\_0] - означает, что для подключения отладчика к ядру 0 (float) используется порт 5555.

Слева на вертикальной панели инструментов нажать на значок отладки "Run and Debug (Ctrl+Shift+D)" (в форме треугольника с жучком). В появившейся панели отладки "RUN AND DEBUG" открыть выпадающий список и выбрать цель для gdb-отладчика `board`. Далее нажать на значок "Start Debugging (F5)" (в форме зеленого треугольника) или клавишу F5. Появится приглашение для выбора порта подключения gdb (по умолчанию 5555). Подтвердить порт 5555 и нажимаем "Enter". Результат: появится отдельное консольное окно с запуском примера на плате:

```
C:/Module/tools/make -C mc12101 rund
make[1]: Entering directory `C:/module/MC12101/examples/first-steps/steps-
risc/step00/mc12101'
mc12101run -p -R -a0 -v testd.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
```

В консольном окне с монитором обновится статус подключения gdb клиента:

```
C:/Module/tools/make -C mc12101 monitor
make[1]: Entering directory `C:/git/nmc-first-steps/build_mc12101/first-steps/steps-
risc/step00/mc12101'
nm_gdb_monitor_mc12101 0 0 5555
Logger name pattern: dbg-obj_[board-num_core-num]
Enter 'q', 'x' or 'exit' to quit.
16:40:48.029 [tcp_0_0] server listening on port 5555
16:42:39.423 [tcp_0_0] server accept the client...
16:42:40.203 [tgt_0_0] target init successfule...
```

В редакторе vscode перейти к файлу `main.cpp` и установить точку останова в функции `main` (например на `return`) через [Menu/Run/Toggle Breakpoint \(F9\)](#) или клавишей `F9`.

Примечание: Если по `(F9)` точка останова не появляется следует проверить настройки vscode: войти в меню [File/Preferences/Settings](#) или нажать `Ctrl+,`, В строке поиска настроек набрать "Debug", разрешить "Allow Breakpoints Everywhere".

На появившейся панели отладки нажать значок [Continue \(F5\)](#) или нажать клавишу `F5`. Программа запустится , а курсор отладки должен переместится на точку останова. В окне запуска программы появится результат работы программы.

```
C:/Module/tools/make -C mc12101 rund
make[1]: Entering directory `C:/module/MC12101/examples/first-steps/steps-
risc/step00/mc12101'
mc12101run -p -R -a0 -v testd.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
NMC Hello 1 0
```

Продолжить и завершить выполнение программы клавишой [Continue \(F5\)](#)

Окно с запуском программы закроется. В окне с монитором обновится статус готовности к

новому подключению.

```
15:10:16.019 [tcp_0_0] client disconnected  
15:10:16.019 [tgt_0_0] target closed.  
15:10:16.019 [tcp_0_0] server listening on port 5555
```

## 1.2. Печать и отладка через `printf`

Исходный текст примера, используемого в данном уроке, содержится в каталоге: `\steps-risc\step00a_printf`.

Помимо средств отладки QEMU и VS CODE часто бывает проще и удобнее пользоваться обычной функцией `printf`.

### 1.2.1. Настройка печати

Для обеспечения вывода на консоль функциями `printf` на языке C/C++ предоставляются специальные библиотеки, так например для процессора nm6407: `libnm6407_io_nmc.a`. При подключении библиотеки требуется ее заключать внутри ключей принудительной линковки: `-Wl,--whole-archive -lnm6407_io_nmc -Wl,--no-whole-archive`

В конфигурационном `.lds` файле требуется объявление специальных секций `.rpc_services`:

```
*****rpc . important ALIGN(0x8) in both *****  
.rpc_services.bss : ALIGN(0x8)  
{  
    *(.rpc_services.bss);  
} > EMI  
.rpc_services : ALIGN(0x8)  
{  

```

После данных конфигураций вывод средствами `printf` станет доступным.



При работе с программой-загрузчиком `mc12101run` также требуется указывать ключ `-p`, разрешающий вывод на консоль.

### 1.2.2. Вывод переменных и регистров

В ряде случаев при отладке ассемблерного кода требуется вывод непосредственно из ассемблера. Для удобства вызова функций `printf` можно подключить набор макросов с ограниченным функционалом (с помощью файла `printfx.hs`), позволяющий выводить форматированные строки с одним или двумя числовыми аргументами. Например:

```

#include "printx.hs"

.data
    tmp: .quad 0xB BBBB BBBB AAAAAAAA

    mtr: .quad 0x0706050403020100, 0x1716151413121110, \
              0x2726252423222120, 0xFFFFFFFFFFFFFF

    mtr32f: .float 0.1, 0.2, 123.3, -0.4
    mtr64f: .double 0.1, 0.2, 123.3, -0.4

.global _asm_print // объявление глобальной метки
.text // начало секции кода
_asm_print: // определение глобальной метки
    push ar4,gr4;
    ar4 = tmp;
    PRINTF("Hello\n")
    PRINTF1("=%x\n",gr4)
    PRINTF1("=%x\n",[ar4])
    PRINTF1("=%x\n",[tmp])
    PRINTF1("=%x\n",tmp)
    PRINTF2("=%x %x\n",ar4,gr4)
    PRINTF2("=%x %x\n",gr4,ar4)
    PRINTF2("=%x %x\n",gr4,gr4)
    PRINTF2("=%x %x\n",ar4,ar4)
    PRINTFL("=%lX\n",[ar4])
    PRINTF2("=%x %x\n",[ar4++],[ar4++])
    ...

```

Выполнение данного кода приведет к следующему выводу:

```

Hello
=d44
=aaaaaaaa
=aaaaaaaa
=30000
=30000 d44
=d44 30000
=d44 d44
=30000 30000
=BBBB BBBB AAAAAAAA
=aaaaaaaa bbbbbbbb

```

Приведенные макросы позволяют вывести на печать обычную строку (`PRINTF`), значение одного аргумента (`PRINTF1`), двух 32-разрядных аргументов (`PRINTF2`) или одного 64-разрядного (`PRINTFL`). Так можно вывести в форматированном виде: значения регистров, содержимое памяти по регистрам, значение констант , значение меток или содержимое памяти по меткам (переменные). Для вызова макросов достаточно включить в ассемблерный файл

`#include "printfx.hs"` и указать путь к файлу в строке сборке.



Для корректного препроцессирования строки `#include "printfx.hs"` ассемблерный файл должен иметь расширение `.S` (с заглавной буквой).

### 1.2.3. Вывод матриц

Для отображения массивов в матричном виде в предоставляется набор макросов, позволяющий осуществить вывод фрагмента матрицы упакованных элементов различных разрядностей в форматированном виде:

Печать целых чисел со знаком:

```
DUMP_8S( Format, mtr, height, width, stride, mode)
DUMP_16S( Format, mtr, height, width, stride, mode)
DUMP_32S( Format, mtr, height, width, stride, mode)
DUMP_64S( Format, mtr, height, width, stride, mode)
```

Печать целых чисел без знака:

```
DUMP_8U( Format, mtr, height, width, stride, mode)
DUMP_16U( Format, mtr, height, width, stride, mode)
DUMP_32U( Format, mtr, height, width, stride, mode)
DUMP_64U( Format, mtr, height, width, stride, mode)
```

Печать чисел с плавающей точкой:

```
DUMP_32F( Format, mtr, height, width, stride, mode)
DUMP_64F( Format, mtr, height, width, stride, mode)
```

- `Format` - строка форматирования одного элемента со стандартным синтаксисом `printf`.
- `mtr` - любой регистр или метка, указывающая на начало матрицы
- `height` - высота матрицы в элементах (8,16,32 или 64-разрядных) в виде константы или любого регистра кроме `arg5,arg7`
- `width` - ширина матрицы в виде константы или любого регистра кроме `arg5,arg7`
- `stride` - смещение в элементах при переходе от начала одной строки к следующей в виде константы или любого регистра кроме `arg5,arg7`
- `mode` - задает вид вспомогательного столбца: 0 - отсутствует вывод, 1 - выводятся адреса строк, 2 - выводятся номера строк

Пример:

```
DUMP_8U( "%0x ", mtr,2,16,16,0)
DUMP_16U("%0x ", mtr,2,8,8,0)
DUMP_32U("%0x ", mtr,2,4,4,0)
DUMP_64U("%0lx ",mtr,2,2,2,0)
DUMP_32F("%.3f ",mtr,2,2,2,0)
```

Реализация самих функций печати находится в файле `mprintx.cpp` поэтому при работе с данными макросами требуется линковка этого файла.

Также вызов функций матричной печати доступен непосредственно из C/C++:

### Файл “main.cpp”

```
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <dumpx.h>

extern "C" {
    int clock_gettime(clockid_t, struct timespec *);
    void asm_print();
};

int main()
{
    printf("NMC Hello \n");

    unsigned long long mtr[] = {0x0706050403020100, 0x1716151413121110,
    0x2726252423222120, 0xFFFFFFFFFFFFFF };
    float      mtr32f[]={0.1,0.34,5,-2353.434,5,-54};
    double     mtr64f[]={0.1,0.34,5,-2353000666434,5,-54};

    printf ("===== c++ print =====\n");

    printf("---- hex      8-bit Matrix view ----:\n");
    dump_8u("%0xh ",mtr,2,16,16,0);
    printf("---- signed   8-bit Matrix view ----:\n");
    dump_8s("%0d ",mtr,2,16,16,1);
    printf("---- unsigned 8-bit Matrix view ----:\n");
    dump_8u("%0d ",mtr,2,16,16,2);

    printf("---- hex      16-bit Matrix view ----:\n");
    dump_16u("%0xh,",mtr,2,8,8,0);
    printf("---- signed   16-bit Matrix view ----:\n");
    dump_16s("%0d ",mtr,2,8,8,1);
    printf("---- unsigned 16-bit Matrix view ----:\n");
    dump_16u("%0d ",mtr,2,8,8,2);
```

```

printf("----- hex      32-bit Matrix view -----:\n");
dump_32u("%0xh ",mtr,2,4,4,0);
printf("----- signed   32-bit Matrix view -----:\n");
dump_32s("%0d ",mtr,2,4,4,1);
printf("----- unsigned 32-bit Matrix view -----:\n");
dump_32u("%0d ",mtr,2,4,4,2);

printf("----- hex      64-bit Matrix view-----:\n");
dump_64u("%0llxh ",mtr,2,2,2,0);
printf("----- signed   64-bit Matrix view-----:\n");
dump_64s("%0lld ",mtr,2,2,2,1);
printf("----- unsigned 64-bit Matrix view-----:\n");
dump_64u("%0lld ",mtr,2,2,2,2);

printf("----- float Matrix view-----:\n");
dump_32f("%.3f ",mtr32f,2,2,2,0);
printf("----- float Matrix exp view-----:\n");
dump_32f("%.3e ",mtr32f,2,2,2,1);
printf("----- 64-bit Matrix view-----:\n");
dump_64s("%0.3f ",mtr64f,2,2,2,0);
printf("----- 64-bit Matrix exp view-----:\n");
dump_64s("%0.3e ",mtr64f,2,2,2,1);

printf ("=====asm print =====\n");

asm_print();

return 777;
}

```

Выполнение данного кода приведет к следующему выводу

```

===== c++ print =====
----- hex      8-bit Matrix view -----
00h 01h 02h 03h 04h 05h 06h 07h 10h 11h 12h 13h 14h 15h 16h 17h
20h 21h 22h 23h 24h 25h 26h 27h ffh ffh ffh ffh ffh ffh ffh ffh
----- signed   8-bit Matrix view -----
0x38022: 0000 0001 0002 0003 0004 0005 0006 0007 0016 0017 0018 0019 0020 0021 0022
0023
0x38032: 0000 0001 0002 0003 0004 0005 0006 0007 0016 0017 0018 0019 0020 0021 0022
0023
----- unsigned 8-bit Matrix view -----
0: 000 001 002 003 004 005 006 007 016 017 018 019 020 021 022 023
1: 032 033 034 035 036 037 038 039 255 255 255 255 255 255 255 255
----- hex      16-bit Matrix view -----
0100h,0302h,0504h,0706h,1110h,1312h,1514h,1716h,
1110h,1312h,1514h,1716h,2120h,2322h,2524h,2726h,

```

```

----- signed 16-bit Matrix view -----:
0x38022: 00256 00770 01284 01798 04368 04882 05396 05910
0x38026: 00256 00770 01284 01798 04368 04882 05396 05910
----- unsigned 16-bit Matrix view -----:
 0: 00256 00770 01284 01798 04368 04882 05396 05910
 1: 04368 04882 05396 05910 08480 08994 09508 10022
----- hex      32-bit Matrix view -----:
03020100h 07060504h 13121110h 17161514h
03020100h 07060504h 13121110h 17161514h
----- signed 32-bit Matrix view -----:
0x38022: 050462976 117835012 319951120 387323156
0x38026: 589439264 656811300 -00000001 -00000001
----- unsigned 32-bit Matrix view -----:
 0: 050462976 117835012 319951120 387323156
 1: 050462976 117835012 319951120 387323156
----- hex      64-bit Matrix view-----:
0706050403020100h 1716151413121110h
2726252423222120h ffffffffffffffh
----- signed 64-bit Matrix view-----:
0x38022: 0506097522914230528 1663540288323457296
0x38026: 2820983053732684064 -000000000000000000000001
----- unsigned 64-bit Matrix view-----:
 0: 0506097522914230528 1663540288323457296
 1: 2820983053732684064 -000000000000000000000001
----- float Matrix view-----:
 0.100      0.340
 5.000 -2353.434
----- float Matrix exp view-----:
0x3802a: 1.000e-01 3.400e-01
0x3802c: 5.000e+00 -2.353e+03
----- 64-bit Matrix view-----:
00000000000000.100 00000000000000.340
00000000000005.000 -23530000666434.000
----- 64-bit Matrix exp view-----:
0x38012: 01.000e-01 03.400e-01
0x38016: 05.000e+00 -2.353e+13

```

## 1.3. Простейшая Программа на Языке Ассемблера

Исходный текст примера, используемого в данном уроке, содержится в файле **step1.S** в каталоге: **\steps-risc\step01**.

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

**Файл “step1.S”**

```
.global __main      // объявление глобальной метки

.text               // 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
```

### 1.3.1. 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 `.text`. Данная запись является сокращением полного объявления секции `.section .text`.



Рекомендуется имя секции кода начинать с префикса `.text`, например: `.section .text.MyCodeSection`. В случае если имя секции имеет префикс `.text` дизассемблер (программа `nm -objdump`), 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 объявлением секции кода `.text` 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.

## Запуск Программы на Симуляторе

Полученный файл `test.abs` может быть выполнен на симуляторе (программа `nmc-qemu`). Эта программа представляет собой программный эмулятор на уровне инструкций. Она выполняет программу, и в строке `Simulation error! Return value 3` отображает значение, возвращаемое пользовательской программой. Такая строка появляется всегда, когда функция `main()` возвращает ненулевое значение. Если же функция возвращает нуль, то результатом будет строка `Successful end of simulation!`

```
c:\Module\MC12101\examples\first-steps\steps-risc\step01\qemu>make run  
nmc-qemu test.abs  
  
Simulation error! Return value 3  
make: *** [run] Error 1
```

## Отладка Программы на GDB-Отладчике

При отладке программы через GDB из командной строки значение регистров можно вывести с помощью команды `p` или `print`. Например

```
(gdb) b main  
Breakpoint 1 at 0x25e: file ../../step1.S, line 5.  
(gdb) c  
Continuing.  
  
Breakpoint 1, main () at ../../step1.S:5  
5          gr0 = 1;  
(gdb) n  
6          gr1 = 2;  
(gdb) n  
7          gr7 = gr0 + gr1;  
(gdb) n  
8          return;  
(gdb) p $gr7  
$1 = 3  
(gdb) c  
Continuing.
```

Вывести список всех доступных регистров можно командой `info registers`

В среде VS CODE вывести значение регистра можно в окне `Registers`, либо `Watch` также задав имя регистра после `$`.

## 1.4. Доступ к Памяти

Исходный текст примера, используемого в данном уроке, содержится в файле `step2.S` в каталоге: `\steps-risc\step02`.

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

#### Файл “step2.S”

```
.global __main           // объявление глобальной метки.

.section .data.MyData   // секция инициализированных данных.
A: .long 1
B: .long 2

.section .bss.MyData1  // секция неинициализированных данных.
C: .space 2<<2

.section .text.AAA      // 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 gr1 загрузили 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;
```

#### 1.4.1. Комментарии к Примеру

В примере описаны секции инициализированных и неинициализированных данных. В секциях инициализированных данных содержатся объявления и инициализация переменных, используемых программой.

```
.section .data.MyData   // секция инициализированных данных.
A: .long 1
B: .long 2
```

Объявление переменной имеет вид **имя\_переменной : тип**, например **A: .long**. Начальное значение или список начальных значений следует за объявлением типа переменной. Тип **.long** соответствует 32-битному целому числу. Тип **.quad** соответствует 64-битному целому числу.

В секциях неинициализированных данных содержатся только объявления переменных,

используемых программой, без их инициализации. Пример:

```
.section .bss.MyData1 // секция неинициализированных данных.  
C: .space 2<<2
```

Означает что по адресу **C** зарезервировано область памяти размером 8 байт. Секция кода демонстрирует команды чтения из памяти:

- Получение адреса переменной. Для этого достаточно просто использовать ее имя: **аг0 = A;** (в адресный регистр **аг0** загружаем адрес переменной **A**)
- Косвенное чтение из памяти в регистр общего назначения **гр0. гр0 = [аг0];**
- Получение значения переменной. Для этого необходимо ее имя заключить в квадратные скобки. **гр1 =[B]; // прямое чтение из памяти в регистр общего назначения** (в регистр общего назначения **гр1** загрузили значение переменной **B**).

Команда **[аг0++]** = **гр2;** выполняет косвенную запись в память из регистра общего назначения с пост-инкрементацией адреса. Это означает, что по адресу, указанному в **аг0** записывается значение, содержащееся в **гр2**, после чего **аг0** увеличивается на 1. Команда **гр7 = [--аг0];** выполняет косвенное чтение из памяти в регистр общего назначения с предекрементацией 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 **гр7** попадает значение **C[1].**

## 1.5. Организация Циклов

Исходный текст примера, используемого в данном уроке, содержится в файле **step3.S** в каталоге: **\steps-risc\step03.**

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

**Файл “step3.S”**

```

.global __main      // объявление глобальной метки.

.section .bss.MyData1 // секция неинициализированных данных.
C:
.space 16<<2      // объявили массив из 16 32-разрядных слов

.section .text.AAA   // начало секции кода.
__main:
    аг0 = C;          // в аг0 загрузили адрес массива —.
    гр0 = 0;           // в гр0 загрузили значение 0.
    гр1 = 16;          // в гр1 загрузили значение 16, равное количеству итераций
в цикле.
Loop:
    [аг0++] = гр0;    // в память по адресу аг0 записываем содержимое гр0, а
                      // затем увеличиваем адрес на 1 (пост-инкрементация).
    гр0++;            // увеличили значение гр0 на 1
    гр1--;            // уменьшили значение гр1 на 1, таким образом установили
                      // флаг в регистре pswr для дальнейшей проверки
    if > goto Loop;   // если условие выполнено, осуществляется переход на
метку Loop.

    return;

```

### 1.5.1. 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 команды 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 нет специального регистра - счётчика циклов). Установка флагов происходит только при выполнении арифметико-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.

## 1.6. Оптимизация Выполнения Цикла

Исходный текст примера, используемого в данном уроке, содержится в файле **step3a.S** в каталоге: **\steps-risc\step03a**.

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

### Файл “step3a.S”

```

.global __main          // объявление глобальной метки.

.section .bss.MyData1 // секция неинициализированных данных.

C: .space 16<<2 // объявление массива из 16 32-разрядных слов
                  // объявление массива из 16 32-разрядных слов

.section .text.AAA    // начало секции кода.

__main:
    аг0 = C;           // в аг0 загружается адрес массива С.
    гр0 = 0;            // в гр0 загружается значение 0.
    гр1 = 16;           // в гр1 загружается значение 16,
                        // равное количеству итераций в цикле.
    гр1--;              // 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 гр1--;
    // две следующих инструкции выполняются до того, как произойдёт переход
    [аг0++] = гр0 with гр0++ noflags;
    нул;
    // ----- здесь произойдёт переход на метку Loop -----

    return;             // сюда перейдёт программа, когда условие не выполнится

```

### 1.6.1. 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 А.1.Типы Команд Перехода данного документа. Упрощённая схема выполнения перехода подразумевает, что компилятор сам рассчитывает количество отложенных инструкций и заполняет их пустыми командами (**нул**). Если программист для выполнения перехода использует инструкцию без ключевого слова **delayed**, например: **if > goto Loop;** то две инструкции **нул** 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 гр1--;** будут выполнены две отложенные инструкции:

```
[аг0++] = гр0 with гр0++ noflags;
нул;
```

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.

## 1.7. Копирование Массива Данных на Скалярном Процессоре

Исходный текст примера, используемого в данном уроке, содержится в файле `step4.S` в каталоге: `\steps-risc\step04`. Пример демонстрирует два способа копирования массива 64-разрядных слов на скалярном процессоре. Первый способ – простое копирование, второй – копирование при помощи регистровых пар.

Файл “`step4.S`”

```

.global __main          // объявление глобальной метки.

.section .data.MyData // секция инициализированных данных
    // массив А из 16 64-разрядных слов заполняется начальными значениями
    .global A
    A: .quad 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB, 0xC,
       0xD, 0xE, 0xF

.section .bss.MyData1 // секция неинициализированных данных.
    .global B
    B: .space 16<<3   // объявляется массив В из 16 64-разрядных слов
    .global C
    C: .space 16<<3   // объявляется массив С из 16 64-разрядных слов

.section .text.AAA    // 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--;
        gr2 = [ar0++]; // чтение из памяти 32-разрядного слова
        [ar1++] = gr2; // запись в память 32-разрядного слова

    // копирование массива данных при помощи регистровых пар
    ar0 = A;
    ar1 = C;
    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--;
        ar2,gr2 = [ar0++]; // чтение из памяти 64-разрядного слова
        [ar1++] = ar2,gr2; // запись в память 64-разрядного слова

return;

```

### 1.7.1. 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 `аг2, 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 `аг2, gr2 = [аг0++]`;

ВСЕГДА младшая часть 64-разрядного слова попадает в `агX`, 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 `агX`, по старшему - `grX`. Таким образом, команда `[аг1++] = gr2, ag2;`

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 `аг0` и `аг1`. 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 = [аг0++]; // аг0 увеличивается на 1  
аг2, gr2 = [аг0++]; // аг0 увеличивается на 2
```

## 2. Работа с целочисленным векторным сопроцессором

### 2.1. Копирование Массива Данных на Векторном Процессоре

Исходный текст примера, используемого в данном уроке, содержится в файле `step4a.S` в каталоге: `steps-fixed/stepi04a`. Пример демонстрирует копирование массива 32-разрядных слов с помощью скалярного процессора и векторного процессора.

Файл “`step4a.S`”

```

#include "printx.hs"
.global __main      // объявление глобальной метки.

.p2align 3
.section .data.MyData // секция инициализированных данных
// массив А из 16 32-разрядных слов заполняется начальными значениями

.global A
A: .long    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

.p2align 3
.section .bss.MyData1 // секция неинициализированных данных.
.global B
B: .space 16<<2      // объявление массива из 16 32-разрядных слов
// объявляется массив В из 16 32-разрядных слов
.global C
C: .space 16<<2      // объявление массива из 16 32-разрядных слов
// объявляется массив С из 16 32-разрядных слов

.section .text.AAA      // начало секции кода.
__main:
                // копирование массива данных
                // с помощью скалярного процессора
ар0 = A;
ар1 = 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--;
gr2 = [ар0++]; // чтение из памяти 32-разрядного слова
[ар1++] = gr2; // запись в память 32-разрядного слова

// копирование массивов данных с помощью векторного процессора
ар0 = A;
ар1 = C;
// массив А подаётся на векторное АЛУ и попадает в afifo без изменений
геп 8 data = [ар0++] with data;
// сохранение во внешней памяти содержимого afifo, заполненного
предыдущей
// векторной инструкцией.

ар5 = ар1; // сохраняем адрес массива С

```

```

    ger 8 [ар1++] = afifo;

    PRINTF("Output\:\n")
    PRINTF1("C addr =%X\n", аг5)
    PRINTF2("[C]={%d,%d}\n", [аг5++], [аг5++])

    return;

```

### 2.1.1. 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 **гер 8 data = [аг0++]** 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

```

SET 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 **гер 8 [аг1++] = afifo;**

осуществляет выгрузку данных из afifo в память с пост-инкрементацией адресного регистра. (**гер** 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 процессоречитываются/записываются 32-разрядные слова, тогда как на векторном процессоре осуществляется чтение/запись 64-разрядных слов. Таким образом, при инкрементации [ар1++]<sup>1</sup> адресный процессор каждый раз увеличивается на два.

## 2.2. Арифметические Операции на Векторном АЛУ

Исходный текст примера, используемого в данном уроке, содержится в файле **step5.S** в каталоге: **steps-fixed/step05**. 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.

### Файл “step5.S”

```
.global __main          // объявление глобальной метки.

.p2align 3              // директива для выравнивания секции по чётному
адресу.

.section .data.MyData  // секция инициализированных данных

// начальное значение для заполнения массива
AA: .quad 0x10000000
// инкремент для первого цикла
BB: .quad 0x20000002
// инкремент для второго цикла
CC: .quad 0x400000040


.section .bss.MyData1 // секция неинициализированных данных.
.p2align 3              // выравнивание начала массива по чётному адресу
// массив из 256-ти 32-х разрядных элементов, который будет заполнен
// возрастающими значениями от 0 до 255
.global A
A: .space 256<<2


.section .text.AAA      // начало секции кода.
__main:
    ar0    = AA;        // в ar0 загружается адрес AA (AA = 10000000h)
    ar4    = BB;        // в ar4 загружается адрес BB (BB = 20000002h)
    ar1,gr1 = A;        // в ar1 и в gr1 загружается адрес буфера A
    gr2    = 31;         // счетчик цикла

#if __NM4__==0
    nb1 = 8000000h;    // разбиение матрицы на два столбца по 32 бита
#else
    sir = 8000000h;    // разбиение матрицы на два столбца по 32 бита
    nb1 = sir;
```

```

#endif

wtw;           // копирование содержимого теневого регистра nb1 в
               // рабочий nb2

// в гам записывается инкремент, который будет добавляться в цикле
// к текущему значению afifo для получения новых значений заполнителя.
rep 1 gam = [arg4];
// в векторный процессор заносится первое значение заполнителя.
rep 1 data = [arg0] 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 [arg1++] = afifo with afifo + gam;
nul;

rep 1 [arg1++] = afifo; // выгрузка в память последнего значения из afifo
arg1 = gr1;           // возвращение в начало массива

arg0 = CC;            // в arg0 загружается адрес CC (CC = 4000000040h1)

// в гам записывается инкремент, который будет добавляться в цикле
// к текущему значению afifo для получения новых значений заполнителя.
rep 32 gam = [arg0];
// в векторный процессор заносятся первые значения заполнителя из А.
gr2 = 3;             // счётчик для второго цикла
arg2 = gr1 with gr2--; // arg2 устанавливается в начало массива
                      // и установка флагов для контроля обнуления счетчика
rep 32 data = [arg2++] with data;

Loop1:
if > delayed goto Loop1 with gr2--;
// массив А заполняется возрастающими значениями
// за один цикл обрабатываются 64 32-разрядных элемента.
rep 32 [arg1++] = afifo with afifo + gam;
nul;

// последние 64 элемента массива сохраняются в памяти
rep 32 [arg1++] = afifo;

return;

```

## 2.2.1. Комментарии к Примеру

Программа состоит из двух частей, выполняющих аналогичные действия по заполнению массива А 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 `гер 1 гам = [ар4];` помещает в `гам` 64-х разрядный вектор `0x0000000200000002`. Этот вектор будет использоваться для инкрементации значений заполнения массива. В результате выполнения инструкции `гер 1 data = [ар0] with data;` в `afifo` попадёт число `0x0000000100000000`, с которого начинается заполнение массива. Далее в цикле происходит заполнение массива. После команды перехода выполняются две отложенные инструкции

```
гер 1 [ар1++] = afifo with afifo + гам;
нул;
```

Старое значение `afifo` сохраняется во внешней памяти, и одновременно с этим к нему прибавляется инкремент - вектор, расположенный в `гам`. Результат операции снова помещается в `afifo`. Таким образом, в массиве заполняются 64 значения (по два в каждом цикле). Команда `гер 1 [ар1++] = 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 `гам` заносится дублированное 32 раза число `0x0000004000000040`. 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.

## 2.3. Операция Взвешенного Суммирования

Исходный текст примера, используемого в данном уроке, содержится в файле `step6.S` в каталоге: `steps-fixed/stepi06`. 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.

### Файл “`step6.S`”

```
.global __main          // объявление глобальной метки.

.p2align 3              // директива для выравнивания секции по чётному адресу.
.section .data.MyData // секция инициализированных данных
    // исходный вектор
    A: .quad 0x8877665544332211
    // место для хранения результата вычислений
    B: .quad 0x0
    // массив Matr содержит значения
    // для заполнения матрицы весовых коэффициентов
    Matr: .quad    0x0100000000000000, \
                  0x0001000000000000, \
                  0x0000010000000000, \
                  0x0000000100000000,
```

```

0x0000000001000000, \
0x0000000000010000, \
0x0000000000000100, \
0x0000000000000001

.macro SET reg,val
#if __NM4__== 0
    \reg = \val;
#else
    sir = \val;
    \reg = sir;
#endif
.endm

.section .text.AAA // начало секции кода.
__main:
    ar1 = Matr;

    SET nb1, 80808080h // разбиение матрицы на 8 столбцов по 8 бит
    SET 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;

```

### 2.3.1. Комментарии к Примеру

Задачей данного примера является перестановка порядка элементов в 64-разрядном векторе из состояния  $A = 0x8877665544332211$  в состояние  $B = 0x1122334455667788$ . Эта перестановка выполняется на устройстве умножения векторного процессора при помощи операции взвешенного суммирования. Основная идея этого преобразования поясняется на рисунке [Рисунок 1](#):

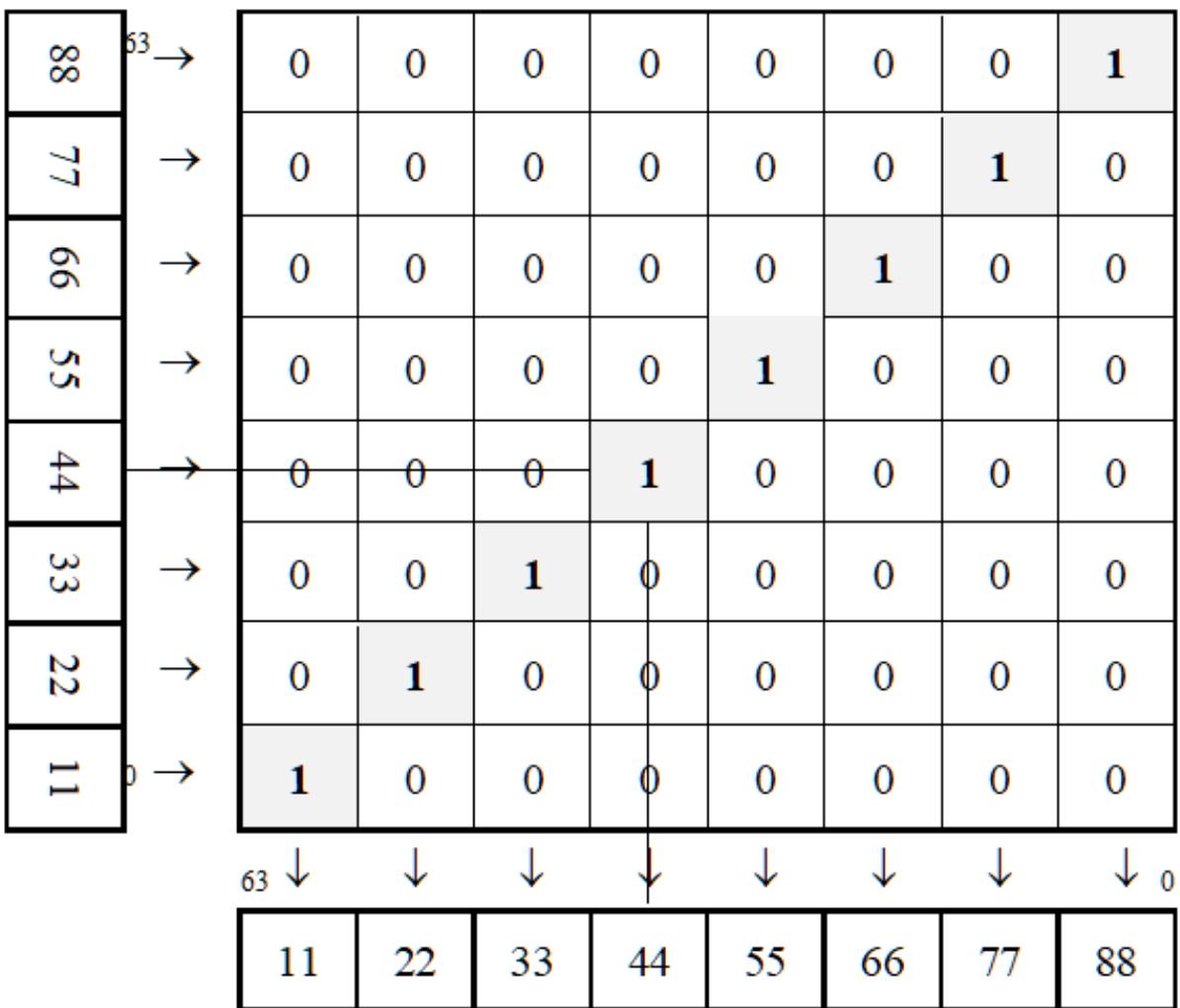


Рисунок 1. Перестановка Элементов Вектора на Матричном Умножителе

Для выполнения операции взвешенного суммирования необходимо заполнить матрицу весовых коэффициентов значениями. Но прежде требуется разбить матрицу на строки и столбцы. Макрос `SET nb1 , 0x80808080`; разбивает матрицу на 8 столбцов. При этом в обе части регистра попадают одинаковые константы. Таким образом, в `nb1` содержится константа `0x80808080808081`. Регистр `nb1` является 64-разрядным регистром. Он отвечает за разбиение теневой матрицы на столбцы. Команда `SET sb , 0x03030303`; разбивает матрицу на 8 строк. При этом в обе части регистра попадают одинаковые константы. Таким образом, в `sb` (64 разряда) содержится константа `0x0303030303030303`. Команда `гер 8 wfifo = [аг1++]`; 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 `гер 1 data = [аг2] with vsum , data, 0;` выполняет взвешенное суммирование с коэффициентами, которые прежде были загружены в рабочую матрицу. Вычисление производится по схеме приведённой на рисунке Рисунок 1. Результат операции попадает в регистр-контейнер `afifo`. Инструкция `гер 1 [аг4] = afifo;` выгружает результат из `afifo` во внешнюю память.

## 2.4. 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.S` в каталоге `steps-fixed/stepi07`. 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, выполняющей негативацию входного
                    // параметра.
}
```

Файл “`step7.S`”

```
.global _Neg          // объявление метки с именем подпрограммы.

.text
_Neg:
    arg5 = arg7 - 2; // адреса в стеке для доступа к входным параметрам.
    push arg0, gr0; // сохранение используемых в подпрограмме регистров.

    gr0 = [--arg5]; // получение значения входного параметра.
    gr7 = - gr0;

    pop arg0, gr0; // восстановление значений регистров при выходе
    return;
```

### 2.4.1. 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.S` содержит реализацию функции `Neg`. Функция возвращает значение передаваемого ей параметра с обратным знаком. `.global _Neg` - 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

```
.text  
_Neg:  
...  
return;
```

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 `аг5` и `гр7`. В отдельных случаях, которые будут описаны ниже, регистр `гр6` также может быть изменён. Первой инструкцией любой ассемблерной функции, в которую передаются входные параметры, должна быть инструкция `аг5 = аг7 - 2;` В регистр `аг5` помещается указатель на место в стеке, ниже которого находятся параметры вызова функции. Адресный регистр `аг7` используется процессором в качестве указателя стека. Это означает, что `аг7` 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 `аг7` 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 аг0,гр0;` 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. `гр0 = [-аг5];` - в регистр `гр0` помещается значение параметра функции. `гр7 = - гр0;` - значение, возвращаемое функцией, должно быть записано в регистр `гр7`. `pop аг0, гр0;` - команда чтения регистровой пары из вершины стека. `return;` - команда возврата из функции.

## 2.5. Передаваемые и Возвращаемые Значения Типа LONG

Исходный текст примера, используемого в данном уроке, содержится в файле `step8.S` в каталоге: `steps-fixed/stepi08`. 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;
    // вызов функции Neg_Scal(a) с параметром а и запись значения,
    // возвращаемого функцией, в переменную б.
    long b = Neg_Scal(a);
    // вызов функции Neg_Vect(a) с параметром а и запись значения,
    // возвращаемого функцией, в переменную с.
    long c = Neg_Vect(a);
    return int(b-c);
}

```

Файл “step8.S”

```

.global _Neg_Scal
.global _Neg_Vect

.p2align 3                // директива для выравнивания секции по чётному
адресу.
.section .bss.my_data
A: .space 1<<3          // выделяется место для 64-разрядной переменной j

.macro SET reg,val
#if __NM4__== 0
    \reg = \val;
#else
    sir = \val;
    \reg = sir;
#endif
.endm

.text
// функция Neg_Scal на скалярном процессоре выполняет обработку
// 64-разрядного числа заменяя его знак на противоположный
_Neg_Scal:
    arg5 = arg7 - 2;      // сохраняется указатель стека
    push arg0, gr0;       // сохраняются регистровые пары в стеке
    push arg1, gr1;
    arg0,gr0 = [--arg5];// из стека считывается входной параметр функции
    gr1 = arg0;           // в gr1 помещается младшее слово параметра

    // в gr1 записывается 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 arg1, gr1;      // восстановление регистровых пар из стека
pop arg0, gr0;
return;             // возвращаемое значение передается в регистрах:
                   // gr6 - старшая часть, gr7 - младшая часть

// функция Neg_Vect на векторном процессоре выполняет обработку
// 64-разрядного числа заменяя его знак на противоположный
_Neg_Vect:
arg5 = arg7 - 2;    // сохраняется указатель стека
push arg0, gr0;     // сохраняются регистровые пары в стеке
push arg1, gr1;
arg1 = A;           // в arg1 загружается адрес j

// nb1 определяет разбиение на элементы 64-разрядного вектора,
// участвующего в арифметических операциях на векторном процессоре
// (nb1 = 0 - разбиения нет).
SET nb1 , 0

wtw;                // копирование содержимого теневого регистра nb1 в
                     // рабочий nb2
// изменение знака 64-разрядного числа
ger 1 data = [--arg5] with 0-data;
// результат помещается в память по адресу, хранящемуся в регистре arg1
ger 1 [arg1] = afifo;
gr7 = [arg1++];     // младшая часть результата копируется ил памяти в
                     // регистр gr7
gr6 = [arg1++];     // старшая часть результата копируется ил памяти в
                     // регистр gr6
pop arg1, gr1;      // восстановление регистровых пар из стека
pop arg0, gr0;
return;

```

## 2.5.1. Комментарии к Примеру

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

### Замена Знака Числа на Скалярном Процессоре

Функция начинается с сохранения в `arg5` 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 `arg0`, старшее – в `gr0`:

`arg0,gr0 = [--arg5];` Поскольку адресный регистр не может использоваться в арифметических

операциях, его значение необходимо скопировать в регистр общего назначения: `gr1 = аг0`; 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:



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 + саггу`; 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`.

### Замена Знака Числа на Векторном Процессоре

Инструкции `SET 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 = [--аг5] with 0-data;` - векторная инструкция, осуществляющая арифметическую операцию на векторном АЛУ (разность operandов X и Y, где на operand X подан нулевой вектор). Операция изменяет знак входного параметра функции. `rep 1 [ар1] = аfifo;` - результат из `afifo` помещается в переменную A, адрес которой находится в `аг1`. Далее старшая и младшая части A копируются в регистры `gr6` и `gr7` соответственно.

## 2.6. Операции Логической Активации и Маскирования на ВП

Исходный текст примера, используемого в данном уроке, содержится в файле `step9.S` в каталоге: `steps-fixed/stepi09`. Пример демонстрирует выполнение операции логической активации и операции маскирования на векторном АЛУ.

### Файл “main.cpp”

```

// функция Mask объявлена как внешняя с —и-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 на чальными значениями
        A[i] = 0x0102030405060708*i;
    }
    Mask(A, 0x44); // вызов функции Mask, первый параметр - адрес массива
                    // значений, второй - маска.
    return 0;
}

```

### Файл “step9.S”

```

.global _Mask           // объявляется глобальная метка _Mask

.p2align 3              // директива для выравнивания секции по чётному
адресу.
.section .data.my_data
Temp: .quad 0x0

.macro SET reg,val
#if __NM4__== 0
\reg = \val;
#else
sir = \val;
\reg = sir;
#endif
.endm

.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: 444444444444444h,
// при этом в регистр ar1 заносится адрес этой переменной.
[ar1 = Temp] = ar1, gr1;

```

SET nb1, 80808080h

```

wtw;
// регистр управления функцией активации, задаёт обработку вектора,
// подаваемого на operand X векторного процессора.
SET 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 or 0;
// выполнение операции маскирования, маска хранится в afifo, в
// operand X попадают данные с шины данных, в operand Y данные из ram
rep 32 data = [ar1] with mask afifo, data, ram;
// результат операции сохраняется во внешней памяти.
rep 32 [ar2++] = afifo;

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

```

## 2.6.1. 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 **0x1122334455667788**, то после применения порога **0x44** к каждому его элементу, получится **0x1122334444444444**. На вход функции подаётся байтовый порог. Функция **Mask** содержит код, позволяющий по этим данным сформировать полноценный 64 разрядный вектор порога.

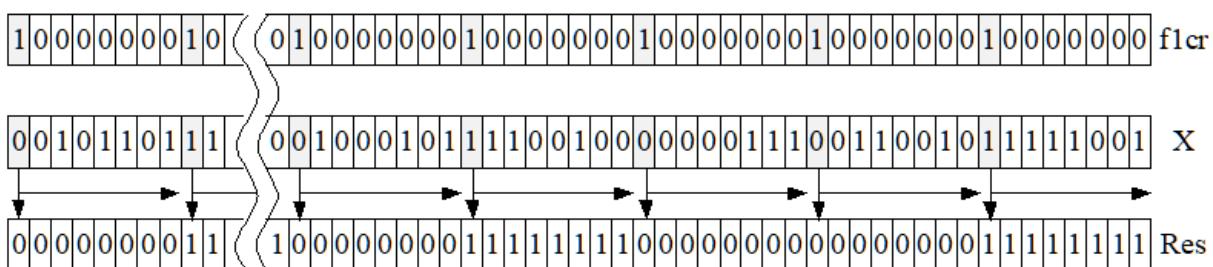
```

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 `arg1`. 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. `SET f1cr , 0x80808080`; 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 = [arg0++]`; - вектора входных данных загружаются в регистр-контейнер `ram`. `rep 32 data = [arg1] 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. [Рисунок 2](#)).



*Рисунок 2. Распространение Знаковых Битов при Создании Маски*

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 нулюм. 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 NeuroMatrix 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. **Рисунок 3**:

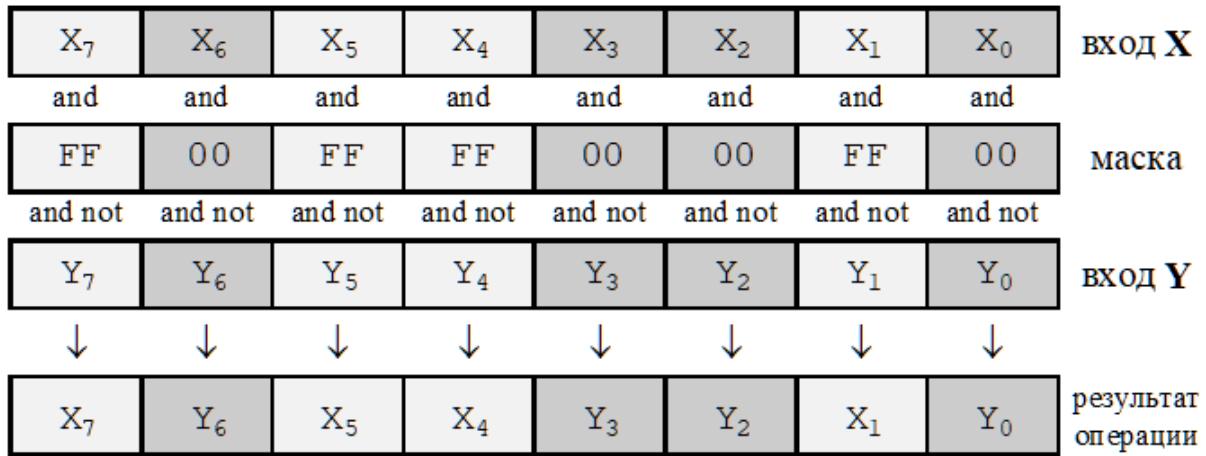


Рисунок 3. Распространение Знаковых Битов при Создании Маски

Операция логического маскирования выполняется при вызове следующей инструкции: `гер 32 data = [ар1] with mask afifo, data, гам;` В данном случае вектора из `afifo` используются в качестве маски, данные, считываемые из внешней памяти – операнд `X`, данные из `гам` – операнд `Y`. Над всеми наборами векторов, принимающих участие в вычислениях, выполняется преобразование:  $(X \text{ and MASK}) \text{ or } (Y \text{ and not MASK})$  Как видно из Рис. Рисунок 3, 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 тех позициях, на которых в `afifo` стоят 1, будут записаны биты из `Temp` (регистр `ар1`), в тех позициях, на которых стоят 0, будут записаны биты из массива данных (`гам`). `гер 32 [ар2++]` = `afifo`; - результат сохраняется во внешней памяти.

## 2.7. Операция Арифметической Активации

Исходный текст примера, используемого в данном уроке, содержится в файле `step10.S` в каталоге: `steps-fixed/stedi010`. 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 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 трех параметров
    AddSaturate( SRC1, SRC2, DST );
    return 0;
}

```

## Файл “step10.S”

```

.global _AddSaturate

.p2align 3      // директива для выравнивания секции по чётному адресу.
.data
Masks:
    .quad 0x0000000000000001      // матрица для первого прохода
    .quad 0x0000000000010000
    .quad 0x0000001000000000
    .quad 0x0001000000000000

    .repnt 4
    .quad 0x0000000000000000
    .endr

    .quad 0x0000000000000001      // матрица для второго прохода
    .quad 0x0000000000001000
    .quad 0x0000000000010000
    .quad 0x0000000001000000

                                // матрица для третьего
    .repnt 4
    .quad 0x0000000000000000
    .endr

    .quad 0x0000000000000001      // прохода
    .quad 0x0000000000010000
    .quad 0x0000000100000000
    .quad 0x0001000000000000

```

```

.quad
.quad 0x0000000100000000      // матрица для четвертого
.quad 0x0000100000000000      // прохода
.quad 0x0001000000000000
.quad 0x0100000000000000

.macro SET reg,val
#if __NM4__== 0
\reg = \val;
#else
sir = \val;
\reg = sir;
#endif
.endm

.text
_AddSaturate:
    ar5 = ar7 - 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

    SET f1cr, 0FF80FF80h // конфигурация арифметической функции активации

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

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

    // поскольку рабочая матрица уже загружена, можно приступить
    // к загрузке теневой матрицы новой порцией весовых коэффициентов
    // и определить новую конфигурацию.

    SET nb1, 80808080h
    SET sb , 00030003h

```

```

// вычисления на рабочей матрице выполняются параллельно с загрузкой
// теневой, следующие две инструкции выполняют преобразование
// разрядностей и поэлементное сложение входных векторов.
гер 32 data = [аг0++], ftw with vsum , data, 0;
гер 32 data = [аг1++] with vsum , data, afifo;

wtw;           // копирование теневой матрицы в рабочую

SET nb1, 80008000h
SET sb, 03030303h
// выполнение арифметической активации
// с последующим преобразованием разрядности
гер 32 ftw with vsum , activate afifo, 0;

// возвращение к началу исходных массивов
// для обработки вторых половин векторов
аг0 = гр0;
аг1 = гр1;
// сохранение результатов первого шага преобразования в гам
гер 32 [аг4],гам = afifo;
wtw;

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

гер 32 data = [аг0++], ftw with vsum , data, 0;
гер 32 data = [аг1++] with vsum , data, afifo;

wtw;
// инструкция активируют данные, преобразует размерность и складывает с
// результатом первого прохода.
гер 32 with vsum , activate afifo, гам;

// результат вычислений сохраняется в памяти.
гер 32 [аг4++] = afifo;

pop аг6, гр6; // восстановление регистровых пар из стека
pop аг4, гр4;
pop аг1, гр1;
pop аг0, гр0;
return;

```

## 2.7.1. 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. [Рисунок 4](#)).

8	0	0	0	0
7	0	0	0	0
6	0	0	0	0
5	0	0	0	0
4	1	0	0	0
3	0	1	0	0
2	0	0	1	0
1	0	0	0	1

A)  4 3 2 1

4	0	0	0	0	1	0	0	0
3	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	1

B)  0 0 0 0 4 3 2 1

8	1	0	0	0
7	0	1	0	0
6	0	0	1	0
5	0	0	0	1
4	0	0	0	0
3	0	0	0	0
2	0	0	0	0
1	0	0	0	0

B)  8 7 6 5

8	1	0	0	0	0	0	0	0
7	0	1	0	0	0	0	0	0
6	0	0	1	0	0	0	0	0
5	0	0	0	1	0	0	0	0

Г)  8 7 6 5 0 0 0 0

+ 0 0 0 0 4 3 2 1

= 8 7 6 5 4 3 2 1

Рисунок 4. Матричные Операции при Поэлементном Сложении Векторов

Функция выполняет следующую последовательность действий:

- Загружает веса в матрицу для выполнения преобразования разрядности, как изображено на Рис. [Рисунок 4А](#);
- 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 граничные значения диапазона;
- Выполняет обратное преобразование из 16-ти в 8 бит, как изображено на Рис. [Рисунок 4Б](#);
- Сохраняет результат первого прохода во внутреннем буфере **гам**;
- Выполняет аналогичные преобразования над старшими половинами входных векторов, как изображено на Рис. [Рисунок 4В](#) 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: **гер 24 wfifo = [агб++], 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 **гер 32 data = [аг0++], ftw with vsum , data, 0;** совмещает в себе выполнение операции взвешенного суммирования с загрузкой новой порции весовых коэффициентов (**ftw**) в теневую матрицу. Инструкция **гер 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 гам + data**, или **0 – activate afifo**. Операция взвешенного суммирования также является арифметической, поэтому в паре с ней выполняется именно функция насыщения. Рис. [Рисунок 5](#) приводит пример того, как регистр **f1cr** (**f2cr**) управляет арифметической функцией активации:

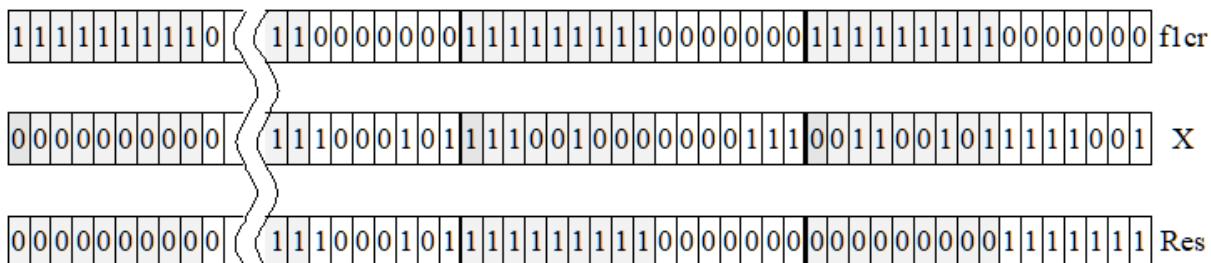


Рисунок 5. Управление Арифметической Функцией Активации

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. [Рисунок 5](#) помечены серым). В случае если значение хотя бы одного знакового бита отличается от значения самого старшего бита (на рисунке затемнёй 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).

## 2.8. Использование Циклического Сдвигателя на ВП

Исходный текст примера, используемого в данном уроке, содержится в файле **step11.S** в

каталоге: `steps-fixed/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-ий и 63-ий биты, 1-ый и 62-ой, и т.д.

### Файл “main.cpp”

```
// функция ReverseBits объявлена внешней с —и-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);    // С 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.S”

```
.global _ReverseBits
.p2align 3    // директива для выравнивания секции по чётному адресу
.data         // весовые коэффициенты для перестановки битов
Weights: .quad
            1<<63, 1<<61, 1<<59, 1<<57, \
            1<<55, 1<<53, 1<<51, 1<<49, \
            1<<47, 1<<45, 1<<43, 1<<41, \
            1<<39, 1<<37, 1<<35, 1<<33, \
            1<<31, 1<<29, 1<<27, 1<<25, \
            1<<23, 1<<21, 1<<19, 1<<17, \
            1<<15, 1<<13, 1<<11, 1<< 9, \
            1<< 7, 1<< 5, 1<< 3, 1<< 1, \
            1<<62, 1<<60, 1<<58, 1<<56, \
            1<<54, 1<<52, 1<<50, 1<<48, \
            1<<46, 1<<44, 1<<42, 1<<40, \
            1<<38, 1<<36, 1<<34, 1<<32, \
            1<<30, 1<<28, 1<<26, 1<<24, \
            1<<22, 1<<20, 1<<18, 1<<16, \
            1<<14, 1<<12, 1<<10, 1<< 8, \
            1<< 6, 1<< 4, 1<< 2, 1<< 0

.macro SET reg,val
#endif __NM4__ == 0
\reg = \val;
```

```

#else
    sir = \val;
    \reg = sir;
#endif
.endm

.text
.ReverseBits:
    ar5 = ar7 - 2;
    push ar0, gr0;

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

    //nb1 = 0xFFFFFFFFh; // 64 столбца
    //sb  = 0xFFFFFFFFh; // 32 строки
    SET nb1, 0xFFFFFFFFh
    SET sb , 0xFFFFFFFFh

    // загрузка первого набора весовых коэффициентов в рабочую матрицу
   гер 32 wfifo = [ар0++],ftw, wtw;
    // загрузка второго набора весовых коэффициентов в теневую матрицу
    гер 32 wfifo = [ар0++],ftw;

    // слово входных данных напрямую из стека загружается в gam
    // и одновременно подаётся на вход X матричного устройства умножения
    гер 1 gam = [--ар5] with vsum , data, 0;
    wtw;
    // после обновления коэффициентов тоже слово входных данных подаётся
    // из gam на вход X, а по пути циклически сдвигается на 1 бит вправо
    гер 1 with vsum , shift gam, afifo;
    гер 1 [ар5] = afifo;

    gr7 = [ар5++];      //младшее слово вектора результата - в gr7
    gr6 = [ар5++];      //старшее слово вектора результата - в gr6

    pop ar0, gr0;
    return;

```

## 2.8.1. 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 `гер 1 gam = [--ар5] with vsum , data, 0;` обращается непосредственно в стек, считывает входной вектор, загружает его в `gam`, а пока он проходит по шине данных дублирует его и отправляет на вход X матричного устройства умножения. Инструкция `гер 1 with vsum , shift gam, 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.

## 2.9. Использование Векторного Регистра VR

Исходный текст примера, используемого в данном уроке, содержится в файле `step12.S` в каталоге: `steps-fixed/stepi012`. 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 —и-связыванием
extern "C" void AddBias( short* buff, int size, long bias );

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

int main()
{
    return 33;
    // 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 );

}
```

### Файл “step12.S”

```
.global _AddBias

.p2align 3          // директива для выравнивания секции по чётному адресу.
.data
    // веса для матрицы, единичная диагональ, данные проходят без изменений
```

```
Weights: .quad 1, 1<<16, 1<<32, 1<<48
```

```
.macro SET reg,val
#if __NM4__== 0
    \reg = \val;
#else
    sir = \val;
    \reg = sir;
#endif
.endm

.section .text.AAA
_AddBias:
    arg5 = arg7 - 2;
    push arg0, gr0;
    push arg4, gr4;

    gr4 = [--arg5];      // адрес массива
    gr0 = [--arg5];      // количество 16-разрядных слов в массиве

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

    // gr0 преобразуется из кол-ва 16-разрядных слов
    // в кол-во 64-разрядных векторов
    arg4 = Weights with gr0 >>= 2;

    // веса загружаются в рабочую матрицу
    rep 4 wfifo = [arg4++], ftw, wtw;

    vrg = [--arg5];      // в регистр vrg загружается константный вектор

    // gr0 будет определять количество циклов,
    // где каждый цикл обрабатывает по 32 вектора
    arg0 = gr4 with gr0 >>= 5;
    arg4 = gr4 with gr0--;

Loop:
    if > delayed goto Loop with gr0--;
    // входные вектора, проходя через матричный умножитель, суммируются с vrg
    rep 32 data = [arg0++] with vsum , data, vrg;
    rep 32 [arg4++] = afifo;

    pop arg4, gr4;
    pop arg0, gr0;
    return;
```

### **2.9.1. 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 векторы. 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 NeuroMatrix NM6403. Описание Языка Ассемблера. Инструкция **vr = [--аг5];** загружает вектор констант из стека в 64-разрядный регистр **vr**. Инструкция **гер 32 data = [аг0++]** with **vsum , data, vr;** добавляет вектор констант, хранящийся в **vr**, к результату взвешенного суммирования.

## **2.10. Создание Библиотеки Макросов**

Исходный текст примера, используемого в данном уроке, содержится в файле **step13a.S** в каталоге: **steps-fixed/stepi013a** На примере предыдущего урока демонстрируются возможности использования макросов. Кроме того, в одном из макросов описаны возможности условной компиляции.

**Файл “macros.h”**

```

// Макрос осуществляет копирование одного массива данных
// в другой.
// 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

    gr1 = \Arg3;           // в gr1 загружается количество итераций
    gr1--;                // установка флага для входления в цикл
Loop\@:                 // объявление метки внутри макроса
    // инструкция, содержащая команду отложенного перехода

    if > delayed goto Loop\@ ;
    ar2, gr2 = [\Arg1++] with gr1--;
    [\Arg2++] = ar2, gr2;
.endm

.macro Push_Pop Arg1
.if \Arg1 == 1 // начало блока условной 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
.endm

```

## 2.10.1. 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 регистраных 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`.

```

// функция Copy объявлена внешней с —и-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
    int i=320000000;
    unsigned t1=0x800000;
    unsigned t0=00000;

    unsigned d=0x800000;
    return 320000000/(t1-t0);
}

```

```

.global _Copy // объявление глобальной метки _Copy
#include "macros.hs"
.section .text.AAA
_Copy:
    arg5 = arg7 - 2;

    Push_Pop 0      // подстановка макроса (сохранение регистров)

    arg0 = [--arg5]; // в arg0 адрес исходного массива A
    arg1 = [--arg5]; // в arg1 адрес массива результата B

AAA arg0, arg1, 16 // подстановка макроса AAA

Push_Pop 1      // подстановка макроса (восстановление регистров)
return;

```

## 2.11. 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 процессоров семейства НМС 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:



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 выполнятся параллельно.

### 2.11.1. Оптимизация Ассемблерного Кода

Исходный текст примера, используемого в данном уроке, содержится в файле `step14.S` в каталоге: `steps-fixed/stepi014`. Пример содержит описание основных подходов, используемых для оптимизации ассемблерного кода. Оптимизация программы разбирается на примере копирования блока данных при помощи векторного процессора.

**Файл “`step14.S`”**

```

.global __main

.p2align 3    // директива для выравнивания секции по чётному адресу.
.section .data.MyData
.global A
A: .quad    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

.p2align 3    // директива для выравнивания секции по чётному адресу.
.section .data.MyData1
.global C
C: .space 16<<3

.text
__main:

#if __NM4__== 0
.bbranch;      // псевдокоманда разрешения параллельного выполнения
                // векторных инструкций.
#endif

arg0 = A;
arg4 = C;

rep 16 data = [arg0++] with data;
rep 16 [arg4++] = afifo;

return;
#if __NM4__== 0
.wait;        // запрет параллельного выполнения векторных инструкций
#endif

```

## 2.11.2. 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 А и С 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

## 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 `аг0..аг3`, второй – `аг4..аг7`. Для того чтобы две векторные 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 `аг2`, то вторая должна использовать какой-либо регистр из диапазона `аг4..аг6` (регистр `аг7` хранит адрес вершины стека). Только в этом случае две векторные инструкции могут выполняться параллельно (с учетом выше названных правил).

Если вернуться к примеру, то для параллельного выполнения инструкций

```
гер 16 data = [аг0++] with data;  
гер 16 [аг4++] = afifo;
```

требуется удовлетворить 3 условия: 1) В начале тела функции указать директиву `.branch` 2) Использовать адресные регистры из разных регистровых групп (`аг0` из одной группы, `аг4` из другой). 3) Расположить входной и выходной массивы в банках памяти, расположенных на разных шинах данных (`local` и `global`). Косвенно обе инструкции используют регистр-контейнер `afifo`. Однако он представляет собой двухпортовый буфер, поэтому векторные инструкции могут параллельно записывать и считывать информацию из него.

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



```
гер 32 data = [аг0++] with data + afifo;  
гер 32 [аг4++] = afifo;
```

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

### 2.11.3. 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. Файл имеет расширение `.lds`. Подробное описание файла конфигурации, его структура и синтаксис описаны в [https://home.cs.colorado.edu/~main/cs1300/doc/gnu/ld\\_3.html](https://home.cs.colorado.edu/~main/cs1300/doc/gnu/ld_3.html)

## 3. Работа с векторным процессором на плавающей точке

### 3.1. Простейшая программа

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step01-simple](#). Пример демонстрирует функцию умножения двух чисел с плавающей точкой двойной точности с прибавлением третьего:  $a*b+c$ .

#### Файл “accmul.S”

```
.global _accmul_asm

.data
    result: .double

.text
_accmul_asm:

    arg5 = arg7 - 2;           // настройка арг5 на первый аргумент в стеке

    fpu 0 rep 1 vreg0 = [--arg5]; // чтение из стека аргумента а в vreg0
    fpu 0 rep 1 vreg1 = [--arg5]; // чтение из стека аргумента б в vreg1
    fpu 0 rep 1 vreg2 = [--arg5]; // чтение из стека аргумента с в vreg2
    fpu 0 .double vreg3 = vreg0 * vreg1 + vreg2; // вычисление a*b+c

    arg5 = result;           // запись адреса буфера в арг5
    fpu 0 rep 1 [arg5] = vreg3; // запись результата в буфер
    gr7=[arg5++];           // читаем младшую часть double результата в gr6
    gr6=[arg5];              // читаем старшую часть double результата в gr7

    return;                  // возвращаем результат через пару gr6,gr7
```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

#### Файл “main.cpp”

```
#include "stdio.h"
double accmul_cpp(double a, double b, double c){
    return a*b+c;
}
extern "C" double accmul_asm(double a, double b, double c);

int main(){
    double a=accmul_asm(0.5555505,5.5550055,0.05550555);
    double b=accmul_cpp(0.5555505,5.5550055,0.05550555);
    printf ("%f %f\n", a,b);
    return 0;
}
```

### 3.1.1. Сборка на плате МЦ121.01

```
cd mc12101
nmc-gcc -o test.abs -O2 -Wall -mnmc4-float -g -std=c99 -Wl,-Map=test.map -Wl,
-Tmc12101-nmpu0.lds -I"C:\Module\MC12101/include" -I../../include -I..
-L"C:\Module\MC12101/lib"     ./accmul.S  ./main.cpp -lnm6407int -Wl,--whole-archive
-lnm6407_io_nmc -lmc12101load_nm -Wl,--no-whole-archive
```

#### Расшифровка ключей:

- o test.abs - выходной исполняемый файл
- O2 - уровень оптимизации (0 - нет оптимизации, 2-максимальная)
- mnmc4-float - указывает компилятору собирать код для архитектуры nmc4 с плавающей точкой
- g - сохраняет отладочную информацию в исполняемом файле
- std=c99 - флаг компилятору использовать стандарт c99
- Map=test.map - флаг линкеру генерировать map-файл (карту размещения секций)
- Tmc12101-nmpu0.lds - флаг линкеру использовать файл конфигурации размещения секций
- I"C:\Module\MC12101/include" - путь к заголовочным файлам библиотеки загрузки и обмена
- L"C:\Module\MC12101/lib" - путь к библиотеке БЗИО
- lnm6407int - библиотека прерываний (используется для вызова clock)
- lnm6407\_io\_nmc - библиотека ввода вывода (используется для вызова printf)
- lmc12101load\_nm - библиотека БЗИО

### 3.1.2. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

#### Расшифровка ключей:

- p - включить выводы printf
- R - сброс платы
- a0 - загружать программу на ядро 0 (с плавающей точкой)
- v - возвращать код возврата из функции main

### 3.1.3. результат запуска

```
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.
```



Сборку и запуск программы также можно осуществить короткими командами `make` и `make run`

### 3.1.4. Комментарии к Примеру

По соглашению о вызове функций при входе в функцию аргументы будут располагаться в стеке следующим образом:

```
00038020: 33fe344e 3fac6b39 (c=0.05550555)
00038022: 5c9e6688 40163853 (b=5.5550055)
00038024: d798d8a9 3fe1c711 (a=0.5555505)
ар5 --> 00038026: 0000035c ffff00e1 (pc pswr)
ар7 --> 00038028:xxxxxxxx xxxxxxxx (вершина стека)
```

Аргументы будем считывать длинными 64-р. словами в порядке их следования с помощью регистра `ар5`. Для доступа к первому параметру с учетом команды преддекремента `--ар5` смещаем указатель от вершины стека `ар7` на 2 (пропуская регистры состояния `pswr` и адрес возврата `pc`):

```
ар5 = ар7 - 2; // настройка ар5 на первый аргумент в стеке
```

В следующих трех командах происходит чтение аргументов из стека и запись в векторные регистры: `vreg0`, `vreg1` и `vreg2` вычислительной ячейки `fpu 0`. Счетчик повторений `гер 1` настраивает чтение в регистр-контейнер `vreg*` только одного 64-разрядного слова.

```
fpu 0 гер 1 vreg0 = [--ар5]; // чтение из стека аргумента а в vreg0
fpu 0 гер 1 vreg1 = [--ар5]; // чтение из стека аргумента б в vreg1
fpu 0 гер 1 vreg2 = [--ар5]; // чтение из стека аргумента с в vreg2
```

Осуществляем арифметическую операцию `a*b+c` в формате `double`:

```
fpu 0 .double vreg3 = vreg0 * vreg1 + vreg2; // вычисление a*b+c
```

Сохраняем результат в память (в буфер `result`)

```
ар5 = result; // запись адреса буфера в ар5
fpu 0 гер 1 [ар5] = vreg3; // запись результата в буфер
```

Согласно "конвенции о вызовах функций" возврат 64р. значений осуществляется через пару регистров: `рг6,рг7`. Поэтому заполняем их содержимым `result`:

```

gr7=[arg5++];           // читаем младшую часть double результата в gr6
gr6=[arg5];             // читаем старшую часть double результата в gr7
return;

```

Более подробно о векторных регистрах и структуре векторного сопроцессора будет рассказано в следующих главах.

## 3.2. Краткое введение в архитектуру

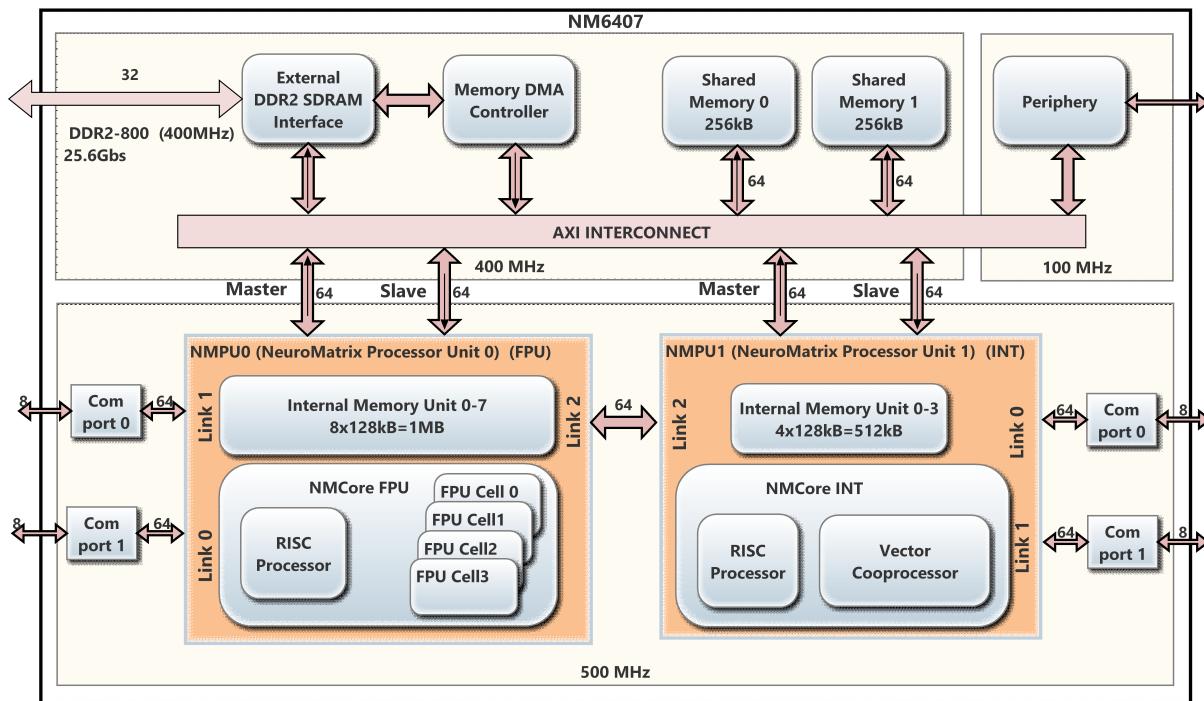


Рисунок 6. Структура процессора 1879ВМ6Я(NM6407)

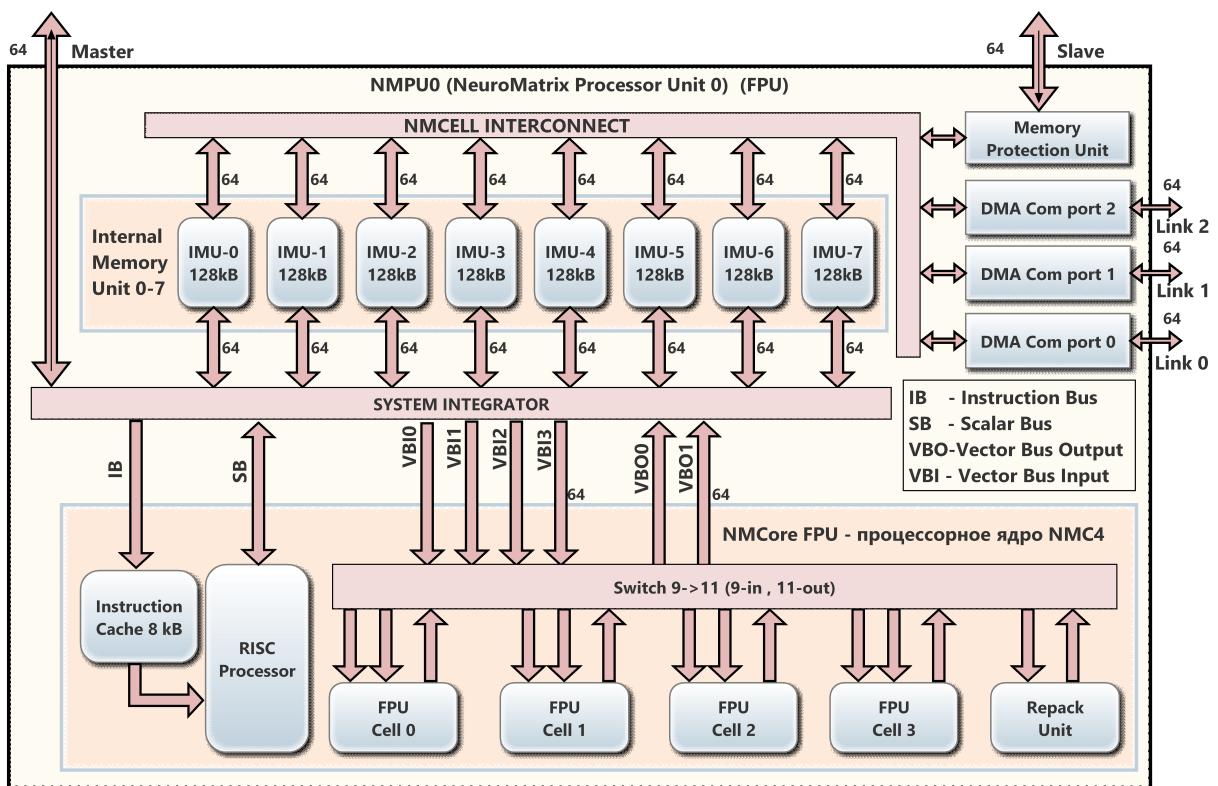


Рисунок 7. Процессорная система NMPU0

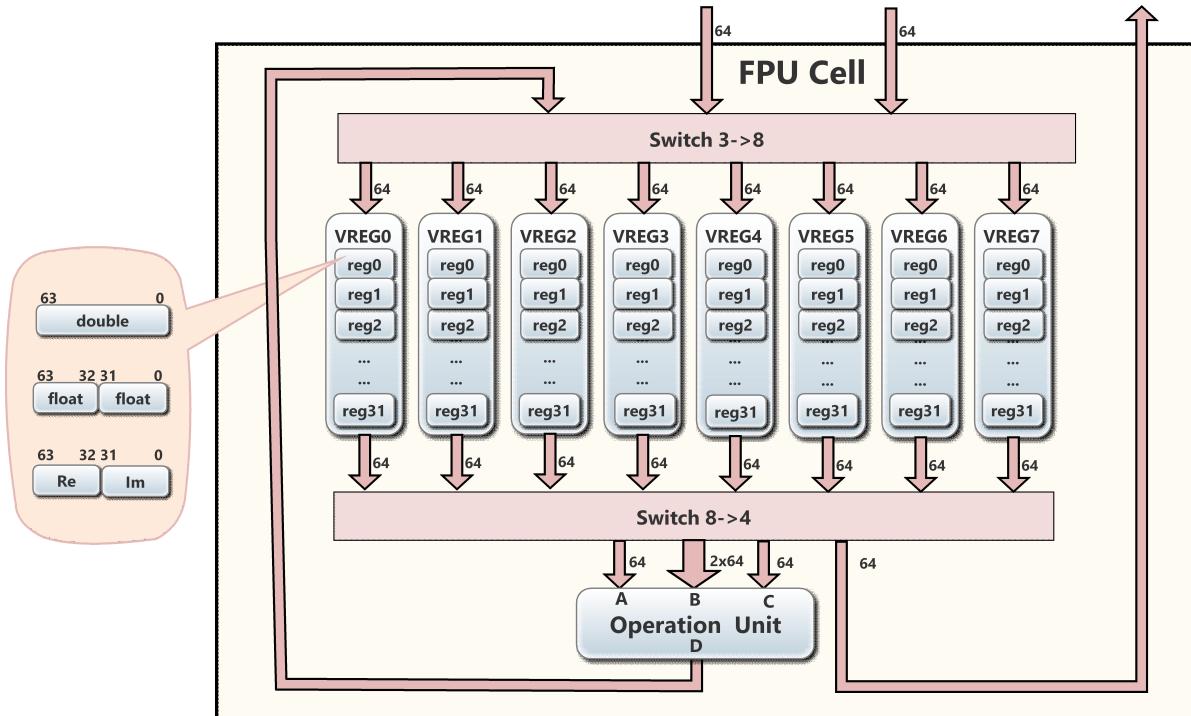


Рисунок 8. Процессорная ячейка FPU

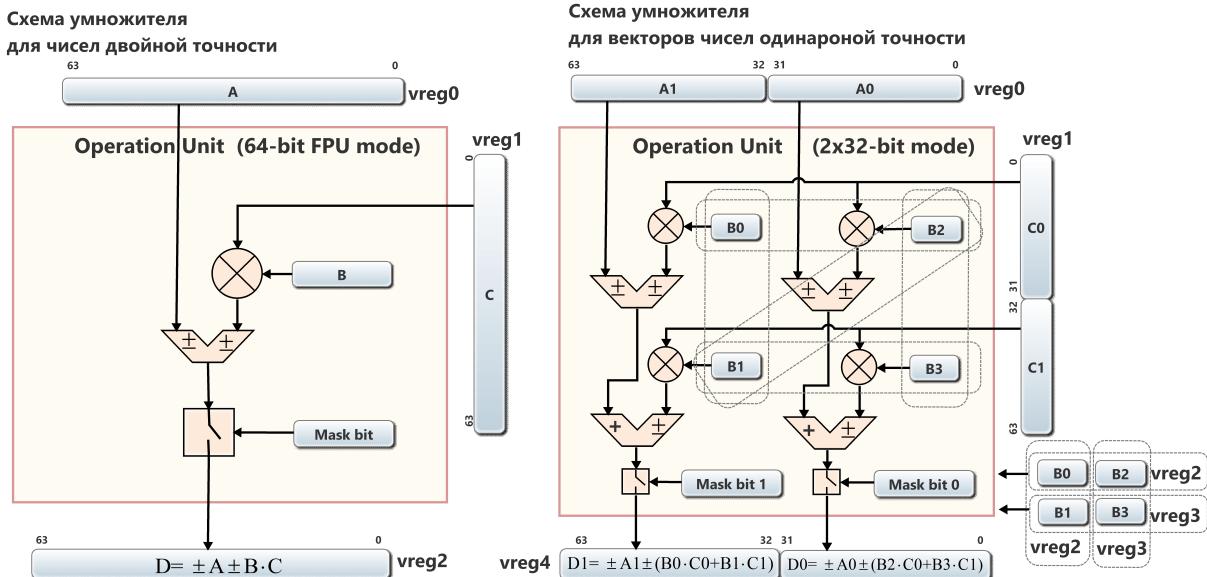


Рисунок 9. Структура операционного узла

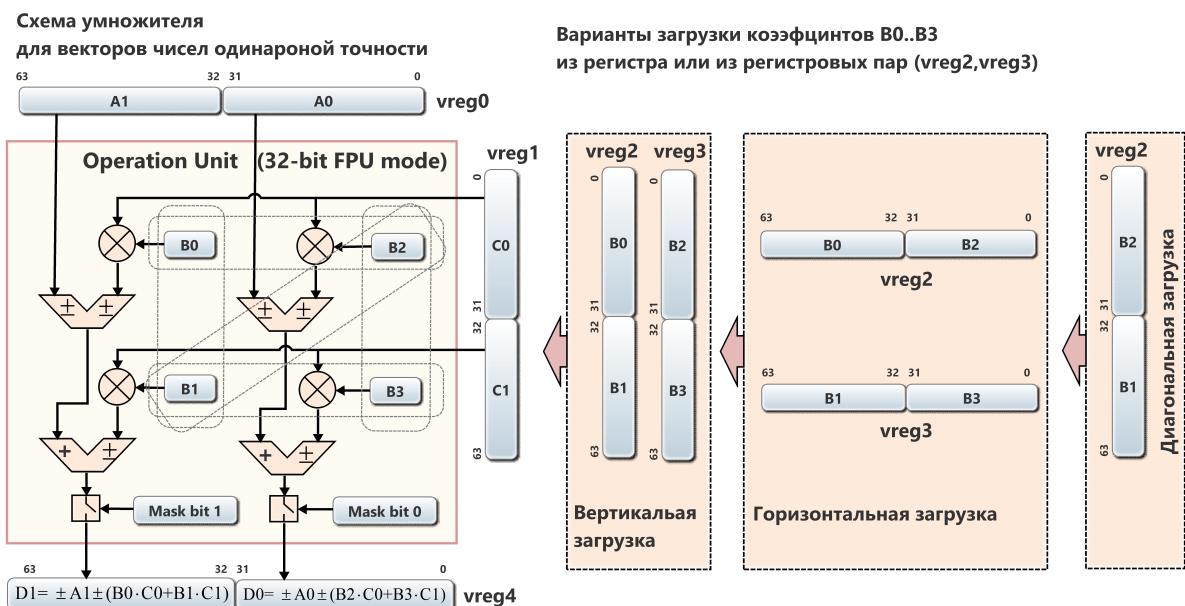


Рисунок 10. Способы загрузки умножителя

Операции умножения

image::06fig-NM6407-OU-modes.png [image,width=608,height=479]

### 3.3. Поэлементная операция $a^*b+c$ над массивом данных

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step02a](#). Пример демонстрирует работу над массивами типа double с поэлементным умножением `d[i]=a[i]*b[i]+c[i]`.

Файл “`accmul.S`”

```

.global _accmul_asm

.data
    result: .quad

.text
_accmul_asm:

    ar5 = ar7 - 2;
    push ar0,gr0;
    push ar1,gr1;
    push ar2,gr2;
    push ar3,gr3;

    ar0 = [--ar5]; // a
    ar1 = [--ar5]; // b
    ar2 = [--ar5]; // c
    ar3 = [--ar5]; // d
    gr7 = [--ar5]; // size

    gr7--;           // loop counter --
    ar5 = ar3;      // store dst address
next_accmul:
    fpu 0 rep 1 vreg0 = [ar0++];
    fpu 0 rep 1 vreg1 = [ar1++];
    fpu 0 rep 1 vreg2 = [ar2++];
    fpu 0 .double vreg3 = vreg0*vreg1+vreg2;
    fpu 0 rep 1 [ar3++] = vreg3;
    if <>0 goto next_accmul with gr7--;

    pop ar3,gr3;
    pop ar2,gr2;
    pop ar1,gr1;
    pop ar0,gr0;

    return;

```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

#### Файл “main.cpp”

```

#include "time.h"
#include "stdio.h"
extern "C" void accmul_asm(double* a, double* b, double* c, double* d, int size);

void accmul_cpp(double* a, double* b, double* c, double* d, int size){
    for(int i=0; i<size; i++){
        d[i]=a[i]*b[i]+c[i];
    }
}

#define SIZE 1024
double buffer_a[SIZE];
double buffer_b[SIZE];
double buffer_c[SIZE];
double buffer_d[SIZE];
double buffer_D[SIZE];

int main(){
    int i;
    clock_t t0, t1;
    // инициализация
    for (i=0; i<SIZE; i++){
        buffer_a[i]=i+1;
        buffer_b[i]=2;
        buffer_c[i]=1000;
    }

    t0=clock();
    t1=clock();
    clock_t dt=t1-t0; // Оценка коррекции замера времени

    t0=clock();
    accmul_asm(buffer_a, buffer_b, buffer_c, buffer_d, SIZE);
    t1=clock();
    clock_t dt_asm=t1-t0-dt; // время аsm-функции

    t0=clock();
    accmul_cpp(buffer_a, buffer_b, buffer_c, buffer_D, SIZE);
    t1=clock();
    clock_t dt_cpp=t1-t0-dt; // время cpp-функции

    printf(" lang: %12s %12s\n", "buffer_d(asm)", "buffer_D(c++)");
    printf(" res: %e %e\n", buffer_d[0], buffer_D[0]);
    printf(" ... ... \n");
    printf(" res: %e %e\n", buffer_d[SIZE-1], buffer_D[SIZE-1]);
    printf(" clocks: %12ld %12ld\n", dt_asm, dt_cpp);
    printf("cl/elem: %12.2f %12.2f\n", 1.0*dt_asm/SIZE, 1.0*dt_cpp/SIZE);

    return 0;
}

```

### 3.3.1. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

### 3.3.2. результат запуска

```
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.
```

### 3.3.3. Комментарии к Примеру

Модифицируем пример из урока для работы с массивами, окружив соответствующие блоки циклом .

## 3.4. Поэлементная операция $a^*b+c$ над массивом данных. SIMD - оптимизация

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step02b](#). Пример демонстрирует оптимизацию работы над массивами типа double с поэлементным умножением  $d[i]=a[i]*b[i]+c[i]$  путем использования SIMD инструкций.

### Файл “accmul.s”

```
//#include "printx.h"
//PRINTF1("arg0=%x\n",arg0)
//PRINTF1("arg1=%x\n",arg1)
//PRINTF1("arg2=%x\n",arg2)
//PRINTF1("arg3=%x\n",arg3)
//PRINTF1("gr3=%x\n",gr3)

.global _accmul_asm
.data
    result: .quad

.text
_accmul_asm:

    arg5 = arg7 - 2;
    push arg0,gr0;
    push arg1,gr1;
```

```

push ar2,gr2;
push ar3,gr3;

ar0 = [--ar5]; // a
ar1 = [--ar5]; // b
ar2 = [--ar5]; // c
ar3 = [--ar5]; // d
gr3 = [--ar5]; // size
gr7 = gr3 >>5; // gr7=size/32
gr3 = gr3<<27 ;
gr3 = gr3>>27 ; // gr3=size%32

with gr7;
if =0 goto accmul_tail with gr7--;

accmul_rep32:
    fpu 0 rep 32 vreg0 = [ar0++];
    fpu 0 rep 32 vreg1 = [ar1++];
    fpu 0 rep 32 vreg2 = [ar2++];
    if > delayed goto accmul_rep32 with gr7--;
    fpu 0 .double vreg3 = vreg0*vreg1+vreg2;
    fpu 0 rep 32 [ar3++] = vreg3;

accmul_tail:
with gr3;
if =0 delayed goto accmul_end with gr3--;
    vlen = gr3;
    nul;

fpu 0 rep vlen vreg0 = [ar0++];
fpu 0 rep vlen vreg1 = [ar1++];
fpu 0 rep vlen vreg2 = [ar2++];
fpu 0 .double vreg3 = vreg0*vreg1+vreg2;
fpu 0 rep vlen [ar3++] = vreg3;

accmul_end:

pop ar3,gr3;
pop ar2,gr2;
pop ar1,gr1;
pop ar0,gr0;

return;

```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

## Файл “main.cpp”

```

#include "time.h"
#include "stdio.h"
extern "C" void accmul_asm(double* a, double* b, double* c, double* d, int size);

void accmul_cpp(double* a, double* b, double* c, double* d, int size){
    for(int i=0; i<size; i++){
        d[i]=a[i]*b[i]+c[i];
    }
}

#define SIZE 1024
double buffer_a[SIZE];
double buffer_b[SIZE];
double buffer_c[SIZE];
double buffer_d[SIZE];
double buffer_D[SIZE];

int main(){
    int i;
    clock_t t0, t1;
    // инициализация
    for (i=0; i<SIZE; i++){
        buffer_a[i]=i+1;
        buffer_b[i]=2;
        buffer_c[i]=1000;
    }

    t0=clock();
    t1=clock();
    clock_t dt=t1-t0; // Оценка коррекции замера времени

    t0=clock();
    accmul_asm(buffer_a, buffer_b, buffer_c, buffer_d, SIZE);
    t1=clock();
    clock_t dt_asm=t1-t0-dt; // время аsm-функции

    t0=clock();
    accmul_cpp(buffer_a, buffer_b, buffer_c, buffer_D, SIZE);
    t1=clock();
    clock_t dt_cpp=t1-t0-dt; // время cpp-функции

    printf(" lang: %12s %12s\n", "buffer_d(asm)", "buffer_D(c++)");
    printf(" res: %e %e\n", buffer_d[0], buffer_D[0]);
    printf(" ... ... \n");
    printf(" res: %e %e\n", buffer_d[SIZE-1], buffer_D[SIZE-1]);
    printf(" clocks: %12ld %12ld\n", dt_asm, dt_cpp);
    printf("cl/elem: %12.2f %12.2f\n", 1.0*dt_asm/SIZE, 1.0*dt_cpp/SIZE);

    return 0;
}

```

### 3.4.1. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

### 3.4.2. результат запуска

```
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.
```

### 3.4.3. Комментарии к Примеру

Модифицируем пример из урока для работы с массивами, окружив соответствующие блоки циклом .

## 3.5. Поэлементная операция $a^*b+c$ над массивом данных. Оптимизация по памяти

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step02b](#). Пример демонстрирует оптимизацию работы над массивами типа double с поэлементным умножением  $d[i]=a[i]*b[i]+c[i]$  путем использования SIMD инструкций.

#### Файл “accmul.s”

```
//#include "printx.hs"
//PRINTF1("arg0=%x\n",arg0)
//PRINTF1("arg1=%x\n",arg1)
//PRINTF1("arg2=%x\n",arg2)
//PRINTF1("arg3=%x\n",arg3)
//PRINTF1("gr3=%x\n",gr3)

.global _accmul_asm
.data
    result: .quad

.text
_accmul_asm:

    arg5 = arg7 - 2;
    push arg0,gr0;
    push arg1,gr1;
```

```

push ar2,gr2;
push ar3,gr3;

ar0 = [--ar5]; // a
ar1 = [--ar5]; // b
ar2 = [--ar5]; // c
ar3 = [--ar5]; // d
gr3 = [--ar5]; // size
gr7 = gr3 >>5; // gr7=size/32
gr3 = gr3<<27 ;
gr3 = gr3>>27 ; // gr3=size%32

with gr7;
if =0 goto accmul_tail with gr7--;

accmul_rep32:
    fpu 0 rep 32 vreg0 = [ar0++];
    fpu 0 rep 32 vreg1 = [ar1++];
    fpu 0 rep 32 vreg2 = [ar2++];
    if > delayed goto accmul_rep32 with gr7--;
    fpu 0 .double vreg3 = vreg0*vreg1+vreg2;
    fpu 0 rep 32 [ar3++] = vreg3;

accmul_tail:
with gr3;
if =0 delayed goto accmul_end with gr3--;
    vlen = gr3;
    nul;

fpu 0 rep vlen vreg0 = [ar0++];
fpu 0 rep vlen vreg1 = [ar1++];
fpu 0 rep vlen vreg2 = [ar2++];
fpu 0 .double vreg3 = vreg0*vreg1+vreg2;
fpu 0 rep vlen [ar3++] = vreg3;

accmul_end:

pop ar3,gr3;
pop ar2,gr2;
pop ar1,gr1;
pop ar0,gr0;

return;

```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

## Файл “main.cpp”

```

#include "time.h"
#include "stdio.h"
extern "C" void accmul_asm(double* a, double* b, double* c, double* d, int size);

void accmul_cpp(double* a, double* b, double* c, double* d, int size){
    for(int i=0; i<size; i++){
        d[i]=a[i]*b[i]+c[i];
    }
}

#define SIZE 1024
__attribute__((section(".data imu1"))) double buffer_a[SIZE];
__attribute__((section(".data imu2"))) double buffer_b[SIZE];
__attribute__((section(".data imu3"))) double buffer_c[SIZE];
__attribute__((section(".data imu4"))) double buffer_d[SIZE];
double buffer_D[SIZE];

int main(){
    int i;
    clock_t t0, t1;
    // инициализация
    for (i=0; i<SIZE; i++){
        buffer_a[i]=i+1;
        buffer_b[i]=2;
        buffer_c[i]=1000;
    }

    t0=clock();
    t1=clock();
    clock_t dt=t1-t0; // Оценка коррекции замера времени

    t0=clock();
    accmul_asm(buffer_a, buffer_b, buffer_c, buffer_d, SIZE);
    t1=clock();
    clock_t dt_asm=t1-t0-dt; // время асс-функции

    t0=clock();
    accmul_cpp(buffer_a, buffer_b, buffer_c, buffer_D, SIZE);
    t1=clock();
    clock_t dt_cpp=t1-t0-dt; // время спп-функции

    printf(" lang: %12s %12s\n", "buffer_d(asm)", "buffer_D(c++)");
    printf(" res: %e %e\n", buffer_d[0], buffer_D[0]);
    printf(" ... ... \n");
    printf(" res: %e %e\n", buffer_d[SIZE-1], buffer_D[SIZE-1]);
    printf(" clocks: %12ld %12ld\n", dt_asm, dt_cpp);
    printf("cl/elem: %12.2f %12.2f\n", 1.0*dt_asm/SIZE, 1.0*dt_cpp/SIZE);

    return 0;
}

```

### 3.5.1. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

### 3.5.2. результат запуска

```
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.
```

### 3.5.3. Комментарии к Примеру

Модифицируем пример из урока для работы с массивами, окружив соответствующие блоки циклом .

## 3.6. Поэлементная операция $a^*b+c$ над массивом данных. Анализ производительности

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step02b](#). Пример демонстрирует оптимизацию работу над массивами типа double с поэлементным умножением  $d[i]=a[i]*b[i]+c[i]$  путем использования SIMD инструкций.

### Файл “accmul.s”

```
//#include "printx.hs"
//PRINTF1("ar0=%x\n",ar0)
//PRINTF1("ar1=%x\n",ar1)
//PRINTF1("ar2=%x\n",ar2)
//PRINTF1("ar3=%x\n",ar3)
//PRINTF1("gr3=%x\n",gr3)

.global _accmul_asm
.data
    result: .quad

.text
_accmul_asm:

    ar5 = ar7 - 2;
    push ar0,gr0;
    push ar1,gr1;
```

```

push ar2,gr2;
push ar3,gr3;

ar0 = [--ar5]; // a
ar1 = [--ar5]; // b
ar2 = [--ar5]; // c
ar3 = [--ar5]; // d
gr3 = [--ar5]; // size
gr7 = gr3 >>5; // gr7=size/32
gr3 = gr3<<27 ;
gr3 = gr3>>27 ; // gr3=size%32

with gr7;
if =0 goto accmul_tail with gr7--;

accmul_rep32:
    fpu 0 rep 32 vreg0 = [ar0++];
    fpu 0 rep 32 vreg1 = [ar1++];
    fpu 0 .double vreg3 = vreg0*vreg1;
    fpu 1 rep 32 vreg2 = [ar2++];
    fpu 1 vreg0 = fpu 0 vreg3;
    if > delayed goto accmul_rep32 with gr7--;
    fpu 1 .double vreg3 = vreg0+vreg2;
    fpu 1 rep 32 [ar3++] = vreg3;

accmul_tail:
with gr3;
if =0 delayed goto accmul_end with gr3--;
    vlen = gr3;
    vnul;

    fpu 0 rep vlen vreg0 = [ar0++];
    fpu 0 rep vlen vreg1 = [ar1++];
    fpu 0 .double vreg3 = vreg0*vreg1;
    fpu 1 vreg0 = fpu 0 vreg3;
    fpu 1 rep vlen vreg2 = [ar2++];
    fpu 1 .double vreg3 = vreg0+vreg2;
    fpu 1 rep vlen [ar3++] = vreg3;

accmul_end:

pop ar3,gr3;
pop ar2,gr2;
pop ar1,gr1;
pop ar0,gr0;

return;

```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

#### Файл “main.cpp”

```
#include "time.h"
#include "stdio.h"

extern "C" void accmul_asm(double* a, double* b, double* c, double* d, int size);

#define SIZE 2*1024
__attribute__((section(".data imu1"))) double buffer_a[SIZE];
__attribute__((section(".data imu2"))) double buffer_b[SIZE];
__attribute__((section(".data imu3"))) double buffer_c[SIZE];
__attribute__((section(".data imu4"))) double buffer_d[SIZE];

int main(){
    size_t t0=clock();
    size_t t1=clock();
    size_t dt=t1-t0;
    printf("correction=%d\n",dt);
    volatile clock_t dt_asm;
    float speed=0, prev_speed=1000;
    for (int sz=4; sz<=SIZE; sz+=4) {
        t0=clock();
        accmul_asm(buffer_a, buffer_b, buffer_c, buffer_d, sz);
        t1=clock();
        dt_asm=t1-t0-dt;

        speed=dt_asm*1.0/sz;
        if (speed<prev_speed-0.05){
            prev_speed=speed;
            printf("size=%4d clocks/element=%f\n",sz,speed);
        }
    }
    printf("size=%4d clocks/element=%f\n",SIZE,speed);
    return 0;
}
```

#### 3.6.1. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

#### 3.6.2. результат запуска

```
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.
```

### 3.6.3. Комментарии к Примеру

Модифицируем пример из урока для работы с массивами, окружив соответствующие блоки циклом .

## 3.7. Поэлементная операция $|a+b|$ над массивом данных. Оптимизация по памяти

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step02b](#). Пример демонстрирует оптимизацию работу над массивами типа double с поэлементным умножением  $d[i]=a[i]*b[i]+c[i]$  путем использования SIMD инструкций.

### Файл “absum.S”

```
//#include "printx.h"
//PRINTF1("arg0=%x\n",arg0)
//PRINTF1("arg1=%x\n",arg1)
//PRINTF1("arg2=%x\n",arg2)
//PRINTF1("arg3=%x\n",arg3)
//PRINTF1("gr3=%x\n",gr3)

.global _abssum_asm
.data
    result: .quad

.text
_abssum_asm:

    arg5 = arg7 - 2;
    push arg0,gr0;
    push arg1,gr1;
    push arg2,gr2;
    push arg3,gr3;

    arg0 = [--arg5]; // a
    arg1 = [--arg5]; // b
    arg2 = [--arg5]; // c
```

```

gr3 = [--ar5]; // size
gr7 = gr3 >>5; // gr7=size/32
gr3 = gr3<<27;
gr3 = gr3>>27; // gr3=size%32

with gr7;
if =0 goto abssum_tail with gr7--;

abssum_rep32:
    fpu 0 rep 32 vreg0 = [ar0++];
    fpu 0 rep 32 vreg1 = [ar1++];
    fpu 0 .double vreg2 = vreg0+vreg1;
    fpu 0 .double vreg3 = /vreg2/;
    fpu 0 rep 32 [ar2++]= vreg3;
    if > goto abssum_rep32 with gr7--;

abssum_tail:
with gr3;
if =0 delayed goto abssum_end with gr3--;
    vlen = gr3;
    vnul;

    fpu 0 rep vlen vreg0 = [ar0++];
    fpu 0 rep vlen vreg1 = [ar1++];
    fpu 0 .double vreg2 = vreg0+vreg1;
    fpu 0 .double vreg3 = /vreg2/;
    fpu 0 rep vlen [ar2++]= vreg3;

abssum_end:
    pop ar3,gr3;
    pop ar2,gr2;
    pop ar1,gr1;
    pop ar0,gr0;

return;

```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

#### Файл “main.cpp”

```

#include "time.h"
#include "stdio.h"

extern "C" void abssum_asm(double* a, double* b, double* c, int size);

void abssum_cpp(double* a, double* b, double* c, int size){
    int i=0;
    for(i=0; i<size; i++){
        double sum=a[i]+b[i];
        c[i]=sum>0?sum:-sum;
    }
}

#define SIZE 2*1024
__attribute__((section(".data imu1"))) double buffer_a[SIZE];
__attribute__((section(".data imu2"))) double buffer_b[SIZE];
__attribute__((section(".data imu3"))) double buffer_c[SIZE];
__attribute__((section(".data imu4"))) double buffer_d[SIZE];

int main(){
    size_t t0=clock();
    size_t t1=clock();
    size_t dt=t1-t0;
    printf("correction=%d\n",dt);

    volatile clock_t dt_asm;
    float speed=0, prev_speed=1000;
    for (int sz=4; sz<=SIZE; sz+=4) {
        t0=clock();
        abssum_asm(buffer_a, buffer_b, buffer_c, sz);
        t1=clock();
        dt_asm=t1-t0-dt;

        speed=dt_asm*1.0/sz;
        if (speed<prev_speed-0.01){
            prev_speed=speed;
            printf("size=%4d clocks/element=%f\n",sz,speed);
        }
    }
    printf("size=%4d clocks/element=%f\n",SIZE,speed);
    return 0;
}

```

### 3.7.1. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

### 3.7.2. результат запуска

```
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.
```

### 3.7.3. Комментарии к Примеру

Модифицируем пример из урока для работы с массивами, окружив соответствующие блоки циклом .

## 3.8. Поэлементная операция $|a+b|$ над массивом данных. Оптимизация по вычислительным ячейкам

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step02b](#). Пример демонстрирует оптимизацию работу над массивами типа double с поэлементным умножением  $d[i]=a[i]*b[i]+c[i]$  путем использования SIMD инструкций.

Файл “absum.S”

```
//#include "printx.hs"
//PRINTF1("arg0=%x\n",arg0)
//PRINTF1("arg1=%x\n",arg1)
//PRINTF1("arg2=%x\n",arg2)
//PRINTF1("arg3=%x\n",arg3)
//PRINTF1("gr3=%x\n",gr3)

.global _abssum_asm
.data
    result: .quad

.text
_abssum_asm:

    arg5 = arg7 - 2;
    push arg0,gr0;
    push arg1,gr1;
    push arg2,gr2;
    push arg3,gr3;
```

```

ar0 = [--ar5]; // a
ar1 = [--ar5]; // b
ar2 = [--ar5]; // c
gr3 = [--ar5]; // size
gr7 = gr3 >>5; // gr7=size/32
gr3 = gr3<<27 ;
gr3 = gr3>>27 ; // gr3=size%32

with gr7;
if =0 goto abssum_tail with gr7--;

abssum_rep32:
    fpu 0 rep 32 vreg0 = [ar0++];
    fpu 0 rep 32 vreg1 = [ar1++];
    fpu 0 .double vreg2 = vreg0+vreg1;
    fpu 1 vreg0 = vreg2;
    fpu 1 .double vreg3 = /vreg0/;
    fpu 1 rep 32 [ar2++]= vreg3;
    if > goto abssum_rep32 with gr7--;

abssum_tail:
with gr3;
if =0 delayed goto abssum_end with gr3--;
    vlen = gr3;
    vnul;

    fpu 0 rep vlen vreg0 = [ar0++];
    fpu 0 rep vlen vreg1 = [ar1++];
    fpu 0 .double vreg2 = vreg0+vreg1;
    fpu 1 vreg0 = vreg2;
    fpu 1 .double vreg3 = /vreg0/;
    fpu 1 rep vlen [ar2++]= vreg3;

abssum_end:

pop ar3,gr3;
pop ar2,gr2;
pop ar1,gr1;
pop ar0,gr0;

return;

```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

### Файл “main.cpp”

```

#include "time.h"
#include "stdio.h"

extern "C" void abssum_asm(double* a, double* b, double* c, int size);

void abssum_cpp(double* a, double* b, double* c, int size){
    int i=0;
    for(i=0; i<size; i++){
        double sum=a[i]+b[i];
        c[i]=sum>0?sum:-sum;
    }
}

#define SIZE 2*1024
__attribute__((section(".data imu1"))) double buffer_a[SIZE];
__attribute__((section(".data imu2"))) double buffer_b[SIZE];
__attribute__((section(".data imu3"))) double buffer_c[SIZE];
__attribute__((section(".data imu4"))) double buffer_d[SIZE];

int main(){
    size_t t0=clock();
    size_t t1=clock();
    size_t dt=t1-t0;
    printf("correction=%d\n",dt);

    volatile clock_t dt_asm;
    float speed=0, prev_speed=1000;
    for (int sz=4; sz<=SIZE; sz+=4) {
        t0=clock();
        abssum_asm(buffer_a, buffer_b, buffer_c, sz);
        t1=clock();
        dt_asm=t1-t0-dt;

        speed=dt_asm*1.0/sz;
        if (speed<prev_speed-0.01){
            prev_speed=speed;
            printf("size=%4d clocks/element=%f\n",sz,speed);
        }
    }
    printf("size=%4d clocks/element=%f\n",SIZE,speed);
    return 0;
}

```

### 3.8.1. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

### 3.8.2. результат запуска

```
mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.
```

### 3.8.3. Комментарии к Примеру

Модифицируем пример из урока для работы с массивами, окружив соответствующие блоки циклом .

## 3.9. Поэлементная операция $|a+b|$ над массивом данных. Оптимизация по потокам

Исходный текст примера, используемого в данном уроке, содержится в каталоге: [/steps-float/step02b](#). Пример демонстрирует оптимизацию работы над массивами типа double с поэлементным умножением  $d[i]=a[i]*b[i]+c[i]$  путем использования SIMD инструкций.

#### Файл “absum.S”

```
//#include "printx.hs"
//PRINTF1("ar0=%x\n",ar0)
//PRINTF1("ar1=%x\n",ar1)
//PRINTF1("ar2=%x\n",ar2)
//PRINTF1("ar3=%x\n",ar3)
//PRINTF1("gr3=%x\n",gr3)

.global _twin_abssum_asm
.data
    result: .quad

.text
_twin_abssum:

    ar5 = ar7 - 2;
    push ar0,gr0;
    push ar1,gr1;
    push ar2,gr2;
    push ar3,gr3;
    push ar4,gr4;
    push ar5,gr5;
    push ar6,gr6;
```

```

ar0 = [--ar5]; // a0
ar1 = [--ar5]; // b0
ar2 = [--ar5]; // c0

ar3 = [--ar5]; // a1
ar4 = [--ar5]; // b1
ar6 = [--ar5]; // c1

gr3 = [--ar5]; // size
gr7 = gr3 >>5; // gr7=size/32
gr3 = gr3<<27 ;
gr3 = gr3>>27 ; // gr3=size%32

with gr7;
if =0 goto twin_abssum_tail with gr7--;

twin_abssum_rep32:
    fpu 0 rep 32 vreg0 = [ar0++];
    fpu 0 rep 32 vreg1 = [ar1++];
    fpu 0 .double vreg2 = vreg0+vreg1;
    fpu 1 vreg0 = vreg2;
    fpu 1 .double vreg3 = /vreg0/;
    fpu 1 rep 32 [ar2++]= vreg3;

    fpu 2 rep 32 vreg0 = [ar3++];
    fpu 2 rep 32 vreg1 = [ar4++];
    fpu 2 .double vreg2 = vreg0+vreg1;
    fpu 3 vreg0 = vreg2;
    fpu 3 .double vreg3 = /vreg0/;
    fpu 3 rep 32 [ar6++]= vreg3;
    if > goto twin_abssum_rep32 with gr7--;

twin_abssum_tail:
with gr3;
if =0 delayed goto twin_abssum_end with gr3--;
    vlen = gr3;
    vnul;

    fpu 0 rep vlen vreg0 = [ar0++];
    fpu 0 rep vlen vreg1 = [ar1++];
    fpu 0 .double vreg2 = vreg0+vreg1;
    fpu 1 vreg0 = vreg2;
    fpu 1 .double vreg3 = /vreg0/;
    fpu 1 rep vlen [ar2++]= vreg3;

    fpu 2 rep vlen vreg0 = [ar3++];
    fpu 2 rep vlen vreg1 = [ar4++];
    fpu 2 .double vreg2 = vreg0+vreg1;

```

```
fpu 3 vreg0 = vreg2;
fpu 3 .double vreg3 = /vreg0/;
fpu 3 rep vlen [arg6++]= vreg3;
```

```
twin_abssum_end:
```

```
pop arg6,gr6;
pop arg5,gr5;
pop arg4,gr4;
pop arg3,gr3;
pop arg2,gr2;
pop arg1,gr1;
pop arg0,gr0;
```

```
return;
```

Вызовем функцию `accmul_asm` из `main`, а также ее эквивалент на языке C++ - `accmul_cpp`:

#### Файл “main.cpp”

```
#include "time.h"
#include "stdio.h"

extern "C" void twin_abssum_asm(double* a0, double* b0, double* c0, double* a1,
double* b1, double* c1, int size);

void twin_abssum_cpp(double* a0, double* b0, double* c0, double* a1, double* b1,
double* c1, int size){
    int i=0;
    for(i=0; i<size; i++){
        double sum=a0[i]+b0[i];
        c0[i]=sum>0?sum:-sum;
        sum=a1[i]+b1[i];
        c1[i]=sum>0?sum:-sum;
    }
}

#define SIZE 2*1024

__attribute__((section(".data imu1"))) double buffer_a0[SIZE];
__attribute__((section(".data imu2"))) double buffer_b0[SIZE];
__attribute__((section(".data imu3"))) double buffer_c0[SIZE];
__attribute__((section(".data imu4"))) double buffer_a1[SIZE];
__attribute__((section(".data imu5"))) double buffer_b1[SIZE];
__attribute__((section(".data imu6"))) double buffer_c1[SIZE];

int main(){
    size_t t0=clock();
```

```

size_t t1=clock();
size_t dt=t1-t0;
printf("correction=%d\n",dt);

volatile clock_t dt_asm;
float speed=0, prev_speed=1000;
for (int sz=4; sz<=SIZE; sz+=4) {
    t0=clock();
    twin_abssum_asm(buffer_a0, buffer_b0, buffer_c0, buffer_a1, buffer_b1,
buffer_c1, sz);
    t1=clock();
    dt_asm=t1-t0-dt;

    speed=dt_asm*0.5/sz;
    if (speed<prev_speed-0.01){
        prev_speed=speed;
        printf("size=%4d clocks/element=%f\n",sz,speed);
    }
}
printf("size=%4d clocks/element=%f\n",SIZE,speed);
return 0;
}

```

### 3.9.1. запуск на плате МЦ121.01

```
mc12101run -p -R -a0 -v test.abs
```

### 3.9.2. результат запуска

```

mc12101run -p -R -a0 -v test.abs
Batch loader for MC121.01 v6.1. (C) 2022 RC Module Inc.
Performing reset...
Done.
Firmware v6.1
Start user program on core 0...
3.141592 3.141592
test.abs :: Core 0 return 0 = 0x0.

```

### 3.9.3. Комментарии к Примеру

Модифицируем пример из урока для работы с массивами, окружив соответствующие блоки циклом .