



NM6403 Software Development Kit

Getting Started with NeuroMatrix® NM6403

Version 1.1

Document ID: 30002-01 33 02 A



Module® and **NeuroMatrix®** are registered trademarks of MODULE Research Center.
All other trademarks are the exclusive property of their respective owners.

Contents

PREFACE.....	1
ABOUT THIS MANUAL	1
ORGANIZATION	1
TYPOGRAPHICAL CONVENTIONS	2
FEEDBACK.....	2
Feedback on This Manual	2
Feedback on NM6403 Software Development Kit	2
HOW TO WRITE PROGRAMS IN NM6403 ASSEMBLY.....	1-1
1.1 LESSON 1: A SIMPLE PROGRAM IN NM6403 ASSEMBLY.....	1-1
1.2 LESSON 2: MEMORY ACCESS	1-4
1.3 LESSON 3: MAKING LOOPS.....	1-6
1.4 LESSON 3A: LOOP OPTIMIZATION.....	1-8
1.5 LESSON 4: COPYING A DATA ARRAY ON RISC-CORE.....	1-10
1.6 LESSON 4A: COPYING A DATA ARRAY ON VCP	1-12
1.7 LESSON 5: ARITHMETICAL OPERATIONS ON VECTOR ALU	1-15
1.8 LESSON 6: WEIGHTED ACCUMULATION	1-18
1.9 LESSON 6A: WEIGHTED ACCUMULATION (CONTINUATION 1)	1-21
1.10 LESSON 6B: WEIGHTED ACCUMULATION (CONTINUATION 2)	1-23
1.11 LESSON 7: HOW TO CALL AN ASSEMBLY FUNCTION FROM C++.....	1-26
1.12 LESSON 8: INPUT PARAMETERS AND RETURN VALUES OF A LONG TYPE	1-29
1.13 LESSON 9: LOGICAL ACTIVATION AND MASKING ON VCP	1-33
1.14 LESSON 10: ARITHMETICAL ACTIVATION ON VCP.....	1-38
1.15 LESSON 11: USE OF CYCLIC SHIFTER ONE BIT RIGHT	1-44
1.16 LESSON 12: USE OF VR VECTOR REGISTER.....	1-47
1.17 LESSON 13: MACROS.....	1-50
1.18 LESSON 13A: HOW TO CREATE A MACRO LIBRARY	1-53

Figures

FIGURE 1-1. GENERAL VIEW OF THE EMUDBG C++ SOURCE DEBUGGER.....	1-4
FIGURE 1-2. CHANGING THE ORDER OF VECTOR ELEMENTS ON THE MULTIPLY UNIT	1-20
FIGURE 1-3. COMPUTATION OF THE SUM AND DIFFERENCE OF VECTOR ELEMENTS.....	1-22
FIGURE 1-4. COMPUTATION OF THE SUM OF SIXTY-FOUR 32-BIT ARRAY ELEMENTS	1-25
FIGURE 1-5. THE STACK CONTENTS AT THE MOMENT OF A FUNCTION CALL	1-28
FIGURE 1-6. SIGN BITS EXPANSION BY LOGICAL ACTIVATION.....	1-36
FIGURE 1-7. LOGICAL MASKING OPERATION.....	1-37
FIGURE 1-8. CHANGING BIT-LENGTH FOR INTERMEDIATE REPRESENTATION.....	1-42
FIGURE 1-9. SATURATION BY ARITHMETICAL ACTIVATION	1-44

The preface describes the contents of the manual, gives a brief annotation of chapters, defines style and symbolic notations used in the document.

About This Manual

This manual helps you learn how to program NeuroMatrix® NM6403 fixed-point DSP.

Before you use this book, you should install NeuroMatrix® NM6403 SDK on your computer.

This manual covers the following topics:

- Step by step tutorial describing how to program NM6403 in assembly and C++ language;
- Using SDK tools to compile, debug and optimize NM6403 programs;
- Basic methods for code optimization;

Organization

This manual is organized into the following chapters:

Chapter 1 How to Write Programs in NM6403 Assembly

This chapter is organized in form of lessons helping you to gain some basic skills in programming the NM6403. It describes how to write programs in the NM6403 Assembly Language and covers the following aspects:

- ◆ Scalar instructions, loops organization, joint execution of an addressing command and an arithmetical operation in one scalar instruction, branch and delayed branch.
- ◆ Vector instructions, using of different calculation nodes of the Vector Co-Processor (VCP).

Chapter 2 Approaches to Programs Optimization

This chapter describes the main approaches used for assembly source code optimization, memory allocation for code and data, distribution of the VCP resources among the instructions executed in parallel.

Chapter 3 Appendix A

This chapter contains additional information regarding the number of instructions executing before the delayed branch occurs, macros, using of the same general purpose register in both parts of a scalar instruction.

Typographical Conventions

The following typographical conventions are used in this manual:

<code>Courier</code>	Denotes text that may be entered at the keyboard: commands, file and program names, and assembler and C++ source. This is most often used in syntax descriptions.
<i>Courier</i>	Shows text that must be substituted with user-supplied information.
Times or <u>Times</u>	Highlights important notes.
<code>//Times</code>	Shows comments to assembly and C++ source.

Note

Boxes like this contain information on significant notes and comments to the context.

Feedback

Feedback on This Manual

If you have feedback on this manual, please contact your supplier, giving:

- the manual's title;
- the manual's document number;
- the page number(s) to which your comments refer;
- a concise explanation of the comment.

General suggestions for additions and improvements are also welcome.

Feedback on NM6403 Software Development Kit

If you have comments or suggestions about the NeuroMatrix® NM6403 Software Development Kit, please contact your supplier or send e-mail to nm-support@module.ru, giving:

- the platform and release of the NM6403 software tools you are using;

- a small sample code fragment which illustrates your comment;
- precise description of your comment or suggestion.

This chapter is organized in form of lessons helping you to gain some basic skills in programming the NM6403.

Each lesson contains an example of a program with comprehensive comments and the information on how to compile it.

Before you start this tutorial, you should install the NeuroMatrix NM6403 SDK, containing code generation tools and the C++ source debugger. If you do not have this SDK you are able to download it from our WEB site: <http://www.module.ru>

1.1 Lesson 1: A Simple Program in NM6403 Assembly

A source code of the example used in this lesson is contained in the `step1.asm` file, located in the directory: `..\Tutorial\Step1`.

Example Description

In the example we load constants into general-purpose registers and then add the contents of the registers returning the sum.

Source Code

```
global __main: label;    // declaration of a global label

begin ".textAAA"        // the beginning of a code section
<__main>                 // definition of the global label
    gr0 = 1;             // loading a constant to a general-purpose register
    gr1 = 2;
    gr7 = gr0 + gr1;     // calculating the sum of two registers
    return;              // return from the routine, the return value is contained in gr7
end ".textAAA";         // the end of the code section
```

Comments to Example

The example starts from a global label declaration. This declaration means that the `__main` label will be defined below in the file. That will be the start address of the user's program.

A label can be declared anywhere in the file, but it is recommended to put label declaration out of sections of code and data.

The `__main` is a special label (two underlines must be put before the "main" word). It defines the beginning of the user program. This name is

called from the startup code, which is automatically added to the program at the linking stage.

A code section follows the declaration of the global label. The code section begins from the opening bracket `begin` and finishes with the closing bracket `end`. Each section has its own name, not necessary a unique one, following the section brackets. The section name at the opening and at the closing brackets must be the same, for example:

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

Note

We recommend starting a code section name from the `text` prefix, for example: `<textMyCodeSection>`. The disassembler (`dump.exe`) uses this prefix to decode the contents of the section. In the opposite case it will leave the section as a binary code.

A label definition follows the `begin ".textAAA"` bracket:

```
<__main>
```

The label definition is always surrounded by the “< >” brackets. The label marks the next nearest instruction before the “;” symbol is met. In the example above the `__main` label marks the `gr0 = 1;` instruction..

The instructions:

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

are the general-purpose register initialization with a constant commands.

The instruction:

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

adds the contents of `gr1` to the contents of `gr0` and stores the result into `gr7`.

If a routine returns a value it puts the return value into the `gr7` register.

Any routine must be finished with a `return` instruction.

The last line in the example is the closing bracket of the code section.

Compiling the Example Code

On a command line, enter the following on a single line:

```
nmcc -g step1.asm libc.lib -m
```

You should not receive any errors, and the file, `step1.abs`, should be created. If you receive an error message, look up that error message in the **NeuroMatrix® NM6403 Programmer's Reference**.

The `nmcc` is a shell, which is used to simplify the multi-file projects compiling. It detects automatically the set of tools to be invoked to compile each file of the project and to get an executable file.

The `nmcc` options can be encountered in an arbitrary order.

The `-g` option turns on generation of debug information.

The `-m` option turns on generation of a map file, containing information about physical memory addresses and size, about location of code and data sections. It also contains a list of global labels and their absolute addresses.

One more item should be added to a command line when compiling only assembler source files. It is a `libc.lib` file, which is a C runtime library containing a startup code and the `start` entry point. The startup code allows executing and debugging user programs with SDK utilities. For instance, the library contains the code indicating that the program has finished. This code modifies predefined flags in shared memory. The flags are used by SDK utilities to detect whether the program is still executing or it has already finished and what the main routine return value is.

Note

The `libc.lib` library is added automatically in the case the `nmcc` shell encounters a `.cpp` file in the command line. If the application is built only of assembly source files, the `libc.lib` library must be added manually.*

If an output file name is not specified in the command line, the shell gets the name of the first file met and uses this name to form the name of the output file adding the extension “`.abs`”. In the example above the first file the shell encounters in the command line is `step1.asm`, therefore the output file name will be `step1.abs`.

Running the Program on Simulator

The `step1.abs` output executable file may be loaded and run on a simulator (the `emurun.exe` utility). This utility is based on an instruction level simulator. It is intended for running the NM6403 absolute executable files but not for debugging them. It runs the programs and prints the user application return value like shown below:

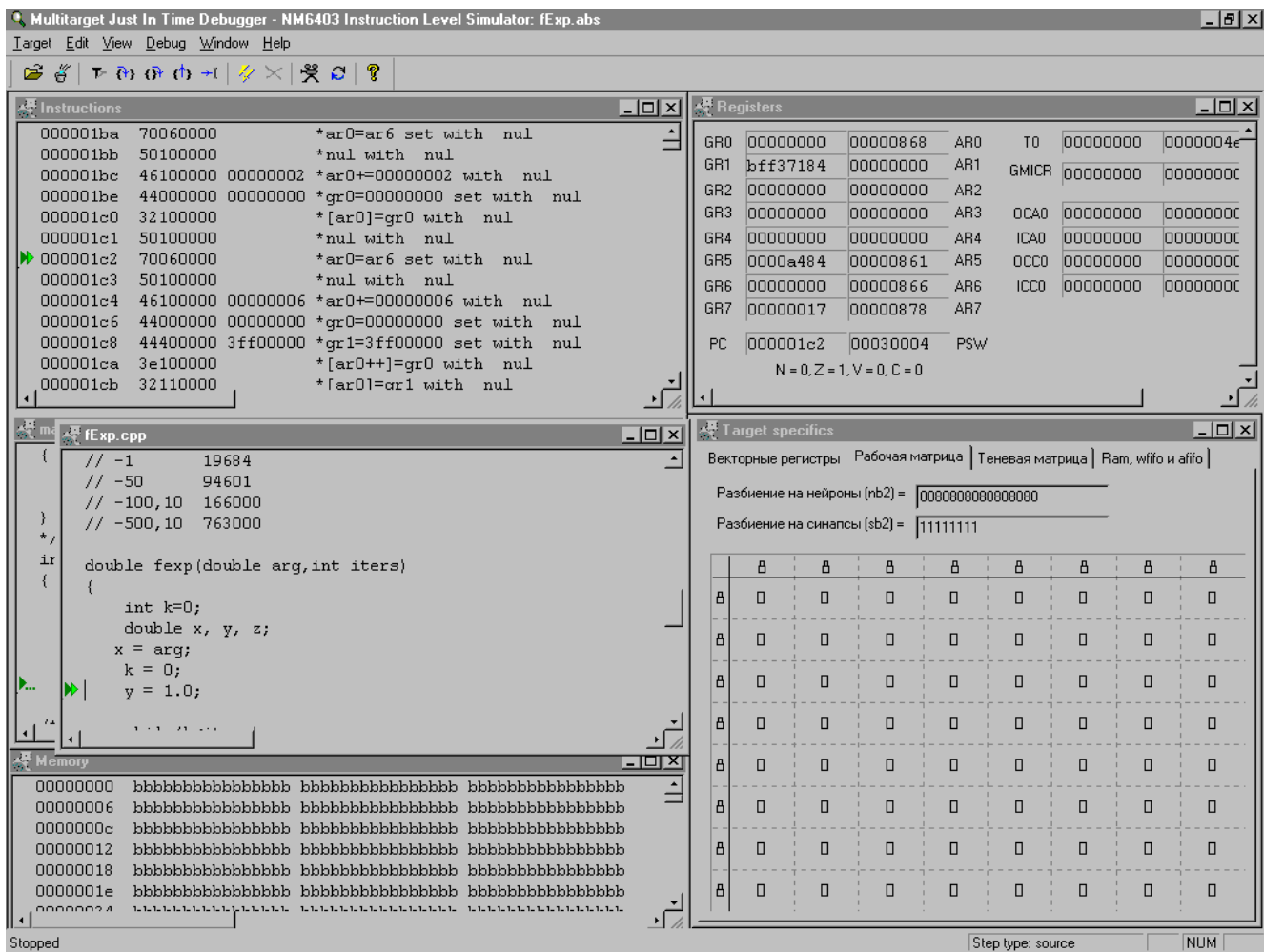
```
Step1.abs:: WARNING: return 3 = 0x3
```


The `main()` routine return value appears in the case it is a nonzero.

Debugging the Program on C++ Source Debugger

To load the NM6403 absolute executable file into the C++ Source Debugger double click on the file icon. The program will be automatically loaded into `emudbg` and will be stopped at the entry point ready for execution the first instruction of startup code.

Figure 1-1. General View of the EMUDBG C++ Source Debugger



To start debugging source code of your application select the VIEW | SOURCES menu item and then the file name you wish to debug. Put the cursor to the line where you want to start debugging from and press the  button (step to cursor) on a toolbar.

Further debugging is going on in a usual way. All debug information like registers state, memory contents and so on are available at the appropriate service windows. Press F1 to get on-line context sensitive help.

1.2 Lesson 2: Memory Access

The source code of the example used in this lesson can be found in the `step2.asm` file in the directory: `..\Tutorial\Step2`.

Example Description

The example provides a description of different types of data sections and methods of memory access.

Source Code

```
global __main: label;    // global label declaration.

data ".MyData"          // section of initialized data.
    A: word = 1;
    B: word = 2;
end ".MyData";

nobits ".MyData1"       // section of non-initialized data.
    global C: word[2];
end ".MyData1";

begin ".textAAA"        // beginning of the code section.
<__main>
    ar0 = A;             // address of an A variable is loaded to ar0.
    gr0 = [ar0];         // value of the memory cell at the A address is loaded to gr0.
    gr1 = [B];           // value of the memory cell at the B address is loaded to gr1.
    gr2 = gr0 + gr1;     // gr2 = A + B.
    ar0 = C;             // address of a C variable is loaded to ar0.
    [ar0++] = gr2;       // the contents of gr2 is stored into memory at the address C[0],
                        // then the address is incremented (post-incrementation).
    gr2 = gr0 - gr1;     // gr2 = A - B.
    [ar0++] = gr2;       // C[1] = A - B, and then ar2 points to C[2].
    gr7 = [--ar0];       // gr7 = C[1]. At first the address is decremented, and then the
                        // processor reads the contents of the C[1] memory cell.

    return;
end ".textAAA";         // end of the code section.
```

Comments to the Example

The example stores the A+B value into C[0], the A-B value into C[1] and loads the contents of C[1] to the gr7 register.

This example introduces sections of initialized and non-initialized data.

The sections of initialized data contain definition and initialization of variables used by the program. A section of this type begins with the opening bracket `data` and ends with the word `end` (a closing bracket), for example:

```
data "MyData"
    A: word = 1;
```

```
B: word = 2;
end "MyData";
```

Definition of a variable has the form “name_of_the_variable: type”, for instance “A: word”. The initial value or a list of initial values follows the definition and the sign “=” stands before it.

```
A: word = 1; // (for more details see Section 2.3.2 of
              // the NM6403 Assembly Language Overview)
```

Sections of non-initialized data contain only definition of variables without their initialization. A section of this type begins with the opening bracket `nobits` and ends with the word `end`. Example:

```
nobits ".MyData1"
    global C: word[2]; // array of two 32-bit words
end ".MyData1";
```

The code section in the example above contains different types of reading from memory commands:

- To obtain the address of a variable we use just its name:
`ar0 = A;`
(the command loads the address of the `A` variable into the `ar0` address register)
- To obtain the value of a variable we take its name into square brackets:
`gr1 = [B]; // direct reading from memory`
(we load the value of the variable `B` to the `gr1` general-purpose register).
- Indirect reading from memory to the `gr0` general-purpose register
`gr0 = [ar0]; // indirect reading from memory`

The instruction `[ar0++] = gr2;` executes indirect storing to memory from a general-purpose register with post-incrementation of the address. It means that at first the contents of `gr2` is stored into memory cell at the address contained in `ar0` and then the address is increased by 1.

The instruction `gr7 = [--ar0];` executes indirect reading from memory to a general-purpose register with pre-decrementation of the address, i.e. before reading the value from memory the address is decreased by 1.

How to Compile the Example

To compile the example `step2.asm` use the following string on a command line:

```
nmcc -g step2.asm libc.lib -m
```

1.3 Lesson 3: Making Loops

The source code of an example used in this lesson can be found in the `step3.asm` file in the directory: `..\Tutorial\Step3`.

Example Description

The example shows the simplest way of making loops. The program fills an array with ascending values.

Source Code

```
global __main: label;

nobits ".MyData1"
    global C:word[16]; // memory allocation for an array of sixteen 32-bit words
end ".MyData1";

begin ".textAAA"
<__main>
    ar0 = C; // the address of the C array is loaded to ar0.
    gr0 = 0;
    gr1 = 16; // gr1 is used as a loop counter.
<Loop>
    [ar0++] = gr0; // the contents of gr0 is stored into memory at the address ar0 and
                  // then the address is increased by 1 (post-incrementation).
    gr0 ++; // the contents of gr0 is increased by 1
    gr1--; // the contents of gr1 is decreased by 1 setting up the condition
           // flags for further checking
    if > goto Loop; // if the condition is true go to the Loop label.

    return;
end ".textAAA";
```

Comments to the Example

The loop is made using a *goto* branch command to an assigned label if certain conditions are fulfilled (the appropriate condition flags are set up).

The command

```
if > goto Loop;
```

executes jump the `Loop` label, in case the condition `>` (greater) is fulfilled (this condition checks the flags N-negative and Z-zero). The branch conditional command checks the flags set by the previous operation. In the case of the example above the operation previous to the branch is `gr1--`. The `gr1` register is used here as a loop counter, because the NM6403 processor has no a special loop counter register.

The condition flags are set only as a result of execution of an arithmetical or logical operation in the right part of a scalar instruction.

How to Write Programs in NM6403 Assembly

For more details about branch conditions see Section 5.1.9.4 Set of Branch Conditions in the document **NeuroMatrix NM6403 Assembly Language Overview**.

How to Compile the Example

To compile the example `step3.asm` you should enter the following string on a command line:

```
nmcc -g -m step3.asm libc.lib
```

1.4 Lesson 3a: Loop Optimization

The source code of the example used in this lesson can be found in the `step3a.asm` file in the directory: `..\Tutorial\Step3a`.

Example Description

The example shows the way of optimization of the loop described in the previous lesson.

Source Code

```
global __main: label;

nobits ".MyData1"
    global C:word[16]; // memory allocation for an array of sixteen 32-bit words
end ".MyData1";

begin ".textAAA"
<__main>
    ar0 = C; // the address of the C array is loaded to ar0.
    gr0 = 0;
    gr1 = 16; // gr1 is used as a loop counter.
    gr1--; // the loop variable is decreased by 1 to enter the loop
           // with correctly set condition flags.

<Loop>
    // if the condition is true the delayed jump to the Loop label is made
    if > delayed goto Loop with gr1--;
    // the two next instructions are executed before the real jump occurs
    [ar0++] = gr0 with gr0++ noflags;
    nul;
    // ----- here the branch to the Loop label will be made -----
    return; // the program will go here when the condition is false
end ".textAAA";
```

Delayed Branches

The NM6403 offers two main types of branching: standard and delayed. **Standard branches** empty the pipeline before performing the branch; this guarantees correct management of the program counter and results in a NM6403 branch taking from two to four cycles. Included in this class are `call`, `goto`, `skip`, `return` and `ireturn`. The compiler automatically adds the correct number of `nul` instructions to make the pipeline empty before the actual branch occurs. For example,

```
if > goto Loop;
```

This branch instruction is a standard branch, therefore the compiler will automatically add two `nul` instructions.

Delayed branches do not empty the pipeline but, rather, guarantee that the next one to three instructions will be executed before the program counter is modified by branch. The result is a branch that requires only a single cycle. Delayed branches do not annul the next instructions. A formal approach allowing the programmer to detect the number of instructions to be executed is given in appendix XXXX.

Here is an example of using the delayed branch:

```
if > delayed goto Loop with gr1--;
    // the two next instructions are executed before the real jump occurs
    [ar0++] = gr0 with gr0++ noflags;
    nul;

    // here the program counter is modified by branch
```

When a delayed branch is fetched, it remains pending until the one to three instructions that follow are executed.

Comments to the Example

In the example above the delayed branch instruction

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

is followed by two more instructions that are executed before the actual branch occurs:

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

the delayed instructions are executed in any case regardless of whether the branch condition is true or not.

Let's consider the loop in details. As it was already mentioned, it consists of a conditional delayed branch instruction and two instructions following it. By the first entry into the loop the instruction:

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

checks the flags set by the previous arithmetical operation, namely: `gr1--`. Here the subtraction made in the right part of the instruction sets the condition flags for checking in the next loop.

In order to prevent the condition flags modification in the loop inner instructions, the `'noflags'` keyword is used after the arithmetical operation. That saves the flags set by the previous arithmetical operation.

Compilation of Example

To compile the example `step3a.asm` enter the following on a command line:

```
nmcc -g step3a.asm libc.lib -m
```

1.5 Lesson 4: Copying a Data Array on RISC-core

The source code of the example used in this lesson can be found in the `step4.asm` file in the directory: `..\Tutorial\Step4`

Example Description

The example shows two methods of copying a 64-bit word array on the scalar processor. The first method is a simple copying, the second one is a copying with the help of register pairs.

Source Code

```
global __main: label;

data ".MyData"
    // array A of sixteen 64-bit words is filled with initial values
    global A: long[16] = ( 01, 11, 21, 31, 41, 5h1, 61, 71, 81, 91,
                          101, 0Bh1, 0Ch1, 131, 141, 151);
end ".MyData";

nobits ".MyData1"
    global B: long[16]; // definition of a destination array B of sixteen 64-bit words
    global C: long[16]; // definition of a destination array C of sixteen 64-bit words
end ".MyData1";

begin ".textAAA"
<__main>
    // simple copying of a data array on the RISC-core
    ar0 = A;
    ar1 = B;
    gr1 = 32; // loop counter (32 loops to copy sixteen 64-bit words)
```

```
    gr1--;                // condition flags setting for the first entry to the loop
<Loop>
    // if the condition is true, the delayed jump to the Loop label is made
    if > delayed goto Loop with gr1--;
        // reading a 32-bit word from memory
        gr2 = [ar0++];
        // writing the 32-bit word to memory
        [ar1++] = gr2;

    // using a register pair to copy the data array
    ar0 = A;
    ar1 = C;
    gr1 = 16;             // loop counter (16 loops to copy sixteen 64-bit words)
    gr1--;                // condition flags setting for the first entry to the loop
<Loop1>
    // if the condition is true, the delayed jump to the Loop1 label is made
    if > delayed goto Loop1 with gr1--;
        // reading a 64-bit word from memory
        gr2,ar2 = [ar0++];
        // writing the 64-bit word to memory
        [ar1++] = ar2,gr2;

    return;
end ".textAAA";
```

Comments to the Example

In the first part of the example data copying is made using one 32-bit register. First the processor reads a word from memory to a register from a primary register file, and then it is copied the contents of the register to memory at another address. Both address registers pointing to the source and destination buffers are incremented. Since it is necessary to copy an array of sixteen 64-bit words and one 32-bit (the lower or the higher part of a 64-bit word) is transferred through the register for one copying loop, then in order to copy the whole array it is necessary to execute **thirty two** loops.

In the second part of the example copying is made over the register pair ar2, gr2 (in a register pair each address register is associated with a general-purpose register with the same number). For one loop of

reading/writing a whole 64-bit word is transferred, that is why the number of copying loops is **sixteen**.

When reading from memory to the register pair `ar2,gr2 = [ar0++]`; the low part of the 64-bit word ALWAYS goes to `arx` and the high part – to `grx` regardless of the order in which the registers are encountered in the pair. The same rules are true when writing the register pair contents to memory. The content of the `arx` register is always stored to the lower (even) address while the contents of the `grx` register - to the higher (odd) address. Thus the instruction: `[ar1++] = gr2,ar2`; stores data to memory in the same order as they have been read regardless of the order in which the registers are encountered in the register pair.

The another important point worth mentioning is how address registers used for memory access are changed. Both in the first and in the second part of the example the same syntax for the `ar0` and `ar1` registers incrementation is used. However in the first part when a 32-bit memory access is executed, address registers values are increased by one and in the second part – by two.

The processor recognizes automatically what type of memory access is used in the given instruction – 32-bit or 64-bit. The presence of a register pair or a 64-bit control register in the instruction causes the 64-bit memory access. But since the addressing is made over 32-bit words, the 64-bit incremental access increases the contents of an address register by two, for example:

```
gr2 = [ar0++];           // ar0 increases by 1
ar2,gr2 = [ar0++];      // ar0 increases by 2
```

Compilation of Example

To compile the example `step4.asm` enter the following on a command line:

```
nmcc -g step4.asm libc.lib -m
```

1.6 Lesson 4a: Copying a Data Array on VCP

The source code of the example used in this lesson can be found in the `step4a.asm` file in the directory: `..\Tutorial\Step4a`

Example Description

The example compares use of the RISC-core and the VCP for copying an array of 32-bit words.

Source Code

```
global __main: label;

data ".MyData"

// an array of sixteen 32-bit words is filled with initial values
```

```
global A: word[16] = ( 0, 1, 2, 3, 4, 5h, 6, 7, 8, 9, 10, 0Bh,
                      0Ch, 13, 14, 15);

end ".MyData";

nobits ".MyData1"

global B:word[16]; // definition of a destination array B of sixteen 32-bit words
global C:word[16]; // definition of a destination array C of sixteen 32-bit words
end ".MyData1";

begin ".textAAA"
<__main>
    // use of the RISC-core for copying an array of 32-bit words
    ar0 = A;
    ar1 = B;
    gr1 = 16; // loop counter (16 loops for copying sixteen 32-bit words)
    gr1--; // condition flags setting for the first entry to the loop

<Loop>
    // if the condition is true, the delayed jump to the Loop label is made
    if > delayed goto Loop with gr1--;
        // reading a 32-bit word from memory
        gr2 = [ar0++];
        // writing the 32-bit word to memory
        [ar1++] = gr2;

    // use of the vector co-processor for copying an array of 32-bit words
    ar0 = A;
    ar1 = C;
    // the data from memory are sent to the Vector ALU and they go to afifo buffer unchanged
    rep 8 data = [ar0++] with data;
    // contents of afifo filled by the previous vector instruction is stored into the external memory
    rep 8 [ar1++] = afifo;

    return;
end ".textAAA";
```

Comments to the Example

Use of the RISC-core was described in detail in the previous lesson. Therefore in this lesson we introduce the vector co-processor.

Use of VCP for copying an array:

As well as the scalar instruction, the vector instruction consists of a left and a right part. The left part contains a command for memory access and addressing, while the right part defines calculations on the vector co-processor.

Left part of the instruction:

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

the processor performs indirect reading from external memory. The address of an input buffer is stored in the address register. The data are loaded to the `data` logical register-container with post-incrementation of the address register. In the right part of the instruction the data passing through the data bus come to the input **X** of the Vector ALU and in this particular case remain unchanged. The right part of the instruction is a brief description of the expression:

```
with data or 0;
```

The results of the most vector instructions are stored into the `afifo` register-container.

An obligatory attribute of a vector instruction is the number of iterations defining the number of 64-bit data vectors being processed by this instruction. The vector instructions are SIMD (Single Instruction Multiple Data) instructions performing the same operation over several data vectors.

For simple operations on vectors like copying or logical operations it is not necessary to configure the VCP. But if you perform arithmetical operations or calculations on the multiply unit (MU), it is required to configure the VCP first and only after that it is possible to make calculations.

The `nb1` and `sb` registers are used to configure the VCP. If calculations are performed on the Vector ALU it is enough to initialize only the `nb1` register:

```
nb1 = 0;
```

```
wtw; // this instruction copies the nb1 shadow matrix control register  
    // to the nb2 active matrix control register
```

but since the logical operation we use in the example of this lesson does not suppose carrying bits, these instructions can be omitted.

The command

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

stores the results from `afifo` into memory with post-incrementation of the address register. (`rep` the number of words to be stored). Data can not be stored in portions (for example, first four words, then four more words), but only the whole `afifo` contents must be processed. The contents of `afifo` cannot be stored into the processor registers or register pairs, but only into memory.

There is one more thing that should be considered carefully. It is a difference in the number of iterations in the RISC-core part and the VCP part of the example: the number of iterations in the scalar loop is sixteen while the number of iterations in a vector instruction is eight. This difference appears from the fact that the VCP always operates on 64-bit vectors. Therefore the `ar0` and `ar1` address registers used in vector instructions are incremented by two.

Compilation of Example

To compile the example `step4a.asm` enter the following on a command line:

```
nmcc -g step4a.asm libc.lib -m
```

1.7 Lesson 5: Arithmetical Operations on Vector ALU

The source code of the example used in this lesson can be found in the `step5.asm` file in the directory: `..\Tutorial\Step5`

Example Description

The example shows how to perform arithmetical operations over a vector array. The operations are executed on the NM6403 Vector ALU. An array of 256 32-bit elements (128 64-bit vectors) is filled with ascending values.

Source Code

```
global __main: label;

data ".MyData"
    // basic value for filling the array

    AA: long = 100000000h1;
    // increment for the first loop
    BB: long = 200000002h1;
    // increment for the second loop
    CC: long = 4000000040h1;
end ".MyData";
```

How to Write Programs in NM6403 Assembly

```
nobits ".MyData1"
    .align;                // pseudo-instruction for alignment of the array start address
                        // to an even address.

    // array of 256 32-bit elements that will be filled with ascending values from 0 to 255
    global A: word[256];
end ".MyData1";

begin ".textAAA"
<__main>
    ar0      = AA;        // the AA address (AA = 100000000hl) is loaded to ar0
    ar4      = BB;        // the BB address (BB = 200000001hl) is loaded to ar4
    ar1,gr1  = A;         // the address of the A buffer is loaded to ar1 and gr1
                        // at the same time

    gr2      = 31;        // loop counter

    nb1 = 80000000h;      // splitting a 64-bit input vector into two 32-bit elements
    wtW;                 // copying the contents of the shadow register nb1 to the
                        // working register nb2

    // the increment that will be added in the loop to the current afifo value to obtain new values of
    // filler is loaded to ram.
    rep 1 ram = [ar4];
    // the first value of filler is put to the vector processor.
    rep 1 data = [ar0] with data;

    gr2--;              // set up the condition flags to entry into the loop
<Loop>
    if > delayed goto Loop with gr2--;
        // the first 64 elements of the output array are filled
        rep 1 [ar1++] = afifo with afifo + ram;
        nul;

    gr2 = 2;            // counter for the second loop
    rep 1 [ar1++] = afifo; // storing the last value from afifo into external memory
    ar1 = gr1 with gr2--; // return to the array beginning and setting up the flags

    ar0 = CC;           // the CC address (CC = 4000000040hl) is loaded to ar0
```

```
// the increment that will be added in the loop to the current afifo value to obtain new values of  
// filler is loaded to ram.  
rep 32 ram = [ar0];  
// the first values of filler are put to the vector processor.  
rep 32 data = [ar1++] with data;  
  
<Loop1>  
if > delayed goto Loop1 with gr2--;  
// the A array is filled with initial values. 64 elements are processed for one loop.  
rep 32 [ar1++] = afifo with afifo + ram;  
nul;  
  
// the last 64 elements are stored into memory  
rep 32 [ar1++] = afifo;  
  
return;  
end ".textAAA";
```

Comments to the Example

The program consists of two parts. In the first part we fill the first sixty-four 32-bit elements of the A array with a set of ascending values (from 0 to 63). In the second part the rest of the array is filled according to the values obtained in the first part.

The program is executed on the VCP and employs the operation of summation on the Vector ALU.

The instruction:

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

puts a 64-bit vector 0000000200000002h1 to ram. This vector will be used for incrementation of array filling values.

The instruction:

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

puts a 64-bit vector 0000000100000000h1 to afifo. Note, the results of the most vector instructions are stored to afifo.

After the delayed branch instruction is fetched and before the actual branch occurs two next processor instructions are executed

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

The first instruction stores the contents of afifo to external memory starting from the address of the output array. The same data is used for

calculation of the next value. At the right part of the vector instruction the old contents of `afifo` is added to the contents of `ram`. The sum is returned to `afifo` for further processing.

The loop described above fills the first 64 32-bit elements of the A array with ascending values from 0 to 64.

The instruction:

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

stores the last 10 elements (the 62th and the 63th).

After the first 64 elements of the array have been filled, they are used for filling the rest of the array. For this the number `0000004000000040h` duplicated thirty two times is put to `ram`. It serves as an increment for modification of fillers in the second loop. The approach to calculations is the same for both loops. The only difference is that in the second case a processor instruction performs filling of 32 vectors of data.

Compilation of Example

To compile the example `step5.asm` enter the following on a command line:

```
nmcc -g step5.asm libc.lib -m
```

1.8 Lesson 6: Weighted Accumulation

The source code of the example used in this lesson can be found in the `step6.asm` file in the directory: `..\Tutorial\Step6`

Example Description

This lesson explains how to use a Multiply Unit (MU), a Shadow and an Active Matrix, which are the parts of the NeuroMatrix VCP. The example considers use of the weighted accumulation operation for bytes permutation in the bounds of a 64-bit data vector.

Source Code

```
global __main: label;

data ".MyData"
    // source vector
    A: long = 8877665544332211h;
    // destination vector
    B: long = 01;
    // the Matr array contains weight coefficients for the Active Matrix
```

```
Matr: long[8] = (0100000000000000h1,
                0001000000000000h1,
                0000010000000000h1,
                0000000100000000h1,
                0000000001000000h1,
                0000000000010000h1,
                00000000000000100h1,
                00000000000000001h1);

end ".MyData";

begin ".textAAA"
<__main>
    ar1 = Matr;
    nb1 = 80808080h;    // the MU and the matrixes are divided into eight 8-bit columns
    sb  = 03030303h;    // the MU and the matrixes are divided into eight rows
    // weight coefficients are loaded into wfifo buffer
    rep 8 wfifo = [ar1++];
    ftw;                // weight coefficients are sent to the Shadow matrix with
                        // decoding. This instruction always requires thirty-two cycles.
    wtw;                // weight coefficients are copied from the Shadow Matrix to the
                        // Active matrix

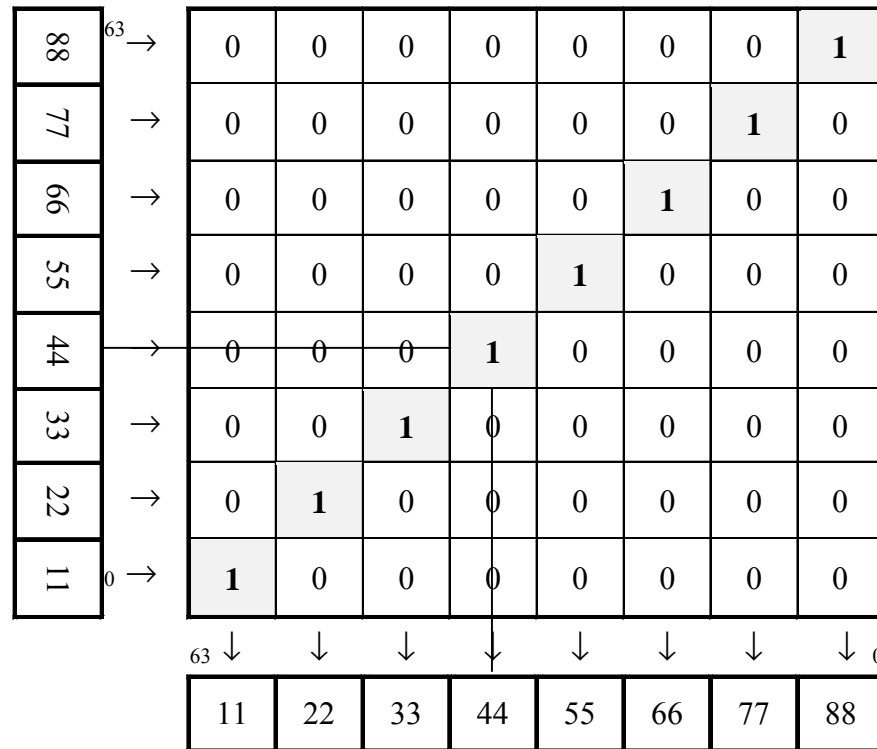
    ar2 = A;
    ar4 = B;

    // the weighted accumulation operation rearranges bytes order in the vector
    rep 1 data = [ar2] with vsum , data, 0;
    // result of the operation is stored from afifo to memory
    rep 1 [ar4] = afifo;
    return;
end ".textAAA";
```

Comments to the Example

The purpose of this example is rearrangement of the order of elements in a 64-bit vector. The input state is $A = 8877665544332211h1$, the output state is $B = 1122334455667788h1$. This rearrangement is made on the multiply unit of the VCP with the help of the weighted accumulation operation. The main idea of this transform is explained in Figure 1-2:

Figure 1-2. Changing the Order of Vector Elements on the Multiply Unit



Before executing the weighted accumulation operation it is necessary to fill the Shadow and then the Active Matrix with weight coefficients. But first of all it is necessary to divide the matrixes into rows and columns.

The `nb1 = 80808080h` instruction divides the Shadow matrix into eight columns. Here identical constants are copied to both parts of the 64-bit register. Thus `nb1` contains the `8080808080808080h1` constant. This register is responsible for the Shadow matrix division into columns.

The `sb = 03030303h` instruction divides the Shadow matrix into eight rows. Here identical constants are copied to both parts of the register. Thus `sb` (64-bit) contains the `0303030303030303h1` constant.

The instruction:

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

loads weights from memory to the `wfifo` register-container. The loading can be made in portions but so that overfilling would not occur. The `wfifo` container has the depth of thirty-two 64-bit words.

The `ftw` instruction makes decoding of weight coefficients stored in `wfifo` into a special form for fast calculations. The results of decoding are stored into the Shadow matrix. This operation always takes thirty-two processor cycles, however it can be executed in parallel with other vector operations.

The `wtw` instruction copies weight coefficients from the Shadow matrix to the Active matrix, which takes just one cycle.

The instruction:

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

performs weighted accumulation with the coefficients that were earlier loaded to the Active matrix. The computation is made according to the scheme shown in Figure 1-2. The result of the operation is stored into the `afifo` register-container.

The instruction:

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

uploads the result from `afifo` to external memory.

Compilation of Example

To compile the example `step6.asm` enter the following on a command line:

```
nmcc -g step6.asm libc.lib -m
```

1.9 Lesson 6a: Weighted Accumulation (Continuation 1)

The source code of the example used in this lesson can be found in the `step6a.asm` file in the directory: `..\Tutorial\Step6a`

Example Description

The example considers use of the weighted accumulation operation for simultaneous computation of the sum and difference of two 32-bit elements of a 64-bit vector.

Source Code

```
global __main: label;

data ".MyData"
    // input vector
    A: long = 3333333322222222h1;
    // output vector
    B: long = 01;
    // the Matr array contains weight coefficients for the Active Matrix
    Matr: long[2] = ( 0000000100000001h1,
                    0FFFFFFFF00000001h1 );
end ".MyData";

begin ".textAAA"
<__main>
    ar1 = Matr;
    nb1 = 80000000h;    // matrix division into two columns with 32 bits in each column
```

How to Write Programs in NM6403 Assembly

```

sb = 03h;           // matrix division into two rows with 32 bits in each row
// weight coefficients are loaded into wfifo buffer, at the same time they are translated to the
// Shadow matrix and then to the Active matrix.
rep 2 wfifo = [ar1++], ftw, wtw;

ar2 = A;
ar4 = B;

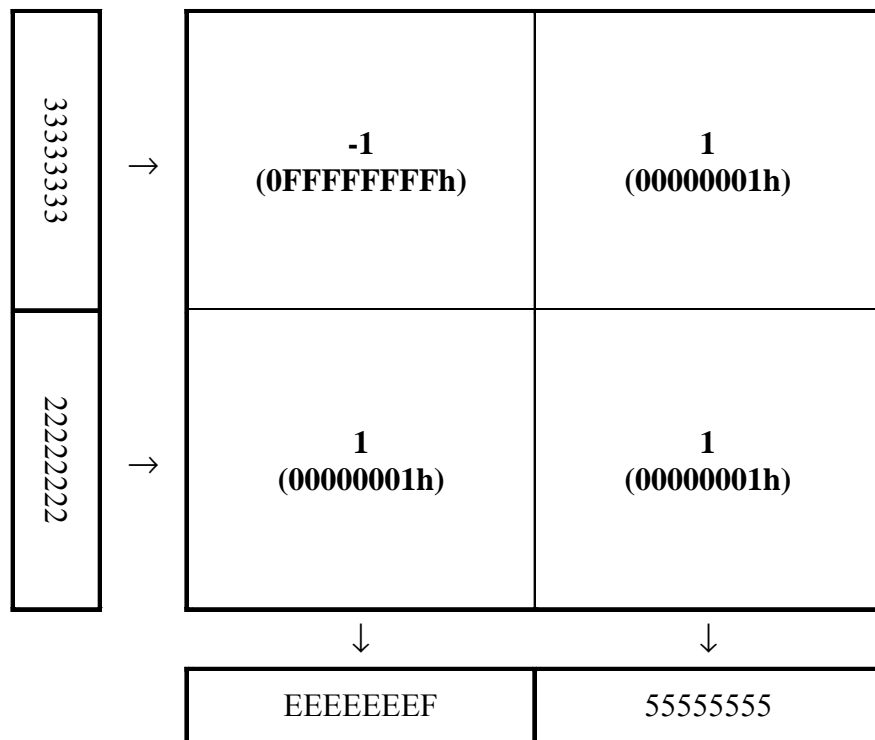
// weighted accumulation allows calculating sum and difference of vector elements
rep 1 data = [ar2] with vsum , data, 0;
// result of the operation is stored from afifo to memory
rep 1 [ar4] = afifo;
return;
end ".textAAA";

```

Comments to the Example

The purpose of this example is simultaneous computation of the sum and difference of elements of a 64-bit vector. This computation is made on the MU with the help of the weighted accumulation operation. The main idea of this transform is explained in Figure 1-3:

Figure 1-3. Computation of the Sum and Difference of Vector Elements



The `nb1 = 80000000h` instruction divides the Shadow matrix into eight columns. Here identical constants are copied to both parts of the register. Thus `nb1` contains the `8000000080000000h1` constant.

The `sb = 03h` instruction divides the matrix into two rows. Here identical constants are copied to both parts of the register. Thus `sb` contains the `0000000300000003h1` constant.

The instruction:

```
rep 2 wfifo = [ar1++], ftw, wtw;
```

loads weight coefficients from memory to the `wfifo` register-container. As soon as the first word of the weights appears in `wfifo`, it is immediately decoded into the Shadow matrix format (this operation is assigned by the `ftw` command). The processor automatically defines the number of weights to be loaded from `wfifo` for further decoding. After the Shadow matrix is filled with decoded weights its contents copies to the Active matrix.

The instruction:

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

executes weighted accumulation with the coefficients that were earlier loaded to the Active matrix. The computation is made according to the scheme shown in Figure 1-3. The result of the operation is copied to the `afifo` register-container.

The instruction:

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

uploads the result from `afifo` to external memory.

Compilation of Example

To compile the example `step6a.asm` enter the following instruction in the command line:

```
nmcc -g step6a.asm libc.lib -m
```

1.10 Lesson 6b: Weighted Accumulation (Continuation 2)

The source code of the example used in this lesson can be found in the `step6b.asm` file in the directory: `..\Tutorial\Step6b`

Example Description

The example considers use of the weighted accumulation operation for simultaneous computation of the sum of sixty-four 32-bit words. It is supposed that overflow does not occur.

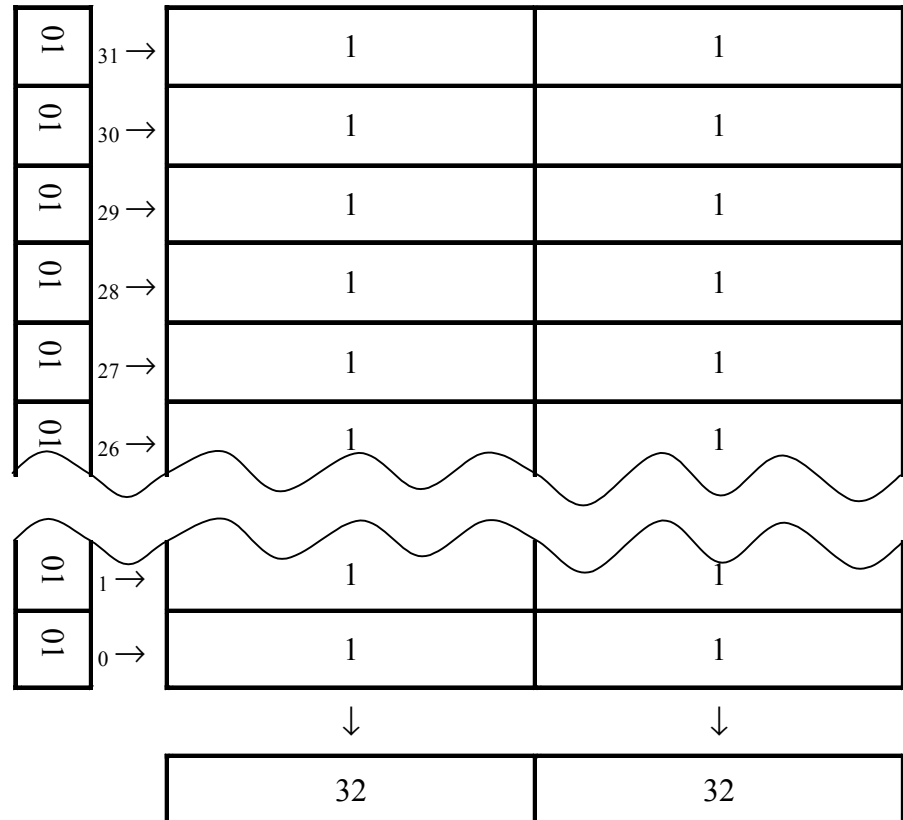
Source Code

```
global __main: label;
```


Comments to the Example

The purpose of this example is to calculate a sum of sixty-four 32-bit elements. This computation is made on the MU with the help of the weighted accumulation operation. The main idea of this transform is explained in Figure 1-4:

Figure 1-4. Computation of the Sum of sixty-four 32-bit Array Elements



The peculiarity of this example is that we load array elements into the Active matrix instead of weight coefficients. Weights are two bit elements. We send them to the input **X** of the Active Matrix.

The `sb = 0AAAAAAAAh` instruction configures the Shadow matrix. This configuration allows the user to load thirty-two vectors into the matrix and then to accumulate elements along each column for one cycle.

The instruction:

```
rep 32 wfifo = [ar1++], ftw, wtw;
```

loads vectors from memory to the `wfifo` register-container for further decoding into the Shadow matrix representation. As soon as the first vector appears in `wfifo`, the decoding process is activated (this operation is assigned by the `ftw` command). After the Shadow matrix is filled with decoded data its content copies to the Active matrix for one processor cycle.

The instruction:

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

executes accumulation of all the thirty two rows. The computation is made according to the scheme shown in Figure 1-4. The result of the operation is stored into to the `afifo` register-container.

The instruction:

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

uploads the result from `afifo` to external memory.

Then two partial sums are added on the RISC-core and the result of the calculation is returned in the `gr7` register.

Compilation of Example

To compile the example `step6b.asm` enter the following on a command line:

```
nmcc -g step6b.asm libc.lib -m
```

1.11 Lesson 7: How to Call an Assembly Function from C++

Starting from this lesson all examples are made in the form of assembly functions called from the main program written in C++.

The source code of the example used in this lesson can be found in the `main.cpp` and `step7.asm` files in the directory: `..\Tutorial\Step7`

Example Description

The example shows how to call a function written in NM6403 assembly from a C++ program.

Source Code

File “`main.cpp`”

```
extern "C" int Neg ( int value );
int main()
{
    int a = 16;
    return Neg(a);           // we call a function performing negation of the input
                            // parameter.
}
```

File “`step7.asm`”

```
global _Neg: label;        // declaration of the function name label

begin ".text"
<_Neg>
    ar5 = ar7 - 2;         // here we get the addresses in the stack to access input
```

```
                                // parameters
push ar0, gr0;                  // the registers used in the function should be stored in the stack

gr0 = [--ar5];                  // this is the way we obtain the input parameter
gr7 = - gr0;

pop ar0, gr0;                   // before return from the function we have to restore the registers
                                // used in the function

return;

end ".text";
```

Comments to the Example

This example shows how to call a function written in assembly language from a C++ program. It also explains some points of the calling convention accepted in the NeuroMatrix SDK.

The `main()` program is written in C++ while the `Neg()` function called from it is implemented in assembly.

To get access to an assembly function it is necessary to declare this function as the external symbol with C binding:

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

This declaration allows using the C naming convention for the function names instead of the C++ convention, which simplifies function names in assembly language. For example, if the function is declared the way it is shown above its assembly name will be: `_Neg`, but if we miss the `extern "C"` in the function declaration, then according to the C++ naming convention its name will be: `_Neg__Fi`.

You may put the function declaration to the header file or directly to the C++ source file.

The `step7.asm` file contains the implementation of the `Neg()` function. The function changes the sign of the input parameter.

To be accessible outside the assembly file the function name should be declared as global:

```
global _Neg: label;
```

According to the C naming convention the “`_`” symbol must be added in front of the label name in order to make it possible to call the function named `Neg` from the file in C++.

Then the code section follows

```
begin ".text"
```

```
<_Neg>
```

```
...
```

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

Actions described inside the code section will be executed by the `Neg()` function call.

Assembly functions developed for further calls from C++ programs should be organized in a certain way. The main requirement is to store the address and general purpose registers before they are used in the function and to restore their previous contents before the processor returns from the function. This requirement covers all the primary registers except `ar5` and `gr7`. In some particular cases described below, the register `gr6` can also be changed by the function.

Before we start executing the function let's revise the stack state. The `ar7` register is treated as a stack pointer and it points to the first non-occupied address in the stack, as shown in Figure 1-5.

Figure 1-5. The Stack Contents at the Moment of a Function Call

<i>empty</i>	0	← ar7
pswr	-1	
pc	-2	
1 st input parameter	-3	
2 nd input parameter	-4	
3 rd input parameter	-5	

When a function is called its input parameters and the `pc` and `pswr` registers are put to the stack. The `pc` and `pswr` registers are put there automatically at the moment of executing the `call` instruction, therefore they are located at the top of the stack.

To get access to the input parameters we store their address into the `ar5` register. The first function instruction should be the instruction:

```
ar5 = ar7 - 2;
```

The `ar5` register contains the pointer to the stack position previous to the first input parameter. To load the parameter from the stack to a register the following memory addressing mode is used:

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

First the `ar5` register decrements by one and then the processor reads data from memory. The next input parameter can be accessed the same way.

As mentioned above all the primary registers other than `ar5` and `gr7` must be stored into the stack before they are used in the function.

The instruction:

```
push ar0,gr0;
```

stores the register pair into the stack's top.

It is recommended to store 64-bit register pairs to the stack instead of alone registers. This recommendation arises from the requirement to keep the stack pointer even. The odd stack pointer may cause the program error in case an interrupt occurred.

More detailed information on how to handle the stack can be found in Section 5.1.4 Stack Commands in **NeuroMatrix NM6403 Assembly Language Overview**.

`gr0 = [--ar5];` - value of the function parameter is loaded to the `gr0` register.

`gr7 = - gr0;` - a function return value must be stored to the `gr7` register.

`pop ar0, gr0;` - restore the previous content of the register pair.

`return;` - return from the function.

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step7.asm main.cpp -oStep7.abs
```

The `nmcc` distinguishes file types by their extensions. For the files with the extension `.cpp` or `.c` it calls the components of the C++ compiler (preprocessor, front-end, code generator and so on). If the `nmcc` meets the C++ files among the input parameters, it automatically adds the C run-time library (`libc.lib`) as a parameter for the linker. Therefore, in case of compiling the C++ code the C run-time library name can be omitted on the command line.

1.12 Lesson 8: Input Parameters and Return Values of a LONG Type

The source code of the example used in this lesson can be found in the `main.cpp` and `step8.asm` files in the directory: `..\Tutorial\Step8`

Example Description

The example explains how to deliver a 64-bit word of a `long` type as an input parameter to a function and how to return a word of a `long` type from a function.

Both functions used in this lesson perform negation of a 64-bit input value. The difference is the `Neg_Scal` function executes this operation on the RISC-core, while the `Neg_Vect` function exploits the vector coprocessor (VCP).

Source Code

File "`main.cpp`"

```
extern "C" {
```

```
    // functions Neg_Scal and Neg_Vect are declared as external symbols with C binding
```

How to Write Programs in NM6403 Assembly

```
long Neg_Scal ( long value );
long Neg_Vect ( long value );
}

int main()
{
    long a = 0x2222222211111111;

    long b = Neg_Scal(a);
    long c = Neg_Vect(a);
    return (int)(b-c);
}
```

File "step8.asm"

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

```
nobits ".my_data"
```

```
    A: long;                // definition of A, a 64-bit variable of a long type
end ".my_data";
```

```
begin ".text"
```

```
// the Neg_Scal function performs negation of an input parameter on the RISC-core
```

```
<_Neg_Scal>
```

```
    ar5 = ar7 - 2;          // pointer to input parameter in the stack
    push ar0, gr0;
    push ar1, gr1;
    ar0,gr0 = [--ar5];      // we read the input parameter from the stack
    gr1 = ar0;              // the low 32-bit word of parameter is copied to gr1
```

```
// we fill gr1 with 0 at the left part of the instruction and simultaneously copy the low part of the
// function parameter with the opposite sign to gr7
```

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

```
// we change the sign of the gr0 register contents taking into account the carry flag value
// set by previous arithmetical operation
```

```
gr6 = gr1 - gr0 - 1 + carry;
```

```
pop ar1, gr1;
pop ar0, gr0;
return;           // a long return value is stored in the registers:
                  // the high 32 bits in gr6, the low 32 bits in gr7

// the Neg_Scal function performs negation of an input parameter on the RISC-core
<_Neg_Vect>
    ar5 = ar7 - 2;    // pointer to input parameter in the stack
    push ar0, gr0;
    push ar1, gr1;
    ar1 = A;         // the address of the A long word is loaded to ar1
    nb1 = 0;        // nb1 = 0 that means we process 64-bit words in the Vector ALU
    wtw;            // copy the contents of the nb1 shadow register to the
                  // nb2 active register

    // perform the negation operation in the Vector ALU, which changes the sign of the 64-bit value
    rep 1 data = [--ar5] with 0-data;
    // the result is put to memory at the address stored in the register ar1
    rep 1 [ar1] = afifo;
    gr7 = [ar1++];   // the low-order part of the result is copied from memory to the
                  // gr7 register
    gr6 = [ar1++];   // the high-order part of the result is copied from memory to the
                  // gr6 register

    pop ar1, gr1;
    pop ar0, gr0;
    return;

end ".text";
```

Comments to the Example

The example consists of two parts performing the same operation – changing of the sign of a 64-bit input value.

Performing Negation on the RISC-core

The function starts from loading the input parameter stack address into ar5 and saving the contents of the registers used in the body of the function.

Now the input parameter can be loaded. As it is of the `long` type (64-bit), it is necessary to use a register pair. In case the register pair is used for memory access the low 32-bit word of a 64-bit word goes to `ar0` and the high 32-bit word - to `gr0`:

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

Since an address register cannot be used in *arithmetical* operations, its value must be copied to a general purpose register:

```
gr1 = ar0;
```

The instruction:

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

fills `gr1` with zero and at the same time changes the sign of the `gr1` register content. After the instruction is performed the `gr1` register will contain zero, while the `gr7` register will contain the old negated value contained in `gr1` before the instruction.

The `gr1` register is used in the left and in the right part of the instruction. In this case a simple rule describes changing of its contents:

Note

The left and the right parts of the instruction are executed simultaneously, and the old contents of the registers involved are used in calculations.

Thus, the old content of the register `gr1` is used in the right part of the instruction and in the result a new value copies to it. Use of the same register in both parts of the instruction is described in the Appendix XXXX.

The instruction:

```
gr6 = gr1 - gr0 - 1 + carry;
```

makes subtraction of the contents of two registers taking into account the carry flag value set by the previous *arithmetical* operation (for more details see Section 5.1.11 Arithmetical Operations in **NeuroMatrix NM6403 Assembly Language Overview**).

In case the function returns a 64-bit value, the low-order part is always loaded to `gr7`, and the high-order part - to `gr6`.

Performing Negation on the VCP

The instructions:

```
nb1 = 0;
```

```
wtw;
```

define division of a 64-bit vector into elements. In case `nb1 = 0` the vector consists of one 64-bit element, witch means that the Vector ALU will perform operations on 64- bit words.

```
rep 1 data = [--ar5] with 0-data; - is a vector instruction  
executing an arithmetical operation on the Vector ALU (difference of
```

operands **X** and **Y**, where a zero vector is sent to the operand **X**). The operation changes the sign of the input parameter.

```
rep 1 [ar1] = afifo; - the result from afifo is stored to the
variable A, which address is in ar1.
```

Then the high-order and the low-order parts of A are copied to the registers gr6 and gr7 respectively.

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step8.asm main.cpp -oStep8.abs
```

1.13 Lesson 9: Logical Activation and Masking on VCP

The source code of the example used in this lesson can be found in the step9.asm file in the directory: ..\Tutorial\Step9

Example Description

In the example we apply threshold function to elements of a data vector. If the element value is greater or equal to the threshold value it is replaced by the threshold value. For example:

Table 1-1. Threshold Application Results

THRESHOLD		VECTOR VALUE		RESULT
44h	apply to	12h	→	12h
44h	apply to	44h	→	44h
44h	apply to	57h	→	44h

The example shows how to sort out this task using the Vector ALU operations of logical activation and masking. Logical activation is applied to input data in one of the *VCP Activation Units (AU)*, while masking is performed in the *VCP Mask Application Unit (MAU)*. More detailed information about the Activation Units and the Mask Application Unit can be found in Sections XXXX and XXXX of the **NeuroMatrix® NM6403 Assembly Language Overview** respectively.

Source Code

File "main.cpp"

```
//function Mask is declared as external with C binding
extern "C" void Mask ( long *A, int mask );

long A[32]; // definition of an array of thirty two 64-bit vectors

int main()
```

How to Write Programs in NM6403 Assembly

```
{
    for ( int i=0; i< 32; i++)
    {
        // the array is filled with initial values
        A[i] = 0x0102030405060708*i;
    }
    Mask(A, 0x44);           // we call the Mask function. The first parameter is the address of
                            // the array, the second parameter is the mask value

    return 0;
}
```

File "step9.asm"

```
global _Mask: label;      // declaration of the _Mask global label

data ".my_data"
    Temp: long = 01;
end ".my_data";
begin ".text"
<_Mask>
    ar5 = ar7 - 2;        // pointer to input parameters in the stack
    push ar0, gr0;
    push ar1, gr1;
    push ar2, gr2;

    ar0 = [--ar5];        // the array address is loaded to ar0
    gr0 = [--ar5];        // the mask value: 00000044h is loaded to gr0

    ar2 = ar0;            // the address of the input array is copied to ar2
    gr1 = gr0 << 8;       // gr1 = 00004400h
    gr0 = gr0 or gr1;     // gr0 = 00004444h
    gr1 = gr0 << 16;      // gr1 = 44440000h
    gr1 = gr0 or gr1;     // gr0 = 44444444h
    ar1 = gr1;            // ar0,gr0 = 4444444444444444h

    // the expanded mask value: 4444444444444444h is stored to the Temp variable,
    // which address is put to ar1
    [ar1 = Temp] = ar1, gr1;
```

```
nb1 = 80808080h;    // The Vector ALU processes eight 8-bit elements in parallel
wtw;
// the flcr activation control register sets the X Activation Unit (AU) configuration
flcr = 80808080h;

// Load thirty-two 64-bit vectors to the internal FIFO of the VCP
rep 32 ram = [ar0++];
// Thirty-two times subtract the threshold value from the vectors
rep 32 data = [ar1] with ram - data;

// apply the logical activation function to the result of the previous operation.
rep 32 with not activate afifo;
// perform masking operation, the thirty-two mask vectors come from afifo, the threshold value
from the data bus, and the input vectors come from ram
rep 32 data = [ar1] with mask afifo, data, ram;
// the result of the operation replaces the initial vector array
rep 32 [ar2++] = afifo;

pop ar2, gr2;
pop ar1, gr1;
pop ar0, gr0;
return;

end ".text";
```

Comments to the Example

The `Mask` function compares the value of each 8-bit vector element with the threshold value. If the vector element exceeds the threshold the function replaces it by the threshold value. For example, if the initial vector contents are `1122334455667788h1`, then after applying the threshold `44h` to its every element, it will result in `1122334444444444h1`.

A byte threshold is given to the function input. The `Mask` function first expands the byte value to prepare a 64-bit threshold vector:

```
gr1 = gr0 << 8;    // gr1= 00004400h
gr0 = gr0 or gr1; // gr0 = 00004444h
gr1 = gr0 << 16; // gr1 = 44440000h
gr0 = gr0 or gr1; // gr0 = 44444444h
```

```

ar1 = gr1;           // copy the gr1 contents to the ar1 pair
                    // complementary register

[ar1 = Temp] = ar1,gr1; // copy the threshold vector to memory
    
```

The last instruction copies the register pair contents to memory at the address `Temp`, and then puts this address to the `ar1` register.

A new item introduced in this example is an activation unit control register: `f1cr` (`f2cr`). The register `f1cr` is used for control of activation block located in the way of data sent to the operand **X** of the vector processor, the register `f2cr` is connected with the operand **Y**.

`f1cr = 80808080h;` defines division of a 64-bit word coming to the input **X** of the VCP for applying a logical activation, into eight 8-bit elements. In the general case this division does not necessarily coincide with that assigned by `nb1` (for more details see Section 3.3.1 Registers `f1cr` and `f2cr` in **NM6403 Assembly Language Overview**).

The instruction:

```
rep 32 ram = [ar0++];
```

loads input data vectors to the `ram` register-container.

The instruction:

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

thirty-two times reads the threshold vector from the same memory address and sends it to the Vector ALU input where subtraction of the input data vectors (`ram`) and the threshold vectors (`data`) is performed. The results of calculations come to `afifo`.

Logical Activation

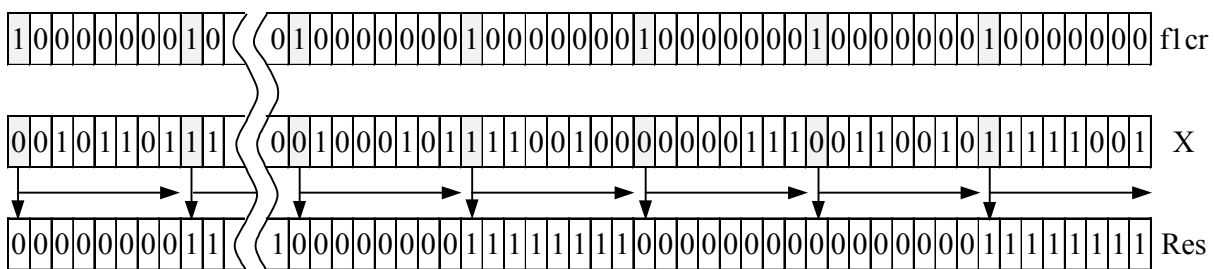
The instruction:

```
rep 32 with not activate afifo;
```

performs logical activation on the results of the previous operation contained in `afifo`.

When executing logical activation the processor analyzes the input data vector bits located in the positions where the register `f1cr` has non-zero bits (see Figure 1-6).

Figure 1-6. Sign Bits Expansion by Logical Activation



The bits located at the same positions as the non-zero bits of `f1cr` expand to the right till they reach the next non-zero `f1cr` bit position.

In the example we have selected the `f1cr` value the way its non-zero bits are set to the same positions as the MSB of the elements of input data vectors, i.e. the sign bits.

If the threshold value is greater then the content of a vector element then the result of subtraction will be a negative value. The MSB of a negative value is set to “1”.

Therefore, after logical activation applied to the results of subtraction the non-negative vector elements will be replaced by `00h` while the negative elements will become `0FFh`.

In case of the instruction above the logical `not` operation is additionally executed over the activated operand changing bit values to the opposite.

In the result of data transform in the VCP the negative difference of vector elements and the threshold will be replaced by `00h`, the other results will be replaced by `0FFh`. The negative difference means that the input vector element value is less than the threshold. The result of the operation will be stored to `afifo`.

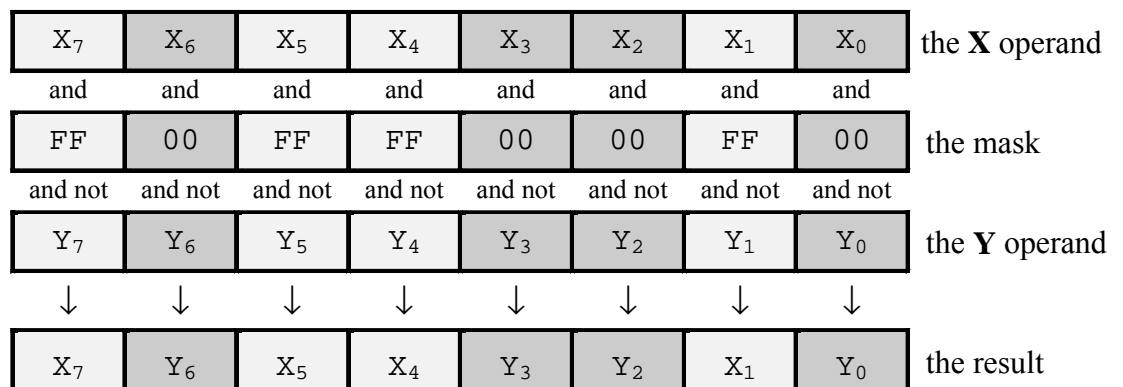
The important question we have to revise is the order of transforms to be applied to an input data vector in the VCP. The detail information about the order can be found in Section 1.4.6 XXXX in **NM6403 Assembly Language Overview**.

According to the VCP structure first the logical activation is applied to the input data vector in the AU and then the `not` operation will be performed on the results of activation.

Logical Mask Application

According to the example the contents of `afifo` is used as a mask that is applied to two input vectors: the input data vector and the threshold vector. The purpose of masking is to select only particular data from both vectors and to combine them into a new vector. The essence of the masking operation is given in Figure 1-7:

Figure 1-7. Logical Masking Operation



The instruction:

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

applies the mask contained in `afifo` to the threshold vector (`data`) as the **X** operand and to the input data vector (`ram`) as the **Y** operand.

The following transform is made over all sets of vectors taking part in computations:

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

As you can see in Figure 1-7, in those positions where the mask bits are set to “1”, the **X** operand bits will pass through to the result vector. In those positions where the mask bits are set to “0” you will find the bits coming from the operand **Y**.

The instruction:

```
rep 32 [ar2++] = afifo;
```

stores thirty-two result vectors into the external memory.

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step9.asm main.cpp -oStep9.abs
```

1.14 Lesson 10: Arithmetical Activation on VCP

The source code of the example used in this lesson can be found in the `step10.asm` file in the directory: `..\Tutorial\Step10`

Example Description

The example explains what the term “arithmetical activation” means and how to use this operation in calculations on the VCP.

The function `AddSaturate()` executes element-by-element add of two byte arrays of elements which values are in the range from -128 to 127.

In case an overflow occurs the function replaces the element content by the minimum possible value `0xFF` (-128) in case of negative overflow or with the maximum possible value `0x7F` (127) in case of positive overflow. In other words, arithmetical activation means saturation.

Source Code

File “`main.cpp`”

```
// the AddSaturate function is declared as external function with C binding
```

```
extern "C" void AddSaturate( long* Src1, long* Src2, long* Dst);
```

```
long SRC1[32];           // first input array
```

```
long SRC2[32];           // second input array
```

```
long DST[32];            // array of results
```

```
int main()
{ //fill the input arrays with initial values
  for (int i = 0; i < 32; i++)
  {
    SRC1[i] = 0x0203040504030201*i;
    SRC2[i] = 0x0807060804050607*i;
  }
  // call the assembly function with three input parameters
  AddSaturate( SRC1, SRC2, DST );
  return 0;
}
```

File "step10.asm"

```
global _AddSaturate: label;
data ".data"
  Masks: long[24] = ( 0000000000000001h1, // matrix for the first pass
                    0000000000010000h1,
                    0000000100000000h1,
                    0001000000000000h1,
                    0000000000000000h1 dup 4,

                    0000000000000001h1, // matrix for the second pass
                    0000000000000100h1,
                    0000000000010000h1,
                    0000000001000000h1,

                    0000000000000000h1 dup 4, // matrix for the third pass
                    0000000000000001h1,
                    0000000000010000h1,
                    0000000100000000h1,
                    0001000000000000h1,

                    0000000100000000h1, // matrix for the fourth pass
                    0000010000000000h1,
                    0001000000000000h1,
                    0100000000000000h1);

end ".data";
```

How to Write Programs in NM6403 Assembly

```
begin ".text"
<_AddSaturate>
    ar5 = sp - 2;
    push ar0, gr0;
    push ar1, gr1;
    push ar4, gr4;
    push ar6, gr6;

    gr0 = [--ar5];           // the first input parameter (SRC1)
    gr1 = [--ar5];           // the second input parameter (SRC2)
    ar4 = [--ar5];           // the third input parameter (DST)

    ar0 = gr0;
    ar1 = gr1;

    ar6 = Masks;             // the weight coefficients buffer address

    flcr = 0FF80FF80h;       // configuration of the first Activation Unit (X)

    // definition of the Active Matrix configuration for the first step of computations
    nb1 = 80008000h;         // 4 columns
    sb  = 03030303h;         // 8 rows

    // all weight coefficients (for four matrixes) are loaded to wfifo right away,
    // and only eight words are transferred to the Shadow Matrix according to values of sb and nb1
    rep 24 wfifo = [ar6++],ftw, wtw;

    // since the working matrix is already loaded, it is possible to start loading a new portion of
    // weight coefficients to the Shadow Matrix and to define new configuration.
    nb1 = 80808080h;         // 8 columns
    sb  = 00030003h;         // 4 rows

    // computations on the Active Matrix are made in parallel with loading the Shadow Matrix,
    // the next two instructions execute transform of word-lengths and addition of input vectors
    // elements with the same index.
    rep 32 data = [ar0++], ftw with vsum , data, 0;
    rep 32 data = [ar1++] with vsum , data, afifo;

    wtw;                     // copy the Shadow Matrix to the Active Matrix
```

```
nb1 = 80008000h;    // 4 columns
sb  = 03030303h;    // 8 rows

// execution of arithmetical activation with further word-length cutting
rep 32 ftw with vsum , activate afifo, 0;

// go back to the beginning of initial arrays for processing of the second halves of vectors
ar0 = gr0;
ar1 = gr1;

// move the results of the first step of transform into ram
rep 32 [ar4],ram = afifo;
wtw;

// the second step of computations completely repeats the first one, the only difference is in
// matrix weights.
nb1 = 80808080h;    // 8 columns
sb  = 00030003h;    // 4 rows

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

wtw;

// the instruction activates data, transforms the word-length and adds to the first pass results.
rep 32 with vsum , activate afifo, ram;

// the result of computations is stored into memory.
rep 32 [ar4++] = afifo;

pop ar6, gr6;
pop ar4, gr4;
pop ar1, gr1;
pop ar0, gr0;
return;

end ".text";
```

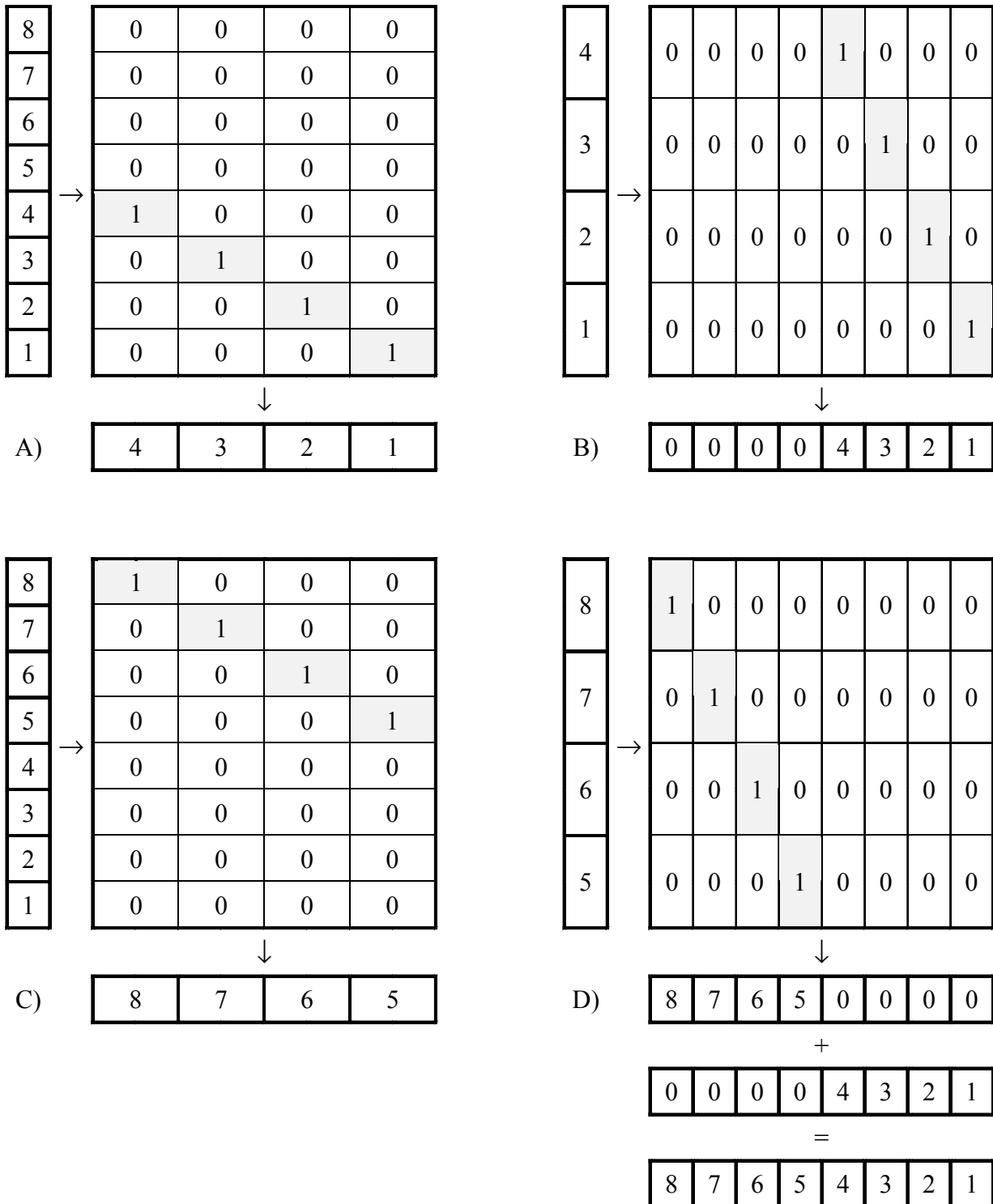
Comments to the Example

In this example we divide all calculations into two stages. This is determined by bit-length transform for intermediate data representation.

How to Write Programs in NM6403 Assembly

At first we process four low-order elements of input vectors and then four high-order ones (see Figure 1-8):

Figure 1-8. Changing Bit-Length for Intermediate Representation



The function performs the following sequence of actions:

- Loads weight coefficients to make bit-length transform from 8-bit to 16-bit representation as shown in Figure 1-8A;

- Makes bit-length transform and adds the elements of input vector arrays with the same index;
- Applies arithmetical activation (saturation) to replace the results exceeding data range: $-128 \dots 127$, by the maximum/minimum range values;
- Makes reverse bit-length transform from 16-bit to 8-bit representation as shown in Figure 1-8B;
- Stores the result of the first pass to the `ram` internal FIFO buffer;
- Makes the same calculations over high-order elements of the input vectors, as shown in Figure 1-8C and in Figure 1-8D. The same instruction adds the results of the first and the second passes.

Let's discuss the algorithm implementation in more details.

Although weight coefficients belong to four for different matrix configurations they are loaded to `wfifo` with one instruction:

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

One of `wfifo` features is ability to load in and out not only whole contents but to parts of weight coefficients too. For example, you are able to load in some additional weight coefficients expanding the previous contents. The maximum number of weight coefficient words to be placed to `wfifo` at the same time is thirty-two.

According to the predefined Shadow Matrix configuration (`nb1` and `sb`) the processor loads the appropriate number of weight coefficient words from `wfifo` and transforms them into the Shadow Matrix internal representation (`ftw`). The rest of weight coefficients stay in `wfifo` waiting for the time to be used. After the Shadow Matrix is filled with new weights the processor copies its contents to the Active Matrix (`wtw`).

The instruction:

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

performs both weighted accumulation in the Multiply Unit (`vsum`) and transforming of a new portion of weights to the Shadow Matrix representation (`ftw`).

The instruction:

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

performs arithmetical activation (saturation) on a vector set stored in `afifo`. After that the saturated data come to the input of the Multiply Unit to be transformed from 16-bit to 8-bit representation.

Why Do We Use the Terms "Arithmetical" and "Logical" Activation

The arithmetical and logical activation are two modes of the Activation Unit. The logical mode turns on when the processor performs a logical operation on the Vector ALU, while the arithmetical mode turns on in case the processor performs arithmetical operations on the Vector ALU

or executes weighted accumulation. The example giving an idea about logical activation has been described in the previous lesson. Here we observe the basic principles of arithmetical activation.

Figure 1-9 explains how the `f1cr` (`f2cr`) register controls execution of arithmetical activation:

Figure 1-9. Saturation by Arithmetical Activation



Exceeding the range $-128 \dots 127$ when processing 16-bit elements can be detected according to the following principles: different values of sign bits. If a value of a 16-bit word lies in the range mentioned above, the bits from 7th to 15th are the sign bits (they are marked with gray in Figure 1-9). In case at least sign bit value differs of the others the Activation Unit detects range exceeding and replaces the element contents by the maximum/minimum range values. This is the basic principle of saturation.

In the `f1cr` register all sign bits are marked with “1”. All the others are marked with “0”. When processing an input data vector the Activation Unit analyses the bits of the input vector located at the same positions as the “1”s of the `f1cr`(`f2cr`) register. If no exceeding is detected the word goes through without any changes. In the other case it is replaced by the maximum/minimum range values depending on the MSB value.

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step10.asm main.cpp -oStep10.abs
```

1.15 Lesson 11: Use of Cyclic Shifter One Bit Right

The source code of the example used in this lesson can be found in the `step11.asm` and `main.cpp` files in the directory:

```
..\Tutorial\Step11
```

Example Description

The lesson explains what the Cyclic Shifter One Bit Right (CSOBR) can be used for, and shows how it works. The CSOBR is one of the computing nodes of the VCP.

The example below swaps the bits of an input 64-bit data word to the reversed order. It swaps the 0th and the 63rd bits, the 1st and the 62nd bits and so on.

Source Code

File "main.cpp"

// the ReverseBits function is declared as external function with C binding

```
extern "C" long ReverseBits(long Src);
```

```
int main()
```

```
{
```

```
    long A = 0x5555EEEEAAAA7777; // input 64-bit word
```

```
    long B = 0xEEEE55557777AAAA; // the expected result of the transform
```

```
    long C = ReverseBits(A);      // the calculated result of the transform
```

```
// comparison of the calculated result with the expected one
```

```
    if (B == C) return 1;        // the calculated result is correct
```

```
    else return -1;             // the calculated result is incorrect
```

```
}
```

File "step11.asm"

```
global _ReverseBits: label;
```

```
data ".data" //weight coefficients for changing the bits order
```

```
Weights: long[64] = ( 11<<63, 11<<61, 11<<59, 11<<57,  
                      11<<55, 11<<53, 11<<51, 11<<49,  
                      11<<47, 11<<45, 11<<43, 11<<41,  
                      11<<39, 11<<37, 11<<35, 11<<33,  
                      11<<31, 11<<29, 11<<27, 11<<25,  
                      11<<23, 11<<21, 11<<19, 11<<17,  
                      11<<15, 11<<13, 11<<11, 11<< 9,  
                      11<< 7, 11<< 5, 11<< 3, 11<< 1,  
                      11<<62, 11<<60, 11<<58, 11<<56,  
                      11<<54, 11<<52, 11<<50, 11<<48,  
                      11<<46, 11<<44, 11<<42, 11<<40,  
                      11<<38, 11<<36, 11<<34, 11<<32,  
                      11<<30, 11<<28, 11<<26, 11<<24,  
                      11<<22, 11<<20, 11<<18, 11<<16,  
                      11<<14, 11<<12, 11<<10, 11<< 8,  
                      11<< 6, 11<< 4, 11<< 2, 11<< 0);
```

```
end ".data";
```

How to Write Programs in NM6403 Assembly

```
begin ".text"
<_ReverseBits>
    ar5 = sp - 2;
    push ar0, gr0;

    ar0 = Weights;           // load the address of Weights array into ar0

    nb1 = 0FFFFFFFFh;       // 64 columns
    sb  = 0FFFFFFFFh;       // 32 rows

    // load the first part of weight coefficients into the Active Matrix
    rep 32 wfifo = [ar0++],ftw, wtw;
    // load the second part of weight coefficients into the Shadow Matrix
    rep 32 wfifo = [ar0++],ftw;

    // an input data word is loaded directly from the stack into ram and at the same time goes to
    // the X input of the Multiply Unit
    rep 1 ram = [--ar5] with vsum , data, 0;
    // copy the contents of the Shadow Matrix to the Active Matrix
    wtw;
    // after changing the weight coefficients the same input word goes from ram to the X input of
    // the Multiply Unit (MU). On the way to the MU it rotates one bit right in the CSOBR.
    rep 1 with vsum , shift ram, afifo;
    // the result is stored into memory
    rep 1 [ar5] = afifo;

    gr7 = [ar5++];         // the low thirty-two bits of the word go to gr7
    gr6 = [ar5++];         // the high thirty-two bits of the word go to gr6

    pop ar0, gr0;
    return;

end ".text";
```

Comments to the Example

In this example we use constant expressions for filling the Weights array. Those constant expressions are calculated at the stage of compiling. The

compiler is able to calculate 64-bit constant expressions for filling the memory cells.

The example shows the Active/Shadow Matrix configuration exploiting the maximum possible number of rows and columns: 64 columns and 32 rows. In this case input data vectors treat as they consist of thirty-two elements two bits length each, but the result vectors contain sixty-four binary elements.

The instruction:

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

loads an input data vector directly from the stack. The processor stores the vector to the `ram` internal FIFO buffer. While the vector is going along the input data bus it copies to the **X** input of the Multiply Unit for the first stage of calculations.

The instruction:

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

activates the Cyclic Shifter (the `shift` key word stands for the CSOBR). Before the data vector from `ram` comes to the X input of the Multiply Unit it goes through the Cyclic Shifter. The CSOBR rotates the whole vector one bit right, so the 0th bit of the vector becomes the 63rd bit, the 1st bit goes to the 0th position and so on.

The rotation one bit right can be applied only to data vectors coming to the **X** input of the Multiply Unit. The CSOBR does not take into account splitting of the vector into elements. When applying rotation one bit right to a data vector, the LSBs of the vector elements become the MSBs of their right neighbors. If such a behavior of the CSOBR affects the correct data processing, it is possible to mask the bits coming from one vector element to another. In our case we do not need to apply any masking.

When the Multiply Unit is split into 32 rows and 64 columns the only even bits of an input data vector are processed, the odd bits are ignored.

So, the input vector passes twice though the Multiply Unit. First the even bits of the vector change their positions. At the second stage the odd bits of the vector become the even ones. It happens when the data vector goes though the CSOBR. The odd bits that became even go the Multiply Unit for further processing. This is the way we process all the bits of the input vector.

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step11.asm main.cpp -oStep11.abs
```

1.16 Lesson 12: Use of VR Vector Register

The source code of the example used in this lesson can be found in the `step12.asm` and `main.cpp` files in the directory:

..\Tutorial\Step12

Example Description

The lesson gives an idea on how to use a `vr` vector bias register. In this example we add a constant, contained in `vr` to all 16-bit vector elements.

Source Code

File “main.cpp”

```
// the AddBias function is declared as an external function with C binding
extern "C" void AddBias( long* buff,    // pointer to a vectors array
                        int size,      // a number of 64-bit vectors in an array
                        long bias );   // bias to be add to vector elements

long Data[1024];    // array of 1024 64-bit vectors
                   // (each vector contains four 16-bit elements )

int main()
{
    // we fill the array vectors with initial values
    Data[0] = 0x0001000100010001;
    for ( int i = 1; i < 1024; i++ )
        Data[i] = Data[i-1] + 0x0001000100010001;

    // call the AddBias function
    AddBias( Data, 1024, 0x0012001200120012 );

    return 0;
}
```

File “step12.asm”

```
global _AddBias :label;

data ".data"
    // weight coefficients, the non-zero elements are placed diagonally, the Multiply Unit makes no
    // changes over incoming data vectors
    Weights: long[4] = ( 11, 11<<16, 11<<32, 11<<48 );
end ".data";

begin ".textAAA"
<_AddBias>
```

```
ar5 = sp - 2;
push ar0, gr0;
push ar4, gr4;

gr4 = [--ar5];      // array address
gr0 = [--ar5];      // a number of vectors in an array

nb1 = 80008000h;    // 4 columns
sb  = 00030003h;    // 4 rows

ar4 = Weights;

// load the weight coefficients into the Active Matrix
rep 4 wfifo = [ar4++], ftw, wtw;

vr = [--ar5];      // load a constant bias vector into vr

// gr0 will contain a number of loops, each loop processes thirty-two vectors
ar0 = gr4 with gr0 >>= 5;
ar4 = gr4 with gr0--;
<Loop>
  if > delayed goto Loop with gr0--;
  // input data vectors, coming through the Multiply Unit, add a vr bias to their contents
  rep 32 data = [ar0++] with vsum , data, vr;
  rep 32 [ar4++] = afifo;

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

Comments to the Example

The AddBias function adds a constant to each 16-bit element of vectors. The processor reforms this operation on the VCP using the `vr` vector bias register.

The `vr` register has an advantage that it does not take long time to load it with a constant and there is no matter how many vectors will be processed per one instruction. The main purpose of this register is to add

a constant bias vector to a result of weighted accumulation. As it seems, it is an alternative to `ram`, containing the same constant vectors. In some cases you can use `vr` instead of the main VCP buffers (`ram`, `data`, `afifo`). It is possible when the `vr` register is used as the **Y** input in weighted accumulation operations. More detailed information can be found in Section 3.3.4 The `vr` Register of the **NeuroMatrix® NM6403 Assembly Language Overview**.

The instruction:

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

loads a constant bias vector directly from the stack to the 64-bit `vr` register.

The instruction:

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

adds the constant bias vector contained in `vr` to the results of weighted accumulation.

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step12.asm main.cpp -oStep12.abs
```

1.17 Lesson 13: Macros

The source code of the example used in this lesson can be found in the `step13.asm` and `main.cpp` files in the directory:

```
..\Tutorial\Step13
```

Example Description

This lesson explains how to declare and use macros and their arguments. It also gives an explanation on how to define labels inside the macro body.

The `Copy` function copies the contents of one buffer to another. In the function body we use a macro declared at the same file. This macro makes copy operation itself.

Source Code

File “`main.cpp`”

```
extern "C" void Copy( long *Src, long *Dst );
long A[16];           // source data array
long B[16];           // results array

int main()
{
```

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

Copy( A, B );           // the Copy function copies the A array to the B array
return 0;
}
```

File “step13.asm”

```
global _Copy: label;

// Declaration of a macro making copy of one 64-bit array to another.
// The first argument is an address of the source array, the second one is the destination array,
// the third argument is the number of array elements.
macro CopyBlock (Arg1, Arg2, Arg3)
    own Loop: label;           // declaration of a label inside the macro

    gr1 = Arg3;               // gr1 is loaded with a number of loop iterations
    gr1--;                     // setting the condition flags for the first entry to the loop
<Loop>
    // conditional delayed branch with modification of the iterator
    if > delayed goto Loop with gr1--;
        rep 1 data = [Arg1++] with data;
        rep 1 [Arg2++] = afifo;
end CopyBlock;

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

    ar0 = [--ar5];           // source array address
    ar1 = [--ar5];           // destination array address
    // here the CopyBlock macro body will be inserted
    CopyBlock(ar0, ar1, 16);

    pop ar2, gr2;
```

```
pop ar1, gr1;
pop ar0, gr0;
return;
end ".textAAA";
```

Comments to the Example

The `Copy` uses the `AAA` macro to copy one array to another. In the function body we call the macro as it would be a function with arguments:

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

The first argument is the `A` source array, the second argument is the `B` destination array, the third argument is the number of elements to be copied: sixteen 64-bit elements.

Before you call a macro it is necessary to describe its body. This description is usually located outside of data and code sections.

Macro description starts from the key word:

```
macro
```

which plays a role of the opening bracket. The `macro` is followed by the macro name and macro arguments put into round brackets.

The macro description finishes with the closing bracket:

```
end
```

followed by the macro name:

```
macro CopyBlock (Arg1, Arg2, Arg3)
```

```
...
```

```
end CopyBlock;
```

When the compiler unfolds the macro, it replaces formal arguments by the real register or constant names and values. Only registers and constant expressions can be used as macro arguments but not instructions or sets of instructions.

When the macro is unfolded it becomes a part of source code ready to compile. More detailed description of macros can be found in Appendix XXXX.

If you are going to use a label inside the macro you have to put the own keyword as the first word in the label declaration.

```
own Loop: label;
```

In that case while unfolding the macro the assembler substitutes the label name with the unique name. Therefore, the macro containing internal label definition can be used in the program more than once.

It is possible to create macro libraries of macros defined in different assembly files. The next lesson explains how to create macro libraries,

but here you will see what to do to use a macro contained in a macro library. If the macro you have to call is defined in a macro library it is possible to import that macro to the current file. Before you call the macro you have to put the following string to your assembly file:

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

where M.mlb is a macro library containing the macro you are going to use.

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step13.asm main.cpp -oStep13.abs
```

1.18 Lesson 13a: How to Create a Macro Library

The source code of the example used in this lesson can be found in the macros1.asm, macros2.asm, step13a.asm and main.cpp files in the directory:

```
..\Tutorial\Step13a
```

Example Description

On the basis of the previous lesson this example shows how to use macros defined in macro libraries and how to create a macro library. Moreover, one of the macros given here contains conditional compiling directives.

Source Code

```
File "macros1.asm"           // This macro copies data from one array to another
macro CopyBlock (Arg1, Arg2, Arg3)
    own Loop: label;           // declaration of a label inside the macro

    gr1 = Arg3;                // gr1 is loaded with a number of loop iterations
    gr1--;                      // setting the condition flags for the first entry to the loop
<Loop>
    // conditional delayed branch with modification of the iterator
    if > delayed goto Loop with gr1--;
        rep 1 data = [Arg1++] with data;
        rep 1 [Arg2++] = afifo;
end CopyBlock;
```

File "macros2.asm"

```
macro Push_Pop (Arg1)
    .if Arg1 xor 1;           // the beginning of the conditional compiling block of code
```

```
    push ar0, gr0;
    push ar1, gr1;
    push ar2, gr2;
.endif;           // the end of the first conditional compiling block
.if Arg1;        // the beginning of the alternative conditional compiling block
    pop ar2, gr2;
    pop ar1, gr1;
    pop ar0, gr0;
.endif;          // the end of the conditional compiling block
end Push_Pop;
```

Comments to the Example

The `Push_Pop` macro contains conditional compiling block. If an actual argument of the macro is '0' the contents of the register pairs is stored into the stack, otherwise the contents of the register pairs is restored from the stack. At the compiling stage the macro will be unfolded according to the argument it receives.

To build conditional compiling blocks we use the `.if`, `.else`, and `.endif` directives. For more details on conditional compilation directives see the paragraphs 2.7.3 Directives `.if`, `.else`, and `.endif`. at the **NeuroMatrix® NM6403 Assembly Language Overview**.

How to Create a Macro Library

To create a macro library that would contain definitions of the `CopyBlock` and the `Push_Pop` macros first you have to enter the following on a command line:

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

The compiler creates the `macros.mlb` macro library containing the `CopyBlock` macro.

To add the `Push_Pop` macro into the library we enter the following on a command line:

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

The `-a` option means that we add macros contained in the current assembly source file to the library created during the previous sessions. More detailed description of macro syntax and some examples are given in Appendix XXXX.

After the macro library is created we are able to import the included macros to our applications calling them the similar way as functions.

File "*main.cpp*"

```
// the Copy function is declared as an external C-binding function
extern "C" void Copy( long *Src, long *Dst );
long A[16];           // source data array
long B[16];           // destination data array

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

    Copy( A, B );     // we call the Copy function
    return 0;
}
```

File "step13a.asm"

```
global _Copy: label; // the _Copy global label definition

// we declare that we are going to use the macros contained in the macros.mlb macro library
import from macros.mlb;

begin ".textAAA"
<_Copy>
    ar5 = ar7 - 2;
    Push_Pop(0); // the first macro declared in the library (store primary registers)

    ar0 = [--ar5]; // the address of the A array is loaded to ar0
    ar1 = [--ar5]; // the address of the B array is loaded to ar1
    // we call the CopyBlock macro defined in the macro library
    CopyBlock(ar0, ar1, 16);

    Push_Pop(1); // one more call of macro (restore primary registers)
    return;
end ".textAAA";
```

Compilation of the Example

To compile the example enter the following on a command line:

```
nmcc -g -m Step13a.asm main.cpp -I..\Include -oStep13a.abs
```

where `Include` is the directory containing the `macros.mlb` macro library.

This chapter describes the main approaches used for assembly source code optimization, memory allocation for code and data, distribution of the VCP resources among the instructions executed in parallel.

2.1 Lesson 14: Methods of Programs Optimizing

The source code of the example used in this lesson can be found in the `step14.asm` file in the directory: `..\Tutorial\Step14`

The example contains description of the main approaches used by assembly **code** optimization.

File "step16.asm"

```
Global __main: label;
```

```
Data ".MyData"
```

```
Global A: long[16] = ( 01, 11, 21, 31, 41, 5h1, 61, 71, 81, 91, 101,  
0Bh1, 0Ch1, 131, 141, 151);
```

```
End ".MyData";
```

```
nobits ".MyData1"
```

```
global C: long[16];
```

```
end ".MyData1";
```

```
begin ".text"
```

```
<__main>
```

```
.branch;
```

*//copying the data array with the help of
the vector processor*

```
ar0 = A;
```

```
ar4 = C;
```

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

```
rep 16 [ar4++] = afifo;
```

```
return;
```

```
.wait;
```

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

Comments to the Example

To see how effectively the program is executed it is necessary to execute it on the accurate emulator `temu`. To do this enter the following instruction in the command line:

```
temu -S <file name1> -B<file name2> -L<file name3>  
a.abs
```

where `a.abs` – name of the absolute file of the program

After execution of the instruction three files will be created:

- `<file name1>` - a file of statistics where general data about the number of instructions (including empty instructions), the number of cycles for which the program is executed, percentage of empty instructions, etc. are given;
- `<file name 2>` - a file containing information on what goes on at each cycle on data buses;
- `<file name 3>` - a listing of the program execution describing which instruction is executed at every cycle.

If one of the keys (`-S`, `-L` or `-B` with file name indication) is not indicated in the instruction, the information will be output to the screen.

Thus, with the help of these files one can observe execution of the program by introducing changes into it.

The main requirement for effective work of the programs:

No **common** resources should be used in the program.

There are three compiler pseudo-instructions in the **assembly**, with the help of which it is possible to enable or ban parallel execution of instructions.

- 1) directive `.branch` allows switching on the mode of parallel execution of the processor instructions. Bit of **parallelism** is set equal to one in all processor instructions following the pseudo-instruction till the moment when the end of the current section has been achieved or the directive `.wait` has been met. By default the bit of **parallelism** is discarded to 0.
- 2) directive `.wait` sets the bit of **parallelism** into 0. In the result each processor instruction will wait execution of the previous instruction before being executed.

For parallel execution in the example of vector instructions

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

```
rep 16 [ar4++] = afifo;
```

the following three conditions should be fulfilled:

- 1) it is necessary to indicate the directive `.branch` at the beginning of the function body.
- 2) use address registers from different register groups (`ar0` from one group, `ar4` from another one).
- 3) Address registers should indicate on different data buses (`local` and `global`)

Since the register-**container** `afifo` is a two-port buffer, vector instructions can write and read information from it in parallel.

There are some cases when vector instructions cannot be executed in parallel, for instance instructions

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

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

can be executed in parallel under no circumstances, because the first instruction uses `afifo` for reading and writing simultaneously.

Compilation of the Example

To compile the example and create files of statistics and listing enter the following instructions in the command line:

```
nmcc Step16.asm Libc.lib -m -cstep16.cfg
```

```
temu -lstep16.lll -sstep16.sss -b step16.abs
```



Research Centre Module
Box: 166, Moscow, 125190, Russia
Tel: +7 (095) 152-9335
Fax: +7 (095) 152-4661
E-Mail: postmast@module.ru
WWW: <http://www.module.ru>

©RC Module, 2000

All rights reserved.

Neither the whole nor any part of the information contained in, or the product described in this overview may be adapted or reproduced in any form except with the prior written permission of the copyright holder.

RC Module reserves the right to make changes without further notices to product herein to improve reliability, function or design. RC Module shall not be liable for any loss or damage arising from the use of any information in this overview or any error or omission in such information, or any incorrect use of the product.