NM6403 Software Development Kit

NeuroMatrix® NM6403 Assembly Language Overview

Version 1.0

JOINT-STOCK COMPANY
RESEARCH CENTRE

Contents

Figures

Tables

Preface

About This Manual

This manual covers the following topics:

- architecture of the processor and features of its main functional nodes from programmer's point of view;
- description of all registers;
- use of internal memory blocks of Vector execution Unit (VU) and their features;
- description of syntax of assembly language with examples;
- structured list of all scalar and vector instructions of NM6403;

Organization

This manual is organized into the following chapters:

Chapter 1 **NeuroMatrix® NM6403 Architecture Overview**

Gives reference information on the NM6403 structure. Contains description of all main computation nodes of the processor.

Chapter 2 **Assembly Language Syntax Overview**

Gives comprehensive information on assembly program structure and rules showing examples of different syntax structures.

Chapter 3 **Registers**

Gives reference information on different types of registers, describes fields of control registers, vector registers, shows examples of registers use in different processor instructions.

Chapter 4 **Format of Processor Instructions**

Gives reference information on the format of scalar and vector instructions of the processor.

Chapter 5	**Assembly Instruction Set Summary**
	Contains a comprehensive set of assembly instructions, gives their syntax, position of commands and operations in an assembly instruction, shows examples of assembly instructions.

Typographical Conventions

The following typographical conventions are used in this manual:

`Courier`	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.
`Courier`	Shows text that must be substituted with user-supplied information.
Times or Times	Highlights important notes.
//Times	Shows comments to assembler 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 NeuroMatrix® NM6403 Software Development Kit

If you have comments or suggestions about the NeuroMatrix® NM6403 Software Development Kit, please contact your supplier or, giving:

- the platform and release of the NeuroMatrix® NM6403 software tools you are using;
- a small sample code fragment which illustrates your comment;
- precise description of your comment or suggestion.

Mail your remarks to the address: nm-support@module.ru

1 NeuroMatrix NM6403 Architecture Overview

1.1 Introduction

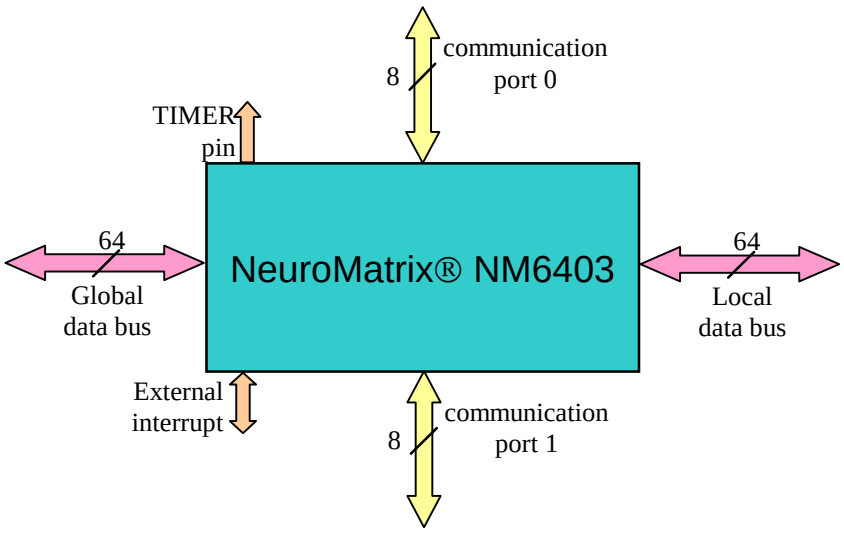
Before describing the NM6403 assembly language it is necessary to introduce some concepts dealing with the processor architecture to be referred to later in this manual.

The processor architecture, illustrated below, reflects a programmer's point of view. Therefore some concepts that cannot be managed from source code are missing.

1.2 External Processor Interface

Processor NM6403 contains four main channels to transmit data to/from peripheral devices. (see Figure 1-1).

Figure 1-1. Scheme of NM6403 External Processor Interface



Global and local data buses are used to access external memory. Memory blocks connected to the global bus are referred to as **global memory**. Memory blocks connected to the local bus are referred to as **local memory**.

In addition to external memory access the processor can send and receive data through two communication ports hardware compatible to TMS320C4x. More detailed description of communication ports and their controls is given in paragraph 3.2.2 on page 3-13.

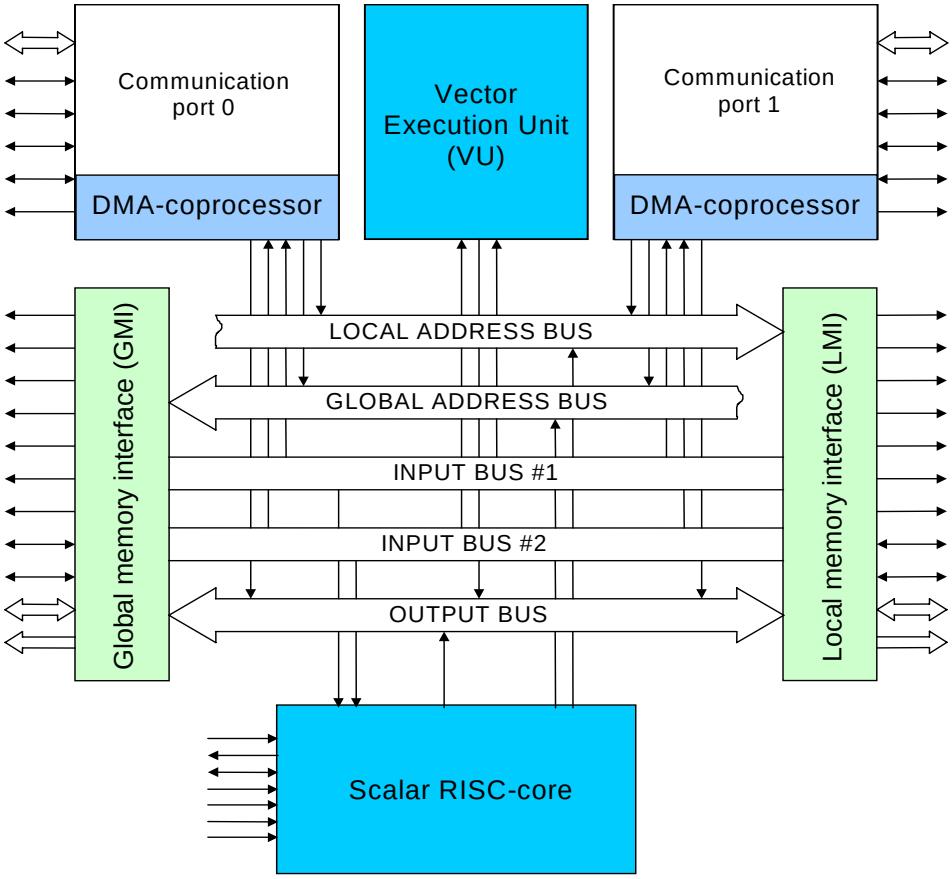
Processor NM6403 does not have internal memory except some special FIFO-like buffers that are used in vector operations. These memory blocks are described later in this manual.

1.3 Common Description of Internal Processor Structure

Processor NM6403 consists of the following internal blocks (Figure 1-2):

- Scalar RISC-core;
- Vector execution Unit (VU) ;
- DMA-coprocessors, managing communication ports (see paragraph 3.2.2 on page 3-13);
- Timers (see paragraph 3.2.7 on page 3-32);
- Global and Local Memory Interface Units managed by control registers to access different types of external memory (see paragraph 3.2.1 on page 3-6 and paragraph 3.2.4 on page 3-23).

Figure 1-2. NeuroMatrix® NM6403 Block Diagram



The Vector execution Unit (VU), which is the main feature of NeuroMatrix® NM6403, operates concurrently with scalar RISC-core and two DMA-coprocessors. This 64-bit engine provides highly parallel operations, allowing simultaneous execution of up to 2048 operations in a single clock cycle. Its architecture gives flexibility of choice in performance/accuracy ratio. Depending on data size, vectors are from one to sixty-four elements long.

The Vector Unit operations are performed on multiple data elements by a single instruction. This is often referred to as SIMD (single instruction, multiple data) parallel processing.

Processor NM6403 has a 64-bit external memory interface. It allows access to one 64-bit word per transaction at each memory bus. Depending on memory access time up to two 64-bit words per cycle can be transmitted.

The instruction set of NM6403 consists of both 32-bit and 64-bit instructions. If an instruction contains a 32-bit constant, it is referred to as a 64-bit instruction. In other cases it is a 32-bit instruction.

The following strings are examples of 32-bit and 64-bit instructions of NM6403:

```
ar0 = 80808080h;      // 64-bit instruction (contains constant value)
gr0 = [ar0];          // 32-bit instruction.
```

More detailed information on the NM6403 instruction set can be found in chapter 5.

1.3.1 Brief Description of RISC-core Components

RISC-core is used for address calculations and for basic operations over general-purpose registers. It is also used to prepare data for the Vector Unit. The NM6403 RISC-core diagram is shown in Figure 1-3.

RISC-core contains a primary register file that contains eight address registers and eight general-purpose registers. In addition to the primary register file there are peripheral control register file and vector register file. The complete information on all NM6403 registers is given in chapter 3.

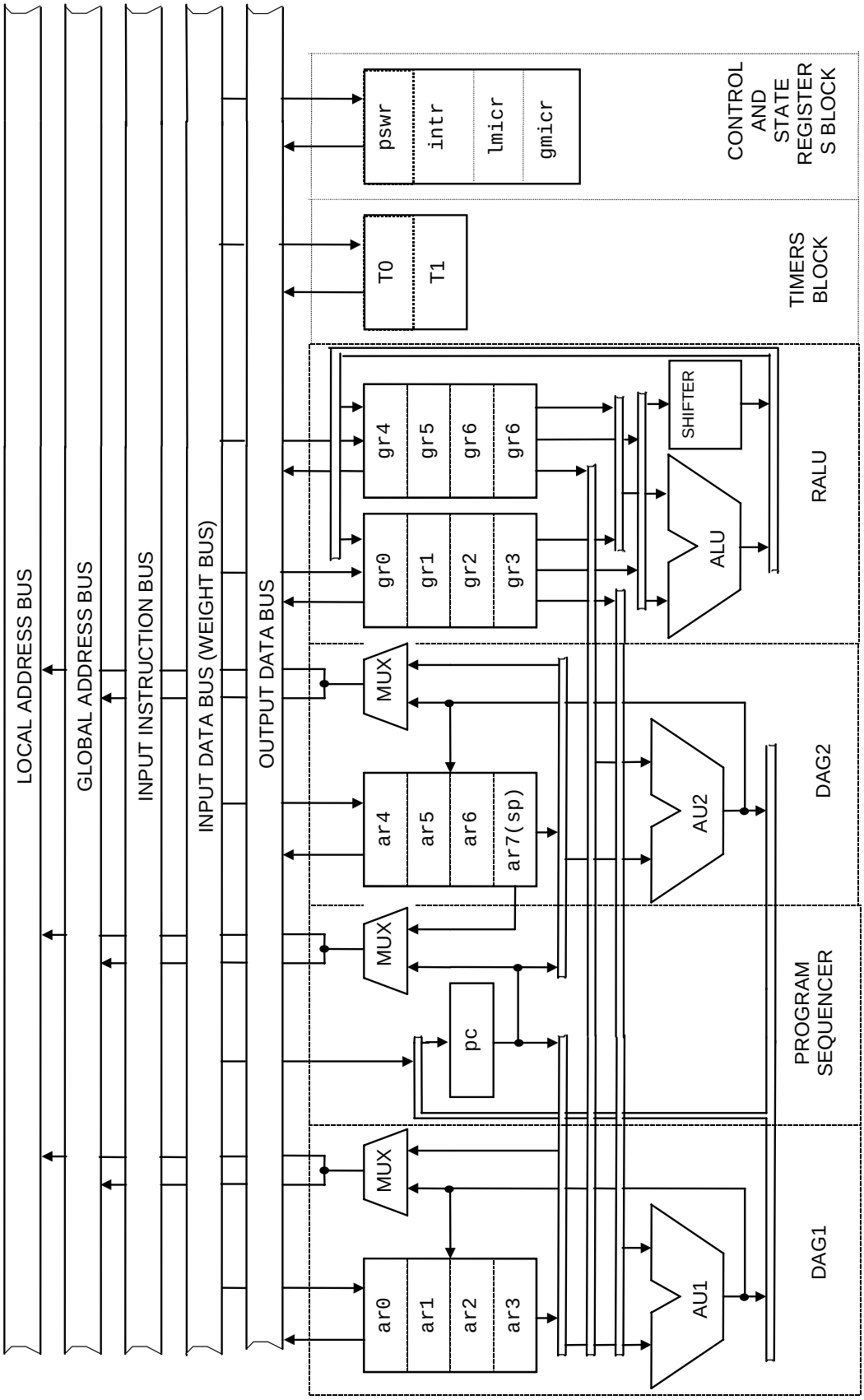
RISC-core supports the following set of operations:

- different types of addressing with or without modification of address registers;
- external memory access to read/write 32-bit and 64-bit words;
- all types of arithmetic and logical operations on general-purpose registers with or without modification of condition flags;
- different types of shift operations at arbitrary bits ranged from 1 to 31;
- conditional/unconditional branches, including delayed branches;
- calls of subroutines with store of return address in stack, including delayed calls;
- multistep multiplication;
- handling of timers;

- selection of external memory access time for different types of memory by handling memory interface control registers;
- handling of the Vector Unit operation nodes configuration.

The complete set of scalar instructions can be found in paragraph 5.1 on page 5-3.

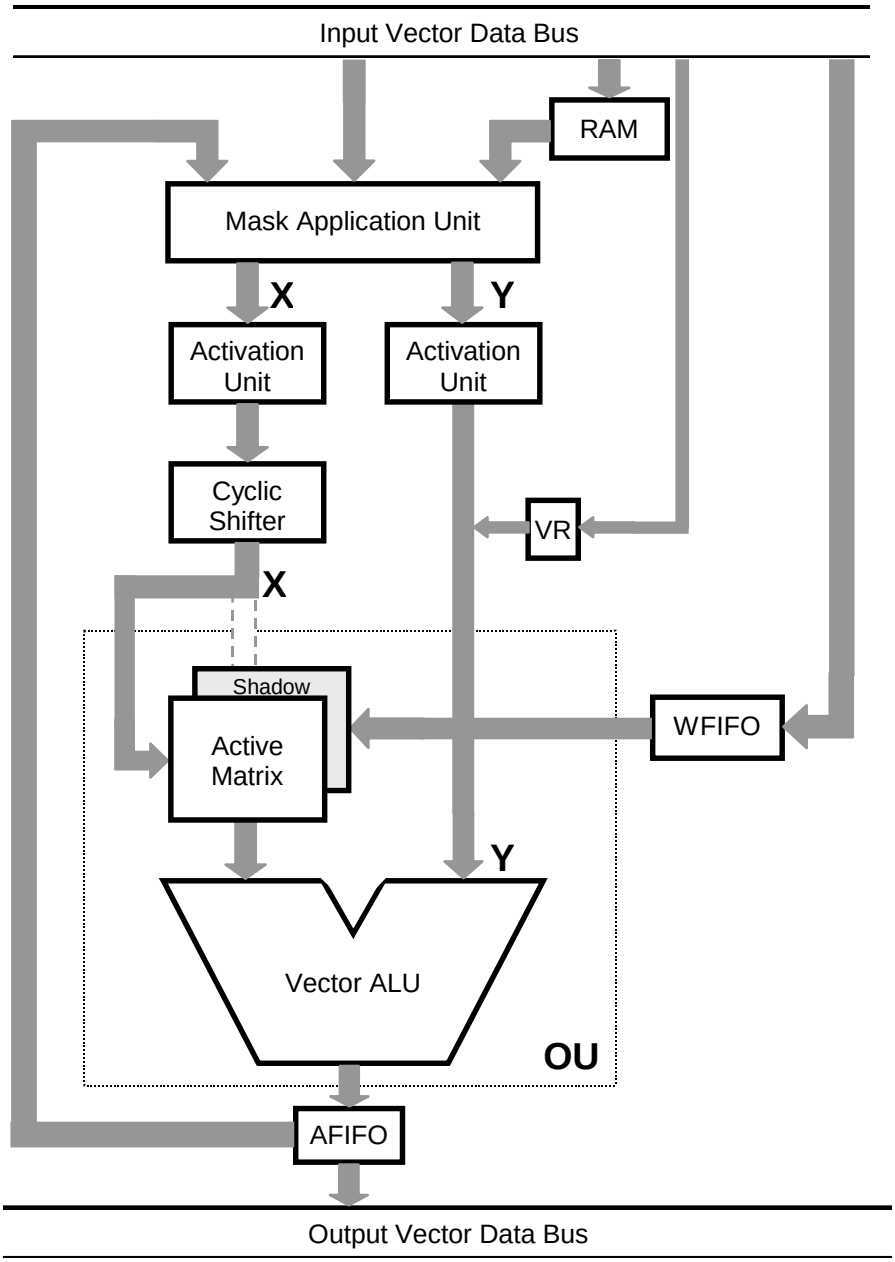
Figure 1-3. NeuroMatrix NM6403 RISC-Core Block Diagram



1.3.2 Brief Description of Vector Unit Architecture

Vector Unit architecture is shown in Figure 1-4.

Figure 1-4. Scheme of NM6403 Vector Unit



Referring to Figure 1-2, the Input Vector Data Bus shown in Figure 1-4 is mapped to the Input Bus #1 and the Input Bus #2, the Output Vector Data Bus is mapped to the Output Bus.

The central node of VU is Operation Unit (OU). It contains the Active/Shadow Matrix block and the Vector ALU.

Operation Unit is used to process multiply adds, arithmetic and logical operations, arbitrary permutation on vectors of packed data. Additional calculation nodes of VU allow the user to perform of mask application

operations, saturation and threshold linear transforms on vectors and matrixes.

The Vector Unit consists of the following components:

- **Active Matrix** – an operation unit that performs multiply add and permute operations. There are two control registers associated with the Active Matrix. They specify its configuration. This means they configure the number of rows and columns that the Active Matrix is divided into. More detailed description of the Active Matrix and its operation can be found in paragraph 1.5.2 on page 1-12;
- **Shadow Matrix** – an operation unit that is used for background load of weights to the Active Matrix. While the Vector Unit performs computations using current weights loaded into the Active Matrix, the new part of weights is loaded into the Shadow Matrix. After new weights are loaded it takes just one cycle to copy them to the Active Matrix. There are two control registers associated with the Shadow Matrix. They specify its configuration. This configuration can differ from the configuration of the Active Matrix;
- **Vector ALU** –an operation unit that performs arithmetic and logical calculations on packed words of data. Calculations are made on all elements of a packed word at the same time. If overflow occurs as a result of arithmetic operation the carry bit will be lost but will not affect the neighbor element in the packed word. This means that there is a “screen” between every two elements of the packed word that doesn’t permit carry bits when overflow occurs. More detailed description of the Vector ALU and its operation can be found in paragraph 1.5.3. on page 1-15.
- **Weights FIFO Buffer (`wfifo`)** – a FIFO-like queue of thirty-two 64-bit words. It is used to store weights that are transferred from external memory to be loaded into the Shadow Matrix. It is treated as a transfer buffer between external memory and the Shadow Matrix. Depending on the Shadow Matrix configuration, more than one suite of weights can be stored. More detailed description of `wfifo` and its features can be found in paragraph 3.3.8 on page 3-58;
- **Internal memory FIFO buffer** (`ram`) – a FIFO-like queue of thirty-two 64-bit words. It is used as one of the input buffers for the Active Matrix or the Vector ALU. This means that data stored in `ram` can be reused in calculations that are made on the Active Matrix or the Vector ALU as many times as necessary. Internal memory FIFO buffer can store up to thirty-two packed words loaded from external memory. The main restriction is that all data stored in `ram` should be used in calculations. More detailed description of `ram` and its features can be found in paragraph 3.3.7 on page 3-56;
- **Pseudo-buffer of data bus** (`data`) – used to manage data coming through the data bus when loading to `ram` or takes part in calculations that are being made on the Active Matrix or the Vector ALU. It allows

the user to redirect data transferring through the data bus from external memory to one of the inputs of the Operation Unit. Pseudo-buffer is treated as the FIFO-like queue of thirty-two 64-bit words. It is used as one of the input buffers for the Active Matrix or the Vector ALU. More detailed description of `data` and its features can be found in paragraph 3.3.6 on page 3-55;

- **Accumulator FIFO buffer** (`afifo`) - a FIFO-like queue of thirty-two 64-bit words. The destination buffer of any operation in the Vector Unit is `afifo`. It is a dual port FIFO. It may be used as one of the input buffers for the Active Matrix or the Vector ALU as well. Before the results of calculations in the Vector Unit become available for the RISC-core they should be stored from `afifo` into external memory. More detailed description of `afifo` and its features can be found in paragraph 3.3.5 on page 3-49;
- **Bias register** (`vr`) – a 64-bit vector register that can be used as input buffer **Y** for calculations made in the Active Matrix. It is treated as a buffer filled with the same packed words. Bias register is available from the RISC-code as a 64-bit read only register. More detailed description of `vr` and its features can be found in paragraph 3.3.4 on page 3-48;
- **Activation units** – units that apply saturation function or threshold function to input vectors. They operate on packed words and transform all elements in parallel. This transformation is also called “activation”, meaning the activation units transform data before they pass to the Active Matrix or the Vector ALU. More detailed description of Activation units and their features can be found in paragraph 1.5.5 on page 1-19;
- **Mask Application Unit** – the unit that applies mask to input vectors of data. Mask Application Unit operates on packed words of data. It has three inputs and two outputs, so it also called “switch 3→2”. More detailed description of Mask Application Unit and its features can be found in paragraph 1.5.4 on page 1-17;
- **Cyclic Shifter** right one bit – a unit that rotates right one bit packed words of data coming through it. The Cyclic Shifter rotates a packed word regardless of the word’s split, so the 0^{th} bit become the 63^{rd} one, the 1^{st} bit become the 0^{th} one and so on. This unit is used to compensate for inability of the Active Matrix to divide input data passing through the input **X** into elements of odd bit length. More detailed description of Cyclic Shifter unit and its features can be found in paragraph 1.5.6 on page 1-20.

1.4 Data Representation in Vector Unit

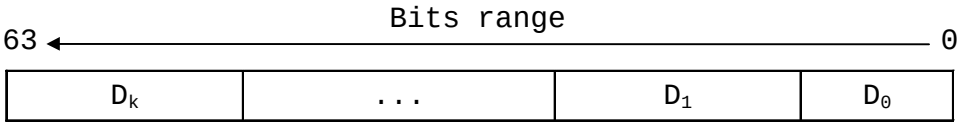
The term “vector” is used to describe 1D-array of uniform data represented as a continuous block in memory. Matrix is an array of vectors.

The Vector Unit processes only 64-bit data, so all its calculation nodes are 64 bits long. It operates only on integer data that are concatenated into 64-bit words (see Figure 1-5). The common representation of a 64-bit word of packed data is represented in the following way:

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

It contains k elements with total bit length equal to 64. Moreover, one packed word **D** can contain data elements of different bit length. The number of elements contained in one packed word depends on their bit lengths and is ranged from 1 to 64.

Figure 1-5 Format of Packed Data Word



1.5 Main Computation Nodes of Vector Unit

This section contains more detailed description of all programmable operation nodes of the Vector Unit and explanation on how the processor performs the following operations:

- multiplication and accumulation, also called “weighted accumulation”;
- arithmetic and logical operations over packed words of data on the Vector ALU;
- mask application to input vectors;
- application of activation functions;
- rotation right one bit of packed words of data coming through the path **X** to the Active Matrix or the Vector ALU.

Moreover, this section contains information on the transform order when data are coming through the different operation units of the Vector Unit activated by a vector instruction.

1.5.1 Data Sources and Paths

Data can be loaded to the processor from local external memory connected to the local data bus and from global external memory connected to the global data bus. There are three internal FIFO buffers that also take part in data processing on the Vector Unit. They are `ram`,

`afifo`, and `wfifo`. To manage data coming directly from the external memory, `data` pseudo-buffer is used (see paragraph 3.3.6 on page 3-55).

Weights FIFO buffer (`wfifo`) is designed to store weights for the Active Matrix, while other buffers (`ram`, `data`, `afifo`) are used as source buffers for data processing. When the processor starts calculations, `ram` and `afifo` are empty. To use `ram` data should first be loaded to it. After that, the data can be used as many times as desired until a new block of data replaces the current one.

Results of any calculations on the Vector Unit are stored into `afifo`. They can be moved to external memory or/and can take part in the next step of calculations. Accumulation buffer is a dual port FIFO; it allows, for example, movement of data to external memory and retrieval of new data from the Vector ALU at the same time.

Data pass to the Operation Unit through two input channels, which are **X** and **Y**. Each of the source buffers (`data`, `ram` or `afifo`) can be connected to one or both data paths.

For example, a vector of packed 64-bit words stored in `ram` can be passed to the Active Matrix along the path **X** or **Y**, or along the **X** and **Y** at the same time.

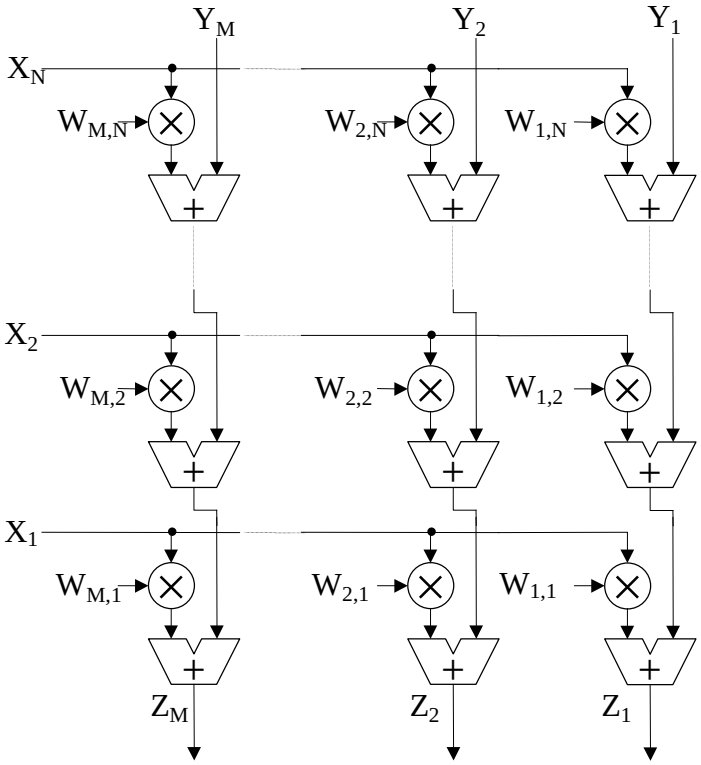
Vector bias register `vr` can also be used as a data source for the input **Y** (see 3.3.4 on page 3-48).

In addition to `data`, `ram` and `afifo` the additional device called “zero” device can also take part in calculations as a data source. If this device is used in a vector instruction, null vectors pass through one of the inputs of the Active Matrix and the Vector ALU.

1.5.2 Weighted Accumulation Procedure

Weighted accumulation procedure is performed on the Active Matrix and the Vector ALU, which are the most significant operation nodes of the NM6403 Vector Unit. This procedure is represented schematically in Figure 1-6.

Figure 1-6. Weighted Accumulation Scheme



The formula of this transform can be represented as follows:

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

where Z_i - an element of an output vector.

X_j - an element of packed word of data coming to the Active Matrix through the input **X**.

Y_i - partial sum, accumulated on previous steps of calculations.

W_{ij} - weight that is loaded into the related cell of the Active Matrix.

M - number of columns of the Active Matrix.

N - number of rows of the Active Matrix.

The Active Matrix has an input that is connected to the path **X** (see Figure 1-7). It is used to load 64-bit words of data from external memory (`data`) or from internal FIFOs (`ram` or/and `afifo`) into the Active Matrix to execute a weighted accumulation procedure (Σ). The results of calculation pass to the Vector ALU to make final addition to the results of previous calculations coming along the path **Y**(Y + Σ).

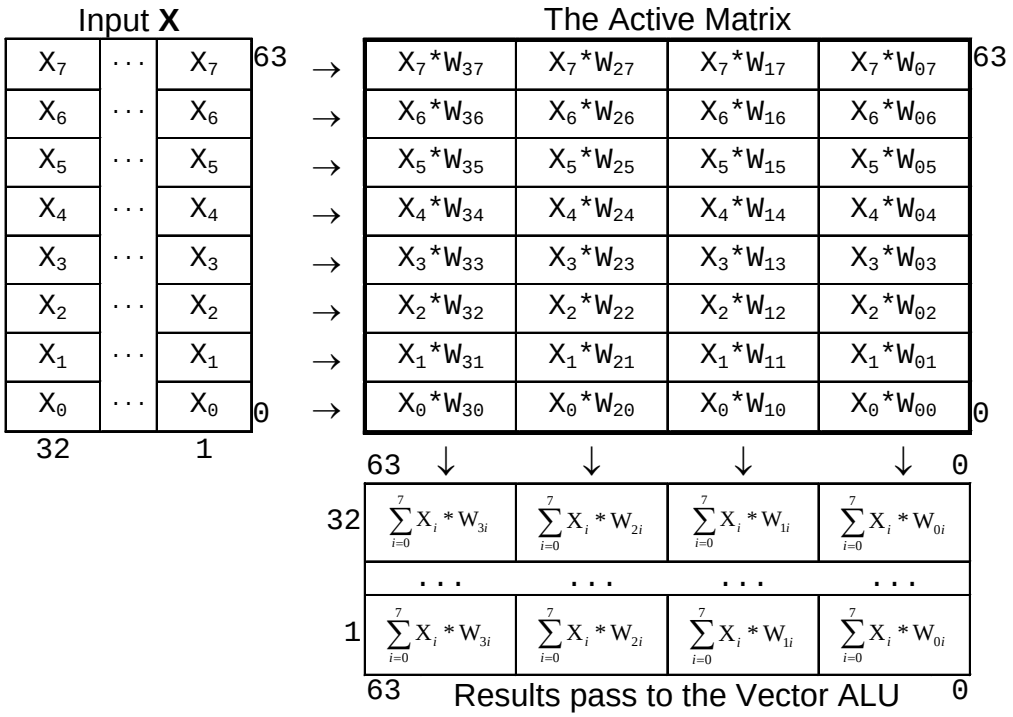
If the Active Matrix is used by a vector instruction the interpretation of data moving along the path **X** differs from that of data moving along the path **Y**.

Every element of the packed words of data passing through the input **X** multiplies by every cell of the related row of the Active Matrix. The results of multiplication are accumulated in every column. The results of weighted accumulation are added to the data moving along the path **Y** in the Vector ALU.

Before starting calculations on the Active Matrix, the Shadow/Active Matrix configuration must be defined and weights should be loaded. The procedure of weights loading is described in paragraph 3.3.8 on page 3-58.

Active Matrix rows number is configured by the `sb2` register. The same register defines the structure of packed 64-bit words passing through the input of the Active Matrix that is connected to the path **X**. The 32-bit constant loaded into `sb2` configures division of input data into elements. More detailed description of the `sb2` register can be found in paragraph 3.3.3 on page 3-44.

Figure 1-7. Weighted Accumulation on the Active Matrix

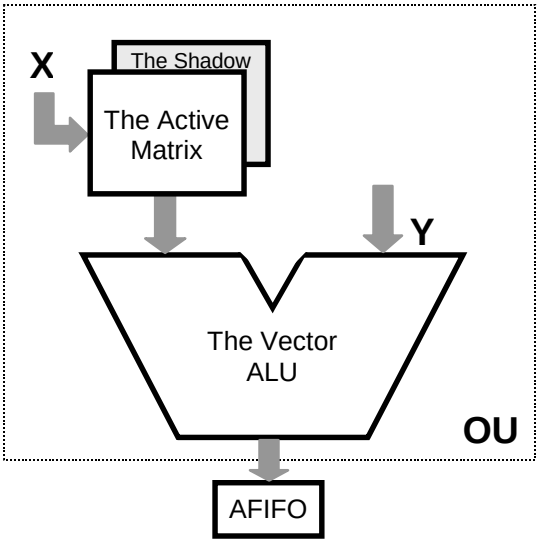


The Active Matrix column number is configured by register `nb2`. The same register defines the structure of packed 64-bit words moving along the path **Y**. The 64-bit constant that is loaded into `nb2` configures bit length of the accumulator elements. More detailed description of the `nb2` register can be found in paragraph 3.3.2 on page 3-40.

Figure 1-7 shows the execution process of one vector instruction. Suppose that thirty-two packed words were preliminary loaded from external memory into `ram`. Another source of data is in external memory. When the weighted accumulation procedure is being performed the data from external memory are being loaded word by word and move along

the path **X** to the Active Matrix. At the same time the data from `ram` passes to the Vector ALU along the path **Y**.

Figure 1-8. Data Path Through the OU in Case of Weighted Accumulation



Every element of the packed word passing through the **X** multiplies by weights stored in cells of the related row of the Active Matrix. The results of multiplication are accumulated in every column. The calculated data are directed to the input **X** the Vector ALU and added to the data moving along the path **Y**. All these operations take one cycle per single packed word. The next cycle the processor performs the same operations over the next pair of input data words and so on up to thirty-two times per vector instruction. The result of these calculations is stored in `afifo`.

When data are stored in the processor's internal FIFOs their split into elements is **undefined**. The split becomes defined when data come along the paths **X** or **Y** of the Operation Unit.

Division into the elements may be different for the same block of data depending on what path, **X** or **Y,** they pass. This statement is correct in case of using the Active Matrix. For example, the Active Matrix is configured in the following manner: there are eight rows and four columns (see Figure 1-7). This means that packed words of data passing through the input **X** are divided into eight elements. So each element has an 8-bit length. If the same block of data passes along the path **Y** it is presented as a set of four 16-bit length elements packed into 64-bit words.

1.5.3 Calculations in the Vector ALU

The Vector ALU is designed to perform arithmetic and logical operations on packed words of data. It has two inputs **X** and **Y**. Source data buffers for the Vector ALU are `ram`, `afifo`, external memory (`data`) and the output of the Active Matrix. Data from all sources except for output of the Active Matrix (see Figure 1-8) can come into the Vector ALU through both inputs **X** and **Y**. Moreover, the "null" device can also be

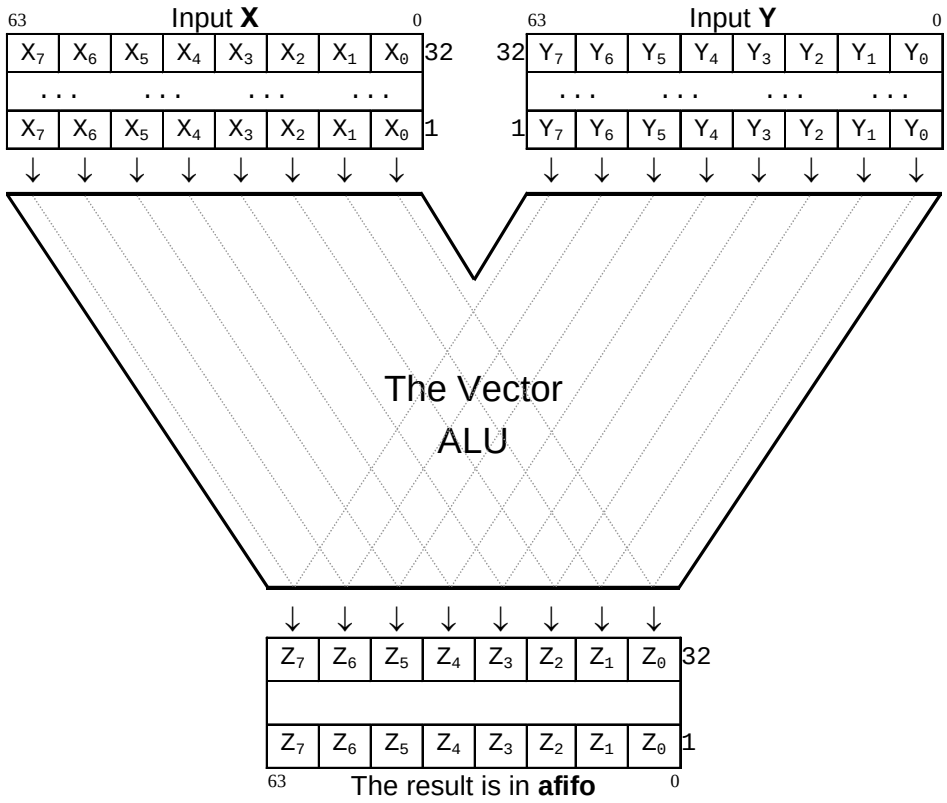
used as a source buffer. The “null” device prevents data from passing through the input.

The Vector ALU allows one more special device as the source buffer for the input **Y**. It is called the “one” device. It generates packed words according to the configuration of the register `nb2`. Thus, every element is equal to 1.

The results of calculations in the Vector ALU are stored in `afifo`.

The inputs **X** and **Y** of the Vector ALU are divided into elements in the same manner. This division is configured by the register `nb2`. The register `sb2` does not affect the division. This is the difference between the **X** configurations of the Active Matrix and of the Vector ALU.

Figure 1-9. Data Processing on Vector ALU



Calculations in the Vector ALU are made over all elements of packed words of input data in parallel (see Figure 1-9). If overflow occurs as a result of arithmetic operation, the carry bit will be lost. This will not affect neighbor elements of the packed word. A “screen” between every two elements of the packed word prevents carry in case of overflow.

Figure 1-10 shows an example of addition of two packed words, divided into eight 8-bit elements.

Figure 1-10. Addition of Two Packed Words on the Vector ALU

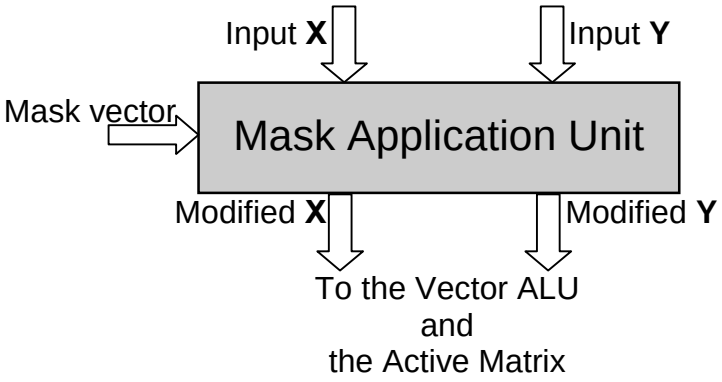
01	80	80	F0	02	FF	FF	01
+							
FE	20	80	1F	02	01	FF	01
=							
FF	A0	00	0F	04	00	FE	02

The cells affected by “screen” protection are shaded. The carry bits of these cells are lost.

1.5.4 Mask Application Procedure

The Vector Unit contains Mask Application Unit to apply masks to input data. This unit has three inputs and two outputs (see Figure 1-11). Data that pass through the inputs **X** and **Y** first go through the Mask Application Unit and then transfer to the Vector ALU and the Active Matrix. If a vector instruction does not include a mask application operation, the data is transferred through the Mask Application Unit without any changes. Otherwise the third input is used to apply mask vector. External memory (`data`), `ram` or `afifo` can be used as the source buffer for the mask vector.

Figure 1-11. Input and Output Data Streams of Mask Application Unit



With other vector operations of NM6403, mask application takes from one to thirty two cycles depending on the number of data words to be transferred. Three words of data come to the Mask Application Unit with each cycle. The first one through the input **X**, the second one through the **Y,** and the third one through the mask input. Two modified packed words come out of the Mask Application Unit.

Data passing through the Mask Application Unit goes to the Active matrix and/or the Vector ALU for further processing.

Mask Application in Conjunction with Weighted Accumulation

There is a vector instruction in NM6403’s instruction set that conjoins mask application operation with weighted accumulation, for example:

Mask **X** **Y**

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

In this case data are processed in the following way:

- bitwise logical AND between the input vector **X** and the mask vector: `X AND MASK`. This operation leaves unchanged only bits of the input vector that are related to the bits of mask equal to one;
- bitwise logical AND between the input vector **Y** and the inverted mask vector: `Y AND NOT MASK`. This operation leaves unchanged only bits of the input vector that are related to the bits of mask equal to zero;
- weighted accumulation of masked data passed through the input **X** (Σ);
- addition of masked vector **Y** to the result of weighted accumulation (Y+Σ).

Mask Application in Conjunction with Logical Operation

There is a vector instruction in NM6403's instruction set that conjoins mask application operation with logical operation in the Vector ALU. For example:

Mask **X** **Y**

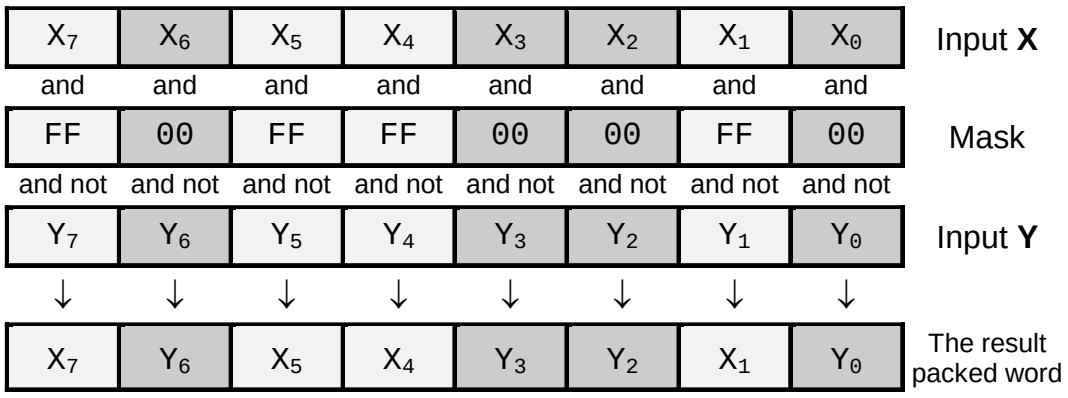
```
rep 32 data = [ar0++] with mask ram, data, afifo;
```

In this case data is processed in the following way:

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

This transformation is also called "logical masking". It contains operations `X AND MASK` and `Y AND NOT MASK` that are processed in the Mask Application Unit. The results of these operations pass through the Vector ALU where bitwise logical OR is performed on them. This data manipulation has simple meaning and is shown in Figure 1-12:

Figure 1-12. Logical Mask Application Procedure



Mask word configures structure of the result one. For the bits of mask equal to 1 the corresponding bits of vector **X** are put to the output vector, for the bits of mask equal to 0, the bits of **Y** are.

Thus, the logical masking allows constructing an output word from the related bits of two input words for one processor cycle.

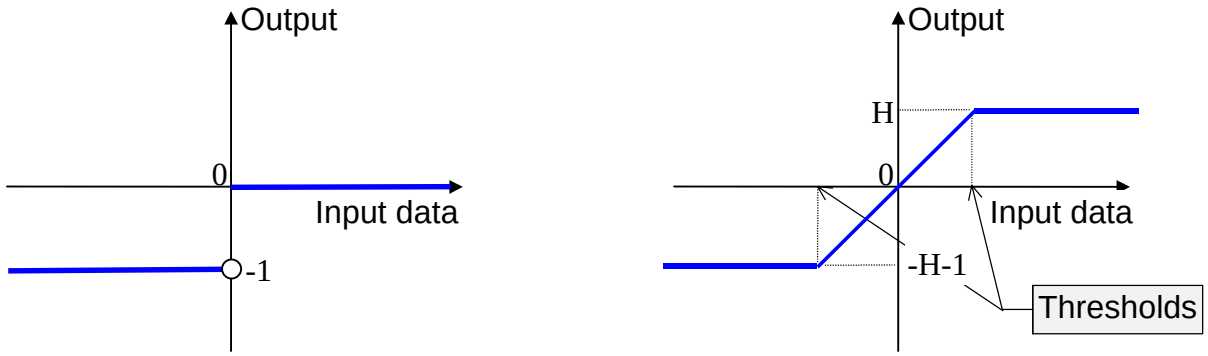
1.5.5 Application of Activation Functions

The Vector Unit contains two activation units to apply saturation and threshold functions to the input data. One of them is connected to the path **X**, another one to **Y** (see Figure 1-4).

There are two types of activation functions:

- Threshold function (see Figure 1-13a);
- Saturation function (see Figure 1-13b).

Figure 1-13. Types of Hardware Implemented Activation Functions



a) Threshold function *b) Saturation function*

Activation units make calculations over packed words of data. Activation units allow applying activation functions to all elements of a packed word at the same time. Special registers: `f1cr` and `f2cr` are used to configure activation units. More detailed information about these registers and the rules for applying them can be found in paragraph 3.3.1 on page 3-34.

Activation units are located between the mask application unit and the Active Matrix or the Vector ALU. Activation functions can be applied to data passing through the both paths **X** and/or **Y**.

The type of activation function applied depends on vector instruction. There are two types of vector instructions: arithmetic and logical. The terms “arithmetic vector instruction” and “logical vector instruction” are described below in this paragraph.

The threshold function can be applied to packed words of data only if they are processed by logical vector instruction. The saturation function can be used only in pair with arithmetic vector instructions. In this way the threshold function is also called “logical activation”, while the saturation function is referred to as “arithmetic activation”.

Arithmetic Activation Function

The saturation function can be used in conjunction with arithmetic vector instructions. The term “arithmetic vector instruction” covers the following list of vector instructions:

- All types of instructions containing the weighted accumulation operation. For example:

  path **X** path **Y**

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

  The reserved word “activate” means that a saturation function is applied to data coming into the Active Matrix through the input **X**;

- All types of instructions containing arithmetic operations. For example:

  path **X** path **Y**

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

  In this case the term “activate” means a saturation function application to data coming into the Vector ALU through the input **Y**.

All vector instructions mentioned above have the same feature. All of them contain arithmetic operations. So, the saturation function is called “arithmetic activation” because it is used together with arithmetic instructions.

Logical Activation Function

The threshold function in the contrary to the saturation function is used only in conjunction with logical vector instructions. The term “logical vector instructions” is associated with the set of vector instructions that contain only logical operations performed in the Vector ALU. For example:

path **X** path **Y**

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

In this case term “activate” means that a threshold function is applied to data coming into the Vector ALU through the input **Y**.

The threshold function is used together with logical vector instructions so it is also called as “logical activation”.

More detailed information about handling of activation units is given in paragraph 3.3.1 on page 3-34.

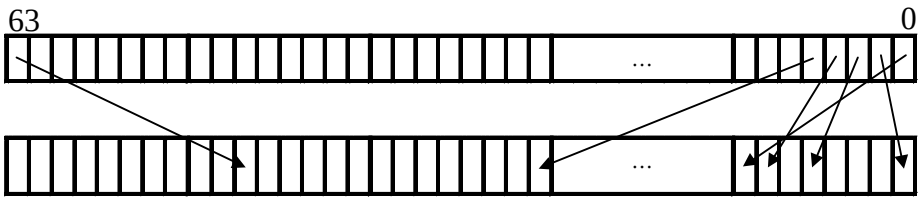
1.5.6 Cyclic Shift Right One Bit

Cyclic shift right one bit can be applied only to data coming through the path **X** (see Figure 1-4). This operation is used to get additional flexibility to binary data processing. The Active Matrix does not support

division of input data (coming in through the input **X**) into odd-bit length elements.

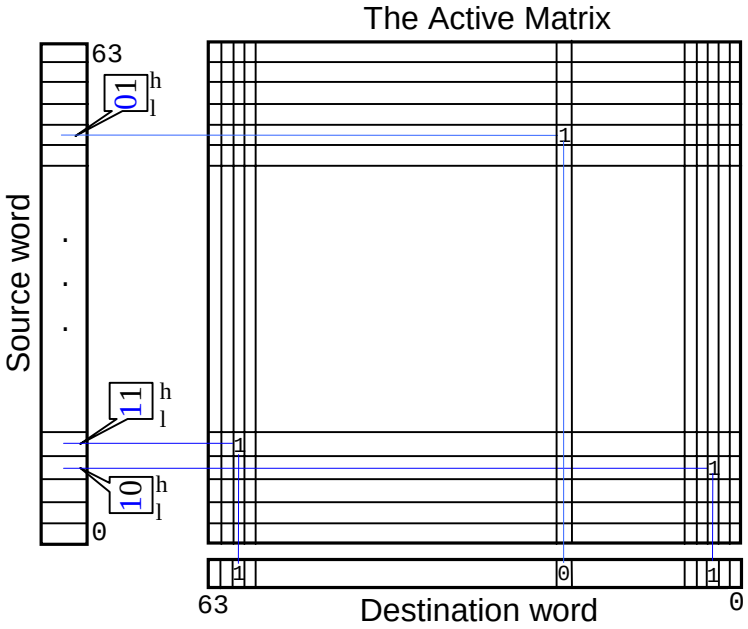
The minimum possible bit length of the elements is two bits. So it is necessary to have an additional calculation unit to process binary data. This one is cyclic shifter. It shifts packed words of input data one bit right. No matter how these words are split up into the elements, the cyclic shifter shifts the whole word. In this case high bits of 2-bit elements become low ones and can now take part in calculations. The low bits become the high ones in the right neighbor element, but they can be masked in the same vector instruction. Let's take a look at the following example:

We need to make complete permutation inside a 64-bit word.



The Active Matrix is configured to thirty-two rows and sixty-four columns. Filling the particular cells of the Active Matrix with 0 or 1 we can make permutation of every low bit of 2-bit elements of the input word to the desired position in the output word.

Figure 1-14. Bitwise Permutation on the Active Matrix



To permute high bits of elements of the source word we have to shift them one right. In this case the high bits become the low ones and can be processed in the same manner as shown in Figure 1-14.

The cyclic shift can be applied to source data in vector instructions together with weighted accumulation or mask application. To activate

cyclic shifter node the key word “shift” is used in a vector instruction. For example,

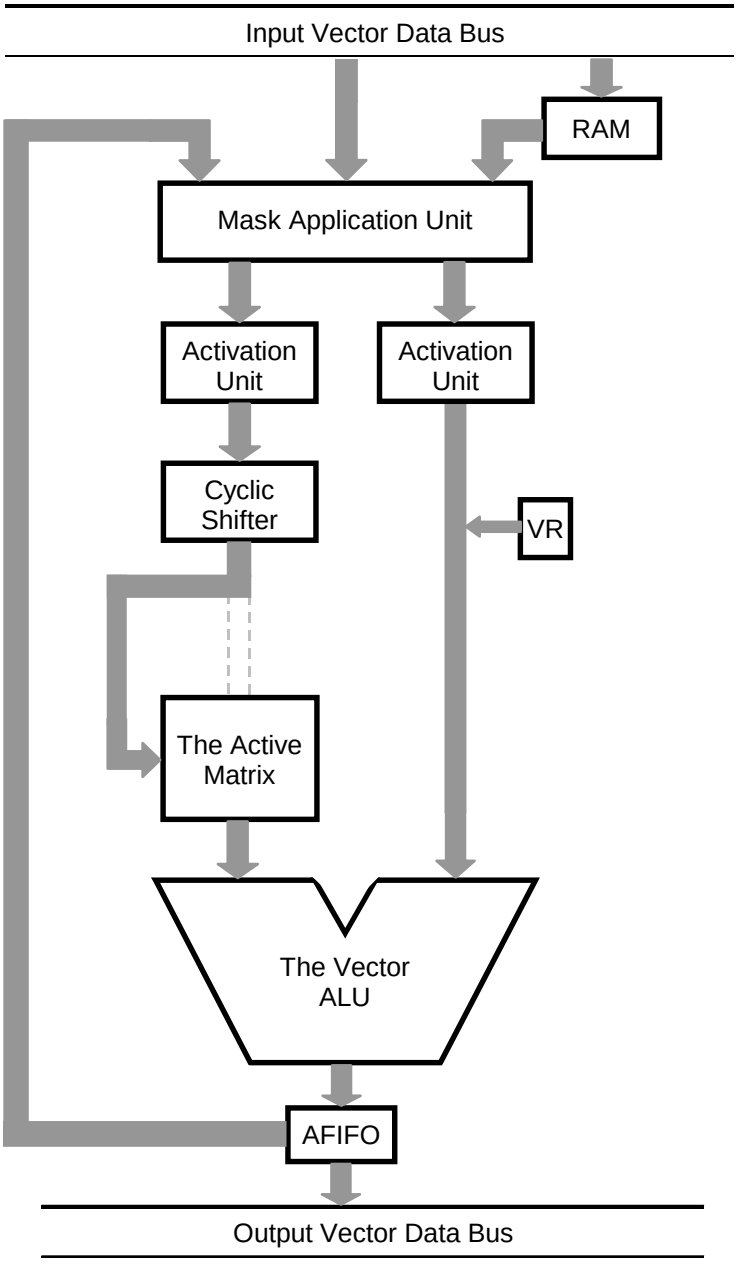
path **X** path **Y**

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

1.5.7 Data Processing Order in Vector Unit

There are six computation nodes in the Vector Unit. Many of them can be involved in calculations by one vector instruction, if it happens the calculations are made according to a particular processing order. This order is shown in Figure 1-15.

Figure 1-15. Data Processing Order in the Vector Unit



Every computation node can modify data or transfer them without modification. The behavior of the nodes is defined by a vector instruction. The special key words are used in the instruction to activate the particular nodes. For instance, “shift” is used to activate cyclic shifter, “mask” is used to activate mask application, and so on. It is possible to activate up to six nodes in one instruction, however, data will be processed in the order as shown in Figure 1-15.

2 Assembly Language Syntax Overview

Programs written in assembly language for NM6403 contain different syntax like assembly directives, instructions, macros, pseudo-instructions and comments. The following lines demonstrate some examples of correct syntax of NM6403 assembly language:

```
MySym: word = 80h;          // Variable definition.
<L1> ar0 = ar2 + gr2;       // Address register modification.
rep 32 [ar0++] = afifo;     // Vector instruction.
```

Instruction syntax does not contain any firm requirements to the line structure, i.e. there are no predefined positions where particular fields of instruction or directive should be located.

Each assembler instruction should end with a symbol ‘;’. If assembler cannot find the instruction-terminating symbol in the current line it considers the instruction to continue to the next line.

General form of NM6403 assembler instruction syntax looks as follows:

```
[<label>] assembler instruction; [//comments]
```

The comments must be single-line.

Labels and comments can be omitted.

2.1 Reserved Words

The tables below contain separately the main group of reserved words and reserved words used to store debug information.

Table 2-1. Set of Reserved Words of NM6403 Assembler

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

Table 2-2. Set of Debug Information Reserved Words

debug_arange	debug_die	debug_die_child
debug_die_endchild	debug_end_sequence	debug_frame_cie
debug_frame_fde	debug_line	debug_macro_def

debug_macro_end_file	debug_macro_start_file	debug_macro_undef
debug_pubname	debug_root_die	debug_source_directory
debug_source_file	debug_start_sequence	

Note

Assembler distinguishes upper- and lowercase symbols. That's why all reserved words should be written by lowercase letters, otherwise the word will be regarded as an identifier.

All processor registers are also reserved words. They are **always** written by lowercase letters. For example, the record '`gr0`' denotes a processor register but '`Gr0`' or '`GR0`' are identifiers.

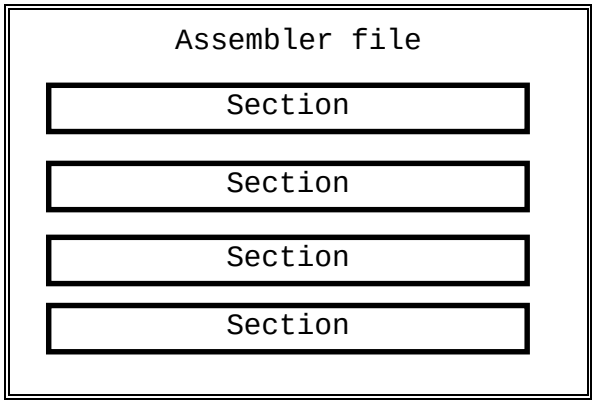
Table 2-3. Set of Registers of NeuroMatrix NM6403

DESIGNATION	PURPOSE	BIT LENGTH
grj	General purpose register j (j=0,…,7)	32
arj	Address register j (j=0,…,7)	32
pc	Program counter	32
pswr	Processor state word register	32
intr	Register of query for interrupt and direct memory access	32
fjcr(h,l)	Activation function j (j=1,2) control register	64(32+32)
vr(h,l)	Bias register	64(32+32)
nb1(h,l)	Neuron boundary register	64(32+32)
sb(h,l)	Synapse boundary register	64(32+32)
gmicr	Register of interface control from global bus	32
lmicr	Register of interface control from local bus	32
ocaj	Output channel j (j=0,1)address register	32
icaj	Input channel j (j=0,1)address register	32
occj	Output channel j (j=0,1)counter	32
iccj	Input channel j (j=0,1)counter	32
dorj	Output channel j (j=0,1)data register	64
dirj	Input channel j (j=0,1)data register	64
tj	Timer j (j=0,1) control register	32

2.2 Assembler File Structure

An assembler file has a definite structure given in Figure 2-1:

Figure 2-1. Assembler File Structure



An assembler file can be conditionally divided into a space of sections and a space between the sections.

2.3 Sections

There are three types of sections in assembly language:

- Section of code;
- Section of initialized data;
- Section of non-initialized data.

Sections of all types in NM6403 assembly language submit to identical design rules. These rules are enumerated below.

Rules of Section Design

A section starts with one of the following reserved word: `begin`, `data` or `nobits` which is the opening bracket and ends with the word `end` (closing bracket). Assembler defines the section type according to the opening bracket. Information about correspondence of the opening bracket reserved words to the section types will be given later in this paragraph. The section name should go right after the opening /closing bracket in the same line. For example a code section `mycest` is to be designed in the following way:

```
begin “mysect” // beginning of the section  mysect

        section body

end “mysect”;   //end of the section mysect
```

There is no comma after the opening bracket and **there should be a comma** after the closing bracket. The section names by the opening and the closing brackets should coincide.

A section name may be of any length within 255 characters. It is a set of capital and small letters, numbers and the symbols:

```
. , / \ : ; " ’ ` [ ] - + = & ! < > ( ) * { } _.
```

Section name **must be** bracketed with double quotes. This name goes to the object file “as is”.

2.3.1 Code Section

This type of sections contains sequence of instructions defining the order of the program execution.

Code section starts with the opening bracket **`begin`**.

For convenience of the further work with code sections it is recommended to form the section name adding the prefix ‘`text`’ to the desired name, for example: `begin “textmycode”`. This prefix is used by an object and executable files decoder (`dump.exe`) to automatically decode this section as a code section.

Not only instructions, but also identifier definitions can be placed to code sections. If the identifier is initialized, i.e. it has an initial value, it is located in the section at the appropriate address. If the identifier is not initialized the place in the section where it is defined is filled with a zero value.

For example a code section may be designed in the following way:

```
begin ".textmycode"
```

// identifiers assignment block.

```
    A: long = 0123456789ABCDEFhl;
    B: word;
```

// processor instructions block.

```
<Label>
    ar0 = A;
    ar2,gr2 = [ar0];
end ".textmycode";
```

The possibility of identifier definitions in the code section is introduced rather for convenience than as a constantly used trick. There are particular types of sections for definition of initialized and non-initialized variables. It is more convenient to work with those types of sections because their position in the processor memory can be managed.

2.3.2 Initialized Data Section

Initialized data sections contain variables definitions combined with their initialization.

An initialized data section begins with the opening bracket **`data`**.

Here is an example of an initialized data section:

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

Assembler **permits** to define non-initialized variables in the initialized data section. For instance the following construction will not cause any error:

```
data ".my_init_data"
    Val1: long = 12;
    Arr:  long[4]; //non-initialized variable.
    Val2: word = 0A5h;
end ".my_init_data";
```

However compiler distinguishes initialized and non-initialized data during compilation. An additional section is created for non-initialized variables where all detected variables are placed. There is a selection rule of the name of non-initialized data section generated by the compiler. Prefix '`.bss`' is added to the existing initialized data section name, for example, for the section «`.my_init_data`» an additional section «`.bss.my_init_data`» is being created and all non-initialized variables are put to it.

Thus if non-initialized variables appear in the initialized data section, assembler creates an additional section of non-initialized data where it puts all found non-initialized variables. It is equivalent to the following construction:

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


nobits ".bss.my_init_data"
    Arr:  long[4]; // non-initialized variable.
end ".bss.my_init_data";
```

If the assembler while processing an initialized data section meets a partly initialized structure or an array it doesn't separate them, but fills non-initialized fields with nulls. If all fields are not initialized, this structure goes to the related section of non-initialized data. In order to leave the structure in its place it is enough to initialize at least one of its fields.

2.3.3 Non-Initialized Data Section

Non-initialized data sections contain only definitions of variables used in the program without their initialization.

Non-initialized data section begins with the opening bracket **`nobits`**.

Here is an example of a non-initialized data section:

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

If an initialized variable appears in the non-initialized data section, assembler ignores its initialization regarding the variable as non-initialized.

2.3.4 Space Between Sections

Space between sections in the assembler file can also be used for location of the following syntactical constructions:

definition of constants and constant expressions including pseudo-functions;

declaration of all possible types of labels;

declaration of variables like `common` and `extern`;

description of structural types of data;

2.4 Constants

This Section describes use of constants and constant expressions in assembler and format of their representation.

There are 32-bit and 64-bit length constants. By default, a constant is considered as 32-bit (short constant). To record 64-bit integer constants, further referred to as long constants, the sign **l**(lowcase 'L') is used and it is added to the end. For example the constant `A = 0x123h` is 32-bit and `B=123hl` is 64-bit.

There are floating point (32-bit) and double (64-bit) point constants. Special compilation time pseudo-functions `float`() and `double`() are used to record these constants in assembler. More detailed information can be found later in this Section.

2.4.1 Constants Representation Formats

Assembler supports six formats of constants:

- Binary integer constant;

- Octal integer constant;
- Decimal integer constant;
- Hexadecimal integer constant;
- Floating-point constant;
- Symbol constant.

2.4.1.1 Binary Integer Constant

A binary integer constant is a line of zeros and ones with the length of up to 64 characters. A binary constant is terminated with a symbol **b**. If the number of characters in a binary constant line is less than 32 or 64, the non used bits (high-order word part) are automatically filled with zero by the assembler. Here are some examples of correctly recorded binary constants:

```
00000000b       //      Constant equal to zero.
001bl           //      Long constant equal to  1.
10000b          //      Constant equal to  16(10).
10101010b       //      Constant equal to 170(10) or  AA(16).
```

2.4.1.2 Octal Integer Constant

An octal integer constant is a line containing numerals from 0 to 7 terminated with a symbol **o**. Here are some examples of correctly recorded octal constants:

```
000o            //      Constant equal to zero.
001o            //      Constant equal to 1.
20ol            //      Long constant equal to 16(10).
252o            //      Constant equal to 170(10) orAA(16).
```

2.4.1.3 Decimal Integer Constant

A decimal integer constant is a line containing numerals from 0 to 9. The decimal short integer is ranged from -2.147.483.648 to 2.147.483.647 for signed and from 0 to 4.294.967.295 for unsigned. The range of long integer is from -9.223.372.036.854.775.808 to 9.223.372.036.854.775.807 for signed and from 0 to 18.446.744.073.709.551.615 for unsigned. Here are some examples of correctly recorded decimal constants:

```
7000            //      Constant equal to 7000(10).
-1              //      Constant equal to -1.
-20l            //      Long constant equal to -20.
170             //      Constant equal to 170(10) or AA(16).
```

The sign «-» can be used to designate negative numbers only in the decimal format.

2.4.1.4 Hexadecimal Integer Constant

A hexadecimal integer constant is a line containing numbers from 0 to 9 and letters `A`, `B`, `C`, `D`, `E` and `F`. Hexadecimal integer constants are terminated with a symbol **h**. The record of a constant should begin with a numeral. If the first significant element is a letter, a zero should be added as a prefix. The symbol «`0`» at the beginning of a hexadecimal constant is necessary to distinguish the constant from an identifier. Here are some examples of correctly recorded hexadecimal constants:

```
000h        //   Constant equal to zero.
01h         //   Constant equal to 1.
20hl        //   Long constant equal to 32(10).
0FFFFFFFFh  //   Constant equal to -1.
```

2.4.1.5 Floating-point Constant

Operations over this type of constants are not hardware supported. Special pseudo-functions are used in assembler to record floating-point constants. For a 32-bit floating-point value the pseudo-function `float()`is used. The pseudo-function `double()`is used for a 64-bit floating-point value. Within the pseudo-function brackets a real number in the usual form can be used, for example:

```
float(123.456)      // Floating-point number (32 bits).
double(-1.02E-3)    // Negative number.
```

Real number format:

`[+|-]num[.numE][+|-]num,` *where* `num` – *decimal numbers.*

Or

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

To emulate floating-point operations on NM6403 the special system library is used. All the operations are made over floating- and double-point values according to IEEE 754 format.

2.4.1.6 String Constant

A string constant is an arbitrary (may be empty) set of ASCII characters taken into double or single quote. If it is necessary to use the character of quote itself, it should be doubled.

Examples of string constants:

"String constant",

'Packed string constant'

A string constant occupies different number of words depending on what commas it is surrounded with. A string in double quote is located in memory according to the convention adopted in C++ – a minimum addressed memory element for each symbol. In case of the NM6403 it is a 32-bit word. Thus, arbitrary access to the string characters is possible.

In case of use of a single quote the characters are «packed» in fours; so one 32-bit word contains four chars. A packed string constant occupies four times less memory then the unpacked one.

A string constant does not contain any other information (a special ending symbol or a string length value) except for those explicitly indicated inside the quotes.

2.4.2 Constant Expression

Besides use of constants in the NM6403 assembly language, it is possible to use constant expressions, which can be calculated at the stage of the program compilation. That's why only the result of the calculations goes to the object code.

Constant expressions are divided into numerical and address expressions. In the first case, the result of the constant expression calculation is treated as a common numerical constant, in the second case - as an address in the memory.

2.4.2.1 Numerical Constant Expression

Numerical constant expression is calculated according to the rules traditional for the expressions. There is a set of common arithmetical operations, bitwise logical operations, multiplication/division, shift and so on. The operations priorities coincide with those in C++. Round brackets are used for partial change in the order of operations execution.

Table 2-4. Summary of Numerical Constant Expression Operations

OPERATIONS	DESCRIPTION
`not`	Bitwise NOT
`-`	Unary minus
`*` `/`	Multiplication Division
`+` `-`	Summation (plus) Subtraction (minus)
`<<` `>>`	Left shift Right shift
`<` `<=`	Less than Less than-equal

`>` `>=`	Greater than Greater than-equal
`==` `!=`	Equal Not equal
`and`	Bitwise AND
`xor`	Bitwise exclusive OR
`or`	Bitwise OR

An example of a numerical constant expression:

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

2.4.2.2 Address Constant Expression

An address constant expression is a combination of numerical constants and addresses of labels and variables. The result of calculation of an address constant expression is a memory address. There are some limitations on an address constant expression. In the address arithmetic only operations of summation and subtraction are allowed for constant expression with the following notes:

the result of summation with a number - is address; summation of two addresses is forbidden;

difference of two addresses from the same section is a numerical constant; it is forbidden to subtract addresses from different sections and the address from a number.

If the indicated conditions for an address constant expression are not met, the assembler generates an error.

Usage of names of registers or indirect memory access is not allowed in constant expressions.

Here is an example of a constant expression:

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

2.4.3 Definition and Use of Constants

In assembler the reserved word `const` is used to define a constant or a constant expression, for instance:

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

The right part of this syntax construction is a numerical constant or a constant expression. The left part is its symbol equivalent that can be further used for initialization of variables and registers.

The lifetime of a constant is the compilation time. The assembler doesn't reserve memory to store a constant. However, if some variable is initialized by a constant, the constant is stored to the variable address.

The constant definition can be located anywhere in the file inside sections as well as outside them. In order to show that the constant does not depend on any section it is a good style to put it to the space between sections.

A constant should be defined before it appears in the assembler instruction, for example:

```
const C = 12; // Constant definition.
...
begin "text"
     ...
     ar0 = C ;   // Its first appearance;
     ...
end "text";
```

Symbol constants are a full equivalent to numerals; they can take part in constant expressions and being used for initialization of variables and registers.

Constants are accessible only inside the file they are defined in.

2.5 Label

Any instruction in a code section can be marked with a label (may be with more than one). Labels are used to execute different kind of branch instructions like function call or conditional jump. A label marks the particular address in memory. It is always associated with an instruction.

A label is presented in the form of a string beginning with a character or the symbol «_». A label consists of Latin characters (both lower- and uppercase), numerals and the symbol «_». In the examples below the correct label names are shown:

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

2.5.1 Label Declaration

Before using a label it should be declared, for instance:

```
MyLabel: label;
```

Label declaration can be located anywhere in an assembler file inside sections as well as outside them. Declaration is simply a message to the compiler that the label should appear in this file.

2.5.2 Label Definition

The programmer makes label definition when a particular assembler instruction is marked with that label. Here is an example of label definition:

```
    gr0 = 123;
<loop>      // point of the label definition.
    [ar0++] = gr0;
     ...
    goto loop;
```

Syntax of label definition is the label surrounded by French quotes, for example `<loop>`. The label marks the instruction that is located right after the label definition. In the example above the instruction `[ar0++] = gr0;` is marked with the label `loop`. Label definition means the label points to the memory address of the assembler instructions it is associated with.

2.5.3 References to a Label

Besides label declaration and definition, references to a label can be met in the program. There can be several references to one label. References are used mainly in branch instructions. In the examples below some references to a label are shown:

```
goto loop;   // reference to the label  loop.
call Func;   // function call, Func - label of its starting point.
gr3 = Func;  // copy an address of the label Func to the address register.
```

A reference to a label contains only its name as it has been declared, without any additional brackets.

Here is an example of a label use:

```
L: label;   // label declaration.
...
begin text
...
<L>             // label definition.
    gr0 = gr2 or gr3;
    ...
    goto L;   // references to the label.
...
end text;
```

2.5.4 Types of Binding and Label Definition Area

Four different types of label binding are supported in assembly language for NM6403:

`local` – labels with local binding;

`global` – labels with global binding;

`extern` – labels with extern global binding;

`weak` – labels with weak global binding.

A reserved word defining the label binding type is located before the label name, for instance:

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

The label binding type defines its definition area.

The simplest type is the **`local`** binding. The definition area of labels of this type is limited by the file where they are defined.

Within the file a `local` label can be declared only once, it can be defined only once, i.e. associated with a particular address in the program. At the same time an arbitrary number of references to the label can be found within the file. References to a `local` label from another file are prohibited.

The `local` labels with the same name can be used in different files because each of them has the own definition area and the definition areas of those labels are not crossed.

It is possible to omit the word `local` in the local label declaration. For example the `local` label can be declared in the following way:

```
MyFunc: label;
```

Moreover it is possible to omit the `local` labels declaration. If the assembler comes across a label definition without preliminary declaration it will consider this label as a `local` one.

Declaration of `local` labels is mainly used to make assembler programs easier to read and more documented.

The labels defined with the **`global`** binding are accessible outside the file where they were defined. It is possible to refer to them from every file of the program. The `global` labels should be used under the following conditions:

two global labels with the same name cannot be used in the program;

a `global` label should be defined in the same file where it has been declared.

All other files of the program can contain references to a `global` label, i.e. to the address, which it was associated with. In order to refer to a label defined in another file it should be declared as external (`extern`).

The **`extern`** type is used when a current file contains at least one reference to a `global` label defined in another file. Only after the `extern` label is declared it becomes accessible to the current file.

Thus a `global` label has the entire program definition area. The `global` label can be defined only once. In the file where it is defined it should be declared as `global`. In all other files of the program it is declared as `extern` if it is necessary to refer to it.

Example:

The file F1.ASM:

```
global MyFunc: label; // global label declaration
...          // the label will be defined in the file.

begin ".text"
...
<MyFunc>              // label definition.
    push ar0, gr0;
    ...
    return;
    ...
    call MyFunc;    // reference to the label.
end ".text";
```

The file F2.ASM:

```
extern MyFunc: label; // external label declaration
...                   // the label is defined in the another file.

begin ".text"
    ...
    call MyFunc; // reference to the external label.
end ".text";
```

There is one more type of labels with global binding. It is **`weak`** binding. The definition area of a `weak` label is the entire program too. The term "`weak`" means that the label has less priority then the `global` label with the same name defined somewhere else in the program.

If there are a `weak` label and a `global` label with the same name in the program the linker ignores the `weak` label and all references are tuned to

the memory address of the `global` label definition. If there is no `global` label with the same name like the `weak` label has, the linker considers the `weak` label as the global one.

The `weak` labels should be used under the following conditions:

two labels with the same name are not allowed in one file;

a label should be defined in the same file where it has been declared;

it is possible to use two and more labels with the same name defined in different files of the program. In this case the linker selects the first label definition address it meets processing object files and ignores others. The linker solves all references to the selected label. The object files order in the linker command line defines which label is selected:

- if there is a `global` label in the program, all `weak` labels with the same name are ignored;
- if there is no `global` label with the same name, the `weak` label becomes global, i.e. it is referred to from other files by declaring it external.

Here is an example of usage of labels with the `weak` binding:

The file `F1.asm`*, where AB is a weak label:*

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

The file `F2.asm`*, where AB is a global label:*

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

The file `F3.asm` *- AB is an external label:*

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

If files `F1.ELF` and `F3.ELF` are gathered in one program by the linker, the `AB` function will be called from the file `F1.ELF`, so the operation `gr0 = gr1 + gr2` will be processed.

If files `F1.ELF`, `F2.ELF` and `F3.ELF` are gathered together, the `AB` function will be called from the file `F2.ELF`, and the operation `gr0 = gr1 - gr2` will be calculated.

Thus the `weak` label is considered global if there is no `global` label with the same name. In case the global label is defined the `weak` one is ignored.

2.6 Variables

A variable in the NM6403 assembly language is an address in the processor memory where a data element is stored. The variable is used to refer to a particular memory cell during calculations.

A variable has a string name which begins with a Latin letter or the symbol «_» and can contain Latin characters (both lower- and uppercase), numerals and the symbol «_». Here are some examples of correct variable names:

`Array_0,`

`__Long_Value,`

`Z987654321A.`

Each variable in the assembly language is associated with the address of a particular memory cell. Variables are accessible for read/write operations.

There are two ways of using variables in the NM6403 assembly language. The first way is to obtain the variable address; the second one is to get the contents of the memory cell the variable points to.

2.6.1 Obtaining the Variable Address

In order to obtain the variable address its name is used. Some examples of correct records of variable addresses obtaining are shown below:

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

2.6.2 Obtaining the Variable Value

In order to obtain the variable value it is necessary to take its name into square brackets. Let's give some examples of correct records of variables **values** obtaining:

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

There are a few types of variables, which are divided into two groups: simple and compound variables.

2.6.3 Fundamental Types

The fundamental types are the minimum hardware supported types considered as a single unit.

As the processor NM6403 supports two base formats – the 32-bit and the 64-bit word there are two data types in assembler, which are called `word` and `long`. In programs they are denoted with syntax words `'word'` for 32-bit and `'long'` for 64-bit variables:

Here are some examples of fundamental type data description:

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

Note

Variables of the `long` type are always located at an even address. Assembler checks their address during compilation and automatically makes alignments inserting a short null word if necessary.

2.6.4 Compound Types

The compound types are formed on the bases of fundamental types. The assembler supports two compound types: arrays and structures.

2.6.4.1 Arrays

An array is a finite ordered set of elements of the same type. An array size (number of elements) is statically assigned, i.e. it is defined during the program compilation. Access to separate array elements is made by

means of the array name and the element index in this array. The first array element is considered to have the zero index.

The array definition should contain the number of elements given in square brackets assigned to an element type.

Here are some examples of an array definition:

```
Word : word[32]; // the array of 32 32-bit words.
Long : long[10]; // the array of 10 64-bit words.
```

Examples of getting the array elements addresses:

```
ar0 = Word[4]; // address of the 4-th word of the array.
ar0 = Long[8]; // address of the 8-th long word of the array.
```

Examples of getting the array elements values:

```
ar0 = [Word[4]]; // value of the 4-th word of the array.
ar0,gr0 = [Long[8]]; // value of the 8-th long word of the array.
```

Note

During indexing by array elements the type of the elements is taken into account. If long words are the elements of the array then the distance between two neighbor array elements is equal to two addressed memory cells, for example:

ar0 = Long[0]; // address 0x00000010;

ar1 = Long[1]; // address 0x00000012;

ar2 = Long[2]; // address 0x00000014;

2.6.4.2 Structures

A structure is a complex variable consisting of a finite set of elements of an arbitrary type. Access to elements of the structure is made by means of assigning the structure value name and the element name (see example below).

In order to assign the definition of a structural variable it is first necessary to make description of the structure. Here is an example of a structure pattern description:

```
struct MyStructName // opening bracket of the structure
    F1 : word;      // structure fields
    F2 : long;      //  -//-
    F3 : long;      //  -//-
end MyStructName;   // closing bracket of the structure
```

Note

Description of the structure pattern is an abstract concept until a variable of this type appears in the program. The structure description itself does not occupy any room in memory. That's why in order to improve the program readability it is better to take pattern description outside sections.

Having the structure description it is possible to define a particular variable of this type, for example:

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

In addition to fundamental types any compound types can be used as elements of an array or a structure.

In order to obtain the address of a structure element the point notation is used, for example:

```
nobits ".data"
    ...
    Struct : MyStructName;
    ...
end ".data";
...
begin text
    ...
    ar0 = Struct.F2;  // address of the element F2 of the structure
    gr0 = [Sruct.F3];// value of the element F3 of the structure
    ...
end;
```

It was mentioned above that the variables of the `long` type are always aligned at an even address. The same is true for the fields of a structure. If a field of the `long` type goes after an odd number of fields of the `short` type, this long type field is aligned by adding the short empty field, for example:

```
// Variable of the structured type
Var5: MyStructName;
// ========== Var5 location in memory ================
Var5.F1 (short)    | 0x00000000
 Empty field       | 0x00000001
```

```
Var5.F2 (long)     | 0x00000002
Var5.F3 (long)     | 0x00000004
```

Note *If a structure contains a field of the `long` type, variables of the structured type are aligned at an even address.*

2.6.5 Initialization of Variables

When defining variables it is possible to initialize them with initial values. The initial value or a list of initial values goes after definition of the variable type after the sign ‘=‘. Here are some examples of variable initialization:

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

For the fundamental formats the initial value is assigned in the form of a constant (or a constant expression), for example:

```
Var1 : word = 123 + 32*100;
```

When variables of the long type are initialized the letter 'L' or 'l' should be placed right after the constant to indicate that this number is considered long. For example:

```
Var2 : long = 0ffffffffffffffhl;
```

In case the compound type variables are initialized the list of initial values is divided by commas and taken in round brackets.

The assembler does not compare types of the initializer and the initialized object.

All `long` initializers should be 'L' terminated. Otherwise, the assembler fills high 32 bits of the long word with null. When this happens the assembler gives the related warning. (see NeuroMatrix NM6403 SDK. Programmer’s Reference, the chapter Assembler).

Here are some examples of initializations of variables of compound types:

```
struct MyStruct
    First : word;
    Second: long;
    Third : word[2];
end MyStruct;
...
```

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

If some of the sequential fields of the compound variable are initialized with the same value a special reserved word `dup` (duplicate) followed by the number of repetitions can be used. Example:

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

2.6.6 Variable Definition Area

Each variable has got a definition area. The possible variable definition areas are listed below:

- `local` - local binding variables are used only within the file where they are defined and are not accessible (invisible) outside it;
- `global` - global binding variables are visible both inside the file where they are defined and outside it. Global variable are accessible from any file of the program;
- `extern` – the term “extern” applied to a variable means that the variable is defined somewhere outside the file and will be referred to in this file;
- `weak` - global variables with the weak binding. The same as global but with less priority. If a definition of a global variable with the same name is found in the program the variables of this type are ignored;
- `common` - common variables can be only declared but not defined. They are accessible from any file of the program. The linker itself reserves the room for the common data.

Some comments and examples to every binding type of variables are given below:

The reserved word ‘`local`’ can be omitted in the local variable specifications; it is considered that if in the definition area of the variable is not particularly indicated in the variable declaration/definition then this variable is the local one.

Here are examples of `local`, `global` and `weak` binding type variable definition:

```
data “.MyData”
            Var1 : word[2] = (0,1);// local variable
    global Var2 : long = 0l;       // global variable
    waek   Var3 : word = 1234;     // weak variable
```

```
end ".MyData";
```

The weak type of binding is a subset of the global type. The same rules are correct for the `weak` variables as for the weak labels (see 2.5.4 on page 2-15).

In order to use a reference to a global variable defined outside the current file it is necessary to declare this variable as `extern`. For instance:

```
extern Var3 : word;
```

No room is reserved in memory for the external variable declared in the file. Linker uses extern declaration to resolve references to the global variable address in memory.

Common variables can be declared as follows:

```
common Var : word;
```

Common variables cannot be initialized with an initial value. The requirements to common variables are:

there cannot be two variables with the same name in one file;

variables with identical names declared in different files of the program are united by the linker into a single unit and a single memory block is allocated;

all references to `common` variables (with the same name) throughout the program refer to the same memory address;

`common` variables with identical names can be of different types, for example, if the variable `common MyCommon: word;` is declared in one file and the variable `common MyCommon: long[4]` - in another file, then the total size of the allocated memory is equal to `sizeof(long)*4`;

memory space for variables is allocated in a special section named "`.common`". This section is created automatically if variables of this binding type are found.

2.6.7 File Areas for Variables Declaration, Definition and Initialization

The linker reserves the required memory space when it processes variable definition. If the variable is initialized with an initial value, this value is stored in the reserved location.

Variables of the types `local`, `global`, `weak` should be defined only **within sections**. Sections of initialized and non-initialized data are used for this purpose. For more details about data sections see Section 2.3 on page 2-5.

The linker doesn't reserve memory when it processes declaration of extern or common variables. That's why it is recommended to declare variables of these binding types **outside sections**.

Here are examples of correct and incorrect variable declaration:

```
data ".data"
    // Section internal area.
    global Aglob: word; // correct declaration;
           Bloc : long; // correct declaration;
    weak   Cweak: word; // correct declaration;
end ".data";

// File area outside sections.
global Dglob: word; // incorrect declaration;
commom Ecomm: long; // correct declaration;
       Floc : long; // incorrect declaration;
weak   Gweak: word; // incorrect declaration;
extern Hextr: word; // correct declaration;

data ".data1"
    // Section internal area.
    commom Icomm: long; // incorrect declaration;
    extern Jextr: word; // incorrect declaration;
end ".data1";
```

2.7 Assembler Directives

This section contains description of the assembler directives available. Unlike the processor instructions, directives are not translated into any special code, but only influence the process of the program compilation. The assembler directives permit:

- to perform conditional compilation;
- to align the program elements like instructions and variables;
- to define the number of data blocks repetitions;
- to contain debug information in an assembler file;
- to permit or not parallel execution of a processor instruction.

The assembler directive line should be terminated with ";".

Two tables of directives are given below. These are the summary table without a list of debugging information directives and the table of debug information directives. The directives are listed in the alphabet order.

Table 2-5. Summary Table of Assembly Directives (Part 1)

MNEMONICS	DESCRIPTION
`.align`	Even address alignment.
`.branch`	Switch on the processor instructions parallel execution.
`.else`	Beginning of the alternative conditional compilation block
`.endif`	End of the conditional compilation block.
`.endrepeat`	End of the instructions repetition block.
`.if condition`	Beginning of the conditional compilation block.
`.repeat` *number of repeats*	Beginning of the instructions repetition block.
`.wait`	Switch off the processor instructions parallel execution.

The assembler has a set of directives to store debug information in an assembler file.

The assembler supports debug information representation according to the standard DWARF, version 2.0. The description of standard DWARF is contained in the document: **TIS Committee "DWARF Debugging Information Format. v2.0"**. All directives of debug information start with the prefix '`.debug_`':

Table 2-6. Summary Table of Assembly Directives (Part 2)

MNEMONICS	DESCRIPTION
`.debug_arange`	Information about the program address ranges;
`.debug_die`	Description of a new DIE (Debug Information Entry);
`.debug_die_child`	Description of a new DIE which is a child DIE of `.debug_die`;
`.debug_die_endchild`	Description of the last child DIE in the chain of child DIEs of `.debug_die`;
`.debug_end_sequence`	Marker of the continuous code block end;
`.debug_frame_cie`	Information about call stack;
`.debug_frame_fde`	Information about call stack;
`.debug_line`	Information about the source text lines;
`.debug_pubname`	Information about global symbols;
`.debug_root_die`	Description of the root DIE of the compilation

	unit (CU);
`.debug_source_directory`	Information about the paths to source texts of the program;
`.debug_source_file`	Information about the source texts files;
`.debug_start_sequence`	Marker of the continuous code block beginning;

Every debug information directive has several arguments enumerated through commas and ends with a semicolon. For example:

```
.debug_line     2, 3, 1;
```

In order to understand correctly the debugging information directives structure and principles of use it is necessary to get familiar with the description of debug information standard DWARF (the document: **TIS Committee "DWARF Debugging Information Format. v2.0"**).

2.7.1 Directive .align

The `.align` directive informs the assembler that the data following the directive should be located at even memory address. Empty space filled with null will be inserted to the code/data section if necessary.

The `.align` directive does not require any additional parameters.

The `.align` directive cannot be used outside sections. If it is used in the code section, the `nul` instruction will be inserted into the program for alignment. If it is used in the initialized data section the empty non-referenced space filled with zero will be inserted. If it is used in the non-initialized data section the current address will be increased by 1.

Here is an example of the `.align` directive use:

In data sections:

```
data "Init"          // section starts always with
                     // an even address.
    Var1: word[5] = ( -1 dup 5);
                     // odd number of elements.
                     // the next variable should be
                     // located by an odd address.
    .align;          // alignment directive;
                     // assembler skips 32-bit word.
                     // starts with an even address.
    Var2: word[2] = ( 5A5A5A5Ah dup 2);
end "Init";
```

The data are located in memory in the following order:

```
00: FFFFFFFF FFFFFFFF   // Var1 5 elements.
```

```
02: FFFFFFFF FFFFFFFF
04: FFFFFFFF 00000000   // inserted zero word.
06: 5A5A5A5A 5A5A5A5A   // Var2 starts from an even address.
```

In code sections:

```
begin "textFunc"     // section starts always with
                     // an even address.
   ...
   gr0 = [Var1];     // long instruction is located
                     // at an even address.
   gr1 = gr0 << 1;   // short instruction.
                     // the next instruction should be
                     // located at an odd address.
   .align;           //alignment  directive;
                     // assembler inserts null.
   gr2 = not gr1;    // short instruction,
                     // is located at an even address.
end "textFunc";
```

2.7.2 Directives .branch and .wait

The .branch directive sets the bit of parallel instructions execution to '1'. All instructions following the directive have this bit set to '1' until the .wait directive is found or the end of file is achieved.

The .wait directive sets the bit of parallel instructions execution to '0'. All instructions following the directive have this bit set to '0' until the .branch directive is found or the end of file is achieved.

The .branch and .wait directives do not require any additional parameters. They affect only instructions but not data. They should be used only inside code sections.

By default the bit of parallel execution is set to '0'. Use of the .branch directive allows the user to switch on the mode of processor instructions parallel execution. Use of the .wait directive allows the user to return the processor to the sequential instruction execution mode.

Here is an example of the .branch and .wait directives use:

```
begin "textFunc"
<MyFunc>          // label of the function beginning;
                  // bit of parallel execution is set to 0 (by default).
    ar0 = Vector;
.branch;          // bit of parallel execution is set to 1.
                  // hereafter in all instructions have this bit set to 1.
```

```
    rep 16 ram = [ar0++]; // vector instruction
                          //It is executed for 16 clock cycles
    gr0 =  gr1 << 4;// scalar instruction
                    // is executed in parallel with the vector instruction.
.wait;              // bit of parallel execution is returned to 0.
                    // hereafter in all instructions have this bit set to 0.
    gr0 = gr1 << 4; // scalar instruction
                    // waits while the vector instruction is not finished.
end "textFunc";
```

2.7.3 Directives .if and .endif

The .if *condition* directive defines the beginning of the block of conditional compiling. It is used together with .endif. The result of the expression following the .if directive is considered as a Boolean. The assembler checks the condition and if it is true the assembler compiles the block.

The conditional compilation unit started with .if should be terminated with .endif. This means the directives are used as brackets, for instance

```
.if AAA > 10;
   . . .
.endif;
```

The block of conditional compiling can be used both in the data sections and in code sections. It cannot be located outside sections.

Here is an example of a program using the block of conditional compiling:

```
const DEBUG = 1; // the constant is used for debug mode.
nobits ".data"   // data section.
...
.if DEBUG;       // beginning of the block of conditional compiling.
    GenRegs: word[8]; // space to store the values of 8 registers.
.endif;          // end of the block of conditional compiling.
end ".data";
...
begin ".text"    // code section.
...
.if DEBUG;    // beginning of the block of conditional compiling.
    [GenRegs[0]] = gr0; // in the debug mode
    [GenRegs[1]] = gr1; // all general purpose
```

```
        [GenRegs[2]] = gr2; // registers are stored to
        [GenRegs[3]] = gr3; // the allocated memory
        [GenRegs[4]] = gr4; // area.
        [GenRegs[5]] = gr5; // In the release mode,
        [GenRegs[6]] = gr6; // when DEBUG = 0 this block
        [GenRegs[7]] = gr7; // is skipped.
    .endif;                 // end of the block of conditional compiling.
    ...
    end ".text";
```

2.7.4 Directives .repeat and .endrepeat

The `.repeat` *`number of repetitions`* directive defines the beginning of the block which should be repeated in memory as many times as the parameter *`number of repetitions`* indicates.

The `.endrepeat` directive is used as a terminator of the instructions block to be repeated. It does not need additional parameters.

The number of repetitions is defined by a positive constant. Brackets can be used as an alternative to cycle. But one should remember that a large number of iterations could largely increase the code volume.

Here is an example of `.repeat ... .endrepeat` block use:

```
begin ".text"
    ...
    gr2 = [Mask];
    gr0 = [ar0++];
.repeat 9; // repeat the block of two instructions for 9 times.
    gr0 = [ar0++] with gr1 = gr0 and gr2;
    [ar1++] = gr1;
.endrepeat; // end of the block.
    [ar1++] = gr1;
    ...
end ".text";
```

No definitions of labels or variables should be found inside the block of repetitions because those definitions would be textually duplicated and it would cause a flow of syntax errors.

2.7.5 Directives of Debugging Information

2.7.5.1 Directive .debug_arange

The directive `.debug_arange` adds an information about an address range described in a current CU (Compilation Unit) for quick search at

the address. The directive has two parameter: an address and a length. The address is an address expression. Consequently, this parameter may be assigned by an expression computing as relocatable.

An example:

```
.debug_arange   function, fend - fbegin;
```

// address range:
// [function .. function + fend - fbegin]

// pertains to the present CU.

2.7.5.2 Directives .debug_die and .debug_die_child

These directives build a new debug information entry DIE in the current CU. The directive `.debug_die` builds the DIE as a brother of the previous DIE. The directive `.debug_die` builds the DIE as a son of the previous DIE.

The first parameter of the directives contains an identifier, which is used as a DIE label to referring to the current DIE from others debugging directives by the assembler. Nothing concerns this label with a DIE name, which is an attribute value of DW_AT_name.

The second parameter of the directives contains an integer tag of DIE type. Values of the various tags are given in the description of the DWARF format. For each quantity of the DWARF format it defines a constant in the macro library dwarf_ct.mlb supplied with the library. The quantity names correspond with the ones used in the standard and begin with DW_TAG_. Hereinafter in the description the names are used instead of the numerical values.

The others directive parameters represent a list of descriptions of DIE attribute:

name, form, value,

name, form, value,…

The attribute name is a constant expression. As a symbolic name it possible to use the names defined in the DWARF standard (also in dwarf_ct.mlb). The standard attribute names begin with DW_TAG_.

The form is the reserved word, which is, determined the form of the attribute value presentation. The possible forms of DIE attribute are presented in the Table 2-7. The DIE Attribute Forms.

Table 2-7. The DIE Attribute Forms.

FORM	DESCRIPTION
`ref`	Reference to another DIE.
`addr`	Address in the destination space.

`flag`	Flag.
`uconst`	Unsigned constant.
`sconst`	Signed constant.
`block`	Data block.
`locdesc`	Location descriptor.
`string`	Character string.

The value is the attribute value, which depending on the attribute form has a various meaning, see Table 2-8.

Table 2-8. The relation of the DIE attribute values with their form.

FORM	MEANING	NOTE
`ref`	DIE label (the first parameter any DIE).	The assembler generate the value of this attribute as the relative CU address of DIE, which was being referred , in a section of an object file with a name “`.debug_info`”.
`addr`	Relocatable address in the destination space.	The assembler handles this address as every relocatable address in the program. The attribute value must be either a label, either a variable, or an address expression.
`flag`	One bit constant.	
`uconst, sconst`	Constant interpreted as a number (correspondingly unsigned or signed).	
`block`	Data block, i.e. doesn't interpreted bite chain, enumerated by coma in figure brackets.	The values in the figure brackets must be either constants interpreted as bytes, or address expressions.
`locdesc`	Instruction sequence of an abstract stack machine. This sequence defines a location of the object, described by DIE.	All sequence must be enclosed in the figure brackets, besides every instruction, including three constants (maybe address constants), also must be enclosed in the figure brackets, see example: {{12,23,24},{24,24,0},...}
`string`	character string.	Every symbol strings used in the debug instructions must be enclosed in single inverted commas, otherwise in accordance

		with assembler every string symbol will represent by four bytes in the debugging information.

Examples of pseudo instruction .debug_*die*:

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

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

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

2.7.5.3 Directive .debug_die_endchild

The directive `.debug_die_endchild` concludes a current level of a DIE tree, then a transition to a high level (to a father of last DIE) occurs, the next DIE will be the brother of the father of last DIE.

This directive doesn't have parameters.

2.7.5.4 Directives .debug_start_sequence и .debug_end_sequence

The directives `.debug_start_sequence` and `.debug_end_sequence` limit scraps of a continuous progran code. A directives `.debug_line` may be located only between these delimiters.

These directives don't have parameters.

2.7.5.5 Directive .debug_frame_cie

The directive adds a new entry (Call Information Entry) to information about a call stack.

It has the next parameters:

- information line about a function generating the CIE,
- code alignment,
- data alignment,
- register number of a return address,
- command list of an abstract machine – instructions contained in the CIE (written in the figure brackets).

An example:

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

2.7.5.6 Directive .debug_frame_fde

The directive adds a new entry (Frame Description Entry) to information about a call stack.

It has the next parameters:

- sequence number of a respective CIE,
- address of a beginning of a given function in the destination space,
- code length of a function,
- register number of a return address,
- command list of an abstract machine – instructions contained in the FDE (written in the figure brackets).

An example:

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

2.7.5.7 Directive .debug_line

The directive adds an information about a line number respective to a given address.

The directive format:

`.debug_line <line number> [,<file number>] [,<column number>]`

The file number must be designated by the directive `.debug_source_file` earlier. The file number may be absent meaning that it is the same at the previous directive `.debug_line`. The very first directive `.debug_line` must possess the file number. The assembler uses the current address as the line address. The third parameter (column

number) may be omitted like the second parameter (file number) in this case the column number in the given line is considered indefinite.

An example:

```
.debug_line      10,5,6;   // the 10th line of the 5th file
.debug_line      11;       // the 11th line of the same file.
```

2.7.5.8 Directive .debug_pubname

In order to speed up a symbol search the directive `.debug_pubname` adds a name of a global symbol and a reference to a DIE, which including an information about this symbol, to a section `.debug_pubname`.

The command has two parameters:

- symbol name in the form of a character string,
- reference to the DIE in the form of a DIE label.

An example:

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

2.7.5.9 Directive .debug_root_die

The directive builds a Compilation Unit (CU) and adds a root DIE to it. The root DIE contains all CU attributs as a whole. All the subsequent directives `.debug_die` builds a new DIE in a CU. The directives `.debug_root_die` and `.debug_die` have the same parameters, i.e. both describe a DIE. This directive is separated form `.debug_die` in order to logically distinguish the root DIE.

The assembly compiler supports an existence of only one compilation unit therefore it is impermissible reentry of the directive `.debug_root_die.`

2.7.5.10 Directive debug_source_directory

The directive has two parameters:

- directory number,
- directory path.

The file number has an informative meaning and must be equal to a sequence number of the directive `.debug_source_directory`. A command numeration begins with one.

All of source files must be available by one of the directories specified by the directive `.debug_source_directory`.

An example:

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

2.7.5.11 Directive debug_source_file

The directive `.debug_source_file` has two parameters:

- file number,
- directory number,
- name,
- size,
- date of a source file.

Though the size and date of file be obligatory but their accuracy isn't verified. They can be used by debugger. All of the source files must be described by the directives `.debug_source_file`. Pseudo commands use the file number to indicate files. The directory number must be specified earlier by the directive `.debug_source_directory`.

An example:

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

2.8 Pseudo Functions

Several pseudo-instructions are introduced to the NM6403 assembler. These pseudo-instructions simplify recording and calculation of constant expressions of the program and make it easier to read.

Assembler processes pseudo functions on the compilation stage. Pseudo-functions use a constant or a constant expression as input data. The result of their work is also a constant, which is then used for variable initialization, registers modification or addressing.

Pseudo-functions can be regarded as a part of constant expressions. That's why they cannot be used in expressions both inside and outside sections.

The NM6403 assembly language contains the following list of pseudo-functions:

Table 2-9. Summary Table of Assembly Pseudo Functions

PSEUDO FUNCTIONS	DESCRIPTION
`double`	Transformation of a 64-bit floating point number into an internal presentation (IEEE-754).
`float`	Transformation of a 32-bit floating point number into an internal presentation (IEEE-754).
`hiword`	Obtaining the higher part of a 64-bit word.
`loword`	Obtaining the lower part of a 64-bit word.

`offset`	Obtaining the offset of the structure field relatively to the structure address.
`sizeof`	Obtaining the fundamental or the compound type size.

2.8.1 Function loword

Function `loword` is used to get the lower part of a 64-bit constant or a constant expression.

The following example shows the way of the pseudo-function `loword` use:

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

This function returns the 32-bit value.

Function `loword` cannot be used in an address expression because address expressions cannot give the result where the number of significant bits exceeds thirty-two bits.

2.8.2 Function hiword

Function `hiword` serves to get the higher part of a 64-bit constant or a constant expression.

The following example shows the way of the pseudo-function `hiword` use:

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

This function returns the 32-bit value.

Function `hiword` cannot be used in an address expression because address expressions cannot give the result where the number of significant bits exceeds thirty-two bits.

2.8.3 Function sizeof

Function `sizeof` serves to get the real size of the desired data type taking into account different internal structure alignments.

The function `sizeof` returns the size of the type measured in 32-bit words.

The following example shows the way of the pseudo-function `sizeof` use:

```
struct S            // declaration of the structure type.
    Var1: word;  // short word, a blank space is after
    Var2: long;  // it before a long word.
    Var3: word[4];
end S;
begin ".text"
    gr0 = sizeof(S);  // the result obtained: 8 words.
end ".text";
```

The function sizeof calculates the size of types but not variables themselves. Here are some examples of correct and incorrect use of `sizeof`:

The **correct** use of the function, because a name of the type is given as the argument:

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

The incorrect use of the function, because a name of the variable is given as the argument instead of the type name:

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

2.8.4 Function offset

The `offset` function serves to get the offset of the desired field in the structure taking into account possible internal alignments.

The `offset` function returns the field offset referred to the structure beginning address. The offset is measured in 32-bit words.

This function is useful to access the field of the structure addressed to through a register.

Here is an example of the `offset` function use:

```
struct S
    Var1: word;
    Var2: long;
end S;
...
data ".data"
    VarStruct: S = (1, -1l,);
end ".data";
...
begin ".text"
    //Put the structure variable address to the register.
```

```
    ar0 = VarStruct;
    // Read the field of the structure.
    ar1, gr1 = [ ar0 += offset(S, Var2) ];
end ".text";
```

The input parameter of the function `offset` is a data type but not a variable name. Here are some examples of correct and incorrect use of `offset`:

The **correct** use of the function, because a name of the type is given as the argument:

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

The incorrect use of the function, because the name of the variable is given as the argument instead of the type name:

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

2.8.5 Functions float and double

Function `float` converts a variable written in the floating-point format into an internal 32-bit presentation according to IEEE-754.

Function `double` converts a variable written in the double format into an internal 64-bit presentation according to IEEE-754.

The processor NM6403 does not support hardware floating-point operations on the hardware level. That's why all the floating-point arithmetic is software implemented in the form of a library included into `libc.lib`.

The assembler uses the format IEEE-754 for internal presentation of floating-point and double numerals.

The floating point and double point format representations used in the NM6403 assembly language are:

`[+|-]num[.numE][+|-]num,` *where* `num` – *decimal numbers*.

Or

`[+|-]num.num`

Here are some examples of the functions `float` and `double` use:

```
Float1: word = float( 1.57 ) - float(-4.32E2);
Float2: word = float( -4.98 ) + float(5.51E2);
Float2: word = float( -2.23E-3 ) + float(5.51E-2);
Double1: long = double(-2.34560976E-18);
```

In the given examples all possible formats of floating-point and double presentation supported by the assembler are shown.

2.9 Using Macros

2.9.1 Purpose of Macros

If user uses a collection of some command repeatedly he can simplify his action using a macro. A macro is a series of assembler commands and instructions that user groups together as a single command to accomplish a task automatically. Here are some typical uses for macros:

- To speed up routine editing and formatting
- To combine multiple commands
- To automate a complex series of tasks
- To make a program more clear

A user can store its macros in an own macro library.

2.9.2 Syntax of Macros

Macro definition:

```
macro macro_name ( [parameter1 [, parameter2 ...]] )
   entity_sequence
end macro_name;
```

Macro call:

```
macro_name( [parameter1 [, parameter2 ...]] );
```

Declaration of an external macro:

```
import [macro_name1 [, macro_name2 ...]] from library_name;
```

2.9.3 Description

A *macro_name* is an arbitrary identifier. Formal arguments should be also identifiers.

A *entity_sequence* should be the sequence of complete syntactic units in assembly language: declarations, definitions, instructions, macro inclusion and etc (except sections). Syntax and semantic verification of a macro are performed only at the macro substitution taking into account the environment and the actual arguments of a macro call.

The next may be used as the formal arguments of macro call:

- Registers
- Indefinite identifiers
- Constant expression

The registers and the identifiers are passed by name the constant expressions are passed only by value. The constant expressions are precomputed and so they of the macro substitution are a number or an address, which are the results of the calculating.

Note *It is impossible to pass a defined program entity by name into the macro. For example, it is impossible to pass a name of a compile time variable which is varied in the macro.*

Note *The pass of indefinite identifiers allows to create a new program entity into the macro by next way:*

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

Macro definitions may be arranged at any place of a program but the place outside the sections at the beginning of a file is more preferably.

Macro substitutions are allowed at any place when the commands from a macro are permitted to use.

Both an evident and an indirect recursive macro call of the same macro are forbidden.

2.9.4 Using Label in Macros

A restriction exists for using a label in the macros that the macro with the label in the body may be included only once, because a error – "`Redefining a label`" will be occur in the next inclusion. In order to use the label in the macros should be declared with a reserved word `own`. The label declared in that way is processed by a particular mode so, as at the every macro substitution this label will have a unique name among all substitution of this macro. For example:

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

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

```
    if <>0 skip Cont;
    Res++;
<Cont>
end FEQ;
```

Macro `FEQ()` using the own label `Cont` for inner branch may be included more than one.

Using the reserved word `own` is permitted only in the macros.

2.9.5 Importing Macros from Marco Library

The `import from` directive enables use the macros from an external macro library.

By means of the import directive the declaration of the external macros directs include definitions of the specified macros from an external macro library to the assembler. A filename of the macro library doesn't contain a pathspec. If a filename extension of macro library is standard (`.mlb`) the filename may be specified without the extension. If the list of the included macros is omitted all macros will be included from specified macro library.

Here examples of declaration of external macros:

```
import mode_constants  from com_decl.mlb;

import mode_constants, irqtab_layout  from
com_decl.mlb;

import  from com_decl.mlb;
```

The first declaration includes a macro from the macro library com_decl.mlb; the second declaration includes two macros from the macro library and the third declaration includes all macros of the macro library.

Assembler searches the macro libraries at first in a current directory, then in directories specified in command line by means key `-I`.

Note *For details about a macro library creation by means of the assembler and its operation mode see NeuroMatrix NM6403 SDK. Programmer's Reference, the chapter Assembler.*

Note *The situation may be occurring when macros from the macro library of an old assembler version are interpreted incorrectly by the new assembler release. In this case it is necessary to recompile the macro library.*

3 Registers

This section describes three register files of the NeuroMatrix® NM6403:

- Primary register file;
- Peripheral control register file;
- Vector register file.

3.1 Primary Register File

The primary register file provides 16 registers in a multiport register file. This file consists of a set of address registers and a set of general-purpose registers, i.e. the registers that are used in most of the processor computing operations.

There are eight address registers and eight general-purpose registers (see Table 3-1). All the registers are 32-bit read/write accessible registers.

Table 3-1. Primary Register File of NeuroMatrix NM6403

ADDRESS REGISTERS	GENERAL-PURPOSE REGISTERS
`ar0`	`gr0`
`ar1`	`gr1`
`ar2`	`gr2`
`ar3`	`gr3`
`ar4`	`gr4`
`ar5`	`gr5`
`ar6`	`gr6`
`ar7(sp)`	`gr7`

3.1.1 Address Registers

The primary function of address registers is to support the variety of indirect addressing modes. The address registers can not be used as general-purpose registers because they do not participate in arithmetic/logical operations.

Address registers are divided into two groups. The first group includes `ar0`…`ar3`, and the second one - `ar4`…`ar7`. This division is due to two data address generators (DAG). As illustrated in Figure 1-3, the first group of the registers is mapped to the DAG1, the second one to the DAG2.

There are some limitations on using the address registers from different groups in the same processor instruction. More information about this will be given in the section describing operations of address registers modification (see 5.1.7 on page 5-16).

Address registers are used only in the left part of an assembly instruction.

Here are some examples of assembly instructions containing address registers:

```
ar0 = ar5;                          // copying
ar2 = ar3 + gr3;                    // modification
[ar4++] = gr7 with gr7 -= gr4;      // store into memory
```

The processor uses the address register `ar7` as the pointer to the top of the system stack. This means that `ar7` is modified automatically when a sub-routine call or an interrupt occurs, and when return from a sub-routine or from an interrupt takes place. The synonym of `ar7` is `sp(Stack Pointer)`. The register `sp` can be used instead of `ar7` and vice versa, for example, the following assembly instructions mean the same:

```
ar5 = ar7 – 2;
```

or

```
ar5 = sp – 2;
```

3.1.2 General-Purpose Registers

The general-purpose registers are capable of storing and supporting operations on 32-bit numbers. The general-purpose registers can be operated upon by ALUs and can be used to support some indirect addressing modes, for example

```
[gr0] = gr4; // store the contents of gr4 into memory
             // at the address contained in gr0.
```

Here are some examples of general-purpose registers:

```
gr0 = gr5;                          // copying.
gr2 = gr1 + gr3;                    // modification.
[ar4++] = gr7 with gr7 -= gr4 ;     // writing to memory
```

Note *NeuroMartix NM6403 RISC-core does not contain any selected register to provide loop counter.*

3.1.3 Register Pairs

An address register and a general-purpose register with the same number are referred to as a register pair. There are eight register pairs. The register pairs are used to support operations on 64-bit words like load/store or some types of addressing modes.

Here are some examples of assembly instructions where a register pair is used:

```
// Load a 64-bit word from memory to the register pair
ar0, gr0 = [ar1++];
```

or

```
// Vector instruction. Load eight 64-bit words from external memory
// and perform bitwise logical complement of each word. The
// example shows use of a register pair for indirect addressing.
rep 8 data = [ar3+=gr3] with not data;
```

An address register and a general-purpose register with different indexes cannot be referred to as a register pair, for example:

```
ar0, gr1 = [ar1++]; // incorrect instruction.
[ar0+=gr1] = gr4;   // incorrect instruction.
```

3.2 Peripheral Control Register File

The peripheral control register file registers are used to control external memory interfaces, communication ports, DMA co-processors, timers and so on.

Table 3-2. Peripheral Control Register File of NeuroMatrix NM6403

NAME	DESCRIPTION	NOTE
gmicr	Global Memory Interface Control Register. It can be programmed to control global memory parameters like page size, address range, wait states, and other operations that control global memory interface.	32 bits.
ica0, icc0	Address register and data counter. They control the 0th communication port in input mode.	Register pair, 32 bits each.
ica1, icc1	Address register and data counter. This register pair controls the 1st communication port in input mode.	Register pair, 32 bits each.
intr	Register of interrupt requests and direct memory access (DMA). It shows the interrupts state, DMA co-processors state, and state of internal buffers of the Vector Unit.	32 bits.
lmicr	Local Memory Interface Control Register. It can be programmed to control local memory parameters like page size, address range, wait states, and other operations that control local memory interface.	32 bits.
oca0, occ0	Address register and data counter. They control the 0th communication port in output mode.	Register pair, 32 bits each.
oca1, occ1	Address register and data counter. They control the 1st communication port in output mode.	Register pair, 32 bits each.
pc	Program counter. It contains the address of the next instruction to be fetched.	32 bits.
pswr	Processor State Word Register. It controls external memory share modes, Timer pin, communication	32 bits.

	ports, interrupt masks, condition flags.	
t0, t1	Timer counter registers.	32 bits.

3.2.1 Register gmicr

The `gmicr` is a 32-bit register that can be programmed to control global memory interface by defining the:

- page size used for the two memory banks at each port;
- address ranges over which memory banks are active;
- wait states;
- other operations that control the memory interface.

Table 3-3 describes the fields in this register.

Register `gmicr` is accessible both for reading and for writing.

The `gmicr` must be set up before any application starts executing. Usually the loader program, which is included to the load and exchange library, sets it up.

Note

The user can change the contents of `gmicr` register himself. However it is necessary to be careful because an incorrectly configured register will not allow the processor to correctly access global memory. It can cause the program halt or incorrect execution.

Table 3-3. Global Memory Interface Control Register (gmicr)

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

Field BOUND

Field `BOUND` specifies address space configuration. It determines the size and address range of bank0, presence of the bank 1, its size and address range (bank 0 is always present). The bits occupied by the field in gmicr register are shaded:

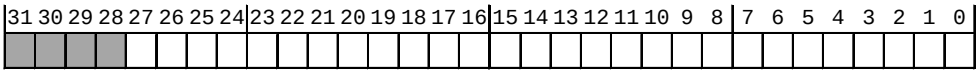


Table 3-4 contains information about the possible variants of global memory configuration.

Table 3-4. Division of Global Bus Address Space into Banks 0/1 According to BOUND

BOUND	SIZE OF BANK0 (64 BITS)	ADDRESS SPACE OF BANK0	SIZE OF BANK1 (64 BITS)	ADDRESS SPACE OF BANK1
0000	2^{15} = 32K	80000000 - 8000FFFF	32K	80010000 - FFFFFFFF
0001	2^{16} = 64K	80000000 - 8001FFFF	64K	80020000 - FFFFFFFF
0010	2^{17} = 128K	80000000 - 8003FFFF	128K	80040000 - FFFFFFFF
0011	2^{18} = 256K	80000000 - 8007FFFF	256K	80080000 - FFFFFFFF
0100	2^{19} = 512K	80000000 - 800FFFFF	512K	80100000 - FFFFFFFF
0101	2^{20} = 1M	80000000 - 801FFFFF	1M	80200000 - FFFFFFFF
0110	2^{21} = 2M	80000000 - 803FFFFF	2M	80400000 - FFFFFFFF
0111	2^{22} = 4M	80000000 - 807FFFFF	4M	80800000 - FFFFFFFF
1000	2^{23} = 8M	80000000 - 80FFFFFF	8M	81000000 - FFFFFFFF
1001	2^{24} = 16M	80000000 - 81FFFFFF	16M	82000000 - FFFFFFFF
1010	2^{25} = 32M	80000000 - 83FFFFFF	32M	84000000 - FFFFFFFF
1011	2^{26} = 64M	80000000 - 87FFFFFF	64M	88000000 - FFFFFFFF
1100	2^{27} = 128M	80000000 - 8FFFFFFF	128M	90000000 - FFFFFFFF
1101	2^{28} = 256M	80000000 - 9FFFFFFF	256M	A0000000 - FFFFFFFF
1110	2^{29} = 512M	80000000 - BFFFFFFF	512M	C0000000 - FFFFFFFF
1111	2^{30} = 1G	80000000 - FFFFFFFF	0	-

Field PAGE1

Field PAGE1 specifies memory page size for the bank 1. The bits occupied by this field in gmicr register are shaded:

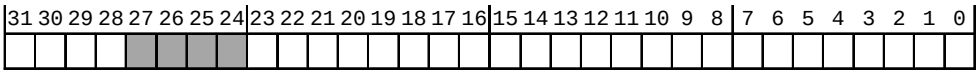


Table 3-5 contains information about possible page sizes supported by the processor.

Table 3-5. Memory Page Sizes According to Field PAGE(0,1)

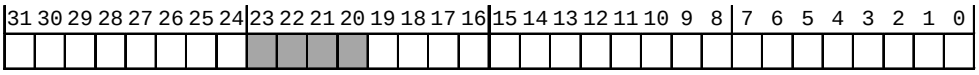
FIELD PAGE(0,1)	MEMORY PAGE SIZE (IN 64-BIT WORDS)
0000	2^{8} = 256 words
0001	2^{9} = 512 words

0010	2^{10} = 1K words
0011	2^{11} = 2K words
0100	2^{12} = 4K words
0101	2^{13} = 8K words
0110	2^{14} = 16K words
0111	2^{15} = 32K words
1000	2^{16} = 64K words
1001	2^{17} = 128K words
1010	2^{18} = 256K words
1011	2^{19} = 512K words
11xx	Reserved

Note *Combinations of bits 1000 and higher (the last 5 rows of the table) are valid only for static memory (SRAM).*

Field PAGE0

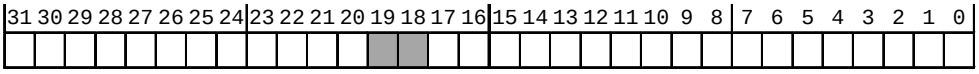
Field PAGE0 specifies memory page size for the bank 0. The bits occupied by this field in gmicr register are shaded:



The possible contents of the field PAGE0 are listed in Table 3-5.

Field TYPE1

Field TYPE1 defines the memory type of bank 1. The bits occupied by it in gmicr register are shaded:



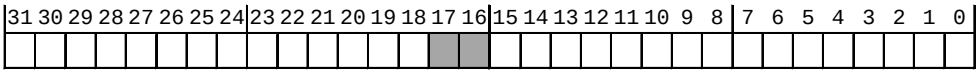
The field assumes four different values.

00	Access to static memory (SRAM).
01,10,11	Access to dynamic memory (DRAM).

The more detailed information on different types of dynamic memory can be found in the document: **NeuroMatrix®NM6403 Processor. User's Guide.**

Field TYPE0

Field TYPE0 defines the memory type of bank 0. The bits occupied by it in gmicr register are shaded:



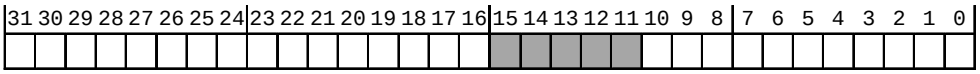
The field assumes four different values.

00	Access to static memory (SRAM).
01,10,11	Access to dynamic memory (DRAM).

The more detailed information about different types of dynamic memory can be found in the document: **NeuroMatrix®NM6403 Processor. User's Guide.**

Field TIME1

Field TIME1 specifies software wait-state count for the bank 1 accesses. Defines the number of cycles to use when software wait states are active. The bits occupied by this field in gmicr register are shaded:



Memory access procedure may consist of up to five phases. Types of these phases and their programmable duration are given in Table 3-6.

Table 3-6. Phases of Memory Access Cycle

PHASE OF MEMORY ACCESS CYCLE		PHASE DURATION		
NAME	DESCRIPTION	DESIGNATION	DRAM	SRAM
RP	Phase of discard of signals $\overline{RAS0}/\overline{CS0}$ and $\overline{RAS1}/\overline{CS1}$.	T_{RP}	(1-2)T	1T
PAGE	Phase of memory page addressing.	T_{PAGE}	(1-2)T	1T
CP	Passive phase of memory cell addressing.	T_{CP}	(0-1)T	(0-3)T
CA	Active phase of memory cell addressing.	T_{CA}	(1-2)T	(1-4)T
BE	Phase of memory entering the high-impedance state.	T_{BE}	(1-2)T	(1-2)T

There are some comments and reference information on the statements of the table:

- Abbreviation T corresponds to one processor clock cycle.
- Phases RP and PAGE take part in the memory access procedure only when the processor accesses different memory page.
- Phase CP is optional. If T_{CP}=0 then this phase does not take part in the memory access procedure.

- Phase BE takes part in memory access procedure only if a *write to memory* command goes right after *read from memory* command.

The format of the field TIME1 depends on memory type, whether SRAM or DRAM is used.

Table 3-7 shows the field TIME1 format in case of use the static memory SRAM, and Table 3-8 shows its format in case the dynamic memory DRAM is used.

Table 3-7. Format of Fields TIME0 and TIME1 of gmicr(lmicr) Register for SRAM

BITS OF TIME1	BITS OF TIME0	DESIGNATION	PHASE DURATION	
15	10	T_{BE}	0 1	2T 1T
14 13	9 8	T_{CP}	00 01 10 11	3T 2T 1T 0T
12 11	7 6	T_{CA}	00 01 10 11	4T 3T 2T 1T

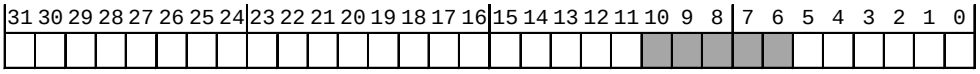
In this sense, zero wait-state is generated by the processor to access SRAM when the field TIME1(TIME0) is equal to 11111_2.

Table 3-8. Format of Fields TIME0 and TIME1 of gmicr(lmicr) Register for DRAM

BITS OF TIME1	BITS OF TIME0	DESIGNATION	PHASE DURATION	
15	10	T_{BE}	0 1	2T 1T
14	9	T_{PAGE}	0 1	2T 1T
13	8	T_{CP}	0 1	1T 0T
12	7	T_{RP}	0 1	2T 1T
11	6	T_{CA}	0 1	2T 1T

Field TIME0

Field `TIME0` specifies software wait-state count for the bank 0 accesses. Defines the number of cycles to use when software wait states are active. The bits occupied by this field in gmicr register are shaded:



The format of the field `TIME0` is the same as that of the field `TIME1`. Refer to Table 3-7 and Table 3-8 for more information.

Field TRAS

Field `TRAS` specifies duration of the active level of $\overline{RAS}$ signal. The bits it occupies in `gmicr` register are shaded:

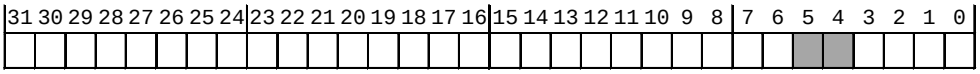


Table 3-9 reflects the dependency of $\overline{RAS}$ signal active level duration on `TRAS` contents.

Table 3-9. Duration of $\overline{RAS}$ Signal Active Level

TRAS	DURATION IN CYCLES (T_{RAS})
00	4
01	3
10	2
11	1

Note

If SRAM memory type is indicated in both fields TYPE0 and TYPE1, the field TRAS contents do not care.

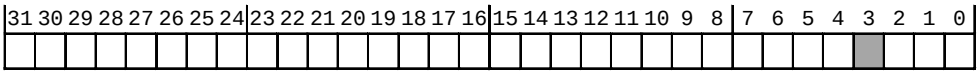
The more detailed information about dynamic memory regeneration can be found in the document: **NeuroMatrix®NM6403 Processor. User's Guide.**

Field RDY1

Field `RDY1` specifies the condition of finish of the bank1 memory access procedure:

- `0` – only due to the internal counter;
- `1` –due to the internal counter and the external READY signal.

The bits it occupies in gmicr register are shaded:

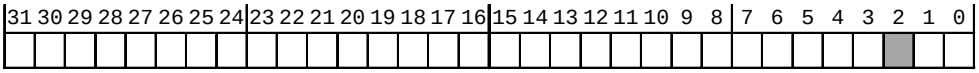


Field RDY0

Field `RDY0` specifies the condition of finish of the bank0 memory access procedure:

- `0` – only due to the internal counter;
- `1` –due to the internal counter and the external READY signal.

The bits it occupies in gmicr register are shaded:



Note

If one of multiprocessor configuration modes is selected in the field SHMEM (see below), the contents of the fields RDY1 and RDY0 do not care.

Field SHMEM

The NeuroMatrix® NM6403 supports construction of dual processor-based shared memory systems. There are three modes of shared memory interface. The special terms are introduced to describe those modes.

Own memory bank - this term means that right after RESET this bank is owned by the processor. To access the own memory bank the processor doesn't need to ask the permission from the other processor.

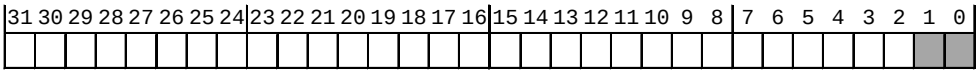
The other's memory bank - this term means that right after RESET this bank is not owned by the processor. To access the other's memory bank the processor must first ask the permission and only after it has got it the processor accesses that bank.

Common memory bank - this term means that the memory bank is accessible for both processors, but is not owned by any one. Each of the processors must ask the permission to access the memory and if the other processor does not access the memory at that moment the processor will be able to access the bank.

Field `SHMEM` specifies the possible configurations of shared global bus interface:

- 00 – multiprocessor configuration of the first type (bank0 - "common", bank1 - "common");
- 01 - multiprocessor configuration of the second type (bank 0 - "own", банк 1 - "common");
- 10 - multiprocessor configuration of the third type (bank 0 - "own", банк 1 - "the other's");
- 11 – one-processor configuration (no memory is shared).

The bits it occupies in gmicr register are shaded:



The more detailed information about the shared memory configurations can be found in the document: **NeuroMatrix®NM6403 Processor. User's Guide.**

3.2.2 Registers of Communication Port Control (ica, icc) (oca, occ)

NM6403 Communication Ports

The NeuroMatrix® NM6403 has got two communication ports hardware compatible to TMS320C4x. Each of them provides a bidirectional communication interface to other NM6403, or C4x, or external peripherals.

NM6403 contains two direct memory access (DMA) coprocessors to manage communication ports. DMA coprocessors are programmable peripherals that support data transferring through the communication ports in asynchronous mode. The DMA coprocessors perform data transfer to and from anywhere in the processor's memory map. Each DMA coprocessor serves the corresponding communication port for input and output.

Coprocessors work in parallel with the central processor unit (CPU). Moreover, they are independent of each other. For example, it is possible to organize the user program so that the processor could simultaneously receive one portion of data through the zero communication port, execute calculations over the second portion and send the third portion through the first communication port to another processor.

Both communication ports are equivalent. The only difference is in their initial state after RESET, when the 0^{th} communication port is set to the output mode and the 1^{st} communication port – to the input mode.

Each of the communication ports can be switched to any of the two states by a user's program.

The following peripheral control registers manage the DMA coprocessors:

`oca0` and `occ0` – address and counter registers controlling DMA0 in the output mode;

`ica0` and `icc0` – address and counter registers controlling DMA0 in the input mode;

`oca1` and `occ1` – address and counter registers controlling DMA1 in the output mode;

`ica1` and `icc1` – address and counter registers controlling DMA1 in the input mode;

The registers, which names start with `i`, are used in **i**nput mode and those starting with o are used in the **o**utput mode.

Communication Ports State after RESET

After RESET, the communication port 0 turns to the output mode and the port 1 – to the input mode. This information is important when the processor is connected with other processors or external peripherals. When two processors are connected through communication ports, one should make sure that at the moment of power on the communication port of the first processor must turn to the input mode while the port of the other one must be in the output mode.

For example, when two NM6403 are linked through a communication port it is necessary to connect the 0^{th} port of one processor to the 1^{st} port of the other one.

CAUTION

*At reset, port 0 is configured as output port and port 1 is configured as input port. When ports of two processors are interconnected it is necessary to connect the port of one to a port of the other that would be in the **opposite** direction at reset.*

Format of Transmitted Data

The processor NM6403 is able to send/receive **only 64-bit data** through communication ports. To achieve compatibility with C4x, it is necessary to exchange 32-bit data blocks of even size.

NM6403 sends/receives words of data in the order from low bits to high bits, i.e. the low bits are transmitted first and then the high bits.

Thus, when C4x receives data from NM6403, first the low 32-bit word comes, then the high one and on the contrary, the word that came from C4x becomes the low part of a 64-bit word in NM6403.

Receiving Data Through a Communication Port

In the input mode a DMA coprocessor is controlled by registers `ica0` and `icc0` (hereafter all comments are given for the 0^{th} communication port registers, however all of them are also correct for the 1^{st} communication port registers).

The `ica0` register determines the address in the memory of NM6403, starting from which the array of received data will be located.

The `icc0` register is a counter. It specifies the number of 64-bit words to be received. The register counts forward. The communication port stays in the input mode until the counter achieves zero. That's why to receive

the desired number of long words it is necessary to initialize the counter with a negative value. Here is an example that receives 100 long words through the comm. port 0:

```
nobits "data"
    InputBuff: long[100];
end "data";

begin "text"
    ...
    ica0 = InputBuff; // address of the input buffer.
    icc0 = -100;      // start receiving 100 long words.
    ...
end "text";
```

The DMA coprocessor starts receiving data right after the register `icc0` is initialized, and receives data until the counter is equal to 0. The counter `icc0` is increased by 1 after each word is received.

There are two types of a communication port initializing in the input mode:

- direct initialization. This is the way of initialization as it is shown in the example above;
- external initialization. In this case DMA coprocessor, having received the first 64-bit word, stores the low thirty-two bits to ica0 regarding them as the address in memory, and stores the higher bits to icc0 treating them as the number of words to receive.

The bit `CP0I` of the field `CP0 control` of the register `pswr` (bit 22) and the bit `CP1I` of the field `CP1 control` (bit 25) define which type of initialization is selected for the 0^{th} and 1^{st} comm. port respectively (see paragraph 3.2.6 page 3-25).

After system RESET all fields of register pswr are equal to zero, so by default the communication ports are tuned on direct initialization.

The following example shows how to select external initialization of the 0^{th} comm. port in the input mode:

```
begin "text"
    ...
    pswr set 400000h; // the 22-nd bit setting to 1.
    ...
end "text";
```

After the bit defining the communication port initialization type is set, the DMA coprocessor is ready to receive data. As soon as the first portion of data is sent from the opposite side of the link, the DMA starts

receiving data and the first word is used by the DMA to initialize the registers `ica0` and `icc0`.

Sending Data Through a Communication Port

In output mode a DMA coprocessor is controlled by registers `oca0` and `occ0` (hereafter all comments are given for the 0th communication port registers, however all them is also correct for the 1st communication port registers).

The `oca0` register determines the address in the memory of NM6403, starting from which the array of data to send is located.

The `occ0` register is a counter. It specifies the number of 64-bit words to be sent. The register counts forward. The communication port stays in output mode until the counter achieves zero. That's why to send the desired number of long words it is necessary to initialize the counter with a negative value. Here is the example that sends 100 long words through the comm. port 0:

```
nobits "data"
    OutputBuff: long[100];
end "data";
begin "text"
    ...
    oca0 = OutputBuff;  // address of the array to send.
    occ0 = -100;        // start sending 100 long words.
    ...
end "text";
```

The DMA coprocessor starts sending data right after the register `occ0` is initialized, and sends data until the counter is equal to `0`. The counter `occ0` is increased by `1` after each word is sent.

If the comm. port on the opposite side of the link is set to the external initialization mode, the sending processor should first send the address of an array in the destination memory and the size of that array. Only after that it is possible to send the array itself. The array size is given without taking into account the first 64-bit word.

Example:

```
nobits "data"
    OutputBuff: long[101];
end "data";

begin "text"
    ...
    ar0 = OutputBuff;
```

```
    ar4 = 80000000h; // address of an array in the destination
                     // memory.
    gr4 = -100;      // array size.
    [ar0] = ar4, gr4; // initialization parameters are stored into the
                      // first word the array to be sent.
    oca0 = ar0;      // address of an array in the source memory.
    occ0 = -101;     // start sending 101 long words.
    ...
end "text";
```

Waiting for the End of Data Transmittion Through a Communication Port

There are two criteria indicating the end of data transmission through a communication port.

The first criterion is the interrupt generated at the end of transmission. It gives a hundred percent guarantee that the data exchange process is complete.

The second criterion is waiting for the zero value in the counter register. It is a simple way of checking the end of data transmission. It is used in most cases.

Example:

```
begin "text"
    oca0 = OutputBuff;
    occ0 = -100;
    ...
<Loop>                  // label of the waiting loop start.
    gr0 = occ0;         // the contents of the counter is read.
    gr0;                // flags of conditional branch are set.
    if < goto Loop;     // the loop is repeated until the
                        // occ0 counter becomes zero.
    ...
end "text";
```

Note

In case of external initialization the counter register inquiry cannot be used as the method of waiting for the transfer finish. External initialization is an asynchronous process; that is why the receiving processor does not know when the transfer starts. The counter register check for zero can cause an error because at the moment of check it is unknown if the data receiving has ended or it has not started yet. In both cases the counter value is zero. With the external initialization the only way of checking the transfer finish is an interrupt.

Simultaneous Data Transfer Through a Communication Port

The NM6403 DMA coprocessor has two pairs of registers. The first one is used to control data input and the second one for data output. These pairs are independent of each other. So nothing prevents simultaneous initialization of a port for input and for output.

Simultaneous initialization of a communication port for input and for output is possible, but it will not give any profit in the data exchange speed. This possibility is supported for compatibility with C4x.

If a communication port is initialized both for input and for output the device on the opposite size of the link determines which mode to chose. If the transfer in one direction has started it should be finished and only after that the transfer in the opposite direction is possible.

Disabling Data Transfer Through a Communication Port

The processor allows the user to disable/enable data transfer through a communication port. Disabling can be set both before data exchange and during the exchange. In the last case the corresponding communication port will be “frozen” until the disabling bit in the pswr register is cleared.

Setting the bit `ICH0` of the field `CP0 control` in the register `pswr` (bit 21) makes disabled the input through the communication port 0.

Setting the bit `ICH1` of the field `CP1 control` in the register `pswr` (bit 24) makes disabled the input through the communication port 1.

Setting the bit `OCH0` of the field `CP0 control` in the register `pswr` (bit 20) makes disabled the output through the communication port 0.

Setting the bit `OCH1` of the field `CP1 control` in the register `pswr` (bit 23) makes disabled the output through the communication port 1.

Here is an example that shows how to make disable the data input through the communication port 1:

```
begin "text"
    ...
    pswr set 1000000h; // the 24-th bit setting to 1.
    ...
end "text";
```

The communication port becomes enabled for transfers by clearing the corresponding bit of `pswr`. Here is an example that shows how to make enable the data input through the communication port 1:

```
begin "text"
    ...
    pswr clear 1000000h; // clear the 24-th bit of pswr.
    ...
end "text";
```

3.2.3 Register intr

The `intr` is a 32-bit register that is used for the information purposes because its contents can not be changed. It is the only read accessible register. An attempt to write to the register `intr` from assembler will cause a syntactic error.

It contains information on interrupt requests and on DMA requests.

The current value of the `intr` reflects the interrupts request and DMA request status. The internal fields of the register and their default contents are given in Table 3-10. The bits marked with 'X' have no effect.

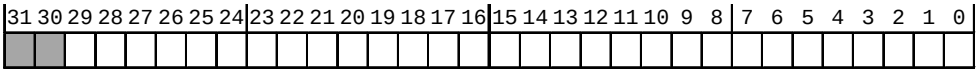
Table 3-10. Register of Interrupt Request and DMA Request (INTR)

BIT	FIELD	BIT NAME	DEFAULT VALUE
31	BS	GBS	0
30		LBS	1
29	DMAR	IC1DR	0
28		IC0DR	0
27		OC1DR	1
26		OC0DR	1
25	PS	CP1S	1
24		CP0S	0
23	AFIFO_VAL	EMPTY	0
24		-	X
21		-	X
20		-	X
19		-	X
18		-	X
			X
17	RAM_VAL	-	X
16		-	X
15		-	X
14		-	X
13		-	X
12	VPF	EMPTA	1
11		FULLA	0
10		EMPTW	1
9		FULLW	0
8	INTREQ	T0R	0
7		SPR	0
6		VPR	0
5		INR	0
4		IC1R	0
3		IC0R	0
2		OC1R	0
1		OC0R	0
0		T1R	0

More detailed description of each field value of `intr` register value is given below.

Field BS

Field `BS` reflects information whether the memory bank0 on the local and global buses belong to the processor at the moment. The bits it occupies in the `intr` register are shaded:

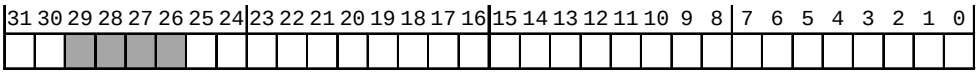


Description of the field bits follows:

BIT	NAME	DESCRIPTION
31	`GBS`	The global bus state: • `0` - the bus belongs to the processor at the moment; • `1` - the bus does not belong to the processor at the moment.
30	`LBS`	The local bus state: • `0` - the bus belongs to the processor at the moment; • `1` - the bus does not belong to the processor at the moment

Field DMAR

Field `DMAR` contains direct memory access requests from the processor communication ports. When the request appears the corresponding bit is set to one; if the request is discarded or there is no request the bit is equal to zero. The bits this field occupies in the `intr` register are shaded:



Description of the field bits follows:

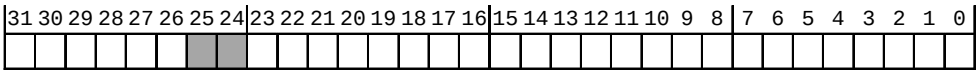
BIT	NAME	DESCRIPTION
29	`IC1DR`	Direct memory access (DMA) request from the data input channel of the port 1.
28	`IC0DR`	DMA request from the data input channel of the port 0.
27	`OC1DR`	DMA request from the data output channel of the port 1.
26	`OC0DR`	DMA request from the data output channel of the port 0.

Note *When a DMA request appears, the corresponding bits of the register* `intr` *are automatically set.*

The bits are automatically cleared when memory access is allowed.

Field PS

Field PS contains information about the current direction of data transfer through the NM6403 communication ports. The bits it occupies in the intr register are shaded:

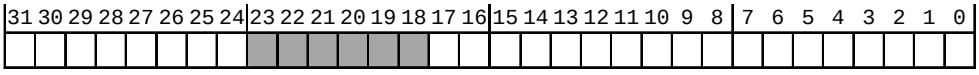


Description of the field bits follows:

BIT	NAME	DESCRIPTION
25	CP1S	Current direction of data transfer through the port 1: • 0 - output mode; • 1 - input mode.
24	CP0S	Current direction of data transfer through the port 0: • 0 - output mode; • 1 - input mode.

Field AFIFO_VAL

Field AFIFO_VAL contains information about the number of 64-bit words in afifo after the current vector instruction execution has completed. The bits it occupies in the intr register are shaded:



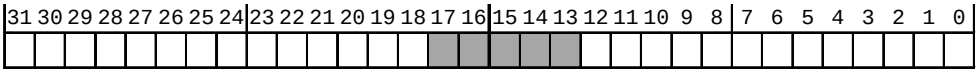
Description of the field bits follows:

BIT	NAME	DESCRIPTION
23	EMPTY	This bit shows whether the afifo is empty or not: • 0 – afifo is empty; • 1 – afifo is not empty and the number of words stored in it is defined by the bits 22 ... 18.
22 21 20 19 18		The number of words stored in afifo within the range from one to thirty-two. Here: 00000 corresponds to one 64-bit word; 00001 corresponds to two 64-bit words; ..... 11111 corresponds to thirty-two 64-bit words.

Field RAM_VAL

Field RAM_VAL contains information about the number of 64-bit words in the ram (internal RAM FIFO of the Vector Unit) after the current vector

instruction execution has been completed. The bits it occupies in the `intr` register are shaded:

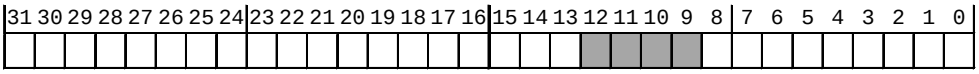


Description of the field bits follows:

BIT	DESCRIPTION
17 16 15 14 13	The number of words in RAM, in the range from 1 to 32. Here: `00000` corresponds to one 64-bit word; `00001` corresponds to two 64-bit words; `.....` `11111` corresponds to thirty-two 64-bit words.

Field VPF

Field `VPF` contains information about the filling degree of the buffers `WFIFO` and `AFIFO` of the Vector Unit. The bits it occupies in the `intr` register are shaded:

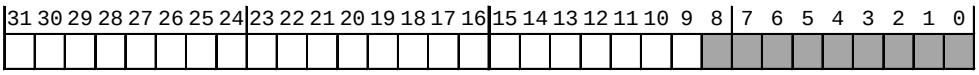


Description of the field bits follows:

BIT	NAME	DESCRIPTION
12	`EMPTA`	Flag of data presence in the `AFIFO` buffer: • `0` - `AFIFO` contains data; • `1` - `AFIFO` is empty.
11	`FULLA`	Flag of the filling degree of the `AFIFO` buffer: • `0` - `AFIFO` is semi filled; • `1` - `AFIFO` is filled up.
10	`EMPTW`	Flag of presence of data in the `WFIFO` buffer: • `0` - `WFIFO` contains data; • `1` - `WFIFO` is empty.
9	`FULLW`	Flag of the fill degree of the `WFIFO` buffer: • `0` – `wfifo` is not full; • `1` – `wfifo` is full.

Field INTEREQ

Field `INTEREQ` reflects interrupts request status. The bits it occupies in the `intr` register are shaded:



Description of the field bits follows:

BIT	NAME	DESCRIPTION
8	T0R	Interrupt request generated by the timer t0.
7	SPR	Interrupt request generated by the RISC-core when an overflow occurs during the scalar arithmetic operation.
6	VPR	Interrupt request generated by an incorrect vector instruction.
5	INR	External (user) interrupt request.
4	IC1R	Interrupt request generated by the communication port 1 in the end of data receiving.
3	IC0R	Interrupt request generated by the communication port 0 in the end of data receiving.
2	OC1R	Interrupt request generated by the communication port 1 in the end of data sending.
1	OC0R	Interrupt request generated by the communication port 0 in the end of data sending.
0	T1R	Interrupt request generated by the timer t1.

Note *When an interrupt request occurs, the corresponding bit of the* `intr` *is automatically set.*
The bit is automatically cleared when the interrupt service routine starts executing.

3.2.4 Register lmicr

The `lmicr` is a 32-bit register that can be programmed to control local memory interface by defining the

- page size used for the two memory banks at each port;
- address ranges over which memory banks are active;
- wait states;
- other operations that control the memory interface.

Register `lmicr` is accessible both for reading and for writing.

The `lmicr` must be set up before any application runs. Usually the loader program, which is included to the load and exchange library, sets it up.

Note *The user can change the contents of* `lmicr` *register himself. However, it is necessary to be careful because an incorrectly configured register will not allow the processor to correctly access global memory. It can cause*

the program halt or incorrect execution.

Register `lmicr` has absolutely the same structure as the register `gmicr`. Table 3-3 describes the fields of this register.

Since the local bus has its own range of addresses, its division into the memory banks is determined as shown in Table 3-11.

Table 3-11. Division of Local Bus Address Space into Banks 0/1 According to BOUND

BOUND	SIZE OF BANK 0 (64 BITS)	ADDRESS SPACE OF BANK 0	SIZE OF BANK 1 (64 BITS)	ADDRESS SPACE OF BANK 1
0000	2^{15} = 32K	00000000 - 0000FFFF	32K	00010000 - 7FFFFFFF
0001	2^{16} = 64K	00000000 - 0001FFFF	64K	00020000 - 7FFFFFFF
0010	2^{17} = 128K	00000000 - 0003FFFF	128K	00040000 - 7FFFFFFF
0011	2^{18} = 256K	00000000 - 0007FFFF	256K	00080000 - 7FFFFFFF
0100	2^{19} = 512K	00000000 - 000FFFFF	512K	00100000 - 7FFFFFFF
0101	2^{20} = 1M	00000000 - 001FFFFF	1M	00200000 - 7FFFFFFF
0110	2^{21} = 2M	00000000 - 003FFFFF	2M	00400000 - 7FFFFFFF
0111	2^{22} = 4M	00000000 - 007FFFFF	4M	00800000 - 7FFFFFFF
1000	2^{23} = 8M	00000000 - 00FFFFFF	8M	01000000 - 7FFFFFFF
1001	2^{24} = 16M	00000000 - 01FFFFFF	16M	02000000 - 7FFFFFFF
1010	2^{25} = 32M	00000000 - 03FFFFFF	32M	04000000 - 7FFFFFFF
1011	2^{26} = 64M	00000000 - 07FFFFFF	64M	08000000 - 7FFFFFFF
1100	2^{27} = 128M	00000000 - 0FFFFFFF	128M	10000000 - 7FFFFFFF
1101	2^{28} = 256M	00000000 - 1FFFFFFF	256M	20000000 - 7FFFFFFF
1110	2^{29} = 512M	00000000 - 3FFFFFFF	512M	40000000 - 7FFFFFFF
1111	2^{30} = 1G	00000000 - 7FFFFFFF	0	-

Refer to the `gmicr` description to get more information on the `lmicr` (see 3.2.1 on page 3-6).

3.2.5 Register pc

The program counter (`pc`) is a 32-bit register containing the address of the next instruction to be fetched.

The processor always fetches a 64-bit word of instructions. This word contains either one long instruction or two short ones. A long instruction is always located at an even address.

The buffer of pre-fetched instructions strongly influences the `pc` register value. Different instruction length makes impossible the accurate

prediction of the pc value, whether it is equal to the current address plus two or plus four.

Due to an ambiguity of the contents of the pc register at every certain moment of time, the assembler does not support the instruction counter addressing.

3.2.6 Register pswr

The pswr register (word of state) contains global information relating to the state of the processor. Typically, operations set the condition flags of the pswr according to whether the result is zero, negative, etc.

After the system RESET it is filled with zero.

Some fields of the register are changed automatically by the processor some fields are programmable. To set or clear the desired fields of the pswr the special processor instructions are intended. Table 3-12 specifies the fields of the pswr.

Table 3-12. Processor State Word (pswr)

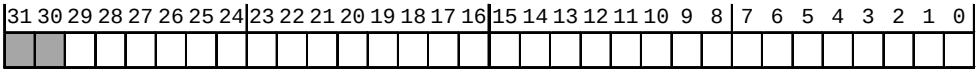
BIT	FIELD	BIT NAME
31 30	BC	GBRE LBRE
29 28 27 26	TIMER pin control	TEN TM2 TM1 TM0
25 24 23	CP1 control	CP1I ICH1 OCH1
22 21 20	CP0 control	CP0I ICH0 OCH0
19 18	T0C	TM0 TE0
17 16	T1C	TM1 TE1
15 14	FCL	WFCL AFCL
13 12 11 10 9 8	INTERRUPT MASKS	T0M SPM VPM INTM IC1M IC0M

7 6 5 4		OC1M OC0M T1M STM
3 2 1 0	FLAGS	N Z V C

The detailed description of each field of the pswr is given below.

Field BC

Field BC specifies access to shared buses. The possible configurations of shared memory interface are listed in the document **NeuroMatrix® NM6403. User's Guide**. The field handles access to the shared bus from the other processor. If the processor owns the shared memory or if it is the common memory, the field allows or does not allow the processor to carry control over the bus to the other processor. The bits occupied by the BC field in the register pswr are shaded:

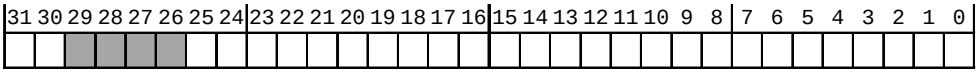


Description of the field bits follows:

BIT	NAME	DESCRIPTION
31	GBRE	Global Bus Request Enable. This bit enables/disables carry of control over the global bus by request from an external device: • 0 - enables carrying of control; • 1 - disables carrying of control.
30	LBRE	Local Bus Request Enable. This bit enables/disables carry of control over the local bus by request from an external device: • 0 - carrying of control is enabled; • 1 - carrying of control is disabled.

Field TIMER pin control

The TIMER pin control field defines the state and behavior of the TIMER pin. The bits occupied by the field in the register pswr are shaded:



Description of the field bits follows:

BIT	NAME	DESCRIPTION
29	TEN	This bit enables/disables output mode of the TIMER pin.

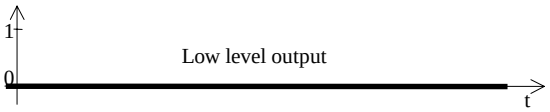
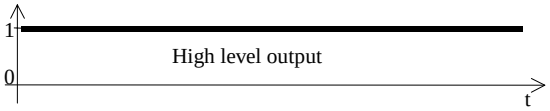
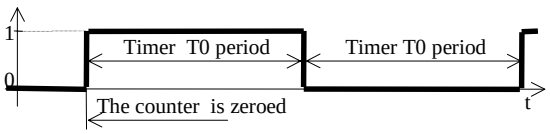
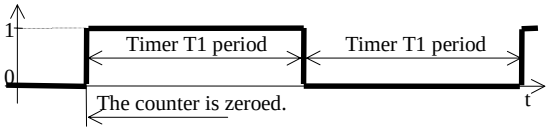
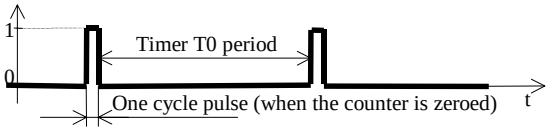
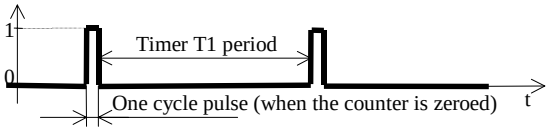
		• 0 - output mode is disabled; • 1 - output mode is enabled.
28	TM2	This field together with the fields TM1 and TM0 controls the output signal configuration on the TIMER pin.
27	TM1	This field together with the fields TM2 and TM0 controls the output signal configuration on the TIMER pin.
26	TM0	This field together with the fields TM2 and TM1 controls the output signal configuration on the TIMER pin.

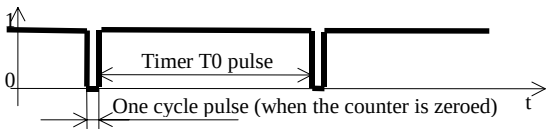
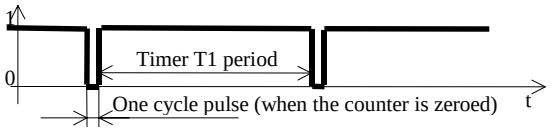
If the bit TEN = 0 none of combinations of bits TM2, TM1 and TM0 will cause change of the TIMER pin state.

In case TEN = 1 the fields TM2, TM1 and TM0 are used to define the shape of the output signal on the TIMER pin. The possible signal shapes are listed in Table 3-13.

Apart from the high level signal and the low level signal, all other signals depend on timers T0/T1 state.

Table 3-13. Output Signals on TIMER pin

TM2	TM1	TM0	STATE OF TIMER PIN
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	

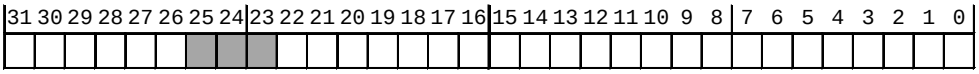
1	1	0	
1	1	1	

In case the timer T0 or T1 works in the loop mode, the presented time diagrams will have periodical character too.

If the timer is programmed for single count, then after the counter achieves zero the timer will be held. Depending on the preset `TM2`, `TM1` and `TM0` either a single pulse with the following return to the basic output level (as in the last four cases), or the output level change takes place.

Field CP1 control

Field `CP1` control defines an operating mode of the 1st communication port. The bits it occupies in the pswr register are shaded:

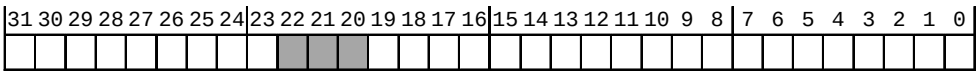


Description of the field bits follows:

BIT	NAME	DESCRIPTION
25	`CP1I`	Bit of external initialization of the input channel of the 1st comm. port: • `0` - direct initialization mode of the input channel. This means that the address and size of the array to be received are prescribed before starting data receiving; • `1` - external initialization of input channel. This means that initialization of the DMA is executed according to the first received word. The first 64-bit word contains the address of the array to be received (low word) and its size (high word).
24	`ICH1`	Bit of enable/disable of data input through the 1st comm. port: • `0` - data input is allowed; • `1` - data input is not allowed.
23	`OCH1`	Bit of enable/disable of data output through the 1st comm. port: • `0` - data output is allowed; • `1` - data output is not allowed.

Field CP0 control

Field CP0 control defines an operating mode of the 0th communication port. The bits it occupies in the pswr register are shaded:



Description of the field bits values follows:

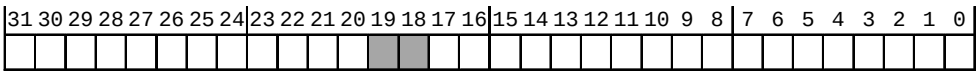
BIT	NAME	DESCRIPTION
22	CP0I	Bit of external initialization of the input channel of the 0th comm. port: • 0 - direct initialization mode of the input channel. This means that the address and size of the array to be received are prescribed before starting data receiving; • 1 - external initialization of input channel. This means that initialization of the DMA is executed according to the first received word. The first 64-bit word contains the address of the array to be received (low word) and its size (high word).
21	ICH0	Bit of enable/disable of data input through the 0th comm. port: • 0 - data input is allowed; • 1 - data input is not allowed.
20	OCH0	Bit of enable/disable of data output through the 0th comm. port: • 0 - data output is allowed; • 1 - data output is not allowed.

Note

The fields CP0 and CP1 control can be modified only by the instructions pswr set or pswr clear (see paragraph 5.1.8 on page 5-17).

Field T0C

Field T0C controls the timer T0. The bits it occupies in the pswr register are shaded:



Description of the field bits follows:

BIT	NAME	DESCRIPTION

19	TM0	Defines an operating mode of the T0.
18	TE0	Enables/disables work of the T0.

Different combinations of bits in this field allow the user to select the timer mode from the following list:

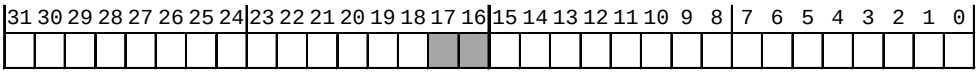
Table 3-14. Timer T0(T1) Operation Modes

TM0	TE0	DESCRIPTION
0	0	All timer operations are held. The contents of the t0 register do not affect the timer.
0	1	When the counter t0 becomes zero, the timer stops and the interrupt request is set. This is the single count mode.
1	x	The timer works in the loop mode. When the counter becomes zero, the interrupt request is set, the counter recovers the initial state and the new period starts. The bit marked with ‘x’ has no effect.

For more details about timers T0 and T1 work see 3.2.7on page 3-32.

Field T1C

The field T1C controls the timer T1. The bits it occupies in the pswr register are shaded:



Description of the field bits follows:

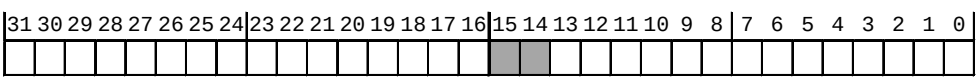
BIT	NAME	DESCRIPTION
17	TM1	Defines an operating mode of the T1.
16	TE1	Enables/disables work of the T1.

Combinations of the bits TM1 and TE1 determine the timer T1 work in accordance with Table 3-14.

For more details about timers T0 and T1 see 3.2.7 on page 3-32.

Field FCL

The field FCL allows the user to clear the current contents of the internal buffers AFIFO and WFIFO. The bits it occupies in the pswr register are shaded:

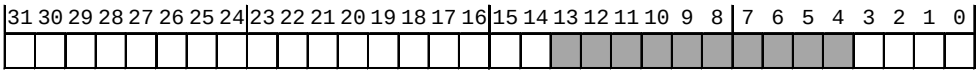


Description of the field bits follows:

BIT	NAME	DESCRIPTION
15	WFCL	Bit of WFIFO clearing enable/disable: • 0 – clearing is not allowed; • 1 – clearing is allowed. In this case the processor sends the signal of the buffer clearing every clock cycle until the user clears the bit.
14	AFCL	Bit of AFIFO clearing enable/disable: • 0 – clearing is not allowed; • 1 – clearing is allowed. Here the processor sends the signal of the buffer clearing every clock cycle until the user clears the bit.

Field INTERRUPT MASKS

Field INTERRUPT MASKS is used to enable/disable the processor interrupts. When the related bit is set, the mask is active, and the interrupt is disabled and vise versa. The bits the field occupies in the pswr register are shaded:



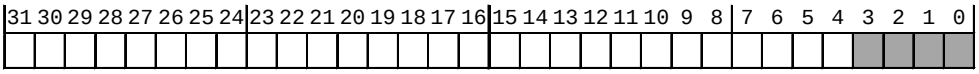
Description of the field bits follows:

BIT	NAME	DESCRIPTION
13	T0M	The timer T0 interrupt mask. An interrupt is sent to the processor when the timer counter t0 achieves zero.
12	SPM	The scalar arithmetic operation overflow interrupt mask.
11	VPM	The incorrect vector instruction interrupt mask.
10	INTM	The external (user) interrupt mask.
9	IC1M	The finish of data input through the 1st communication port. An interrupt is sent when the input channel counter achieves zero.
8	IC0M	The finish of data input through the 0th communication port. An interrupt is sent when the input channel counter achieves zero.
7	OC1M	The finish of data output through the 1st communication port. An interrupt is sent when the output channel counter achieves zero.
6	OC0M	The finish of data output through the 0th communication port. An interrupt is sent when the output channel

		counter achieves zero.
5	T1M	The timer T1 interrupt mask. An interrupt is sent when the timer counter t1 achieves zero.
4	ST	Single step/debug interrupt mask.

Field FLAGS

Field FLAGS is a set of condition flags that are set or cleared by the arithmetic operations on the RISC-core. They provide information about the properties of the result or output of arithmetic/logical/shift operations. The bits the field occupies in the pswr register are shaded:



Description of the field bits follows:

BIT	NAME	DESCRIPTION
3	N	**Negative Condition Flag.** 1 if a negative result of an arithmetic/logical/shift operation on the scalar processor is generated, 0 otherwise.
2	Z	**Zero Condition Flag.** 1 if a zero result of an arithmetic/logical/shift operation on the scalar processor is generated, 0 otherwise.
1	V	**Overflow Condition Flag.** 1 if an integer overflow occurs in the result of an arithmetic/logical/shift operation on the scalar processor, 0 otherwise.
0	C	**Carry Flag.** 1 if a carry or borrow occurs in the result of an arithmetic operation on the scalar processor, otherwise 0. For shift operations, this flag is set to the value of the last bit shifted out.

3.2.7 Timer Counters t0 and t1

The 32-bit timer counter registers increment with each cycle of the related timer clock. When the timer counter becomes zero, the interrupt is sent to the processor.

When the timer is in the loop mode the counter is filled with the initial value whenever it achieves zero. When the timer is in the single count mode the counter achieves zero and stops.

Both timer counters t0 and t1 are read/write accessible registers.

The register t0 is considered a system register and it is usually used for regeneration of dynamic memory. In this case a user program cannot use it. If the processor works only with static memory, the user program can use both registers.

The registers `t0` and `t1` are controlled independently, each of them is controlled by the appropriate field in the register pswr. Depending on the contents of those fields, the timer can do the following:

- generate an interrupt when passing through zero;
- stop counting when achieving zero;
- having achieved zero, initialize itself with the period variable and start counting a new loop.

The timer makes a full run through the entire count range (from `0` to `0x`FFFFFFFF) for:

- 107,37 sec at 40 MHz;
- 85,9 sec at 50 MHz.

Here is an example of work with a timer register:

```
begin "text"
     t1 = 0;    // the timer counter t1 is zeroed
     ...
     // calculations are made.
     ...
     gr7 = t1;  //  the time of calculations measured in the processor
                // clock cycles is recorded to gr7.
end "text";
```

Note *After a value is stored into the register t0(t1) 6 more cycles pass before a new counter value will become active.*

3.3 Vector Register File

The vector register file provides control of the Vector Unit. The registers are used to handle data stream coming through the Vector Unit during calculations, to store temporary data, to load weights to the Active Matrix and so forth.

Vector registers are divided into two groups. The first one contains the Vector Unit control registers. They define the configuration of the Active and the Shadow Matrixes, the Vector ALU and so forth.

The second group includes so-called “registers-containers” that are the Vector Unit internal memory blocks based on the FIFO principle. All registers-containers are up to thirty two 64-bit words long.

Table 3-15. Vector Register File of NeuroMatrix NM6403

REGISTER	DESCRIPTION	NOTE
`f1cr,f2cr`	Activation function control registers.	64 bits. Write accessible.
`nb1`	This register divides the Shadow Matrix into columns.	64 bits. Write accessible.
`nb2`	This register divides the Active Matrix into columns and the Vector ALU input into elements.	64 bits. Not accessible.
`sb`	This register is a superposition of the `sb1` and `sb2` registers.	64 bits. Write accessible.
`sb1`	This register divides the Shadow Matrix into rows.	32 bits. Not accessible.
`sb2`	This register divides the Active Matrix into rows.	32 bits. Not accessible.
`vr`	Bias register.	64 bits. Write accessible.
`afifo`	Register-container that is used to store the result of every vector instruction.	32x64 bits.
`data`	Pseudo register-container that is used to manipulate data loading from external memory and redirect them to the Active Matrix or the Vector ALU. It is mapped to input data buses of NM6403.	32x64 bits.
`ram`	Register-container that is used to store and reuse the same data block in calculations.	32x64 bits.
`wfifo`	Register-container that is used to store weights those are then loaded to the Shadow Matrix.	32x64 bits.

3.3.1 Registers f1cr and f2cr

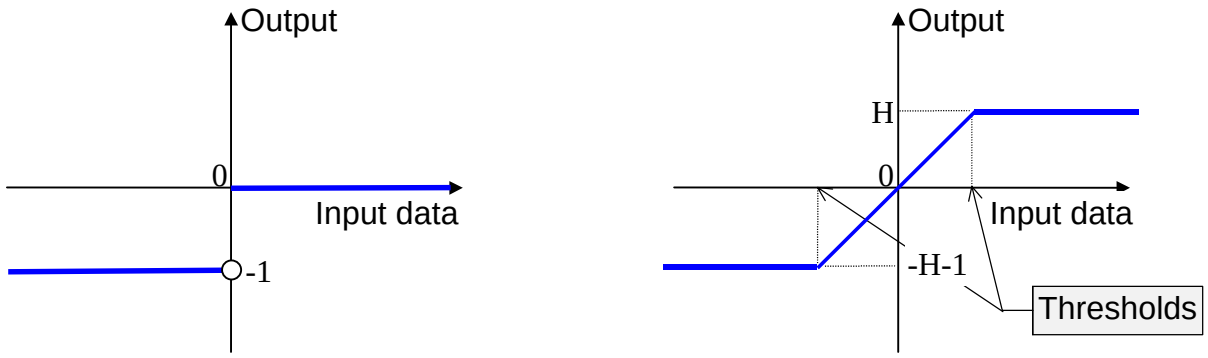
The `f1cr` and `f2cr` are 64-bit write accessible registers. They are used to control configuration of operands when applying piecewise-linear transform to input data in the Activation Units (see 1.5.5 on page 1-19).

The data coming through the input channels **X** and **Y** of the Vector Unit can be transformed by piecewise-linear functions called activation functions. It happens if the reserved word '`activate`' is used in a vector instruction.

There are two types of activation functions:

- threshold function (see Figure 3-1a);
- saturation function (see Figure 3-1b).

Figure 3-1. Types of Embedded Activation Functions



a) Threshold function *b) Saturation function*

The type of the activation function to be used is completely defined by type of operation in the right part of a vector instruction. The threshold function is used only with logical operations, while the saturation function only with arithmetic functions. The entire sets of vector logical and arithmetic operations are given in paragraph 5.1.12 on page 5-24 and 5.1.11 on page 5-23 respectively.

The `f1cr` and `f2cr` play the following role in the activation transform:

configure the split of 64-bit words of input data into elements;

define the threshold *H* of the saturation function.

The `f1cr` register specifies configuration of the input channel **X** when the data are coming through the Activation Unit, and the `f2cr` specifies configuration of the input channel **Y**.

The `f1cr` and `f2cr` registers split 64-bit words of input data into elements. An activation function is executed over every element, i.e. all elements composing a long word are transformed simultaneously and independently.

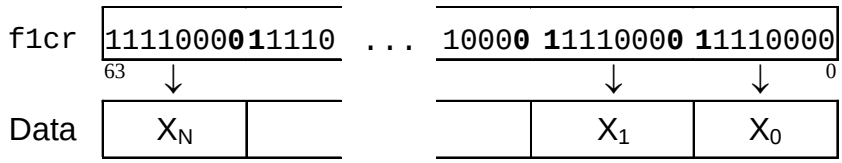
The `f1cr` and `f2cr` registers specify configuration of data coming through the Activation Units, while the registers `sb2` and `nb2` configure the Active Matrix and the Vector ALU. The configurations may differ, but in most cases they are the same.

How f1cr (f2cr) Splits Input Data

The f1cr register will be used in the following description, but everything said below is also correct for `f2cr` unless otherwise arranged.

The boundary between elements is specified in the f1cr register when in the pair of the bits the right bit is '`1`' while the left bit is '`0`' (see Figure 3-2). Any other combinations of the neighbor bits do not affect the boundaries.

Figure 3-2. Division of a 64-bit Word into Elements by f1cr (f2cr)



The smallest size of an element is two bits (the low bit is '0' and the high bit is '1').

The sequence of zeros and ones in f1cr is arbitrary. That is why a long word of data can be split into an arbitrary number of elements (within the range from 1 to 32) of an arbitrary length with the total number of bits equal to 64.

Use of f1cr(f2cr) in Saturation Function (Arithmetic Activation)

The data words processing in the Vector Unit are 64 bits long. The register f1cr has the same length. That is why every bit of f1cr corresponds to the bit of data.

When arithmetic activation is applied to input data by means of the saturation function (see Figure 3-1b), the bit values of f1cr corresponding to inner bits of data elements specify the threshold.

A packed word of input data is split into elements. The most significant bit (MSB) of every element always corresponds to the bit of f1cr equal to '1'. (see Figure 3-2). As illustrated in Table 3-16 the number of bits '1' located to the right of the MSB specify the threshold.

Table 3-16. List of Thresholds for 8-bit Data

BITS OF F1CR	THRESHOLD
10000000_2	127
11000000_2	63
11100000_2	31
11110000_2	15
11111000_2	7
11111100_2	3
11111110_2	1

The possible threshold values are a power of 2 minus one ($H = 2^n - 1$).

The next examples explain mechanism of the saturation function application to elements of input data. The 8-bit element is considered. The threshold value is equal to 31.

The first example shows how the saturation function affects the element of data, which is non-negative, but less than or equal to the threshold. Three most significant bits of all values from the described range are '0':

f1cr: ...**111**00000... ← *the upper threshold is '31'(0x1F)*

data field: `...000`10110... ← *value of the element is '22'(0x16)*

after activation: ...00010110... ← *value of the element is '22'(0x16)*

The Activation Unit checks three most significant bits of the element. They all are '0'. In this case, the value of the element is not changed.

The second example shows how the saturation function affects the element of data, which is negative, but greater than or equal to the negative threshold. Three most significant bits of all values from the described range are '1':

f1cr: ...**111**00000...← *the negative threshold is '-32'(0xE0*)

data field: ...**111**10110... ← *the element value is '-10'(0xF6)*

after activation:...11110110... ← *the element value is '-10'(0xF6)*

The Activation Unit checks three most significant bits of the element. All they are '1'. In this case the value of the element is not changed.

The third example shows how the saturation function transforms the element of data, which is positive and exceeds the threshold. Three most significant bits of all values from the described range are not the same.

f1cr: ...**111**00000... ← *the positive threshold is '31'(0x1F)*

data field: ...**010**10110... ← *value of the element is '86'(0x56)*

after activation: ...00011111... ← *value of the element is '31'(0x1F)*

The Activation Unit checks three most significant bits of the element. One or two of them are '1'. The MSB is '0', this means the value is positive, but it exceeds the threshold. In this case the contents of the element is saturated, i.e. it is substituted by the threshold value.

The last example shows how the saturation function transforms the element of data, which is negative and exceeds the negative threshold. Three most significant bits of all values from the described range are not the same.

f1cr: ...**111**00000... ← *the low threshold is '-32'(0xE0)*

data field: ...**110**10110... ← *value of the element is '-42'(0xD6)*

after activation: ...11100000... ← *value of the element is '-32'(0xE0)*

The Activation Unit checks three most significant bits of the element. One or two of them are '0'. The MSB is '1', this means the value is negative, but it exceeds the negative threshold. In this case the contents of the element is saturated, i.e. it is substituted by the negative threshold value.

Thus, if the bits of an element corresponding to bits '1' of the f1cr register are:

the same - the content of the element is not changed by the arithmetic activation.

not the same, and MSB is '0' - the contents of the element is substituted by the positive threshold.

not the same, and MSB is '1' - the contents of the element is substituted by the negative threshold.

Use of f1cr(f2cr) in Threshold Function (Logical Activation)

Logical activation is applied to input data by means of the threshold function (see Figure 3-1a). A packed word of input data is split into elements. As can be seen in Figure 3-2, the most significant bit (MSB) of every element always corresponds to the bit of `f1cr` equal to '1'.

The MSB of every element specifies the behavior of the threshold function. If the MSB of an element is '0', the element is non-negative. The threshold function substitutes the element value for zero. It looks like spread of '0' along the element up to the list significant bit.

If the MSB of an element is '1', the element is negative. The threshold function substitutes the element value for -1. It looks like spread of '1' along the element up to the list significant bit.

The next examples explain mechanism of the threshold function application to elements of input data. The 8-bit element is considered.

The first example shows how the threshold function affects the element of data, which is non-negative. The most significant bit of all non-negative elements is '0':

`f1cr:` `...0|10000000|1...`

data field: `.....00010110...` ← *input value is 22(0x16)*

after activation: `.....00000000...` ← *result value is 0*

The data element value is equal to 22 (positive). The result of processing by the logical activation function is zero.

The second example shows how the threshold function affects the element of data, which is negative. The most significant bit of all non-negative elements is '1':

`f1cr:` `...0|10000000|1...`

data field: `.....10010110...` ← *input value is -106(0x96)*

after activation: `.....11111111...` ← *result value is -1(0xFF)*

The data element value is equal to -106 (negative). The result of processing by the logical activation function is equal to -1.

Thus, the logical activation function (threshold function) substitutes non-negative values of data elements for 0, and negative values for -1.

Load the f1cr (f2cr) Register

The f1cr is a 64-bit register. The instruction set of NeuroMatrix® NM6403 does not allow loading a 64-bit constant to registers directly. However there are several indirect methods of initializing the `f1cr` register.

Method 1. Piecemeal Load.

The NM6403 processor allows separate access to the high and low parts of the `f1cr` register. The high and low parts of the `f1cr` have special notation as follows:

- `f1crh` - the high 32-bit part of the `f1cr` register;
- `f1crl` - the low 32-bit part of the `f1cr` register.

Here is an example of the `f1cr` register piecemeal load:

```
f1crl = 80808080h;    // load the low part of f1cr.
f1crh = 40404040h;    // load the high part f1cr.
```

As a result of the command the 64-bit constant `40404040808080hl` will be loaded to the `f1cr` register.

Method 2. Load the same constant to the both parts of the register.

If the same constant is to be loaded to the low and the high part of the register, the following instruction can be used:

```
f1cr = 80808080h; // load the same constant to the high
                  // and to the low part of f1cr.
```

The processor automatically recognizes access to 64-bit register and copies the same constant to the both parts of the register. As a result of the command the 64-bit constant `80808080808080hl` will be loaded to the `f1cr` register.

Method 3. Load the contents of the memory location.

The register `f1cr` can be initialized with a 64-bit constant located in memory in the following manner:

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

...

begin ".text"
    ...
    f1cr = [MyF1CR]; // load 64-bit constant from memory to f1cr.
    ...
end ".text";
```

The processor automatically recognizes access to 64-bit register and loads a 64-bit constant from memory. As a result of the command the 64-bit constant `0123456789ABCDEFhl` will be loaded to the `f1cr` register.

Table 3-17 presents the most frequently used constants for the `f1cr` register initialization. It is supposed that a 32-bit constant is loaded to `f1cr` as described in the method two, and a 64-bit constant is read from memory.

Table 3-17. Constants Frequently Used for the f1cr(f2cr) Register Initialization

ELEMENT SIZE	NUMBER OF ELEMENTS IN 64-BIT WORD	CONSTANT TO LOAD IN F1CR (F2CR)
64 bits	1	`f1crh = 80000000h`
32 bits	2	`80000000h`
21 bits	3	`4000020000100000hl`
16 bits	4	`80008000h`
10 bits	6	`0802008020080200hl`
8 bits	8	`80808080h`
4 bits	16	`88888888h`
2 bits	32	`AAAAAAAAh`

3.3.2 Register nb1(nb2)

The `nb1` is 64-bit write accessible register. It belongs to the Shadow Matrix. It controls division of the Shadow Matrix into columns.

The `nb2` is 64-bit not accessible register. It belongs to the Active Matrix. It controls:

- division of the Active Matrix into columns;
- division of the Vector ALU input data into elements.

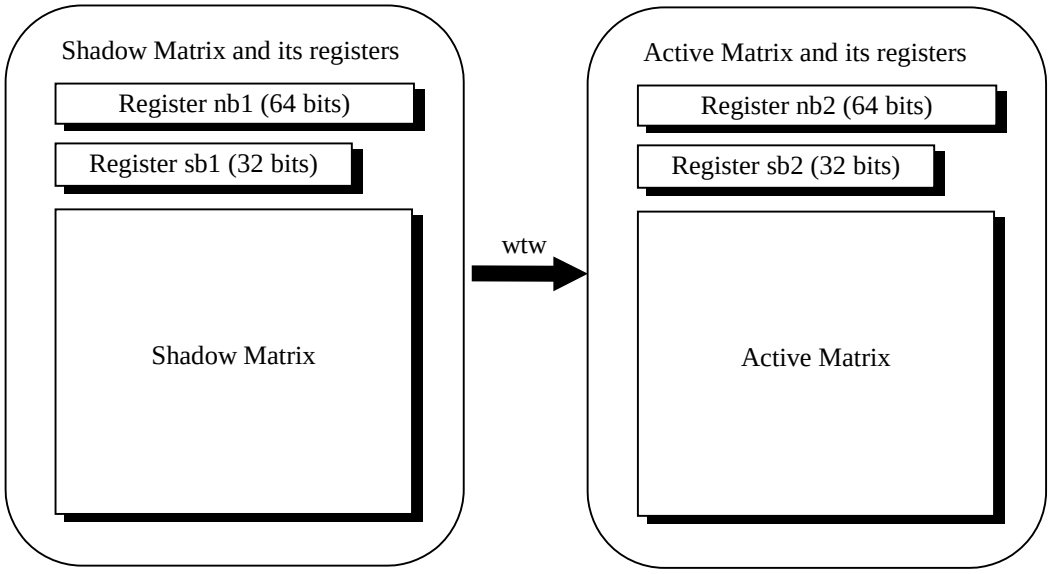
The Vector Unit incorporates two operation matrixes: the Active and the Shadow. The Active Matrix participates in calculations while on the background the Shadow Matrix is used to prepare the next portion of weights that will be used on the next step of calculations. This architectural features allow simultaneous calculations and preparing of new weights. When the new weights are prepared, it takes one clock cycle to copy the contents of the Shadow Matrix to the Active Matrix. The vector instruction `wtw` is used for this purpose.

As shown in Figure 3-3, each Matrix is associated with a pair of registers (please, do not mix it up with a register pair):

`nb[1,2]` - determines the Matrix division into columns;

`sb[1,2]` - determines the Matrix division into rows.

Figure 3-3. The Shadow and the Active Matrixes of the Vector Unit



The registers associated with the Shadow Matrix are `nb1` and `sb1`, and registers of the Active Matrix are `nb2` and `sb2`.

Description of registers `sb1` and `sb2` is given later in paragraph 3.3.3 on page 3-44. Hereafter the registers `nb1` and `nb2` are described.

The nb2 register is not accessible directly from assembler. For indirect access to `nb2` the `nb1` register is intended. In order to modify `nb2`, it is necessary to make the following steps:

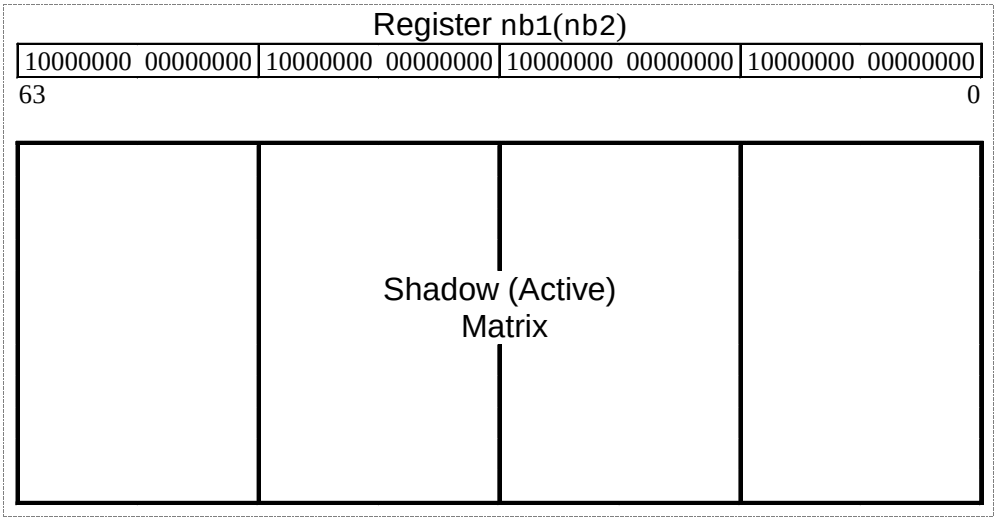
- Load a 64-bit constant to the `nb1` register;
- Execute the `wtw` instruction to copy the contents of the Shadow Matrix and the associated registers to the Active Matrix and its registers. It will change the contents of `nb2`.

Use of nb1(nb2)

The `nb1(nb2)` register controls division of the Shadow (Active) Matrix of the Vector Unit into columns. The `nb2` register also specifies split of packed data words when the data are processed on the Vector ALU.

Then using `nb1(nb2)` for boundaries specification the bits that are set to '`1`' are treated as the most significant bits (MSB) of elements (see Figure 3-4).

Figure 3-4. Split of the Shadow(Active) Matrix into Columns by the nb1(nb2) Register



The minimum column width is one bit. The nb1(nb2) register allows the user to split the Matrix into up to 64 columns. Width of columns is ranged from one to sixty-four bits.

As the output of the Active Matrix is the input of the Vector ALU, the nb2 register specifies configuration of the input **X** of the Vector ALU, too. According to the architecture of the Vector ALU the inputs **X** and **Y** must have the same configuration. So, the nb2 register specifies configuration of both input channels entering the Vector ALU.

It is possible to divide the Matrix into columns (the Vector ALU inputs into elements) of different width. The example below shows the division of 64-bit words into the elements in the following manner: 32-bit|16-bit|16-bit:

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

begin "text"
    nb1 = [NB1]; // load a 64-bit constant from memory to nb1.
    . . .
    wtw;         // copy the contents of nb1 to nb2.
    . . .
end "text";
```

If it isn't necessary to split the Matrix into columns zero may be written to register nb1. The processor sets the boundary of the elements at the 63rd bit.

Load the nb1Register

The nb1 is a 64-bit register. The instruction set of NeuroMatrix® NM6403 does not allow loading a 64-bit constant to registers directly.

However there are several indirect methods of initializing the nb1 register.

Method 1. Partial Load.

The NM6403 processor allows separate access to the high and low parts of the nb1 register. The high and low parts of the nb1 have special notation as follows:

- nb1h - the high 32-bit part of the nb1 register;
- nb1l - the low 32-bit part of the nb1 register.

Here is an example of the nb1 register partial load:

```
nb1l = 80808080h;    // load the low part of nb1.
nb1h = 40404040h;    // load the high part nb1.
```

As a result of the command the 64-bit constant 40404040808080hl will be loaded to the nb1 register.

Method 2. Load the same constant to the both parts of the register.

If the same constant is to be loaded to the low and the high part of the register, the following instruction can be used:

```
nb1 = 80808080h; // load the same constant to the high
                 // and to the low part of nb1.
```

The processor automatically recognizes access to a 64-bit register and copies the same constant to the both parts of the register. As a result of the command the 64-bit constant 80808080808080hl will be loaded to the nb1 register.

Method 3. Load the contents of the memory location.

The register nb1 can be initialized with a 64-bit constant located in memory in the following manner:

```
data ".data"
    MyNB1: long = 0123456789ABCDEFhl;
    ...
end ".data";

...
begin ".text"
    ...
    f1cr = [MyNB1]; // load 64-bit constant from memory to nb1.
    ...
end ".text";
```

The processor automatically recognizes access to 64-bit register and loads a 64-bit constant from memory. As a result of the command the 64-bit constant 0123456789ABCDEFhl will be loaded to the nb1 register.

Table 3-18 presents the most frequently used constants for the nb1 register initialization. It is supposed that a 32-bit constant is loaded to nb1 as described in the method two, and a 64-bit constant is read from memory.

Table 3-18. The Constants Frequently Used for the nb1 Register Initialization

ELEMENT SIZE	NUMBER OF ELEMENTS	CONSTANT TO LOAD IN NB1
64 bits	1	0
32 bits	2	80000000h
21 bits	3	4000020000100000hl
16 bits	4	80008000h
10 bits	6	0802008020080200hl
8 bits	8	80808080h
4 bits	16	88888888h
2 bits	32	AAAAAAAAh
1 bits	64	FFFFFFFFh

3.3.3 Register sb (sb1 and sb2)

The sb is a 64-bit write accessible register. It is used to split the Shadow and the Active Matrix of the Vector Unit into rows.

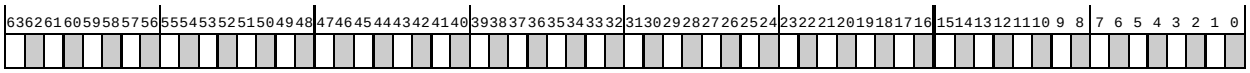
The sb register is not accessible for reading. The assembler treats an attempt to read data from sb as syntax error.

Structure of sb

The sb register is a superposition of the sb1 and sb2 registers. The 32-bit sb1 and sb2 registers control split of the Shadow/Active Matrix into rows. The sb1 register is associated with the Shadow Matrix, while sb2 is an attribute of the Active Matrix. The sb1 and sb2 registers are not directly accessible from a user program. Direct access to sb1 and sb2 is impossible due to the NM6403 architectural restrictions. The sb register is used to modify the contents of the sb1 and sb2 registers.

The sb register looks like a “tooth-comb”. Each bit of sb belongs either to sb1, or to sb2. All bits of one register are **mixed** with the bits of the other register (see Figure 3-5).

Figure 3-5. Register sb and its Component Registers sb1 and sb2



As shown in the figure above, odd bits of sb belong to sb1 (they are not shaded), while the even bits belong to sb2 (they are shaded).

Only bits of the sb1 register (odd bits) are changed when the sb register is initialized with a constant. Assembler allows the user to write new data to even bits of sb, but it does not affect the sb2 state.

Use of sb (sb1, sb2)

The sb register can be treated as a logical representation of sb1 and sb2. Later in this Section the cause of this representation will be explained. The sb is directly accessible for writing. It allows the user to initialize the sb1 register and then sb2.

The sb1 register controls split of the Shadow Matrix into rows. The sb2 register determines the same configuration for the Active Matrix and describes input **X** configuration.

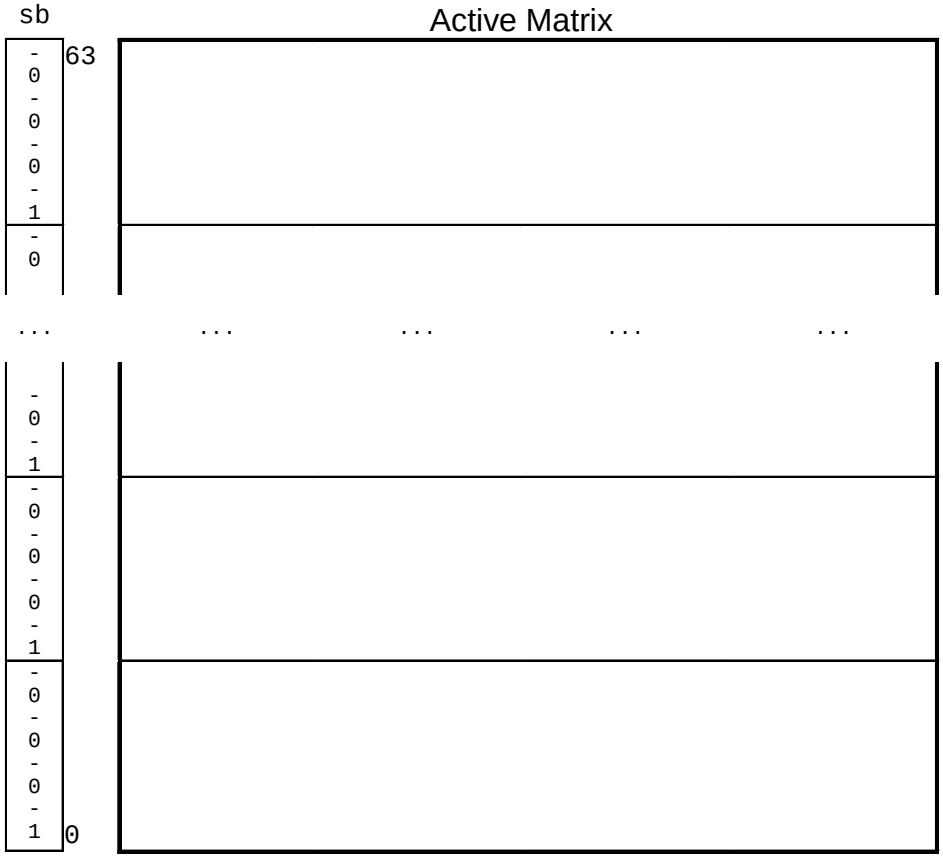
To modify sb1 the odd bits of the sb register must be initialized. Writing to the even bits of sb is ignored. The content of sb1 is copied to sb2 by the wtw instruction.

The 32-bit sb2 register controls the 64-bit input **X** the Active Matrix. This means that its every bit of sb2 handles two bits of data coming through the input **X** of the Active Matrix. It is the main point of the difference between nb2 and sb2.

Then using sb1(sb2) for boundaries specification the bits that are set to '1' are treated the least significant bits (LSB) of elements (see Figure 3-6). This is another point of difference between nb1(nb2) and sb1(sb2).

Figure 3-6 shows the Active Matrix division into rows by means of the sb(sb2) register. Bits marked with "-" belong to sb1 and do not affect the Active Matrix split.

Figure 3-6. Active Matrix Division into Rows Using sb(sb2) Register



The figure above explains where the logical model of the `sb` register came from. This model is like a “comb”. The “comb teeth” are the bits of `sb1` and `sb2`. This model allows to “stretch” 32-bit registers for sixty-four bits.

The minimum element size is two bits. In this case `sb2=0FFFFFFFFh`. The maximum element size is sixty-two bits, here `sb2=1`.

In case `sb1(sb2)=0` the processor automatically sets the LSB of `sb1(sb2)` to ‘`1`’.

Load the sb Register

The `sb` is a 64-bit register. The instruction set of NeuroMatrix® NM6403 does not allow loading a 64-bit constant to registers directly. However there are several indirect methods of initializing the `sb` register.

Method 1. Partial Load.

The NM6403 processor allows separate access to the high and low parts of the `sb` register. The high and low parts of the `sb` have special notation as follows:

- `sbh` - the high 32-bit part of the `sb` register;
- `sbl` - the low 32-bit part of the `sb` register.

Here is an example of the `nb1` register partial load:

```
sbl = 80808080h;    // load the low part of sb.
```

```
sbh = 40404040h;    // load the high part sb.
```

As a result of the command the 64-bit constant `40404040808080hl` will be loaded to the `sb` register.

Method 2. Load the same constant to the both parts of the register.

If the same constant is to be loaded to the low and the high part of the register, the following instruction can be used:

```
sb = 80808080h;     // load the same constant to the high
                    // and to the low part of sb.
```

The processor automatically recognizes access to a 64-bit register and copies the same constant to the both parts of the register. As a result of the command the 64-bit constant `80808080808080hl` will be loaded to the `sb` register.

Method 3. Load the contents of the memory location.

The register `sb` can be initialized with a 64-bit constant located in memory in the following manner:

```
data ".data"
    MySB: long = 0123456789ABCDEFhl;
    ...
end ".data";

...

begin ".text"
    ...
    f1cr = [MySB]; // load 64-bit constant from memory to sb.
    ...
end ".text";
```

The processor automatically recognizes access to a 64-bit register and loads a 64-bit constant from memory. As a result of the command the 64-bit constant `0123456789ABCDEFhl` will be loaded to the `sb` register.

Table 3-19 presents the most frequently used constants for the `sb` register initialization. It is supposed that a 32-bit constant is loaded to `sb` as described in the method two, and a 64-bit constant is read from memory.

Table 3-19. The Most Frequently Used Split Constants for sb Initialization

ELEMENT SIZE	NUMBER OF ELEMENTS	CONSTANT TO LOAD IN SB
64 bits	1	0
32 bits	2	2
20 bits	3	2000020000200002hl
16 bits	4	00020002h
10 bits	6	0208020080200802hl
8 bits	8	02020202h
4 bits	16	22222222h

2 bits	32	0AAAAAAAAh

In order not to mix up, what bits of the sb register must be filled and what should not, it is possible to copy the same value to odd and even bits. It will not cause any problem because the even bits corresponding to sb2 will be ignored. Thus, for example, initialization of sb with the constant 02020202h is equivalent to sb = 03030303h.

3.3.4 Register vr

The 64-bit write accessible vr register is used only in operations of weighted accumulation acting as the operand **Y** (see 1.5.2 on page 1-12). The contents of vr do not come through the Active Matrix, but goes directly to the Vector ALU (see Figure 1-4). The main function of vr is to add the same bias to all elements of a data vector.

Use of the vr Register

When the processor executes the vsum operation the contents of the vr register is read each cycle and added to the result of weighted accumulation on the Active Matrix.

Here is an example of the vr register use in the operation of weighted accumulation:

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

In the result of this instruction execution twenty 64-bit words will be read from the memory location address ar0, each of those words will be multiplied by the weights contained in the Active Matrix and the contents of vr will be added to the result.

Load the vr Register

The vr is a 64-bit register. The instruction set of NeuroMatrix® NM6403 does not allow loading a 64-bit constant to registers directly. However, there are several indirect methods of initializing the vr register.

Method 1. Partial Load.

The NM6403 processor allows separate access to the high and low parts of the vr register. The high and low parts of the vr have special notation as follows:

- vrh - the high 32-bit part of the vr register;

- `vrl` - the low 32-bit part of the `vr` register.

Here is an example of the `nb1` register partial load:

```
vrl = 80808080h;    // load the low part of vr.
vrh = 40404040h;    // load the high part vr.
```

As a result of the command the 64-bit constant `40404040808080hl` will be loaded to the `vr` register.

Method 2. Load the same constant to the both parts of the register.

If the same constant is to be loaded to the low and the high part of the register, the following instruction can be used:

```
vr = 80808080h;     // load the same constant to the high
                    // and to the low part of vr.
```

The processor automatically recognizes access to a 64-bit register and copies the same constant to the both parts of the register. As a result of the command the 64-bit constant `80808080808080hl` will be loaded to the `vr` register.

Method 3. Load the contents of the memory location.

The `vr` register can be initialized with a 64-bit constant located in memory in the following manner:

```
data ".data"
    MyVR: long = 0123456789ABCDEFhl;
    ...
end ".data";

...

begin ".text"
    ...
    vr = [MyVR]; // load 64-bit constant from memory to vr.
    ...
end ".text";
```

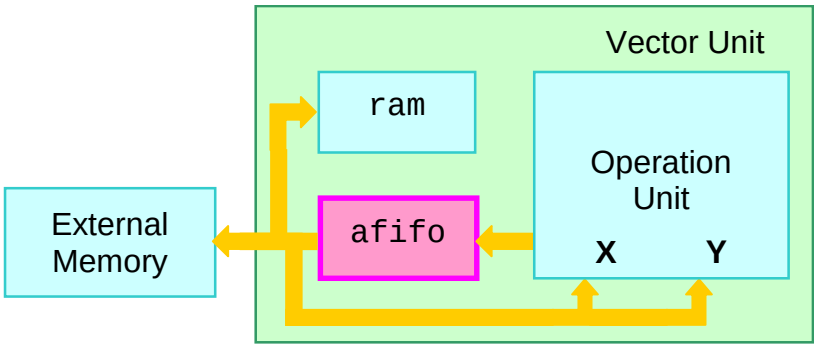
The processor automatically recognizes access to a 64-bit register and loads a 64-bit constant from memory. As a result of the command the 64-bit constant `0123456789ABCDEFhl` will be loaded to the `vr` register.

3.3.5 Register-Container afifo

The `afifo` vector register is a dual port FIFO buffer that is able to contain up to thirty two 64-bit words.

It mainly serves as an accumulator to store the result of the last vector instruction.

Figure 3-7. Interaction of afifo with Other Devices of the NM6403



The register `afifo` is accessible both for reading and writing. But it can be used only in vector instructions of the processor. Data from memory cannot be directly stored into `afifo` avoiding the Operation Unit. This means that `afifo` has only one input port, and this input port is connected to the output port of the Operation Unit (see Figure 3-7). Later a method of data load from memory to `afifo` will be shown.

The `afifo` buffer participates directly or indirectly in every vector instruction except for instructions of weights loading to the Active Matrix. The buffer can be used in both parts of a vector instruction. It appears in the left part to store the calculation results into memory. If `afifo` appears in the right part of an instruction it participates in calculations as the operand **X** and/or **Y**.

Despite its name “accumulation FIFO” `afifo` cannot collect results of a few vector instructions. The term “accumulation ” means the results of the previous operation can be used as input data for the current operation.

The contents of `afifo` are changed by every vector instruction (with the exception of weights loading). If `afifo` contains data, i.e. it is not empty, the next vector instruction must store data into memory or reuse them as the input data for the next step of calculation. An attempt to execute the instruction that makes calculations but does not store or reuse the contents of `afifo` will cause the exception.

To avoid incorrect use of `afifo` it is necessary to follow three rules:

- do not read data from the empty `afifo` (empty `afifo` cannot be used as an input operand of vector instruction);
- do not store data into non-empty `afifo` if the current instruction does not store the previous contents of `afifo` into memory, or does not reuse it as an input operand;
- all internal FIFOs used by a vector instruction must contain the same number of data.

Clear the Contents of afifo

The intr register contains a field that reflects fullness of afifo. The field VPF contains the bit EMPTA (bit 12: "afifo is empty"/"afifo is not empty") and the bit FULLA (bit 11: "afifo is full"/"afifo is not full") (see 3.2.3 on page 3-19). The bits reflect the dynamic state of afifo (the way its state changes during a vector instruction execution).

Field AFIFO_VAL of the intr register contains information about the number of words in afifo after execution of an instruction or a group of instructions is completed. This information changes more slowly than the fields EMPTA and FULLA (it describes the result, but not the process).

The contents of afifo can be cleared by setting to '1' the bit AFCL (bit 14) of the field FCL in the register pswr (see 3.2.6 on page 3-25). After the bit AFCL is set, it must be cleared, otherwise the Vector Unit will clear afifo every clock cycle until the bit AFCL is set to '0'. The following example shows a fragment of the source code that clears the contents of afifo:

```
begin ".text"
    ...
    pswr set   4000h; // the bit AFCL is set to '1'
    pswr clear 4000h; // the bit AFCL is cleared, afifo is empty
    ...
end ".text";
```

Load Data to afifo

The afifo register is a FIFO queue that is intended for containing up to thirty-two 64-bit words. The counter of a vector instruction defines the fullness of afifo, for example:

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

The instruction above makes calculations and stores twenty-four 64-bit words of the result in afifo.

Though information comes to afifo only after it passed the Operation Unit, it is possible to load data from memory without changes. The following example shows the way of loading ten 64-bit words of data from memory to afifo:

```
rep 10 data = [ar0++] with data; // data copies directly to afifo
```

The Vector Unit executes a bitwise logical OR with a zero vector that does not cause any change of data. Now afifo contains ten long words loaded directly from memory.

As mentioned above, the partial loading of data to afifo is not permitted. This means it is impossible to load, for instance, five words with the first instruction and ten more with the second one. The afifo buffer can be loaded only in case it is empty or it is used as the input buffer in the right part of the current vector instruction.

Store Data from afifo into Memory and into ram

The contents of `afifo` can be stored into memory. For example:

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

The instruction above stores one long word from `afifo` into memory. If the right part of a vector instruction is omitted the `afifo` buffer becomes empty.

The `afifo` buffer cannot be partially emptied. This means it is impossible to store into memory ten of sixteen words contained in `afifo`.

The data contained in `afifo` can be copied to `ram`. For example:

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

The instruction above stores the results of calculation into memory and copies the same data to `ram`. Old contents of `ram` will be replaced with the new one. Please, note that it is impossible to copy `afifo` only to `ram` without writing to memory.

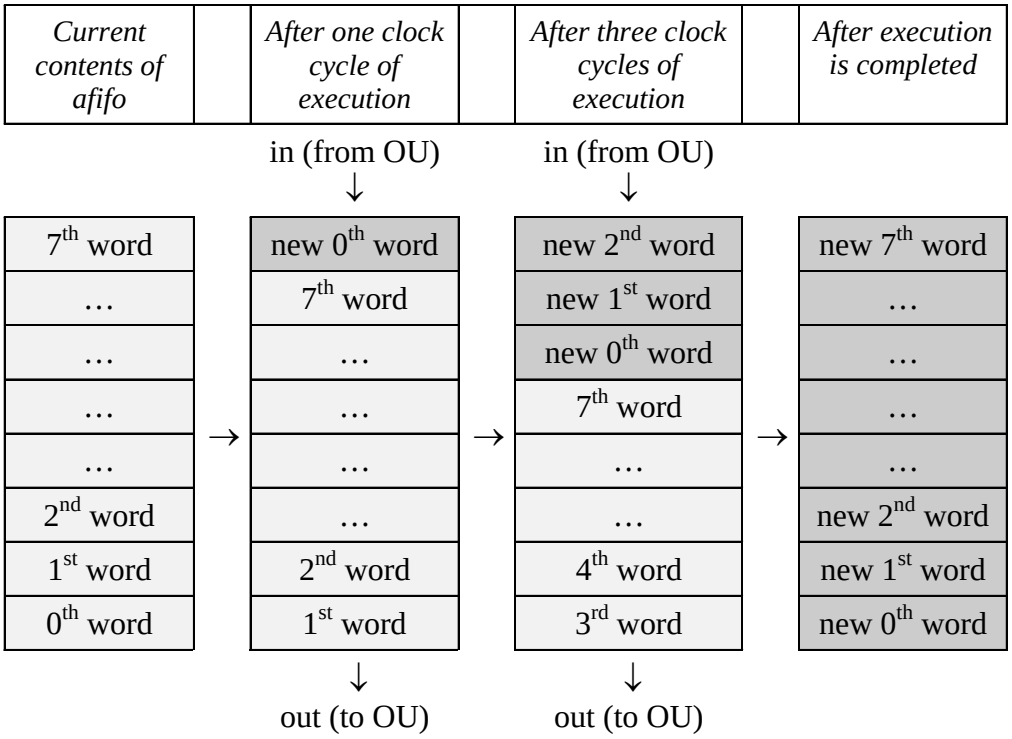
Note *The contents of afifo **cannot** be stored into a register or a register pair, but only into memory.*

Use of afifo as the Input Buffer in Vector Operations

The contents of the dual port `afifo` buffer can be used by the next vector instruction for further calculations. In this case `afifo` is treated as the operand **X** or/and **Y** (see Figure 3-7).

The current content of `afifo` is sent to the Operation Unit. A vector instruction is executed for up to thirty two clock cycles depending on the number of data words it processes. To process one word of data it takes one cycle. As the first word of `afifo` comes out to the Operation Unit (OU), the first word of the new result comes in from the OU. After execution is complete, `afifo` is filled with the new results of calculation. For instance, take a look on the sequence of data changing in `afifo` during the vector instruction execution:

Figure 3-8. Contents of afifo on Different Stages of Vector Instruction Execurion



The following examples show how the `afifo` buffer is used as the operand **X** of the OU, the operand **Y** or both **X** and **Y**.

The contents of `afifo` can be sent to the input **X** of the Operation Unit, for example:

```
                                  X operand      Y operand
                                      ↓              ↓
rep 32 data = [ar0++] with afifo and data;
```

This instruction performs a logical bitwise AND of the contents of the memory location and the data stored in `afifo` at the moment. The operation is executed on the Vector ALU; the first of the two operands in the right part of the instruction goes to the input **X**, and the second one – to the input **Y**.

The contents of `afifo` can be sent to the input **Y** as well:

```
                                                 X       Y
                                                 ↓       ↓
rep 32 data = [ar0++] with vsum , data, afifo;
```

This instruction performs a weighted accumulation of the contents of the memory location and adds the contents of `afifo` to the result of the matrix operation.

The contents of `afifo` can be sent both to the input **X** and **Y**:

```
                        X       Y
                        ↓       ↓
rep 32 with afifo + afifo;
```

This instruction duplicates the contents of `afifo`.

The results of all three operations above are accumulated to `afifo`.

Simultaneous Store into Memory and Use of afifo as Input Operand

The `afifo` register allows the user to perform simultaneous data store into memory and use them as the input data for the next vector instruction, for example:

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

The instruction above performs store of the contents of `afifo` into memory. The same data participate in the subtraction operation in the right part of the instruction. After execution is completed the old contents of `afifo` are stored into memory and the new contents are calculated as the difference between the old contents of `afifo` and `ram`.

It is also possible to copy the contents of `afifo` to `ram`. But this command can be made only together with store of data into memory. For instance:

```
rep 20 [ar0++],ram = afifo with not afifo;
```

In this instruction the contents of `afifo` is stored into memory, copied to `ram` and at the same time is used for the operation of negation in the right part of the instruction. The negation is performed over the old contents of `afifo`.

Use of afifo in Bitwise Mask Operations

The bitwise mask application is executed on the Operation Unit. For mask operations the third input called “mask input” is used in addition to **X** and **Y**. More detailed information about bitwise mask application can be found in paragraph 1.5.4 on page 1-17. The contents of `afifo` can be used as a mask, for example:

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

This instruction performs bitwise mask application on the Vector ALU.

Another example demonstrates use of `afifo` in mask application combined with weighted accumulation:

```
                                        Mask    X    Y
                                         ↓      ↓    ↓
Rep 32 data = [ar0++] with vsum afifo, ram, data;
```

Cause of Errors When Working With afifo

The following reasons can cause errors when working with `afifo`:

- an attempt to read data from empty `afifo`;
- an attempt to read more data than `afifo` contains;
- an attempt to read less data than `afifo` contains;
- an attempt to write results to non-empty `afifo`.

In case an error occurs due to the reasons indicated above, the instruction containing an error is not executed. The processor replaces it with an empty instruction `vnul` and generates the incorrect vector instruction interrupt.

3.3.6 Logical Register-Container data

The vector register `data` is a logical register used to manage data flow passing along an input data bus from the external memory to the Vector Unit during execution of a vector instruction.

Every clock cycle a 64-bit word of data comes from memory to an input of the Vector Unit. This data word is treated as the current content of the `data` register. The `data` register is introduced to control the data flow coming from memory. It allows the user to redirect this flow to the desired input of the Vector Unit.

So, the `data` register can be considered a pseudo-buffer of an input data bus. A pseudo-buffer is organized according to the FIFO principle and can contain up to thirty-two 64-bit words. The number of words in `data` depends on the number of words processing by the current vector instruction.

Use of data Register to Handle Data on Fly

The `data` register is used to handle data flow coming from memory, for example:

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

The vector instruction above performs logical bitwise OR of data loaded from memory and the contents of the internal `ram` buffer. This instruction processes thirty-two words of data. The data coming from memory are directed to the input **Y** of the Vector Unit.

The same way data coming from memory can be directed to the input **X**:

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

The instruction above adds the contents of the internal `ram` buffer to the data loaded from the memory location, which address is given by `ar0`.

The data from memory can be directed to both inputs **X** and **Y** of the Vector Unit, for example:

```
                                    X       Y
                                    ↓       ↓
rep 32 data = [ar0++] with data + data;
```

The instruction above duplicates on fly the data loaded from memory.

Use of data Register in Bitwise Mask Operations

The bitwise mask application is executed on the Operation Unit. For mask operations the third input called “mask input” is used in addition to **X** and **Y**. More detailed information about bitwise mask application can be found in paragraph 1.5.5 on page 1-19. The contents of `data` can be used as a mask, for example:

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

This instruction performs bitwise mask application on the Vector ALU.

Another example demonstrates use of `data` in mask application combined with weighted accumulation:

```
                                        Mask    X     Y
                                         ↓      ↓     ↓
rep 32 data = [ar0++] with vsum data, ram, afifo;
```

Cause of Errors When Working With data Register

The following reasons can cause errors when working with `data`:

- an attempt to use `data` in the right part of instruction without loading data from memory in the left part;

In case an error occurs due to the reason indicated above, the instruction containing an error is not executed. The processor replaces it with an empty instruction `vnul` and generates the incorrect vector instruction interrupt.

3.3.7 Register-Container ram

The `ram` vector register is a FIFO buffer that is able to contain up to thirty-two 64-bit words. The `ram` has a single bidirectional port, so contrary to `afifo` the data can be either stored into `ram` or loaded from `ram`, but not at the same time.

It mainly serves as a buffer to store the data that can be reused many times in vector operations.

The `ram` register is accessible both for reading and writing. However it can be used only in the vector instructions. Data can be stored into `ram` directly from memory or from `afifo` (see 3.3.5 on page 3-49). The contents of `ram` cannot be directly stored into the external memory, but only through the Operation Unit.

The contents of `ram` are used as input data for operations on the Vector ALU and for weighted accumulation on the Active Matrix. The words of

data contained in ram can be sent both to the input **X** of the Operation Unit and to the input **Y**.

When the new data are stored into ram its old contents are lost. The ram vector register can be repeatedly used in vector operations. However, all data contained in ram must participate in computations. The number of data in ram and the counter of a vector instruction must be the same. Mismatch of those counters will cause an incorrect vector instruction interrupt.

Field RAM_VAL of the intr register (see 3.2.3 on page 3-19) contains information about the number of words in ram after execution of an instruction or a group of instructions executed in parallel. This information is changed once during an instruction execution and it describes the resultant number of words in ram.

Load Data to ram

Data are loaded directly from the external memory, for example:

```
rep 16 ram = [ar0++];
```

This instruction describes load of sixteen 64-bit words of packed data from the external memory to ram. The old contents of the buffer are lost even if thirty-two words were stored there.

When ram is loaded from memory the data go through the input data bus, so it is possible to use the data register to direct them to the Operation Unit inputs, for example:

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

This instruction loads the contents of the memory location to ram and the same data are transferred to the Vector ALU input **X** and add to the contents of afifo.

Use of ram in Operations on the Operation Unit

The data stored in ram can be directed to the input **X** of the Operation Unit, for example:

```
                                  X       Y
                                  ↓       ↓
rep 32 data = [ar0++] with ram and data;
```

This instruction executes a logical bitwise AND of data stored in ram and the contents of the memory location. The operation is executed on the Vector ALU.

The same contents of ram can be directed to the input **Y**:

```
                                        X     Y
                                        ↓     ↓
rep 32 data = [ar0++] with vsum , data, ram;
```

This instruction executes weighted accumulation on the Active Matrix.

The contents of ram can be directed to both inputs: **X** and **Y**, for example:

```
                X       Y
                ↓       ↓
rep 32 with ram + ram;
```

This instruction duplicates the contents of `ram`.

Use of ram in Bitwise Mask Operations

The bitwise mask operations are executed on the Mask Application Unit. In addition to the inputs **X** and **Y** the Mask Application Unit contains the Mask input. For more details about mask application see paragraph 1.5.4 on page 1-17.

The contents of `ram` can be directed to the Mask input, for example:

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

This instruction executes the bitwise mask operation on the Vector ALU.

Another example demonstrates use of `ram` in mask operation combined with weighted accumulation:

```
                                    Mask    X       Y
                                     ↓      ↓       ↓
rep 32 data = [ar0++] with vsum ram, data, afifo;
```

Cause of Errors when Working with ram

The following reasons can cause errors when using `ram` in vector instructions:

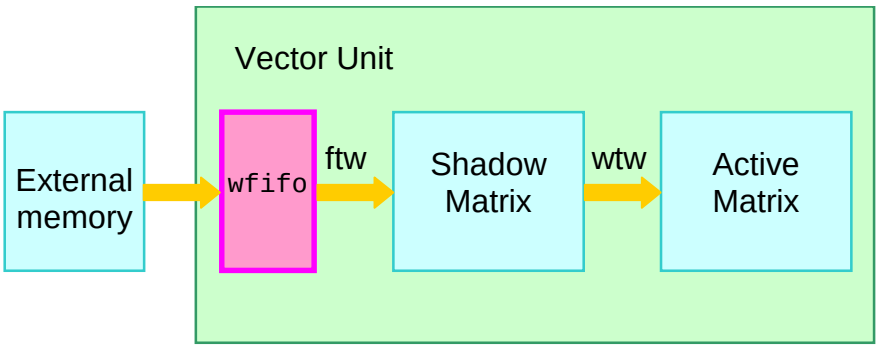
- an attempt to read data from empty `ram`;
- an attempt to read more data than `ram` contains;
- an attempt to read less data than ram contains;
- an attempt to simultaneously write to `ram` and to use its old contents.

In case an error occurs when working with `ram`, the instruction containing an error is not executed. The processor replaces it with an empty instruction nul and generates the incorrect vector instruction interrupt.

3.3.8 Register-Container wfifo

The `wfifo` vector register is a dual port FIFO buffer that is able to contain up to thirty-two 64-bit words. It serves as a container for weight coefficients that are then loaded to the Shadow Matrix and to the Active Matrix.

Figure 3-9. Load Weights from External Memory



As can be seen from the figure, `wfifo` is a buffer that is used for fast data loading from the external memory. It allows the processor to decrease external data bus activity.

The `wfifo` register is accessible both for reading and writing. However it can be used only in vector instructions. Data are loaded `wfifo` directly from the external memory. The data stored in `wfifo` can be directed only to the Shadow Matrix.

The `wfifo` register cannot be used as a source buffer for operations on the Operation Unit. Its only purpose is to store weights.

Unlike other register-containers, the partial data load to `wfifo` is possible. The data contained in `wfifo` may also be partially transferred to the Shadow Matrix.

For example, on the first stage eight long words are loaded to `wfifo`. Then twenty four more words are loaded, so the number of words in `wfifo` become thirty two. The same happens when data are transferred from `wfifo` to the Shadow Matrix. The number of data words to be transferred depends on the Matrix configuration. If the Matrix is divided into eight rows, eight long words will be loaded from `wfifo` to it. But `wfifo` is able to store up to thirty two words, so it is not necessary to use external data buses to load weights each time the user needs to reload the Matrix.

Note

Weights transfer from `wfifo` to the Shadow Matrix always takes thirty-two clock cycles. This is the main restriction to the Vector Unit. All algorithm designers must take it into account. But this operation can be executed on the background of other vector and scalar instructions. So, if weights are changed less often then every 32 clock cycles, the weights loading will not decrease performance of the NeuroMatrix® NM6403.

Clear the Contents of wfifo

The `intr` register contains a field that reflects fullness of `wfifo`. The field `VPF` contains the bit `EMPTW` (bit 10: " `wfifo` is empty"/" `wfifo` is not empty") and the bit `FULLW` (bit 9: "`wfifo` is full"/"`wfifo` is not

full") (see 3.2.3 on page 3-19). The bits reflect the dynamic state of `wfifo` (the way its state changes during a vector instruction execution).

The field `WFIFO_VAL` of the `intr` register contains information about the number of words in `wfifo` after execution of an instruction or a group of instructions is completed. This information changes more slowly then the fields `EMPTW` and `FULLW` (it describes the result, but not the process).

The contents of `wfifo` can be cleared by setting to '`1`' the bit `WFCL` (bit 15) of the field `FCL` in the register `pswr` (see 3.2.6 on page 3-25). After the bit `WFCL` is set, it must be cleared, otherwise the Vector Unit will clear `wfifo` every clock cycle until the bit `WFCL` is set to '`0`'. The following example shows a fragment of the source code that clears the contents of `wfifo`:

```
begin ".text"
    ...
    pswr set   8000h; // the bit WFCL is set to '1'
    pswr clear 8000h; // the bit WFCL is cleared, wfifo is empty
    ...
end ".text";
```

Load Weights to wfifo

Weights are loaded to `wfifo` directly from memory. The `wfifo` vector register can be filled with data for one or several instructions, for example:

```
rep 16 wfifo = [ar0++]; // load of the first sixteen words
rep 16 wfifo = [ar1++]; // load of the second sixteen words from
                        // the different memory location
```

The first instruction above loads sixteen long words of weight coefficients to `wfifo`, and the second one loads sixteen more words from the different memory location. After the instructions are executed the total number of words in `wfifo` is thirty-two.

Transfer Weight from wfifo to Shadow Matrix

The weights contained in `wfifo` are transferred to the Shadow Matrix. To do that the instruction `ftw` is used. It can be a separate instruction, for example:

```
ftw;
```

or it may be included to another vector instruction, for example:

```
rep 32 data = [ar0++], ftw with not data;
```

Simultaneous Load from Memory and Transfer to Shadow Matrix

The `wfifo` vector register is a dual port FIFO buffer. It can be used for simultaneous load of the new weights and transfer the old ones to the Shadow Matrix, for example:

```
rep 32 wfifo = [ar0++], ftw;
```

This instruction executes the parallel load/transfer command. If `wfifo` was empty before the instruction starts executing then the weights transferring to the Shadow Matrix will be locked until the first data word is loaded from memory to `wfifo`. When the first data word appears in the buffer, it will be transferred to the Shadow Matrix in parallel with loading the next data word from memory.

If `wfifo` contained some weights to the moment of the instruction execution, then according to the FIFO principle they will be the first ones to be sent to the Shadow Matrix. At the same time the new data will be loaded to `wfifo`. If the number of weights contained in `wfifo` to the moment of the instruction execution is not enough to fill the Shadow Matrix, a part of new loaded data will be transferred to it.

Load Several Weight Sets to wfifo

The number of words to load to the Shadow Matrix from `wfifo` is defined by the Matrix division into rows. Each row of the Matrix is associated with a 64-bit data word. (see 3.3.3 on page 3-44).

If the number of rows of the Shadow Matrix is rather small it is possible to contain in `wfifo` more than one weight set. Several weight sets can be loaded to `wfifo` at once and then they will be partially transferred to the Shadow Matrix. It will cause a decrease in data bus activity and allow the user application to exploit external buses more effectively, for example:

```
data "dataMatrix"
    Matrix: long[32] = (...);
end "dataMatrix";

begin "text"
...
sb = 02020202h; // The matrix is divided into eight rows.
ar0 = Matrix;   // Load the matrix address to the register.
rep 32 wfifo = [ar0++], ftw, wtw; //load the weights
...             // with simultaneous transferring eight of
                // them to the Shadow Matrix.
                // Twenty-four words remained in wfifo.
ftw, wtw;       // Load the next eight words
                // to the Shadow Matrix.
...             // Sixteen words remained in wfifo.
ftw, wtw;       // Load the next eight words
                // to the Shadow Matrix.
...             // Eight words remained in wfifo.
ftw, wtw;       // Load the next eight words
                // to the Shadow Matrix.
...             // wfifo is empty.
```

```
end "text";
```

A weight transferring from `wfifo` to the Shadow Matrix does not engage the external bus, so it can proceed in parallel with execution of other vector and scalar instructions.

Cause of Errors when Working with wfifo

The following reasons cause errors when working with `wfifo`:

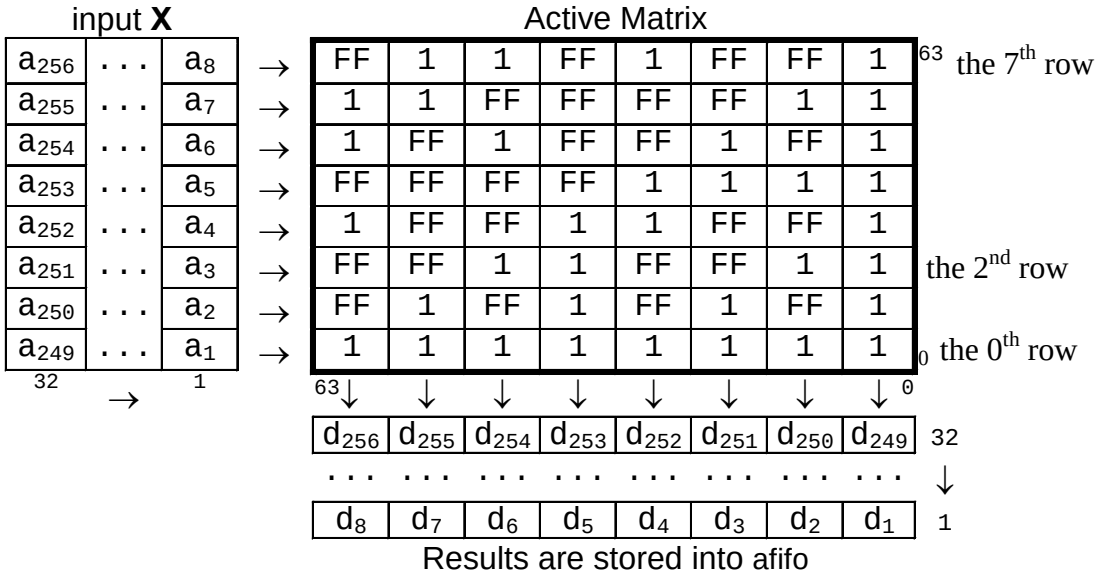
- an attempt to read data from empty `wfifo`;
- an attempt to load data to full `wfifo` without simultaneous transfer to the Shadow Matrix;

In case an error occurs when working with `wfifo`, the instruction containing an error is not executed, and the processor is blocked.

Example of Weights Load to wfifo

The following example describes the procedure of weights loading to the Active Matrix. It is assumed that the Matrix is divided into eight rows and eight columns. Figure 3-10 presents the diagram of data processing on the Active Matrix:

Figure 3-10. Load Weights to the Active Matrix



Weights are loaded to the Active Matrix from the external memory. The memory contents are represented by 64-bit words. Each word consists of eight 8-bit elements. The registers `nb2` and `sb2` specify configuration of the Active Matrix. Each row is associated with a separate 64-bit word. The 0th word is loaded to the 0th matrix row. As it can be seen from Figure 3-10, the rows of the Active Matrix are ordered from the bottom to the top. This order is specified according to the NeuroMatrix® NM6403 bit order (LSB). Memory addresses increase in the same direction.

In assembly language the record of weights array is represented as follows:

```
data "data"
    Matrx: long[8] = (00101010101010101hl,  // the zero row
                      0FF01FF01FF01FF01hl,  //the LS byte
                      0FFFF0101FFFF0101hl,  // the second row
                      001FFFF0101FFFF01hl,
                      0FFFFFFFF01010101hl,
                      001FF01FFFF01FF01hl,
                      00101FFFFFFFF0101hl,
                      0FF0101FF01FFFF01hl);  // the seventh row
end "data";
```

Since the NM6403 processor Active Matrix is divided by the register nb2 into eight columns, the least significant bytes of the weight words of the array are loaded to the 0^{th} column, the first bytes go to the first column, etc.

Note

The Active Matrix rows are ordered from the bottom to the top, the 0^{th} word of a weight array is loaded to the 0^{th} row. But in an assembly file the words of data to be loaded to the Matrix are represented in the inverse order, from the top to the bottom (see the example above).

Weights are loaded from memory. The following example shows how to load weights to the Active Matrix for further calculations:

```
begin "text"
<_Func>
    nb1 = 80808080h;  // division of the matrix into columns
    sb  = 03030303h;  // division of the matrix into rows.

    ar0 = Matrx;      // Load weight buffer address to ar0.
    rep 8 wfifo = [ar0++], ftw, wtw;
    ...
end "text";
```

First of all the registers `nb1` and `sb` are initialized. They define the future Active Matrix configuration. This configuration becomes active after the instruction `wtw` is completed.

The 64-bit `nb1` and `sb` registers are initialized with 32-bit constants. The processor copies those values to both high and low parts of the registers, i.e. nb1 is initialized with the long constant `8080808080808080hl`. The same is true for the register `sb`. More detailed information about the registers `nb1` and `sb` can be found in paragraphs 3.3.2 on page 3-40 and 3.3.3 on page 3-44 respectivly.

Thus the future Active Matrix configuration is defined.

The address of weight array is put to an address register and then data are loaded from memory to `wfifo`.

The `ftw` instruction activates data transfer from `wfifo` to the Shadow Matrix. This operation always takes thirty-two cycles, no matter how many words are loaded to the Matrix. Their conversion to the internal presentation lasts thirty-two clock cycles. However the conversion is executed in parallel with other vector and scalar instructions and starts from the moment of when the first word appears in `wfifo`.

When the weights transferred to the Shadow Matrix, the instruction `wtw` is executed. It copies the contents of the Shadow Matrix to the Active Matrix. It also copies the contents of the `nb1` and `sb1` registers to `nb2` and `sb2` respectively. This operation lasts one clock cycle. Weights' loading to the Active Matrix is complete.

4 Format of Processor Instructions

The NeuroMatrix® NM6403 incorporates 32-bit and 64-bit instruction. Any instruction contains two processor operations like addressing and arithmetic operation. In this sense NM6403 is a scalar microprocessor with static LIW-architecture.

Length of Instruction Word

The processor instruction formats are:

Short instruction is a 32-bit instruction. It does not contain a constant.

Long instruction is a 64-bit instruction. It contains a 32-bit constant.

The NeuroMatrix® NM6403 address size is thirty-two bits. One memory cell is used to store short instruction, two memory cells – for long one.

Note

Long instructions are always located at even address. On a compilation stage the assembler aligns long instructions to an even address filling empty spaces with 'nul' instructions if necessary.

Accessing memory the NM6403 fetches two short instructions, or one long instruction at a time. That is why register `pc` defining the address of the next instruction has always an even value. For more details about the `pc` register see 3.2.5 on page 3-24.

Types of Processor Instructions

All instructions of the NeuroMatrix® NM6403 are divided into two groups:

- **Scalar instructions** are used to control the scalar RISC-core, timers, to execute all register operations (accessible for reading/writing) with the exception of the vector registers;
- **Vector instructions** are used to control the Vector Unit.

Structure of Scalar Instructions

Every instruction of the NeuroMatrix® NM6403 consists of **two parts** that are conventionally called "left" and "right". The processor simultaneously executes both parts of an instruction.

The **left** part of an instruction is for **address commands** like memory access, address modification and so on. The **right** part is for **arithmetic-logical operations** except address calculation and memory addressing.

The left and the right parts of an instruction are split by the reserved word `with`. Here is an example of a scalar processor instruction:

```
gr0 = [ar0++] with gr1 = gr3 << 4;
```

Left part of an instruction, address command

Right part of an instruction, arithmetic operation

Hereafter the left part of an instruction is called a *command* and the right one is an *operation*.

In an assembler program the left or the right part of an instruction can be omitted. However, since the processor cannot execute only the left or the right part of an instruction, an empty operation `nul` is automatically added instead of the omitted part during compilation, i.e. an assembly instruction written as:

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

is treated by assembler as:

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

The same for the left part:

```
gr1 = gr3 << 4;
```

is regarded as

```
nul with gr1 = gr3 << 4;
```

To improve readability of the program, in case the left or the right part of an instruction is not used, the reserved word `with` can be omitted.

Structure of Vector Instructions

The vector instructions as well as the scalar ones are split into the left and the right parts. But they have an additional field that presents in all vector instructions except for single instructions `ftw` and `wtw`. This field contains information about a number of repetitions. One vector instruction can process from one to thirty-two long words of data. It works according to SIMD (Single Instruction Multiple Data) principle. The same manipulation performs on up to thirty two 64-bit words of data. Here is an example of a vector instruction:

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

Left part of an instruction, address command

Right part of an instruction, arithmetic operation

The left and the right part of a vector instruction are split by the reserved word '`with`'. The repetition field "`rep 32`" defines how many 64-bit words will be processed by the instruction. In most cases a vector instruction will be executed for as many clock cycles, as the counter value is, because the Vector Unit spends one clock cycle to process each long word of data.

In case the left part of an instruction is omitted, the repetition field and the word with are still on their positions, for instance:

```
rep 16 with ram - 1; // correct instruction
```

Any other forms of instruction like the following one:

```
rep 16 ram - 1; // incorrect instruction
```

or

```
with ram - 1; // incorrect instruction
```

are false. The assembler will inform about it.

Note *Vector and scalar instructions cannot be mixed in one instruction. No matter that one of them has the left part omitted and the other one – the right part.*

4.1 Types of Scalar Instructions

A scalar instruction consists of two parts. The certain types of scalar commands or operations can be met in each part. The table below shows the groups of scalar commands that can be met in the left and in the right part of a scalar instruction.

Table 4-1. Position of Different Types of Commands in a Scalar Instruction

LEFT PART OF SCALAR INSTRUCTION (COMMAND)	RIGHT PART OF SCALAR INSTRUCTION (OPERATION)
• Commands of loading registers; • Commands of copy registers value; • Commands of address arithmetic; • Special scalar commands; • Commands of unconditional and conditional branch; • Commands of unconditional and conditional function calls; • Commands of return from a function or an interrupt; • Empty command.	• Arithmetic operations; • Logical operations; • Shift operations; • Empty operation.

In a scalar instruction any type of command from the left column of the table can be placed together with any type of operation from the right column. However, two types of command/operation from the same column cannot stand together.

The following instruction can be considered as an example of a scalar instruction containing a left and a right part:

```
gr4 = [ar0++] with gr0 = gr1 and not gr2;
```

In the left part the register `gr4` is initialized with the contents of the memory. The register `ar0` points to the memory address of a buffer to process. After the memory cell is read `ar0` increments to point to the next memory address (post-increment).

In the right part a three-operand logical operation is executed. The processor performs bitwise logical AND of the contents of `gr1` register and the bitwise logical complement of the contents of the `gr2` register. The result is stored into the `gr0` register.

The general-purpose registers can be met in both parts of the instruction, but the address registers are used only in the left part. Use of the same general-purpose registers in both parts of a scalar instruction is discussed in the document NeuroMatrix NM6403 SDK. Assembly Language Overview. Detailed Description of Instructions.

4.2 Types of Vector Instructions

A vector instruction, as well as a scalar one, consists of a left and a right part. The left part is for address operations, the right one is for vector operations. The table below shows, which types of commands can be located in the left part and which in the right part of a vector instruction.

Table 4-2. Position of Different Types of Commands in Vector Instruction

LEFT PART OF VECTOR INSTRUCTION (COMMAND)	RIGHT PART OF VECTOR INSTRUCTION (OPERATION)
• Command of loading data to the Vector Unit internal FIFOs; • Command of storing the processed data from the Vector Unit internal accumulation FIFO (`afifo`) to memory; • Special vector commands (`ftw`, `wtw`); • Empty vector command.	• Weighted accumulation (multiplication and accumulation); • Mask application operation; • Arithmetic operations; • Logical operations; • Cyclic shift right operation; • Activation functions application; • Store control vector registers in memory; • Empty operation.

Here is an example of a vector instruction with a left and a right part:

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

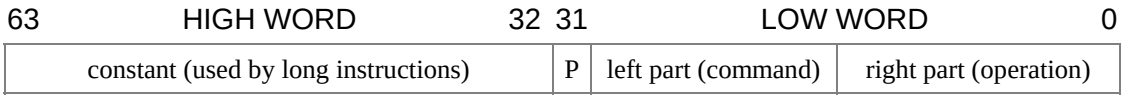
The reserved word '`with`' splits the instruction into the left and the right part. The repeat counter `rep` *`number`* contains the number of 64-bit words to be processed. The command of data loading to the internal buffer `ram` is executed in the left part of the instruction. In the right part the data passing to `ram` via the data bus are dubbed and directed to the Active Matrix to execute weighted accumulation.

4.3 Structure of Processor Instruction Word

The processor instructions are of 32-bit and 64-bit length. The 64-bit instructions have the same structure of low word as the 32-bit ones. The high word is used to contain 32-bit constant.

The processor instruction word has codes of two commands, a flag of parallel execution and a constant used by this instruction (only 64-bit instructions):

Figure 4-1. Structure of NM6403 Mashine Instruction Word



Bit `P` indicates whether the processor should execute vector and scalar instructions using different processor resources sequentially or in parallel.

Table 4-3 gives a list of scalar and vector instructions with indication of their length in 32-bit words.

The first column of the table indicates the conventional index of a processor instruction word. These indexes are introduced to make the reference to processor instruction word structure easier and they are used later in this document.

Table 4-3. The Complete List of NeuroMatrix NM6403 Mashine Instructions

№	INSTRUCTION MEANING	LENGTH	CONTENTS OF THE HIGH WORD OF AN INSTRUCTION
1.1	Load/store of a register in memory	1	–
1.2	Load/store of a register in memory	2	Address incrementation
2.1	«register-register» transfer	1	–
2.2	Load of a constant to a register	2	Constant
3.1	Address register modification	1	–
3.2	Address register modification	2	Modification constant
3.3	Empty instruction	1	–
3.4	Empty instruction	2	Any constant
4.1	Sub-routine call	1	–
4.2	Sub-routine call	2	Relative incrementation constant
4.3	Return from an interrupt/ sub-routine	1	–
5.1	Vector load/store	1	–
5.2	Load of weights from memory	1	–
5.3	Empty vector instruction	1	–

The complete description of the instructions NM6403 see chapter 7 of the document NeuroMatrix NM6403 SDK. User's Guide.

5 Assembly Instruction Set Summary

This section gives a structured summary of the instruction set of the NeuroMatrix® NM6403 assembly language. The summary consists of two subsections: overview of the set of scalar and the set of vector instructions. Instructions are grouped according to their types. For each command or operation its position in the assembler instruction is given (in what part of the instruction – left or right it may appear).

5.1 NM6403 Scalar Instructions Summary

This section gives the complete list of the processor scalar instructions with brief comments.

All scalar instructions of the processor are grouped into tables according to their functionality. The first column explains the instruction functionality. The second one describes the instruction syntax. The third one contains the size of an instruction and the fourth gives a type of the instruction code.

Size of an assembly instruction is defined by the left part. If a constant is used the instruction takes 64-bit. A 32-bit constant may appear only in the left part of the instruction. Shift values are not regarded as constant because they are put to the particular instruction field (6 bits) inside the right part of the processor instruction.

In case syntax defines only the right part of an assembly instruction, the instruction size is missed (marked with “-”).

Since the processor instructions contain the left and the right parts, the discussed command or the operation is underlined. Non-underlined parts of an instruction are actual commands or operations given to retain the idea of the instruction structure.

Unless otherwise arranged, the arbitrary address register or general-purpose register can be used in syntax description.

5.1.1 No Operation Command

DESCRIPTION	SYNTAX	SIZE	TYPE
No operation command	**nul;**	1	3.3
Long no operation command	**nul Const;**	2	3.4
No operation command in the left part of an instruction	**nul with gr0 += gr1;**	-	3.3, 3.4
No operation command in the left part of an instruction (*)	**gr0 += gr1;**	-	3.3, 3.4
No operation command in the left part of an instruction	**nul Const with gr0 += gr1;**	2	3.4
No operation command in the right part of an instruction (*)	**[ar0++] = gr0;**	1	3.3

No operation command in the right part of an instruction (**)	**with gr0 = gr1 >> 0;** ar0 = gr2 with gr0 = gr1 >> 0;	-	3.3, 3.4

Note *Instructions marked with (*) show that in case there is a no operation command in the left or the right part of an instruction the no operation part can be omitted.*

Note *In the instruction marked with (**) any type of shift 0 or shift 32 is considered a no operation command. The content of the source register does not copy to the destination register. The content of the destination register remains the same as before the operation.*

5.1.2 Load Commands

Load commands are located only in the left part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Load the contents of the source memory location expressed by a constant into an address register.	**ar0 = [Const_Addr];** ar0 = [Const] with gr1 = gr2 and gr3;	2	1.2
Load the contents of the source memory location expressed by a constant into a general-purpose register.	**gr0 = [Const_Addr];** gr0 = [Const] with gr1 = gr2 A>> 1;	2	1.2
Load the contents of the source memory location expressed by a constant into a register pair. (*)	**ar0,gr0 = [Const_Addr];** ar0,gr0 = [Const] with gr1 += gr2;	2	1.2
Load the contents of the source memory location into an address register.	**ar0 = [ar1];** ar0 = [ar1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a general-purpose register.	**gr0 = [ar1];** gr0 = [ar1] with gr1 = not gr2;	1	1.1
Load the contents of the source memory location into a register pair. (*)	**ar0,gr0 = [ar1];** ar0,gr0 = [ar1] with gr1 += gr2;	1	1.1
Load the contents of the source memory location into an address register (general-purpose register addressing).	**ar0 = [gr4];** ar0 = [gr4] with gr1 = -gr2;	1	1.1
Load the contents of the source memory location into a general-purpose register (general-purpose register addressing).	**gr0 = [gr1];** gr0 = [gr1] with gr1 = gr2 << 10;	1	1.1

Load the contents of the source memory location into a register pair (general-purpose register addressing). (*)	**ar0,gr0 = [gr1];** ar0,gr0 = [gr1] with gr1 = gr2 or gr3;	1	1.1
Load the contents of the source memory location into an address register with the address postincrementation.	**ar0 = [ar1++];** ar0 = [ar1++] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a general-purpose register with the address postincrementation.	**gr0 = [ar1++];** gr0 = [ar1++] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a register pair with the address postincrementation The address is incremented by 2. (*)	**ar0,gr0 = [ar1++];** ar0,gr0 = [ar1++] with gr1 += gr2;	1	1.1
Load the contents of the source memory location into an address register with the address preincrementation.	**ar0 = [--ar1];** ar0 = [--ar1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a general-purpose register with the address preincrementation.	**gr0 = [--ar1];** gr0 = [--ar1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a register pair with the address preincrementation The address is decremented by 2. (*)	**ar0,gr0 = [--ar1];** ar0,gr0 = [--ar1] with gr1 += gr2;	1	1.1
Load the contents of the source memory location into an address register with the address postincrementation by a value of the related general-purpose register. (*)	**ar0 = [ar1++gr1];** ar0 = [ar1++gr1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a general-purpose register with the address postincrementation by a value of the related general-purpose register. (*)	**gr0 = [ar1++gr1];** gr0 = [ar1++gr1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a register pair with the address postincrementation by the value of a related general-purpose register. (*)	**ar0,gr0 = [ar1++gr1];** ar0,gr0 = [ar1++gr1] with gr1 += gr2;	1	1.1
Load the contents of the source memory location into an address register with the address preincrementation by a value of the related general-purpose register. (*)	**ar0 = [ar1+=gr1];** ar0 = [ar1+=gr1] with gr1 -= gr2;	1	1.1

Load the contents of the source memory location into a general-purpose register with the address preincrementation by a value of the related general-purpose register. (*)	**gr0 = [ar1+=gr1];** gr0 = [ar1+=gr1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a register pair with the address preincrementation by a value of the related general-purpose register. (*)	**ar0,gr0 = [ar1+=gr1];** ar0,gr0 = [ar1+=gr1] with gr1 += gr2;	1	1.1
Load the contents of the source memory location into an address register with preliminary initialization of an address register by a value of the related general-purpose register. (*)	**ar0 = [ar1=gr1];** ar0 = [ar1=gr1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a general-purpose register with preliminary initialization of an address register by a value of the related general-purpose register. (*)	**gr0 = [ar1=gr1];** gr0 = [ar1=gr1] with gr1 -= gr2;	1	1.1
Load the contents of the source memory location into a register pair with preliminary initialization of an address register by a value of the related general-purpose register. (*)	**ar0,gr0 = [ar1=gr1];** ar0,gr0 = [ar1=gr1] with gr1 += gr2;	1	1.1
Load the contents of the source memory location into an address register with preliminary initialization of an address register by the constant expression.	**ar0 = [ar1=Const_Addr];** ar0 = [ar1=Const] with gr1 -= gr2;	2	1.1
Load the contents of the source memory location into a general-purpose register with preliminary initialization of an address register by the constant expression.	**gr0 = [ar1=Const_Addr];** gr0 = [ar1=Const] with gr1 -= gr2;	2	1.1
Load the contents of the source memory location into a register pair with preliminary initialization of an address register by the constant expression. (*)	**ar0,gr0 = [ar1=Const_Addr];** ar0,gr0 = [ar1=Const] with gr1 += gr2;	2	1.1
Load the contents of the source memory location into an address register with the address preincrementation by the constant expression.	**ar0 = [ar1+=Const];** **ar0 = [ar1-=Const];** ar0 = [ar1+=Const] with gr1 -= gr2;	2	1.1
Load the contents of the source memory location into a general-purpose register with the address preincrementation by the constant expression.	**gr0 = [ar1+=Const];** **gr0 = [ar1-=Const];** gr0 = [ar1+=Const] with gr1 -= gr2;	2	1.1

Load the contents of the source memory location into a register pair with the address preincrementation by the constant expression. (*)	**ar0,gr0 = [ar1+=Const];** **ar0,gr0 = [ar1-=Const];** `ar0,gr0 = [ar1+=Const] with gr1;`	2	1.1

Note *In the instructions marked with (*) register pairs are used. They are formed by address registers and general-purpose registers with the same number, for instance: (ar3, gr3) or (ar5, gr5). Registers with different numbers cannot form a register pair. A register pair contains a 64-bit word. The high thirty-two bits are always located in the address register; the low bits are in the general-purpose register. This thing does not depend on the order of the registers in an assembly instruction. For more information about register pairs see 3.1.3 on page 3-4.*

Note *The type of a load instruction: load 64 bits or 32 bits, is defined by a recipient register.*

5.1.3 Store Commands

Store commands are located only in the left part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Store the contents of an address register into the destination memory location expressed by a constant.	**[Const_Addr] = ar0;** `[Const] = ar0 with gr1 = gr2 and gr3;`	2	1.2
Store the contents of a general-purpose register into the destination memory location expressed by a constant.	**[Const_Addr] = gr0;** `[Const] = gr0 with gr1 = gr2 A>> 1;`	2	1.2
Store the contents of a register pair into the destination memory location expressed by a constant. (*)	**[Const_Addr] = ar0,gr0;** `[Const] = ar0,gr0 with gr1 += gr2;`	2	1.2
Store the contents of an address register into the destination memory location.	**[ar0] = ar1;** `[ar0] = ar1 with gr1 -= gr2;`	1	1.1
Store the contents of a general-purpose register into the destination memory location.	**[ar1] = gr0;** `[ar1] = gr0 with gr1 = not gr2;`	1	1.1
Store the contents of a register pair into the destination memory location. A long word is stored into memory. (*)	**[ar1] = ar0,gr0;** `[ar1] = ar0,gr0 with gr1 += gr2;`	1	1.1
Store the contents of an address	**[gr0] = ar4;**	1	1.1

register into the destination memory location (general-purpose register addressing).	`[gr0] = ar4 with gr1 = -gr2;`		
Store the contents of a general-purpose register into the destination memory location (general-purpose register addressing).	**`[gr0] = gr1;`** `[gr0] = gr1 with gr1 = gr2 << 10;`	1	1.1
Store the contents of a register pair into the destination memory location. A long word is stored into memory. (general-purpose register addressing). (*)	**`[gr1] = ar0,gr0;`** `[gr1] = ar0,gr0 with gr1 = gr2 or gr3;`	1	1.1
Store the contents of an address register into the destination memory location with the address postincrementation.	**`[ar0++] = ar1;`** `[ar0++] = ar1 with gr1 -= gr2;`	1	1.1
Store the contents of a general-purpose register into the destination memory location with the address postincrementation.	**`[ar1++] = gr0;`** `[ar1++] = gr0 with gr1 -= gr2;`	1	1.1
Store the contents of a register pair into the destination memory location with the address postincrementation. The address is incremented by 2. (*)	**`[ar1++] = ar0,gr0;`** `[ar1++] = ar0,gr0 with gr1 += gr2;`	1	1.1
Store the contents of an address register into the destination memory location with the address predecrementation.	**`[--ar1] = ar0;`** `[--ar1] = ar0 with gr1 -= gr2;`	1	1.1
Store the contents of a general-purpose register into the destination memory location with the address predecrementation.	**`[--ar1] = gr0;`** `[--ar1] = gr0 with gr1 -= gr2;`	1	1.1
Store the contents of a register pair into the destination memory location with the address predecrementation. The address is decremented by 2. (*)	**`[--ar1] = ar0,gr0;`** `[--ar1] = ar0,gr0 with gr1 += gr2;`	1	1.1
Store the contents of an address register into the destination memory location with the address postincrementation by a value of the related general-purpose register. (*)	**`[ar1++gr1] = ar0;`** `[ar1++gr1] = ar0 with gr1 -= gr2;`	1	1.1
Store the contents of a general-purpose register into the destination memory location with the address postincrementation by a value of the related general-purpose register. (*)	**`[ar1++gr1] = gr0;`** `[ar1++gr1] = gr0 with gr1 -= gr2;`	1	1.1
Store the contents of a register pair into the destination memory location with the address postincrementation by a value of the	**`[ar1++gr1] = ar0,gr0;`** `[ar1++gr1] = ar0,gr0 with gr1 += gr2;`	1	1.1

related general-purpose register. (*)			
Store the contents of an address register into the destination memory location with the address preincrementation by a value of the related general-purpose register. (*)	**[ar1+=gr1] = ar0;** [ar1+=gr1] = ar0 with gr1 -= gr2;	1	1.1
Store the contents of a general-purpose register into the destination memory location with the address preincrementation by a value of the related general-purpose register. (*)	**[ar1+=gr1] = gr0;** [ar1+=gr1] = gr0 with gr1 -= gr2;	1	1.1
Store the contents of a register pair into the destination memory location with the address preincrementation by a value of the related general-purpose register. (*)	**[ar1+=gr1] = ar0,gr0;** [ar1+=gr1] = ar0,gr0 with gr1 += gr2;	1	1.1
Store the contents of an address register into the destination memory location with preliminary initialization of an address register by a value of the related general-purpose register. (*)	**[ar1=gr1] = ar0;** [ar1=gr1] = ar0 with gr1 -= gr2;	1	1.1
Store the contents of a general-purpose register into the destination memory location with preliminary initialization of an address register by a value of the related general-purpose register. (*)	**[ar1=gr1] = gr0;** [ar1=gr1] = gr0 with gr1 -= gr2;	1	1.1
Store the contents of a register pair into the destination memory location with preliminary initialization of an address register by a value of the related general-purpose register. (*)	**[ar1=gr1] = ar0,gr0;** [ar1=gr1] = ar0,gr0 with gr1 += gr2;	1	1.1
Store the contents of an address register into the destination memory location with preliminary initialization of an address register by the constant expression.	**[ar1=Const_Addr] = ar0;** ar0 = [ar1=Const] with gr1 -= gr2;	2	1.2
Store the contents of a general-purpose register into the destination memory location with preliminary initialization of an address register by the constant expression.	**[ar1=Const_Addr] = gr0;** [ar1=Const] = gr0 with gr1 -= gr2;	2	1.2
Store the contents of a register pair into the destination memory location with preliminary initialization of an address register by the constant expression. (*)	**[ar1=Const_Addr] = ar0,gr0;** [ar1=Const] = ar0,gr0 with gr1 += gr2;	2	2
Store the contents of an address register into the destination memory location with the address	**[ar1+=Const] = ar0;** **[ar1-=Const] = ar0;**	2	1.2

preincrementation by the constant expression.	`[ar1+=Const] = ar0` with `gr1 -= gr2;`		
Store the contents of a general-purpose register into the destination memory location with the address preincrementation by the constant expression.	**`[ar1+=Const] = gr0;`** **`[ar1-=Const] = gr0;`** `[ar1+=Const] = gr0` with `gr1 -= gr2;`	2	1.2
Store the contents of a register pair into the destination memory location with the address preincrementation by the constant expression. (*)	**`[ar1+=Const] = ar0,gr0;`** **`[ar1-=Const] = ar0,gr0;`** `[ar1+=Const] = ar0,gr0` with `gr1;`	2	1.2

Note *In the instructions marked with (*) register pairs are used. They are formed by address registers and general-purpose registers with the same number, for instance: `(ar3, gr3)` or `(ar5, gr5)`. Registers with different numbers cannot form a register pair. A register pair contains a 64-bit word. The high thirty-two bits are always located in the address register, the low bits are in the general-purpose register. This thing does not depend on the order of the registers in an assembly instruction. For more information about register pairs see 3.1.3 on page 3-4.*

Note *The type of a store instruction: store 64 bits or 32 bits, is defined by a source register.*

5.1.4 Stack Access Commands

Stack access commands are located only in the left part of an assembly instruction. Only contents of the read/write accessible registers can be stored into the system stack and restored from the stack (see Table 5-1).

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Push the contents of an address register onto the top of the system stack.	**`push ar0;`** `push ar0` with `gr7 = gr1 + gr2;`	1	1.1
Push the contents of a general-purpose register onto the top of the system stack.	**`push gr0;`** `push gr0` with `gr7 = gr0;`	1	1.1
Push the contents of a register pair onto the top of the system stack. (*)	**`push ar0,gr0;`** `push ar0,gr0` with `gr1 = not gr1;`	1	1.1
Pop the contents of the top of the system stack into an address register.	**`pop ar0;`** `pop ar0` with `gr7 = gr0 << 2;`	1	1.1
Pop the contents of the top of the system stack into a general-purpose register.	**`pop gr0;`** `pop gr0` with `gr7 = gr0;`	1	1.1

Pop the contents of the top of the system stack into a register pair. (*)	**`pop ar0,gr0;`** `pop ar0,gr0 with gr1 += gr2;`	1	1.1
Remove a value from the top of the system stack.	**`pop;`** `pop with gr7 -= gr0;`	1	1.1

Note *In the instruction marked with (*) a 64-bit word is stored at the top of the system stack keeping the top even. The requirement to the stack top to be even arises from the fact that in case an interrupt occurs and a program returns from an interrupt the program can return to the correct address only if the stack top was even at the moment of the interrupt. Since an interrupt can occur at any moment, it is necessary to always keep the stack top even. That's why while working with the system stack it is recommended to push/pop register pairs. It takes the same time to push/pop a register pair as a single register.*

5.1.5 Register Copy Commands

This group of commands copies the contents of a source register to a destination register. Any read accessible register can be used as the source register. Any write accessible register can be used as the destination register.

Note *Memory to memory copy commands are not supported by the processor. Constant to register copy commands are referred to as 'register initialization with a constant' commands (see 5.1.6 on page 5-15).*

Table 5-1 contains a list of read/write accessible registers and register pairs of the NeuroMatrix® NM6403:

Table 5-1. List of Read/Write Accessible Registers and Register Pairs

REGISTERS		REGISTER PAIRS
ar0, ... ar7(sp)	gr0, ... gr7	(ar0, gr0), ... (ar7, gr7)
icc0	ica0	(icc0, ica0)
icc1	ica1	(icc1, ica1)
occ0	oca0	(occ0, oca0)
occ1	oca1	(occ1, oca1)
t0	t1	(t0, t1)
pswr	pc	(pswr, pc)
lmicr	gmicr	-

Any register and register pair given in the table above can be both the source and the destination.

Table 5-2 contains a list of write accessible registers of the NeuroMatrix® NM6403:

Table 5-2. List of Write Accessible Registers

32-BIT REGISTERS		64-BIT REGISTERS
nb1l	nb1h	nb1
sbl	sbh	sb
f1crl	f1crh	f1cr
f2crl	f2crh	f2cr
vrl	vrh	vr

All write accessible registers are the Vector Unit control registers. The Vector Unit registers are 64 bits long. But it is possible to access their 32-bit low- and high-order parts separately. The low-order parts of all vector registers are marked with index l and the high-order parts – with index h.

The only read-only register is a 32-bit `intr`.

Copy commands are located only in the left part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Copy the contents of a source address register to a destination address register. (*)	**ar2 = ar0;** **ar2 = ar0 set;** `ar2 = ar0 with gr1 = gr2 and gr3;`	1	2.1
Copy the contents of an address register to a general-purpose register.	**gr0 = ar3;** `gr0 = ar3 with gr1 = gr0 A>> 1;`	1	2.1
Copy the contents of a general-purpose register to an address register. (*)	**ar0 = gr5;** **ar0 = gr5 set;** `ar0 = gr5 with gr1 = gr0 and gr5;`	1	2.1
Copy the contents of a source general-purpose register to a destination general-purpose register.	**gr0 = gr5;** `gr0 = gr5 with gr1 = gr0 or gr1;`	1	2.1
Copy the contents of a source register pair to a destination register pair.	**ar0,gr0 = ar4,gr4;** `ar0,gr0 = ar4,gr4 with not gr1;`	1	2.1
Copy the contents of an address register pair to a register pair. The same contents copies to both destination registers. (**)	**ar0,gr0 = ar5;** `ar0,gr0 = ar5 with gr1++;`	1	2.1
Copy the value of a general-purpose register to a register pair. After this operation both destination registers have the same contents. (**)	**ar0,gr0 = gr4;** `ar0,gr0 = gr4 with gr2 = gr1 - 1;`	1	2.1
Copy the contents of an address register to the low/high part (32-bit) of a special	**nb1l = ar5;**	1	2.1

vector control register.	`nb1l = ar5 with gr0 += gr1;`		
Copy the contents of a general-purpose register to the low/high part (32-bit) of a special vector control register.	**`vrh = gr4;`** `vrh = gr4 with gr1 = gr1 << 3;`	1	2.1
Copy the contents of an address register to a special vector control register (64-bit). The same contents copies to the high and low parts of the destination register. (**)	**`sb = ar5;`** `sb = ar5 with gr0 -= gr1;`	1	2.1
Copy the contents of a general-purpose register to a special vector control register (64-bit). The same contents copies to the high and low parts of the destination register. (**)	**`f1cr = gr4;`** `f1cr = gr4 with gr1 = not gr1;`	1	2.1
Copy the contents of a register pair to a special vector control register (64 bit). The contents of the address register copies to the high part of the destination register.	**`nb1 = ar5,gr5;`** `nb1 = ar5,gr5 with gr0 -= gr1;`	1	2.1
Copy the contents of a special register to an address register.	**`ar5 = lmicr;`** `ar5 = lmicr with gr0 = - gr1;`	1	2.1
Copy the contents of a special register to a general-purpose register.	**`gr7 = t1;`** `gr7 = t1 with gr1 = not gr1;`	1	2.1
Copy the contents of a special register to a register pair. The same contents copies to both destination registers. (**)	**`ar5,gr5 = ica1;`** `ar5,gr5 = ica1 with gr0 = - gr1;`	1	2.1

Note *Copy commands allow the user to copy the contents of one address register to another inspite of they may belong to different address register groups (see 3.1.1 on page 3-3).*

Note *Some of copy commands are very similar to address register modification commands. The difference is that the assembler translates them into different processor instruction words. The modifier '`set`' is used to specify the processor instruction word. If this modifier is set up, the assembler will always translate this command to the instruction 2.1/2.2. The modifier 'set' can be omitted because the assembler translates it to a copy command by default.*

Note *In the instructions marked with (**) processor copy the contents of a 32-bit register to a special vector control register (64-bit) or to a register pair. After this operation both destination registers or high and low parts of a special vector control register have the same contents.*

5.1.6 Register Initialization with Constant

Commands of register initialization with a constant or a constant expression are used to transfer a 32-bit constant to a destination register. The destination register can be any write accessible register (see Table 5-1 and Table 5-2).

Commands of register initialization with a constant are located only in the left part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Initialization of an address register with a constant (*)	**ar2 = Const;** **ar2 = Const set;** ar2 = Const with gr1--;	2	2.2
Initialization of a general-purpose register with a constant.	**gr0 = Const;** gr0 = Const with gr1 = gr0 A>> 1;	2	2.2
Initialization of a register pair with a constant. (**)	**ar2,gr2 = Const;** ar2,gr2 = Const with gr1++;	2	2.2
Initialization of a special register with a constant.	**occ1 = Const;** occ1 = Const with gr1 = - gr2;	2	2.2
Initialization of a low/high part (32-bit) of a vector control register with a constant.	**sbl = Const;** gr0 = Const with gr1 = gr0 >> 12;	2	2.2
Initialization of a vector control register (64-bit) with a 32-bit constant. The constant copies to both low and high parts of the destination register. (**)	**nb1 = Const;** nb1 = Const with gr1 = gr2 - 1;	2	2.2

Note *Some of copy commands are very similar to address register modification commands. The difference is that the assembler translates them into different processor instruction words. The modifier '`set`' is used to specify the processor instruction word. If this modifier is set up, the assembler will always translate this command to the instruction 2.1/2.2. The modifier 'set' can be omitted because the assembler translates it to a copy command by default.*

Note *In the instructions marked with (**) processor copy the contents of a 32-bit register to a special vector control register (64-bit) or to a register pair. After this operation both destination registers or high and low parts of a special vector control register have the same contents.*

5.1.7 Address Register Modification Commands

Address register modification commands are located only in the left part of an assembly instruction.

The address register modification commands are sensitive to the address register groups (see 3.1.1 on page 3-3). All address registers are divided into two groups. The first group contains the registers `ar0,...,ar3` while the second one contains the registers `ar4,...,ar7`. The division is due to two address generators (DAG) of the processor (see Figure 1-3). The first group of registers is mapped to the DAG1, the second group to the DAG2.

Address registers from the same group can stand in a modification command but not from different groups. For example, the instruction

```
ar0 = ar2+gr2; // correct command
```

is correct while the instruction

```
ar0 = ar4+gr4; // error command
```

will cause the compilation error.

Division into the groups does not concern general-purpose registers, i.e. any address register can be modified by any general-purpose register.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Modification of a destination address register with the contents of a sources address register. (*)	**`ar3 = ar0 addr;`** `ar3 = ar0 addr with gr1 = gr2 and gr3;`	1	3.1
Modification of a destination address register with the contents of a source general-purpose register. (*)	**`ar0 = gr7 addr;`** `ar0 = gr7 addr with gr1 = gr2 xor gr3;`	1	3.1
Modification of a destination address register with the sum of registers of a register pair.	**`ar5 = ar6+gr6;`** `ar5 = ar6+gr6 with gr1 = gr2 - gr3;`	1	3.1
Modification of a destination address register with the sum of an address register and a constant.	**`ar4 = ar6+Const;`** `ar4 = ar6+Const with gr1 = gr2 C>> 1;`	2	3.2
Modification of an address register with a constant. (*)	**`ar1 = Const addr;`** `ar5 = ar6+Const with gr1 = gr2;`	2	3.2
Incrementation of an address register.	**`ar4++;`** `ar4++ with gr1 += gr2;`	1	3.1
Decrementation of an address register.	**`ar4--;`** `ar4-- with gr1 = - gr2;`	1	3.1

Incrementation of an address register with the contents of the related general-purpose register. This command is valid only for registers of a register pair.	**ar2+=gr2;** ar2+=gr2 with gr1 = not gr2;	1	3.1
Incrementation of an address register with a constant.	**ar4+=Const;** ar4+=Const with gr1 -= gr2;	2	3.2
Decrementation of an address register with a constant.	**ar4-=Const;** ar2-=Const with gr1;	2	3.2

Note

Some of address register modification commands are very similar to copy commands (see 5.1.5 on page 5-12). The difference is that the assembler translates them into different processor instruction words. The commands marker with (*) contain the modifier 'addr', which is used to specify the processor instruction word. If this modifier is set, the assembler will always translate this command to the mashine instruction 3.1/3.2 (see 4.3 on page 4-6).

5.1.8 Register pswr Modification Commands

The register pswr describes the processor status.

Although pswr is a read/write accessible register, there are special assembly instructions to modify it.

The pswr modification instructions are located in the left part of a scalar instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Set the desired bits of the pswr register, i.e. perform a bitwise logical OR of the contents of the pswr register and a constant.	**pswr set Const;** pswr set Const with gr1++;	2	2.3
Clear the desired bits of the pswr register, i.e. perform a bitwise logical AND NOT of the contents of the pswr register and a constant	**pswr clear Const;** pswr clear Const with gr1--;	2	2.3

5.1.9 Branch Commands

The NeuroMatrix® NM6403 supports the following types of branch commands:

- Conditional and unconditional branch;
- Conditional and unconditional sub-routine call;
- Conditional and unconditional return from a sub-routine;

- Conditional and unconditional return from an interrupt.

All of branch commands can be standard or delayed. To distinguish the delayed branch command from the standard the reserved word `delayed` is written before the branch command.

Processor will execute a delayed jump if the reserved word delayed is found in the assembly instruction.

Conditional branch is executed according to the flags preset in pswr. The flags are set by the operation executed before the current branch instruction.

Only arithmetic and logical operations in the right part of an assembly instruction affect the flags status.

5.1.9.1 Branch Unconditionally

Commands of unconditional branch are located only in the left part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Branch unconditionally to the absolute address given by a constant.	**`goto Const_Addr;`** `goto Const_Addr with gr1++;`	2	4.2
Branch unconditionally to the address given by an address register.	**`goto ar0;`** `goto ar0 with gr1--;`	1	4.1
Branch unconditionally to the address given by a general-purpose register.	**`goto gr0;`** `goto gr0 with gr1 = -gr1;`	1	4.1
Branch unconditionally to the address given by the sum of the contents of registers of a register pair.	**`goto ar0+gr0;`** `goto ar0+gr0 with gr1 = not gr1;`	1	4.1
Branch unconditionally to the address given by the sum of the contents of an address register and a constant.	**`goto ar0+Const;`** **`goto ar0-Const;`** `goto ar0+Const with gr1 = gr1 << 2;`	2	4.2
Relative branch unconditionally to the address given by a constant.	**`skip Const_Addr;`** `skip Const_Addr with gr1 = gr2 and gr3;`	2	4.2
Relative branch unconditionally to the address given by a general-purpose register.	**`skip gr0;`** `skip gr0 with gr1--;`	1	4.1
Branch unconditionally, delayed. (*)	**`delayed goto gr0;`** `delayed goto gr0 with gr1++;`	1 or 2	4.1, 4.2

Note *The instruction marked with (*) shows an example of a delayed unconditional branch syntax. The reserved word 'delayed' can be used*

with any unconditional branch described above.

The separation of the delayed branch is introduced by a workaround for a simplified programming. From a branch command fetching to a jump execution it is elapse from one to three cycles. This time the processor has time to extra fetch up to three instructions. These instructions are named delayed.

Placed after a branch command the delayed instructions are performed before the execution of the jump. They are performed at any case independently of a branch condition performance.

An accurate number of delayed instructions depends on: its length, a memory location and a type of the branch command.

5.1.9.2 Sub-Routine Call

The sub-routine call commands listed below are unconditional commands. To provide a conditional call it is necessary to put a condition expression in front of the call command, for example:

```
call MyFunc;          // call unconditionally
if =0 call MyFunc;    // call conditionally
```

A set of condition expressions can be found in paragraph 5.1.9.4 on page 5-20.

Sub-routine call commands are located only in the left part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Sub-routine call unconditionally. A label defines the address of the sub-routine first instruction.	**`call Const_Addr;`** `call Const_Addr with gr1++;`	2	4.2
Sub-routine call unconditionally. An address register contains the address of the sub-routine first instruction.	**`call ar0;`** `call ar0 with gr1--;`	1	4.1
Sub-routine call unconditionally. A general-purpose register contains the address of the sub-routine first instruction.	**`call gr0;`** `call gr0 with gr1 = -gr1;`	1	4.1
Sub-routine call unconditionally. The address of the sub-routine first instruction is given by the sum of registers of a register pair.	**`call ar0+gr0;`** `call ar0+gr0 with gr1 = not gr1;`	1	4.1
Sub-routine call unconditionally. The address of the sub-routine first instruction is given by the sum of the	**`call ar0+Const;`** **`call ar0-Const;`**	2	4.2

contents of an address register and a constant expression.	`goto ar0+Const` with `gr1 = gr1 << 2;`		
Relative sub-routine call unconditionally. A label defines the address of the sub-routine first instruction.	**`callrel Const_Addr;`** `callrel Const_Addr` with `gr1++;`	2	4.2
Relative sub-routine call unconditionally. A register defines the address of the sub-routine first instruction.	**`callrel gr0;`** `callrel gr0` with `gr1--;`	1	4.1
Sub-routine call, delayed.(*)	**`delayed call gr0;`** `delayed call gr0` with `gr1++;`	1 or 2	4.1, 4.2

Note *The instruction marked with (*) shows an example of a delayed unconditional sub-routine call syntax.. For more details see the note of the preceding section 5.1.9.1.*

5.1.9.3 Return from Sub-Routine/Interrupt

Commands of return from a subroutine/interrupt are located only in the left part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Return from a sub-routine.	**`return;`** `return` with `gr7 = gr1 + gr2;`	1	4.3
Return from an interrupt.	**`ireturn;`** `ireturn` with `gr7 = gr0;`	1	4.3
Return from a sub-routine, delayed.	**`delayed return;`** `delayed return` with `gr7 = gr1 + gr2;`	1	4.3
Return from an interrupt, delayed.	**`delayed ireturn;`** `delayed ireturn` with `gr7 = gr0;`	1	4.3

5.1.9.4 Branch Conditions

All conditional branches occur or do not occur depending on the flags that were preliminary set in pswr register by one of the previous instructions.

The flags are set only by arithmetic/logical operations in the right part of a scalar instruction.

Thus, in order to make the conditional branch possible, the scalar arithmetic, logical or shift operation is executed before the branch. This

operation sets the flags and makes possible the conditional branch execution. For example:

```
begin "text"
...
gr2 = [ar0++] with gr0--;  // Operation setting the condition flags.
if > goto Label;           // Conditional branch to the Label.
...
end "text";
```

Commands of conditional branch are located only in the left part of an assembly instruction.

The column "Syntax" contains the underlined part of the instruction showing the condition syntax. In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	FLAGS
Zero	**`if =0 goto ...;`** `if =0 goto Label with gr1++;`	Z
Nonzero	**`if <>0 goto ...;`** `if <>0 goto gr0 with gr1--;`	~Z
Greater than	**`if > delayed goto ...;`** `if > delayed goto ar0 with gr1--;`	~Z AND ~N
Less than	**`if < skip ...;`** `if < skip Label with gr1--;`	N
Greater than or equal to	**`if >= call ...;`** `if >= call Label with gr1--;`	~N
Less than or equal to	**`if <= callrel ...;`** `if <= callrel Label with gr1--;`	N OR Z
Unsigned higher than or same as	**`if u>= goto ...;`** `if >= goto Label with gr7 -= gr1;`	~C
Unsigned lower than	**`if u< return ...;`** `if u< return with gr7 = gr1 noflags;`	C
Carry	**`if carry call ...;`** `if carry call ar0 with gr7 -= gr1;`	C
No carry	**`if not carry return ...;`** `if not carry return with gr7 = gr1;`	~C
Overflow	**`if vtrue call ...;`** `if vtrue call ar0 with gr7 -= gr1;`	V
No overflow	**`if vfalse return ...;`** `if vfalse return with gr7 = gr1;`	~V

Greater than with overflow	**if v> goto ...;** if v> goto ar0 with gr7 -= gr1;	~ ((N XOR V) OR Z)
Less than with overflow	**if v< delayed goto ...;** if v< delayed goto gr1 with gr7 = gr1;	N XOR V
Greater than-equal sign with overflow bit check.	**if v>= callrel ...;** if v>= callrel gr0 with gr7 -= gr1;	~ (N XOR V)
Less than-equal sign with overflow bit check.	**if v<= ireturn ...;** if v<= ireturn with gr7 = gr1;	(N XOR V) OR Z

5.1.10 Set of Basic Scalar Operations

All the processor scalar operations are located in the right part of an assembly instruction, i.e. after the reserved word '`with`'. Only general-purpose registers can be used in scalar operations.

Scalar operations are three-operand operations. Any general-purpose register can be both source and destination operand of a scalar expression, for example:

- `gr0 = gr1 + gr2;` - the destination operand;
- `gr1 = gr0 + gr2;` - the first source operand;
- `gr0 = gr0 + gr0;` - the destination and the source operand.

The processor supports three forms of scalar operations:

- a standard scalar expression:
  `gr1 = gr2 + gr3;`
  This form of an expression has the source and the destination operands. It changes the contents of the destination general-purpose register and sets the condition flags.
- a scalar expression that does not set the flags:
  `gr1 = gr2 + gr3 noflags;`
  This form of expression has the source and the destination operands. It changes the contents of the destination general-purpose register but does not change the condition flags. The modifier 'noflags' is used to notify that the scalar operation does not change the flags. This modifier can be applied to all scalar operations with the exception the shift operations.
- A scalar expression that that does not have the destination operand:
  `gr2 + gr3;`
  or just
  `gr2;`
  This form of an expression is used to set the condition flags. No one of the registers participating in the expression changes the contents. The form is not allowed for shift operations.

5.1.11 Arithmetic Operations

Arithmetic operations are located only in the right part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty left part is given.

DESCRIPTION	SYNTAX	TYPE
Add the contents of two source registers and store the sum in the destination register.	**gr0 = gr1 + gr2;** ar0 = gr0 with gr0 = gr1 + gr2;	6.3

Add the source operand to the contents of the destination register and store the sum in the destination register.	**gr0 += gr1;**
	equivalent to: gr0 = gr0 + gr1;
	ar0 = 100h with gr0 += gr1;
		6.3

Add one to the contents of the source register and store the sum in the destination register.	**gr1 = gr2 + 1;** ar1+=gr1 with gr1 = gr2 + 1;	6.3

Increment the contents of the register by one.	**gr1++;**
	equivalent to: gr1 = gr1 + 1;
	ar1++ with gr1++;
		6.3

Subtract the second source operand from the contents of the first source register and store the result in the destination register.	**gr1 = gr0 - gr7;** [ar1++] = ar0 with gr1 = gr0 - gr7;	6.3
Subtract the source operand from the contents of the destination register and store the result in the destination register.	**gr1 -= gr7;** equivalent to: gr1 = gr1 - gr7; [--ar1] = gr0 with gr1 -= gr7;	6.3
Subtract one from the contents of the source register and store the result in the destination register.	**gr1 = gr2 - 1;** call gr4 with gr1 = gr2 - 1;	6.3
Decrement the contents of the register by one.	**gr1--;** equivalent to: gr1 = gr1 - 1; ar1-- with gr1--;	6.3
Add carry bit to the contents of the source register and store the result in the destination register.	**gr1 = gr2 + carry;** ar4 += gr4 with gr1 = gr2 + carry;	6.3
Add carry bit to the sum of the source registers and store the result in the destination register.	**gr1 = gr2 + gr6 + carry;** ar4++ with gr1 = gr2 + gr6 + carry;	6.3
Subtract carry bit from the contents of the source register and store the result in the destination register.	**gr1 = gr2 - 1 + carry;** ar6 -= gr6 with gr1 = gr2 - 1 + carry;	6.3
Subtract bit from the difference of the source registers and store the result in the destination register.	**gr1 = gr2 - gr6 - 1 + carry;** ar4-- with gr1 = gr2 - gr6 - 1 + carry;	6.3

Load the difference between 0 and the source operand into the destination register.	**gr1 = - gr5;** goto gr4 with gr1 = - gr5;	6.3
Make the first step of multi-step multiplication. The second source operand should always be gr7.	**gr1 = gr2 *: gr7;** ar4,gr4 = gr0 with gr1 = gr2 *: gr7;	6.3
Make next steps of multi-step multiplication. The second source operand should always be gr7.	**gr1 = gr2 * gr7;** [ar6] = gr6 with gr1 = gr2 * gr7;	6.3
An example of an arithmetic operation that does not change the condition flags.	**gr1 = gr2 + gr2 noflags;** ar5++ with gr1 = gr2 + gr2 noflags;	6.3
An example of an arithmetic operation that sets the condition flags, but does not change the contents of the registers involved.	**gr1 + gr2;** [ar1++] = AAA with gr1 + gr2;	6.3

5.1.12 Logical Operations

Logical operations are located only in the right part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty left part is given.

DESCRIPTION	SYNTAX	TYPE
Load the bitwise logical OR of the source registers to the destination register.	**gr0 = gr1 or gr2;** ar0 = gr0 with gr0 = gr1 or gr2;	6.2
Load the bitwise logical AND of the source registers to the destination register.	**gr1 = gr2 and gr3;** ar0 = 100h with gr1 = gr2 and gr3;	6.2
Perform a bitwise exclusive OR of two source operands and store the result in the destination register.	**gr2 = gr3 xor gr4;** ar0 = gr0 with gr2 = gr3 or gr4;	6.2
Load the bitwise logical complement of the source register to the destination register.	**gr1 = not gr2;** ar0 = 100h with gr1 = not gr2;	6.2
Perform a bitwise logical OR of the second register and the bitwise logical complement of the first register, and store the result in the destination register.	**gr0 = not gr1 or gr2;** ar0++ with gr0 = not gr1 or gr2;	6.2
Perform a bitwise logical OR of the first register and the bitwise logical complement of the second register, and store the result in the destination register.	**gr1 = gr2 or not gr3;** ar6-- with gr1 = gr2 or not gr3;	6.2
Perform a bitwise logical OR of the bitwise logical complement of the first register and the bitwise logical complement of the second register, and store the result in the destination register.	**gr1 = not gr2 or not gr3;** return with gr1 = not gr2 or not gr3;	6.2
Perform a bitwise logical AND of the second register and the bitwise logical	**gr2 = not gr3 and gr4;**	6.2

complement of the first register, and store the result in the destination register.	`ar4+=gr4 with gr0 = not gr1 and gr2;`	
Perform a bitwise logical AND of the first register and the bitwise logical complement of the second register, and store the result in the destination register.	**`gr3 = gr4 and not gr5;`** `[ar6]=gr6 with gr3 = gr4 and not gr5;`	6.2
Perform a bitwise logical AND of the bitwise logical complement of the first register and the bitwise logical complement of the second register, and store the result in the destination register.	**`gr3 = not gr4 and not gr5;`** `[ar6]=gr6 with gr3 = gr4 and not gr5;`	6.2
Perform a bitwise exclusive OR of the second register and the bitwise logical complement of the first register, and store the result in the destination register.	**`gr2 = not gr3 xor gr4;`** `[Addr]=gr0 with gr2 = not gr3 xor gr4;`	6.2
Perform a bitwise exclusive OR of the first register and the bitwise logical complement of the second register, and store the result in the destination register.	**`gr3 = gr4 xor not gr5;`** `ar0 = ar2 with gr2 = gr4 xor not gr5;`	6.2
Set all bits to zero in the destination register.	**`gr6 = false;`** `[ar0] = ar5,gr5 with gr6 = false;`	6.2
Set all bits to 1 in the destination register.	**`gr0 = true;`** `ar0=[ar2=10h] with gr0 = true;`	6.2
Copy the contents of the source general-purpose register to the destination general-purpose register. (*)	**`with gr2 = gr4;`** `gr0 = gr5 with gr2 = gr4;`	6.2

Note

The operation marked with (*) uses the reserved word 'with' in front of it, otherwise the compiler will regard the instruction as a copy command and put it to the left part of an assembly instruction. In that case it will not change the condition flags state because the commands executable in the left part of the assembly instruction do not affect the flags. Compare:

```
gr2 = gr3;      // command of copying in the left part
                // of the instruction (for example:
                // gr2 = gr3 with gr4 += gr5;)
```

```
with gr2 = gr3; // logical operation in the right part
                // of the instruction (for example:
                // [ar0+=5] = ar7  with gr2 = gr3;)
```

5.1.13 Flags Setting Operations

This section gives arithmetic and logical expressions that affect the condition flags but do not change the contents of the registers involved.

All the expressions given below are located only in the right part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty left part is given.

DESCRIPTION	SYNTAX	TYPE
Set the flags according to the contents of the register.	**gr0;** `if > goto gr2 with gr0;`	6.2
Calculates the difference between 0 and the contents of the register and set the flags according to the result of this operation.	**- gr1;** `[ar4++gr4] = ar2,gr2 with - gr1;`	6.3
Add the contents of the second register to the contents of the first register and set the flags according to the result of this operation.	**gr2 + gr3;** `ar0 = 100h with gr2 + gr3;`	6.3
Add one to the contents of the source register and set the flags according to the result of this operation.	**gr4 + 1;** `gr2++ with gr4 + 1;`	6.3
Subtract the contents of the second register from the contents of the first register and set the flags according to the result of this operation.	**gr0 - gr7;** `ar0 += gr0 with gr0 + gr7;`	6.3
Subtract one from the contents of the source register and set the flags according to the result of this operation.	**gr2 - 1;** `gr7 = [ar0++gr0] with gr2 - 1;`	6.3
Add carry bit to the contents of the source register and set the flags according to the result of this operation.	**gr1 + carry;** `ar4 -= gr4 with gr1 + carry;`	6.3
Add carry bit to the sum of the source registers and set the flags according to the result of this operation.	**gr2 + gr6 + carry;** `ar4++ with gr2 + gr6 + carry;`	6.3
Subtract carry bit from the contents of the source register and set the flags according to the result of this operation.	**gr2 - 1 + carry;** `goto Addr with gr2 - 1 + carry;`	6.3
Subtract bit from the difference of the source registers and set the flags according to the result of this operation.	**gr2 - gr6 - 1 + carry;** `ar4-- with gr2 - gr6 - 1 + carry;`	6.3
Make the bitwise logical OR of the source registers and set the flags according to the result of this operation.	**gr1 or gr2;** `ar0 = gr0 with gr1 or gr2;`	6.2
Make the bitwise logical AND of the source registers and set the flags according to the result of this operation.	**gr2 and gr3;** `ar0 = 100h with gr2 and gr3;`	6.2
Make the bitwise exclusive OR of the source registers and set the flags according to the result of this operation.	**gr3 xor gr4;** `ar0 = gr0 with gr3 or gr4;`	6.2
Make the bitwise logical complement of the source register and set the flags according to the result of this operation.	**not gr2;** `ar0 = 100h with not gr2;`	6.2
Perform a bitwise logical OR of the second register and the bitwise logical complement of the first register, and set the flags according to the result of this operation.	**not gr1 or gr2;** `ar0++ with not gr1 or gr2;`	6.2

Perform a bitwise logical OR of the first register and the bitwise logical complement of the second register, and set the flags according to the result of this operation.	**gr2 or not gr3;** ar6-- with gr2 or not gr3;	6.2
Perform a bitwise logical OR of the bitwise logical complement of the first register and the bitwise logical complement of the second register, and set the flags according to the result of this operation.	**not gr2 or not gr3;** return with not gr2 or not gr3;	6.2
Perform a bitwise logical AND of the second register and the bitwise logical complement of the first register, and set the flags according to the result of this operation.	**not gr3 and gr4;** ar4+=gr4 with not gr1 and gr2;	6.2
Perform a bitwise logical AND of the first register and the bitwise logical complement of the second register, and set the flags according to the result of this operation.	**gr4 and not gr5;** [ar6]=gr6 with gr4 and not gr5;	6.2
Perform a bitwise logical AND of the bitwise logical complement of the first register and the bitwise logical complement of the second register, and set the flags according to the result of this operation.	**not gr4 and not gr5;** [--ar6]=gr2 with gr4 and not gr5;	6.2
Perform a bitwise exclusive OR of the second register and the bitwise logical complement of the first register, and set the flags according to the result of this operation.	**not gr3 xor gr4;** [Addr]=gr0 with not gr3 xor gr4;	6.2
Perform a bitwise exclusive OR of the first register and the bitwise logical complement of the second register, and set the flags according to the result of this operation.	**gr2 xor not gr7;** [ar0++]=gr0 with gr2 xor not gr7;	6.2
Set bit N, clear others bits.	**true;** ar0=[ar2=10h] with true;	6.2
Set bit Z, clear others bits.	**false;** [ar0] = ar5,gr5 with false;	6.2

5.1.14 Shift Operations

Shift operations are located only in the right part of an assembly instruction.

In addition to instruction syntax (bold) an example of a scalar instruction with the non-empty left part is given.

DESCRIPTION	SYNTAX	TYPE
Left-shift the contents of the source register by an arbitrary number of bits and store the result in the destination register. Shift constant is from 1 to 31. Low-order bits are filled with zeros and the high-order bits are shifted out through the carry bit.	**gr0 = gr1 << 10;** ar0 = gr0 with gr0 = gr1 << 10;	6.1

Left-shift the contents of the destination register by an arbitrary number of bits and store the result back into the destination register (reduced notation). Shift constant is from 1 to 31. Low-order bits are filled with zeros and the high-order bits are shifted out through the carry bit.	**gr0 <<= 2;** equivalent to: gr0 = gr0 << 2; [ar0] = gr0 with gr0 <<= 2;	6.1
Logical right-shift the contents of the source register by arbitrary bits and store the result in the destination register. Shift constant is from 1 to 31. High-order bits are filled with zeros and the low-order bits are shifted out through the carry bit.	**gr1 = gr2 >> 24;** ar0 = 100h with gr1 = gr2 >> 24;	6.1
Logical right-shift the contents of the destination register by arbitrary bits and store the result back into the destination register (reduced notation). Shift constant is from 1 to 31. High-order bits are filled with zeros and the low-order bits are shifted out through the carry bit.	**gr1 >>= 3;** equivalent to: gr1 = gr1 >> 3; [ar0++] = gr3 with gr1 >>= 3;	6.1
Arithmetical right-shift the contents of the source register by arbitrary bits and store the result in the destination register. Shift constant is from 1 to 31. High-order bits are sign-extended as they are right-shifted, and the low-order bits are shifted out through the carry bit.	**gr2 = gr3 A>> 5;** ar0 += gr0 with gr2 = gr3 A>> 5;	6.1

Arithmetical right-shift the contents of the destination register by arbitrary bits and store the result back into the destination register (reduced notation). Shift constant is from 1 to 31. High-order bits are sign-extended as they are right-shifted, and the low-order bits are shifted out through the carry bit.	**gr2 A>>= 9;**
	equivalent to: gr2 = gr2 A>> 9;
	skip Addr with gr2 A>>= 9;
		6.1

Rotate the contents of the source register left arbitrary bits and store the result in the destination register. Shift constant is from 1 to 31.	**gr3 = gr4 R<< 6;** ar0 -= gr0 with gr3 = gr4 R<< 6;	6.1

Rotate the contents of the destination register left arbitrary bits and store the result back into the destination register (reduced notation). Shift constant is from 1 to 31.	**gr3 R<<= 12;**
	equivalent to: gr3 = gr3 R<< 12;
	goto gr0 with gr3 R<<= 12;
		6.1

Rotate the contents of the source register right arbitrary bits and store the result in the destination register. Shift constant is from 1 to 31.	**gr3 = gr4 R>> 7;** [--ar0] = gr0 with gr3 = gr4 R>> 7;	6.1
Rotate the contents of the destination register right arbitrary bits and store the result back into the destination register (reduced notation). Shift constant is from 1 to 31.	**gr3 R>>= 14;** equivalent to: gr3 = gr3 R>> 14; ar5–with gr3 R>>= 14;	6.1
Rotate the contents of the source register left one bit through the carry bit and store the result into the destination register. Shift value must be 1.	**gr4 = gr5 C<< 1;** ar0 -= gr0 with gr4 = gr5 C<< 1;	6.1

Rotate the contents of the destination register left one bit through the carry bit and store the result back into the destination register (reduced notation). Shift value must be 1.	**gr4 C<<= 1;**
	equivalent to: gr4 = gr4 C<< 1;
	gr0 = gr7 with gr4 C<<= 1;
		6.1

Rotate the contents of the source register right one bit through the carry bit and store the result into the destination register. Shift value must be 1.	**gr5 = gr6 C>> 1;** [ar0+=2] = gr0 with gr5 = gr6 C>> 1;	6.1

Rotate the contents of the destination register right one bit through the carry bit and store the result back into the destination register (reduced notation). Shift value must be 1.	**gr5 C>>= 1;**
	equivalent to: gr5 = gr5 C>> 1;
	ireturn with gr5 C>>= 1;
		6.1

5.2 Vector Instructions

This section gives the complete list of the processor scalar instructions with brief comments.

All vector instructions of the processor are grouped into tables according to their functionality. The first column explains the instruction functionality. The second one describes the instruction syntax. The third one contains the size of an instruction and the fourth gives a type of the instruction code.

Since the processor instructions contain the left and the right parts, the discussed command or the operation is underlined. Non-underlined parts of an instruction are actual commands or operations given to retain the idea of the instruction structure.

5.2.1 Data Load and Store in Vector Instructions

Commands of memory access are located only in the left part of a vector instruction.

At the memory access a number of loaded/stored long words equals to a number of repetitions which are specified in the instruction (`rep` *`number_of_repetitions`*).

All the registers participating in memory addressing must contain even values, because the vector instructions process only 64-bit words of data.

In addition to instruction syntax (bold) an example of an actual vector instruction with the non-empty right part is given.

Load from Memory Commands

DESCRIPTION	SYNTAX	SIZE	TYPE
Load the contents of the source memory location into the Vector Unit to process data on the Active Matrix or on the Vector ALU. According to repetitions counter the appropriate number of long words are loaded from the same memory location. The address register contains the source memory location address.	**data = [ar0];** rep 4 data = [ar0] with not data;	1	5.1

Load the contents of the source memory location into the `ram` buffer. According to repetitions counter the appropriate number of long words are loaded from the same memory location. The general-purpose register contains the source memory location address. (*)	**`ram = [gr2];`** `rep 12 ram = [gr2] with data + 0;`	1	5.1
Load the contents of the source memory location into the `ram` buffer. According to repetitions counter the appropriate number of long words are loaded from the same memory location. The address register is initialized with the contents of the related general-purpose register that points to the source memory location. (*)	**`ram = [ar4=gr4];`** `rep 1 ram = [ar4=gr4] with data + 1;`	1	5.1
Load the contents of the source memory location into the Vector Unit. According to repetitions counter the appropriate number of long words are loaded from the source memory location with postincrementation by 2.	**`data = [ar0++];`** `rep 32 data = [ar0++] with data;`	1	5.1
Load the contents of the source memory location into the Vector Unit. According to repetitions counter the appropriate number of long words are loaded from the source memory location with predecrementation by 2.	**`data = [--ar4];`** `rep 16 data = [--ar4] with data or ram;`	1	5.1
Load the contents of the source memory location into the `wfifo` buffer. According to repetitions counter the appropriate number of long words are loaded from the source memory location with postincrementation by the contents of the related general-purpose register.	**`wfifo = [ar0++gr0];`** `rep 8 wfifo = [ar0++gr0], ftw;`	1	5.1
Load the contents of the source memory location into the `ram` buffer. According to repetitions counter the appropriate number of long words are loaded from the source memory location with preincrementation by the contents of the related general-purpose register. (*)	**`ram = [ar0+=gr0];`** `rep 12 ram = [ar0+=gr0] with afifo - 1;`	1	5.1

Store in Memory Commands

DESCRIPTION	SYNTAX	SIZE	TYPE
Store the contents of the `afifo` buffer into the destination memory location given by the address register. All data words are stored into the same address, so only the last word will actually be stored in that location.	**`[ar0] = afifo;`** `rep 4 [ar0] = afifo with not afifo;`	1	5.1

Store the contents of the `afifo` buffer into the destination memory location given by the general-purpose register. All data words are stored into the same address, so only the last word will actually be stored in that location.	**`[gr2] = afifo;`** `rep 2 [gr2],ram = afifo with afifo + 0;`	1	5.1
Store the contents of the `afifo` buffer into the destination memory location. All data words are stored into the same address, so only the last word will actually be stored in that location. The address register is initialized with the contents of the related general-purpose register that points to the destination memory location.	**`[ar4=gr4] = afifo;`** `rep 1 [ar4=gr4] = afifo with ram + 1;`	1	5.1
Store the contents of the `afifo` buffer into the destination memory location with the address postincrementation by 2.	**`[ar0++] = afifo;`** `rep 4 [ar0++] = afifo with vsum ,ram,0;`	1	5.1
Store the contents of the `afifo` buffer into the destination memory location with the address predecrementation by 2.	**`[--ar4] = afifo;`** `rep 16 [--ar4] = afifo with ram - 1;`	1	5.1
Store the contents of the `afifo` buffer into the destination memory location with the address postincrementation by the contents of the general-purpose register.	**`[ar0++gr0] = afifo;`** `rep 8 [ar0++gr0] = afifo with not ram;`	1	5.1
Store the contents of the `afifo` buffer into the destination memory location with the address preincrementation by the contents of the general-purpose register.	**`[ar0+=gr0] = afifo;`** `rep 12 [ar0+=gr0] = afifo with not ram;`	1	5.1
Store the contents of the `afifo` buffer into the destination memory location and simultaneously into the `ram` buffer. (*)	**`[ar0++],ram = afifo;`** `rep 5 [ar0++],ram = afifo with 0-1;`	1	5.1

Note *In the right part of the instruction marked with (*) the `ram` buffer cannot be used because it has only one input/output port.*

5.2.2 Vector No Operation Commands

The instruction set of NM6403 provides for eventuality of an absence of the address command in the left part of the vector instruction. Moreover a nul vector command exists and may be used in a program. The main feature of the vector no operation command is that it is translated to the zero machine code.

DESCRIPTION	SYNTAX	SIZE	TYPE
No operation	**vnul;**	1	5.3
The absence of the address command in the left part of the vector instruction.	**rep 32 with not ram;**	1	5.3

5.2.3 Vector Logical Operations

Logical operations over operands **X** and **Y** are located only in the right part of a vector instruction. The operands **X** and **Y** are used to parameterize the operations in the right part of the vector instruction. The internal buffers of the Vector Unit such as `ram`, `data`, `afifo` and 'null' device are actually used in the vector instructions instead of the **X** and **Y**.

The vector logical operations are executed in the Vector ALU. According to the nature of logical operations they do not require the Vector ALU to be configured.

In addition to instruction syntax (bold) an example of an actual vector instruction with the non-empty left part is given.

If a vector instruction does not contain memory access commands, the left part of the instruction can be omitted (see paragraph 4.2 on page 4-6).

DESCRIPTION	SYNTAX	TYPE
Calculate the bitwise logical OR of the source operands and store the results into the `afifo`. (*)	**with X or Y;** `rep 4 data = [ar0+=gr0] with data or ram;`	7.1
Calculate the bitwise logical AND of the source operands and store the results into the `afifo`.	**with X and Y;** `rep 32 wfifo = [ar2++] with ram and afifo;`	7.1
Calculate the bitwise exclusive OR of the source operands and store the results into the `afifo`.	**with X xor Y;** `rep 10 [ar0++] = afifo with afifo xor ram;`	7.1
Calculate the bitwise logical complement of the source operand and store the results into the `afifo`.	**with not X;** `rep 16 data = [ar4++],ftw with not data;`	7.1
Calculate the bitwise logical OR of the second operand and the bitwise logical complement of the first operand, and store the results into the `afifo`.	**with not X or Y;** `rep 8 [ar2++] = afifo with not afifo or ram;`	7.1
Calculate the bitwise logical OR of the first operand and the bitwise logical complement of the second operand, and store the results into the `afifo`.	**with X or not Y;** `rep 1 [gr4] = afifo with ram or not afifo;`	7.1
Calculate the bitwise logical OR of the bitwise logical complement of the first operand and the bitwise logical complement of the second operand, and store the results into the `afifo`.	**with not X or not Y;** `rep 3 with not afifo or not ram;`	7.1

Calculate the bitwise logical AND of the second operand and the bitwise logical complement of the first operand, and store the results into the `afifo`.	**`with not X and Y;`** `rep 2 with not afifo and ram;`	7.1
Calculate the bitwise logical AND of the first operand and the bitwise logical complement of the second operand, and store the results into the `afifo`.	**`with X and not Y;`** `rep 9 [ar2++] = afifo with afifo and not ram;`	7.1
Calculate the bitwise logical AND of the bitwise logical complement of the first operand and the bitwise logical complement of the second operand, and store the results into the `afifo`.	**`with not X and not Y;`** `rep 24 ftw with ram and not afifo;`	7.1
Calculate the bitwise exclusive OR of the second operand and the bitwise logical complement of the first operand, and store the results into the `afifo`.	**`with not X xor Y;`** `rep 21 data = [ar0++] with not data xor ram;`	7.1
Fill the `afifo` with long words equal to `0000000000000000hl`.	**`with vfalse;`** `rep 16 [ar3++] = afifo with vfalse;`	7.1
Fill the `afifo` with vectors equal to `0FFFFFFFFFFFFFFFhl`.	**`with vtrue;`** `rep 32 wfifo = [ar5++gr5] with vtrue;`	7.1

Note

If the null device is used in the instruction marked with (*) it can be omitted, for example

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

In this case the data loaded from memory pass to the `afifo` without any changes.

5.2.4 Vector Arithmetic Operations

Arithmetic operations over operands **X** and **Y** are located only in the right part of a vector instruction. The operands **X** and **Y** are used to parameterize the operations in the right part of the vector instruction. The internal buffers of the Vector Unit such as `ram`, `data` and `afifo` are actually used in the vector instructions instead of the **X** and **Y**.

The vector arithmetic operations are executed in the Vector ALU. Before the Vector ALU is able to execute arithmetic operations it must be configured by the `nb1` register (see paragraph 3.3.2 on page 3-40).

In addition to instruction syntax (bold) an example of an actual vector instruction with the non-empty left part is given.

If a vector instruction does not contain memory access commands, the left part of the instruction can be omitted (see paragraph 4.2 on page 4-6).

DESCRIPTION	SYNTAX	TYPE
Add the operand **X** to the contents of the operand **Y** and store the results in the `afifo`.	**`with X + Y;`** `rep 4 data = [ar0+=gr0] with data + ram;`	7.2
Subtract the contents of the operand **Y** from the operand **X** to and store the results in the `afifo`.	**`with X - Y;`** `rep 32 wfifo = [ar2++] with ram - afifo;`	7.2
Load the difference between 0 and the operand Y to the `afifo`.	**`with 0 - Y;`** `rep 16 wfifo = [ar2++] with 0 - ram;`	7.2
Add 1 to each element of the packed words contained in the operand **X** and store the results in the `afifo`.	**`with X + 1;`** `rep 8 ftw with afifo + 1;`	7.2
Fill the `afifo` with the vectors whose every element is equal to 1.	**`with 0 + 1;`** `rep 4 [ar4+=gr4] = afifo with 0 + 1;`	7.2
Subtract 1 from each element of the packed words contained in the operand **X** and store the results in the `afifo`.	**`with X - 1;`** `rep 8 ram = [--ar2] with data - 1;`	7.2
Fill the `afifo` with the vectors whose every element is equal to -1.	**`with 0 - 1;`** `rep 8 with 0 - 1;`	7.2

5.2.5 Mask Application Operations

Mask application operations over operands **X** and **Y** are located only in the right part of a vector instruction. The operands **X** and **Y** are used to parameterize the operations in the right part of the vector instruction. The internal buffers of the Vector Unit such as `ram`, `data` and `afifo` are actually used in the vector instructions instead of the **X** and **Y**.

The mask application operations are executed in the Mask Application Unit. Before the Mask Application Unit is able to execute mask application operations it must be configured by the `nb1` and `sb` registers. (see paragraph 3.3.2 on page 3-40 and paragraph 3.3.3 on page 3-44 respectivly).

In addition to instruction syntax (bold) an example of an actual vector instruction with the non-empty left part is given.

If a vector instruction does not contain memory access commands, the left part of the instruction can be omitted (see paragraph 4.2 on page 4-6).

DESCRIPTION	SYNTAX	TYPE
Apply the mask **M** to the operands **X** and **Y**, then perform the bitwise logical OR to the results of mask application (**X** AND **M**) OR (**Y** AND ~**M**) and store the results into the `afifo`.	**`with mask M, X, Y;`** `rep 4 data = [ar0+=gr0] with mask afifo, data, ram;`	7.3

Apply the mask **M** to the operands **X** and **Y**, then shift the result of mask application to the operand **X** right one bit, perform the bitwise logical OR to the resulting data and store the results into the `afifo`.	**`with mask M, shift X, Y;`** `rep 8 data = [--ar0] with mask afifo, shift data, ram;`	7.3

Description of the mask application procedure can be found in the paragraph 1.5.4 on page 1-17. The cyclic shift of the operand right one bit is described in the paragraph 1.5.6 on page 1-20.

5.2.6 Weighted Accumulation

Weighted accumulation operations over operands **X** and **Y** are located only in the right part of a vector instruction. The operands **X** and **Y** are used to parameterize the operations in the right part of the vector instruction. The internal buffers of the Vector Unit such as `ram`, `data` and `afifo` are actually used in the vector instructions instead of the **X** and **Y**. The bias vector `vr` can also be used as the operand **Y**.

The weighted accumulation operations are executed in the Active Matrix. Before the Active Matrix is able to execute weighted accumulation operations it must be configured by the registers `nb1` and `sb` (see paragraph 3.3.2 on page 3-40 and paragraph 3.3.3 on page 3-44 respectivly).

In addition to instruction syntax (bold) an example of an actual vector instruction with the non-empty left part is given.

If a vector instruction does not contain memory access commands, the left part of the instruction can be omitted (see paragraph 4.2 on page 4-6).

In case a mask is not applied to the **X** and **Y**, the first operand of weighted accumulation is omitted, but the comma must stay on place to distinguish the positions of the **X** and **Y** operands.

DESCRIPTION	SYNTAX	TYPE
Perform weighted accumulation of the operand **X** and store the results into the `afifo`.	**`with vsum , X, 0;`** `rep 12 data = [ar2++gr2] with vsum , data, 0;`	7.4
Perform weighted accumulation of the operand **X**, then add the content of the operand **Y** to the result of weighted accumulation and store the results into the `afifo`.	**`with vsum , X, Y;`** `rep 32 ram = [--ar2] with vsum , data, vr;`	7.4
Apply the mask **M** to the operands, then perform weighted accumulation of the (**M** AND **X**) operand, add the operand (**M** AND ~**Y**) to the result of weighted accumulation and store the	**`with vsum M, X, Y;`** `rep 8 data = [ar2++] with vsum ram, data, afifo;`	7.4

results into the `afifo`.		
Perform weighted accumulation of the operand **X** shifted right one bit, then add the content of the operand **Y** to the result of weighted accumulation and store the results into the `afifo`.	**`with vsum , shift X, Y;`** `rep 1 [ar2=gr2] with vsum , shift ram, ram;`	7.4

5.2.7 Activation Operations

Activation operations are executed over operands in the right part of a vector instruction. To activate the operands the reserved word '`activate`' is used in front of the operand. Activation operations are applied to the operands independently.

The type of the activation function to be applied to the operands depends on the type of an entire operation. The saturation function is used together with arithmetic operations, while the threshold function is used together with logical ones.

The activation operations are executed in two Activation Units. Before the Activation Units are able to apply the activation functions to the operands each of them must be configured by the registers `f1cr` (for the operand **X**) and `f2cr` (for the operand **Y**). (see paragraph 3.3.1 on page 3-34).

In addition to instruction syntax (bold) an example of an actual vector instruction with the non-empty left part is given.

Logical Activation

Logical activation of data can be performed with any logical operation given in 5.2.3 on page 5-32. In the table below examples of activation of operands standing in different positions in the right part of a vector instruction are listed.

DESCRIPTION	SYNTAX	TYPE
Activate the operand **X** before the bitwise logical OR is calculated.	**`with activate X or Y;`** `rep 32 data = [ar0++] with activate data or ram;`	7.1
Activate the operand **X**, before the bitwise logical AND of the **Y** and the bitwise logical complement of **X** is calculated.	**`with not activate X and Y;`** `rep 16 data = [--ar0] with not activate data and ram;`	7.1
Activate the operand **Y** before the bitwise logical OR is calculated.	**`with X or activate Y;`** `rep 8 data = [ar0++gr0] with data or activate ram;`	7.1
Activate the operand **Y**, before the bitwise logical AND of the **X** and the bitwise logical complement of **Y** is calculated.	**`with X and not activate Y;`** `rep 12 [--ar0] = afifo with afifo and not activate ram;`	7.1

Activate both operands before the bitwise exclusive OR is executed.	**with activate X xor activate Y;** `rep 8 ftw with activate afifo xor activate ram;`	7.1
Activate both operands before the bitwise logical AND of the bitwise logical complement of both operands is executed.	**with not activate X and not activate Y;** `rep 2 with not activate afifo and not activate ram;`	7.1
Activate the result of (**M** AND **X**) operation before the bitwise logical OR of the masked operands is executed.	**with mask M, activate X, Y;** `rep 8 data = [ar0++] with mask afifo, activate data, ram;`	7.1
Activate the result of (~**M** AND **Y**) operation before the bitwise logical OR of the masked operands is executed.	**with mask M, X, activate Y;** `rep 1 data = [--ar0] with mask afifo, data, activate ram;`	7.1
Activate the results of **M** application to both operands before the bitwise logical OR of the masked operands is executed.	**with mask M, activate X, activate Y;** `rep 4 with mask afifo, activate ram, activate ram;`	7.1
Activate the result of (**M** AND **X**) operation before the cyclic shift right one bit and the bitwise logical OR of the masked operands are executed.	**with mask M, shift activate X, Y;** `rep 12 with mask afifo, shift activate data, ram;`	7.1

Arithmetic Activation

Arithmetic activation of data can be performed with any vector arithmatic operation given in paragraph 5.2.4 on page 5-33. In the table below examples of activation of operands standing in different positions in the right part of a vector instruction are listed.

DESCRIPTION	SYNTAX	TYPE
Activate the operand **X** and then add the activated **X** to the operand **Y**.	**with activate X + Y;** `rep 32 data = [ar0++] with activate data + ram;`	7.2
Activate the operand **Y** and then subtract the activated **Y** from the operand **X**.	**with X - activate Y;** `rep 8 data = [ar0++gr0] with data - activate ram;`	7.2
Activate the both operands and then add the activated **X** to the activated **Y**.	**with activate X + activate Y;** `rep 16 data = [--ar0] with activate data + activate ram;`	7.2
Activate the operand **X** and then perform weighted accumulation of the activated **X** and add the result	**with vsum , activate X, Y;** `rep 2 data = [ar4++] with vsum , activate data, ram;`	7.2

to the contents of the operand **Y**.		
Activate the operand **Y** and then add the result to the result of weighted accumulation of **X**.	**with vsum , X, activate Y;** `rep 2 [ar6++] = afifo with vsum , ram, activate afifo;`	7.2
Perform the weighted accumulation over the masked activated operands.	**with vsum M, activate X, activate Y;** `rep 5 with vsum ram, activate ram, activate afifo;`	7.2
Perform the weighted accumulation over the activated shifted operand **X**.	**with vsum , shift activate X, 0;** `rep 30 ftw with vsum , shift activate ram, 0;`	7.2

5.2.8 Weights Loading

This section contains all vector commands necessary to load weight coefficients to the Shadow and the Active Matrixes of the Vector Unit.

Weights loading operations are located in the left part of a vector instruction.

In addition to instruction syntax (bold) an example of an actual vector instruction with the non-empty right part is given.

DESCRIPTION	SYNTAX	SIZE	TYPE
Load weights from memory to the `wfifo` buffer.	**rep 24 wfifo = [ar0++];** `rep 24 wfifo = [ar0++] with ram + 1;`	1	5.2
Load weights from memory to the `wfifo` buffer and transfer the requested part of the weights to the Shadow Matrix.	**rep 32 wfifo = [ar1++], ftw;** `rep 32 wfifo = [ar1++],ftw with 0 - 1;`	1	5.2
Load weights from memory to the `wfifo` buffer and transfer the requested part of the weights both to the Shadow Matrix and then to the Active Matrix.	**rep 32 wfifo = [ar1++], ftw, wtw;** `rep 32 wfifo = [ar1++],ftw with 0 - 1;`	1	5.2
Transfer the weights from `wfifo` to the Shadow Matrix. (**)	**ftw;** `rep 16 ftw with not ram and afifo;`	1	5.3
Copy the contents of the Shadow Matrix and the configuration registers to the Active Matrix and its configuration registers. (*)	**wtw;** `rep 12 wtw with activate afifo;`	1	5.3

Note

In the instruction marked with (*) used the operation wtw the copying the contents of Shadow Matrix to the Active Matrix. Because of the hardware error of the NM6403 processor exists, at the using of the operation `wtw` it is necessary to use the following instruction sequence, in that way parallel execution of others instructions being blocked:

```
.wait
      nb1 = value;
```

```
      wtw;
.branch;
```

Note

The operation `ftw` *marked with (**) can be used both separated and in the left part of any vector command, for example*

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

5.2.9 Store the Vector Unit Control Registers

To store the vector unit control registers `f2cr`, `f1cr`, `nb2`, `sb`, `vr` in memory a special command is used. The enumerated registers are not directly read accessible, but their contents can be obtained indirectly.

DESCRIPTION	SYNTAX	SIZE	TYPE
Vector registers saving in `afifo` buffer.	**`rep 5 with store vregs;`**	1	7.5

All the vector unit control registers are 64-bit long. The repeat counter in a vector instruction must be equal to 5 according to the number of registers to store. The instruction above stores the registers into the `afifo` buffer. So the contents of the registers can be stored in any desired memory location. For more details see 5.2.1 on page 5-29.

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

Printed in Russia Data of issue: February 20, 2006