NM6403 Software Development Kit

Application Report

Fast Hadamard Transform

Revision 2.0

Written by Vitali Kashkarov

JOINT-STOCK COMPANY
RESEARCH CENTRE

Contents

INTRODUCTION 3

BRIEF DESCRIPTION OF HADAMARD TRANSFORM 4

ALGORITHM IMPLEMENTATION IN C LANGUAGE 6

IMPLEMENTATION OF THE ALGORITHM ON NM6403 7

- CONCEPT OF CALCULATIONS 7
- IMPLEMENTATION OF THE FIRST THREE STEPS OF FHT 7
  - Loading of the Active Matrix 9
  - Making the calculations 11
  - Methods of Addressing 13
- IMPLEMENTATION OF THE NEXT FIVE STEPS OF FHT 13
  - Making the calculations 16

CONCLUSION 18

- BIBLIOGRAPHY 19

This document describes an approach to Fast Hadamard Transform (FHT) programming on the NeuroMatrix® NM6403 processor. The following indexes are used as parameters defining the transform:

- input data bit length: 8 bits (signed data),
- number of steps: 8,
- output data bit length: 16 bits (signed data),
- source vector size: 256 bytes,
- result vector size: 256 short words (16-bit).

The program was implemented on the NM6403 processor. The main features of this processor are the matrix operational node with the 64x64 bit capacity and the possibility of software assigning the processed data capacity. More detailed information about NM6403 see in [1].

The document contains detailed comments for methods of processor programming in assembly language. Special attention is drawn to work with the processor matrix operational node.

Hadamard transform is made over integer numbers. Two-valued Walsh functions taking values `1, -1` are used as basis functions. That's why Hadamard transform does not have any other operations besides summation and subtraction.

Hadamard transform can be expressed in the form of multiplication of matrix by vector. The transform matrix - Hadamard matrix - consists of `+1` and `-1`. The matrix rows and columns are orthogonal. Hadamard matrix can be defined recursively:

$$H_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{ and } H_{2N} = \frac{1}{\sqrt{2}}\begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix}$$

For example, the matrix of the eighth order looks like:

Fig. 1 Hadamard Matrix of the 8-th Order.

$$H_8 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

If a vector $\boldsymbol{x}$ with the length N ($N=2^n$) is the input one then a vector $\boldsymbol{y}$ obtained in the result of Walsh-Hadamard transform is equal to $\boldsymbol{y} = H_N\boldsymbol{x}$.

The transform described above is called discrete (DHT). However fast Hadamard transform (FHT) is more often used in practice. The following table can describe its essence:

Tab. 1 Steps of Fast Hadamard Transform.

Input data	Step one (1)	Step two (2)	Step three (3)
a_1	$b_1 = a_1 + a_2$	$c_1 = b_1 + b_3$	$d_1 = c_1 + c_5$
a_2	$b_2 = a_1 - a_2$	$c_2 = b_2 + b_4$	$d_1 = c_2 + c_6$
a_3	$b_3 = a_3 + a_4$	$c_3 = b_1 - b_3$	$d_1 = c_3 + c_7$
a_4	$b_4 = a_3 - a_4$	$c_4 = b_2 - b_4$	$d_1 = c_4 + c_8$
a_5	$b_5 = a_5 + a_6$	$c_5 = b_5 + b_7$	$d_1 = c_1 - c_5$
a_6	$b_6 = a_5 - a_6$	$c_6 = b_6 + b_8$	$d_1 = c_2 - c_6$
a_7	$b_7 = a_7 + a_8$	$c_5 = b_5 - b_7$	$d_1 = c_3 - c_7$
a_8	$b_8 = a_7 - a_8$	$c_6 = b_6 - b_8$	$d_1 = c_4 - c_8$

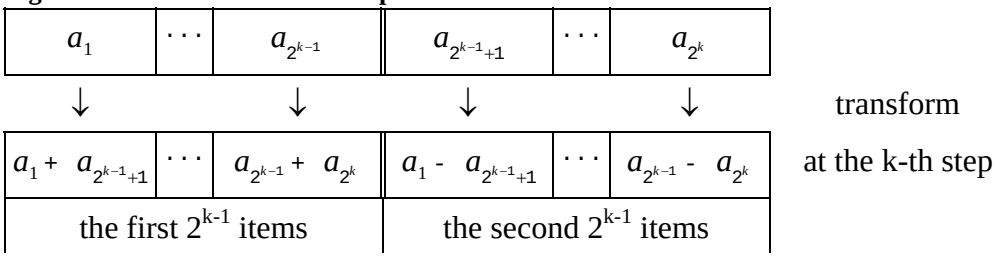
Fig. 1 shows that the elements of Step 3 of the transform can be expressed through the input array elements in the following way:

$$d_1 = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8$$
$$d_2 = a_1 - a_2 + a_3 - a_4 + a_5 - a_6 + a_7 - a_8$$
$$d_3 = a_1 + a_2 - a_3 - a_4 + a_5 + a_6 - a_7 - a_8$$
$$d_4 = a_1 - a_2 - a_3 + a_4 + a_5 - a_6 - a_7 + a_8$$
$$d_5 = a_1 + a_2 + a_3 + a_4 - a_5 - a_6 - a_7 - a_8$$
$$d_6 = a_1 - a_2 + a_3 - a_4 - a_5 + a_6 - a_7 + a_8$$
$$d_7 = a_1 + a_2 - a_3 - a_4 - a_5 - a_6 + a_7 + a_8$$
$$d_8 = a_1 - a_2 - a_3 + a_4 - a_5 + a_6 + a_7 - a_8$$

It is clear from Fig. 1 that the coefficients `1` and `-1` in front of the input vector items exactly correspond to the Hadamard matrix of the 8-th order.

In general the transform at the *k*-th step defines the interaction of 2^k input vector elements. Here each element of the first 2^{k-1} ones is filled with sum of it's prior value and the value of the element which is at the distance of 2^{k-1} elements from it and the second 2^{k-1} elements are filled with the differences (see Fig. 2):

Fig. 2 Transforms at the k-th Step.



In the result of the transform every item of the output vector is expressed through all items of the input one. However if we go one transform step back we will see that there are two data blocks and complete interrelationship between the elements exists in each block. At the same time in the formula of transform of the items of one block none of the items of the other block participates, i.e. the blocks are absolutely independent. For example, every two items are independent from their neighbors at the first step because only their values:

```
b1 = a1 + a2
b2 = a1 - a2
```

are included into the formula of their calculation. That's why, for example, calculation of the values of pairs of elements (b_1, b_2), (b_3, b_4), ...(b_{2^k-1}, b_{2^k}) can be made independently. Thus independently calculated blocks can be extracted at each step up to the last one.

Hadamard transform in C language can be written in the following way:

```
void Adamar( int    AStepsNum,  // number of steps
             int*   AData)      // data array
{
  int    DataSz, i, j, jj, Block, Pair, Ind0, Ind1;
  int    Item0, Item1;

  /* preparation */
  DataSz = 1 << StepsNum;     // calculation of the array size

  /* Hadamard transform */
  Block = 1;
  for( i = 0; i < AStepsNum; i++ ) {
    Pair = Block;        // distance between (a1 , a2^k+1 )pairs
    Block += Block;      // independent block size for the current step
    for( j = 0; j < DataSz; j += Block) {
      for( jj = 0; jj < Pair; jj++ ) {
        Ind0 = j + jj;        // first item index calculation
        Ind1 = ind0 + Pair;   // second item index
        Item0 = AData[Ind0];  // get old values
        Item1 = AData[Ind1];
        AData[ind0] = Item0 + Item1; //store new values
        AData[ind1] = Item0 - Item1;
      }
    }
  }
}
```

This example is for 32-bit data processing. The input parameter `AStepNum` assigns the number of transform steps (`AStepNum = 8`). The parameter `AData` gets a pointer at the input data array of 256 elements.

The example of Hadamard transform implementation in C is designed for sequential execution of instructions that is typical for Pentium like architectures. In that case equal time is needed to calculate each step. There is no any parallelism in the example above. That's why this text compilation into the NM6403 processor codes will not give satisfactory results.

NeuroMatrix® NM6403 processor allows software changing of the processing data bit length. It means that having assigned the corresponding configuration of its Active Matrix participating in the calculations, it is possible to execute transforms over 8-bit data during several steps. After the theoretically calculated 8-bit capacity of the results is reached its is possible to transform data into 16-bit form and continue the calculations, etc. Maximum possible accumulator size that could be implemented on the NM6403 processor is 64 bit. So if the initial data capacity doesn't exceed 8 bit for signed data, it is possible to make 56 steps of Hadamard transform without exceeding data range of accumulator. Here 2^{56}(~10^{17}) of 64-bit memory words will be needed to place the result vector.

The matrix operational node of NeuroMatrix® NM6403 allows making calculations of up to 5 transform steps in parallel. This approach is described below.

Concept of calculations

The FHT task is divided into two routines. The first one calculates first three steps of FHT and converts data from 8 bits representation to 16 bits. The second one makes calculations for the next 5 steps of FHT in parallel.

Both routines have C-callable interface, so they are available from C file. The C `main()` program fills an input buffer with initial values. After that it starts clocks counting and calls the hand optimized assembly routines to make calculations over the buffer. After calculations are made it counts number cycles spent on execution of the routines and calculates check sum to ensure that the results are correct. If the check sum is equal to the expected one the `main` returns number of cycles otherwise error code (-1).

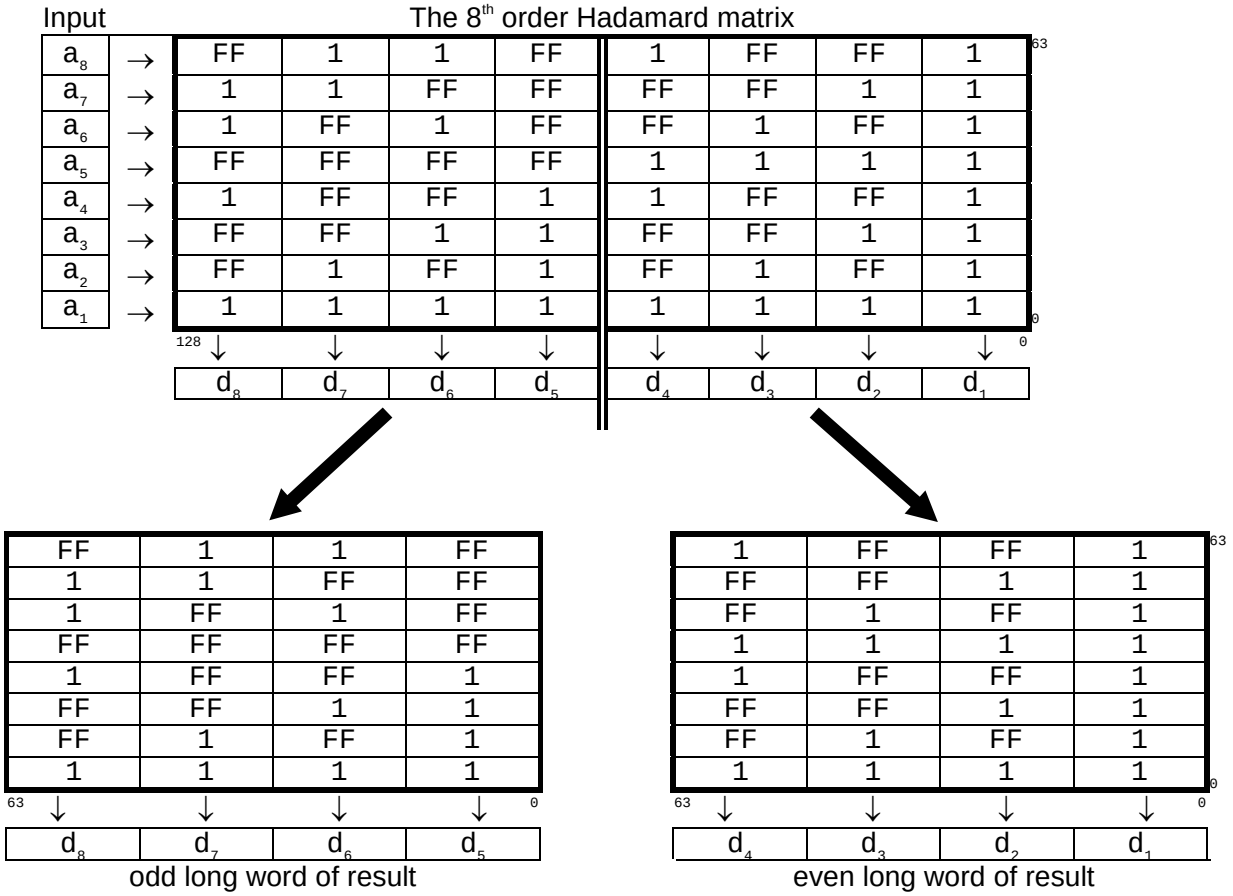
Implementation of the first three steps of FHT

Since the input data bit length does not exceed 8 bits and the results of execution are stored into 16 bits, no any preliminary input data transform is required. It means that the input data is processed "as is". We assume that they are packed into 64-bit words so the NeuroMatrix® NM6403 operates 8 elements in parallel. The results of calculation are accumulated into 16-bit elements packed into 64-bit words.

Let's consider the Hadamard matrix of 8^{th} order (Fig. 1). It has eight columns and eight rows. Each 8-bit element of a long input word multiplies to each cell of a related row in the Hadamard matrix. The results of multiplication are accumulated over each column. To avoid data overflow

it's necessary to accumulate the results into 16-bit cells. That is why the total accumulator bit length should be at least 128 bits.

Fig. 3 Division of the 8th order Hadamard matrix into sub-matrixes.



From Fig. 3 you can see that it is possible to divide the Hadamard matrix into two sub-matrixes. The first sub-matrix is used to calculate even long words of the result, the second one to compute odd long words.

Each sub-matrix contains 32 cells that is every cycle the NeuroMatrix® NM6403 makes 32 MAC (multiplications and accumulations).

We make calculations in the following order:

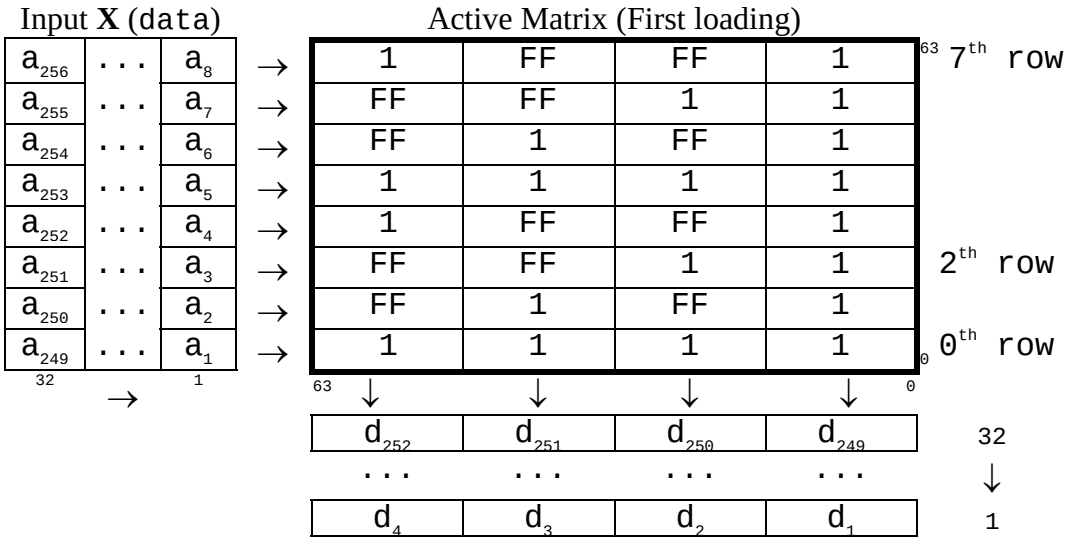
- Load first sub-matrix of Hadamard weights into the Active Matrix of the processor;
- Calculate the even long words of result. The results are accumulated in `afifo`. Its capacity is 32 long words;
- Store them into even positions of the result array;
- At the same time reload the new weights into the Active Matrix;
- Make second round of calculations and store the results into odd positions of the result array.

Below in this document you can see description of implementation of each step of calculations mentioned above.

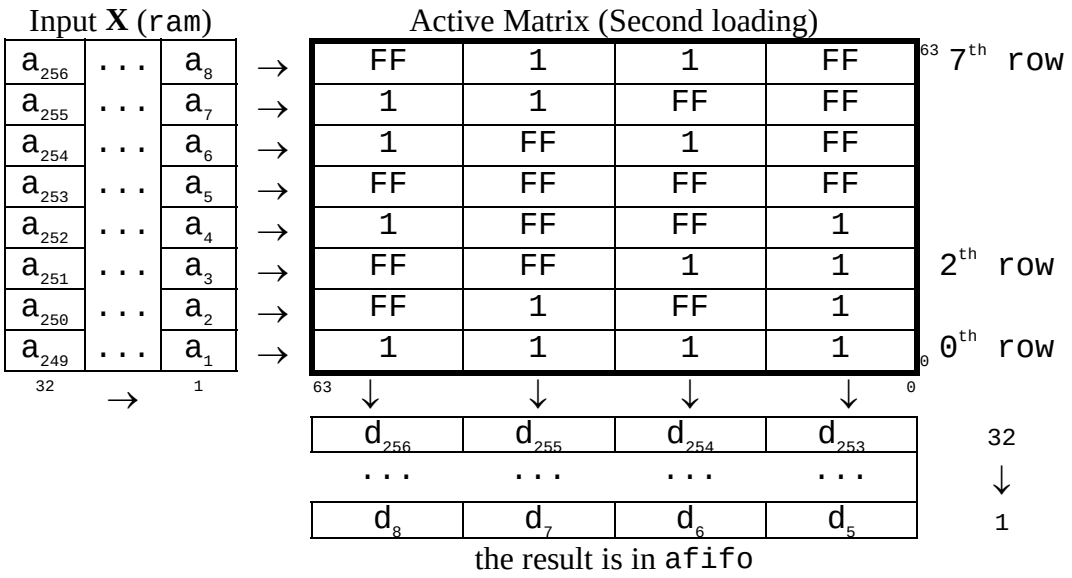
Loading of the Active Matrix

The data processing scheme that is used in the NeuroMatrix® MN6403's Matrix Operational Node (MON) is shown in Fig. 4:

Fig. 4 Data processing in the Active Matrix.



a) First stage of calculations (even long words of result



b) Second stage of calculations (odd long words of result

Suppose the weights were preliminary loaded into the Active Matrix from the external memory. The way of weights loading is described below.

The input data stream goes in through the input **X**. The input and output buffers are of FIFO type. They can contain up to 32 of 64-bit words. Each of the FIFO's: `ram`, `data`, `afifo` can be used as the source for **X.** The results are always accumulated in `afifo`.

The weights are stored into the memory in form of an array of 64-bit words. The number of words to be loaded into the Active Matrix is defined by the `sb2` register. This register configures the number of rows in the Active Matrix. Each row is assigned to one 64-bits word. The first long word of the array is loaded into the 0^{th} row of the matrix, second one into the 1^{th} row and so on.

From Fig. 4 it can be seen that rows of the Active Matrix are numerated upwards. This way of numeration is caused by numeration of bits in a word (bits in a word are ordered from the right to the left, the low-order bit is at the right position). Memory addresses increase is made in the same direction.

The following assembly text block defines the weights array that will be loaded into the Active Matrix to calculate first three steps of Hadamard transform.

```
data ".data"
    M_1_3: long[16] = ( 0000100010001000hl, // zero row
                        0FFFF0001FFFF0001hl,//low-order bit of the word
                        0FFFFFFF00010001hl, // the second row
                        00001FFFFFFF0001hl,
                        0000100010001000hl,
                        0FFFF0001FFFF0001hl,
                        0FFFFFFF00010001hl,
                        00001FFFFFFF0001hl, // the seventh row

                        0000100010001000hl, // zero row of second part
                        0FFFF0001FFFF0001hl, // of weights
                        0FFFFFFF00010001hl,
                        00001FFFFFFF0001hl,
                        0FFFFFFFFFFFFFFFhl,
                        00001FFFF0001FFFFhl,
                        000010001FFFFFFFhl,
                        0FFFF00010001FFFFhl); // the seventh row
end ".data";
```

As the NM6403 processor's Active Matrix is divided into 4 columns by `nb2` register, the low-order 16-bit word of the 0^{th} long word in the array goes into the 0^{th} column, the first 16-bit word into the first one, etc.

Note

It is important that though the Active Matrix rows are numerated upwards, the 0^{th} long word at the top of the array presented above will go to the 0^{th} row of the Active Matrix.

Weights are loaded from memory into a special internal memory block of the processor called `wfifo`. This block is also of FIFO type. It can be used only for loading weights into the Matrix Operational Node. `wfifo` has one additional feature if compared with other internal memory blocks. It can be filled in for several operations.

Loading of data from memory into the Active Matrix is described by the following instructions in assembly:

```
begin ".text"
<_Steps_1_3>
.branch; // switch on the parallel vector instructions execution.
    nb1 = 80008000h; // Configuration of the columns number.
    sb  = 03030303h; // Configuration of the rows number.
    ar0 = M_1_3;     // Address of the weights array.
    rep 16 wfifo = [ar0++], ftw, wtw; // Loading of weights.
    ...
end ".text";
```

At first the registers `nb1` and `sb` are filled with the constants that configure the number of columns and rows of the Shadow Matrix. The `nb1` and `sb1` are associated with the Shadow Matrix. In fact the Active Matrix configuration will come in force only after the `wtw` instruction is executed.

The `nb1` and `sb` registers are 64-bit. If they are initialized with a 32-bit instruction the processor copies this value into both 32-bit parts of a long word, i.e. the register `nb1` is initialized with a long constant `8080808080808080hl`. The same is true for the `sb` register. Usage of the `nb1` and `sb` (`sb1`, `sb2`) registers is described in more details in [3].

After the control registers `nb1` and `sb` are initialized the configuration of the Shadow Matrix and the future configuration of the Active Matrix are defined.

The address of the weights array is put into the address register. Then 16 long words of data are loaded from memory to `wfifo`.

The instruction `ftw` transfers data from `wfifo` to the Shadow Matrix. The number of words that will be read from `wfifo` is defined by `sb1`.(in our case the number of rows in the Shadow Matrix is 8, so 8 long words will be transferred).

`ftw` always takes 32 cycles, regardless of the number of words that have to be loaded into the Matrix Operational Node. For example, in the case of the 8^{th} order Hadamard matrix only eight 64-bit words should be loaded, but their conversion into the internal representation lasts 32 cycles. However conversion takes place in parallel with weights loading and starts one cycle after the first word is loaded into `wfifo`.

After the weights are transferred into the Shadow Matrix the instruction `wtw` is executed. It copies the contents of the Shadow Matrix into the Active Matrix for one cycle. At the same time the values of the registers `nb1` and `sb1` are copied to `nb2` and `sb2`. Loading of the Active Matrix is complete.

Making the calculations

The following assembly text block defines the most important part of the routine that calculates the first three steps of Hadamard transform:

```
// Section of source code.
begin ".text"
// The routine for the first 3 steps of FHT.
<_Steps_1_3>
```

```
.branch; // switch on the parallel vector instructions execution.
    ...
    // Load the columns configuration of the Shadow Matrix.
    gr0 = 80008000h;
    // Copy the columns configuration into nb1 | Init gr4 with zero.
    nb1 = gr0 with gr4 = false;
    // Load the rows configuration of the Shadow Matrix.
    sb  = 03030303h;
    // Load the address of weights array | gr4 = 1;
    ar0 = M_1_3 with gr4++;
    // Load 16 long words to wfifo, transfer 8 words to the Shadow Matrix
    // and copy the Shadow Matrix contents to the Active Matrix.
    rep 16 wfifo = [ar0++], ftw, wtw;
    // Load the address of source buffer | gr4 = 2;
    ar0 = [--ar5] with gr4 <<= 1;
    // Load the address of destination buffer | gr5 = 4; <- Increment value
    ar4 = [--ar5] with gr5 = gr4 << 1;
    // ar5 points to the destination buffer address + 2 (next long word).
    ar5 = ar4 + gr4 with gr4 = gr5;
    //  Load source data, make calculations and
    //  transfer 8 long words from wfifo to the Shadow Matrix.
    rep 32 ram = [ar0++],ftw with vsum , data, 0;
    //  Store results in memory.  // These two instructions are
    rep 32 [ar4++gr4] = afifo;    // executed in parallel.
    // This part of code is used to step over the silicon bug.
.wait; // switch off the parallel vector instructions execution.
    // Copy the same constant to nb1 to lock wtw execution
    // until ftw is finished.
    nb1 = gr0;
    // Copy the Shadow Matrix contents to the Active Matrix.
    wtw;
.branch; // switch on the parallel vector instructions execution.
    // Make a second part of calculations.
    rep 32 with vsum , ram, 0;   // These two instructions are
    // Store results in memory.  // executed in parallel.
    rep 32 [ar5++gr5] = afifo;   //
    // Return from the routine.
    return;
end ".text";
```

After the Active Matrix is loaded it is possible to start calculations themselves. The source and result buffers are allocated in different memory blocks at different data buses. In this case operations of the source data reading and of the result data writing can be executed in parallel.

Each vector instruction contains internal loop counter that defines elementary loops ranged from 1 to 32. The internal memory blocks depth defines the maximum number of internal loops. The blocks can contain up to 32 long words.

The size of processing buffers in our task is small enough to implement any external loop.

Some lines of assembly text above need additional explanation.

Methods of Addressing

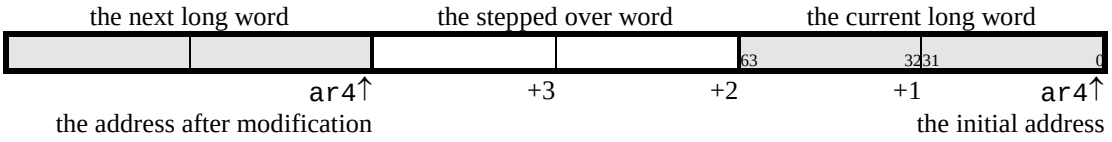
The instruction

```
rep 32 [ar4++gr4] = afifo;
```

stores 32 long words of result in memory and each cycle modifies the address register `ar4` with an increment value contained in `gr4`. The address modification type used in the instruction is post-modification. First a 64-bit word is stored into memory and after that address is modified.

The NeuroMatrix® NM6403 addresses to 32-bit words. There are two ways to address 64-bit words. If you need to address neighbor 64-bit word, you may use incremental addressing `[ar0++]`. In vector instructions it always means add 2 to access next long word. If you need another regular method of addressing you have to use increment by general-purpose register: `[ar4++gr4]`. General-purpose register has to contain even value to access only even addresses. For example, if you wish to access long word over long word, `gr4` should be equal to 4. It is shown in Fig. 5:

Fig. 5 Modification of the address register in vector instruction.

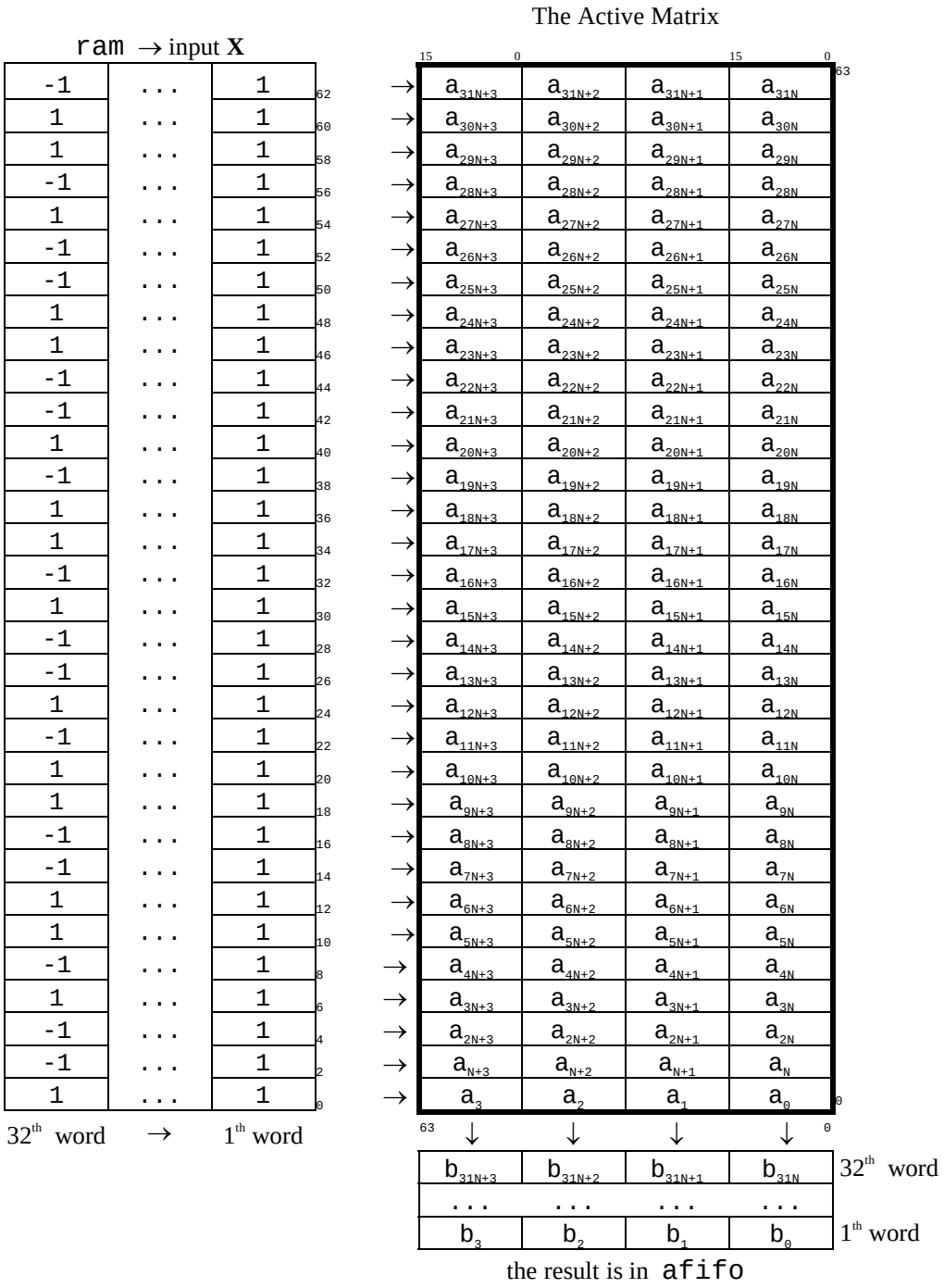


Implementation of the next five steps of FHT

As mentioned above Hadamard transform includes only operations of summation and subtraction. Hence there is a possibility to use the processor Active Matrix having divided it into 32 rows. The bit length of data coming to input **X** is 2 bit. It is enough to store numbers `1,0` and `-1`.

Division of the Active Matrix into 32 rows allows executing 32 operations of summation/subtraction in each column. When 16-bit data are processed the Active Matrix is divided into 4 columns. That's why it is possible to execute 128 operations of summation/subtraction by one cycle (see Fig. 6).

Fig. 6 The Active Matrix configuration to perform 128 arithmetic operations per cycle.



The main question is how to provide interaction of all 32 rows of the matrix. Let's look at the following table:

Tab. 2 Connections between elements of five sequential steps of FHT.

Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
1	**1 + 2**	1 + 3	1 + 5	1 + 9	1 + 17
2	1 - 2	2 + 4	2 + 6	2 + 10	2 + 18
3	**3 + 4**	**1 - 3**	3 + 7	3 + 11	3 + 19

4	3 - 4	2 - 4	4 + 8	4 + 12	4 + 20
5	**5 + 6**	5 + 7	1 - 5	5 + 13	5 + 21
6	5 - 6	6 + 8	2 - 6	6 + 14	6 + 22
7	**7 + 8**	**5 - 7**	**3 - 7**	**7 + 15**	**7 + 23**
8	7 - 8	6 - 8	4 - 8	8 + 16	8 + 24
9	**9 + 10**	9 + 11	9 + 13	1 - 9	9 + 25
10	9 - 10	10 + 12	10 + 16	2 - 10	10 + 26
11	**11 + 12**	**9 - 11**	11 + 15	3 - 11	11 + 27
12	11 - 12	10 - 12	12 + 16	4 - 12	12 + 28
13	**13 + 14**	13 + 15	9 - 13	5 - 13	13 + 29
14	13 - 14	14 + 16	10 - 16	6 - 14	14 + 30
15	**15 + 16**	**13 - 15**	**11 - 15**	7 - 15	15 + 31
16	15 - 16	14 - 16	12 - 16	8 - 16	16 + 32
17	**17 + 18**	17 + 19	17 + 21	17 + 25	1 - 17
18	17 - 18	18 + 20	18 + 22	18 + 26	2 - 18
19	**19 + 20**	**17 - 19**	19 + 23	19 + 27	3 - 19
20	19 - 20	18 - 20	20 + 24	20 + 28	4 - 20
21	**21 + 22**	21 + 23	17 - 21	21 + 29	5 - 21
22	21 - 22	22 + 24	18 - 22	22 + 30	6 - 22
23	**23 + 24**	**21 - 23**	**19 - 23**	**23 + 31**	7 - 23
24	23 - 24	22 - 24	20 - 24	24 + 32	8 - 24
25	**25 + 26**	25 + 27	25 + 29	17 - 25	9 - 25
26	25 - 26	26 + 28	26 + 30	18 - 26	10 - 26
27	**27 + 28**	**25 - 27**	27 + 31	19 - 27	11 - 27
28	27 - 28	26 - 28	28 + 32	20 - 28	12 - 28
29	**29 + 30**	29 + 31	25 - 29	21 - 29	13 - 29
30	29 - 30	30 + 32	26 - 30	22 - 30	14 - 30
31	**31 + 32**	**29 - 31**	**27 - 31**	23 - 31	15 - 31
32	31 - 32	30 - 32	28 - 32	24 - 32	16 - 32

Note

In this table numbers in every column refer to the numbers of cells of the previous step (column).

From Tab. 2 you can see that it is possible to express values of elements after 8th step of FHT through the values of elements after 3rd step of FHT.

For example let's see how the value of the 7th cell at the 8th step is expressed through the values of the cells at the 3rd step (see the marked cells).

The value of the 7th cell at the 8th step is recorded as the sum of the values of the 7th and the 23rd cells of the 7th step. Each of these cells is expressed in the same manner through the values of the cells of the 6th step, etc.

In the result the value of the 7th cell after 8 steps of FHT is expressed through values of the cells after 3 steps of FHT in the following way:

$$b_7 = a_1+a_2-a_3-a_4-a_5-a_6+a_7+a_8+a_9+a_{10}-a_{11}-a_{12}-a_{13}-a_{14}+a_{15}+a_{16}+ a_{17}+a_{18}-a_{19}-a_{20}-a_{21}-a_{22}+a_{23}+a_{24}+a_{25}+a_{26}-a_{27}-a_{28}-a_{29}-a_{30}+a_{31}+a_{32}$$

The same can be expressed by a 63-bit constant:
`05FF55FF55FF55FF5hl`

where 2 bits are assigned for each sign.

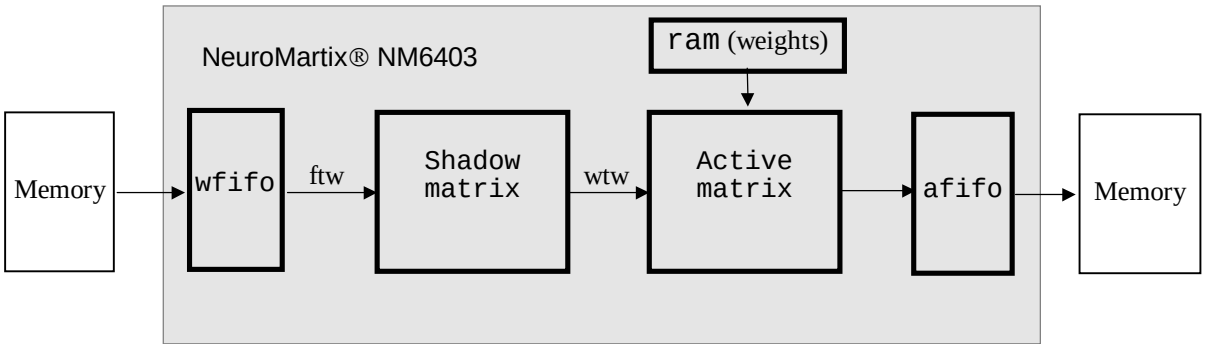
The scheme of data processing differs from the previous one because we load data into the Active Matrix instead of weights as we did above.

We make calculations of five steps of FHT in the following order:

- Load weights from external memory into `ram`;
- Load 32 even long words of input data into `wfifo` → ShM → ActM;
- Load 32 odd long words of input data into `wfifo` → ShM. In the same instruction make calculations of even words in parallel;
- Store even words of the result in memory.
- Make calculations of odd words and store the result in memory.

The data processing scheme used in this routine is presented in Fig. 7:

Fig. 7 Scheme of data processing in NM6403 for 5 parallel steps of FHT.



Making the calculations

The following assembly text block defines the most important part of the routine that calculates next five steps of Hadamard transform:

```
// Section of source code.
begin ".text"
<_Steps_4_8>
.branch; // switch on the parallel vector instructions execution.
    // Load the address of weights array | Init gr0 with zero.
    ar6 = M_4_8 with gr0 = false;
    // Load the columns configuration of the Shadow Matrix.
    gr2 = 80008000h;
    // Load the rows configuration of the Shadow Matrix.
    sb  = 0FFFFFFFFh;
    // Copy the columns configuration into nb1 | gr0 = 1;
    nb1 = gr2 with gr0++;
```

```
    // Load the address of source buffer | gr0 = 2;
    ar0 = [--ar5] with gr0 <<= 1;
    // ar1 points to the source buffer address+2 (next long word)| gr1 = 4
    ar1 = ar0 + gr0 with gr1 = gr0 << 1;
    // Load the address of destination buffer | gr4 = 2;
    ar4 = [--ar5] with gr4 = gr0;
    // ar5 points to the destination buffer address + 2 | gr5 = 4
    ar5 = ar4 + gr4 with gr5 = gr1;
    // gr0 = 4 | gr4 = 4 <- Increments to access odd/even long words.
    gr0 = gr1 with gr4 = gr1;

    // Load source data to wfifo -> ShM -> ActM
    rep 32 wfifo = [ar0++gr0], ftw, wtw;
    // Load weights to ram;  // These two instructions
    rep 32 ram = [ar0++];    // are executed in parallel.

    // Load next part of source data to wfifo -> ShM | Make calculations.
    rep 32 wfifo = [ar1++gr1], ftw with vsum , ram, 0;
    // Store results in memory. // These two instructions
    rep 32 [ar4++gr4] = afifo;  // are executed in parallel.

    // This part of code is used to step over the silicon bug.
.wait;    // switch off the parallel vector instructions execution.
    // Copy the same constant to nb1 to lock wtw execution
    // until ftw is finished.
    nb1 = gr2;
    // Copy the Shadow Matrix contents to the Active Matrix.
    wtw;
.branch;  // switch on the parallel vector instructions execution.

    // Make calculations.
    rep 32 with vsum , ram, 0;  // These two instructions
    // Store results in memory  // are executed in parallel.
    rep 32 [ar5++gr5] = afifo;

    // Return from the routine.
    return;
.wait;
end ".text";
```

Full source codes of the application and build utilities are available. You can find them in NEURO\EXAMPLES\FHT directory of installed NeuroMatrix® NM6403 SDK or please visit our web site http://www.module.ru.

The NeuroMatrix® NM6403 architecture is suitable for FHT. It takes 349 cycles to calculate 256-point FHT including C calling overhead.

To make calculations of 256-point FHT radix-2 it takes 2048 arithmetic operations. It means that the effective performance of the processor is:

(8 operations * 256 elements) / 349 cycles = ~5.8 arithmetic operations per cycle.

From the other hand the real calculations are differ from the theory. The first step is based on radix-8 calculations; the second one is on radix-32. The total number of arithmetic operations is more than was estimated above.

It can be evaluated in the following way: in the routine `Steps_1_3` eight arithmetic operations were made over each element of data, in the routine `Steps_4_8` – thirty-two operations. The total number of operations per one element is 8 + 32 = 40, so the real performance of the processor is:

(40 operations * 256 elements) / 349 cycles = ~29.3 arithmetic operations per cycle.

Tab. 3 The performance of NeuroMatrix® NM6403 on 256-point FHT.

Description	Number of arithmetic operations per cycle	Number of arithmetic operations per second at 50 MHz
Effective performance	5.8	$2,9*10^8$
Operational performance	29.3	$1,4*10^9$

Here is the comparative table of time spent for 21-step Hadamard transform execution according to the testing results:

Tab. 4 Performance comparison results on 2^{21} FHT.

Processor	Frequency	Execution Time
Pentium II	300 MHz	2.58 sec
NM6403	40 MHz	0.42 sec
Alfa 21164	533 MHz	0.33 sec

Bibliography

1) RC Module "NeuroMatrix® NM6403. Architectural Overview". http://www.module.ru/files/archover.pdf

2) RC Module. " NeuroMatrix® NM6403 SDK. Assembly Language Overview". http://www.module.ru/files/asmover.pdf

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

©RC Module, 1999

All rights reserved.

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

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

Printed in Russia Data of issue: 1999 April, 7