POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Algorithms Parallelization in ASIC Design

Relatori:
Prof. Mariagrazia Graziano
Prof. Marco Vacca
Prof. Maurizio Zamboni

Candidata:
Gina Jiang

October 2017

Acknowledgments

I would like to express my gratitude to my supervisor Prof. Mariagrazia Graziano for giving me the passion for digital hardware design and how to optimize it.

Other thanks to Prof. Maurizio Zamboni for giving me some suggestions during the writing of this thesis.

A special thanks to Marco Vacca, Giovanna Turvani and Giulia Santoro for giving me advices and clarifications of the algorithms implemented.
Without you this thesis work wouldn't have been possible.

Summary

In the last decades we have seen the computational time of integrated circuits been reduced more and more.
Especially an ASIC has major impact in this improvement. An ASIC (application-specific integrated circuit) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. This thesis presents efficient implementation of various algorithms by using the ASIC approach while exploiting the parallelization and computing optimization whenever possible. The goal is to compute using the ASAP (As Soon As Possible) approach and, therefore, also to present a pareto efficiency.

Futhermore this thesis is used to be compared with emergent and modern general purpose multi-core processors. In particular we will do some comparisons on the Logic-In-Memory architecture and the systolic array architecture regarding the clock cycles required and the number of resources used.
The Logic-In-Memory architecture is a general purpose processor where logic and memory are embedded in a unique entity. These entities are interconnected together to allows the exchanging of data among different units. The systolic array architecture is made of arrays of processors which are connected to a small number of nearest neighbours in a mesh-like topology. Processors perform a sequence of operations on data that flows between them.

We have choosen and implemented, with ASIC approach, a set of algorithms used for image processing, filter, image compression, and other use.
The main preferred characteristics are the parallelism, the required data storage, the communication between distant and near cells, and data dependencies.

- **Summed area Table**: is an algorithm for quickly and efficiently generating the sum of values in a rectangular subset of a grid. Here we can perform some addition in parallel and wisely exploit some data dependencies between near and distant cells.

- **Discrete Cosine Transform**: a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. This computation is divided in two parts: the first one perform all the N^2 cosine products in parallel, the second one perform the N sums, each sum has as addends N cosine products.

- **Binomial Filter**: is a smoothing filters used to enhance noisy images (at the

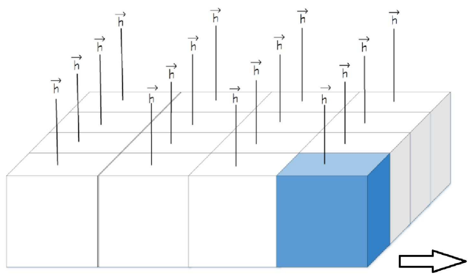
expense of blurring). It requires the comunication of all cells in the neighborhood, and all cells can perform in parallel. Here we can exploit some partial results to be used by other cells.

- **Finite impulse Response**: is a filter structure that can be used to implement almost any sort of frequency response digitally. It has been chosen, because it requires some memory to store the previous input, the multiplication can be done in parallel, while the additions has to be in sequence.
- **Transport equation problem**: it describes physical phenomena where particles, energy, or other physical quantities are transferred inside a physical system. Each cell has to perform vaious operations and communicate with near cells.
- **Magnetostatic field calculation 3D**It computes the magnetostatic field of a single cells with the magnetostatic field contributions between this cell and all the cells in a given 3D space. All cells compute its own contributions in parallel, and then we perform the summation of all these contributions.

Some algorithms have been already implemented on the LIM architecture but not in a "smart" way. The Summed Area Table and the Binomial Filter can be reimplemented using my approach where some partial results can be stored and exploited by other cells (instead of re-evaluating them).

Last but not least, we will do some analysis with the experimental Thessa tool that, given a C code, generate a parallel code which can be used to program the LIM.
We will give some advice on how to maximize the parallelism for this tool.

$$\sum_{k'=1}^{N_z}\left(\sum_{i'=1}^{N_x}\sum_{j'=1}^{N_y}\begin{bmatrix} TG_{xx} & TG_{xy} & TG_{xz} \\ TG_{yx} & TG_{yy} & TG_{yz} \\ TG_{zx} & TG_{zy} & TG_{zz} \end{bmatrix}_{(i-i',j-j',k-k')}\begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}_{(i',j',k')}\right)$$



$$\begin{bmatrix} TG_{xx} & TG_{xy} & TG_{xz} \\ TG_{yx} & TG_{yy} & TG_{yz} \\ TG_{zx} & TG_{zy} & TG_{zz} \end{bmatrix}_{(i-i',j-j',k-k^*)}\begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}_{(i',j',k^*)}$$

Figure 1: A section of the Magnetostatic field calculation 3D, where each cell perform the same operation and then we sum all the results

Table of contents

List of figures

Chapter 1

Introduction

The topic discussed in this work of thesis is based on an ASIC (Application-specific integrated circuit) architecture to implement some algorithm regarding the image processing and imange filtering.

1.1 Advantages of an ASIC based design

An application-specific integrated circuit (ASIC), is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use.
The advantages of an ASIC can be divided into four major areas: Unit cost, performance, power consumption and flexibility.

1.1.1 Unit cost

One of the biggest advantages of an ASIC based product with respect to a general purpose product is the lower unit cost once a certain volume has been reached. Unfortunately the volume required to offset the high NRE costs of an ASIC is very high which means that many projects are never a candidate for ASICs.[1]

1.1.2 Performance

Another advantages for using an ASIC is the higher performance. It has been done a comparison of various designs and it was found that an ASIC design was on average 3.2 times faster than an FPGA [2].

1.1.3 Power consumption

An ASIC architecture usually has lower power consumption than a comparable FPGA. This is due to the reconfigurability of the FPGA: there is a lot of logic in an FPGA which is used only for configuration. While the dynamic power consumption of the reconfiguration logic is practically 0, all of the configuration logic contributes to the leakage power.[1]

Chapter 2

Integral Image Algorithm

The *integral image algorithm*, also known as *summed area table* is a two-dimensional table generated from an input image. Integral image is a very popular and important algorithm in computer vision and computer graphics applications. Especially in real-time computer vision, it is usually used to accelerate calculating the sum of a rectangular area.

The aim of the IIA is to calculate the sum of values in a rectangular subset of a grid. In particular, the final value of a point inside a grid can be written as:

$$I(x,y) = \sum_{x'=1}^{x} \sum_{y'=1}^{y} i(x',y') \tag{2.1}$$

As shown in the figure 2.1, the value of the dark blue rectangle can be calculated as the sum of all the light blue rectangles.

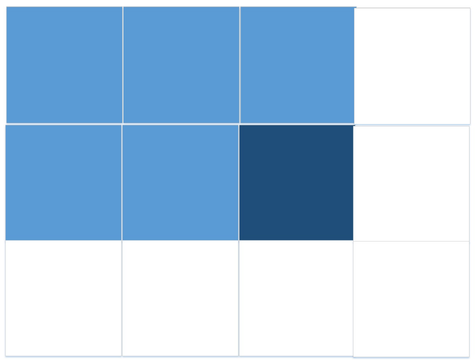


Figure 2.1: SAT of a single cell

2.1 Method used

I decided to use the approach ASAP (as soon as possible), so in this implementation we try to maximize the parallelism. To better explain, we first look to the Integral Image of a 1D array. At each step i we have the array partitioned in array of size 2^i, and each of them has its own integral image values.
In the next step *i+1*, we pair two consecutive array of size 2^i, (thus having an array of 2^{i+1}) and do the Integral image.
Since we have already computed the partial integral image, for the new one, the elements from *1* to 2^i has the correct values, while for the elements from 2^i to 2^{i+1} we need to sum the value stored in position 2^i. Figure 2.2 represents this method, while figure 2.3 shows an example of an 8-array.

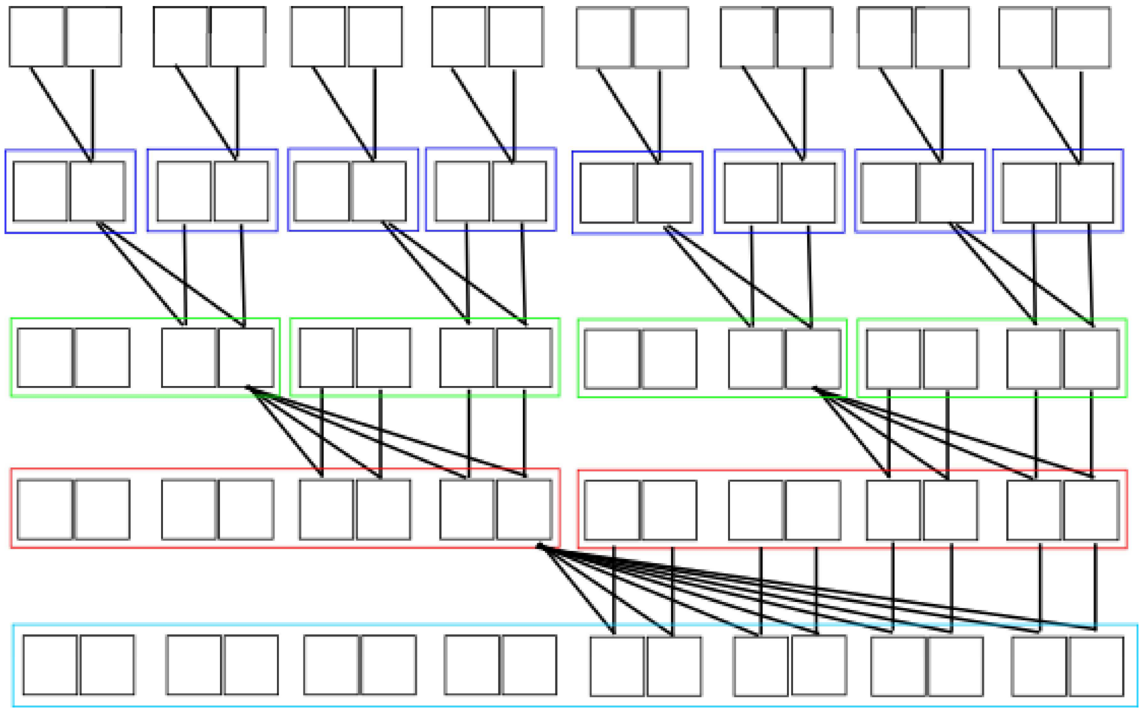


Figure 2.2: Schematic of the SAT of a 1D array

When we want to handle the integral image of a 2D array, we can compute the same method both vertically and horizontally.

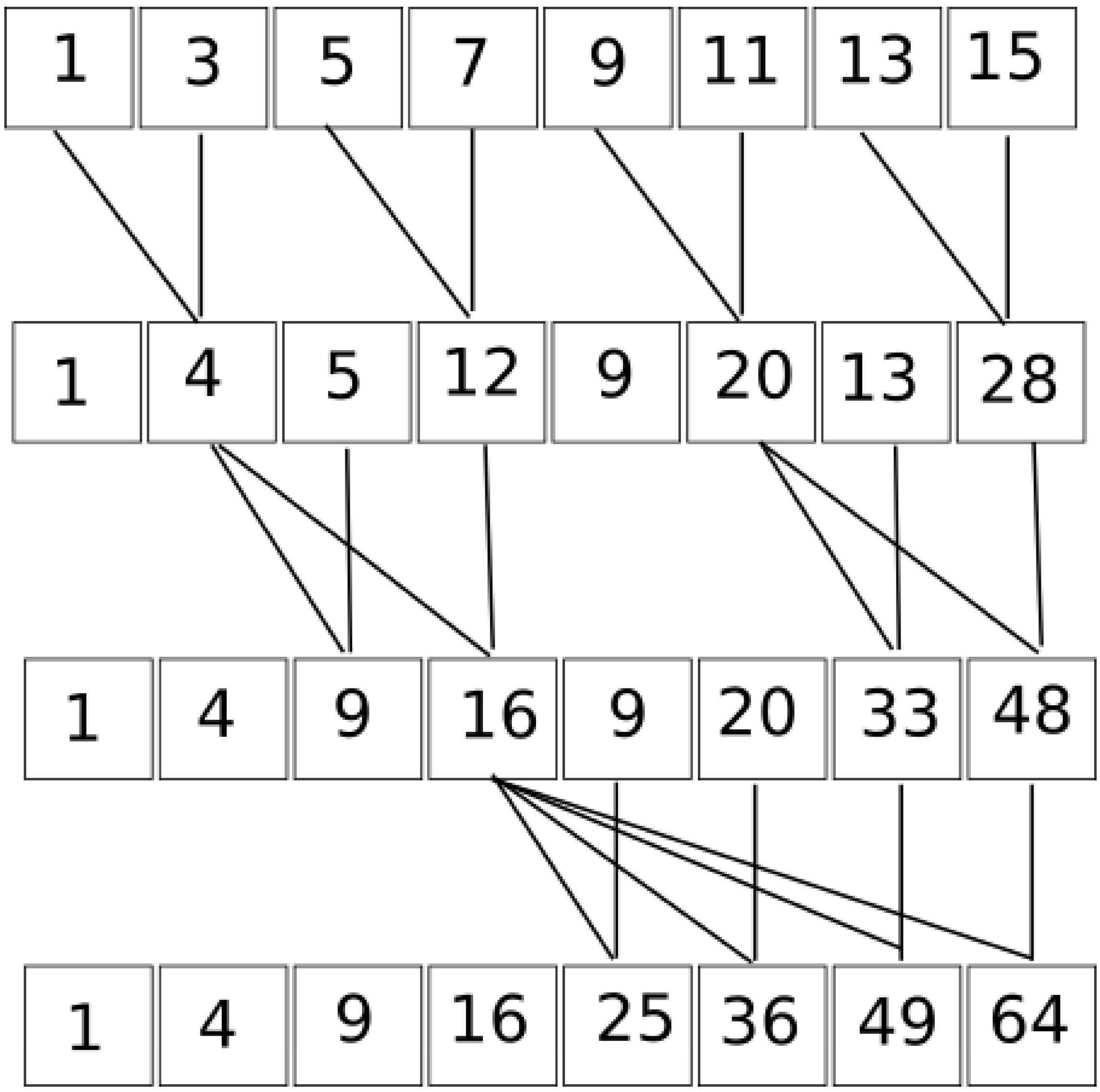


Figure 2.3: SAT of a 1D array with some values

2.2 Hardware Implementation

The algorithm has been properly implemented for an ASIC architecture that privileges the throughput (i.e. computes as fast as possible). The pseudocode is the following:

1. Divide the NxN matrix into small 2x2 matrices and compute the corresponding summed area table (SAT). This step is shown in figures 2.4 and 2.5

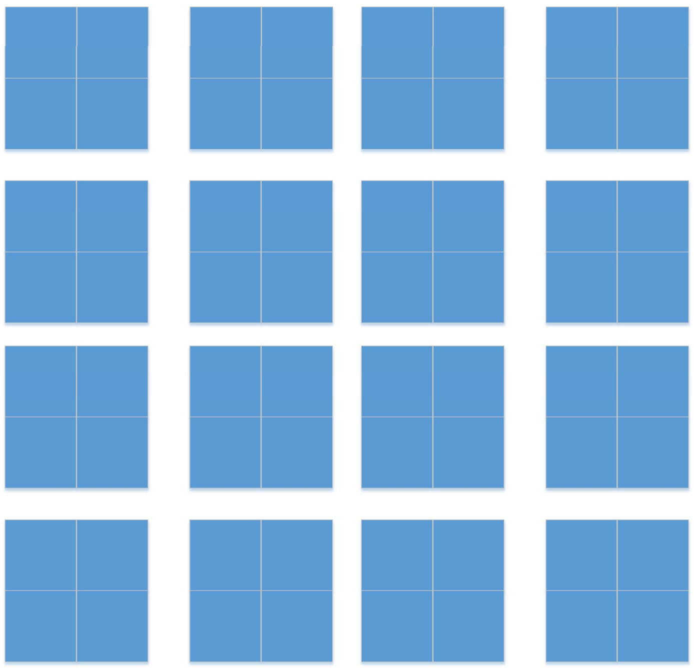


Figure 2.4: Subdivision in 2x2 matrices

2. Group all the above matrices into 4 matrices and evaluate the corresponding SATs. The computation will be faster because we have some partial SATs. When we sum vertically, we just sum to all the lower half cells with the corresponding cell of the last row of the upper half matrix. This is done because the last row of the upper half contains the partial SAT of the small matrices computed in the previous step. Figure 2.7 shows that after this operation the left half side of this 4x4 matrix has its SAT computed. Similarly we sum horizontally: as shown in figure 2.8 we add to all cells of the right side the

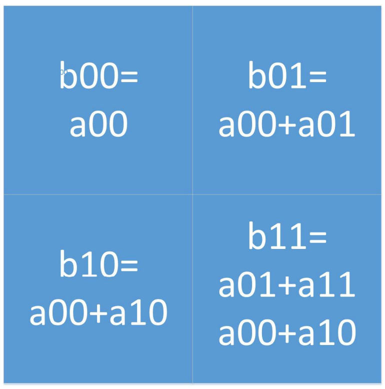


Figure 2.5: SAT of a 2x2 matrix

corresponding cell of the last column of the left half matrix. Now all cells of this 4x4 matrix has the SAT computed

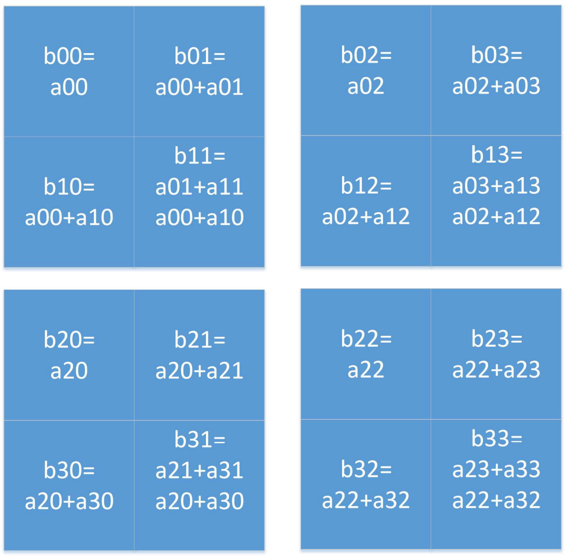


Figure 2.6: Step 2-Initial state of a 4x4 matrix

3. We repeat the prevoius step until we reach the matrix of dimension NxN

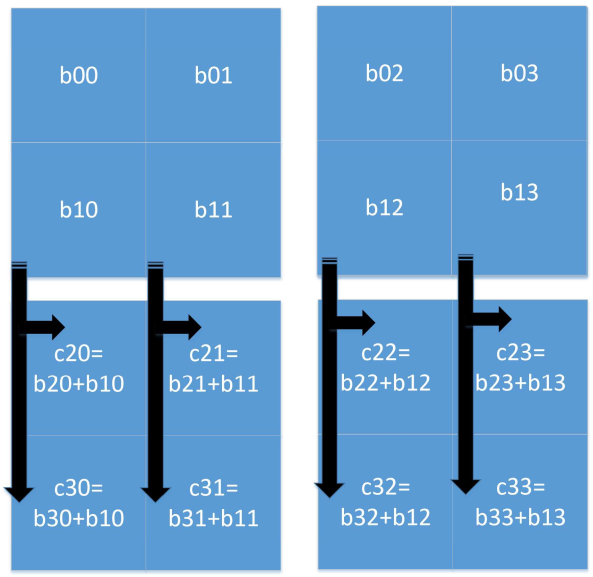


Figure 2.7: Step 2-Sum vertically.

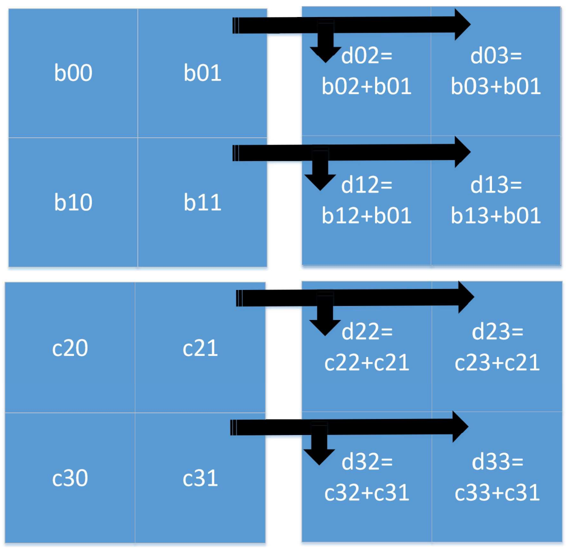


Figure 2.8: Step 2-Sum horizonatlly.

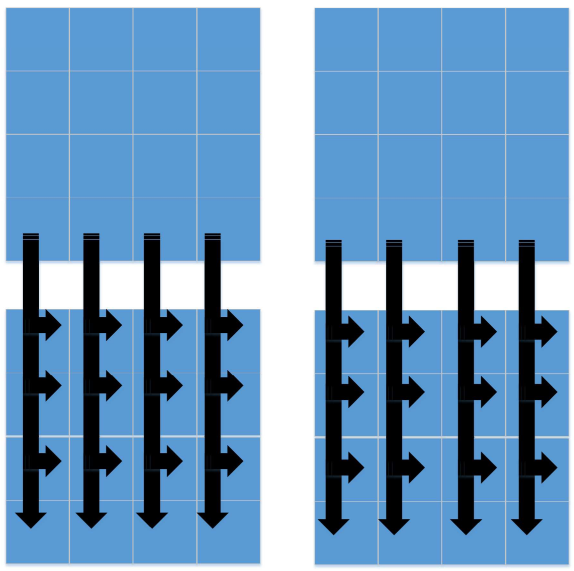


Figure 2.9: Step 3-Sum vertically.

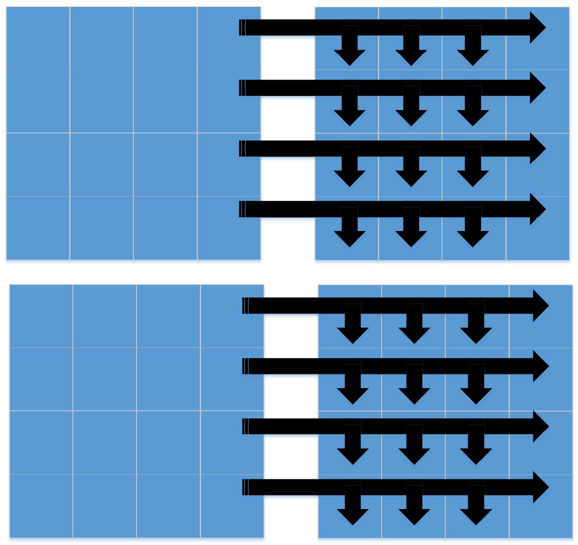


Figure 2.10: Step 3-Sum horizonatlly.

2.3 Pipelined Implementation

As explained before, in every step we divide the original matrix into small matrices of the size 2^ix2^i (i represent the number of the actual step), and compute the SAT of each matrix independently from one another. The computed SATs will be used in the next step to evaluate the SATs of matrices with size 2^{i+1}x2^{i+1}. We repeat this process until we reach the SAT of the original matrix. Figure 2.11 represent the scheme of this implementation for a 8x8 matrix. The simbol $\bigoplus$ means that it will evaluate the SAT of a matrix 2^{i+1}x2^{i+1} by using the SATs of 4 matrices 2^ix2^i. By looking to the picture, we can clearly imagine to put some pipeline registers. This is possible because in each step i we need only the values calculated in the previous step $i-1$, we don't need the other values found in the other steps.

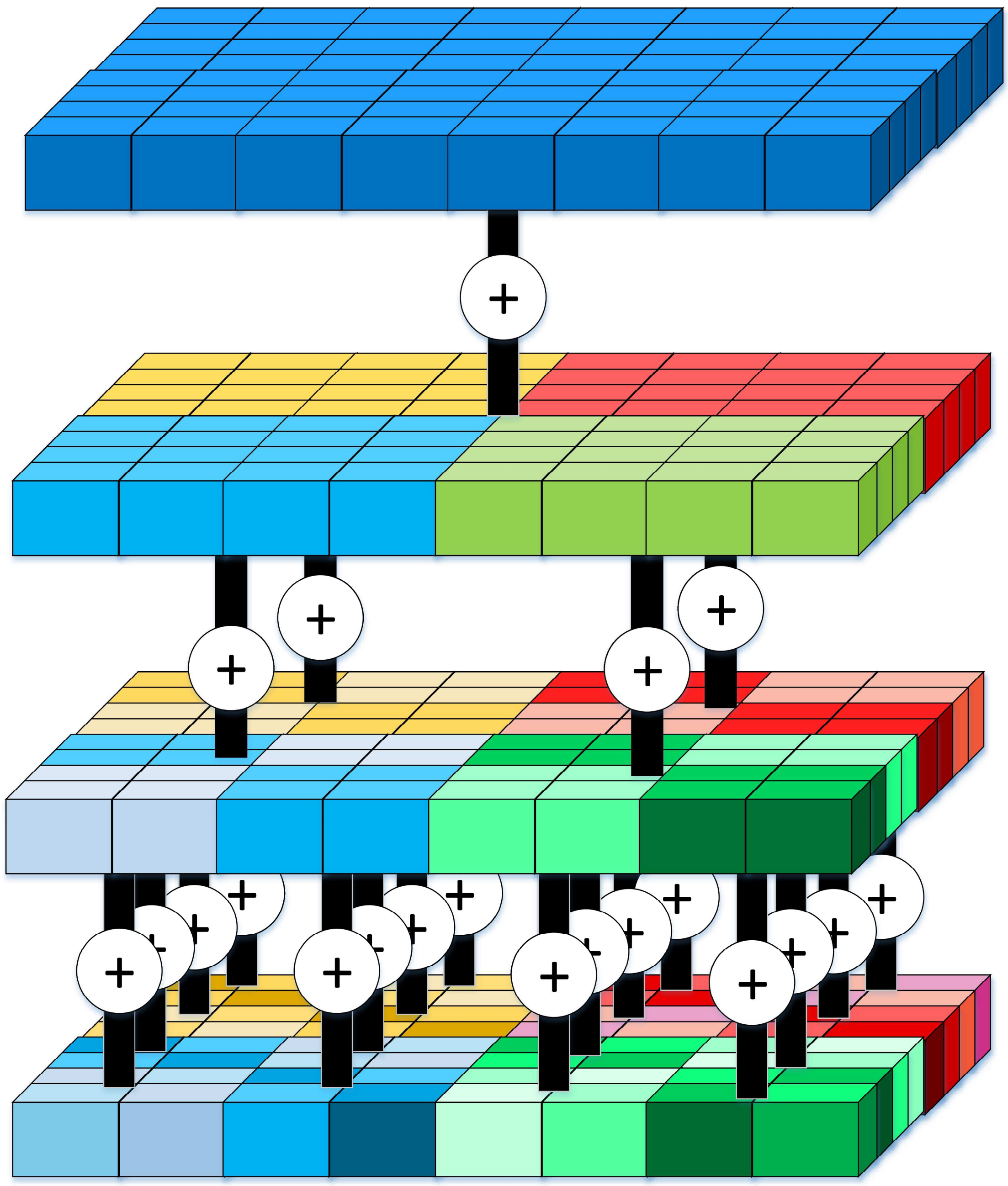
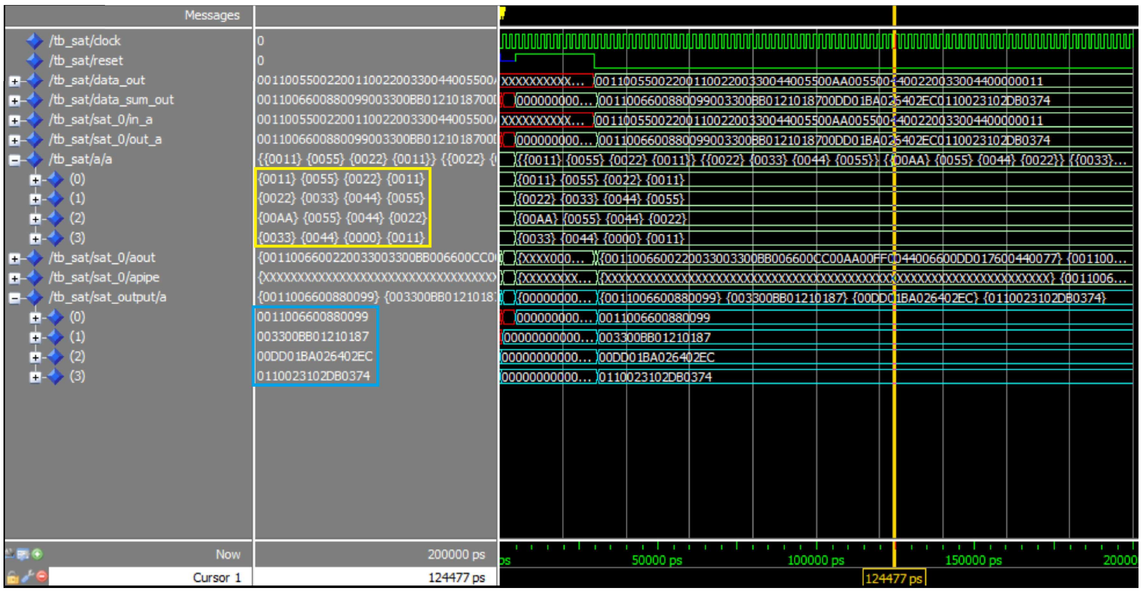


Figure 2.11: Scheme of the SAT algorithm's imlementation.

2.4 Simulation and Test

I did some tests of this implementation with different size and dimension. In the fig.2.12 we can see the results of a matrix 4x4.



```
Messages
/tb_sat/clock              0
/tb_sat/reset              0
/tb_sat/data_out           00110055002200110022003300440055004
/tb_sat/data_sum_out       001100660088009900330OBB01210187000
/tb_sat/sat_0/in_a         00110055002200110022003300440055004
/tb_sat/sat_0/out_a        001100660088009900330OBB01210187000
/tb_sat/a/a                {{0011} {0055} {0022} {0011}} {{0022} {
  (0)                      {0011} {0055} {0022} {0011}
  (1)                      {0022} {0033} {0044} {0055}
  (2)                      {00AA} {0055} {0044} {0022}
  (3)                      {0033} {0044} {0000} {0011}
/tb_sat/sat_0/aout         {00110066002200330033000BB006600CC0
/tb_sat/sat_0/apipe        {XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
/tb_sat/sat_output/a       {0011006600880099} {003300BB01210187
  (0)                      0011006600880099
  (1)                      003300BB01210187
  (2)                      00DD01BA026402EC
  (3)                      0110023102DB0374
Now                        200000 ps
Cursor 1                   124477 ps
```

Figure 2.12: Integral Image-Results of the simulation

The input matrix is the one highlighted in yellow, while the output matrix is highlighted in light blue. We start from these values

$$\begin{bmatrix} 0x11 & 0x55 & 0x22 & 0x11 \\ 0x22 & 0x33 & 0x44 & 0x55 \\ 0xAA & 0x55 & 0x44 & 0x22 \\ 0x33 & 0x44 & 0x0 & 0x11 \end{bmatrix} = \begin{bmatrix} 17 & 85 & 34 & 17 \\ 34 & 51 & 68 & 85 \\ 170 & 85 & 68 & 34 \\ 51 & 68 & 0 & 17 \end{bmatrix}$$

we partition the original matrix in matrices 2x2

$$\begin{bmatrix} \begin{pmatrix} 0x11 & 0x55 \\ 0x22 & 0x33 \end{pmatrix} & \begin{pmatrix} 0x22 & 0x11 \\ 0x44 & 0x55 \end{pmatrix} \\ \\ \begin{pmatrix} 0xAA & 0x55 \\ 0x33 & 0x44 \end{pmatrix} & \begin{pmatrix} 0x44 & 0x22 \\ 0x0 & 0x11 \end{pmatrix} \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} 17 & 85 \\ 34 & 51 \end{pmatrix} & \begin{pmatrix} 34 & 17 \\ 68 & 85 \end{pmatrix} \\ \\ \begin{pmatrix} 170 & 85 \\ 51 & 68 \end{pmatrix} & \begin{pmatrix} 68 & 34 \\ 0 & 17 \end{pmatrix} \end{bmatrix}$$

and we perform the summed area table for each matrix 2x2

$$\begin{bmatrix} \begin{pmatrix} 17 & 17+85 \\ 34+17 & 51+17+85+34 \end{pmatrix} & \begin{pmatrix} 34 & 17+34 \\ 68+34 & 85+17+34+68 \end{pmatrix} \\ \\ \begin{pmatrix} 170 & 85+170 \\ 51+170 & 68+170+85+51 \end{pmatrix} & \begin{pmatrix} 68 & 34+68 \\ 0+68 & 17+68+34 \end{pmatrix} \end{bmatrix} =$$

$$\begin{bmatrix} \begin{pmatrix} 17 & 102 \\ 51 & 187 \end{pmatrix} & \begin{pmatrix} 34 & 51 \\ 102 & 204 \end{pmatrix} \\ \\ \begin{pmatrix} 170 & 255 \\ 221 & 374 \end{pmatrix} & \begin{pmatrix} 68 & 102 \\ 68 & 119 \end{pmatrix} \end{bmatrix}$$

If we translate it in hexdecimal we get

$$\begin{bmatrix} \begin{pmatrix} 0x11 & 0x66 \\ 0x33 & 0xBB \end{pmatrix} & \begin{pmatrix} 0x22 & 0x33 \\ 0x66 & 0xCC \end{pmatrix} \\ \\ \begin{pmatrix} 0xAA & 0xFF \\ 0xDD & 0x176 \end{pmatrix} & \begin{pmatrix} 0x44 & 0x66 \\ 0x44 & 0x77 \end{pmatrix} \end{bmatrix}$$

If we look the signal *aout* on the fig 2.12, we can see the partial summed area table.

$$aout =$$
$$0x00110066002200330033 00BB006600CC00AA00FF0044006600DD017600440077$$

we know that each row has 4 elements of 16 bits

$$\begin{bmatrix} 1^{st}row = 0x0011006600220033 \\ 2^{nd}row = 0x003300BB006600CC \\ 3^{rd}row = 0x00AA00FF00440066 \\ 4^{th}row = 0x00DD017600440077 \end{bmatrix} = \begin{bmatrix} 0x11 & 0x66 & 0x22 & 0x33 \\ 0x33 & 0xBB & 0x66 & 0xCC \\ 0xAA & 0xFF & 0x44 & 0x66 \\ 0xDD & 0x176 & 0x44 & 0x77 \end{bmatrix}$$

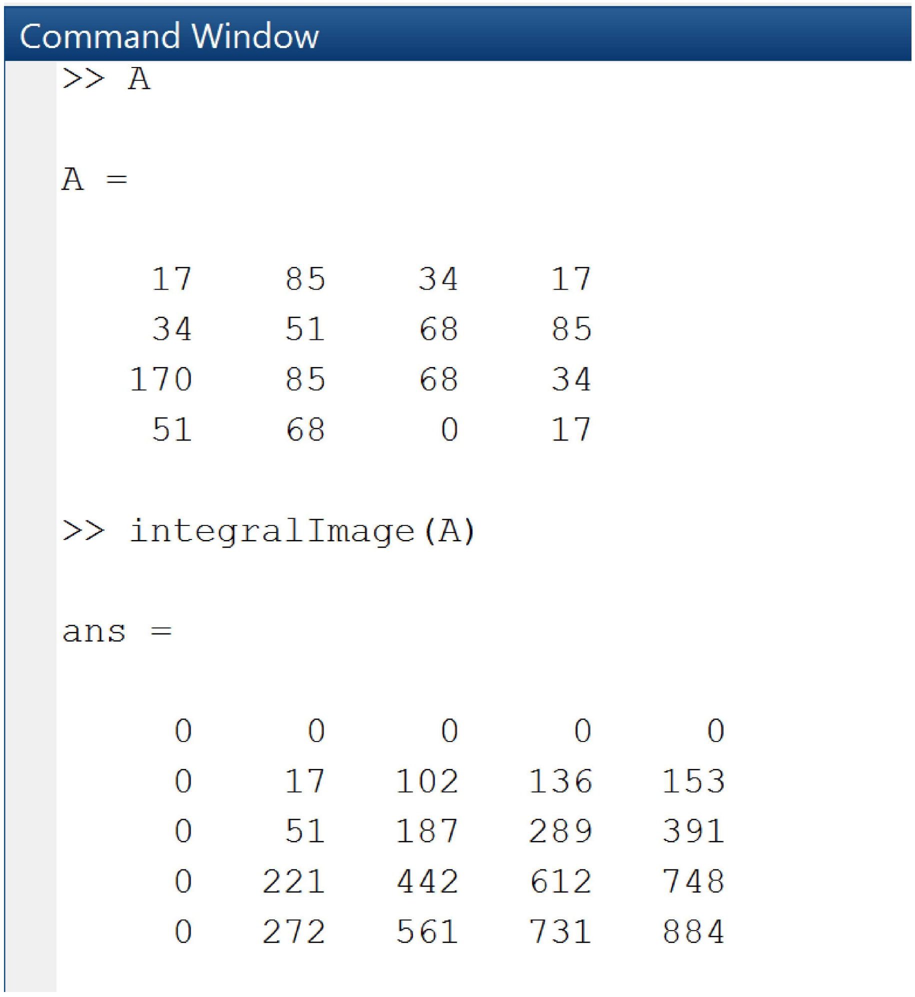
which is the same we obtained above.
Now we have to evaluate the summed area table of the matrix 4x4 by using the partial sum we have just found. As described before (section 2.2), when we sum vertically, we sum to all the lower half cells with the corresponding cell of the last row of the upper half matrix. Similarly we sum horizontally

$$\begin{bmatrix} 17 & 102 & 34 & 51 \\ 51 & 187 & 102 & 204 \\ 170 & 255 & 68 & 102 \\ 221 & 374 & 68 & 119 \end{bmatrix}$$
$$\Downarrow$$
$$\text{sum vertically}$$
$$\Downarrow$$
$$\begin{bmatrix} 17 & 102 & 34 & 51 \\ 51 & 187 & 102 & 204 \\ 170+51 & 255+187 & 68+102 & 102+204 \\ 221+51 & 374+187 & 68+102 & 119+204 \end{bmatrix}$$
$$\Downarrow$$
$$\text{sum horizontally}$$
$$\Downarrow$$
$$\begin{bmatrix} 17 & 102 & 34+102 & 51+102 \\ 51 & 187 & 102+187 & 204+187 \\ 221 & 442 & 170+442 & 306+442 \\ 272 & 561 & 170+561 & 323+561 \end{bmatrix}$$

Finally we get

$$\begin{bmatrix} 17 & 102 & 136 & 153 \\ 51 & 187 & 289 & 391 \\ 221 & 442 & 612 & 748 \\ 272 & 561 & 731 & 884 \end{bmatrix} = \begin{bmatrix} 0x11 & 0x66 & 0x88 & 0x99 \\ 0x33 & 0xBB & 0x121 & 0x187 \\ 0xDD & 0x1BA & 0x264 & 0x2EC \\ 0x110 & 0x231 & 0x2DB & 0x374 \end{bmatrix}$$

Which is the same result given in the simulation (fig 2.12, signal *sat_output/a*) and also by MATLAB as shown below



```
>> A

A =

    17    85    34    17
    34    51    68    85
   170    85    68    34
    51    68     0    17

>> integralImage(A)

ans =

     0     0     0     0     0
     0    17   102   136   153
     0    51   187   289   391
     0   221   442   612   748
     0   272   561   731   884
```

Figure 2.13: Integral Image-Checking with MATLAB

2.5 Comparison

There are two typical existed image integral algorithms on GPUs. The first is the Inclusive Scan algorithm. The second is the Balanced Trees Parallel Scan algorithm. Compared with the Inclusive Scan algorithm, the proposed scheme reduces logarithmically the global computation time without any pipeline (with pipeline, the improvement will be even greater). Compared with the Balanced Trees Parallel Scan algorithm, the proposed algorithm only needs about half of the global computation time. The theoretical implementation shows that the proposed algorithm gets the best performance compared with the two above integral algorithms.

2.5.1 Inclusive Scan algorithm

We perform the inclusive scan for each row and evaluate the partial sum. Finally we perform the inclusive scan for each column. Figure 2.14 shows an example of this implementation for a matrix 3x3. [6]

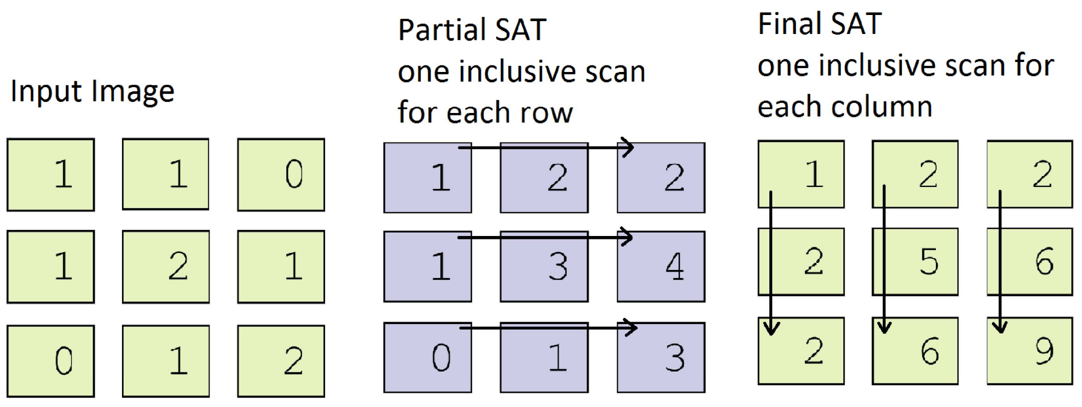


Figure 2.14: Scheme of the inclusive scan algorithm

2.5.2 Balanced Trees Parallel Scan algorithm

For simplicity, i will explain the algorithm for a 1D-array. The goal is to evaluate the SAT of a single row which is also known as the prefix sum, also called cumulative sum, inclusive scan. [4]

A prefix sum can be calculated in parallel by the following steps.

1. Build a balanced binary tree on the input data and sweep it to and from the root.
2. Traverse down from leaves to root building partial sums at internal nodes in the tree.
3. Traverse back up the tree building the scan from the partial sums.

Figure 2.15 shows the scheme for an array of dimension 16. For a matrix, we implement this algorithm for each row and then for each column.

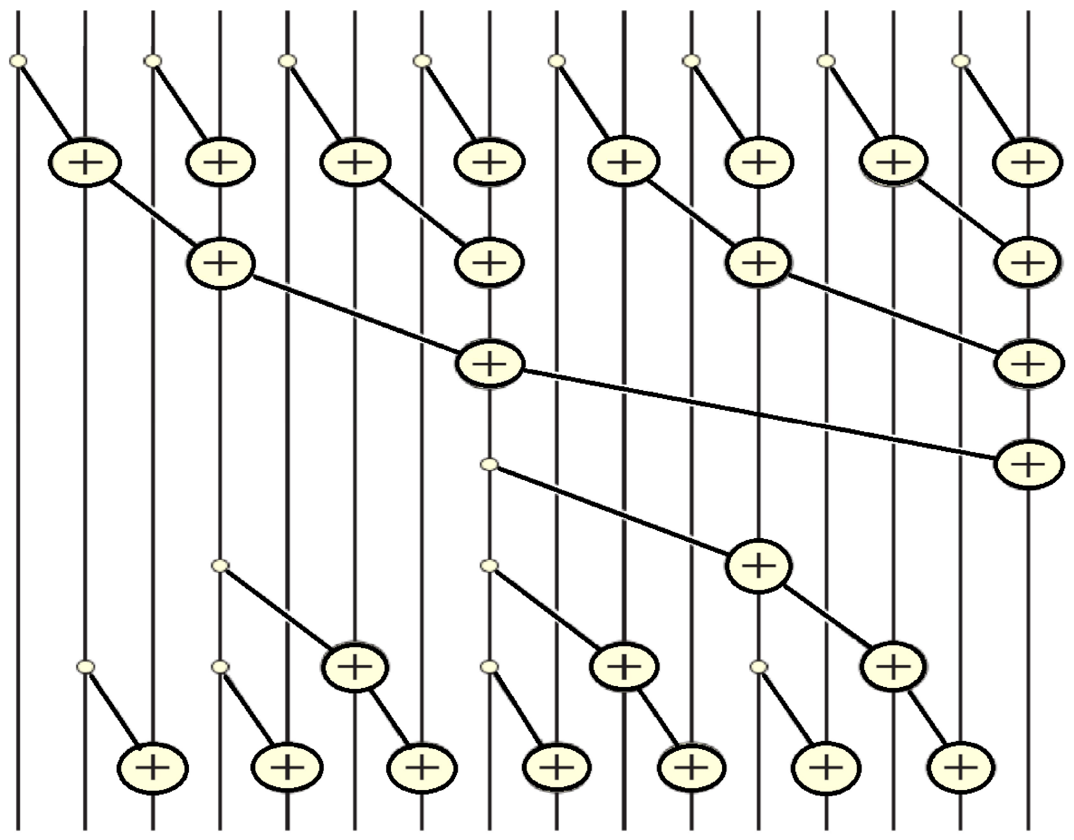


Figure 2.15: Parallel Balnced Tree Algorithm for a 1D array

2.5.3 Comparing the 3 algorithm implementations

Now that we know the implementation of these algorithm we can compare the area, i.e. the number of adders required, and the time between them. In the table 2.5.3

the unit time t is the time required to get the result of a single sum.

	Inclusive scan	Balanced tree	This work	This work pipelined
Area	$2N(N-1)$	$2N(2N - log_2 N - 2)$	$N \cdot N \cdot log_2 N$	$N \cdot N$
Time	$2(N-1) \cdot t$	$2(2log_2 N - 1)t$	$(log_2 N)2t$	$2t$

2.5.4 Logic-In-Memory architecture

The *Logic-In-Memory* is a new architecture where logic and memory are embedded as unique entity instead of two separated ones (fig. 2.16). [8]

The memory is made of small entities called *bricks* populating two different layers: the bricks of each layer can communicate each other like in a matrix structure and only specifc bricks, called *pillars*, are allowed to exchange data with the upper layer(fig. 2.17).

The upper layer is composed of a smaller number of bricks having a bigger memory capacity than the ones in the bottom layer: here the bricks are more in number but their memory capacitance is small. In addition, only this plane is able to communicate with the programmable logic layer: it is made of one ALU (arithmetic and logic unit) for each brick of the bottom layer.

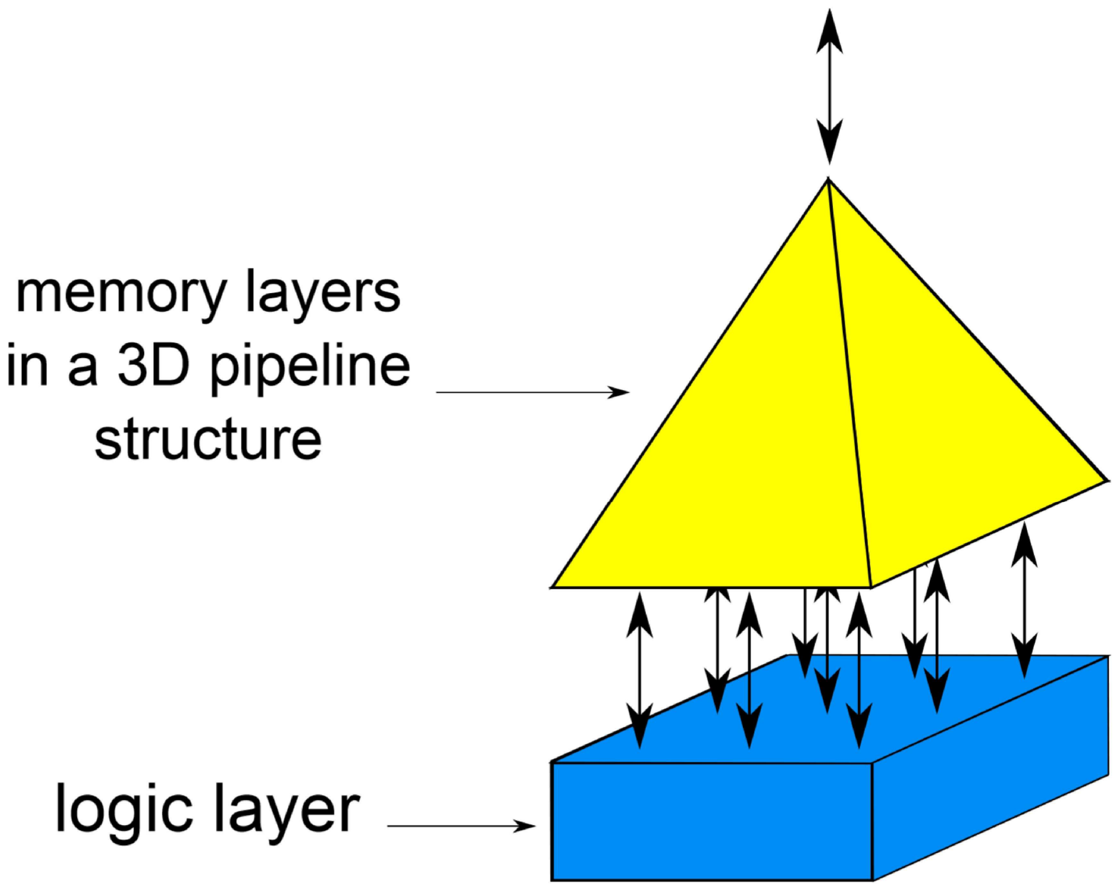


Figure 2.16: Graphics representation of the architecture 2.0: the yellow pyramid represent the 3D pipeline of smart memories whereas the blue layer represent the programmable logic layer.

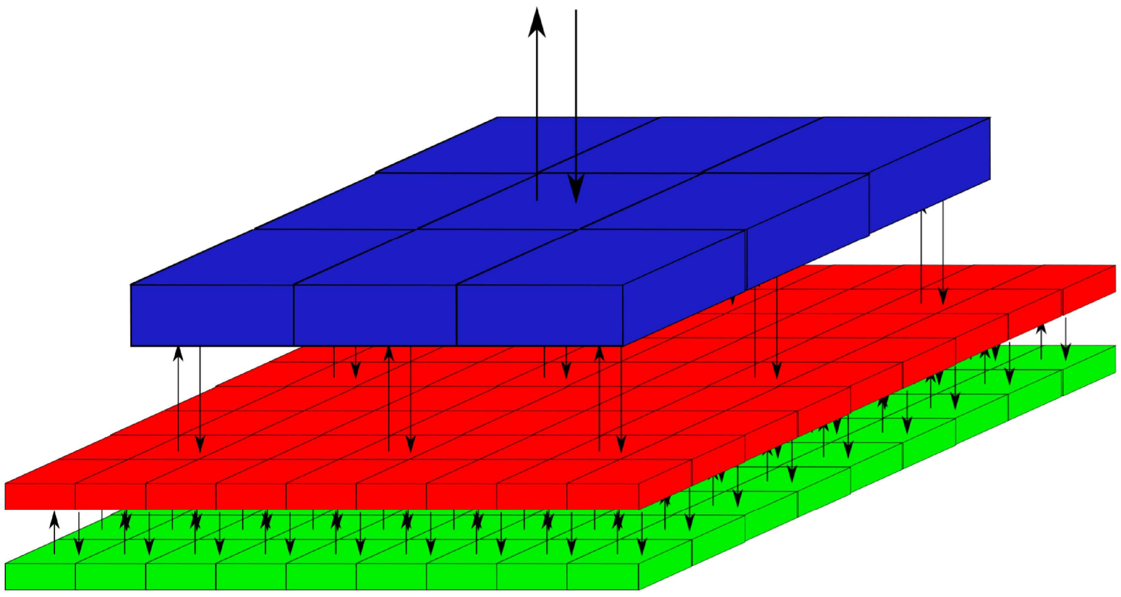


Figure 2.17: Structure of the LIM. The green layer represent the logic and is composed of 81 ALUs communicating each one with every brick of the bottom layer (red); the blue layer represent the upper plane and is composed of 9 bricks, one every 9 bricks of the bottom layer. Only the pillars of the bottom layer can communicate with the upper layer; only the pillar of the upper layer can exchange data with the outside.

2.5.5 Comparison with the LIM architecture

I take the results of the LIM's architecture The pseudocode for the LIM architecture is the following:

1. Each cell reads its own value;
2. Sums its value to the one received from the NORTH and sends the result to the SOUTH;
3. Samples the value coming from the WEST, sums it to the previous result and sends it to the EAST;
4. Writes the final result in its own memory;

The main advantage of the LIM architecture is the parallelism: the presence of many cells that can work autonomously greatly increases the speed of the computation of algorithms that can be executed in parallel. The duration of the algorithm depends on the number of cells of the grid: in particular, supposing to have a grid of $N \cdot N$ cells, the total duration is equal to:

- N clock cycles to write the data;
- 6N clock cycles to compute the algorithm;
- 2N+1 clock cycles to read the data through a remote read.

Therefore,

$$t = N + 6N + 2N + 1 = 9N + 1$$

	LIM	This work	This work pipelined
Area	$N \cdot N$ cells	$N \cdot N \cdot log_2 N$ adders	$N \cdot N$ adders
Time	$9N + 1$ cycles	Time for $2(log_2 N)$ adders	Time for 2 adders

2.6 Characteristics of the Integral Image Algorithm

PROCESSING ELEMENTS

1- Which kind of Processing element?

Adder

2- Functionality

Addition, summation

3- Complexity

Not pipelined	Area $N \cdot N \cdot log_2(N)$ adders
	Time($log_2 N$)2x(time for an addition)
Pipelined	Area N^2 adders
	Time 2x(time for an addition)

4- Parallelism

All cells perform their Integral Image value in parallel.

5-Reconfigurability

No

6- Programmability

No

7- Need a dedicated memory?

If the architecture is pipelined, you need to store the partial sum.

If not pipelined, no memory is required.

8- Relationship with I/O

INPUT: values of the matrix's cells

OUTPUT: result of the integral image algorithm

MEMORY ELEMENTS

1- Need a clever memory LIM?

No, but can be implemented

2- Is there a data search algorithm?

No

3-Interface mechanism with other PE or memories

Communication required between cells (even between distant cells e.g. the center cell and the last cell).

4- Access mechanism

(No memory for this implementation)

5- Hierarchization

(No memory for this implementation)

6- Cache coherency

(No memory for this implementation)

7- Is it a a transactional memory?

(No memory for this implementation)

8- Are there virtualization (paging) mechanisms?

(No memory for this implementation)

ENCODING INFORMATION

1-Which encoding is used?

Binary encoding

CONNECTIONS

1-Packet Exchange Protocol
Directly

2-Timing (asynchronou/synchronous)
Synchronous

3-Are there multiple instances?
Yes, but the summation is not the same for all (some cells requires more data exchanging than others)

4-Heterogeneity (Local/Distant I/O Connections)
Heterogeneous when storing the initial values to the cells, not heterogeneous when exchanging the data (sometimes you exchange data between local cells, the other times you exchange between distant cells.)

5-Are there any buffers?
There are pipeline registers.

Chapter 3

Discrete cosine transform

A discrete cosine transform (DCT) expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. DCTs are important to numerous applications in science and engineering, from lossy compression of audio (e.g. MP3) and images (e.g. JPEG). The DCT is often used in signal and image processing, especially for lossy compression, because it has a strong "energy compaction" property: in typical applications, most of the signal information tends to be concentrated in a few low-frequency components of the DCT. [9]

There are different variations of the discrete cosine transform, but the most common one is the "DCT-II" which is also used in JPEG image compression and by MATLAB ®:

$$X_k = \sum_{n=0}^{N-1} w(k) x_n cos\left[\frac{\pi}{N}(n+\frac{1}{2})k\right], \qquad k = 0,1,...,N-1 \tag{3.1}$$

$$w(x) = \begin{cases} \frac{1}{\sqrt{N}} & k = 0 \\ \\ \sqrt{\frac{2}{N}} & 1 \leqslant k \leqslant N-1 \end{cases}$$

We can see the equation 3.1 as a product of matrices: we have to multiply the input vector x_i with a matrix of coefficients for cosine. Thus the equation 3.1 can be rewritten as

$$\begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} cos_{0,0} & cos_{0,1} & \cdots & cos_{0,n-1} \\ cos_{1,0} & cos_{1,1} & \cdots & cos_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ cos_{n-1,0} & cos_{n-1,1} & \cdots & cos_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

where the matrix of coefficients for cosine is the following

$$\begin{bmatrix} \frac{1}{\sqrt{N}}cos(\frac{\pi}{N}(0+\frac{1}{2})\cdot 0) & \frac{1}{\sqrt{N}}cos(\frac{\pi}{N}(1+\frac{1}{2})\cdot 0) & \cdots & \frac{1}{\sqrt{N}}cos(\frac{\pi}{N}(N-1+\frac{1}{2})\cdot 0) \\ \sqrt{\frac{2}{N}}cos(\frac{\pi}{N}(0+\frac{1}{2})\cdot 1) & \sqrt{\frac{2}{N}}cos(\frac{\pi}{N}(1+\frac{1}{2})\cdot 1) & \cdots & \sqrt{\frac{2}{N}}cos(\frac{\pi}{N}(N-1+\frac{1}{2})\cdot 1) \\ \vdots & \vdots & \ddots & \vdots \\ \sqrt{\frac{2}{N}}cos(\frac{\pi}{N}(0+\frac{1}{2})\cdot N-1) & \sqrt{\frac{2}{N}}cos(\frac{\pi}{N}(1+\frac{1}{2})\cdot N-1) & \cdots & \sqrt{\frac{2}{N}}cos(\frac{\pi}{N}(N-1+\frac{1}{2})\cdot N-1) \end{bmatrix}$$

3.1 Hardware Implementation

3.1.1 Not pipelined version

The result can be seen as a sum of products. Each element of the output array can be written as:

$$X_i = x_0cos_{i,0} + x_1cos_{i,1} + \cdots + x_{n-1}cos_{i,n-1}$$

We can perform all the multiplications in parallel and then do an accumulator (or a chain of additions) with these product results. As we will see in the DCT synthesys result (10.2), there is no practical difference between the pipelined and not pipelined versions. The only difference is the area for the pipeline registers and the time for the addition chain over the multiplications (One addition is faster than one multiplications, however if the n number of additions in sequence to do is big enough, this time can offset the time for a single multiplication).

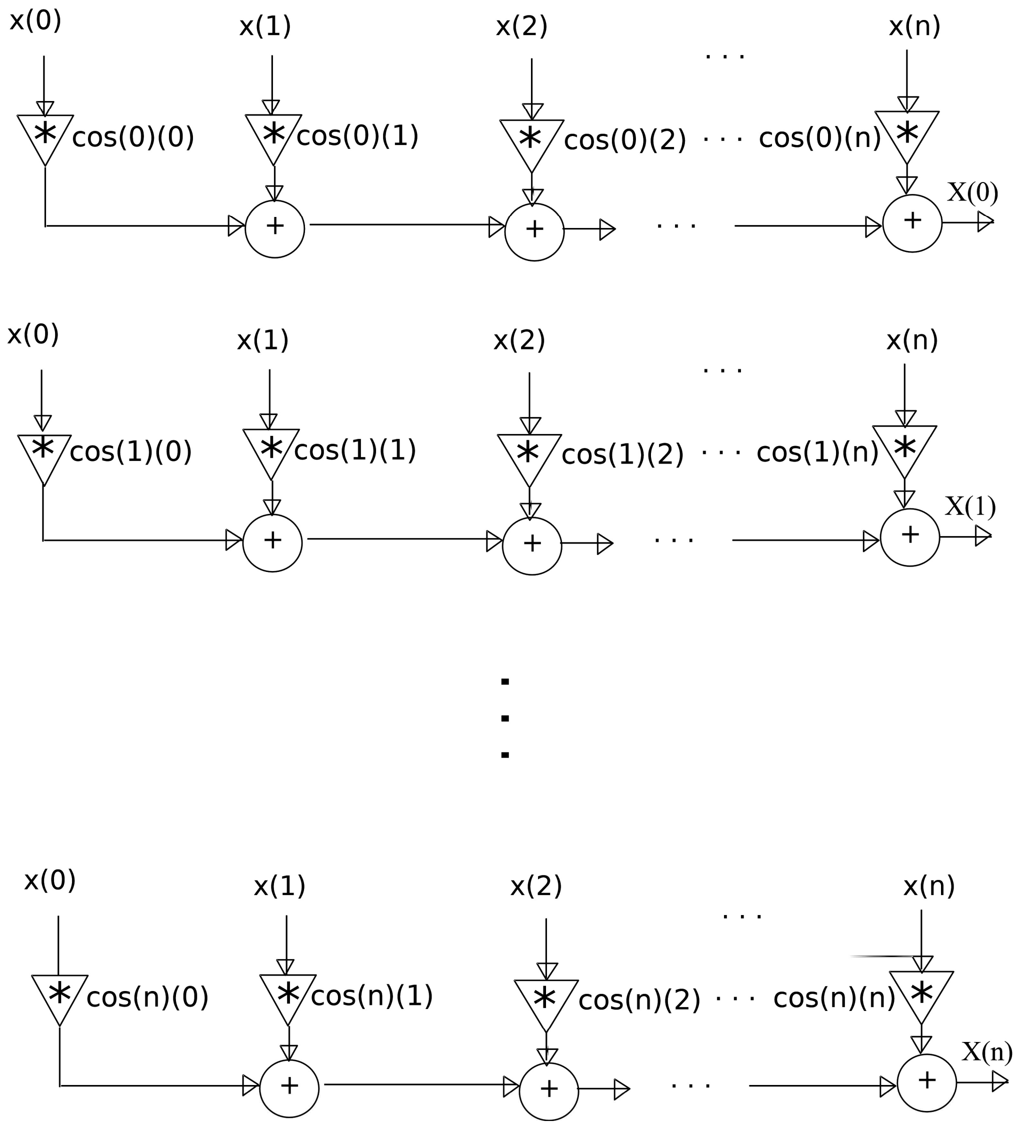


Figure 3.1: Implementation of DCT algorithm (not pipelined). All multiplications are done in parallel

3.1.2 Pipelined version

The result can be seen as a sum of products. Each element of the output array can be written as:

$$X_i = x_0 cos_{i,0} + x_1 cos_{i,1} + \cdots + x_{n-1} cos_{i,n-1}$$

Therefore, since there is no data dependencies between these elemnts, we can evaluate them in parallel.

1. In the first clock cycle we can perform the multiplication $x_0 cos_{i,0}$ for each element of the array.
2. In the second clock cycle we evaluate the second multiplication and sum it to the previous result $[x_0 cos_{i,0}] + x_1 cos_{i,1}$
3. In the third clock cycle we evaluate the third multiplication and sum it to the previous result $[x_0 cos_{i,0} + x_1 cos_{i,1}] + x_2 cos_{i,2}$
4. We repeat this process until we reach the last term of this sum of product. In this clock cycle we have the final result $X_i = x_0 cos_{i,0} + x_1 cos_{i,1} + \cdots + x_{n-1} cos_{i,n-1}$

In the following, we have the process explained in a matrix form, while figure 3.2 represents this implementation in a Register to Transfer Level.

$$\begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} x_0 cos_{0,0} & + & x_1 cos_{0,1} & + & \cdots & + & x_{n-1} cos_{0,n-1} \\ x_0 cos_{1,0} & + & x_1 cos_{1,1} & + & \cdots & + & x_{n-1} cos_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \underbrace{x_0 cos_{n-1,0}}_{1^{st} step} & \underbrace{+ x_1 cos_{n-1,1}}_{2^{nd} step} & & + & \cdots & \underbrace{+ x_{n-1} cos_{n-1,n-1}}_{last \quad step} & \end{bmatrix}$$

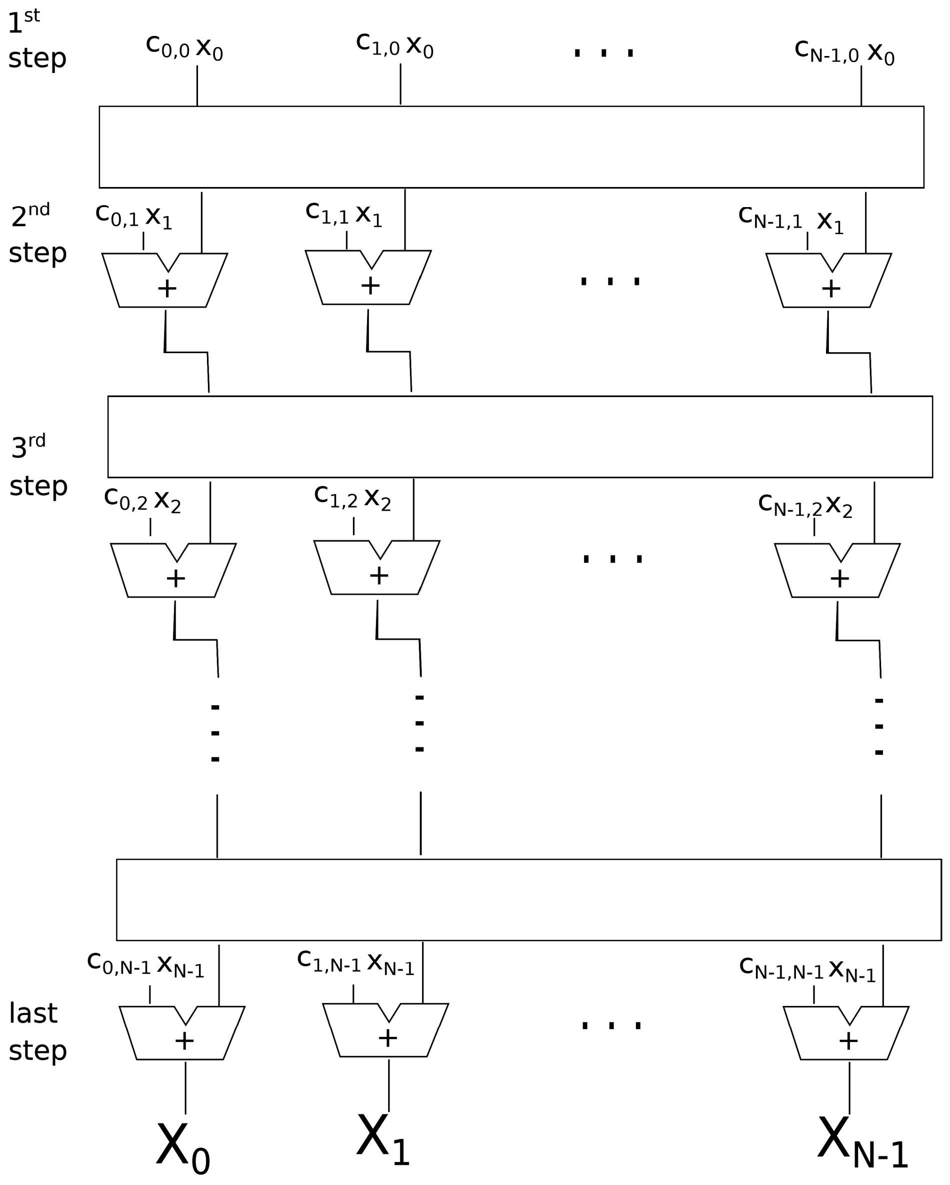
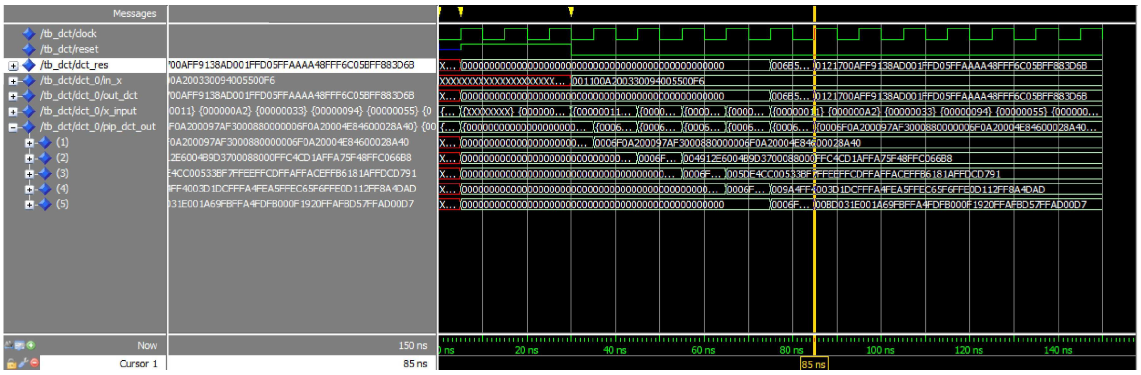


Figure 3.2: Implementation of DCT algorithm (pipelined)

3.2 Simulation and Test

I performed some tests of this implementation with different size and dimension. In the fig.3.3 we can see the results of an array of six elements.



```
Messages
/tb_dct/clock
/tb_dct/reset
/tb_dct/dct_res            '00AFF9138AD001FFD05FFAAAA48FFF6C05BFF883D6B
/tb_dct/dct_0/in_x         )0A200330094005500F6
/tb_dct/dct_0/out_dct      '00AFF9138AD001FFD05FFAAAA48FFF6C05BFF883D6B
/tb_dct/dct_0/x_input      0011} {000000A2} {00000033} {00000094} {00000055} {0
/tb_dct/dct_0/pip_dct_out  F0A200097AF3000880000006F0A20004E84600028A40} {00
  (1)                      '0A200097AF3000880000006F0A20004E84600028A40
  (2)                      12E6004B9D3700088000FFC4CD1AFFA75F48FFC066B8
  (3)                      E4CC00533BF7FFEEFFCDFFAFFACEFFB6181AFFDCD791
  (4)                      4FF4003D1DCFFFA4FEA5FFEC65F6FFE0D112FF8A4DAD
  (5)                      )31E001A69FBFFA4FDFB000F1920FFAFBD57FFAD00D7

Now       150 ns
Cursor 1  85 ns
```

Figure 3.3: Discrete cosine transform-Results of the simulation

The simulation shows the input array:

$$in\_x = 0x001100A200330094005500F6$$

that stands for

$$x_0 = 0x0011$$
$$x_1 = 0x00A2$$
$$x_2 = 0x0033$$
$$x_3 = 0x0094$$
$$x_4 = 0x0055$$
$$x_5 = 0x00F6$$

which in decimal is

$$x_0 = 17$$
$$x_1 = 162$$
$$x_2 = 51$$
$$x_3 = 148$$
$$x_4 = 85$$
$$x_5 = 246$$

During the first 6 clock cycles, we can see the partial sum and multiplications. The output is available in the 6^{th} clock cycle after the reset:

$dct\_res = 0x0121700AFF9138AD001FFD05FFAAAA48FFF6C05BFF883D6B$

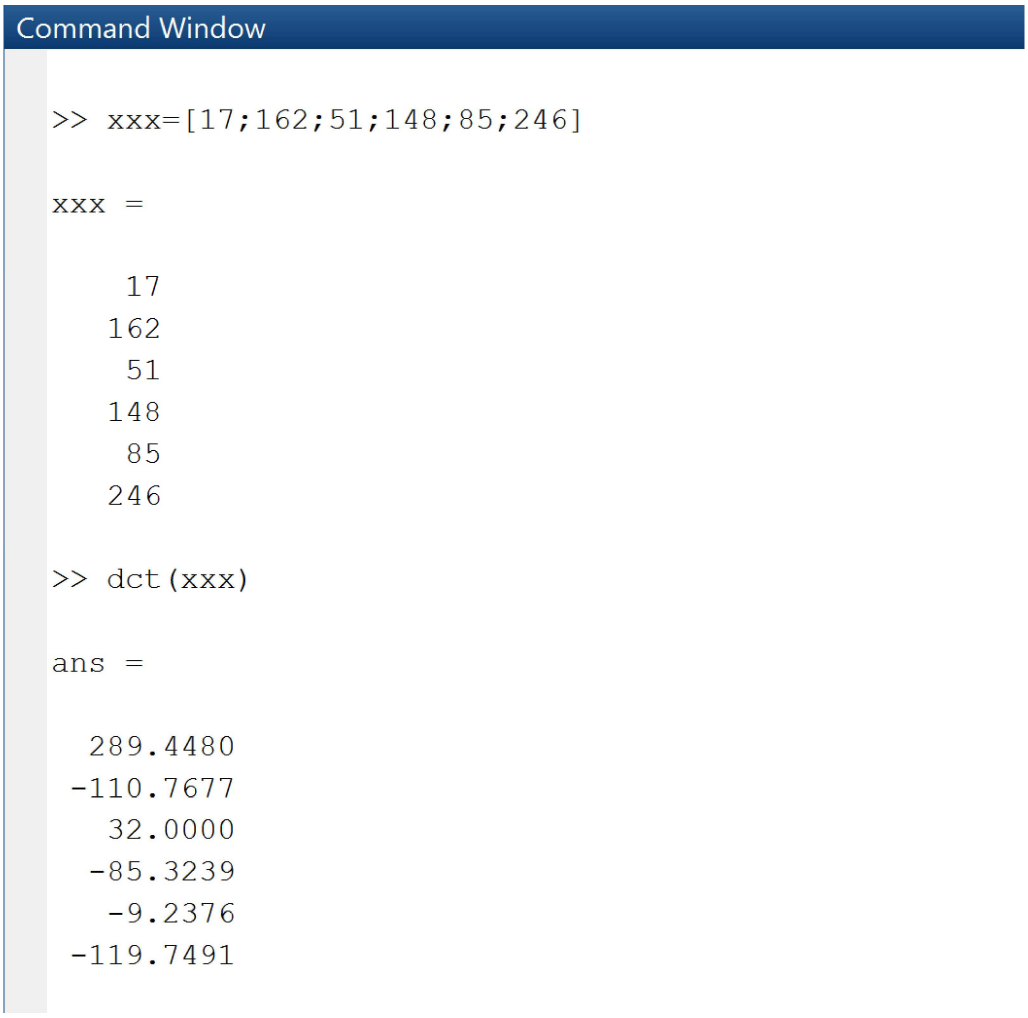
that stands for

$$\begin{gathered} X_0 = 0x0121700A \\ X_1 = 0xFF9138AD \\ X_2 = 0x001FFD05 \\ X_3 = 0xFFAAAA48 \\ X_4 = 0xFFF6C05B \\ X_5 = 0xFF883D6B \end{gathered}$$

and if we convert them in decimal by reserving 16 bits for the floating points, we get

$$\begin{gathered} X_0 = 289.437653 \\ X_1 = -110.778610 \\ X_2 = 31.988358 \\ X_3 = -85.334839 \\ X_4 = -9.248611 \\ X_5 = -119.760086 \end{gathered}$$

We checked this result by using the function *dct* on Matlab (Figure 3.4). We see that it differs from the Matlab result by a value in the order of 1/100. This depends on the precision used for the implementation, especially when converting the cosine values into binary.



```
>> xxx=[17;162;51;148;85;246]

xxx =

    17
   162
    51
   148
    85
   246

>> dct(xxx)

ans =

  289.4480
 -110.7677
   32.0000
  -85.3239
   -9.2376
 -119.7491
```

Figure 3.4: DCT result in MATLAB

3.3 Comparison

3.3.1 Systolic array architecture

A systolic array can be seen as a homogeneous network of funtional units (called also **nodes** or *Data Processing Units (DPU)*) which elaborate data and exchange them with their neighbours following a flow; in particular each FU receives the data from its upstream FU, elaborates it and passes it downstream. Each FU is indipendent from the others and executes only one operation over a huge amount of data; for this reason, systolic arrays are classified as SIMD (single instruction, multiple data) architectures. The particular name (”systolic”) derives from the fact that inside the array there is a continuous flow of data propagating from a FU to the next one resembling the flow of the blood in the human body. Usually, systolic arrays cannot be programmed and for this reason the circuit has to be designed $ad - hoc$. This kind of structure is made of simple FUs which usually implement a simple operation (addition, multiplication, ecc..) and that have a very small memory capability. The main purpose of each node is to compute a data and send it to the next FU following the flow. [11]

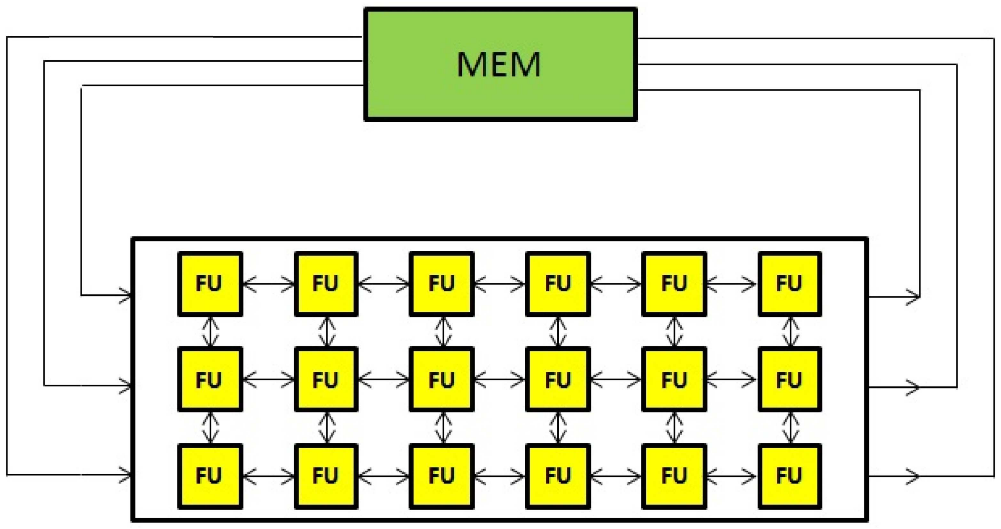


Figure 3.5: A generic figure of a systolic array; it is possible to notice the presence of parallel inputs and parallel outputs. Data are taken from the memory, eleborated along the multiple paths of FUs and then stored again in the memory.

3.3.2 Comparison with systolic array implementation

The DCT has been implemented in a systolic array architecture [11] with the following structure (Fig 3.6)

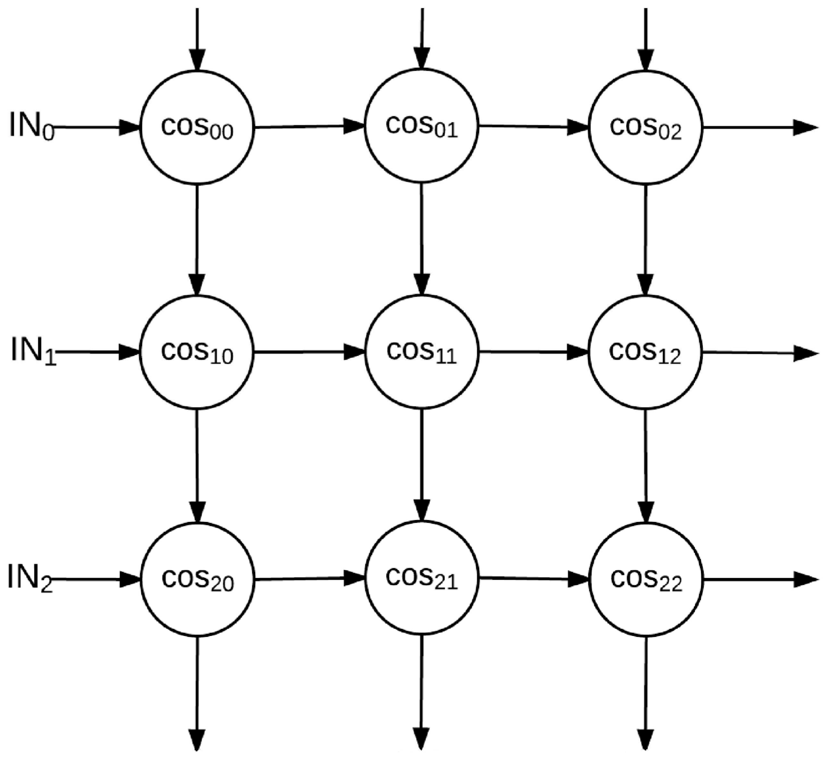


Figure 3.6: Implementation of DCT algorithm in a systolic array architecture

Given an input array (from the left), each element of this array will be multiplied by the values of the cosine matrix, previously stored in the internal register (Fig 3.7). At the output of each processing element (Fig 3.7), we have tha product $IN_i \cdot cos_{i,j}$ towards the bottom and the IN_i value toward the right.

The simulation's results of this implementation are shown in figure 3.8 with a check of the result in MATLAB (Fig. 3.9) As we can see from the simulation, for an array of size N, we need $2N - 1$ clock cycles to get the result. On the other hand, by looking to the structure of the systolic array, we can deduct the area to be $N \cdot N$ processing elements.

An important observation has to be done for the table 3.3.2. For the implementation not pipelined described in 3.1, the critical path is made of one multiplier and a

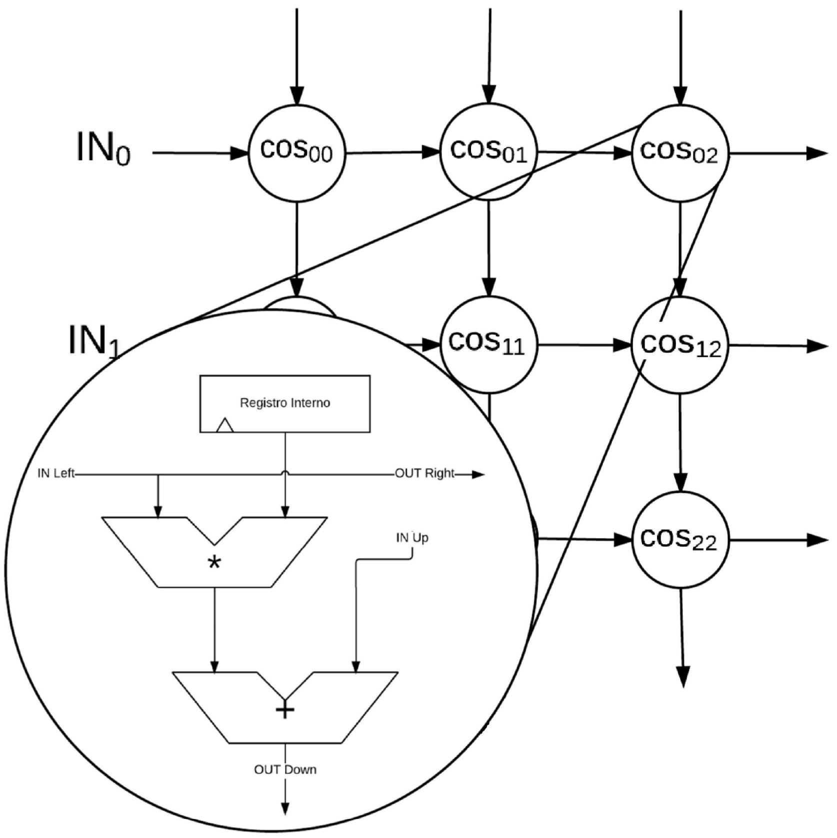


Figure 3.7: Processing element structure

cascade of $N-1$ adders, this is due because all the multiplication can be done in parallel, while the adders have some dependencies between them.

	Systolic array	This work	This work pipelined
Area	N^2 *moltiplications* N^2 *additions*	N^2 *moltiplications* N^2 *additions*	N *moltiplications* N *additions*
Time	$(2N-1)$ x (time for an addition and a multiplication)	$(N-1)$x(time for an addition) + time for a single multiplication	time for an addition and a multiplication. Also the synthesy's results 10.2 are concordant with this theory.

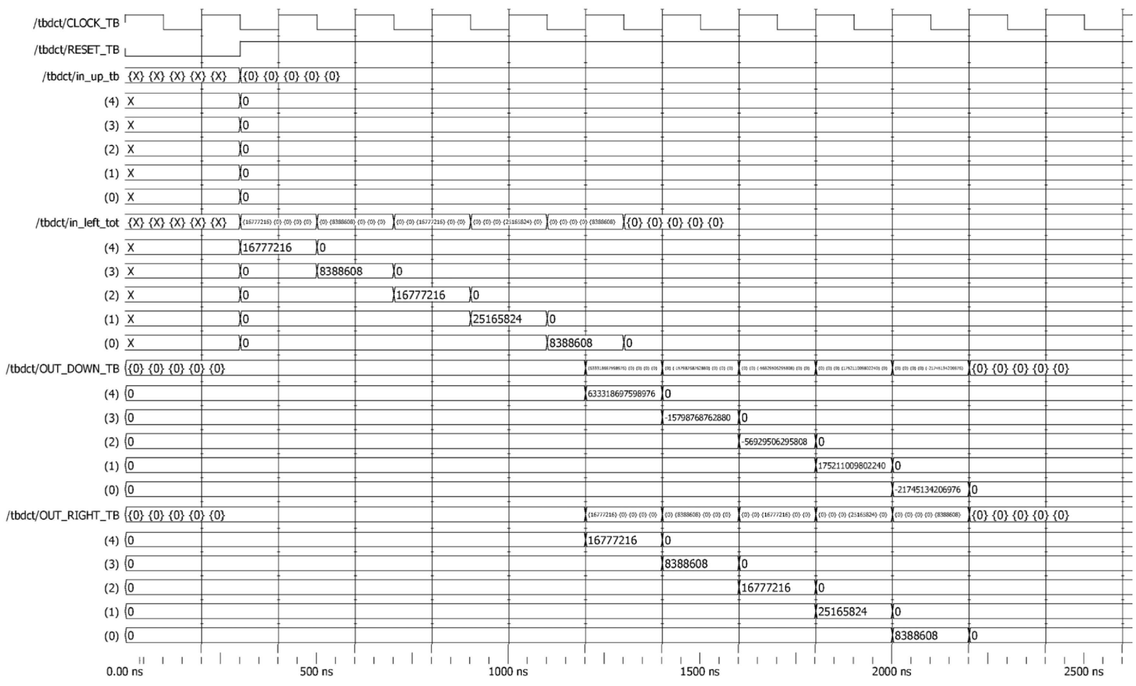
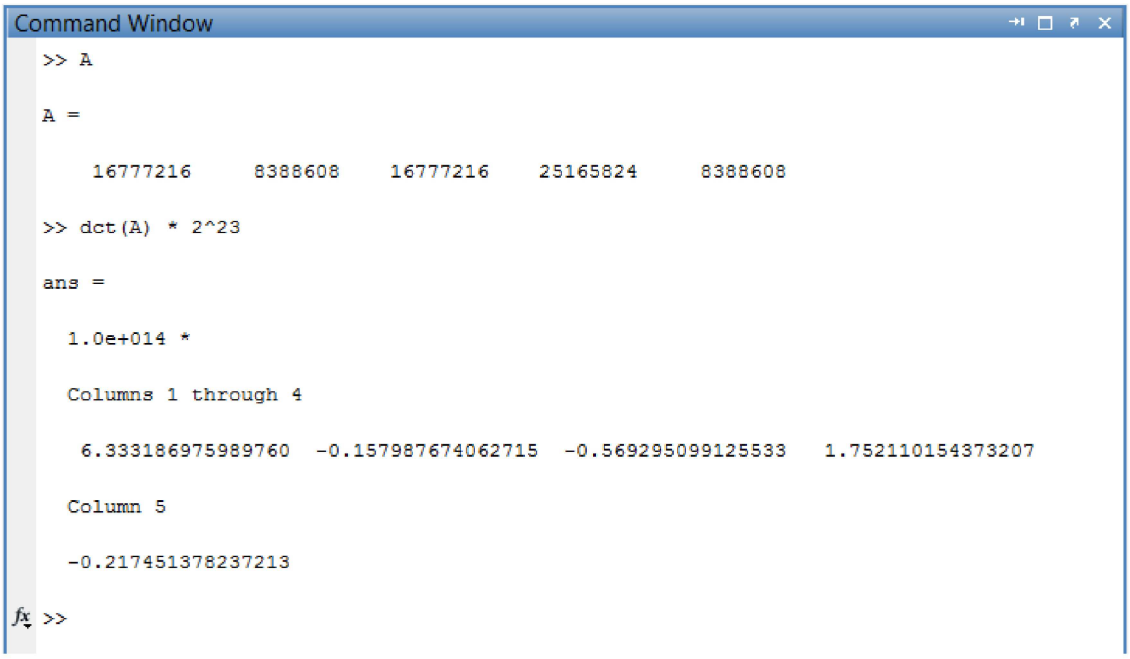


Figure 3.8: Results of the DCT in a systolic array architecture



```
>> A

A =

    16777216     8388608    16777216    25165824     8388608

>> dct(A) * 2^23

ans =

  1.0e+014 *

  Columns 1 through 4

   6.333186975989760  -0.157987674062715  -0.569295099125533   1.752110154373207

  Column 5

  -0.217451378237213

>>
```

Figure 3.9: Check results in MATLAB for systolic array architecture

3.4 Characteristics of DCT

PROCESSING ELEMENTS

1- Which kind of Processing element?

Adder, multiplier

2- Functionality

Addition, mulltiplication, summation

3- Complexity

Not pipelined	Area:	N^2 multiplier, $N \cdot (N-1)$ additions
	Time:	time for a multiplier followed by a chain of $N-1$ adder)
Pipelined	Area:	N multiplier, N additions
	Time:	time for a multiplication followed by an adder

4- Parallelism

Multiplications and summation are done in parallel.

5-Reconfigurability

No

6- Programmability

No

7- Need a dedicated memory?

If the architecture is pipelined, you need to store the partial value of the DCT. If not pipelined, no memory is required.

8- Relationship with I/O

INPUT: values of the cosine matrix, values of the input vector

OUTPUT: result of the DCT algorithm

MEMORY ELEMENTS

1- Need a clever memory LIM?

No, but can be implemented

2- Is there a data search algorithm?

No

3-Interface mechanism with other PE or memories

Communication required between local cells

4- Access mechanism

(No memory for this implementation)

5- Hierarchization

(No memory for this implementation)

6- Cache coherency

(No memory for this implementation)

7- Is it a a transactional memory?

(No memory for this implementation)

8- Are there virtualization (paging) mechanisms?

(No memory for this implementation)

ENCODING INFORMATION

1-Which encoding is used?

Binary encoding for the input vector

Binary encoding (by reserving some bits for the decimal part) for the cosine matrix and for the result

CONNECTIONS

1-Packet Exchange Protocol

Directly

2-Timing (asynchronou/synchronous)

Synchronous

3-Are there multiple instances?

Yes, except for the first row (no adders required)

4-Heterogeneity (Local/Distant I/O Connections)

Heterogeneous when storing the values of the DCT to the cells, and when exchanging the data

5-Are there any buffers?

There are pipeline registers.

3.5 LLM DCT

There's a faster DCT algorithm, sometimes called LLM for its authors: Loeffler, Ligtenberg, and Moschytz. [10] As said before (section 3), we use this formula for the computation of the Discrete Cosine Transformation.

$$X_k = \sum_{n=0}^{N-1} w(k) x_n cos\left[\frac{\pi}{N}(n+\frac{1}{2})k\right], \qquad k = 0,1,...,N-1 \tag{3.2}$$

$$w(x) = \begin{cases} \frac{1}{\sqrt{N}} & k = 0 \\ \sqrt{\frac{2}{N}} & 1 \leqslant k \leqslant N-1 \end{cases}$$

It has been found that for a certain kind of matrix, we can reduce the number of multiplications.
In this work I present the algorithm for a 8-point DCT [10].
First of all, let's have a look to the cosine matrix. Setting $\gamma(k) = cos(\frac{2\pi k}{32})$ the 8-point DCT matrix is

$$\frac{1}{2}\begin{bmatrix} \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) \\ \gamma(1) & \gamma(3) & \gamma(5) & \gamma(7) & -\gamma(7) & -\gamma(5) & -\gamma(3) & -\gamma(1) \\ \gamma(2) & \gamma(6) & -\gamma(6) & -\gamma(2) & -\gamma(2) & -\gamma(6) & \gamma(6) & \gamma(2) \\ \gamma(3) & -\gamma(7) & -\gamma(1) & -\gamma(5) & \gamma(5) & \gamma(1) & \gamma(7) & -\gamma(3) \\ \gamma(4) & -\gamma(4) & -\gamma(4) & \gamma(4) & \gamma(4) & -\gamma(4) & -\gamma(4) & \gamma(4) \\ \gamma(5) & -\gamma(1) & \gamma(7) & \gamma(3) & -\gamma(3) & -\gamma(7) & \gamma(1) & -\gamma(5) \\ \gamma(6) & -\gamma(2) & \gamma(2) & -\gamma(6) & -\gamma(6) & \gamma(2) & -\gamma(2) & \gamma(6) \\ \gamma(7) & -\gamma(5) & \gamma(3) & -\gamma(1) & \gamma(1) & -\gamma(3) & \gamma(5) & -\gamma(7) \end{bmatrix}$$

We know that the DCT can be seen as a product of matrices:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) \\ \gamma(1) & \gamma(3) & \gamma(5) & \gamma(7) & -\gamma(7) & -\gamma(5) & -\gamma(3) & -\gamma(1) \\ \gamma(2) & \gamma(6) & -\gamma(6) & -\gamma(2) & -\gamma(2) & -\gamma(6) & \gamma(6) & \gamma(2) \\ \gamma(3) & -\gamma(7) & -\gamma(1) & -\gamma(5) & \gamma(5) & \gamma(1) & \gamma(7) & -\gamma(3) \\ \gamma(4) & -\gamma(4) & -\gamma(4) & \gamma(4) & \gamma(4) & -\gamma(4) & -\gamma(4) & \gamma(4) \\ \gamma(5) & -\gamma(1) & \gamma(7) & \gamma(3) & -\gamma(3) & -\gamma(7) & \gamma(1) & -\gamma(5) \\ \gamma(6) & -\gamma(2) & \gamma(2) & -\gamma(6) & -\gamma(6) & \gamma(2) & -\gamma(2) & \gamma(6) \\ \gamma(7) & -\gamma(5) & \gamma(3) & -\gamma(1) & \gamma(1) & -\gamma(3) & \gamma(5) & -\gamma(7) \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}$$

More explicitly

$$\begin{aligned} X_0 &= (x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7) \cdot \gamma(4) \\ X_1 &= (x_0 - x_7) \cdot \gamma(1) + (x_1 - x_6) \cdot \gamma(3) + (x_2 - x_5) \cdot \gamma(5) + (x_3 - x_4) \cdot \gamma(7) \\ X_2 &= [(x_0 + x_7) - (x_3 + x_4)] \cdot \gamma(2) + [(x_1 + x_6) - (x_2 + x_5)] \cdot \gamma(6) \\ X_3 &= (x_0 - x_7) \cdot \gamma(3) + (x_6 - x_1) \cdot \gamma(7) + (x_5 - x_2) \cdot \gamma(1) + (x_4 - x_3) \cdot \gamma(5) \\ X_4 &= (x_0 + x_7 + x_3 + x_4) \cdot \gamma(4) - (x_5 + x_6 + x_1 + x_2) \cdot \ \gamma(4) \\ X_5 &= (x_0 - x_7) \cdot \gamma(5) + (x_6 - x_1) \cdot \gamma(1) + (x_2 - x_5) \cdot \gamma(7) + (x_3 - x_4) \cdot \gamma(3) \\ X_6 &= [(x_0 + x_7) - (x_3 + x_4)] \cdot \gamma(6) + [(x_2 + x_5) - (x_1 + x_6)] \cdot \gamma(2) \\ X_7 &= (x_0 - x_7) \cdot \gamma(7) + (x_6 - x_1) \cdot \gamma(5) + (x_2 - x_5) \cdot \gamma(3) + (x_4 - x_3) \cdot \gamma(1) \end{aligned}$$

we can see that we can do some addition in parallel and some of them in series due to the data dependencies.
Moreover, we can reduce the number of multiplications. For instance for X_0 we need only one multiplication, and for X_4 only 2.

3.5.1 Architecture of the LLM DCT

The architecture is shown in fig 3.10

The stages of the LLM Algorithm for an 8-point DCT, numbered 1 to 4, are parts that have to be executed in series and can not be evaluated in parallel because of data dependencies. However calculations inside one stage can be parallelized.
Stage 1 consists of 8 additions/subtractions.
In Stage 2 , the algorithm splits into two parts. One part is for even coefficients (only additions and subtractions) and the second part is for odd coefficients (rotations).

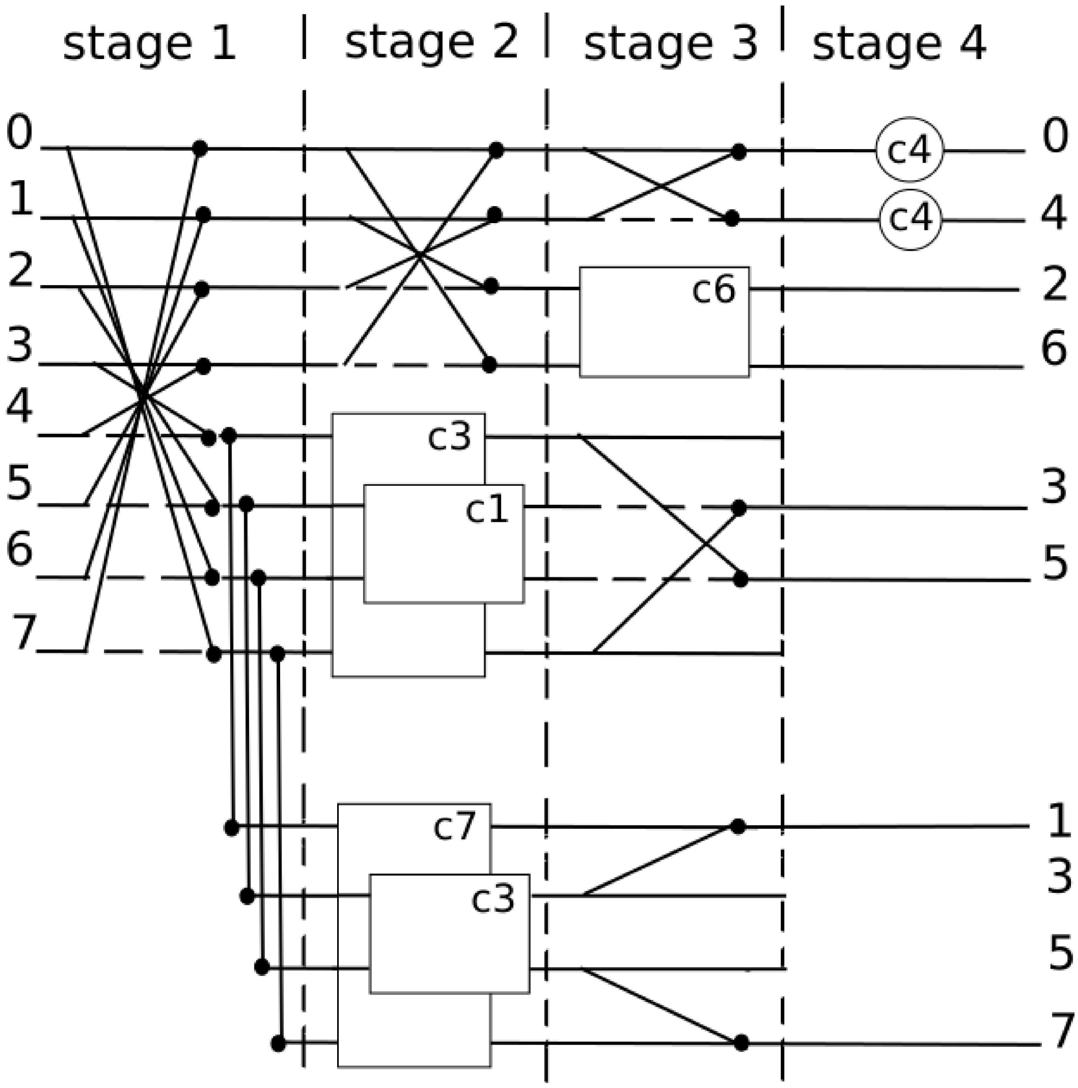
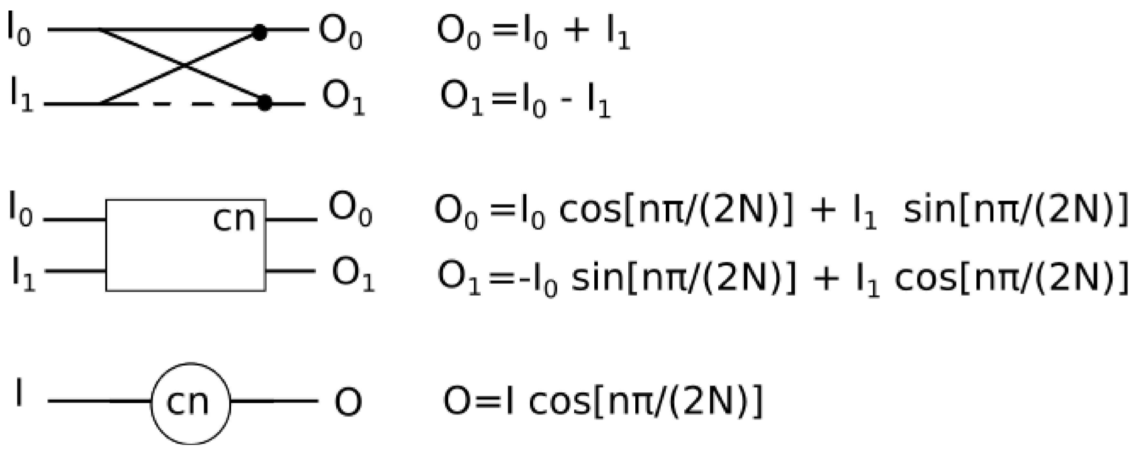


Figure 3.10: LLM DCT for an 8-point with 17 multiplications. For symbols, see figure 3.11

The even part is nothing else than a 4-point DCT, again separating in even and odd part in stage 3.
Figure 3.11 explains the building blocks of the algorithm. The second building block, the rotation, can be calculated using only 3 multiplications and 3 additions instead of 4 multiplications and 2 additions using the equivalence shown in equation 3.3. Since $a = cos(\frac{n\pi}{2N})$ and $b = sin(\frac{n\pi}{2N})$ are constants and known *a priori*, also the values $(b - a)$ and $-(b + a)$ are constants, so we do not need an hardware component to

compute them.

$$\begin{aligned} y_0 &= a \cdot x_0 + b \cdot x_1 = (b-a) \cdot x_l + a \cdot (x_0 + x_l) \\ y_0 &= -b \cdot x_0 + a \cdot x_1 = -(b+a) \cdot x_0 + a \cdot (x_0 + x_l) \\ a &= cos(\frac{n\pi}{2N}) \\ b &= sin(\frac{n\pi}{2N}) \end{aligned} \tag{3.3}$$



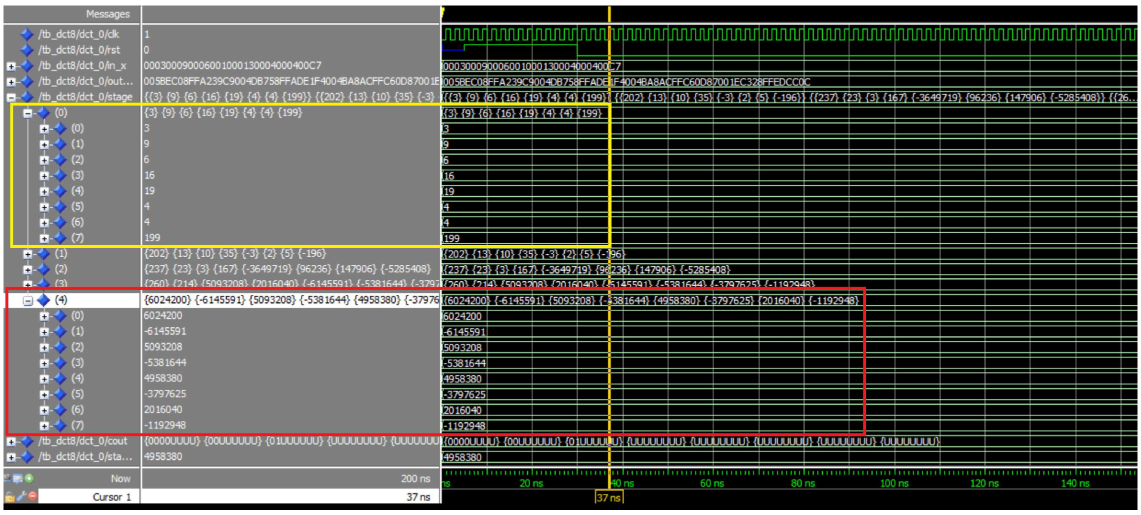
I_0
I_1
O_0
O_1
$O_0 = I_0 + I_1$
$O_1 = I_0 - I_1$
cn
$O_0 = I_0 \cos[n\pi/(2N)] + I_1 \sin[n\pi/(2N)]$
$O_1 = -I_0 \sin[n\pi/(2N)] + I_1 \cos[n\pi/(2N)]$
I
O
$O = I \cos[n\pi/(2N)]$

Figure 3.11: Symbols used to describe the LLM DCT for an 8-point with 17 multiplications (fig.3.10)

3.5.2 Simulation and test

I performed some tests of this implementation with different values.
In the fig.3.12 we can see the results of an array of 8 elements using the LLM structure.
In the yellow box, we can see the input vector to be

$$x_0 = 3$$
$$x_1 = 9$$
$$x_2 = 6$$



```
Messages
/tb_dct8/dct_0/clk      1
/tb_dct8/dct_0/rst      0
/tb_dct8/dct_0/in_x     00030009000600100013000400040 0C7
/tb_dct8/dct_0/out...   005BEC08FFA239C9004DB758FFADE1F4004BA8ACFFC60D87001E
/tb_dct8/dct_0/stage    {{3} {9} {6} {16} {19} {4} {4} {199}} {{202} {13} {10} {35} {-3}
  (0)                   {3} {9} {6} {16} {19} {4} {4} {199}
    (0)                 3
    (1)                 9
    (2)                 6
    (3)                 16
    (4)                 19
    (5)                 4
    (6)                 4
    (7)                 199
  (1)                   {202} {13} {10} {35} {-3} {2} {5} {-196}
  (2)                   {237} {23} {3} {167} {-3649719} {96236} {147906} {-5285408}
  (3)                   {260} {214} {5093208} {2016040} {-6145591} {-5381644} {-3797
  (4)                   {6024200} {-6145591} {5093208} {-5381644} {4958380} {-37976
    (0)                 6024200
    (1)                 -6145591
    (2)                 5093208
    (3)                 -5381644
    (4)                 4958380
    (5)                 -3797625
    (6)                 2016040
    (7)                 -1192948
/tb_dct8/dct_0/cout     {0000UUUU} {00UUUUUU} {01UUUUUU} {UUUUUUUU} {UUUUUUU
/tb_dct8/dct_0/sta...   4958380
Now                     200 ns
Cursor 1                37 ns
```

Figure 3.12: LLM 8-point Discrete cosine transform-Results of the simulation

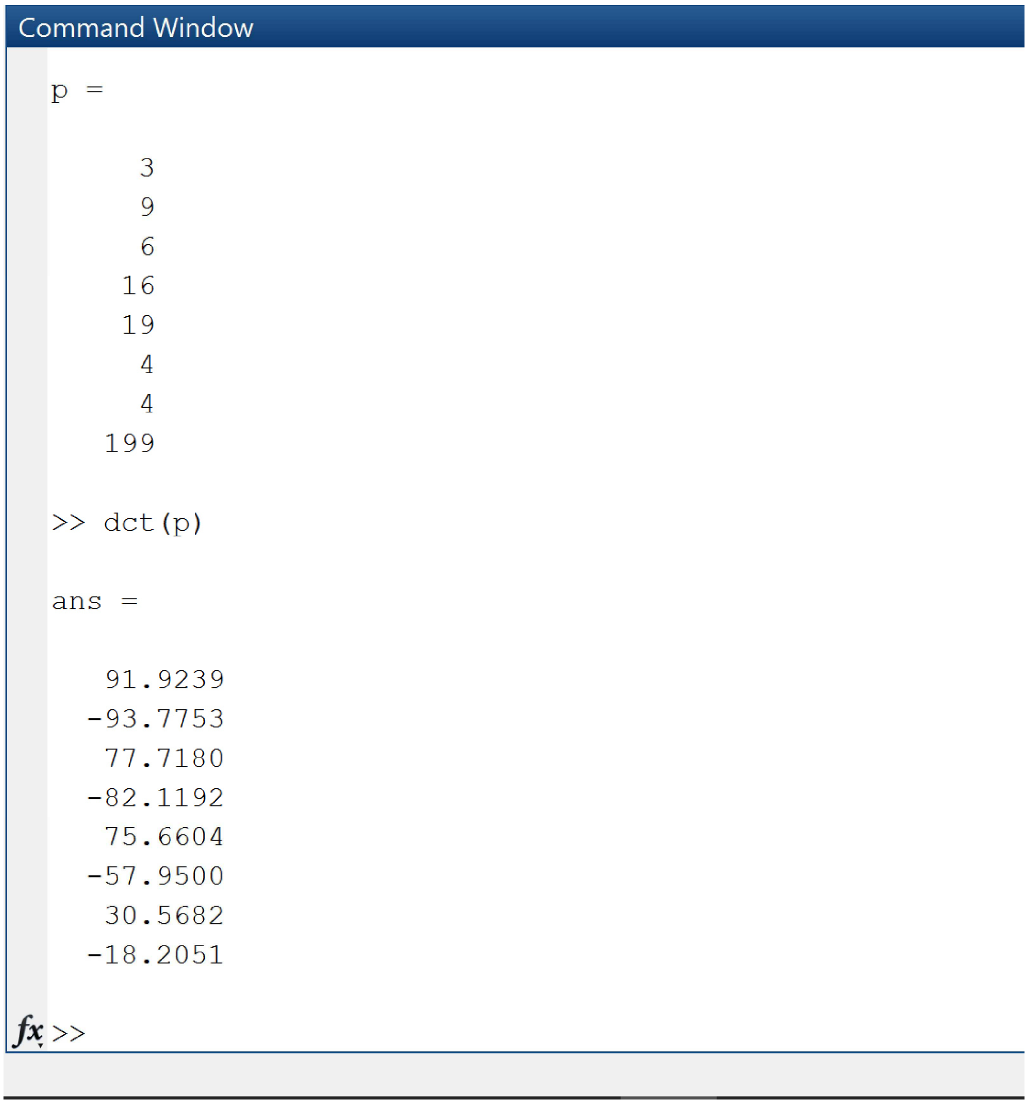
$$
\begin{aligned}
x_3 &= 16\\
x_4 &= 19\\
x_5 &= 4\\
x_6 &= 4\\
x_7 &= 199
\end{aligned}
$$

Between the yellow box and the red box, we can see the results of the different stages.

The final result is shown in the red box. Since we reserved 16 bits for the decimal part, these vector has to be shifted by 16 position, i.e. divided by $2^{16} = 65536$.

$$
\begin{aligned}
X_0 &= 6024200/2^{16} = 91.9219\\
X_1 &= -6145591/2^{16} = -93.7742\\
X_2 &= 5093208/2^{16} = 77.7161\\
X_3 &= -5381644/2^{16} = -82.1173\\
X_4 &= 4958380/2^{16} = 75.65887\\
X_5 &= -3797625/2^{16} = -57.9471\\
X_6 &= 2016040/2^{16} = 30.7623\\
X_7 &= -1192948/2^{16} = -18.2029
\end{aligned}
$$

We checked this result by using the function *dct* on Matlab (Figure 3.13). We see that it differs from the Matlab result by a value in the order of 1/100. This depends on the precision used for the implementation, especially when converting the cosine values into binary.



```
p =

     3
     9
     6
    16
    19
     4
     4
   199

>> dct(p)

ans =

   91.9239
  -93.7753
   77.7180
  -82.1192
   75.6604
  -57.9500
   30.5682
  -18.2051

fx >>
```

Figure 3.13: LLM 8-point DCT result in MATLAB

3.5.3 Comparison

Since this architecture is fon an 8-point DCT, I will compare it with the other structure for the DCT shown before (subsection 3.1 and 3.3.2).

N=8	Systolic array	This work	This work pipelined	LLM	LLM pipelined
Area	8^2 *multiplications* 8^2 *additions*	8^2 *moltiplications* 8^2*additions*	8 *moltiplications* 8*additions*	17 *moltiplications* 33*additions*	4.25 *moltiplications* 8.25*additions*
Time	$(2 \cdot 8 - 1)$ x *(time for a multiplication followed by an addition)*	8 x *(time for a multiplication followed by an addition)*	*time for a multiplication followed by an addition*	*time for 3 multiplications and 6 addition*	*time for 1 multiplication and 2 additions*

We can see that the LLM (both pipelined and not) is the best solutions in terms of area.
In terms of time, if we look to the pipelined LLM architecture, we have to consider the longest critical path between the four stages. This is found in the rotation block, which takes the time for a multiplication and 2 additions. The Data Flow Graph is shown in fig. 3.14. So looking to the pipelined architectures of the LLM and the one shown in this work (subsection 3.1), it's better the one i presented above.
If we look to the architecture without any pipeline register, the critical path will be 1 multiplication and 4 additions. To find this value, we can consider the critical path for each output element, by taking into account that a rotation block take the time for one multiplication and 2 additions. A scheme is shown in fig. 3.15. In conclusion, without any pipeline registers, it's better the LLM architecture in terms of time.

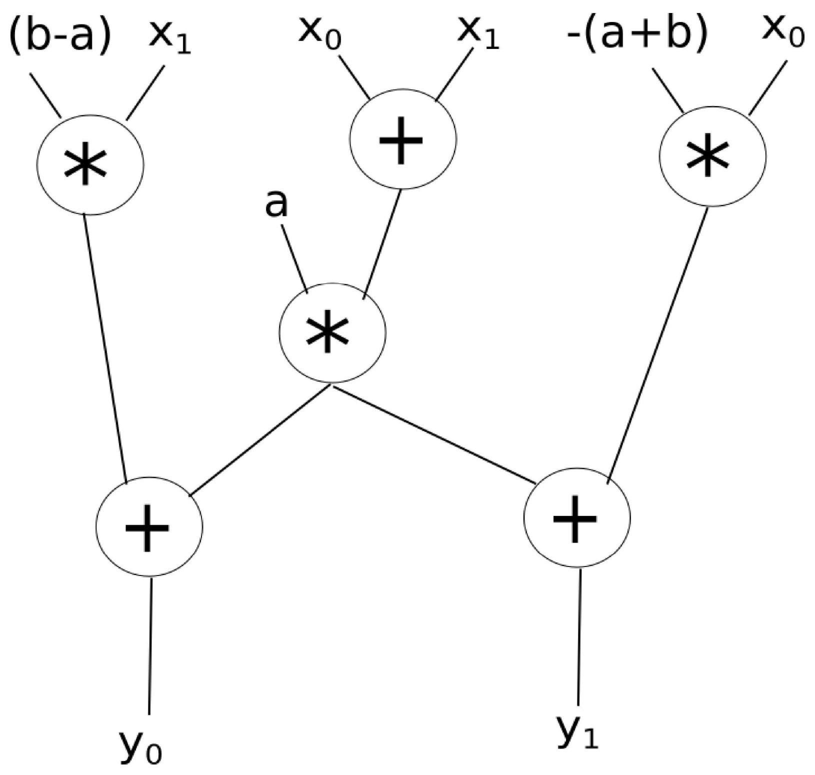


Figure 3.14: DFG for a rotation block

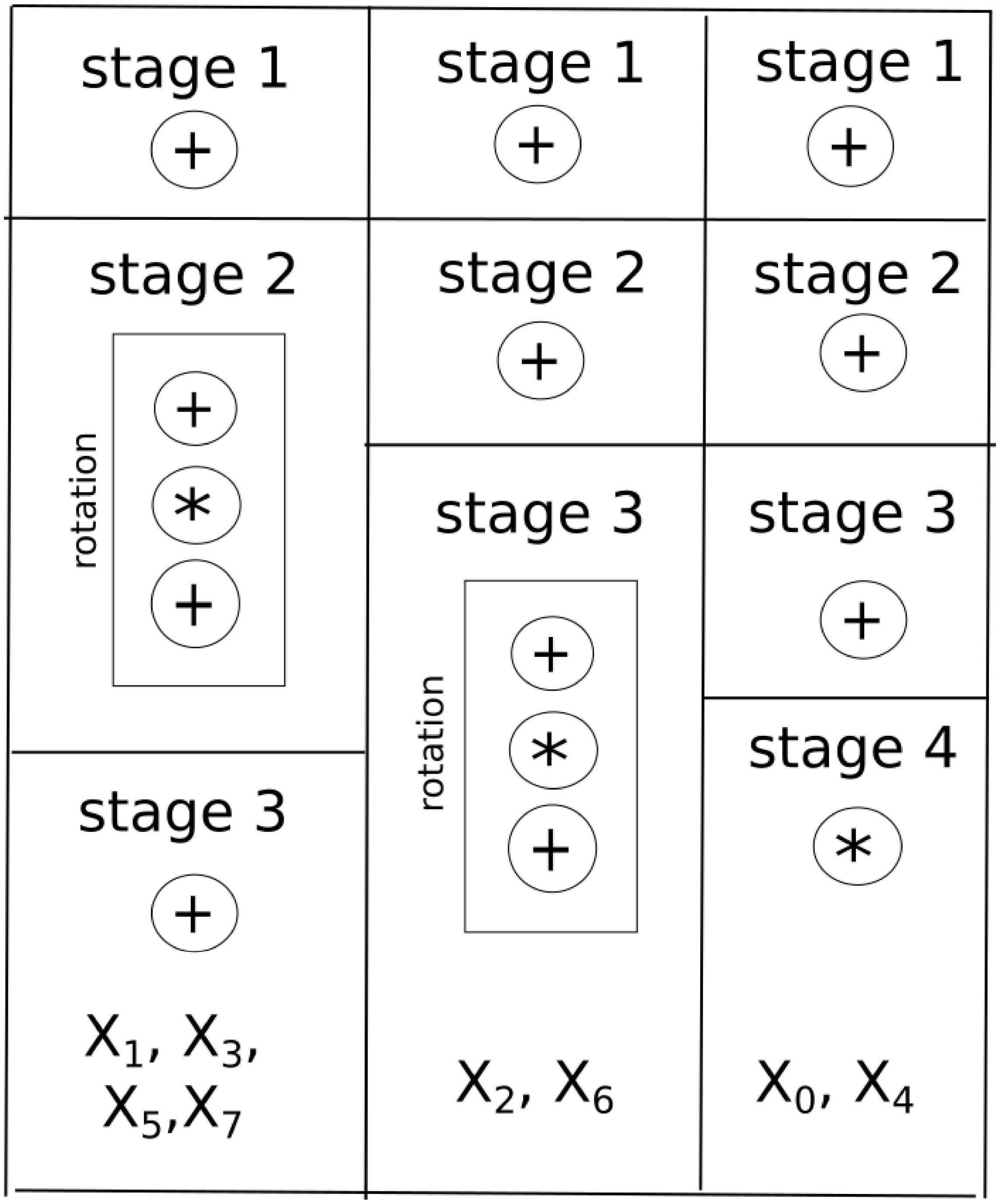
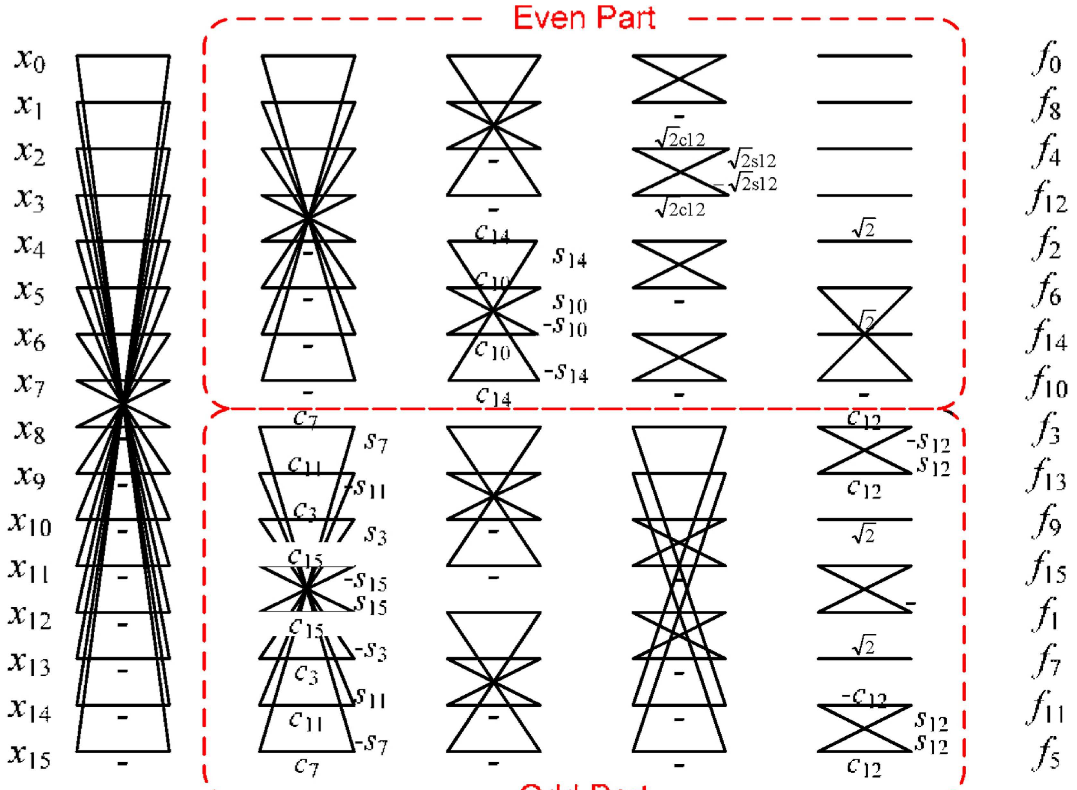


Figure 3.15: Critical path for each output element

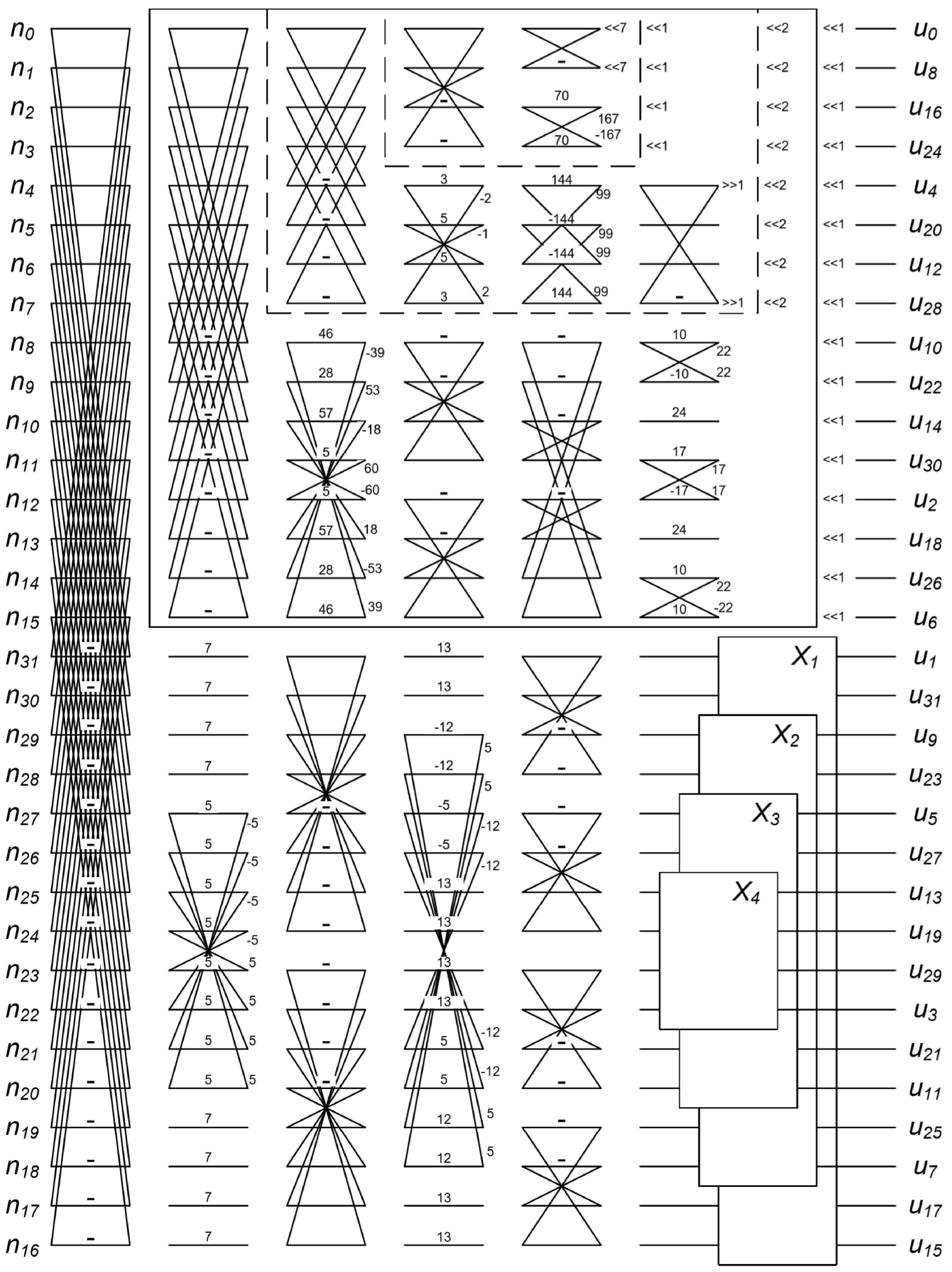
3.5.4 High Order LLM DCT

Many studies has been done for the LLM architecture, and it has been theorized that the LLM can be applied for the DCT that has a size of 2^n. [12]
Although we can see a recursive approach for the even parts at top, such as an addition/substraction, nothing can be said for the odd parts. A brute-force search is required. [13]
Fig. 3.16 and fig. 3.17 show an implementation for the LLM of order-16 and order-32 respectively.



$$c_k = \cos\left(\frac{k\pi}{32}\right), \quad s_k = \sin\left(\frac{k\pi}{32}\right).$$

Figure 3.16: Order-16 LLM fast DCT algorithm



n0 n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n31 n30 n29 n28 n27 n26 n25 n24 n23 n22 n21 n20 n19 n18 n17 n16
u0 u8 u16 u24 u4 u20 u12 u28 u10 u22 u14 u30 u2 u18 u26 u6 u1 u31 u9 u23 u5 u27 u13 u19 u29 u3 u21 u11 u25 u7 u17 u15
X1
X2
X3
X4
<<7
<<1
<<2
>>1

Figure 3.17: Fast algorithm for the forward order-32 integer transform with the fast algorithms for the forward order-4, order-8, and order-16 integer transforms embedded.

3.5.5 Characteristics of the LLM DCT 8-point

PROCESSING ELEMENTS

1- Which kind of Processing element?

Adder, multiplier, substractor

2- Functionality

Addition, mulltiplication, substraction, summation

3- Complexity

Not pipelined	Area:	17 multiplier, 33 additions
	Time:	8x(time for 3 multipliers and 6 adders)
Pipelined	Area:	4.25 multiplier, 8.25 additions
	Time:	time for a multiplication and by 2 additions

4- Parallelism

Each operation belonging in a certain phase, can be done in parallel. (Fig. 3.10)

5-Reconfigurability

No

6- Programmability

No

7- Need a dedicated memory?

Yes, it needs to store the cosine values.

If not pipelined, no memory is required.

8- Relationship with I/O

INPUT: values of the input vector

OUTPUT: result of the DCT algorithm

MEMORY ELEMENTS

1- Need a clever memory LIM?

No

2- Is there a data search algorithm?

No

3-Interface mechanism with other PE or memories

Communication required between local cells

4- Access mechanism

(No memory for this implementation)

5- Hierarchization

(No memory for this implementation)

6- Cache coherency

(No memory for this implementation)

7- Is it a a transactional memory?

(No memory for this implementation)

8- Are there virtualization (paging) mechanisms?

(No memory for this implementation)

ENCODING INFORMATION

1-Which encoding is used?

Binary encoding (by reserving some bits for the decimal part) for the cosine matrix and for the result

CONNECTIONS

1-Packet Exchange Protocol

Directly

2-Timing (asynchronou/synchronous)

Synchronous

3-Are there multiple instances?

Yes, but the architecture is not simmetric (3.10)

4-Heterogeneity (Local/Distant I/O Connections)

Heterogeneous when storing the values of the input vector.

5-Are there any buffers?

There are pipeline registers.

Chapter 4

Binomial Filter

Binomial filters are simple and efficient structures based on the binomial coefficients for implementing Gaussian filtering. To extract these coefficients, we could use Pascal's Triangle.
The rows of Pascal's triangle are conventionally enumerated starting with row $n = 0$ at the top (the 0t̂h row). The entries in each row are numbered from the left beginning with $k = 0$ and are usually staggered relative to the numbers in the adjacent rows. The triangle may be constructed in the following manner: In row 0 (the topmost row), there is a unique nonzero entry1. Each entry of each subsequent row is constructed by adding the number above and to the left with the number above and to the right, treating blank entries as 0. For example, the initial number in the first (or any other) row is 1(the sum of 0 and 1), whereas the numbers 1 and 3 in the third row are added to produce the number 4 in the fourth row.

$$\begin{array}{lccccccccccccc}
n=0 & & & & & & & 1 & & & & & & \\
n=1 & & & & & & 1 & & 1 & & & & & \\
n=2 & & & & & 1 & & 2 & & 1 & & & & \\
n=3 & & & & 1 & & 3 & & 3 & & 1 & & & \\
n=4 & & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
n=5 & & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
n=6 & 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}$$

The first odd sized serie is:

$$1 \quad 2 \quad 1$$

this form the basis for the 3x3 binomial filter.
The weights of the binomial filter are biggest at the center and taper down towards

the outer areas of the neighborhood.
To create the 2D low pass binomial filter, we have to form the outer product of the row with its corresponding column. In simple terms take the row and multiply each element in it by the first value in the row. Then take the row and multiply each element in it by the second value in the row. Repeat this until we have multiplied the row by every element in the row. Then stack each of these resulting rows to make a square table or matrix. [14]
For example, the 3x3 matrix is generated as

$$\begin{bmatrix}1\\2\\1\end{bmatrix} \cdot \begin{bmatrix}1 & 2 & 1\end{bmatrix} = \begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix}$$

After multipling, we need to normalize, so the overall 3x3 binomial filter is:

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix}$$

In particular, given a cell in position (x,y), the final value of the computation should be:

$$\begin{aligned} f(x,y) = \frac{1}{16}[&1(x-1,y-1) + 2(x-1,y) + 1(x-1,y+1)+\\ &+ 2(x,y-1) + 4(x,y) + 2(x,y+1)+\\ &+ 1(x+1,y-1) + 2(x+1,y) + 1(x+1,y+1)] \end{aligned} \tag{4.1}$$

As shown in the figure below, the value of the dark blue rectangle can be calculated as the weighted mean of all the light blue rectangles:

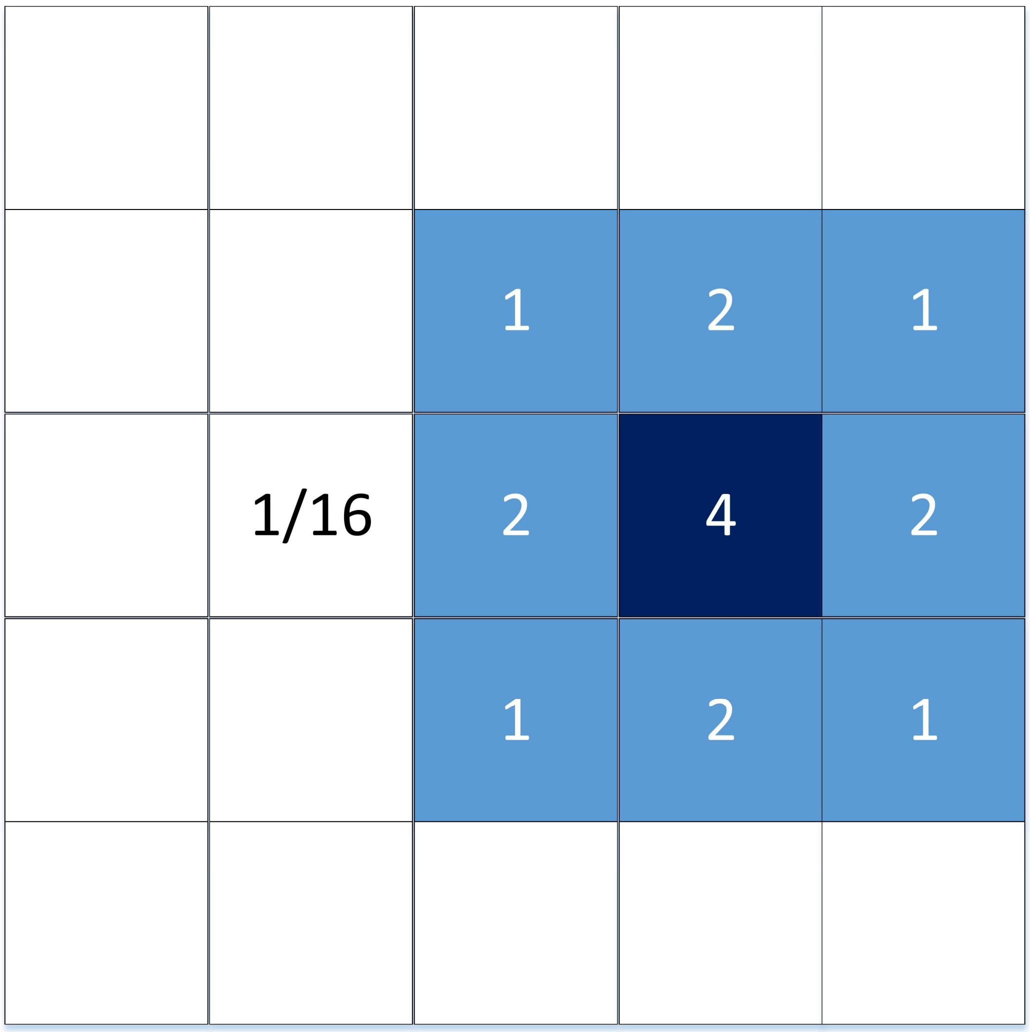


Figure 4.1: Binomial Filter for a generic cell

4.1 Hardware implementations

4.1.1 Hardware implementation type 1

This implementation will focus only to the possible cells for the binomial filter, i.e. the ones not in the border of the matrix, bevause they are leaved unchanged.

1. for each row, whenever we find three consecutive cells, we evaluate the sum

   $$S_{x,y} = cell_{x-1,y} + 2 \cdot cell_{x,y} + cell_{x+1,y}$$

   In order to so, we need 2 adder: we shift by one position the value of the central cell (i.e. multiply by two) and sum it with the left and the right cells as shown in fig 4.2 while fig 4.3 shows the overall picture of this step.

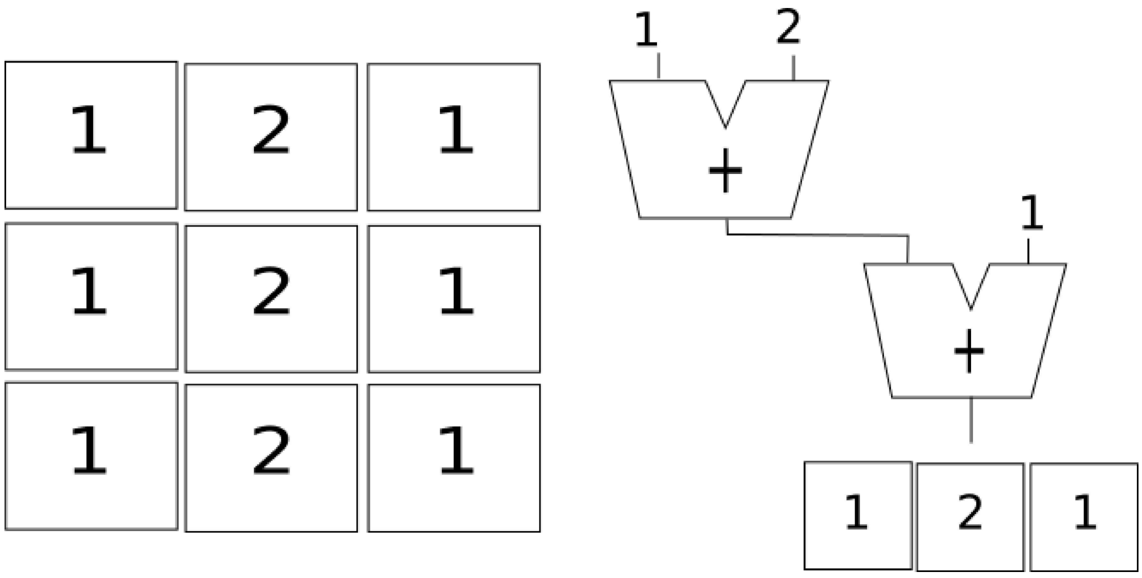
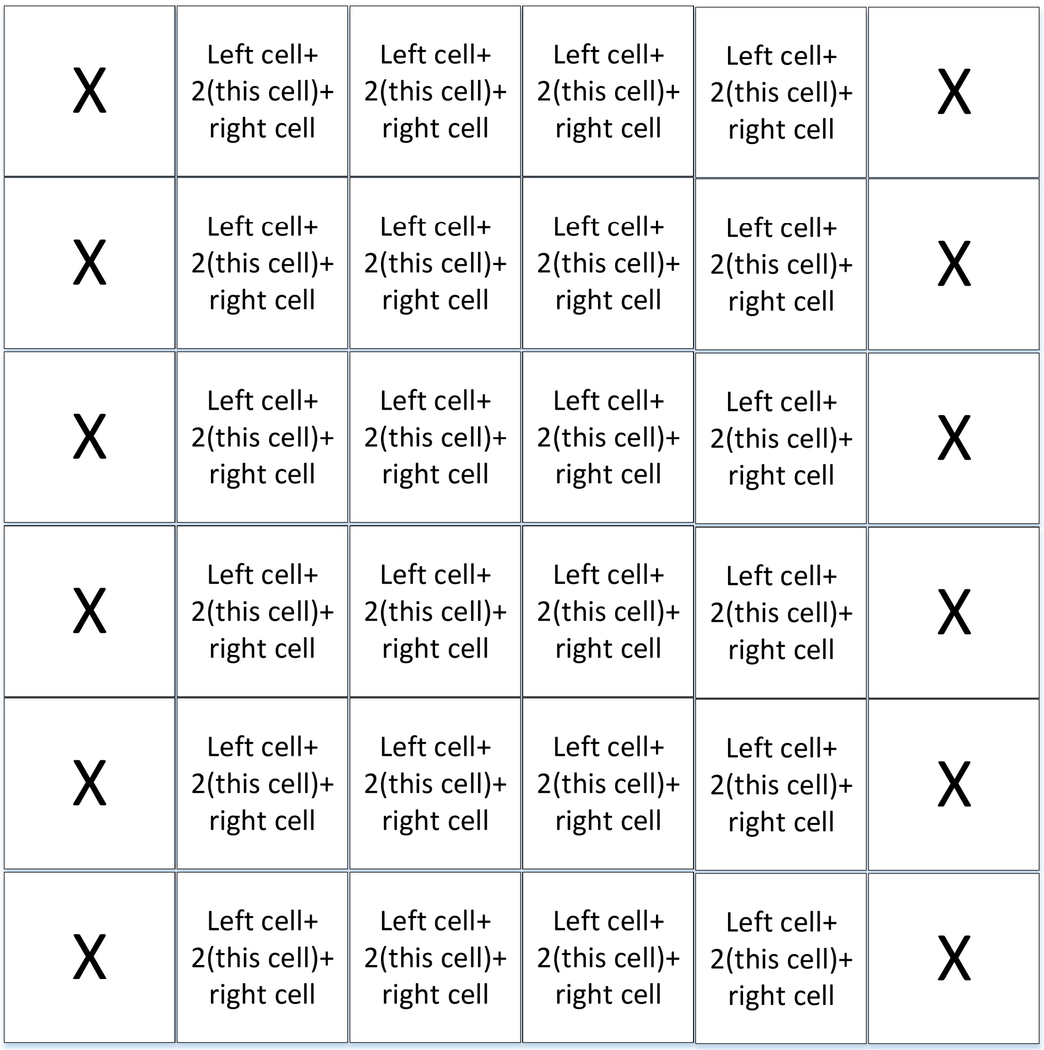


Figure 4.2: Weighted sum for three consecutive cells



X	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	X
X	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	X
X	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	X
X	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	X
X	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	X
X	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	Left cell+ 2(this cell)+ right cell	X

Figure 4.3: First part of the binomial filter HW implementation

2. Taking the result of the previous step as a matrix, for each column whenever we found three consecutive cells we evaluate the sum

$$SUM_{x,y} = S_{x,y-1} + 2 \cdot S_{x,y} + S_{x,y+1}$$

which is equal to

$$\begin{aligned} SUM_{x,y} =& (cell_{x-1,y-1} + 2 \cdot cell_{x,y-1} + cell_{x+1,y-1}) + \\ & + 2 \cdot (cell_{x-1,y} + 2 \cdot cell_{x,y} + cell_{x+1,y}) + \\ & + (cell_{x-1,y+1} + 2 \cdot cell_{x,y+1} + cell_{x+1,y+1}) = \\ =& cell_{x-1,y-1} + 2 \cdot cell_{x,y-1} + cell_{x+1,y-1} + \\ & + 2 \cdot cell_{x-1,y} + 4 \cdot cell_{x,y} + 2 \cdot cell_{x+1,y} + \\ & + cell_{x-1,y+1} + 2 \cdot cell_{x,y+1} + cell_{x+1,y+1} \end{aligned} \tag{4.2}$$

fig 4.4 shows this step

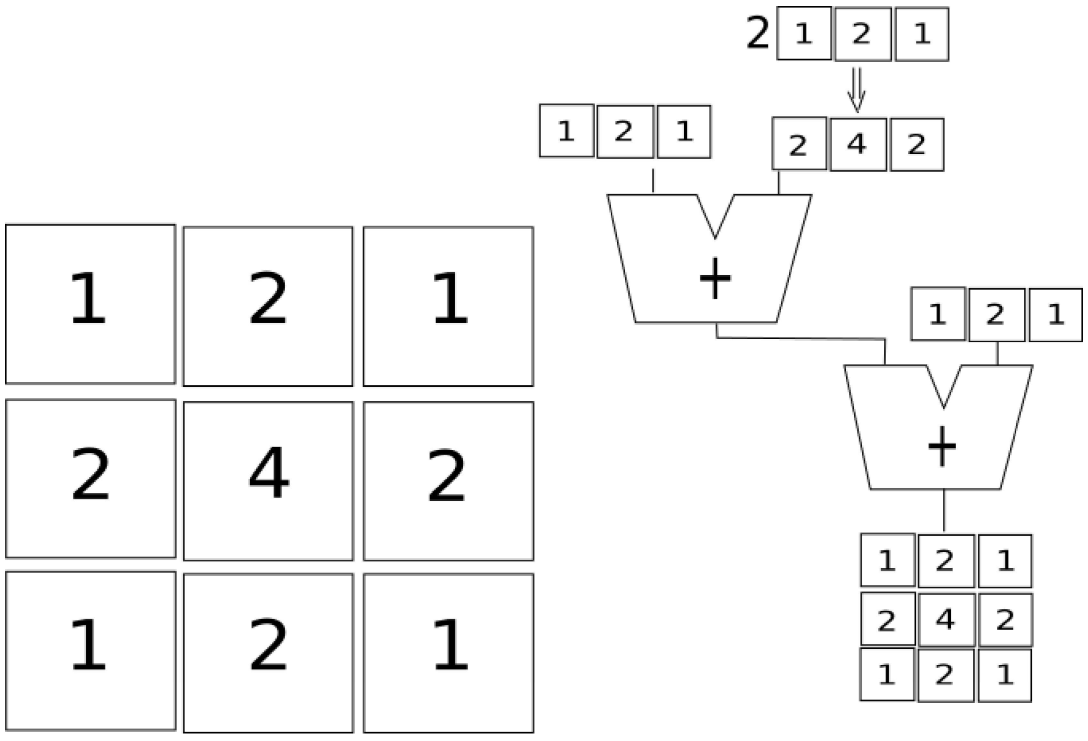


Figure 4.4: Weighted sum for 3x3 cells

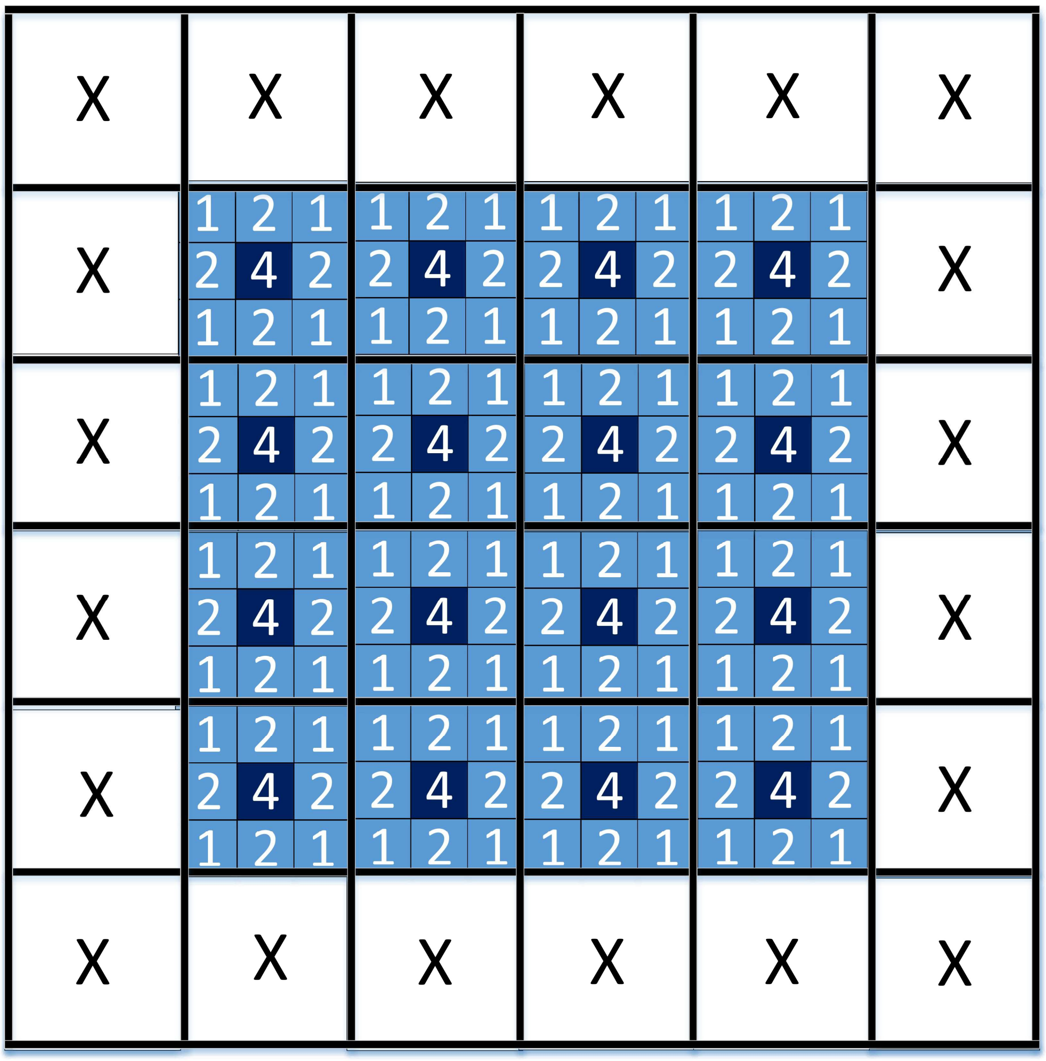


Figure 4.5: Weighted sum for all 3x3 cells

3. Now that we have the weighted sum for all the 3x3 cells, we have to divided by 16. In order to do so, the quickest way is to shift the result by 4 bits. The values of the cells in the border of the matrix, are unchanged in the final result.

4.1.2 Hardware implementation type 2

This implementation will focus to all the cells. The only difference between this implementation and the one described in the subsection 4.1.1 is that now we apply the Binomial Filter also for the cells in the border of the matrix. We take the original matrix and we set a neighborhood of cells at $'0'$ as shown in fig **??** where the original matrix has benn highlighted in black. The procedure to evaluate the final result of the binomial filter is the same as explained in the subsection 4.1.1.

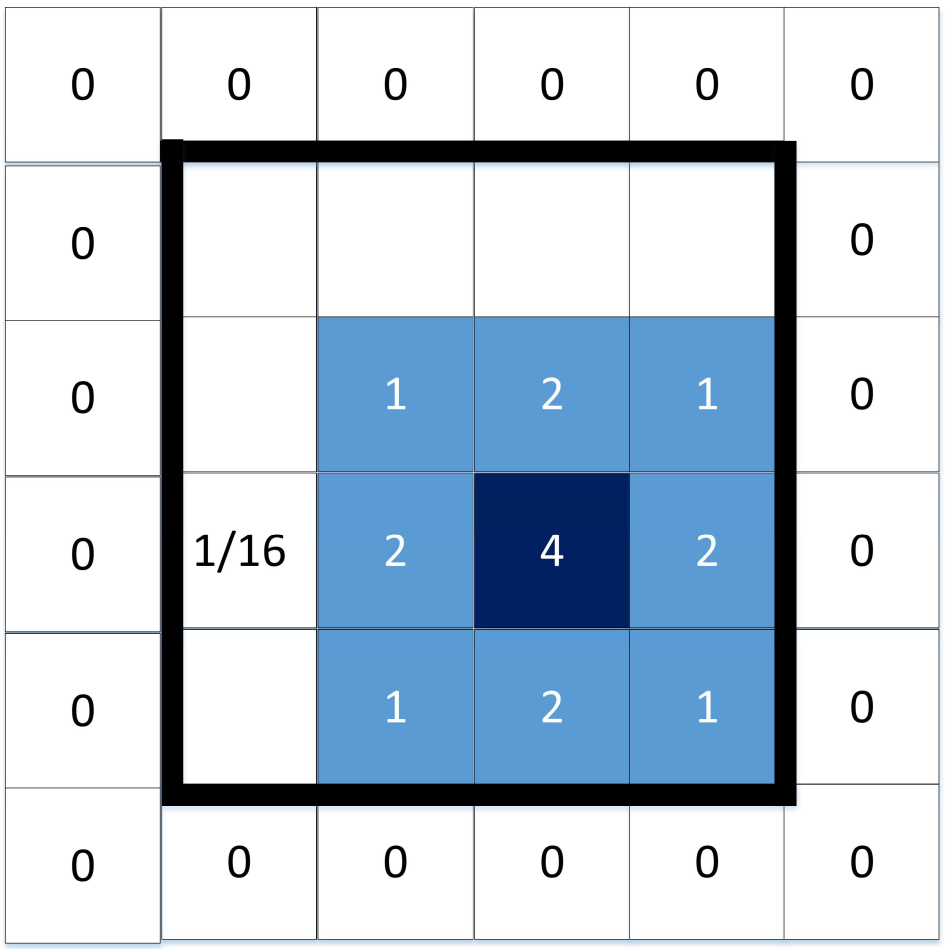
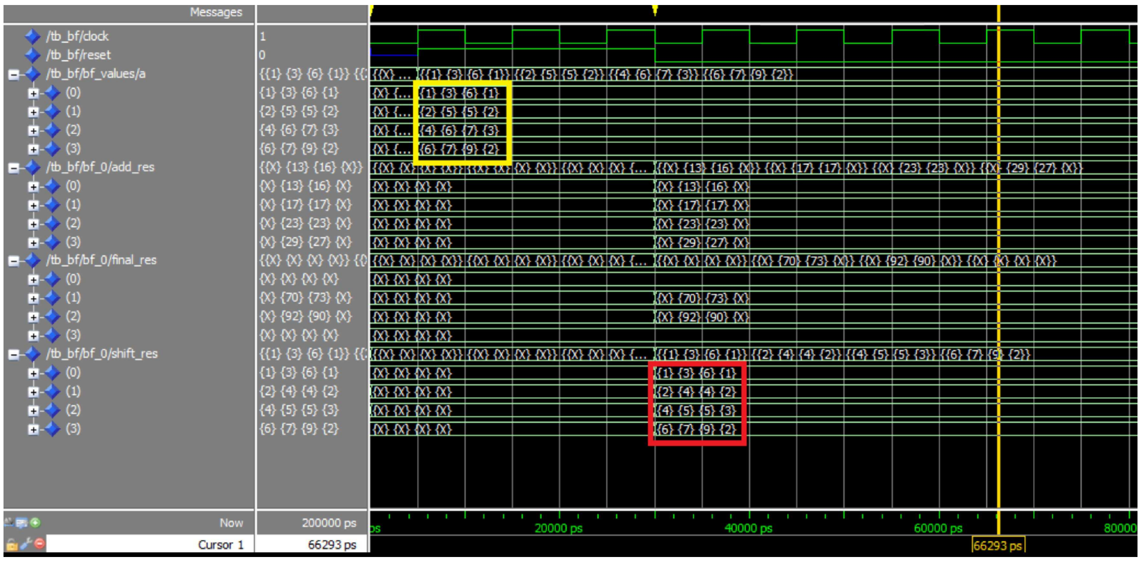


Figure 4.6: Binomial Filter, second implemetation

4.2 Simulations

4.2.1 Simulation type 1

The simulation shown in this work is referred to the hardware implementation descrived in the subsection 4.1.1 where the values of the matrix's border are leaved unchanged. In this testbench, we have a matrix 4x4 (fig.4.7 highlighted in yellow, and the output highlighted in red).



```
Messages
/tb_bf/clock            1
/tb_bf/reset            0
/tb_bf/bf_values/a      {{1} {3} {6} {1}} {{
  (0)                   {1} {3} {6} {1}
  (1)                   {2} {5} {5} {2}
  (2)                   {4} {6} {7} {3}
  (3)                   {6} {7} {9} {2}
/tb_bf/bf_0/add_res     {{X} {13} {16} {X}}
  (0)                   {X} {13} {16} {X}
  (1)                   {X} {17} {17} {X}
  (2)                   {X} {23} {23} {X}
  (3)                   {X} {29} {27} {X}
/tb_bf/bf_0/final_res   {{X} {X} {X} {X}} {{
  (0)                   {X} {X} {X} {X}
  (1)                   {X} {70} {73} {X}
  (2)                   {X} {92} {90} {X}
  (3)                   {X} {X} {X} {X}
/tb_bf/bf_0/shift_res   {{1} {3} {6} {1}} {{
  (0)                   {1} {3} {6} {1}
  (1)                   {2} {4} {4} {2}
  (2)                   {4} {5} {5} {3}
  (3)                   {6} {7} {9} {2}
Now                     200000 ps
Cursor 1                66293 ps
```

Figure 4.7: Binomial Filter-Results of the simulation type 1

At the beginning we have an input array that contains the values of the matrix

$$\begin{bmatrix} 1 & 3 & 6 & 1 \\ 2 & 5 & 5 & 2 \\ 4 & 6 & 7 & 3 \\ 6 & 7 & 9 & 2 \end{bmatrix}$$

The first step of this implementation is to do the partial weighted sum for three consecutive cell

$$S_{x,y} = cell_{x-1,y} + 2 \cdot cell_{x,y} + cell_{x+1,y}$$
$$\forall x \in [1,n-2] \quad \forall y \in [0,n-1]$$

$$\begin{bmatrix} X & 1+2\cdot3+6 & 3+2\cdot6+1 & X \\ X & 2+2\cdot5+5 & 5+2\cdot5+2 & X \\ X & 4+2\cdot6+7 & 6+2\cdot7+3 & X \\ X & 6+2\cdot7+9 & 7+2\cdot9+2 & X \end{bmatrix}$$

which will lead us to the result

$$\begin{bmatrix} X & 13 & 16 & X \\ X & 17 & 17 & X \\ X & 23 & 23 & X \\ X & 29 & 27 & X \end{bmatrix}$$

which is the same shown in the simulation (fig 4.7, *add_res* array).
Now we have to do the final sum

$$SUM_{x,y} = S_{x,y-1} + 2\cdot S_{x,y} + S_{x,y+1}$$
$$\forall x \in [1,n-2] \quad \forall y \in [1,n-2]$$

$$\begin{bmatrix} X & X & X & X \\ X & 13+2\cdot17+23 & 16+2\cdot17+23 & X \\ X & 17+2\cdot23+29 & 17+2\cdot23+27 & X \\ X & X & X & X \end{bmatrix} = \begin{bmatrix} X & X & X & X \\ X & 70 & 73 & X \\ X & 92 & 90 & X \\ X & X & X & X \end{bmatrix}$$

this matrix is the same we obtained in the simulation (fig 4.7, *final_res* array). The work is not over yet because we need to divide these sums by 16

$$\begin{bmatrix} X & X & X & X \\ X & \frac{70}{16} & \frac{73}{16} & X \\ X & \frac{92}{16} & \frac{90}{16} & X \\ X & X & X & X \end{bmatrix} = \begin{bmatrix} X & X & X & X \\ X & 4.375 & 4.5625 & X \\ X & 5.75 & 5.625 & X \\ X & X & X & X \end{bmatrix}$$

Since the values of the pixel are integer, we can round the matrix as

$$\begin{bmatrix} X & X & X & X \\ X & 4 & 4 & X \\ X & 5 & 5 & X \\ X & X & X & X \end{bmatrix}$$

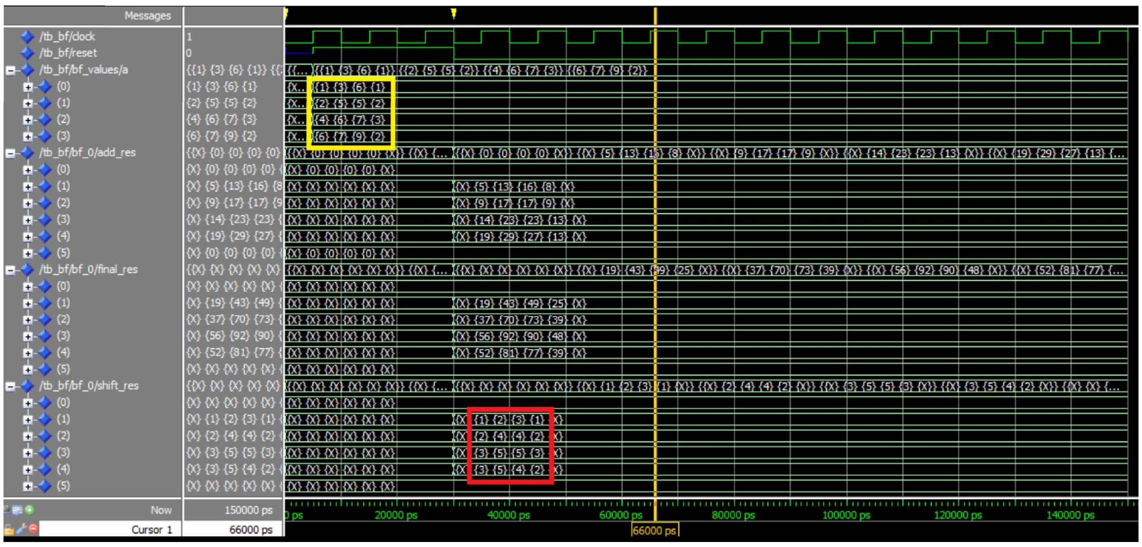
so the final result is

$$\begin{bmatrix} 1 & 3 & 6 & 1 \\ 2 & 4 & 4 & 2 \\ 4 & 5 & 5 & 3 \\ 6 & 7 & 9 & 2 \end{bmatrix}$$

which is the same as shown in the simulation (fig 4.7, *shifft_res* array).

4.2.2 Simulation type 2

The simulation shown in this work is referred to the hardware implementation descrived in the subsection 4.1.2 where also the final values of the matrix's border are computed using the binomial filter criteria. In this testbench, we have a matrix 4x4 (fig.4.7 highlighted in yellow, and the output highlighted in red).



```
Messages
/tb_bf/clock            1
/tb_bf/reset            0
/tb_bf/bf_values/a      {{1} {3} {6} {1}} {{
  (0)                   {1} {3} {6} {1}
  (1)                   {2} {5} {5} {2}
  (2)                   {4} {6} {7} {3}
  (3)                   {6} {7} {9} {2}
/tb_bf/bf_0/add_res     {{X} {0} {0} {0} {0}
  (0)                   {X} {0} {0} {0} {0} {
  (1)                   {X} {5} {13} {16} {8
  (2)                   {X} {9} {17} {17} {9
  (3)                   {X} {14} {23} {23} {
  (4)                   {X} {19} {29} {27} {
  (5)                   {X} {0} {0} {0} {0} {
/tb_bf/bf_0/final_res   {{X} {X} {X} {X} {X}
  (0)                   {X} {X} {X} {X} {X} {
  (1)                   {X} {19} {43} {49} {
  (2)                   {X} {37} {70} {73} {
  (3)                   {X} {56} {92} {90} {
  (4)                   {X} {52} {81} {77} {
  (5)                   {X} {X} {X} {X} {X} {
/tb_bf/bf_0/shift_res   {{X} {X} {X} {X} {X}
  (0)                   {X} {X} {X} {X} {X} {
  (1)                   {X} {1} {2} {3} {1} {
  (2)                   {X} {2} {4} {4} {2} {
  (3)                   {X} {3} {5} {5} {3} {
  (4)                   {X} {3} {5} {4} {2} {
  (5)                   {X} {X} {X} {X} {X} {
Now                     150000 ps
Cursor 1                66000 ps
```

Figure 4.8: Binomial Filter-Results of the simulation type 2

At the beginning we have an input array that contains the values of the matrix

$$\begin{bmatrix} 1 & 3 & 6 & 1 \\ 2 & 5 & 5 & 2 \\ 4 & 6 & 7 & 3 \\ 6 & 7 & 9 & 2 \end{bmatrix}$$

For this implementation, we will perform the binomial filter for the matrix that has a border of $'0'$ and in the center the input matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 6 & 1 & 0 \\ 0 & 2 & 5 & 5 & 2 & 0 \\ 0 & 4 & 6 & 7 & 3 & 0 \\ 0 & 6 & 7 & 9 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Now we perform the same steps described in section 4.2.1. We do the partial weighted sum for three consecutive cell

$$S_{x,y} = cell_{x-1,y} + 2 \cdot cell_{x,y} + cell_{x+1,y}$$
$$\forall x \in [1,n-2] \quad \forall y \in [0,n-1]$$

$$\begin{bmatrix} X & 0 & 0 & 0 & 0 & X \\ X & 2\cdot 1+3 & 1+2\cdot 3+6 & 3+2\cdot 6+1 & 6+2\cdot 1 & X \\ X & 2\cdot 2+5 & 2+2\cdot 5+5 & 5+2\cdot 5+2 & 5+2\cdot 2 & X \\ X & 2\cdot 4+6 & 4+2\cdot 6+7 & 6+2\cdot 7+3 & 7+2\cdot 3 & X \\ X & 2\cdot 6+7 & 6+2\cdot 7+9 & 7+2\cdot 9+2 & 9+2\cdot 2 & X \\ X & 0 & 0 & 0 & 0 & X \end{bmatrix}$$

which will lead us to the result

$$\begin{bmatrix} X & 0 & 0 & 0 & 0 & X \\ X & 5 & 13 & 16 & 8 & X \\ X & 9 & 17 & 17 & 9 & X \\ X & 14 & 23 & 23 & 13 & X \\ X & 19 & 29 & 27 & 13 & X \\ X & 0 & 0 & 0 & 0 & X \end{bmatrix}$$

which is the same shown in the simulation (fig 4.8, *add_res* array).
Now we have to do the final sum

$$SUM_{x,y} = S_{x,y-1} + 2 \cdot S_{x,y} + S_{x,y+1}$$
$$\forall x \in [1,n-2] \quad \forall y \in [1,n-2]$$

$$\begin{bmatrix} X & X & X & X & X & X \\ X & 2\cdot 5+9 & 2\cdot 13+17 & 2\cdot 16+17 & 2\cdot 8+9 & X \\ X & 5+2\cdot 9+17 & 13+2\cdot 17+23 & 16+2\cdot 17+23 & 8+2\cdot 9+13 & X \\ X & 9+2\cdot 14+19 & 17+2\cdot 23+29 & 17+2\cdot 23+27 & 9+2\cdot 13+13 & X \\ X & 14+2\cdot 19 & 23+2\cdot 29 & 23+2\cdot 27 & 13+2\cdot 13 & X \\ X & X & X & X & X & X \end{bmatrix} =$$

$$= \begin{bmatrix} X & X & X & X & X & X \\ x & 19 & 43 & 49 & 25 & X \\ X & 37 & 70 & 73 & 39 & X \\ X & 56 & 92 & 90 & 48 & X \\ X & 52 & 81 & 77 & 39 & X \\ X & X & X & X & X & X \end{bmatrix}$$

this matrix is the same we obtained in the simulation (fig 4.8, *final_res* array). The work is not over yet because we need to divide these sums by 16

$$\begin{bmatrix} X & X & X & X & X & X \\ X & \frac{19}{16} & \frac{43}{16} & \frac{49}{16} & \frac{25}{16} & X \\ X & \frac{37}{16} & \frac{70}{16} & \frac{73}{16} & \frac{39}{16} & X \\ X & \frac{56}{16} & \frac{92}{16} & \frac{90}{16} & \frac{48}{16} & X \\ X & \frac{52}{16} & \frac{81}{16} & \frac{77}{16} & \frac{39}{16} & X \\ X & X & X & X & X & X \end{bmatrix} = \begin{bmatrix} X & X & X & X & X & X \\ X & 1.1875 & 2.6875 & 3.0625 & 1.5625 & X \\ X & 2.3125 & 4.375 & 4.5625 & 2.4375 & X \\ X & 3.5 & 5.75 & 5.625 & 3 & X \\ X & 3.25 & 5.0625 & 4.8125 & 2.4375 & X \\ X & X & X & X & X & X \end{bmatrix}$$

Since the values of the pixel are integer, we can round the matrix as

$$\begin{bmatrix} X & X & X & X & X & X \\ X & 1 & 2 & 3 & 1 & X \\ X & 2 & 4 & 4 & 2 & X \\ X & 3 & 5 & 5 & 3 & X \\ X & 3 & 5 & 4 & 2 & X \\ X & X & X & X & X & X \end{bmatrix}$$

4.3 Comparison

4.3.0.1 Logic-In-Memory implementation

The architecture of the Logic-In-memory was already explained in the subsection 2.5.4. The pseudocode for the LIM is

1. Each cell reads its own value;
2. Reads all neighbouring cells data (8 different values);
3. Sums all values and divides them by 16;
4. Writes the final result in its own memory;

The main advantage of the LIM architecture is the parallelism: the presence of many cells that can work autonomously greatly increases the speed of the computation of algorithms that can be executed in parallel. [8] It has been calculated that the total duration for the computation is equal to:

- N clock cycles to write the data;
- 28 clock cycles to compute the algorithm;
- 2N+1 clock cycles to read the data through a remote read. Therefore,

$$t = N + 28 + 2N + 1 = 3N + 29$$

	LIM	This work version 1	This work version 2
Area	$N \cdot N$ cells	$2N \cdot (N-2) + {}$ $+2(N-2)^2$ adders	$(4N-4)+$ $2(N-2)^2$ adders
Time	$3N + 29$ cycles	time for 4 adders	time for 4 adders

As for this work by looking to the section 4.1, we know that the time required is the time for 4 adder in sequence, this because 2 adder are required for doing the weighted sum of 3 consecutive cells, and the other 2 for doing the weighted sum of the 9 cells

4.4 Characteristics of the Binomial Filter

PROCESSING ELEMENTS

1- Which kind of Processing element?

Adder (static shifter)

2- Functionality

Addition, (static shifting i.e. multiply by 2 and 4, division by 16)

3- Complexity

Version 1	Area: $2[N(N-2)+(N-2)^2]$ adders
	Time: time for 4 additions
Version 2	Area $4(N-1)+2(N-2)^2$ adders
	Time: time for 4 additions

4- Parallelism

All the cell can perform their binomial filter value.

5-Reconfigurability

No

6- Programmability

No

7- Need a dedicated memory?

If the architecture is pipelined, you need to store the partial sum.

If not pipelined, no memory is required.

8- Relationship with I/O

INPUT: values of the matrix's cells

OUTPUT: result of the binomial filter algorithm

MEMORY ELEMENTS

1- Need a clever memory LIM?

No, but can be implemented

2- Is there a data search algorithm?

No

3-Interface mechanism with other PE or memories

Communication required between cells in the neighborhood

4- Access mechanism

(No memory for this implementation)

5- Hierarchization

(No memory for this implementation)

6- Cache coherency

(No memory for this implementation)

7- Is it a a transactional memory?

(No memory for this implementation)

8- Are there virtualization (paging) mechanisms?

(No memory for this implementation)

ENCODING INFORMATION

1-Which encoding is used?

Binary encoding

CONNECTIONS

1-Packet Exchange Protocol
Directly

2-Timing (asynchronou/synchronous)
Synchronous

3-Are there multiple instances?
Yes

4-Heterogeneity (Local/Distant I/O Connections)
Heterogeneous when storing the initial values to the cells and when exchanging the data (For version 2, the border cells require less computation and less data exchanging)

5-Are there any buffers?
No

Chapter 5

Digital Filter FIR

A digital filter is a system performing a certain function on an input stream $X[n]$ (samples are received at constant rate) and generating an output stream called $Y[n]$. In general it's possible to have multiple input streams and multiple output streams (for example in a parallel architecture this allows to speed up the filter).

5.1 Digital filters properties

Filters are systems characterized by the following properties [15]:

1. **Linearity**: it is possible to represent the output as an overlapping of the input pulses. Let $\delta[n-k]$ be the input signal and $h_k[n]$ the filter response, using overlapping property, assuming that the input is a sum of pulses:

$$x[n] = \sum_k (x[n] \cdot \delta[n-k])$$

   so the output is: $h_k[n] = F(\delta[n-k])$ (F is the filter funtion), so applying linearity:

$$F(x[n]) = F(\sum_k (x[n] \cdot \delta[n-k]))$$

$$y[n] = \sum_k x[n] \cdot h_k[n]$$

   It means that if the input is a weighted sum of pulses, also the output will have the same form.

2. **Time invariance**: the shape of the answer h_k is not dependent on the selected time k. So applying the same input signal at different time (i.e. different k), the filter response will be always the same, just time-shifted. Therefore:

$$h_k[n] = h[n-k]$$

The expression for the output found before can be rewritten as:

$$y[n] = \sum_k x[n] \cdot h_k[n] = \sum_k x[n] \cdot h[n-k] = x[n] * h[n]$$

where it has been used the definition of convolution product.

3. **Stability**: we expect to have at the output a stable stream of samples whose values do not diverge. This is true if a sufficient numbers of bit has been employed for representing the samples. Assuming that the system is BIBO (bounded input bounded output), the following condition should be satisfied:

$$S = \sum_k |h[n]| < \infty$$

4. **Causality**: if we apply an input at time k, we expect that the output doesn't come before time k. Assuming that $x[n] = 0$ for $n < 0$, then $y[n] = 0$ for $k < 0$: this property is true only if $h[n] = 0$ for $n < 0$

5.2 Hardware Implementation

An FIR filter can be easily implemented using just three digital hardware elements, a unit delay (D Flip Flop), a multiplier, and an adder. The unit delay simply updates its output once per sample period, using the value of the input as its new output value. In the convolution sum,

$$y[n] = \sum_k x[n] \cdot h[n-k] = \sum_k x[n-k] \cdot h[n]$$

notice that at each n we need access to $x(n)$, $x(n-1)$, $x(n-2)$, $\cdots$, $x(n-k)$. We can maintain this set of values by cascading a set of D Flip Flops to form a delay line, as shown in Fig. 5.1

For each of the previous k inputs, we have to scale them by $h(0)$, $h(1)$, $\cdots$, $h(k)$. To obtain these values, we simply put a multiplier as shown in the following picture (Fig. 5.2). To obtain the final result, we have to sum all the results of these multiplications.

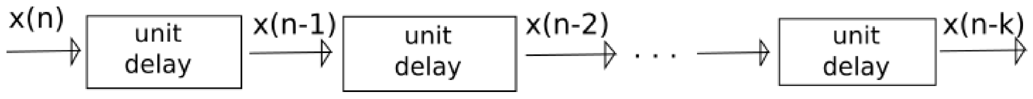


Figure 5.1: Cascading D Flip Flops

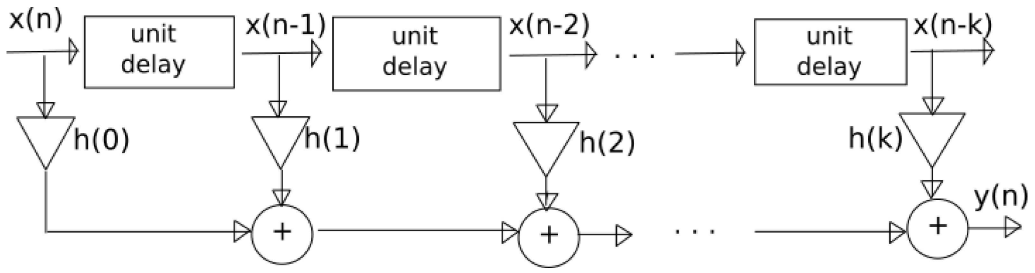


Figure 5.2: FIR hardware implementation

5.3 Simulation

The simulation shown in this work has 6 constants (fig. 5.3, *fir_constants_value* section)

$$\begin{aligned} h_0 &= 12 \\ h_1 &= 10 \\ h_2 &= 8 \\ h_3 &= 6 \\ h_4 &= 4 \\ h_5 &= 2 \end{aligned}$$

and it will take into account the last 6 inputs.

When we start sampling after the reset, we set all the previous inputs to $'0'$, therefore the output will be simply $y[n] = x[n] \cdot h_0$.
In fact in the simulation, as soon as the reset was set low, we see the output (fig. 5.3,*fir_res* signal) to be

$$y[n] = x[n] \cdot h_0 = 1 \cdot 12 = 12.$$

In the next cycle we get

$$y[n] = x[n] \cdot h_0 + x[n-1] \cdot h_1 = 10 \cdot 12 + 1 \cdot 10 = 130.$$

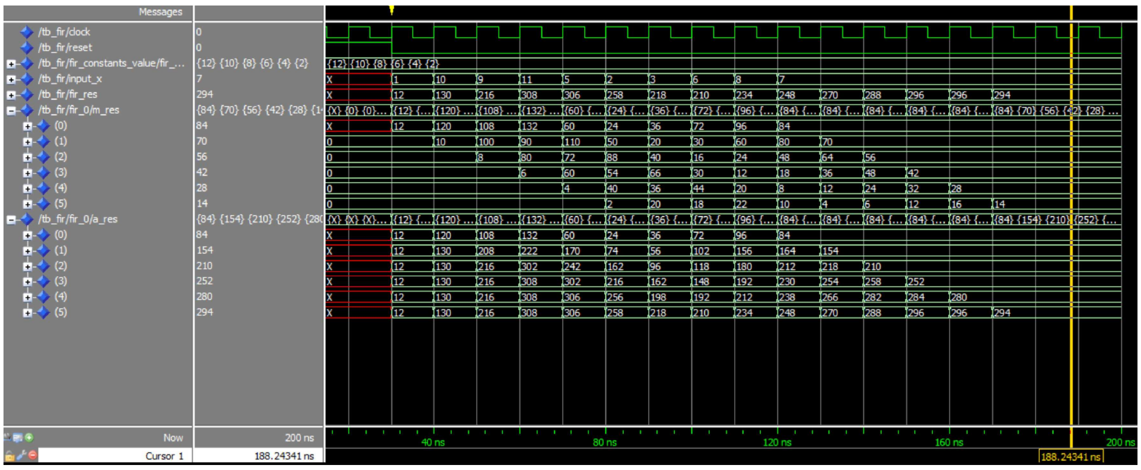


Figure 5.3: FIR's simulation

For the next clock cycles we repeat the same procedure.
In the end, when the input has not changed for more than 6 clock cycles, we can see the result to be

$$y[n] = \sum_{k=0}^{6} x[n-k] \cdot h[k] = 7 \sum_{k=0}^{6} = 7 \cdot 42 = 294$$

5.4 Characteristics of Filter FIR

PROCESSING ELEMENTS

1- Which kind of Processing element?

Adder, multiplier, registers

2- Functionality

Addition, multiplications, storing.

3- Complexity

Area:	$N-1$ adders
	$N-1$ registers
	N multipliers
Time:	Time for one registers, one multiplication and $N-1$ additions

4- Parallelism

Registers and multiplications can work in parallel, The chain made of adder not (fig. 5.2).

5-Reconfigurability

No

6- Programmability

No

7- Need a dedicated memory?

It needs some memory to store the previous $N-1$ input values.
It is possible also to store the N constants

8- Relationship with I/O

INPUT: values of the input and values of the constatnts.
OUTPUT: result of the filter FIR algorithm

MEMORY ELEMENTS

1- Need a clever memory LIM?

No, but can be implemented

2- Is there a data search algorithm?

No

3-Interface mechanism with other PE or memories

Communication required between local elements

4- Access mechanism

(No memory for this implementation)

5- Hierarchization

(No memory for this implementation)

6- Cache coherency

(No memory for this implementation)

7- Is it a a transactional memory?

(No memory for this implementation)

8- Are there virtualization (paging) mechanisms?

(No memory for this implementation)

ENCODING INFORMATION

1-Which encoding is used?

Binary encoding

CONNECTIONS

1-Packet Exchange Protocol

Directly

2-Timing (asynchronou/synchronous)

Synchronous

3-Are there multiple instances?

Yes

4-Heterogeneity (Local/Distant I/O Connections)

Heterogeneous. Each elements is connected to the previous and following elementss. (No connections between distant elements)

5-Are there any buffers?

There are some registers.

Chapter 6

Transport equation problem

Fluid dynamics and transport phenomena, such as heat and mass transfer, play a vitally important role in human life.
Gases and liquids surround us, flow inside our bodies, and have a profound influence on the environment in which we live.
Fluid flows produce winds, rains, floods, and hurricanes.
Convection and diffusion are responsible for temperature fluctuations and transport of pollutants in air, water or soil.
The ability to understand, predict, and control transport phenomena is essential for many industrial applications,such as aerodynamic shape design, oil recovery from an underground reservoir, or multiphase/multicomponent flows in furnaces, heat exchangers, and chemical reactors.
The traditional approach to investigation of a physical process is based on observations, experiments, and measurements. The amount of information that can be obtained in this way is usually very limited and subject to measurement errors.
Alternatively, an analytical or computational study can be performed on the basis of a suitable mathematical model. As a rule, such a model consists of several differential and/or algebraic equations which make it possible to predict how the quantities of interest evolve and interact with one another. [16]

The equations and the functionality used in this work, are the same described in the work of a student for the Logic-In-Memory.
The transport equation is a partial differential equation that describes the distribution of heat (or variation in temperature) in a given region over time.

$$\frac{\partial}{\partial x} + \frac{\partial}{\partial y} = -\frac{\partial}{\partial t} \tag{6.1}$$

Temperature values are calculated over a grid and initialized with a gaussian distribution. The points of the grid represent the local indexes (ix, iy) of the matrix that contains the temperature values. The domain is shown in orange in the picture below. Boundaries are represented in white. Data are evolved along Y=X direction, i.e. towards the up-right corner of the coordinate system [8].

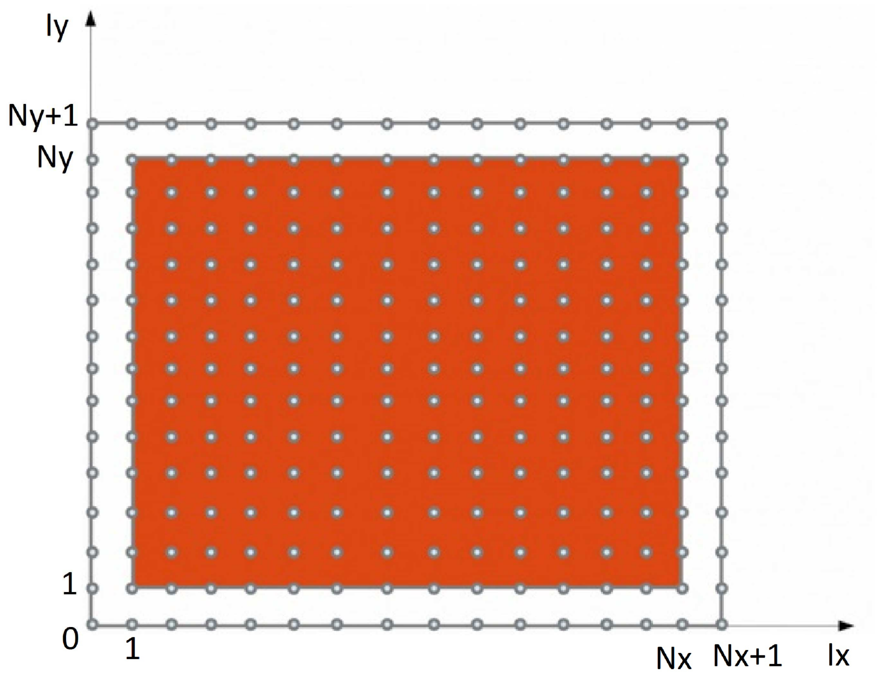


Figure 6.1: Grid for the Transport equation Problem

Each dot of the grid represents a cell. All the cells inside the orange rectangle have to calculate the new value of temperature and the cells at the boundary have to copy the calculated value from the cells at the border of the orange rectangle. Each cell works autonomously performing these calculations:

$$temp = t_0 - \alpha[t(i+1,j) - t(i-1,j) + t(i,j+1) - t(i,j-1)] \tag{6.2}$$

As shown below, each cell reads the values of temperature of the adjacent cells to compute the equation illustrated above: After that, the cells near the boundary copy the calculated values to the boundary cells:

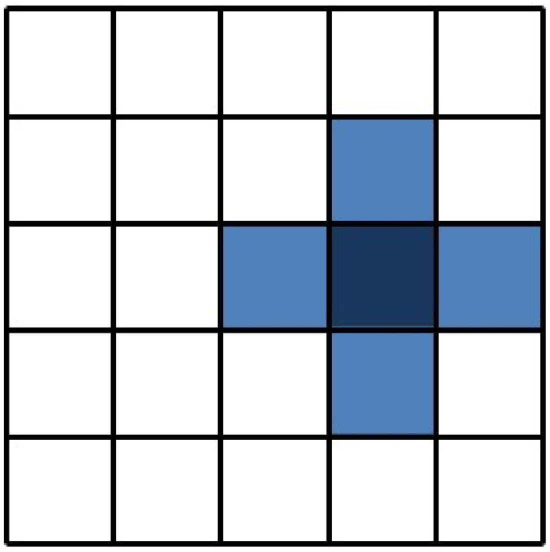


Figure 6.2: Transport Equation Problem for a single cell

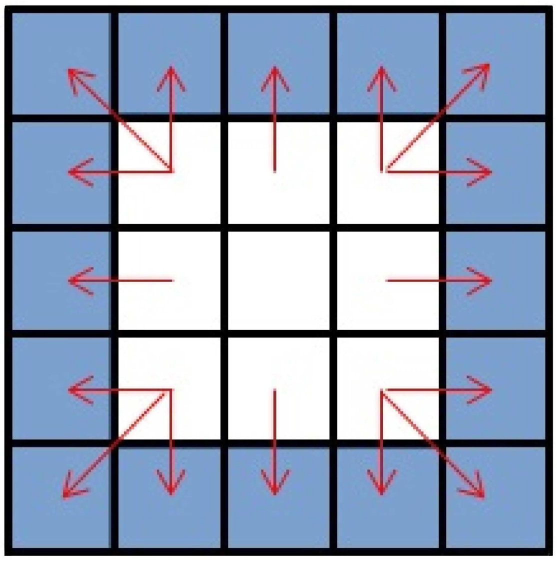


Figure 6.3: Propagarting the values to the boundary

6.1 Hardware Implementation

Since each cell has to compute the same equation, and since there is no data dependencies between them, we can implement a component that compute this result. Let's call it *tep_unit*. The *tep_unit* has as input

- t0: the value of the interested cell
- tr: the value of the cell on the right side
- tup: the value of the cell on the upper side
- tl: the value of the cell on the left side
- tdn: the value of the cell on the lower side
- alpha : the value of theconstant

with these values, it is possible to calculate the final value.
The equation above can be rewritten as :

$$temp = t_0 + \alpha[t(i-1,j) + t(i,j-1)] - [t(i+1,j) + t(i,j+1)] \tag{6.3}$$

For the cell on the border or on the angle of the matrix, we will compute the equation as if there were some zero cells in the neighborhood. In fig. 6.4 we can see the DFG.

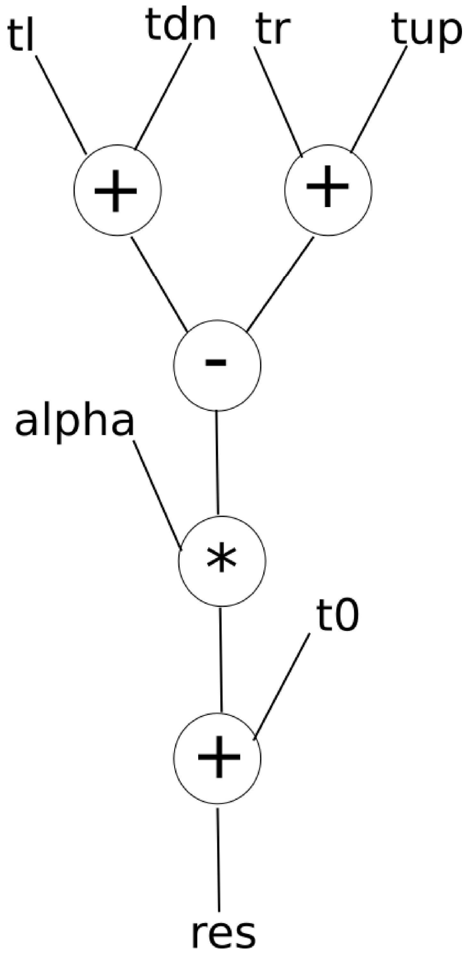
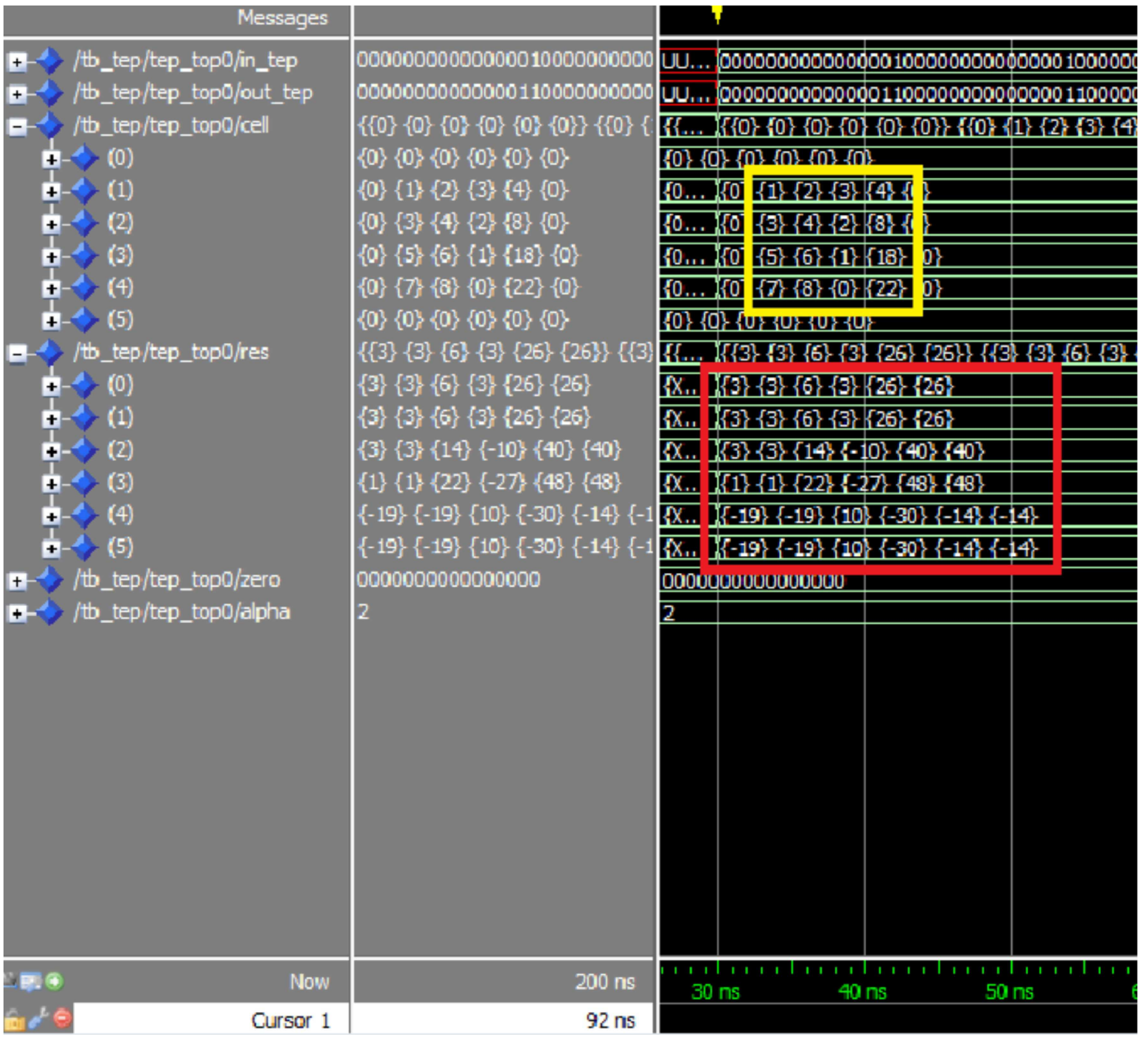


Figure 6.4: Tep_unit DFG

6.2 Simulation

The simulation shown in this work has a 4x4 matrix (fig. 6.5, highlighted in yellow)

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 8 \\ 5 & 6 & 1 & 18 \\ 7 & 8 & 0 & 22 \end{bmatrix}$$



```
Messages
/tb_tep/tep_top0/in_tep    0000000000000000100000000000
/tb_tep/tep_top0/out_tep   0000000000000000110000000000
/tb_tep/tep_top0/cell      {{0} {0} {0} {0} {0} {0}} {{0} {
   (0)                     {0} {0} {0} {0} {0} {0}
   (1)                     {0} {1} {2} {3} {4} {0}
   (2)                     {0} {3} {4} {2} {8} {0}
   (3)                     {0} {5} {6} {1} {18} {0}
   (4)                     {0} {7} {8} {0} {22} {0}
   (5)                     {0} {0} {0} {0} {0} {0}
/tb_tep/tep_top0/res       {{3} {3} {6} {3} {26} {26}} {{3}
   (0)                     {3} {3} {6} {3} {26} {26}
   (1)                     {3} {3} {6} {3} {26} {26}
   (2)                     {3} {3} {14} {-10} {40} {40}
   (3)                     {1} {1} {22} {-27} {48} {48}
   (4)                     {-19} {-19} {10} {-30} {-14} {-1
   (5)                     {-19} {-19} {10} {-30} {-14} {-1
/tb_tep/tep_top0/zero      0000000000000000
/tb_tep/tep_top0/alpha     2

Now                        200 ns
Cursor 1                   92 ns
```

Figure 6.5: Transport Equation Problem's simulation

If we compute the equation 6.3 for each element we will get

$$\begin{bmatrix} 3 & 6 & 3 & 26 \\ 3 & 14 & -10 & 40 \\ 1 & 22 & -27 & 48 \\ -19 & 10 & -30 & -14 \end{bmatrix}$$

Now we have to propagate the values on the border to the boundary, thus having

$$\begin{bmatrix} 3 & 3 & 6 & 3 & 26 & 26 \\ 3 & 3 & 6 & 3 & 26 & 26 \\ 3 & 3 & 14 & -10 & 40 & 40 \\ 1 & 1 & 22 & -27 & 48 & 48 \\ 19 & -19 & 10 & -30 & -14 & -14 \\ 19 & -19 & 10 & -30 & -14 & -14 \end{bmatrix}$$

6.3 Comparison with LIM's architecture

I take the results of the LIM's architecture. The pseudocode for the LIM architecture is the following:

1. Each cell reads its own value;
2. Reads N-W-S-E cells data (4 different values);
3. Computes sums/subtractions and a multiplication;
4. Boundary cells just copy their neighbours' values;
5. Writes the final result in its own memory;

As already said, the main advantage of the LIM architecture is the parallelism: the presence of many cells that can work autonomously greatly increases the speed of the computation of algorithms that can be executed in parallel.
In particular, the duration of the algorithms doesn't depend on the number of cells composing the grid. The total duration for the computation is equal to:

- N clock cycles to write the data;

- $16 \cdot i$clock cycles to compute the algorithm;
- $2N + 1$ clock cycles to read the data through a remote read.

Therefore,

$$t = N + 16 \cdot i + 2N + 1 = 3N + 16 \cdot i + 1$$

	LIM	This work
Area	N^2 cells	$3N^2$ adders, N^2 substractors, multipliers
Time	$3N + 16 \cdot i + 1$	(time for 3 additions) + time for a single multiplication

6.4 Characteristics of Transport Equation Problem

PROCESSING ELEMENTS

1- Which kind of Processing element?

Adder, multiplier, substractors

2- Functionality

Addition, multiplications, substractions.

3- Complexity

Area:	$3N^2$ adders
	N^2 substractors
	N^2 multipliers
Time:	Time for one one multiplication and 3 additions

4- Parallelism

Each cell can evaluate their Transport Equation Problem output

5-Reconfigurability

No

6- Programmability

No

7- Need a dedicated memory?

No

8- Relationship with I/O

INPUT: values of the cells in the matrix

OUTPUT: result of the Transport Equation Problem. algorithm

MEMORY ELEMENTS

1- Need a clever memory LIM?

No, but can be implemented

2- Is there a data search algorithm?

No

3-Interface mechanism with other PE or memories

Communication required between local elements

4- Access mechanism

(No memory for this implementation)

5- Hierarchization

(No memory for this implementation)

6- Cache coherency

(No memory for this implementation)

7- Is it a a transactional memory?

(No memory for this implementation)

8- Are there virtualization (paging) mechanisms?

(No memory for this implementation)

ENCODING INFORMATION

1-Which encoding is used?

Binary encoding

CONNECTIONS

1-Packet Exchange Protocol
Directly

2-Timing (asynchronou/synchronous)
Synchronous

3-Are there multiple instances?
Yes

4-Heterogeneity (Local/Distant I/O Connections)
Heterogeneous. Each cell exchanges data with local cells (No connections between distant cells)

5-Are there any buffers?
No.

Chapter 7

Magnetostatic field calculation 3D

To compute the total magnetostatic energy, the ferromagnetic systems can be discretized as three-dimensional cubes.

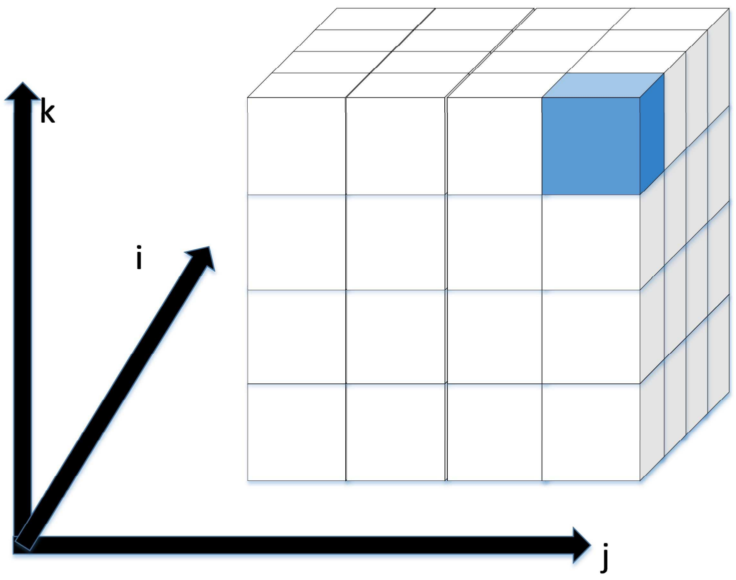


Figure 7.1: The selected cell is supposed to sum all the magnetostatic field contributions between it and all the other cells

The following is the implementation of an algorithm shown in the paper [17], related to the evaluation of the contribution of the magnetostatic field to the total energy of a system.
In general, the magnetostatic field is connected to the magnetization $[\overrightarrow{M}]$ through the magnetizing tensor [**TG**].

The evaluation of the magnetostatic field of a given cell requires the summation on all cells of the mesh due to the the long range of the dipole interaction.

$$\overrightarrow{H}(ijk) = -M_s \sum_{i'=1}^{N_x}\sum_{j'=1}^{N_y}\sum_{k'=1}^{N_z} \begin{bmatrix} TG_{xx} & TG_{xy} & TG_{xz} \\ TG_{yx} & TG_{yy} & TG_{yz} \\ TG_{zx} & TG_{zy} & TG_{zz} \end{bmatrix}_{(i-i',j-j',k-k')} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}_{(i',j',k')} \tag{7.1}$$

7.1 Product TG and M

Each cell $(i'j'k')$ has a finite state machine that evaluates the magnetostatic field contribution between the cell *ijk* and the cell *i'j'k'*.

$$\overrightarrow{h}(i-i',j-j',k-k') = \begin{bmatrix} TG_{xx} & TG_{xy} & TG_{xz} \\ TG_{yx} & TG_{yy} & TG_{yz} \\ TG_{zx} & TG_{zy} & TG_{zz} \end{bmatrix}_{(i-i',j-j',k-k')} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}_{(i',j',k')}$$

.

The matrix $\mathbf{TG}_{(i-i',j-j',k-k')}$ is the magnetizing tensor between the 2 cells, while $\overrightarrow{M}_{(i',j'k')}$ is the magnetization of the cell i',j',k'.

The FSM is composed by the following states:

- **READ_MX**: save m_x value
- **READ_MY**: save m_y value
- **READ_MZ**: save m_z value
- **READ_TGxx**: read TG_{xx} value and computing $op1 = TG_{xx} \cdot m_x$
- **READ_TGxy**: read TG_{xy} value and computing $op2 = TG_{xy} \cdot m_y$
- **READ_TGxz**: read TG_{xz} value and computing $h_x = TG_{xz} \cdot m_z + op1 + op2$.
- **READ_TGyx**:read TG_{yx} value and computing $op1 = TG_{yx} \cdot m_x$
- **READ_TGyy**:read TG_{yy} value and computing $op2 = TG_{yy} \cdot m_y$
- **READ_TGyz**: read TG_{yz} value and computing $h_y = TG_{yz} \cdot m_z + op1 + op2$
- **READ_TGzx**:read TG_{zx} value and computing $op1 = TG_{zx} \cdot m_x$

- **READ_TGzy**:read TG_{zy} value and computing $op2 = TG_{zy} \cdot m_y$
- **READ_TGzz**: read TG_{zz} value and computing $h_z = TG_{zz} \cdot m_z + op1 + op2$

7.2 Logic Plane

This component groups all the cells in a given plane, and enables the computation of the magnetostatic field contributions between the cell ijk and all the cells in a given plane (i.e. k is fixed).

$$\overrightarrow{h}(i-i',j-j',k-k^*) \qquad \forall i' < N_x, \quad j' < N_y, \qquad k = k^*$$

which can be also seen as $\overrightarrow{h}(i-i',j-j',k-k^*)$ and can be computed as

$$\begin{bmatrix} TG_{xx} & TG_{xy} & TG_{xz} \\ TG_{yx} & TG_{yy} & TG_{yz} \\ TG_{zx} & TG_{zy} & TG_{zz} \end{bmatrix}_{(i-i',j-j',k-k^*)} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}_{(i',j',k^*)} \qquad \forall i' < N_x, \quad j' < N_y,$$

All this computation can be done in parallel.

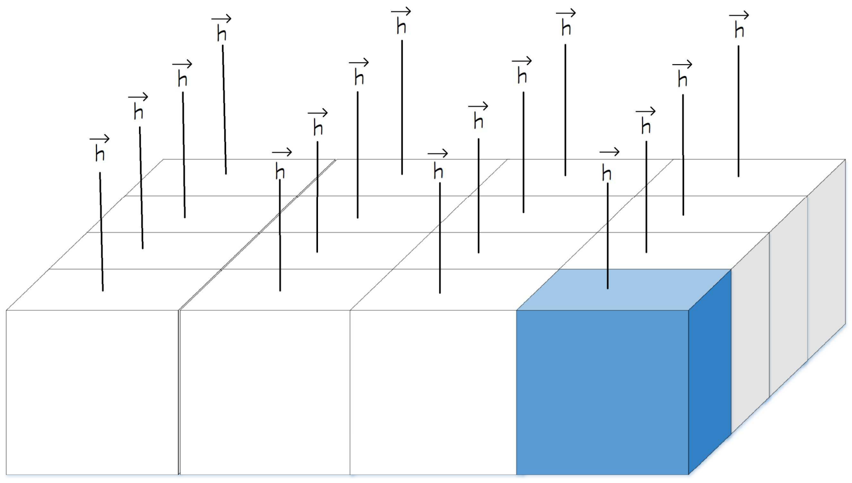


Figure 7.2: Magnetostatic field of each cell in a given plane

7.3 Sum all

The *sum_all* is a component that given a fixed k^* (i.e. a given plane) evaluates the magnetostatic field of all cells in a fixed plane. The input is given by the *logic plane*, as shown in the fig. 7.2.

$$\sum_{i'=1}^{N_x}\sum_{j'=1}^{N_y}\begin{bmatrix} TG_{xx} & TG_{xy} & TG_{xz} \\ TG_{yx} & TG_{yy} & TG_{yz} \\ TG_{zx} & TG_{zy} & TG_{zz} \end{bmatrix}_{(i-i',j-j',k-k^*)} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}_{(i',j',k^*)}$$

By looking to the formula 7.1, we have to evaluate the sum for each plane, so we store this sum and wait for the next plane to be computed and added to the previous sum.

Of course when we reach the k^{th} plane, we stop the sum and we set a signal to tell that the calculation is finished.

Overall, this component perform the following computation

$$\sum_{k'=1}^{N_z}\left(\sum_{i'=1}^{N_x}\sum_{j'=1}^{N_y}\begin{bmatrix} TG_{xx} & TG_{xy} & TG_{xz} \\ TG_{yx} & TG_{yy} & TG_{yz} \\ TG_{zx} & TG_{zy} & TG_{zz} \end{bmatrix}_{(i-i',j-j',k-k')} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}_{(i',j',k')}\right)$$

7.4 Simulation and Test

I wrote a testbench to check the correct behaviour of this architecture.

7.4.1 Testbench of the cell

First we check that the component *cell* is working according to the specifications. We can distinguish the 4 main phase of this test as shown in fig. 7.3. The 4 phases are

1. **Reading the magnetization**: here we read the magnetization in the 3 direction. This phase is highlighted in yellow in the figure.
   7.3. $m_x = 1$, $m_y = 2$, $m_z = 3$,

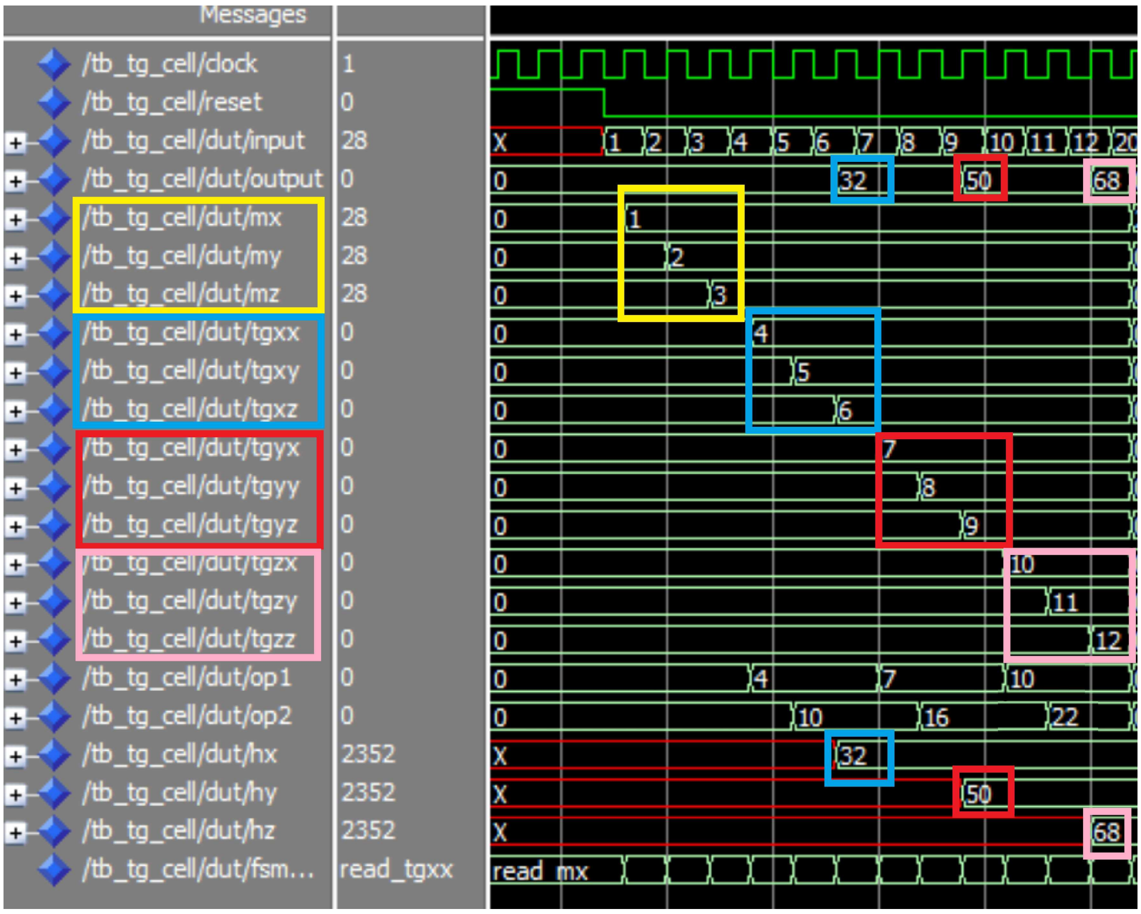


Figure 7.3: The phases of a single cell

2. **Reading the magnetization x-axis**: Reading the magnetizing tensor contributions between 2 cells. Right now we look only to the contributions along the x-axis with the x-axis, y-axis, z-axis.
   In the meantime we can evaluate the magnetostatic field h_x. This phase is highlighted in blue in the figure.
   $tg_{xx} = 4$, $tg_{xy} = 5$, $tg_{xz} = 6$
   $op1 = m_x \cdot tg_{xx} = 4$
   $op2 = m_y \cdot tg_{xy} = 10$
   $h_x = op1 + op2 + m_z \cdot tg_{xz} = 32$
   $output = h_x = 32$

3. **Reading the magnetization y-axis**: Similarly as the previous phase, we

look only to the magnetizing tensor contributions along the y-axis with the x-axis, y-axis, z-axis.
In the meantime we can evaluate the magnetostatic field h_y. This phase is highlighted in red in the figure.
$tg_{yx} = 7$, $tg_{yy} = 8$, $tg_{yz} = 9$
$op1 = m_x \cdot tg_{yx} = 7$
$op2 = m_y \cdot tg_{yy} = 16$
$h_x = op1 + op2 + m_z \cdot tg_{yz} = 50$
$output = h_x = 50$

4. **Reading the magnetization z-axis**: Similarly as the previous phase. This phase is highlighted in pink in the figure.
$tg_{yx} = 10$, $tg_{yy} = 11$, $tg_{yz} = 12$
$op1 = m_x \cdot tg_{yx} = 10$
$op2 = m_y \cdot tg_{yy} = 22$
$h_x = op1 + op2 + m_z \cdot tg_{yz} = 68$
$output = h_x = 68$

After these 4 phases, the cell starts again from phase 1 as shown in fig. 7.4.

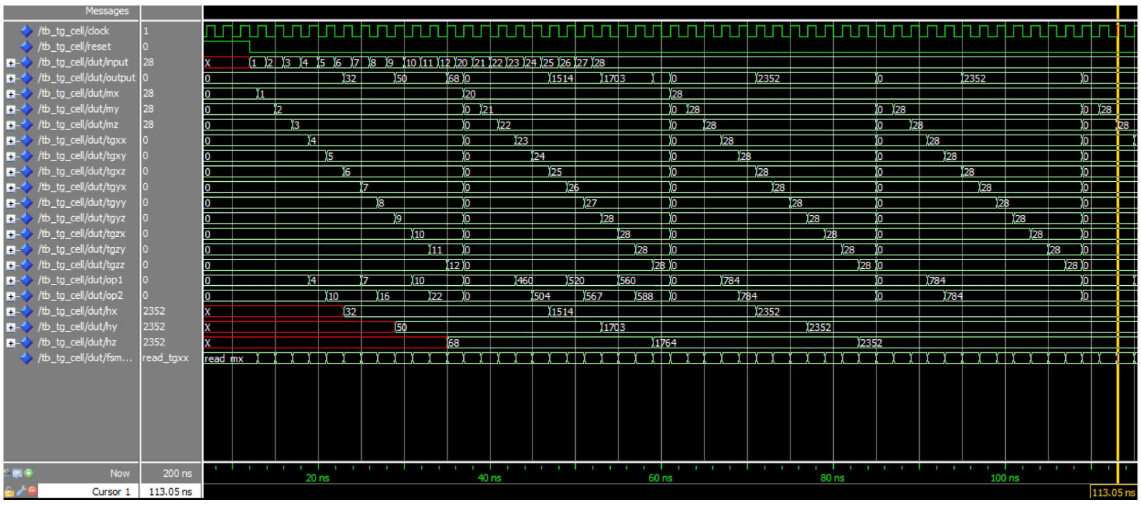


Figure 7.4: Testbench of a single cell

7.4.2 Testbench of the Top entity

After testing the *cell* components, we integrate them (*logic_plane*) with the *sum_all* accumulator.
The *logic_plane* and the *sum_all* components have to be synchronized, so a delay is needed (i.e. the logic plane evaluate the magnetostatic field of a plane and then the accumulator is ready to sum).
For simplicity, we test the behaviour of a parallelepiped which base is 2x2 and which height is 4.

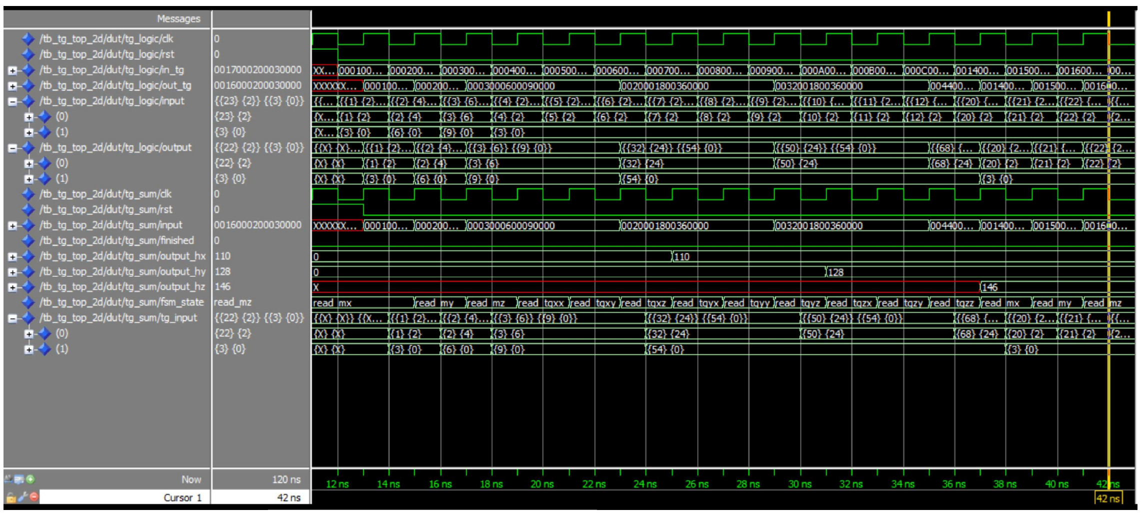
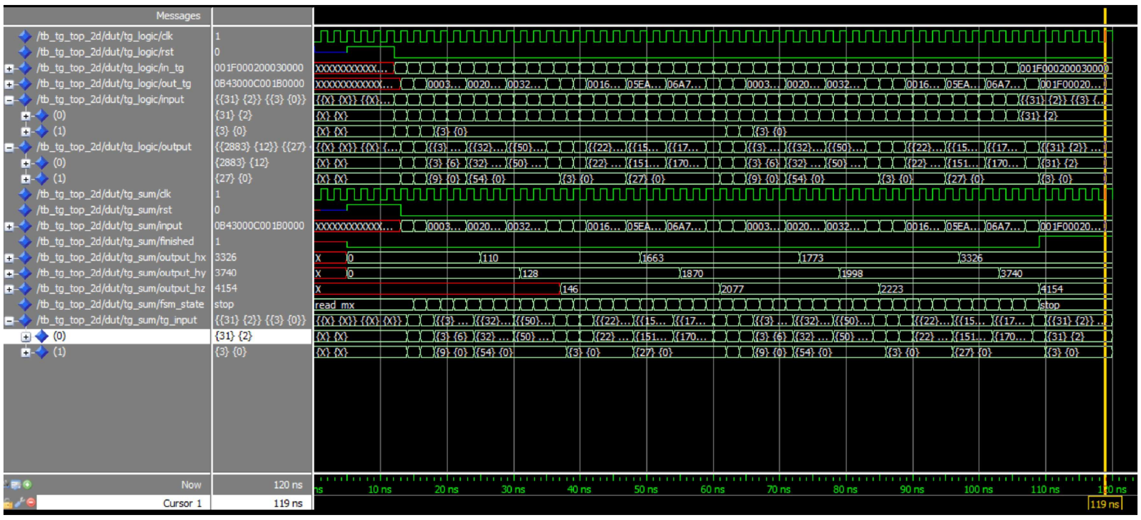

Messages
/tb_tg_top_2d/dut/tg_logic/clk
/tb_tg_top_2d/dut/tg_logic/rst
/tb_tg_top_2d/dut/tg_logic/in_tg
/tb_tg_top_2d/dut/tg_logic/out_tg
/tb_tg_top_2d/dut/tg_logic/input
/tb_tg_top_2d/dut/tg_logic/output
/tb_tg_top_2d/dut/tg_sum/clk
/tb_tg_top_2d/dut/tg_sum/rst
/tb_tg_top_2d/dut/tg_sum/input
/tb_tg_top_2d/dut/tg_sum/finished
/tb_tg_top_2d/dut/tg_sum/output_hx
/tb_tg_top_2d/dut/tg_sum/output_hy
/tb_tg_top_2d/dut/tg_sum/output_hz
/tb_tg_top_2d/dut/tg_sum/fsm_state
/tb_tg_top_2d/dut/tg_sum/tg_input
Now 120 ns
Cursor 1 42 ns

Figure 7.5: Testbench of the top entity - first plane

As we can see in the fig. 7.5, in the first 3 clock cycles we see the input m_x, my, mz (*tb_tg_top_2d/dut/tg_logic/input* signal) of each of the 4 cells of the first plane.
In the next 9 clock cycles we have all the 9 values of the **TG** matrix (*tb_tg_top_2d/dut/tg_logic/input* signal). When we have the required input we can compute the output value h_x, h_y, h_z of each cell (in the signal *tb_tg_top_2d/dut/tg_logic/output* we have h_x at 23 ns, h_y at 29 ns, h_z at 35). To better understand the FSM see the testbench of a single cell (7.4.1), the only difference is that now we have 4 cells (2x2 matrix) computing in parallel.
When we have all the cells h_x, h_y, h_z values, we can compute the H_x, H_y, H_z of the plane. In the figure 7.5, at 25 ns we have the H_x value of the first plane which is given by the sum of the h_x values computed by the cells (32+24+54+0=110). Same

thing for H_y and H_z.
Without doing any more tedious calculations, we can see in fig. 7.6 that we accumulate the H_x, H_y, H_z values of each plane in order to get the final value $\overrightarrow{H}$ of the whole parallelepiped.



```
Messages
/tb_tg_top_2d/dut/tg_logic/clk          1
/tb_tg_top_2d/dut/tg_logic/rst          0
/tb_tg_top_2d/dut/tg_logic/in_tg        001F000200030000
/tb_tg_top_2d/dut/tg_logic/out_tg       0B43000C001B0000
/tb_tg_top_2d/dut/tg_logic/input        {{31} {2}} {{3} {0}}
    (0)                                 {31} {2}
    (1)                                 {3} {0}
/tb_tg_top_2d/dut/tg_logic/output       {{2883} {12}} {{27}
    (0)                                 {2883} {12}
    (1)                                 {27} {0}
/tb_tg_top_2d/dut/tg_sum/clk            1
/tb_tg_top_2d/dut/tg_sum/rst            0
/tb_tg_top_2d/dut/tg_sum/input          0B43000C001B0000
/tb_tg_top_2d/dut/tg_sum/finished       1
/tb_tg_top_2d/dut/tg_sum/output_hx      3326
/tb_tg_top_2d/dut/tg_sum/output_hy      3740
/tb_tg_top_2d/dut/tg_sum/output_hz      4154
/tb_tg_top_2d/dut/tg_sum/fsm_state      stop
/tb_tg_top_2d/dut/tg_sum/tg_input       {{31} {2}} {{3} {0}}
    (0)                                 {31} {2}
    (1)                                 {3} {0}
Now                                     120 ns
Cursor 1                                119 ns
```

Figure 7.6: Testbench of the top entity - whole parallelepiped

7.5 Characteristics of Magnetostatic field calculation 3D

PROCESSING ELEMENTS

1- Which kind of Processing element?
Adder, multiplier

2- Functionality
Addition, multiplications (Evaluatin the multiplication between matrix and vectors. Another function is the accumulator.).

3- Complexity

Area:	$i \cdot j$ cells
	one accumulator
Time:	(12 clock cycles)$\cdot k$

4- Parallelism
Each cell can evaluate their magnetostatic contributions.

5-Reconfigurability
No

6- Programmability
No

7- Need a dedicated memory?
Yes, to store the magnetization values (m_x,m_y,m_z)

8- Relationship with I/O
INPUT: values of the magnetization values, magnetizing tensor of each cells.
OUTPUT: result of the magnetostatic field. algorithm

MEMORY ELEMENTS

1- Need a clever memory LIM?

No, but can be implemented

2- Is there a data search algorithm?

No

3-Interface mechanism with other PE or memories

No

4- Access mechanism

(No memory for this implementation)

5- Hierarchization

(No memory for this implementation)

6- Cache coherency

(No memory for this implementation)

7- Is it a a transactional memory?

(No memory for this implementation)

8- Are there virtualization (paging) mechanisms?

(No memory for this implementation)

ENCODING INFORMATION

1-Which encoding is used?

Binary encoding

CONNECTIONS

1-Packet Exchange Protocol

Directly

2-Timing (asynchronou/synchronous)

Synchronous

3-Are there multiple instances?

Yes

4-Heterogeneity (Local/Distant I/O Connections)

Heterogeneous when storing the values from the Input connections.

5-Are there any buffers?

No.

Chapter 8

Smith Waterman

In this chapter I have to understand how to use the architecture done by a student for his thesis. I tried to explain the theory of his work in just few pages, but for a better understand I suggest to read his thesis [18].

8.1 The scoring model. The BLOSUM62 matrix

When two sequences are compared the question that needs an answer is:
How much they are similar?
The scoring model that we want to consider is based on the use of *Substitutional matrix* and *Gap penalty* in the chosen alignment algorithm.
Substitutional matrix and Gap penalty have been created to evaluate different mutation processes that can occur (as we can see in Fig. 8.1).
With a Substitutional matrix we consider amino acids *matching* and *substitutions.* While with the Gap penalty we consider amino acid *insertions* or *deletions.*

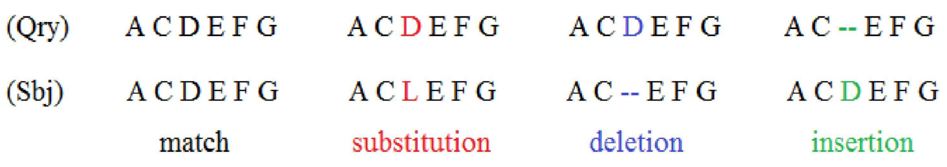
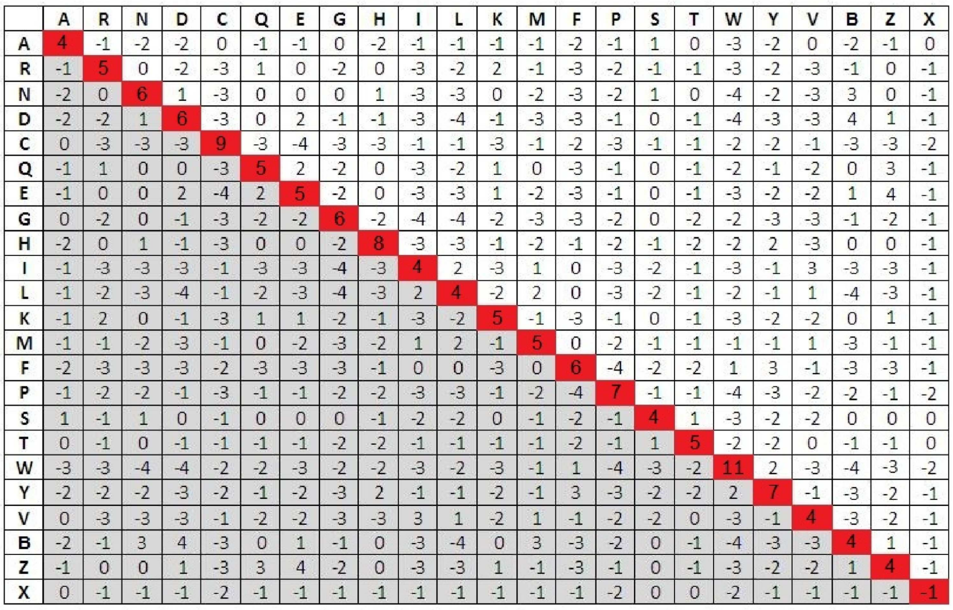


Figure 8.1: Alignments with matching, substitution, insertion and deletion.

The BLOSUM62 matrix was introduced in 1992 by Henikoff & Henikoff [19] to give a score for substitution in the amino acid sequence comparisons. matrix assigns to each pair of AAs (amino acids), a value that indicates the degree of similarity. A positive value in the matrix (Fig. 8.2) means that the two amino acids are similar and they are frequently exchanged each other.



	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	3	-3	-2	0	-1	-4	-3	-3	4	1	-1
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1

Figure 8.2: The BLOSUM62 substitution matrix.

8.2 Smith-Waterman Local alignment method

Smith and Waterman in 1981 described a Local alignment method whose aim is to find common regions between two protein (Qry, Sbj) through calculation of similarity score.
This algorithm can be substantially described in three steps:

1. **Initialization**; first row F(i,0) and first column F(0,j) are initialized to 0. Since it is always better to start a new local alignment instead of extend alignments that have got a negative score.

2. **Score matrix filling**; the score is calculated with the Eq. (8.1) cell by cell.

$$F(i,j) = max \begin{cases} 0 \\ F(i-1,j-1) + s(Qry(i),Sbj(j)) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases} \tag{8.1}$$

   if the first term '0' in the equation is selected as max it means that the previous alignment is ended or not alignment is possible.
   Since this algorithm evaluates local alignments, in this matrix there are not negative score.

3. **output_writing**: works concurrently with the second process. It checks, in each clock cycle, if the stored Sbj id is changed. If the Sbj id is different, this process will write on the output file the n of Sbj id and the associated Maximum Alignment Score.

8.3 Simulation and Test

The testbench of this architecture is divided in 3 phases:

1. **sub_matr_and_gap_load** : configure the systolic array trough the loading of all the storage registers with values coming from the substitutional matrix and gap files.

2. **s_w_computation**: It starts when the configuration process is terminated and reads from the DB file the Sbj_id number and the associated amino acids sequence. In this phase we compute the S-W algorithm. This process is ended when all the database is scanned.

3. **output_writing**: It checks, in each clock cycle, if the stored Sbj id is changed. If the Sbj id is different, this process will write on the output file the n. of Sbj id and the associated Maximum Alignment Score.

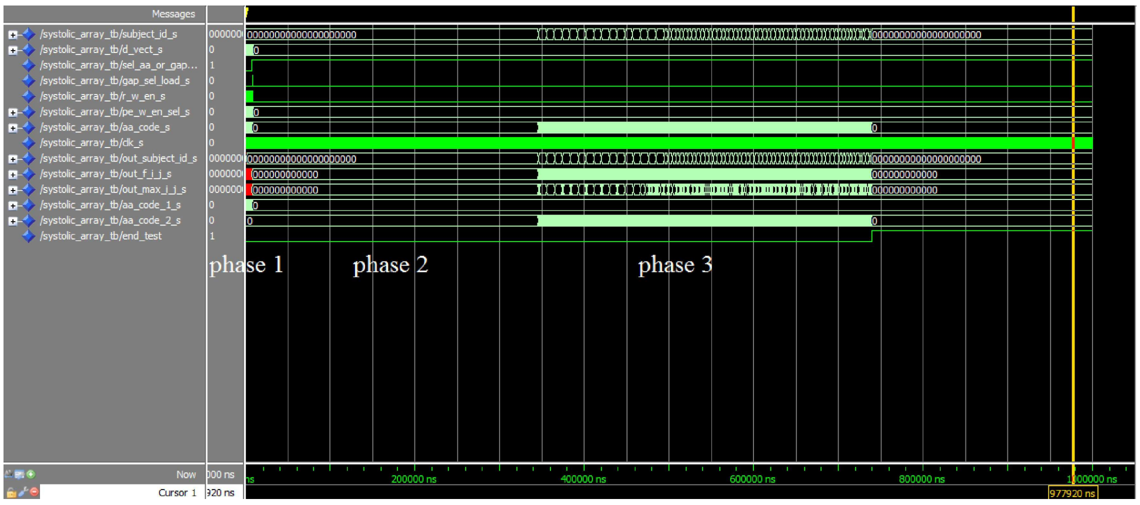

Messages
/systolic_array_tb/subject_id_s
/systolic_array_tb/d_vect_s
/systolic_array_tb/sel_aa_or_gap...
/systolic_array_tb/gap_sel_load_s
/systolic_array_tb/r_w_en_s
/systolic_array_tb/pe_w_en_sel_s
/systolic_array_tb/aa_code_s
/systolic_array_tb/clk_s
/systolic_array_tb/out_subject_id_s
/systolic_array_tb/out_f_i_j_s
/systolic_array_tb/out_max_i_j_s
/systolic_array_tb/aa_code_1_s
/systolic_array_tb/aa_code_2_s
/systolic_array_tb/end_test
phase 1
phase 2
phase 3
Now
Cursor 1
200000 ns
400000 ns
600000 ns
800000 ns
1000000 ns
977920 ns

Figure 8.3: Phases of the testbench

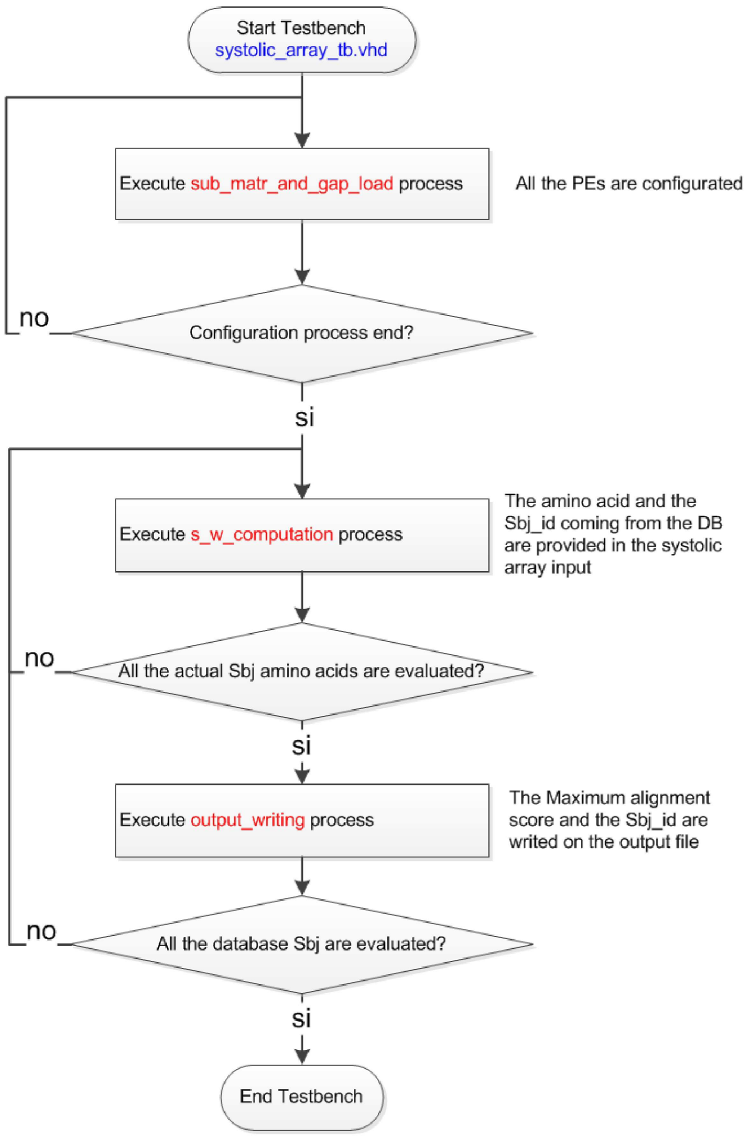

Start Testbench
systolic_array_tb.vhd
Execute sub_matr_and_gap_load process
All the PEs are configurated
no
Configuration process end?
si
Execute s_w_computation process
The amino acid and the Sbj_id coming from the DB are provided in the systolic array input
no
All the actual Sbj amino acids are evaluated?
si
Execute output_writing process
The Maximum alignment score and the Sbj_id are writed on the output file
no
All the database Sbj are evaluated?
si
End Testbench

Figure 8.4: Main flow chart of *systolic_array_tb.vhd* testbench. In blue the testbench name, in red the processes name

8.3.1 Phase 1-sub_matr_and_gap_load

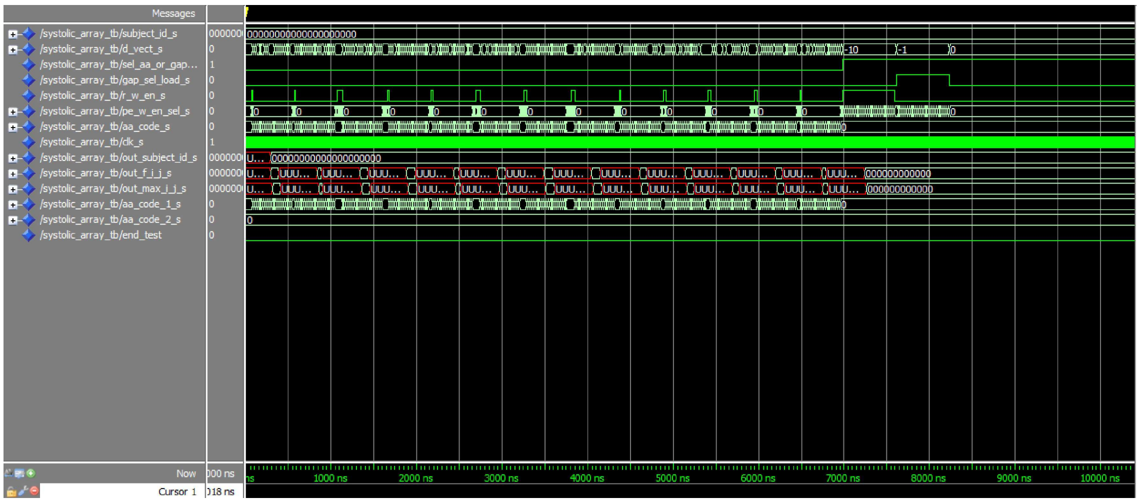

Messages
/systolic_array_tb/subject_id_s
/systolic_array_tb/d_vect_s
/systolic_array_tb/sel_aa_or_gap...
/systolic_array_tb/gap_sel_load_s
/systolic_array_tb/r_w_en_s
/systolic_array_tb/pe_w_en_sel_s
/systolic_array_tb/aa_code_s
/systolic_array_tb/clk_s
/systolic_array_tb/out_subject_id_s
/systolic_array_tb/out_f_i_j_s
/systolic_array_tb/out_max_i_j_s
/systolic_array_tb/aa_code_1_s
/systolic_array_tb/aa_code_2_s
/systolic_array_tb/end_test
Now
Cursor 1
1000 ns
2000 ns
3000 ns
4000 ns
5000 ns
6000 ns
7000 ns
8000 ns
9000 ns
10000 ns

Figure 8.5: Phase 1 of the testbench

In this phase we start to configure all the processing elements.
We need a file where are stored all the Qry AAs in the right format accepted by our testbench:
out_casual_case_config.txt that is composed by:

- 23 rows that correspond to the used amino acids number.
- Each of these rows is associated to an amino acid (see fig. 8.6).
- The first value of each row is the number of PE that must be loaded with the row associated to the amino acid. The following numbers identify the PE id number that must store the Substitutional matrix column.
- If the first value is 0 it means that the amino acid associated to this row is not present into the Qry.

out_casual_case_config.txt

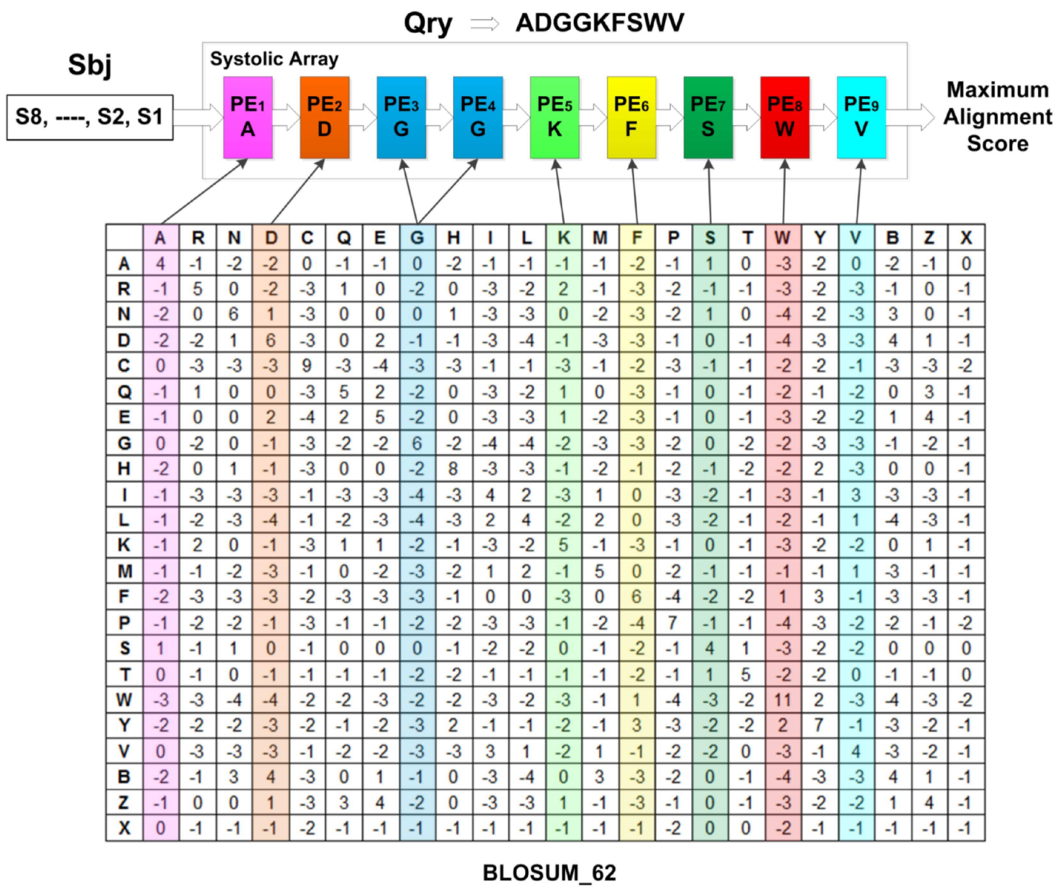
```
1    17
1    19
0
5     5     6    12    22    27
0
1     7
2     2    23
0
0
3    11    26    28
4     3    15    20    25
4     4     9    10    13
2    18    29
0
0
1    21
3     1    24    30
0
2    14    16
1     8
0
0
0
```

we can see that in the first rows it's written "1 17", that means that we need to store the similarity values (fig. 8.2) of the first amino acid ('A') to the PE whose id is 17.

In other words we need to store in the 17-PE the values of the first column (the pink one in fig. 8.7), as we can see also in the simulation (fig. 8.8)

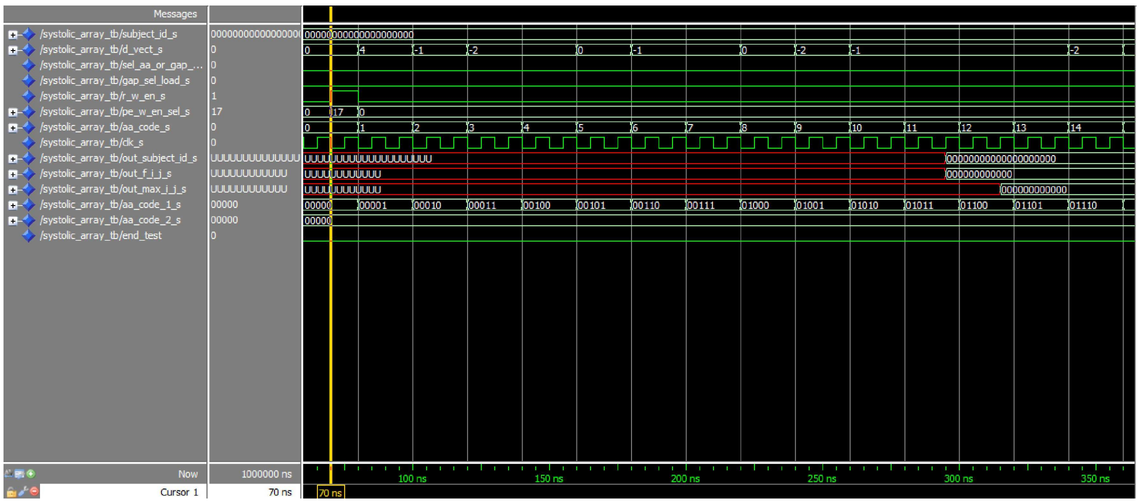
Amino Acid	**Identification Code**	
	(decimal)	**(binary)**
A	1	00001
R	2	00010
N	3	00011
D	4	00100
C, U	5	00101
Q	6	00110
E	7	00111
G	8	01000
H	9	01001
I	10	01010
L	11	01011
K	12	01100
M	13	01101
F	14	01110
P	15	01111
S	16	10000
T	17	10001
W	18	10010
Y	19	10011
V	20	10100
B	21	10101
Z	22	10110
X	23	10111

Figure 8.6: Amino acid encoding



	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	3	-3	-2	0	-1	-4	-3	-3	4	1	-1
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1

Figure 8.7: S-W systolic array initialized with the Substitutional matrix columns: in each PE of the systolic array is stored the column that corresponds to the associated Qry amino acid.



```
Messages
/systolic_array_tb/subject_id_s        0000000000000000(
/systolic_array_tb/d_vect_s            0
/systolic_array_tb/sel_aa_or_gap_...   0
/systolic_array_tb/gap_sel_load_s      0
/systolic_array_tb/r_w_en_s            1
/systolic_array_tb/pe_w_en_sel_s       17
/systolic_array_tb/aa_code_s           0
/systolic_array_tb/clk_s               0
/systolic_array_tb/out_subject_id_s    UUUUUUUUUUUUUU
/systolic_array_tb/out_f_i_j_s         UUUUUUUUUUUU
/systolic_array_tb/out_max_i_j_s       UUUUUUUUUUUU
/systolic_array_tb/aa_code_1_s         00000
/systolic_array_tb/aa_code_2_s         00000
/systolic_array_tb/end_test            0

Now        1000000 ns
Cursor 1   70 ns
```

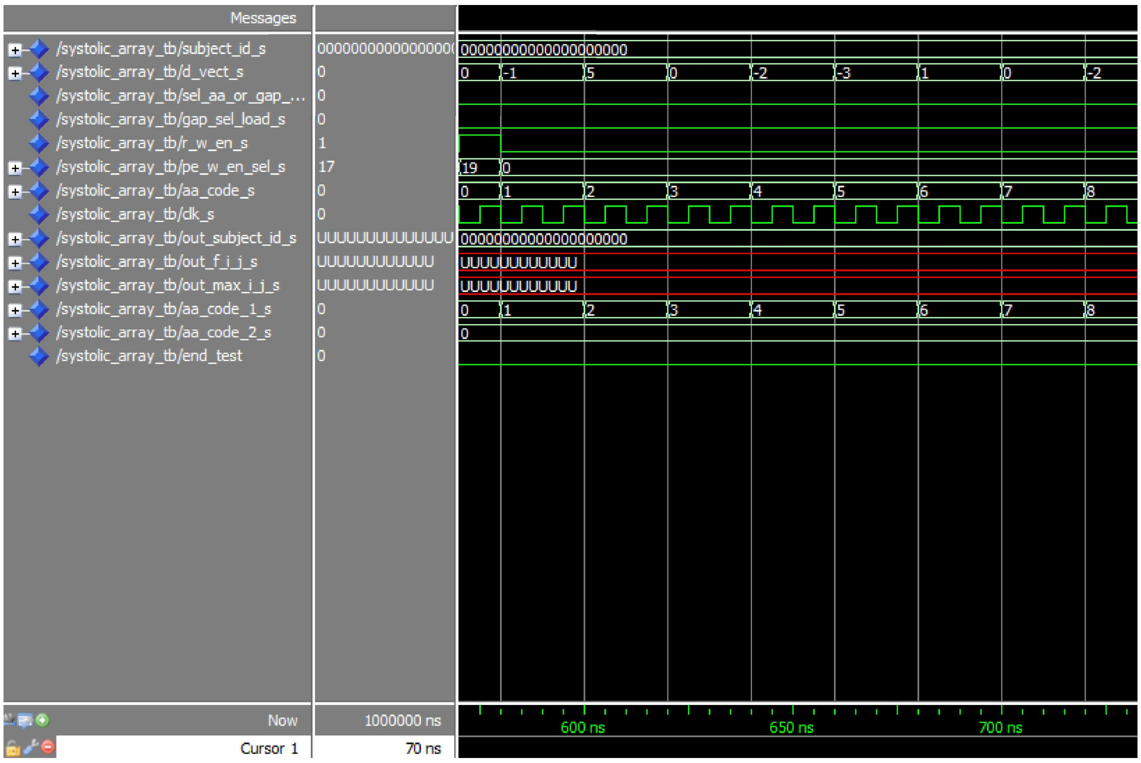
Figure 8.8: Loading into the 17-PE the Substitutional matrix column associated to the amino acid 'A'

In the simulation we select the 17-PE ('*r_w_en_sel_s*'=1 and '*pe_w_en_sel_s*'=17 @ 70 ns) then we load the substitutional matrix column associated to the amino acid 'A' (the pink one in fig. 8.7).

d_vect_s	4	-1	-2	-2	0	-1	-1	...
aa_code_s	1	2	3	4	5	6	7	...
Amino Acid	A	R	N	D	C	Q	E	...

Loading for 17-PE

The second row of the file *out_casual_case_config.txt* (as seen in section 8.3.1 has "1   19", so we do the same thing as before where we store the second column of the BLOSUM62 matrix (fig. 8.7).



```
Messages
/systolic_array_tb/subject_id_s        0000000000000000   00000000000000000000
/systolic_array_tb/d_vect_s            0                  0  -1  5  0  -2  -3  1  0  -2
/systolic_array_tb/sel_aa_or_gap_...   0
/systolic_array_tb/gap_sel_load_s      0
/systolic_array_tb/r_w_en_s            1
/systolic_array_tb/pe_w_en_sel_s       17                 19  0
/systolic_array_tb/aa_code_s           0                  0  1  2  3  4  5  6  7  8
/systolic_array_tb/clk_s               0
/systolic_array_tb/out_subject_id_s    UUUUUUUUUUUUUU     00000000000000000000
/systolic_array_tb/out_f_i_j_s         UUUUUUUUUUUU       UUUUUUUUUUUU
/systolic_array_tb/out_max_i_j_s       UUUUUUUUUUUU       UUUUUUUUUUUU
/systolic_array_tb/aa_code_1_s         0                  0  1  2  3  4  5  6  7  8
/systolic_array_tb/aa_code_2_s         0                  0
/systolic_array_tb/end_test            0

Now        1000000 ns      600 ns      650 ns      700 ns
Cursor 1   70 ns
```

Figure 8.9: Loading into the 19-PE the Substitutional matrix column associated to the amino acid 'R'

d_vect_s	-1	5	0	-2	-3	1	0	...
aa_code_s	1	2	3	4	5	6	7	...
Amino Acid	A	R	N	D	C	Q	E	...

Loading for 19-PE

The third row has only the value ’0’, so no PE has to be configured for the third amino acid ’N’.
The fourth row has ”5 5 6 12 22 27”. Therefore we need to configure five PEs (i.e. 5, 6, 12, 22, 27) with the values associated to the Amino Acid ’D’ (orange column in fig. 8.7) as shown in fig. 8.10.

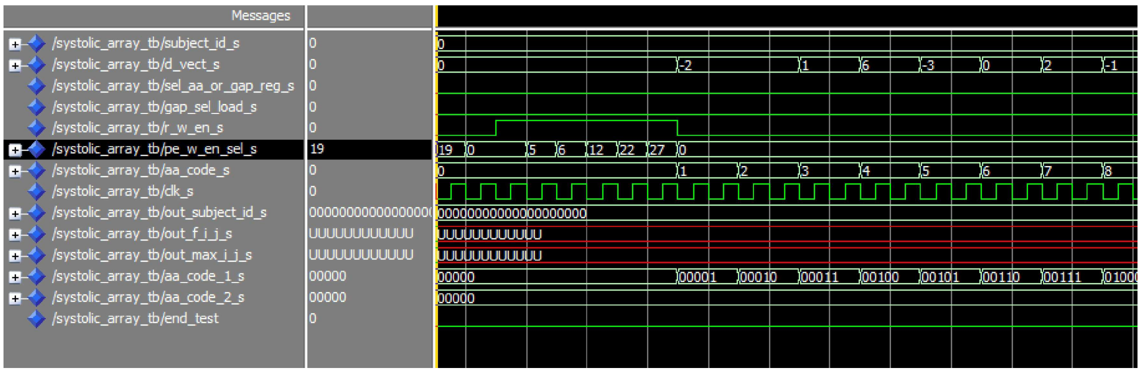

Messages
/systolic_array_tb/subject_id_s 0
/systolic_array_tb/d_vect_s 0
/systolic_array_tb/sel_aa_or_gap_reg_s 0
/systolic_array_tb/gap_sel_load_s 0
/systolic_array_tb/r_w_en_s 0
/systolic_array_tb/pe_w_en_sel_s 19
/systolic_array_tb/aa_code_s 0
/systolic_array_tb/clk_s 0
/systolic_array_tb/out_subject_id_s 0000000000000000
/systolic_array_tb/out_f_i_j_s UUUUUUUUUUUU
/systolic_array_tb/out_max_i_j_s UUUUUUUUUUUU
/systolic_array_tb/aa_code_1_s 00000
/systolic_array_tb/aa_code_2_s 00000
/systolic_array_tb/end_test 0
0
0 -2 1 6 -3 0 2 -1
19 0 5 6 12 22 27 0
0 1 2 3 4 5 6 7 8
00000000000000000000
UUUUUUUUUUUU
UUUUUUUUUUUU
00000 00001 00010 00011 00100 00101 00110 00111 0100
00000

Figure 8.10: Loading into the 5, 6, 12, 22, 27-PEs the Substitutional matrix column associated to the amino acid ’D’

d_vect_s	-2	-2	1	6	-3	0	2	...
aa_code_s	1	2	3	4	5	6	7	...
Amino Acid	A	R	N	D	C	Q	E	...

Loading for 5, 6, 12, 22, 27-PEs

We keep this process until we reach the last PE to be configured, which is the number 8 (row 20).
At the end of this configuration (fig. 8.11), we need to store the two gap values (written in the file *gap_open_ext.txt*) to all the PEs, the first value is the Open gap (fig. 8.12) while the second one is the Extension gap (fig. 8.13).

The signal *gap_sel_load* is high only when storing the extension gap, while *sel_aa_or_gap_reg* is at ’1’ only when storing both the penalties (open and extension)

8.3.2 Phase 2- s_w_computation

When the configuration phase ends, we can start the second process called *s_w_computation*already explained in section 8.3 and showed in fig. 8.3. Here we scan the DataBase file

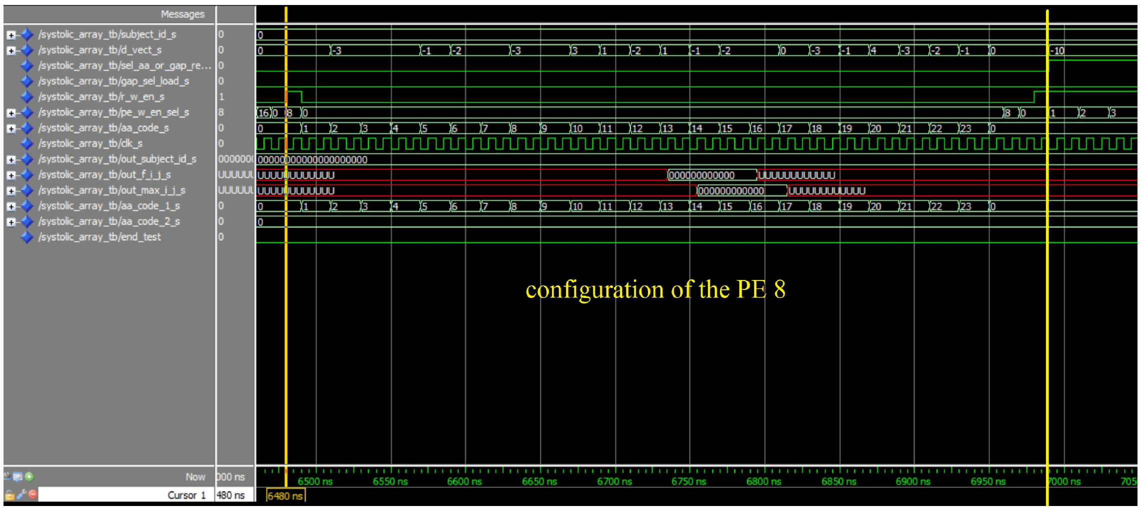

Messages
/systolic_array_tb/subject_id_s
/systolic_array_tb/d_vect_s
/systolic_array_tb/sel_aa_or_gap_re...
/systolic_array_tb/gap_sel_load_s
/systolic_array_tb/r_w_en_s
/systolic_array_tb/pe_w_en_sel_s
/systolic_array_tb/aa_code_s
/systolic_array_tb/clk_s
/systolic_array_tb/out_subject_id_s
/systolic_array_tb/out_f_i_j_s
/systolic_array_tb/out_max_i_j_s
/systolic_array_tb/aa_code_1_s
/systolic_array_tb/aa_code_2_s
/systolic_array_tb/end_test
configuration of the PE 8
Now
Cursor 1
6480 ns

Figure 8.11: Loading into the 8-PEs the Substitutional matrix column associated to the amino acid 'V'. On the right side, we see that we start to load the open gap penalty.

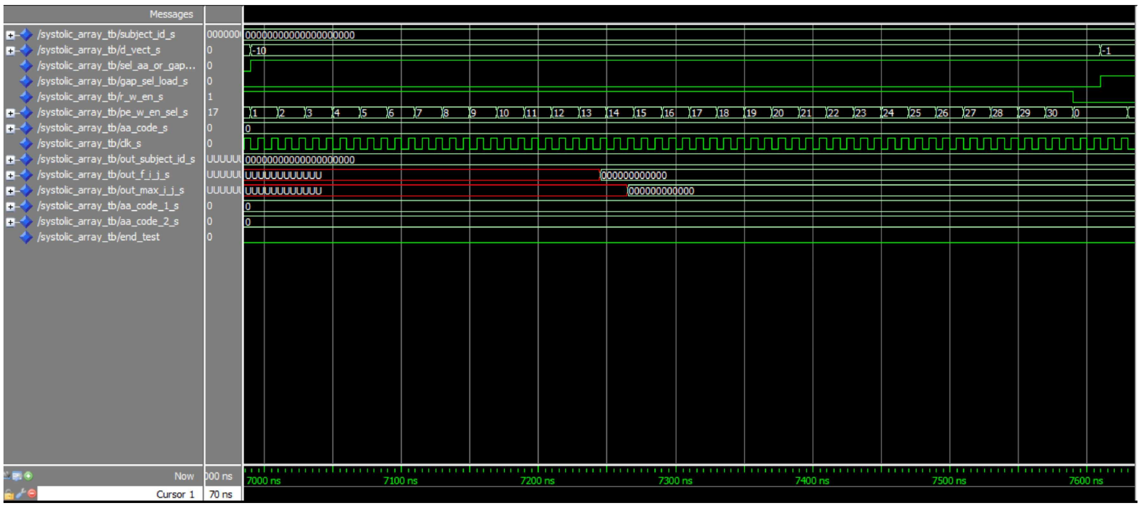
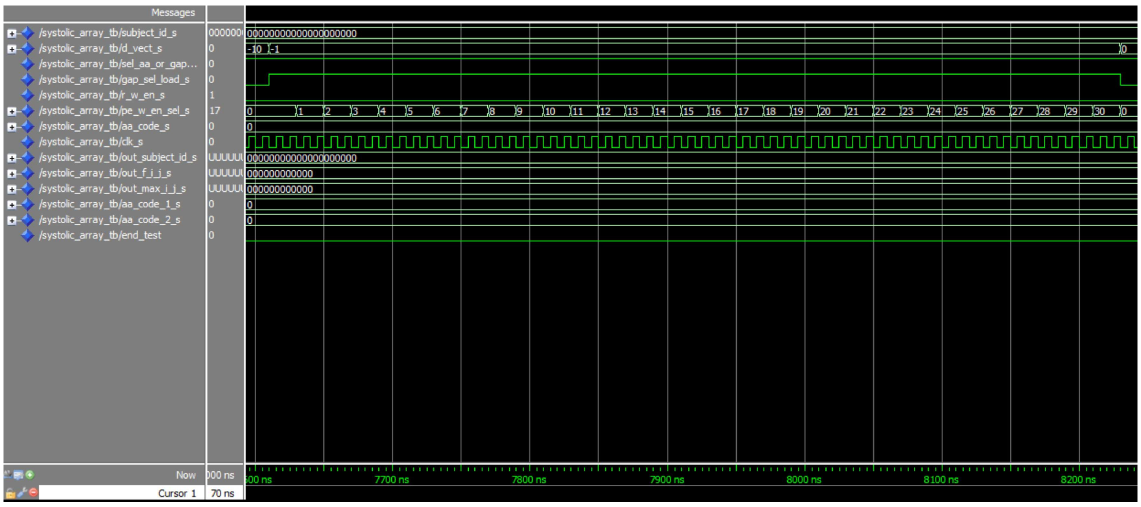

Messages
/systolic_array_tb/subject_id_s
/systolic_array_tb/d_vect_s
/systolic_array_tb/sel_aa_or_gap...
/systolic_array_tb/gap_sel_load_s
/systolic_array_tb/r_w_en_s
/systolic_array_tb/pe_w_en_sel_s
/systolic_array_tb/aa_code_s
/systolic_array_tb/clk_s
/systolic_array_tb/out_subject_id_s
/systolic_array_tb/out_f_i_j_s
/systolic_array_tb/out_max_i_j_s
/systolic_array_tb/aa_code_1_s
/systolic_array_tb/aa_code_2_s
/systolic_array_tb/end_test
Now
Cursor 1

Figure 8.12: Loading into all PEs the open gap penalty which value is -10

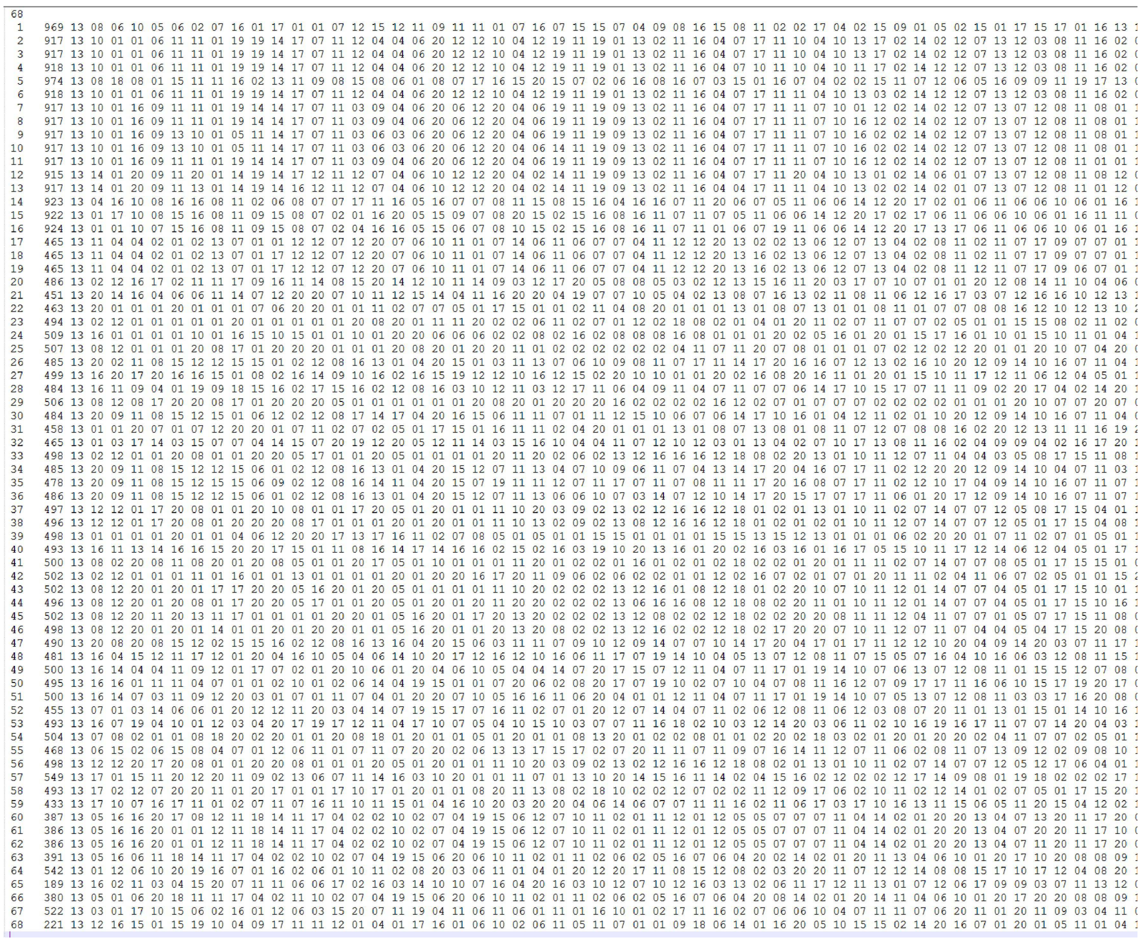
DB_num_casual_case_2.txt which is composed by: -In the first row of the file is written the number of sequences that compose this database.
- In this file all the letters are converted in number using, for the conversion, a code that doesn't take into account the amino acids frequency (fig. 8.6). The encoding code is written directly into the program.
- The protein id number is the first value of the row (in this case 68).



```
Messages
/systolic_array_tb/subject_id_s        000000(   00000000000000000000
/systolic_array_tb/d_vect_s            0         -10 -1                                                        0
/systolic_array_tb/sel_aa_or_gap...    0
/systolic_array_tb/gap_sel_load_s      0
/systolic_array_tb/r_w_en_s            1
/systolic_array_tb/pe_w_en_sel_s       17        0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 0
/systolic_array_tb/aa_code_s           0         0
/systolic_array_tb/clk_s               0
/systolic_array_tb/out_subject_id_s    UUUUU(    00000000000000000000
/systolic_array_tb/out_f_i_j_s         UUUUU(    000000000000
/systolic_array_tb/out_max_i_j_s       UUUUU(    000000000000
/systolic_array_tb/aa_code_1_s         0         0
/systolic_array_tb/aa_code_2_s         0         0
/systolic_array_tb/end_test            0
Now       )00 ns
Cursor 1  70 ns
```

Figure 8.13: Loading into all PEs the extension gap penalty which value is -1

- The number of AAs that compose this protein is the second value in the same row.
- All the other numbers are the encoded amino acids.



```
68
 1   969 13 08 06 10 05 06 02 07 16 01 17 01 01 07 12 15 12 11 09 11 11 01 07 16 07 15 15 07 04 09 08 16 15 08 11 02 02 17 04 02 15 09 01 05 02 15 01 17 15 17 01 16 13
 2   917 13 10 01 01 06 11 11 01 19 19 14 17 07 11 12 04 04 06 20 12 12 10 04 12 19 11 19 01 13 02 11 16 04 07 17 11 10 04 10 13 17 02 14 02 12 07 13 12 03 08 11 16 02
 3   917 13 10 01 01 06 11 11 01 19 19 14 17 07 11 12 04 04 06 20 12 12 10 04 12 19 11 19 01 13 02 11 16 04 07 17 11 10 04 10 13 17 02 14 02 12 07 13 12 03 08 11 16 02
 4   918 13 10 01 01 06 11 11 01 19 19 14 17 07 11 12 04 04 06 20 12 12 10 04 12 19 11 19 01 13 02 11 16 04 07 10 11 10 04 10 11 17 02 14 12 12 07 13 12 03 08 11 16 02
 5   974 13 08 18 08 01 15 11 11 16 02 13 11 09 08 15 08 06 01 08 07 17 16 15 20 15 07 02 06 16 08 16 07 03 15 01 16 07 04 02 02 15 11 07 12 06 05 16 09 09 11 19 17 13
 6   918 13 10 01 01 06 11 11 01 19 19 14 17 07 11 12 04 04 06 20 12 12 10 04 12 19 11 19 01 13 02 11 16 04 07 17 11 11 04 10 13 03 02 14 12 12 07 13 12 03 08 11 16 02
 7   917 13 10 01 16 09 11 11 01 19 14 14 17 07 11 03 09 04 06 20 06 12 20 04 06 19 11 19 09 13 02 11 16 04 07 17 11 11 07 10 01 12 02 14 02 12 07 13 07 12 08 11 08 01
 8   917 13 10 01 16 09 11 11 01 19 14 14 17 07 11 03 09 04 06 20 06 12 20 04 06 19 11 19 09 13 02 11 16 04 07 17 11 11 07 10 16 12 02 14 02 12 07 13 07 12 08 11 08 01
 9   917 13 10 01 16 09 13 10 01 05 11 14 17 07 11 03 06 03 06 20 06 12 20 04 06 19 11 19 09 13 02 11 16 04 07 17 11 11 07 10 16 02 02 14 02 12 07 13 07 12 08 11 08 01
10   917 13 10 01 16 09 13 10 01 05 11 14 17 07 11 03 06 03 06 20 06 12 20 04 06 14 11 19 09 13 02 11 16 04 07 17 11 11 07 10 16 02 02 14 02 12 07 13 07 12 08 11 08 01
11   917 13 10 01 16 09 11 11 01 19 14 14 17 07 11 03 09 04 06 20 06 12 20 04 06 19 11 19 09 13 02 11 16 04 07 17 11 11 07 10 16 12 02 14 02 12 07 13 07 12 08 11 01 01
12   915 13 14 01 20 09 11 20 01 14 19 14 17 12 11 12 07 04 06 10 12 12 20 04 02 14 11 19 09 13 02 11 16 04 07 17 11 20 04 10 13 01 02 14 06 01 07 13 07 12 08 11 08 12
13   917 13 14 01 20 09 11 13 01 14 19 14 16 12 11 12 07 04 06 10 12 12 20 04 02 14 11 19 09 13 02 11 16 04 04 17 11 11 04 10 13 02 02 14 02 01 07 13 07 12 08 11 01 12
14   923 13 04 16 10 08 16 16 08 11 02 06 08 07 07 17 11 16 05 16 07 07 08 11 15 08 15 16 04 16 16 07 11 20 06 07 05 11 06 06 14 12 20 17 02 01 06 11 06 06 10 06 01 16
15   922 13 01 17 10 08 15 16 08 11 09 15 08 07 02 01 16 20 05 15 09 07 08 20 15 02 15 16 08 16 11 07 11 07 05 11 06 06 14 12 20 17 02 17 06 11 06 06 10 06 01 16 11 11
16   924 13 01 01 10 07 15 16 08 11 09 15 08 07 02 04 16 16 05 15 06 07 08 10 15 02 15 16 08 16 11 07 11 01 06 07 19 11 06 06 14 12 20 17 13 17 06 11 06 06 10 06 01 16
17   465 13 11 04 04 02 01 02 13 07 01 01 12 12 07 12 20 07 06 10 11 01 07 14 06 11 06 07 07 04 11 12 12 20 13 02 02 13 06 12 07 13 04 02 08 11 02 11 07 17 09 07 07 01
18   465 13 11 04 04 02 01 02 13 07 01 17 12 12 07 12 20 07 06 10 11 01 07 14 06 11 06 07 07 04 11 12 12 20 13 16 02 13 06 12 07 13 04 02 08 11 02 11 07 17 09 07 07 01
19   465 13 11 04 04 02 01 02 13 07 01 17 12 12 07 12 20 07 06 10 11 01 07 14 06 11 06 07 07 04 11 12 12 20 13 16 02 13 06 12 07 13 04 02 08 11 12 11 07 17 09 06 07 01
20   486 13 02 12 16 17 02 11 11 17 09 16 11 14 08 15 20 14 12 10 11 14 09 03 12 17 20 05 08 08 05 03 02 12 13 15 16 11 20 03 17 07 10 07 01 01 20 12 08 14 11 10 04 06
21   451 13 20 14 16 04 06 06 11 14 07 12 20 20 07 10 11 12 15 14 04 11 16 20 20 04 19 07 07 10 05 04 02 13 08 07 16 13 02 11 08 11 06 12 16 17 03 07 12 16 16 10 12 13
22   463 13 20 01 01 01 20 01 01 01 07 06 20 20 01 01 11 02 07 07 05 01 17 15 01 01 02 11 04 08 20 01 01 01 13 01 08 07 13 01 01 08 11 01 07 07 08 08 16 12 10 12 13 10
23   494 13 02 12 01 01 01 01 01 20 01 01 01 01 01 20 08 20 01 11 11 20 02 02 06 11 02 07 01 12 02 18 08 02 01 04 01 20 11 02 07 11 07 07 02 05 01 01 15 15 08 02 11 02
24   509 13 16 01 01 01 01 10 01 16 15 10 15 01 01 10 01 20 20 06 06 06 02 02 08 02 16 02 08 08 08 16 08 01 01 01 20 02 05 16 01 20 01 15 17 16 01 10 01 15 10 11 01 04
25   507 13 08 12 01 01 01 20 08 17 01 20 20 20 01 01 01 20 08 20 01 20 20 11 01 02 02 02 02 02 02 04 11 07 11 20 07 08 01 01 01 07 02 12 02 12 20 01 01 20 10 07 04 20
26   485 13 20 02 11 08 15 12 12 15 15 01 02 12 08 16 13 01 04 20 15 01 03 11 13 07 06 10 09 08 11 07 17 11 14 17 20 16 16 07 12 13 02 16 10 20 12 09 14 10 16 07 11 04
27   499 13 16 20 17 20 16 16 15 01 08 02 16 14 09 10 16 02 16 15 19 12 12 10 16 12 15 02 20 10 10 01 01 20 02 16 08 20 16 11 01 20 01 15 10 11 17 12 11 06 12 04 05 01
28   484 13 16 11 09 04 01 19 09 18 15 16 02 17 15 16 02 12 08 16 03 10 12 11 03 12 17 11 06 04 09 11 04 07 11 07 07 06 14 17 10 15 17 07 11 11 09 02 20 17 04 02 14 20
29   506 13 08 12 08 17 20 20 08 17 01 20 20 20 05 01 01 01 01 01 01 20 08 20 01 20 20 20 16 02 02 02 02 16 12 02 07 01 07 07 07 02 02 02 02 01 01 01 20 10 07 07 20 07
30   484 13 20 09 11 08 15 12 15 01 06 12 02 12 08 17 14 17 04 20 16 15 06 11 11 07 01 11 12 15 10 06 07 06 14 17 10 16 01 04 12 11 02 01 10 20 12 09 14 10 16 07 11 04
31   458 13 01 01 20 07 01 07 12 20 20 01 07 11 02 07 02 05 01 17 15 01 16 11 11 02 04 20 01 01 01 13 01 08 07 13 08 01 08 11 07 12 07 08 08 16 02 20 12 13 11 11 16 19
32   465 13 01 03 17 14 03 15 07 07 04 14 15 07 20 19 12 20 05 12 11 14 03 15 16 10 04 04 11 07 12 10 12 03 01 13 04 02 07 10 17 13 08 11 16 02 04 09 09 04 02 16 17 20
33   498 13 02 12 01 01 20 08 01 01 20 20 05 17 01 01 20 05 01 01 01 01 20 11 20 02 06 02 13 12 16 16 16 12 18 08 02 20 13 01 10 11 12 07 11 04 04 03 05 08 17 15 11 08
34   485 13 20 09 11 08 15 12 12 15 06 01 02 12 08 16 13 01 04 20 15 12 07 11 13 04 07 10 09 06 11 07 04 13 14 17 20 04 16 07 17 11 02 12 20 20 12 09 14 10 04 07 11 03
35   478 13 20 09 11 08 15 12 15 15 06 09 02 12 08 16 14 11 04 20 15 07 19 11 11 12 07 11 17 07 11 07 08 11 11 17 20 16 08 07 17 11 02 12 10 17 04 09 14 10 16 07 11 07
36   486 13 20 09 11 08 15 12 12 15 06 01 02 12 08 16 13 01 04 20 15 12 07 11 13 06 06 10 07 03 14 07 12 10 14 17 20 15 17 07 17 11 06 01 20 17 12 09 14 10 16 07 11 07
37   497 13 12 12 01 17 20 08 01 01 20 10 08 01 01 17 20 05 01 20 01 01 11 10 20 03 09 02 13 02 12 16 16 12 18 01 02 01 13 01 10 11 02 07 14 07 07 12 05 08 17 15 04 01
38   496 13 12 12 01 17 20 08 01 20 20 20 08 17 01 01 01 20 01 20 01 01 11 10 13 02 09 02 13 08 12 16 16 12 18 01 02 01 02 01 10 11 12 07 14 07 07 12 05 01 17 15 04 08
39   498 13 01 01 01 01 20 01 01 04 06 12 20 20 17 13 17 16 11 02 07 08 05 01 05 01 01 15 15 01 01 01 01 15 15 13 15 12 13 01 01 01 06 02 20 20 01 07 11 02 07 01 05 01
40   493 13 16 11 13 14 16 16 15 20 20 17 15 01 11 08 16 14 17 14 16 16 02 15 02 16 03 19 10 20 13 16 01 20 02 16 03 16 01 16 17 05 15 10 11 17 12 14 06 12 04 05 01 17
41   500 13 08 02 20 08 11 08 20 01 20 08 05 01 01 20 17 05 01 10 01 01 01 11 20 01 02 02 01 16 01 02 01 02 18 02 02 01 20 01 11 11 02 07 14 07 07 08 05 01 17 15 15 01
42   502 13 02 12 01 01 01 11 01 16 01 01 13 01 01 01 01 20 01 20 20 16 17 20 11 09 06 02 06 02 02 01 01 12 02 16 07 02 01 07 01 20 11 11 02 04 11 06 07 02 05 01 01 15
43   502 13 08 12 20 01 20 01 17 17 20 20 05 16 20 01 20 05 01 01 01 01 11 10 20 02 02 02 13 12 16 01 08 12 18 01 02 20 10 07 10 11 12 01 14 07 07 04 05 01 17 15 10 01
44   496 13 08 12 20 01 20 08 01 17 20 20 05 17 01 01 20 05 01 20 01 20 11 20 20 02 02 02 13 06 16 16 08 12 18 08 02 20 11 01 10 11 12 01 14 07 07 04 05 01 17 15 10 16
45   502 13 08 12 20 11 20 13 11 17 01 01 01 01 20 20 01 05 16 20 01 17 20 13 20 02 02 02 13 12 08 02 02 12 18 02 02 20 20 08 11 11 12 04 11 07 07 01 05 07 17 15 11 08
46   498 13 08 12 20 01 20 01 14 01 01 20 01 20 20 01 01 05 16 20 01 01 20 13 20 08 02 02 13 12 16 02 02 12 18 02 17 20 20 07 10 11 12 07 11 07 04 04 05 04 17 15 20 08
47   490 13 20 08 20 08 15 12 02 15 15 16 02 12 08 16 13 16 04 20 15 06 03 11 11 07 09 10 12 09 14 07 07 10 14 17 20 04 17 01 17 11 12 12 10 20 04 09 14 20 03 07 11 17
48   481 13 16 04 15 12 11 17 12 01 20 04 16 10 05 04 06 14 10 20 17 12 16 12 10 16 06 11 17 07 19 14 10 04 05 13 07 12 08 11 07 15 05 07 16 04 10 16 06 03 12 08 11 15
49   500 13 16 14 04 04 11 09 12 01 17 07 02 01 20 10 06 01 20 04 06 10 05 04 04 14 07 20 17 15 07 12 11 04 07 11 17 01 19 14 10 07 06 13 07 12 08 11 01 15 15 12 07 08
50   495 13 16 16 01 11 11 04 07 01 01 02 10 01 02 06 14 04 19 15 01 01 07 20 06 02 08 20 17 07 19 10 02 07 10 04 07 08 11 16 12 07 09 17 17 11 16 06 10 15 17 19 20 17
51   500 13 16 14 07 03 11 09 12 20 03 01 07 01 11 07 04 01 20 20 07 10 05 16 16 11 06 20 04 01 01 12 11 04 07 11 17 01 19 14 10 07 05 13 07 12 08 11 03 03 17 16 20 08
52   455 13 07 01 03 14 06 06 01 20 12 12 11 20 03 04 14 07 19 15 17 07 16 11 02 07 01 20 12 07 14 04 07 11 02 06 12 08 11 06 12 03 08 07 20 11 01 13 01 15 01 14 10 16
53   493 13 16 07 19 04 10 01 12 03 04 20 17 19 17 12 11 04 17 10 07 05 04 10 15 10 03 07 07 11 16 18 02 10 03 12 14 20 03 06 11 02 10 16 19 16 17 11 07 07 14 20 04 03
54   504 13 07 08 02 01 01 08 18 20 02 20 01 01 20 08 18 01 20 01 01 05 01 20 01 01 08 13 20 01 02 02 08 01 01 02 20 02 18 03 02 01 20 01 20 20 02 04 11 07 07 02 05 01
55   468 13 06 15 02 06 15 08 04 07 01 12 06 11 01 07 11 07 20 20 02 06 13 13 17 15 17 02 07 20 11 11 07 11 09 07 16 14 11 12 07 11 06 02 08 11 07 13 09 12 02 09 08 10
56   498 13 12 12 20 17 20 08 01 01 20 20 08 01 01 01 20 05 01 20 01 01 11 10 20 03 09 02 13 02 12 16 16 12 18 08 02 01 13 01 10 11 02 07 14 07 07 12 05 12 17 06 04 01
57   549 13 17 01 15 11 20 12 20 11 09 02 13 06 07 11 14 16 03 10 20 01 01 11 07 01 13 10 20 14 15 16 11 14 02 04 15 16 02 12 02 02 12 17 14 09 08 01 19 18 02 02 02 17
58   493 13 17 02 12 07 20 20 11 01 20 17 01 01 17 10 17 01 20 01 01 08 20 11 13 08 02 18 10 02 02 12 07 02 02 11 12 09 17 06 02 10 11 02 12 14 01 02 07 05 01 17 15 20
59   433 13 17 10 07 16 17 11 01 02 07 11 07 16 11 10 11 15 01 04 16 10 20 03 20 20 04 06 14 06 07 07 11 11 16 02 11 06 17 03 17 10 16 13 11 15 06 05 11 20 15 04 12 02
60   387 13 05 16 16 20 17 08 12 11 18 14 11 17 04 02 02 10 02 07 04 19 15 06 12 07 10 11 02 01 11 12 01 12 05 05 07 07 07 11 04 14 02 01 20 20 13 04 07 13 20 11 17 20
61   386 13 05 16 16 20 01 01 12 11 18 14 11 17 04 02 02 10 02 07 04 19 15 06 12 07 10 11 02 01 11 12 01 12 05 05 07 07 07 11 04 14 02 01 20 20 13 04 07 20 20 11 17 10
62   386 13 05 16 16 20 01 01 12 11 18 14 11 17 04 02 02 10 02 07 04 19 15 06 12 07 10 11 02 01 11 12 01 12 05 05 07 07 07 11 04 14 02 01 20 20 13 04 07 11 20 11 17 20
63   391 13 05 16 06 11 18 14 11 17 04 02 02 10 02 07 04 19 15 06 20 06 10 11 02 01 11 02 06 02 05 16 07 06 04 20 02 14 02 01 20 11 13 04 06 10 01 20 17 10 20 08 08 09
64   542 13 01 12 06 10 20 19 16 07 01 16 02 06 01 10 11 02 08 20 03 06 11 01 04 01 20 12 20 17 11 08 15 12 08 02 03 20 20 11 07 12 12 14 08 08 15 17 10 17 12 04 08 20
65   189 13 16 02 11 03 04 15 20 07 11 11 06 06 17 02 16 03 14 10 10 07 16 04 20 16 03 10 12 07 10 12 16 03 13 02 06 11 17 12 11 13 01 07 12 06 17 09 09 03 07 11 13 12
66   380 13 05 01 06 20 18 11 11 17 04 02 11 10 02 07 04 19 15 06 20 06 10 11 02 01 11 02 06 02 05 16 07 06 04 20 08 14 02 01 20 14 11 04 06 10 01 20 17 20 20 08 08 09
67   522 13 03 01 17 10 15 06 02 16 01 12 06 03 15 20 07 11 19 04 11 06 11 06 01 11 01 16 10 01 02 17 11 16 02 07 06 06 10 04 07 11 11 07 06 20 11 01 20 11 09 03 04 11
68   221 13 12 16 15 01 15 19 10 04 09 17 11 11 12 01 04 01 17 16 01 06 10 02 06 11 05 11 07 01 01 09 18 06 14 01 16 20 05 10 15 15 02 14 20 16 07 01 20 01 05 11 01 04
```

Figure 8.14: Section of the database file to be loaded in this testbench

8.3.3 Phase 3- output_writing

In the *subject_id_s* and *out_subject_id_s*, we can see the code for the selected protein, and in the *aa_code_s* the proteins associated to such protein (the list of aminoacids is consistent with the database provided in the file *DB_num_casual_case.txt*).

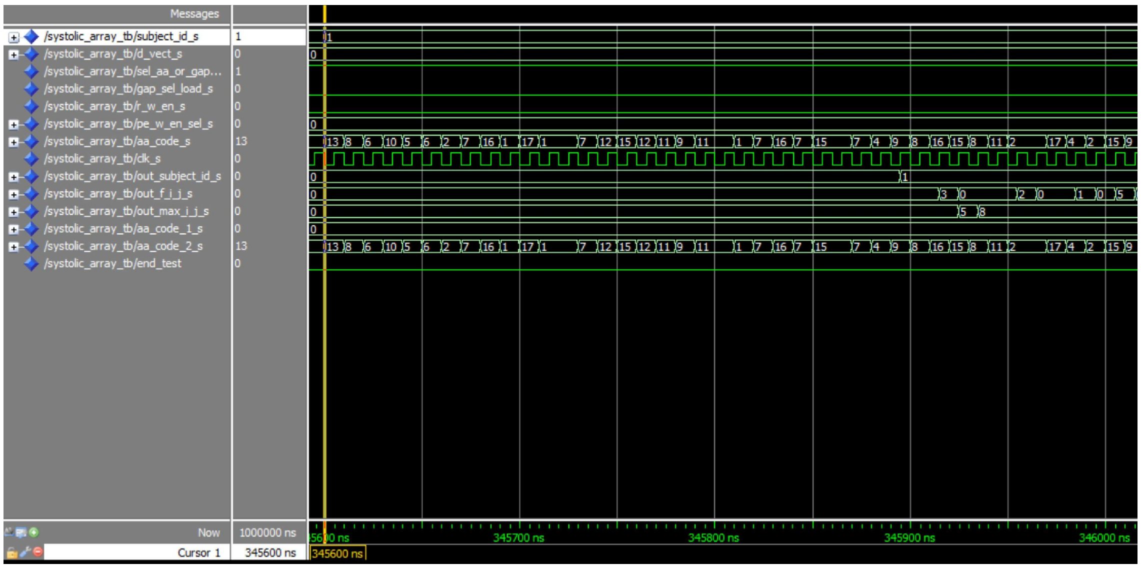

Messages
/systolic_array_tb/subject_id_s 1
/systolic_array_tb/d_vect_s 0
/systolic_array_tb/sel_aa_or_gap... 1
/systolic_array_tb/gap_sel_load_s 0
/systolic_array_tb/r_w_en_s 0
/systolic_array_tb/pe_w_en_sel_s 0
/systolic_array_tb/aa_code_s 13
/systolic_array_tb/clk_s 0
/systolic_array_tb/out_subject_id_s 0
/systolic_array_tb/out_f_i_j_s 0
/systolic_array_tb/out_max_i_j_s 0
/systolic_array_tb/aa_code_1_s 0
/systolic_array_tb/aa_code_2_s 13
/systolic_array_tb/end_test 0
Now 1000000 ns
Cursor 1 345600 ns
345700 ns
345800 ns
345900 ns
346000 ns
345600 ns

Figure 8.15: scanning the list of aminoacid (*aa_code_s*) of the first protein while computing the Maximum Alignment Score

The *out_f_i_j_s* contains the Score matrix value calculated in the last PE (matrix cell F(i,j)) and the *out_max_i_j_s* is the Maximum Alignment Score. Both these value are evaluated and shifted through the systolic array. As expected, we can clearly see the Maximum Alignment Score increasing. In th fig. 8.16 the value increases from an initial value of 5 to 149. When all the process ends, we can write for each protein the Maximum Alignment Score evaluated in *out_id_subj_and_max_i_j.txt* file

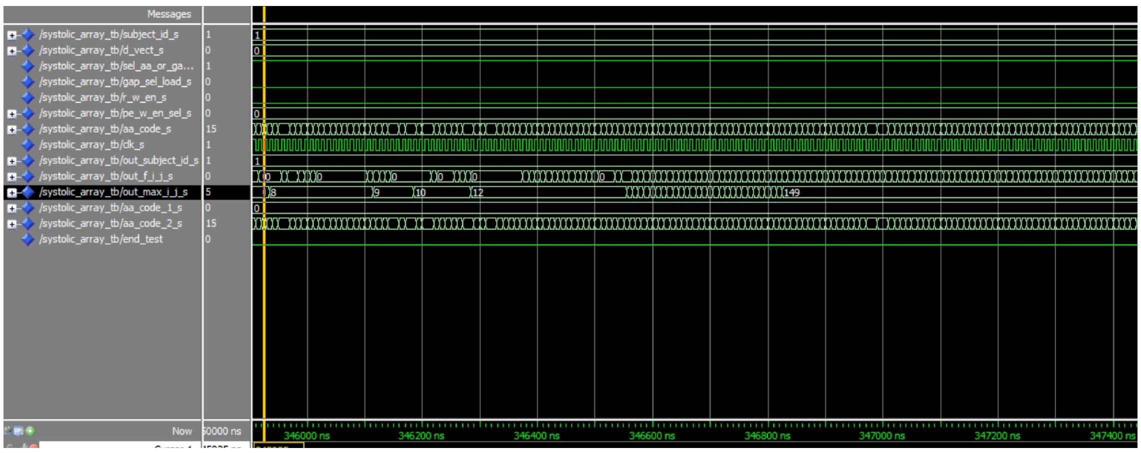
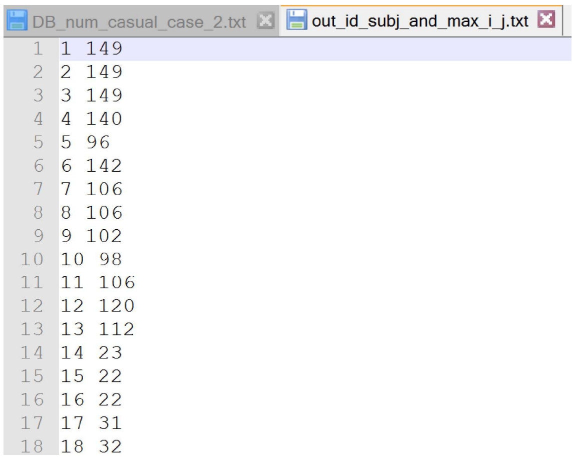

Messages
/systolic_array_tb/subject_id_s 1
/systolic_array_tb/d_vect_s 0
/systolic_array_tb/sel_aa_or_ga... 1
/systolic_array_tb/gap_sel_load_s 0
/systolic_array_tb/r_w_en_s 0
/systolic_array_tb/pe_w_en_sel_s 0
/systolic_array_tb/aa_code_s 15
/systolic_array_tb/clk_s 1
/systolic_array_tb/out_subject_id_s 1
/systolic_array_tb/out_f_i_j_s 0
/systolic_array_tb/out_max_i_j_s 5
/systolic_array_tb/aa_code_1_s 0
/systolic_array_tb/aa_code_2_s 15
/systolic_array_tb/end_test 0
8
9
10
12
149
Now 60000 ns
346000 ns
346200 ns
346400 ns
346600 ns
346800 ns
347000 ns
347200 ns
347400 ns

Figure 8.16: Computing the Maximum Alignment Score (at the end *out_max_i_j_s* = 149) for the first protein



```
1 149
2 149
3 149
4 140
5 96
6 142
7 106
8 106
9 102
10 98
11 106
12 120
13 112
14 23
15 22
16 22
17 31
18 32
```

Figure 8.17: In the first row of *out_id_subj_and_max_i_j.txt* file we can see that the first protein has 149 as Maximum Alignment Score

Chapter 9

Singular Value Decomposition

Given a complex matrix A having m rows and n columns, the matrix product $U\Sigma V^T$ is a singular value decomposition for a given matrix A if

- U and V, respectively, have orthonormal columns.
- Σ has nonnegative elements on its principal diagonal and zeros elsewhere.
- $A = U\Sigma V^T$

One application of the SVD is data compression. Consider some matrix A with rank five hundred; that is, the columns of this matrix span a 500-dimensional space. Encoding this matrix on a computer is going to take quite a lot of memory! We might be interested in approximating this matrix with one of lower rank. How close can we get to this matrix if we only approximate it as a matrix with rank one hundred, so that we only have to store a hundred columns?
It turns out that you can prove that taking the n largest singular values A, replacing the rest with zero (to form Σ), and recomputing $U\Sigma V^T$ gives you the provably best n-rank approximation to the matrix. [21]

It means that we can take a list of n unique vectors, and approximate them as a linear combination of k unique vectors.[22]

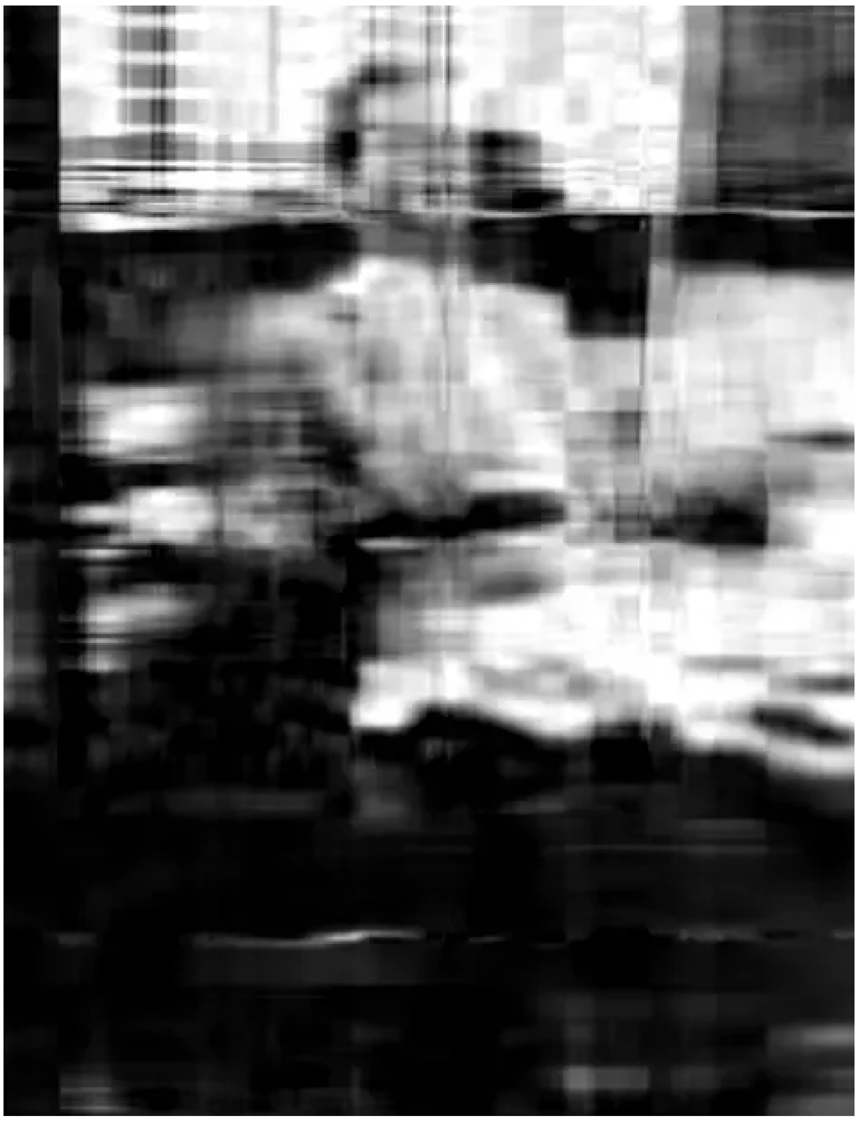


Figure 9.1: Image made of k=10 unique row vectors

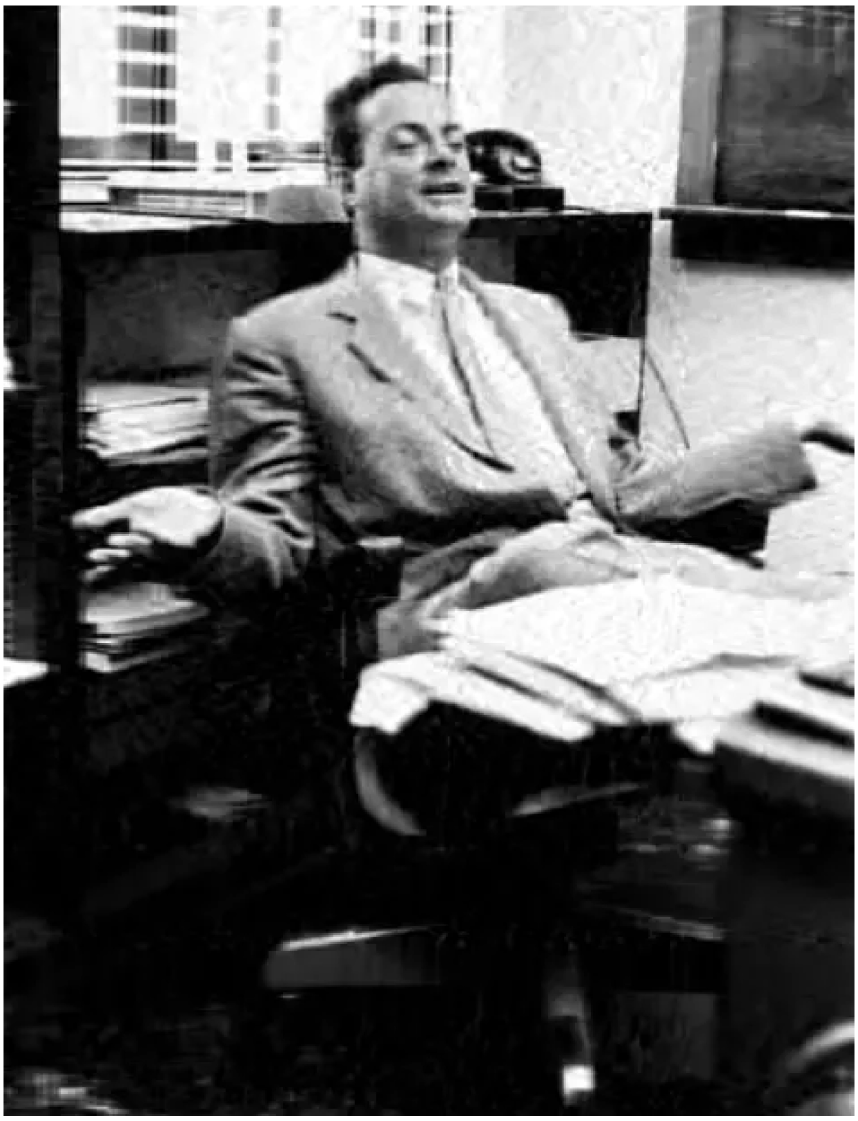


Figure 9.2: Image made of k=50 unique row vectors

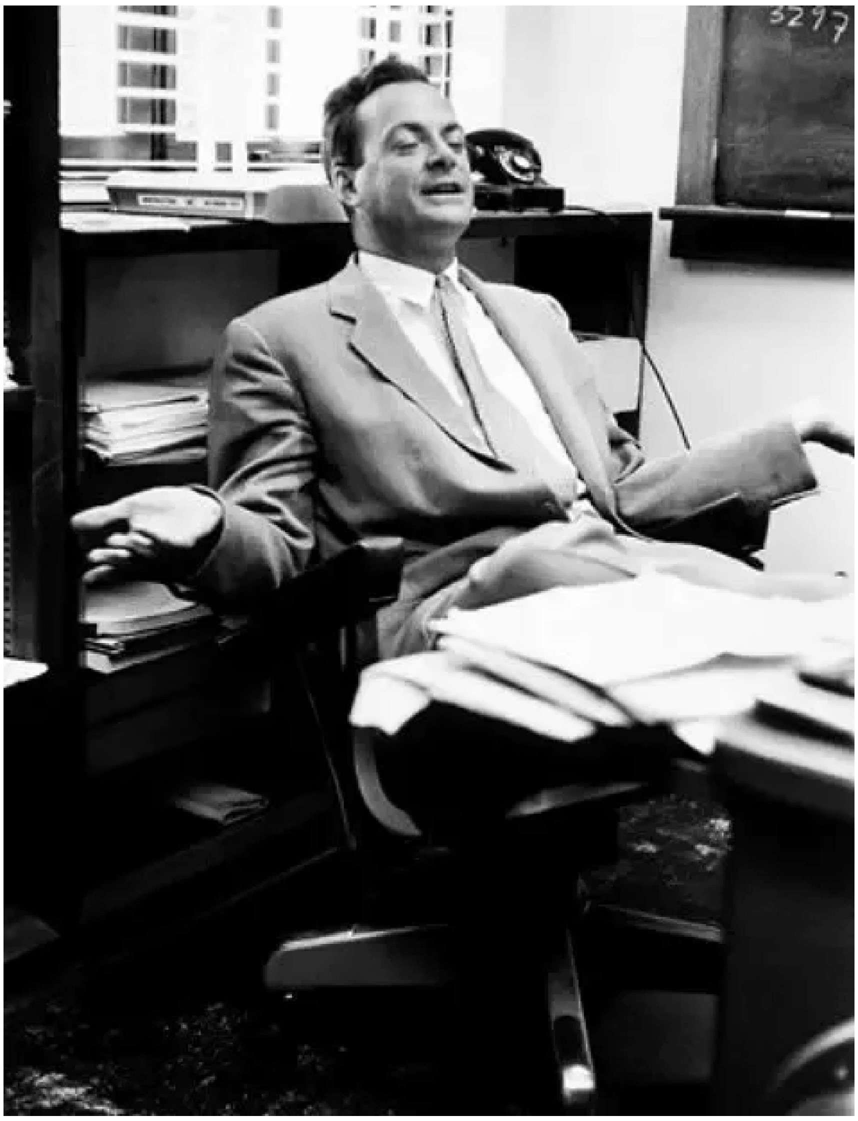


Figure 9.3: Image made of 400 unique row vectors

9.1 Example of a SVD computation

For simplicity, we compute the SVD of a 2x2 matrix C:

$$C = \begin{bmatrix} 5 & 5 \\ -1 & 7 \end{bmatrix}$$

and we want to write it as $C = U\Sigma V^T$.
We have to take into account these 2 equations:

- $C^T C = V\Sigma^T \Sigma V^T$
- $CV = U\Sigma$

We first use the first equation $C^T C = V\Sigma^T \Sigma V^T$, and find the eigenvalues and the eigenvectors of $C^T C$. The eigenvalues will be the entries of the diagonal entries of $\Sigma^T \Sigma$, while the eigenvectors will be the entries of V.
Let's compute $C^T \cdot C$

$$C^T \cdot C = \begin{bmatrix} 5 & -1 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} 5 & 5 \\ -1 & 7 \end{bmatrix} = \begin{bmatrix} 26 & 18 \\ 18 & 74 \end{bmatrix}$$

Now we can find the eigenvalues of the matrix $C^T \cdot C$.
We know that an eigenvalue of $C^T \cdot C$, is a solution of the polynomial equation $det(C^T \cdot C - \lambda I) = 0$, where I is the identity matrix.

$$det(C^T \cdot C - \lambda I) = det\left(\begin{bmatrix} 26 & 18 \\ 18 & 74 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) =$$

$$= det \begin{bmatrix} 26-\lambda & 18 \\ 18 & 74-\lambda \end{bmatrix} = (26-\lambda)(74-\lambda) - 18 \cdot 18 =$$

$$= \lambda^2 - 100\lambda + 1600$$

Since $\lambda^2 - 100\lambda + 1600$ is a second degree equation, we can solve it using the general formula

$$a\lambda^2 + b\lambda + c$$

$$\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\lambda = \frac{-100 \pm \sqrt{100^2 - 4 \cdot 1600}}{2}$$

which will provide the two solutions $\lambda_1 = 20$ and $\lambda_2 = 80$.
Now we want to find the eigenvector associated to the eigenvalue $\lambda_1 = 20$. We know that this eigenvector $\mathbf{v}$ solve the equation $(C^T \cdot C - \lambda I)\mathbf{v} = 0$

$$(C^T \cdot C - \lambda_1 I)v_1 = (C^T \cdot C - 20I)v_1 = \begin{bmatrix} 6 & 18 \\ 18 & 54 \end{bmatrix} \begin{bmatrix} v_{1,1} \\ v_{1,2} \end{bmatrix} = 0$$

We can see it as an equations system

$$\begin{cases} 6 \cdot v_{1,1} + 18 \cdot v_{1,2} = 0 \\ 18 \cdot v_{1,1} + 54 \cdot v_{1,2} = 0 \end{cases}$$

and have $v_{1,1} = -3, \quad v_{1,2} = 1, \quad$ therefore $v_1 = \begin{bmatrix} -3 \\ 1 \end{bmatrix}$.
If we want the matrix V to be othonormal, then each vector of the matrix V has to be a unit vector (i.e. have lenght 1). To make it possible, we divide each entries of the vector by the lenght of the vector. The lenght of v_1 is:

$$|v_1| = \sqrt{v_{1,1}^2 + v_{1,2}^2} = \sqrt{(-3)^2 + 1^2} = \sqrt{10}$$

$$v_1 = \begin{bmatrix} \frac{-3}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} \end{bmatrix}$$

Now we can do the same steps for the second eingenvector

$$(C^T \cdot C - \lambda_2 I)v_2 = (C^T \cdot C - 80I)v_2 = \begin{bmatrix} -54 & 18 \\ 18 & -6 \end{bmatrix} \begin{bmatrix} v_{2,1} \\ v_{2,2} \end{bmatrix} = 0$$

$$\begin{cases} -54 \cdot v_{2,1} + 18 \cdot v_{2,2} = 0 \\ 18 \cdot v_{2,1} - 6 \cdot v_{2,2} = 0 \end{cases}$$

$$v_2 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \Longrightarrow \text{orthonormal} \Longrightarrow v_2 = \begin{bmatrix} \frac{1}{\sqrt{10}} \\ \frac{3}{\sqrt{10}} \end{bmatrix}$$

Now we can write the matrix V

$$V = \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} \frac{-3}{\sqrt{10}} & \frac{1}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & \frac{3}{\sqrt{10}} \end{bmatrix}$$

and the matrix Σ. We know that Σ is a diagonal matrix, so its transpose matrix is $\Sigma^T = \Sigma$, and doing $\Sigma^T\Sigma$ is the same thing of doing the square of each diagonal entries of Σ. As said before at the beginning, the eigenvalues are the entries of $\Sigma^T\Sigma$, so to find the matrix Σ we just need to make the square root of the eigenvalues and put them in diagonal.

$$\Sigma = \begin{bmatrix} \sqrt{\lambda_1} & 0 \\ 0 & \sqrt{\lambda_2} \end{bmatrix} = \begin{bmatrix} \sqrt{20} & 0 \\ 0 & \sqrt{80} \end{bmatrix} = \begin{bmatrix} 2\sqrt{5} & 0 \\ 0 & 4\sqrt{5} \end{bmatrix}$$

Now we need to find the last part of the SVD, which is the matrix U. To find it we use the second property listed at the beginning of this section

- $CV = U\Sigma$

$$CV = \begin{bmatrix} 5 & 5 \\ -1 & 7 \end{bmatrix} \begin{bmatrix} \frac{-3}{\sqrt{10}} & \frac{1}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & \frac{3}{\sqrt{10}} \end{bmatrix} = \begin{bmatrix} -\sqrt{10} & 2\sqrt{10} \\ \sqrt{10} & 2\sqrt{10} \end{bmatrix} = U\Sigma$$

To find U, we have to decompose $\begin{bmatrix} -\sqrt{10} & 2\sqrt{10} \\ \sqrt{10} & 2\sqrt{10} \end{bmatrix}$ as the product of U and Σ

$$\begin{bmatrix} -\sqrt{10} & 2\sqrt{10} \\ \sqrt{10} & 2\sqrt{10} \end{bmatrix} = \begin{bmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{bmatrix} \begin{bmatrix} 2\sqrt{5} & 0 \\ 0 & 4\sqrt{5} \end{bmatrix}$$

If we split the matrix, we obtain

$$\begin{aligned} -\sqrt{10} = u_{1,1} \cdot 2\sqrt{5} + u_{1,2} \cdot 0 &\implies u_{1,1} = -\tfrac{1}{\sqrt{2}} \\ 2\sqrt{10} = u_{1,1} \cdot 0 + u_{1,2} \cdot 4\sqrt{5} &\implies u_{1,2} = \tfrac{1}{\sqrt{2}} \\ \sqrt{10} = u_{2,1} \cdot 2\sqrt{5} + u_{2,2} \cdot 0 &\implies u_{2,1} = \tfrac{1}{\sqrt{2}} \\ 2\sqrt{10} = u_{2,1} \cdot 0 + u_{2,2} \cdot 4\sqrt{5} &\implies u_{2,2} = \tfrac{1}{\sqrt{2}} \end{aligned}$$

Then finally

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

9.2 Difficulties for the SVD in ASIC

As you can see in the example above, the SVD is a complex algorithm. I had some problems in designing it in ASIC especially for a generic matrix nxn. In order we have

- **Determinant**, find the determinant requires times and memory. Greater the dimension of the matrix, harder the computation of the determinant.
- **Eigenvalues** assuming we have find the Characteristic polynomial of the matrix, we will have a polynomial equation of degree n, and we have to find n solutions of this polynomial in hardware.
- **Eigenvectors** assuming we have all the n eigenvalues, for each of the we have to solve a system of n equations with n variables. In total there will be n^2 equations to be solved.

Chapter 10

Synthesis Results

After having properly tested the architecture of each hardware implementation, the following step is its synthesis to determine the maximum clock frequency, area and power consumption.
In some synthesys the chosen sizes are smaller because I used the *Pentium 4 adder* and the *booth multiplier* as adder and as multiplier. Due to their complexity, the synthesys requires much time. In other case, I used the library *ieee.std_logic_arith* for the adder and the multiplier.

10.1 Integral Image

- 16 bits
- size = 8x8
- Pentium 4 adder, Booth multiplier

	Integral Image
	Pipelined
Total cell area	37055.540208 μm^2
Data arrival time	0.62 ns
Internal Power	3.9107mW
Switching Power	1.8435mW
Total Dynamic Power	5.7542mW
Leakage Power	0.3724 mW
Total Power	6.1266 mW

- 32 bits

- size = 16x16
- ieee.std_logic_arith

	Integral Image Pipelined
Total cell area	164944.777137 μm^2
Data arrival time	1.49 ns
Internal Power	16.6224mW
Switching Power	5.3584mW
Total Dynamic Power	21.9808mW
Leakage Power	1.5235 mW
Total Power	23.5043 mW

- 16 bits
- size = 8x8
- ieee.std_logic_arith

	Integral Image Pipelined
Total cell area	14549.606922 μm^2
Data arrival time	0.77 ns
Internal Power	1.8278mW
Switching Power	0.6281mW
Total Dynamic Power	2.4558mW
Leakage Power	0.1380 mW
Total Power	2.5938 mW

10.2 Discrete Cosine Transformation

For this DCT's hadware implementation (3.1), I synthesized both the pipeline and not pipeline architecture. I also synthesized a small version of the LLM architecture (3.5.1). The not pipelined version, consumes more power and take a little time more than the pipelined one. However it has a smaller area due to the absence of pipeline registers.

10.2.1 DCT pipelined

- 8 bits for integer part
- 8 bits for fractional part
- size = 4 (cosine matrix 4x4)
- Pentium 4 adder, Booth multiplier

	Discrete Cosine Transformation Pipelined
Total cell area	$70428.471340 \mu m^2$
Data arrival time	1.83 ns
Internal Power	$4.5759 mW$
Switching Power	$3.4865 mW$
Total Dynamic Power	$8.0625 mW$
Leakage Power	0.6491 mW
Total Power	$8.7116 mW$

- 16 bits for integer part
- 16 bits for fractional part
- size = 9 (cosine matrix 9x9)
- ieee.std_logic_arith

	Discrete Cosine Transformation Pipelined
Total cell area	$142762.305198 \mu m^2$
Data arrival time	$1.78\ ns$
Internal Power	$7.1778 mW$
Switching Power	$5.3690 mW$
Total Dynamic Power	$12.5467 mW$
Leakage Power	$1.1714\ mW$
Total Power	$13.7181 mW$

10.2.2 DCT not pipelined

- 8 bits for integer part
- 8 bits for fractional part
- size = 4
- Pentium 4 adder, Booth multiplier

	Discrete Cosine Transformation Not Pipelined
Total cell area	$69801.783318 \mu m^2$
Data arrival time	$2.05\ ns$
Internal Power	$5.1175 mW$
Switching Power	$3.9349 mW$
Total Dynamic Power	$9.0523 mW$
Leakage Power	$0.6446\ mW$
Total Power	$9.6970 mW$

- 16 bits for integer part
- 16 bits for fractional part

- size = 9 (cosine matrix 9x9)
- ieee.std_logic_arith

	Discrete Cosine Transformation Not Pipelined
Total cell area	132202.285691μm^2
Data arrival time	2.05 ns
Internal Power	45.6441mW
Switching Power	34.0752mW
Total Dynamic Power	79.7267mW
Leakage Power	1.2945 mW
Total Power	81.0204mW

10.2.3 LLM DCT

The LLM DCT architecture shows better performance in everything.
However we remember the complexity of the design for high order (subsection 3.5.4).

- 8 bits for integer part
- 8 bits for fractional part
- size = 4
- Pentium 4 adder, Booth multiplier

	LLM DCT
Total cell area	21937.344292μm^2
Data arrival time	1.70 ns
Internal Power	2.8416mW
Switching Power	2.3913mW
Total Dynamic Power	5.2329mW
Leakage Power	0.2063 mW
Total Power	5.4392 mW

- 16 bits for integer part
- 16 bits for fractional part
- size = 8
- ieee.std_logic_arith

	LLM DCT
Total cell area	$29145.562493 \mu m^2$
Data arrival time	2.96 ns
Internal Power	$11.4379 mW$
Switching Power	$9.7003 mW$
Total Dynamic Power	$21.1382 mW$
Leakage Power	0.2789 mW
Total Power	21.4174 mW

10.3 Binomial Filter

I synthesized both the implementation described in the subsection 4.1.1 and 4.1.2. We can clearly see that the version 1 is better than the version 2 because it requires less computation, actually it leaves the border values unchanged.

10.3.1 Binomial Filter v1

- 16 bits
- size = 4x4
- Pentium 4 adder, Booth multiplier

	Binomial Filter v1
Total cell area	$3795.379296 \mu m^2$
Data arrival time	0.87 ns
Internal Power	1.5640 mW
Switching Power	$1.0043 mW$
Total Dynamic Power	$2.5684 mW$
Leakage Power	0.0405198 mW
Total Power	$2.6089 mW$

- 32 bits
- size = 9x9
- ieee.std_logic_arith

	Binomial Filter v1
Total cell area	$18318.384653 \mu m^2$
Data arrival time	1.39 ns
Internal Power	10.1199 mW
Switching Power	$6.6068 mW$
Total Dynamic Power	$16.7267 mW$
Leakage Power	0.1970 mW
Total Power	$16.9237 mW$

10.3.2 Binomial Filter v2

- 16 bits
- size = 4x4
- Pentium 4 adder, Booth multiplier

	Binomial Filter v2
Total cell area	$8223.321808 \mu m^2$
Data arrival time	0.93 ns
Internal Power	3.3129 mW
Switching Power	$2.0846 mW$
Total Dynamic Power	$5.3975 mW$
Leakage Power	0.0870836 mW
Total Power	$5.4846 mW$

- 32 bits
- size = 9x9
- ieee.std_logic_arith

	Binomial Filter v2
Total cell area	$21143.050352 \mu m^2$
Data arrival time	1.39 ns
Internal Power	11.5835 mW
Switching Power	$7.5642 mW$
Total Dynamic Power	$19.1476 mW$
Leakage Power	0.2275 mW
Total Power	$19.3751 mW$

10.4 FIR

- 16 bits
- size = 8
- Pentium 4 adder, Booth multiplier

	FIR Filter
Total cell area	$35128.637271 \mu m^2$
Data arrival time	2.55 ns
Internal Power	2.3964 mW
Switching Power	$1.8437 mW$
Total Dynamic Power	4.2401 mW
Leakage Power	0.3212 mW
Total Power	$4.5612 mW$

- 32 bits
- size = 81
- ieee.std_logic_arith

	FIR Filter
Total cell area	$141218.759616 \mu m^2$
Data arrival time	5.97 ns
Internal Power	5.2915 mW
Switching Power	$4.3805 mW$
Total Dynamic Power	9.6725 mW
Leakage Power	1.1524 mW
Total Power	$10.8229 mW$

10.5 Transport Equation Problem

- 16 bits
- size = 3x3
- Pentium 4 adder, Booth multiplier

	Transport Equation Problem
Total cell area	43544.045428μm^2
Data arrival time	1.88 ns
Internal Power	8.9910 mW
Switching Power	7.1353mW
Total Dynamic Power	16.1263 mW
Leakage Power	0.4151 mW
Total Power	16.5418mW

- 32 bits
- size = 9x9
- ieee.std_logic_arith

	Transport Equation Problem
Total cell area	153012.244757μm^2
Data arrival time	1.47 ns
Internal Power	15.8138 mW
Switching Power	13.7583mW
Total Dynamic Power	29.5719 mW
Leakage Power	1.3007 mW
Total Power	30.8707mW

10.6 Magnetostatic field calculation 3D

- 32 bits
- size = 5x4x4 (80 cells)
- ieee.std_logic_arith

	Magnetostatic field calculation 3D
Total cell area	115854.867133μm^2
Data arrival time	0.04 ns
Internal Power	26.6830 mW
Switching Power	17.0509mW
Total Dynamic Power	43.7346 mW
Leakage Power	1.0923 mW
Total Power	44.8267mW

Chapter 11

Thessa tool

This experimental tool was the work of another student [23]. It's a tool that given a C code, it translates it into a Mid-level Intermediate Representation code (MIRcode). After that the tool can show the Program Dependence Graph (PDG), the Control Flow Graph (CFG) and the Quotient graph (Q) of the program in MIRcode.

11.1 Analysis of Thessa on the Binomial filter algorithm

The binomial filter implemented in this tools, is different from the one in this thesis. The difference is that in this case it performs an arithmetic average as you can see below in C code (mine was a weighted average)

```
#include <stdio.h>
#define V 10000

int main(){
int w = 8;
int h = 8;
int in[h][w] = {{ 0,0,0,0,0,0,0,0},{ 0,1,6,6,1,1,1,0},{
    0,1,6,6,1,1,1,0},{ 0,1,6,6,1,1,1,0},{ 0,1,6,6,1,1,1,0},{
    0,1,6,6,1,1,6,0},{ 0,1,6,6,1,1,6,0},{ 0,0,0,0,0,0,0,0}};
int out[h][w];
int i;
int j;
int divisor = 8;

for (i = 1; i < h-1; i++){
for (j = 1; j < w-1; j=j+1){

```

```
out[i][j] = (in[i-1][j-1] + in[i-1][j] + in[i-1][j+1] + in[i][j-1] + in[i][j] + in[i][j+1] + in[i+1][j-1] + in[i+1][j] + in[i+1][j+1])/divisor;
}
}

return 0;
}
```

11.1.1 Output generated

11.1.1.1 MIRcode

Having the C code, we have yo translate it in a lower language as the MiRcode and obtain the following code.

```
0         ['$s', 'i', '1']
1         ['+', '_f1', 'i', '1']
2         ['-', '_f0', 'i', '1']
3   L3:   ['for', '>=', 'i', '7', 'goto', 'L0']
4         ['$s', 'j', '1']
5         ['+', '_f3', 'j', '1']
6         ['-', '_f2', 'j', '1']
7   L2:   ['for', '>=', 'j', '7', 'goto', 'L1']
8         ['+', '_t0', 'in[i-1][j]', 'in[i-1][j-1]']
9         ['+', '_t1', '_t0', 'in[i-1][j+1]']
10        ['+', '_t2', '_t1', 'in[i][j-1]']
11        ['+', '_t3', '_t2', 'in[i][j]']
12        ['+', '_t4', '_t3', 'in[i][j+1]']
13        ['+', '_t5', '_t4', 'in[i+1][j-1]']
14        ['+', '_t6', '_t5', 'in[i+1][j]']
15        ['+', '_t7', '_t6', 'in[i+1][j+1]']
```

```
16        ['/', '_t8', '_t7', 'divisor']
17        ['$s', 'out[i][j]', '_t8']
18        ['+', '_t9', 'j', '1']
19        ['$s', 'j', '_t9']
20        ['+', '_f3', 'j', '1']
21        ['-', '_f2', 'j', '1']
22        ['goto', 'L2']
23  L1:   ['+', '_t10', 'i', '1']
24        ['$s', 'i', '_t10']
25        ['+', '_f1', 'i', '1']
26        ['-', '_f0', 'i', '1']
27        ['goto', 'L3']
28  L0:   ['$s', 'reset', '0']
```

11.1.1.2 Program Dependence Graph (PDG)

After the generation of the MIRcode, we have to understand the data and memory dependencies between each single instruction or some critical sections. We obtain the following graph (Program Dependence Graph (PDG)) where each number represent a single instruction (as you can see in the previous MiRcode).

This graph is used to recognize the point in the code where it's not possible to parallelize the code. This point are highlighted by the SSCs component that in the figure are colored in gray box. This graph has also a different color and shape edge. Here we will list all the different color and their meaning:

- Blue edge : Memory Output Dependence
- Red edge : Register Flow Dependence
- Light Green edge : Memory Flow Dependence
- Black edge : Memory Anti-Dependence
- Brown dashed edge : Data Dependence with indexes involved

- Dark Green dotted edge : Control Dependence

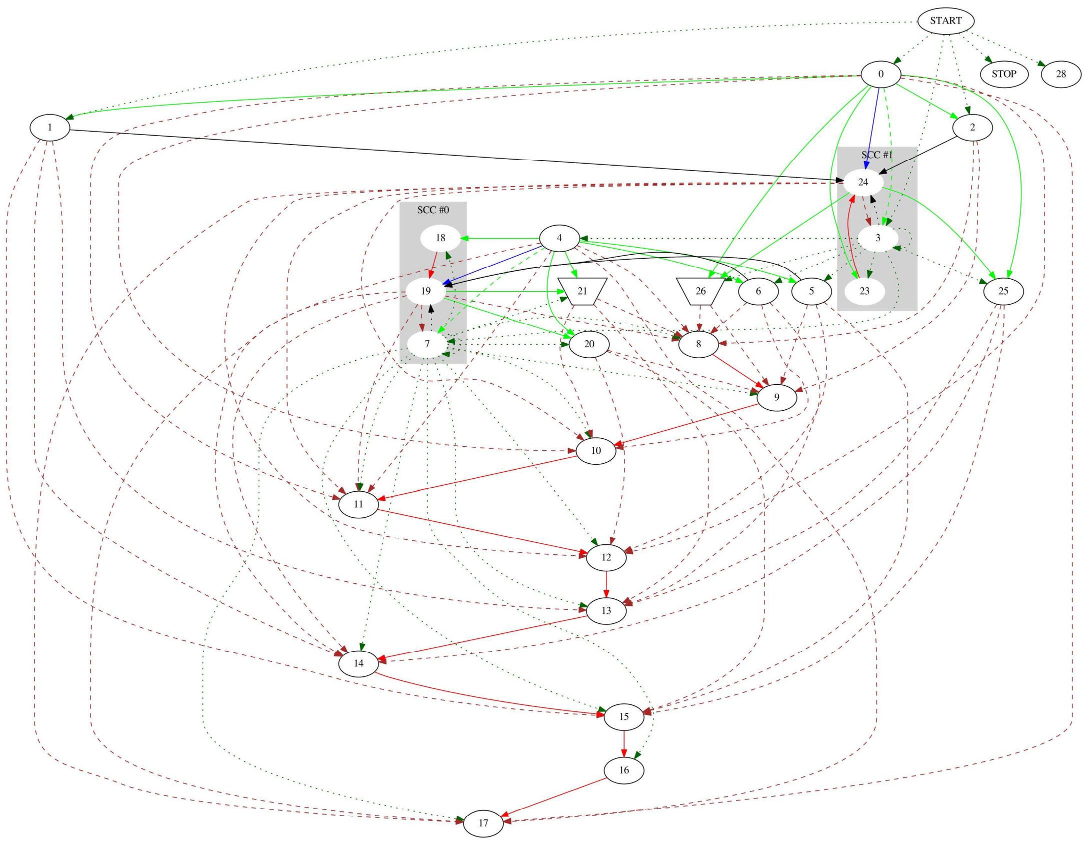


Figure 11.1: Program Dependence Graph (PDG) of the Binomial Filter's MIRcode

11.1.1.3 Quotient Graph (Q) and topological sort

Quotient Graph (Q) is just the same graph PDG but with the SCCs components collapsed into a unique node. This graph allows us to run a topological sort.

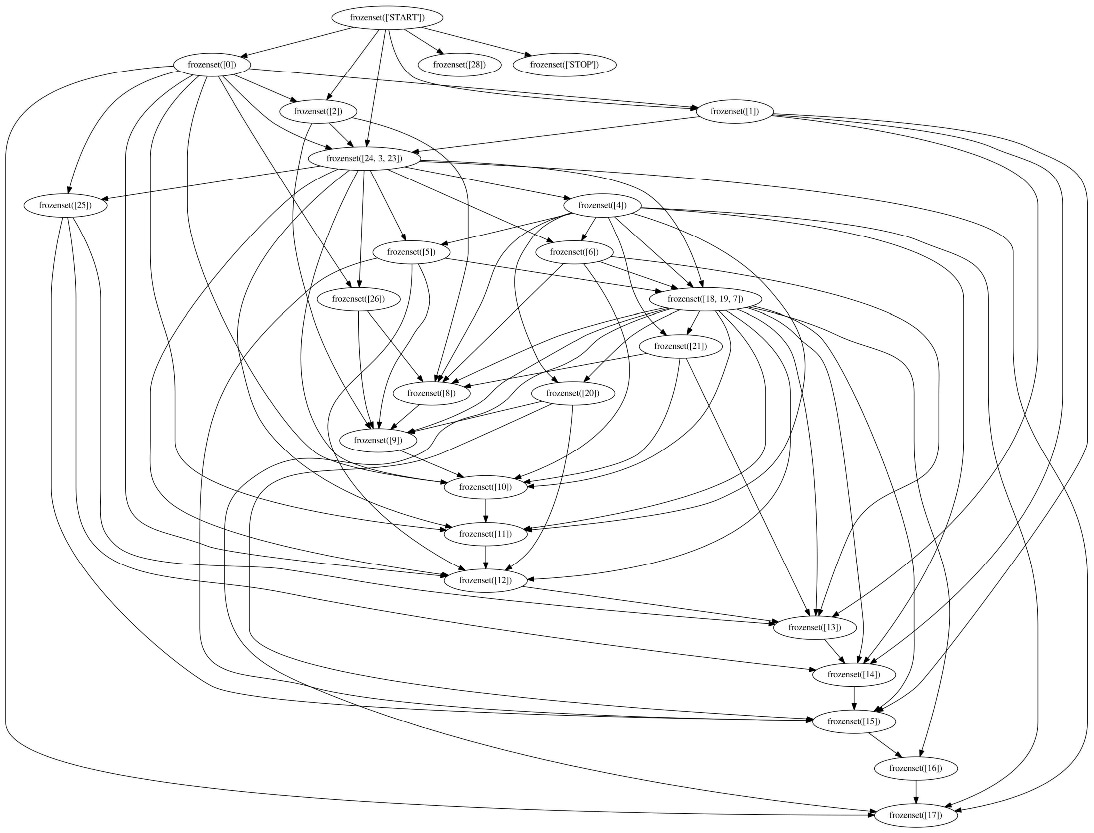


Figure 11.2: Quotient Graph Q of the Binomial Filter's MIRcode

The topological sort can be defined as a linear ordering of the vertexes of a directed acyclic graph (DAG). In general can be done only if the graph has no cycle inside, if there are some cycles (in general each SCCs correspond to a cycle) we can generate the quotient graph (Q) and then use the topological sort on this graph.

```
DEBUG   - graphmanipulation -l:1449     :>> The following list is the topological sort
DEBUG   - graphmanipulation -l:1453     :>> frozenset(['START'])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([28])
DEBUG   - graphmanipulation -l:1453     :>> frozenset(['STOP'])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([0])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([1])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([2])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([24, 3, 23])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([4])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([6])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([5])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([18, 19, 7])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([20])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([26])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([21])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([8])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([9])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([10])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([11])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([12])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([25])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([13])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([14])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([15])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([16])
DEBUG   - graphmanipulation -l:1453     :>> frozenset([17])
```

Figure 11.3: Topological sort of the Binomial Filter's MIRcode

11.1.2 Code Generation hints

The instruction in the SCC (Strongly Connected Components), shown as gray box, have to be performed sequentially in the same core with the instruction linked with black arrows towards these SCC. (example instructions (1) and (2) points to the SCC #1 in fig. 11.1).
While the instructions linked in red lines (Register Flows dependencies) can be performed sequentially in pipeline.
To simplify the understanding, i have drawn the PDG graph (shown before in fig. 11.1) with some blocks. In the next lines, if I want to talk about the n^{th} instruction

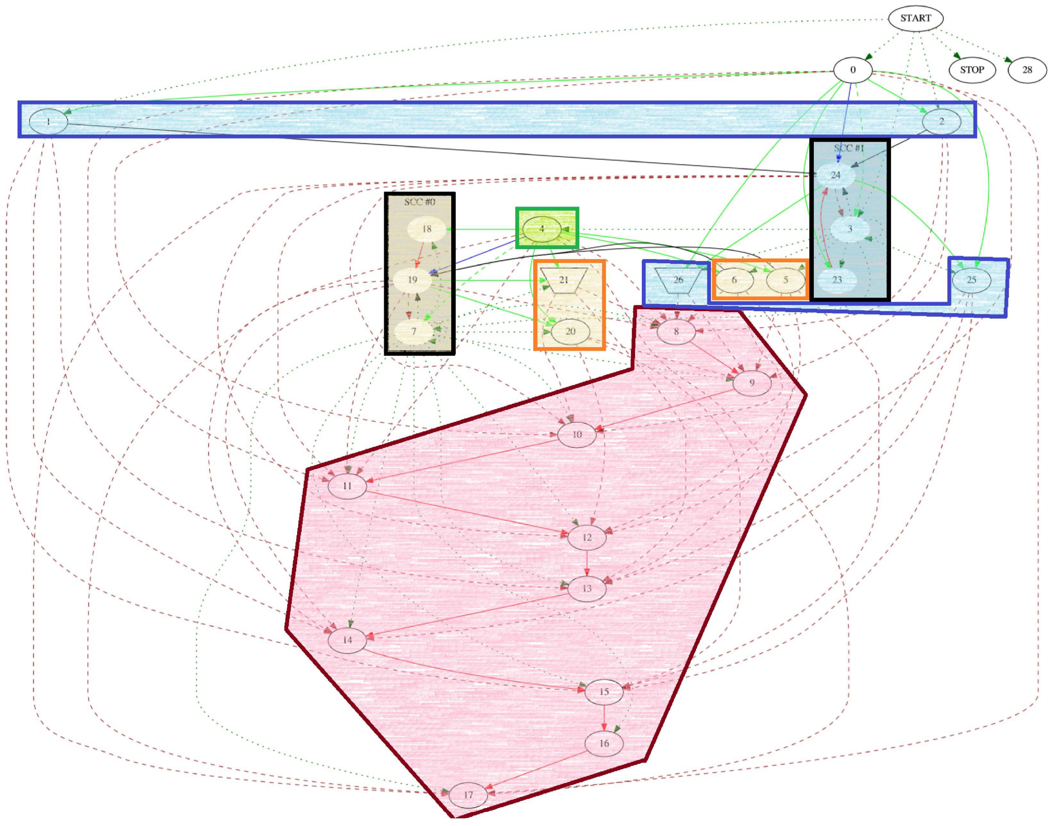


Figure 11.4: PDG divided in blocks of the Binomial Filter's MIRcode

of the MIRcode generated (section 11.1.1.1), I will write (n).

- The Red blocks shows the instruction with data dependencies. You can either put them in a single core (easier to manage, but the core will have more work to do), or have each instruction in a single core (faster but requires more area).

- The blocks with black border represent the SCC.
- The Blue blocks, represent the instructions ((1)(2)) before the first loop (3), and the instructions ((25)(26)) are put before going backwards (27) to the loop (3). Since in the computation, they are near to the SCC-loop ((24)(25)(3)) I suggest to put them all together.
- The orange blocks has similar characteristic with the blue ones. The instructions (5)(6) are before the loop (7) and the lines (20)(21) are executed before going backwards (22) to the loop (7). Lines (18)(19)(7) are in SCC because here we have the update (increase) of the variable 'j' and the comparison. I suggest to put all these instructions in the same core.
- The Green block it's an instruction performed after the first loop and before the second loop. You can put with the blue blocks or with the orange ones.

11.1.3 Comments and improvements

I can't comment on the MIRcode generation, because i don't know this language and i don't understand the meaning of instructions [(1) (2) (25) (26)] and [(5) (6) (20) (21)]. Maybe they are used for the synchronization of the for loop (3) and loop (7).
For other lines, it's easier to understand the translation from C code to MIRcode.

Here i will post some quotation of the original work [23].

> *"The topological sort doesn't give the best solution, but what it gives is always correct (with respect to the correct starting graph)."*

I can't comment on the utility of this topological sort, because i didn't use it. Futhermore, i believe you need to see the MIRcode to see the dependencies and the PDG to see the SCCs.

"Only the nodes that are inside the loop are relevant for the optimization. This because the nodes that don't need to be executed multiple time because they are inside a for can be are not liked in particular to as specific core, but can used and merged to other one,"

That's true, but to optimize better, i suggest the **Loop unrolling**, a technique that attempts to optimize a program's execution speed at the expense of its binary size.
In other words, instead of writing this piece of code, (because in a given time it performs iteratively the binomial filter of ONLY a single cell)

```
for (i = 1; i < h-1; i++){
 for (j = 1; j < w-1; j=j+1){
  out[i][j] = ...
 }
}
```

write

```
 out[1][1] = ...
 out[1][2] = ...
 out[1][3] = ...
 out[1][4] = ...
 ...
 out[6][3] = ...
 out[6][4] = ...
 out[6][5] = ...
 out[6][6] = ...
```

this will enable the computation of all the cells in parallel.
Another suggestion is to use a computation optimization like the one i used for my binomial filter: instead of doing

```
 out[1][1] = ...+...+ in[2][0] + in[2][1] + in[2][2]; //row[0], row[1],
      row[2]
 out[2][1] = ...+ in[2][0] + in[2][1] + in[2][2] + ...; //row[1], row
      [2], row[3]
 out[3][1] = in[2][0] + in[2][1] + in[2][2] + ... + ...;// row[2], row
      [3], row[4]
```

you could simplify by writing

```
temp=in[2][0] + in[2][1] + in[2][2];
out[1][1] = ...+...+ temp //row[0], row[1], row[2]
out[2][1] = ...+temp + ...; //row[1], row[2], row[3]
out[3][1] = temp + ... + ...;// row[2], row[3], row[4]
```

This will save you to do the same additions three times.

11.2 How to use

11.2.1 Install python 2.7

To install python 2.7.13 you can see this website It is adviced to install these pre-requisites before installing Python

```
 $ sudo apt-get update
 $ sudo apt-get install build-essential checkinstall
 $ sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-
     dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
```

download and extract the package (if you cannot download it normally, open the package and then extract it manually)

```
$ cd /usr/src
$ wget https://www.python.org/ftp/python/2.7.13/Python-2.7.13.tgz
$ tar xzf Python-2.7.13.tgz
```

compile python source

```
$ cd Python-2.7.13
$ sudo ./configure
$ sudo make altinstall
```

to see which version of python you have, open the terminal and digit

```
$ python2 --version
```

11.2.2 PIP (Python package manager) and setuptools

First, make sure you have the lastest version of pip (Python package manager) installed. if you do not, digit the following lines

```
$ sudo apt install python-pip
$ pip install --upgrade pip
```

Before installing networkx, you need the setuptools

```
$sudo apt-get install python-setuptools
```

11.2.3 networkx

Now you can try to install networkx

```
$ pip install networkx
```

If you do not have permission to install software systemwide, you can install into your user directory using the "–user" flag:

```
$ pip install --user networkx
```

11.2.4 Graphviz

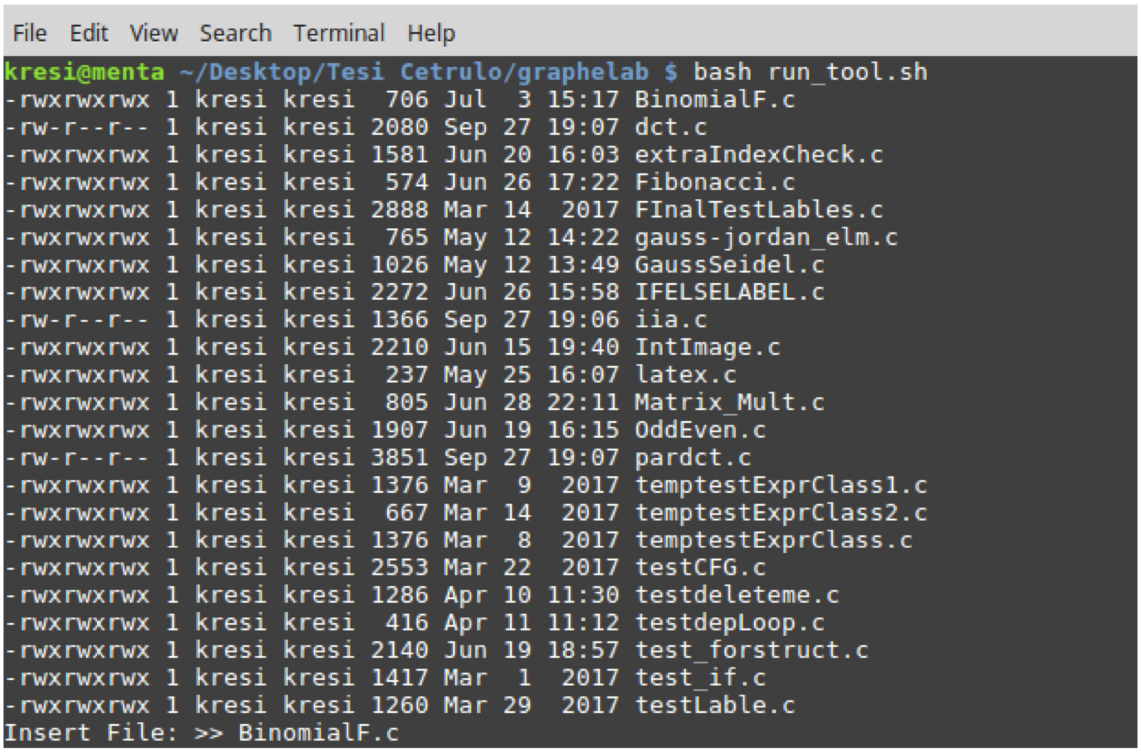
Now you need to install *Graphviz*, an open source graph visualization software and other things:

```
$sudo apt-get install graphviz
```

11.2.5 Running Tessa tool

You can run the tessa tool. Open the terminal and go to the folder "*graphelab*" and run the bash script

```
$ bash run\_tool.sh
```



```
kresi@menta ~/Desktop/Tesi Cetrulo/graphelab $ bash run_tool.sh
-rwxrwxrwx 1 kresi kresi  706 Jul  3 15:17 BinomialF.c
-rw-r--r-- 1 kresi kresi 2080 Sep 27 19:07 dct.c
-rwxrwxrwx 1 kresi kresi 1581 Jun 20 16:03 extraIndexCheck.c
-rwxrwxrwx 1 kresi kresi  574 Jun 26 17:22 Fibonacci.c
-rwxrwxrwx 1 kresi kresi 2888 Mar 14  2017 FInalTestLables.c
-rwxrwxrwx 1 kresi kresi  765 May 12 14:22 gauss-jordan_elm.c
-rwxrwxrwx 1 kresi kresi 1026 May 12 13:49 GaussSeidel.c
-rwxrwxrwx 1 kresi kresi 2272 Jun 26 15:58 IFELSELABEL.c
-rw-r--r-- 1 kresi kresi 1366 Sep 27 19:06 iia.c
-rwxrwxrwx 1 kresi kresi 2210 Jun 15 19:40 IntImage.c
-rwxrwxrwx 1 kresi kresi  237 May 25 16:07 latex.c
-rwxrwxrwx 1 kresi kresi  805 Jun 28 22:11 Matrix_Mult.c
-rwxrwxrwx 1 kresi kresi 1907 Jun 19 16:15 OddEven.c
-rw-r--r-- 1 kresi kresi 3851 Sep 27 19:07 pardct.c
-rwxrwxrwx 1 kresi kresi 1376 Mar  9  2017 temptestExprClass1.c
-rwxrwxrwx 1 kresi kresi  667 Mar 14  2017 temptestExprClass2.c
-rwxrwxrwx 1 kresi kresi 1376 Mar  8  2017 temptestExprClass.c
-rwxrwxrwx 1 kresi kresi 2553 Mar 22  2017 testCFG.c
-rwxrwxrwx 1 kresi kresi 1286 Apr 10 11:30 testdeleteme.c
-rwxrwxrwx 1 kresi kresi  416 Apr 11 11:12 testdepLoop.c
-rwxrwxrwx 1 kresi kresi 2140 Jun 19 18:57 test_forstruct.c
-rwxrwxrwx 1 kresi kresi 1417 Mar  1  2017 test_if.c
-rwxrwxrwx 1 kresi kresi 1260 Mar 29  2017 testLable.c
Insert File: >> BinomialF.c
```

Figure 11.5: List of all your C files

A list of all the C codes (in the folder ”*modules/class/test*”) will appear. You have to digit the C file you want to analyze If you get the error:
ImportError: No module named pylab
Is probably because you need some packages which provide additional functionality:

- **NumPy** provides matrix representation of graphs and is used in some graph algorithms for high-performance matrix computations.
- **SciPy** provides sparse matrix representation of graphs and many numerical scientific tools.
- **pandas** provides a DataFrame, which is a tabular data structure with labeled axes.
- **Matplotlib** provides flexible drawing of graphs.
- **PyGraphviz** and **pydot** provide graph drawing and graph layout algorithms via *GraphViz*.

- **PyYAML** provides YAML format reading and writing.
- **gdal** provides shapefile format reading and writing.
- **lxml** used for GraphML XML format.

Since I keep getting some errors, i decided to install all the packages.
To install *networkx* and all optional packages, do:

```
$ pip install networkx[all]
```

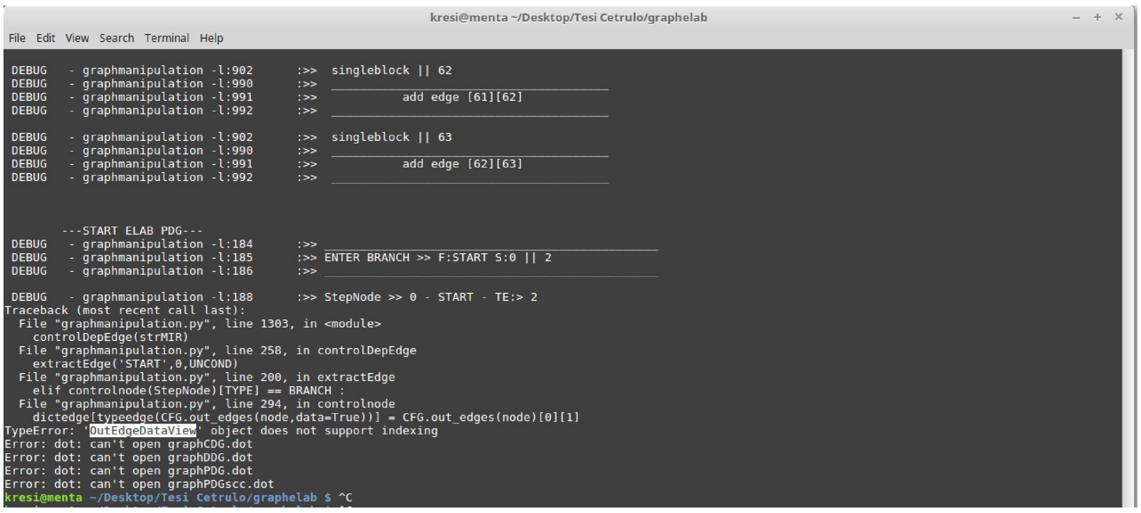
To explicitly install all optional packages, do:

```
$ pip install numpy scipy pandas matplotlib pygraphviz pydot pyyaml
    gdal
```

If you get the error:
gnome-open: command not found

```
$sudo apt-get install libgnome2-0
```

Now this is where my exploration on tessa tools ends. I get the error:
TypeError: 'OutEdgeDataView' object does not support indexing I tried to find where



```
 DEBUG   - graphmanipulation -l:902      :>>  singleblock || 62
 DEBUG   - graphmanipulation -l:990      :>>  _______________________________________
 DEBUG   - graphmanipulation -l:991      :>>            add edge [61][62]
 DEBUG   - graphmanipulation -l:992      :>>  _______________________________________

 DEBUG   - graphmanipulation -l:902      :>>  singleblock || 63
 DEBUG   - graphmanipulation -l:990      :>>  _______________________________________
 DEBUG   - graphmanipulation -l:991      :>>            add edge [62][63]
 DEBUG   - graphmanipulation -l:992      :>>  _______________________________________



        ---START ELAB PDG---
 DEBUG   - graphmanipulation -l:184      :>> ________________________________________________
 DEBUG   - graphmanipulation -l:185      :>> ENTER BRANCH >> F:START S:0 || 2
 DEBUG   - graphmanipulation -l:186      :>> ________________________________________________

 DEBUG   - graphmanipulation -l:188      :>> StepNode >> 0 - START - TE:> 2
Traceback (most recent call last):
  File "graphmanipulation.py", line 1303, in <module>
    controlDepEdge(strMIR)
  File "graphmanipulation.py", line 258, in controlDepEdge
    extractEdge('START',0,UNCOND)
  File "graphmanipulation.py", line 200, in extractEdge
    elif controlnode(StepNode)[TYPE] == BRANCH :
  File "graphmanipulation.py", line 294, in controlnode
    dictedge[typeedge(CFG.out_edges(node,data=True))] = CFG.out_edges(node)[0][1]
TypeError: 'OutEdgeDataView' object does not support indexing
Error: dot: can't open graphCDG.dot
Error: dot: can't open graphDDG.dot
Error: dot: can't open graphPDG.dot
Error: dot: can't open graphPDGscc.dot
kresi@menta ~/Desktop/Tesi Cetrulo/graphelab $ ^C
```

kresi@menta ~/Desktop/Tesi Cetrulo/graphelab
File Edit View Search Terminal Help

Figure 11.6: OutEdgeDataView

the object 'OutEdgeDataView' was declared but I didn't find it on the script. Then, after searching on internet, I figure it out it was written in the file ”*reportviews.py*” which was in the folder ” */.local/lib/python2.7/site-packages/networkx/classes*” And without this modified file, i can't get the graphs

Bibliography

[1] Performance driven FPGA design with an ASIC perspective. Andreas Ehliar, 2009

[2] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics",in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2007.

[3] Actel, "Igloo low-power flash fpgas handbook,"2008. [Online]. Available: http://www.actel.com/documents/IGLOO_HB.pdf

[4] 2013 by Yong Cao, Referencing UIUC ECE408/498AL Course Notes (Lecture 10 - Virginia Tech)

[5] Prefix sum - Parallel algorithm - Wikipedia

[6] Patrick Cozzi,University of Pennsylvaniam, Summed area tables, CIS 565 - Spring 2011

[7] Qingqing Dang, Shengen Yan, Ren Wu, A fast integral image generation algorithm on GPUs, 2014 IEEE

[8] 2015, Mario Cofano, Mariagrazia Graziano, Maurizio Zamboni, Design of a Logic-in-Memory architecture for massive parallel algorithms

[9] Discrete cosine transform - Wikipedia

[10] PRACTICAL FAST 1-D DCT ALGORITHMS WITH 11 MULTIPLICATIONS, Christoph Loeffler, Adriaan Lieenberg, and George S. Moschytz

[11] 2015, Filasieno Francesco, Garino Simone, Garlando Umberto, Progettazione di Sistemi a Basso Consumo : Array Sistolico Riconfigurabile

[12] LLM Integer Cosine Transform and Its Fast Algorithm Chi-Keung Fong, Student Member, IEEE, and Wai-Kuen Cham, Senior Member, IEEE

[13] High Order Integer Transform Design for Video Compression,Jie Dong, Member, IEEE, and Yan Ye, Member, IEEE

[14] DIGITAL IMAGE FILTERING By Fred Weinhaus

[15] Integrated system architecture, lecture notes by E. Raviola, 2017

[16] A Guide to Numerical Methods for Transport Equations, Dmitri Kuzmin

[17] DEVELOPPEMENT D'UN CODE DE CALCUL MICROMAGNETIQUE 2D ET 3D: APPLICATION A DES SYSTEMES REELS DE TYPES FILMS,

PLOTS ET FILS, Liliana-Daniela BUDA

[18] Analysis and Design of an Optimized HW Accelerator for Protein Alignment, Gianvito Urgese, Mariagrazia Graziano, Maurizio Zamboni, September 2012, Politecnico di Torino

[19] S. Henikoff, J. G. Henikoff, Amino acid substitution matrices from protein blocks in Proceedings of the National Academy of Sciences of the United States of America, v. 89(22), pp. 10915-10919, November 1992.

[20] T. F. Smith, M. S. Waterman, "Identification of Common Molecular Subsequences" in Journal of Molecular Biology, v. 147.

[21] Andrew Gibiansky, Cool Linear Algebra: Singular Value Decomposition, http://andrew.gibiansky.com/blog/mathematics/cool-linear-algebra-singular-value-decomposition/

[22] Jason Liu, on Quora https://www.quora.com/What-is-an-intuitive-explanation-of-singular-value-decomposition-SVD

[23] Eugenio CETRULO, Mariagrazia Graziano, Maurizio Zamboni, Algorithm parallelization for Logic-in-Memory architectures