POLITECNICO DI TORINO

Department of Electronics and Telecommunications
Master's Degree in Electronic Engineering

Master's Thesis

VLSI architecture of a high speed Wiener Filter for video coding

Supervisor:
Prof. Maurizio Martina

Candidate:
Sandro Di Paola

July 14, 2020

Acknowledgments

I would like to thank Professor Maurizio Martina, for supporting me during my work of thesis, for being always present and available, despite the distance and for sharing ideas and suggestions with me. In the same way I would like to thank my colleague and friend Giorgio, with who I have worked intensely during these months and who, with his ideas and dedication to work, has contributed to the realization of this work of thesis.
I would also like to thank all those who have supported me in these years.
Thanks to Federica, who has shared this experience with me, who has always been close to me, who has enjoyed with me the achievements and who has supported me in the most difficult moments.
I would like to thank Giorgio who has always been a point of reference in these years, both for the study and mainly because he has always been a true friend.
I would also like to thank my friends Gianluca and Andrea who shared this course of studies with me and my oldest friends, Andrea, Gabriele and Ninni.
Finally, I would like to thank my family, without which I could not have realized all this. Thanks to my parents, who with their sacrifices, their encouragement and their support have helped me during this path. Thanks to my brother, my sister, my grandparents and all the rest of the family who have always believed in me.

Summary

In the modern age, the use of video has become fundamental in communication and this has led to their use through an increasing number of devices. Thanks to the emerging media, such as social networks, streaming, internet and mobile devices in addition to the old media, such as television, videos are an omnipresent tool in everyday life. An important aspect is that the increase in videos distribution has been accompanied by the demand for higher quality and this has represented a problem due to the fixed memory and limited bandwidth of the devices. In fact, video is the most expensive support in terms of memory and bandwidth, due to the evolution of different formats, starting from High Definition (HD) and Ultra High Definition (UHD), up to 4K UHD which provide an increasing number of pixels per frame and therefore, an increasing need for memory and bandwidth for transmission. These features, in contrast to the limited bandwidth and devices memory, are the main reason why video coding was born. When we talk about storage and transmission of video, we always talk about video compression.
The key idea of video coding is to compress this enormous quantity of data into an encoded bit-stream, thanks to the elimination of redundant elements. Redundancy is the part of the message that is not fundamental and can be eliminated without damaging the information. In each frame we can find large amounts of redundancy, that can be of different types, for example it can be spatial correlation, close pixels usually have close values or temporal correlation between two successive frames, close frames usually have the same subject.
In order to remove the redundancy or to rebuild one frame, AV1, a new open video coding format which is the evolution of the previous VP9 and that competes with HEVC, employs some different steps, one of these steps consists in the use of several loop-filters. The goal is to remove ringing artifacts and improve the quality of reconstructed frame after adding the prediction to the error, so they are applied to a decoded frame.
This work of thesis focus its attention on in-loop filters, in particular on one of the loop restoration filter: the Wiener Filter, so a new hardware architecture implementing the separable symmetric normalized Wiener Filter, is presented. Starting from an article present in the literature, which analyzes the algorithm from a mathematical

point of view and then, analyzing the software implementation of the filter in the codec, we explore different architectural solutions and then we present the overall architecture. The key idea of the Wiener Filter is that all the pixels of a degraded frame can be reconstructed through the pixels surrounding them. In particular we analyze the pixels in a *w x w* window around the pixel that need to be recontructed, where $\boldsymbol{w = 2r + 1}$ with *r* integer number.
The key elements are:

- $\boldsymbol{H = E[XX^T]}$: the autocovariance of x which can be rewritten as:

$$H = \begin{bmatrix} H_{00} & H_{01} & \dots & H_{0,w-1} \\ H_{10} & H_{11} & \dots & H_{1,w-1} \\ \dots & \dots & \dots & \dots \\ H_{w-1,0} & H_{w-1,1} & \dots & H_{w-1,w-1} \end{bmatrix} \tag{1}$$

  in which each $\boldsymbol{H_{ij}}$ element is a matrix of size *w x w*.

- $\boldsymbol{M = E[YX^T]}$: the cross correlation of *x* with the source y which can be rewritten as:

$$M = \begin{bmatrix} M_0 & M_1 & \dots & M_{w-1} \end{bmatrix} \tag{2}$$

  in which each $\boldsymbol{M_i}$ element is a vector of size *1 x w*.

- **a** and **b**: vertical and horizontal filters of size *w x 1*.
  In addition several constraints of normalization and symmetry are added to **a** and **b**:

  - $a(i) = a(w-1-i)$ and $b(i) = b(w-1-i)$ for $i = 0, 1, ..., r-1$

  - $\sum a(i) = \sum b(i) = 1$

The key idea is to start with a first guess of horizontal and vertical filters **a** and **b**, after which the filter **a** is computed, keeping **b** fixed. Once the value of filter **a** is computed, it is used to compute the new value of filter **b**.
In this way, the overall algorithm can be divided in two main steps:
1) **Update a**, fixing **b**: to find the final values of **a** vector, starting from a guess **b** vector.
2) **Update b**, fixing **a**: to find the final values of **b** vector, using the new **a** vector.

A first basic architecture has been built without taking into account any area, resource and performance constraints. It simply executes correctly the algorithm present in the codec and obtains the same output values. This basic architecture is not suitable because it has many critical issues, but it is an excellent starting point to create a new architecture that executes the same algorithm, but at the same time respects any design specs.
After synthesizing the basic architecture, tests were performed that identified several critical issues. Our focus was on performance in terms of speed. Several timing reports were analyzed, they showed the two main problems of the basic architecture: the excessive length of some combinational paths and the high complexity of the division operation. In detail, the timing reports show how, with a clock of 10 ns you get a negative slack of -221.70 ns, which implies a maximum operating frequency of 4.3 MHz. Starting from the basic architecture, some changes were made to the blocks placed inside the critical paths.
First of all, after analyzing the critical paths showed by the timing reports, we tried to reduce the length of the combinational paths by placing pipeline registers, so that each combinational path did not include more than one operator within it. After applying this adjustment, we performed the same tests with the same parameters, obtaining a negative slack of -56.58 ns which implies a maximum operating frequency of 15.15 MHz, an improvement of one order of magnitude. Finally, a new restoring divider has been implemented to replace the previous one.
The final tests performed on the overall architecture, showed how, with a clock of 10 ns, you get a negative slack of -1.38 ns, which implies a maximum operating frequency of 87.87 MHz, an overall improvement close to two orders of magnitude.

List of figures

Table of contents

Chapter 1

Introduction

In the modern age, the use of video has become fundamental in communication and this has led to their use through an increasing number of devices.
Thanks to the emerging media, such as social networks, streaming, internet and mobile devices in addition to the old media, such as television, videos are an omnipresent tool in everyday life.
According to the periodical report of Cisco Visual Networking Index (Cisco VNI), which studies the use of the different networking (Wi-Fi, 3G, 4G, 5G), this trend will not end in the coming years thanks to different factors.
First of all, there will be an increasing number of internet users,per capita devices and also the number of global public Wi-Fi hotspots will increase a lot. Thanks to this huge availability of connections, people will be able to watch videos at all times and marketing is also taking advantage of this situation by investing money in video advertising. In this scenario, video bandwidth demand will be more than 82% of total demand. [1]
Another important aspect is that the increase in videos distribution has been accompanied by the demand for higher quality and this has represented a problem due to the fixed memory and limited bandwidth of the devices.
In fact, video is the most expensive support in terms of memory and bandwidth, due to the evolution of different formats, starting from High Definition (HD) and Ultra High Definition (UHD), up to 4K UHD which provide an increasing number of pixels per frame and therefore, an increasing need for memory and bandwidth for transmission.
These features, in contrast to the limited bandwidth and devices memory, are the main reason why video coding was born.
In this thesis work a new hardware architecture implementing one of the loop filtering algorithm present in the AV1 codec is presented. Starting from the codec, we explore different architectural solutions and then we present the overall architecture. More in detail every single block of the starting architecture and the tests performed on it

are presented. After that, the improvements that have been made to the individual blocks of the base architecture to get the new architecture working at high speed are presented. Finally, after showing the results of the tests on the new architecture, the achieved improvements will be highlighted.

Chapter 2

Background

Nowadays, whenever we talk about storage and transmission of video, we always talk about video compression.
The key idea of video coding is to compress this enormous quantity of data into an encoded bit-stream, thanks to the elimination of redundant elements.
Redundancy is the part of the message that is not fundamental and can be eliminated without damaging the information. In each frame we can find large amounts of redundancy, that can be of different types, for example it can be spatial correlation, close pixels usually have close values or temporal correlation between two successive frames, close frames usually have the same subject. [2]

2.1 Formats and AV1

Starting from 1984, when the International Telecommunication Union (ITU), launched H.120, the first digital video coding technology standard, different encoding formats have emerged in this context.(2.1) [3]

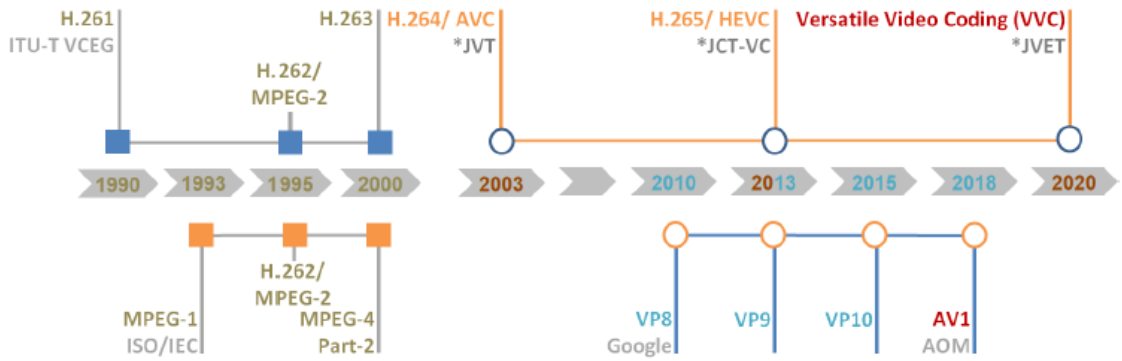

Figure 2.1: Video coding standards development timeline. [7]

The main recent formats are the VP9 video codec, launched by Google in 2013 and the High Efficiency Video Coding (HEVC), the standard not royalty-free, also developed in 2013 by ISO/IEC Moving Picture Experts Group (MPEG) and ITU-T Video Coding Experts Group (VCEG).
In 2015, the Alliance for Open Media (AOMedia) was born, it is a consortium of more than 30 companies co-founded by Google. The idea is to develop a solution that is open and royalty-free. [4]
The solution is to create a new open video coding format, AOMedia Video 1 (AV1), which is the evolution of the previous VP9 and that competes with HEVC. In particular the idea is to improve scalability, flexibility, so as to be compatible with modern devices and improve performance in terms of compression.

2.1.1 AV1 Structure

In order to remove the redundancy or to rebuild one frame, AV1 employs some different steps (2.2). Starting from the source, each frame is subjected to the following operations which are described in detail in the article **"An Overview of Core Coding Tools in the AV1 Video Codec"**:

- ***TRANSFORM***: This block transforms the remaining error after the subtraction of prediction from the source frame into frequency domain, using Discrete Cosine Transform (DCT) and Discrete Sine Transform (DST) over a square block of different dimension, according to the case under evaluation.
  - *Transform Block Partition*: AV1 allows blocks to be partizioned into units of different sizes, instead of using fixed transform unit sizes.
  - *Extended Transform Kernels*: Both inter and intra blocks can use a largest set of transform kernels.

- ***ENTROPY CODING***: Thanks to a specific alphabet, a multi-symbol encoder code every syntax elements.
  - *Multi-symbol Entropy Coding*: AV1 uses a symbol-to-symbol adaptive multi-symbol arithmetic coder, this is an high precision coder and this means that it enables tracking probabilities of less common elements with high precision.
  - *Level Map Coefficient Coding*: AV1 changes the level map design for dimensional transformation coefficients modeling and compression to better capture the distribution of coefficient in the space.
- ***LOOP-FILTERING***: There are several loop-filters, the goal is to remove ringing artifacts and improve the quality of reconstructed frame after adding the prediction to the error, so they are applied to a decoded frame. A new feature is that AV1 separates horizontal and vertical filtering for each plane. The filtering tool used by AV1 are:
  - *Constrained Directional Enhancement Filter (CDEF)*: This filter estimates the direction of the edge by applying a low pass filter and to avoid extrasignaling the decoder uses an algorithm that minimizes the quadratic error with respect to the ideal case.
  - *Loop Restoration Filters*: AV1 has new tools that are applied in loop and are selected in mutual exclusion from a unit called loop-restoration unit (LRU)
    - *Separable symmetric normalized Wiener Filter*: A 7x7 Wiener Filter filters the pixels, the coefficients are included in the bitstream. It is necessary to send to each horizontal and vertical filter only 3 parameters thanks to the constraints of normalization and symmetry.
    - *Dual self-guided filter*: This filter uses a guide image which is the same image to be filtered. The outputs of the two filters are combined with the weight contained in the bitstream to obtain the final restored LRU.
  - *Frame Super-resolution*: AV1 implements a new super-resolution coding frame.In this way the frame is coded in a lower spatial resolution and then super-resolved in-loop at maximum resolution. To do this, AV1 uses the loop-restoration tools seen above at high resolution.
- ***FILM GRAIN SYNTHESIS***: It is the final step of the reconstruction, this filter is used to remove the grain component from the signal and transmits descriptive parameters to the decoder. This step is applied outside the encoding

and decoding loops. Due to the fact that the film grain is random, it is difficult to compress them with standard methods. Instead, the grain is removed before compression and encoded in the bitsream. In AV1 the bitrate needed to rebuild the grain with acceptable quality is reduced.

- ***INTRA-PREDICTION***: The goal is to predict the pixel of a frame using the information given from the frame itself.
  VP9 implement DC and TM mode, that are 2 non-directional predictor and also 10 intra prediction modes, they include 8 directional modes that correspond to angles from 45 to 207 degree. AV1 improve the intra coder in various way:

  – *Enhanced Directional Intra Prediction*: There are 56 directional intra modes in AV1. A more accurate angle set to improve directional intra modes. The angle is represented by a nominal angle plus a delta which can assume value from -3 to +3.

  – *Non-directional Smooth Intra Predictors*: 3 new smooth predictors, Smooth_V, Smooth_H and Smooth are added. Using quadratic interpolation, these predictors predict the block in many directions. Finally, PAETH predictor replace the TM mode.

  – *Recursive-filtering-based Intra Predictor*: For luma blocks, FILTER_INTRA are designed, the aim is to evaluate spatial correlation on the edges. In addition, five filter intra modes are implemented.

  – *Chroma Predicted from Luma*: The inter predictor Chroma from Luma starting from reconstructed luma pixels, models chroma pixels. It determines the needed parameters directly from the bitstream.

  – *Color Palette as a Predictor*: For every plane of a block, the color palette predictor is represented by a color palette and color indices for every pixel of the block.

  – *Intra Block Copy*: Previously reconstructed block is used as reference to the actual intra coder. To perform this operation, IntraBC is implemented, it allows to make a copy of a reconstructed block as a prediction in the current frame.

- ***INTER-PREDICTION***: It has the same goal of the Intra-Prediction block, but uses the information over two successive frames. AV1 has better inter coder with respect to VP9.

  – *Extended Reference Frames*: The number of references for each frame are extended from 3 to 7. In addition to that, two near past frames and two

future frames are added. In the AV1 we can find a wide set of references that allow to encode in an optimal way different types of videos that have dynamic temporal correlations.

- *Dynamic Spatial and Temporal Motion Vector Referencing*: A new MV reference selection is implemented to obtain better MV references. AV1 uses a temporal motion field estimation mechanism to create temporal candidates.

- *Overlapped Block Motion Compensation (OBMC)*: This block can decrease prediction errors near edges. In order to make OBMC easily fit, 2-side casual overlapping algorithm is implemented.
- *Warped Motion Compensation*: Two affine prediction modes are implemented, global and local warped motion compensation. Global motion is used for handling camera motions, the local warped motion is used to describe varying local motion. Affine warping is limited to small degrees and this is good for hardware implementation.
- *Advanced compound prediction*: In order to make inter coder more versatile, new compound prediction tools are implemented. So different compound prediction modes are implemented, in detail: Compound wedge prediction, Difference-modulated masked prediction, Frame distance based compound prediction and Compound inter-intra prediction.

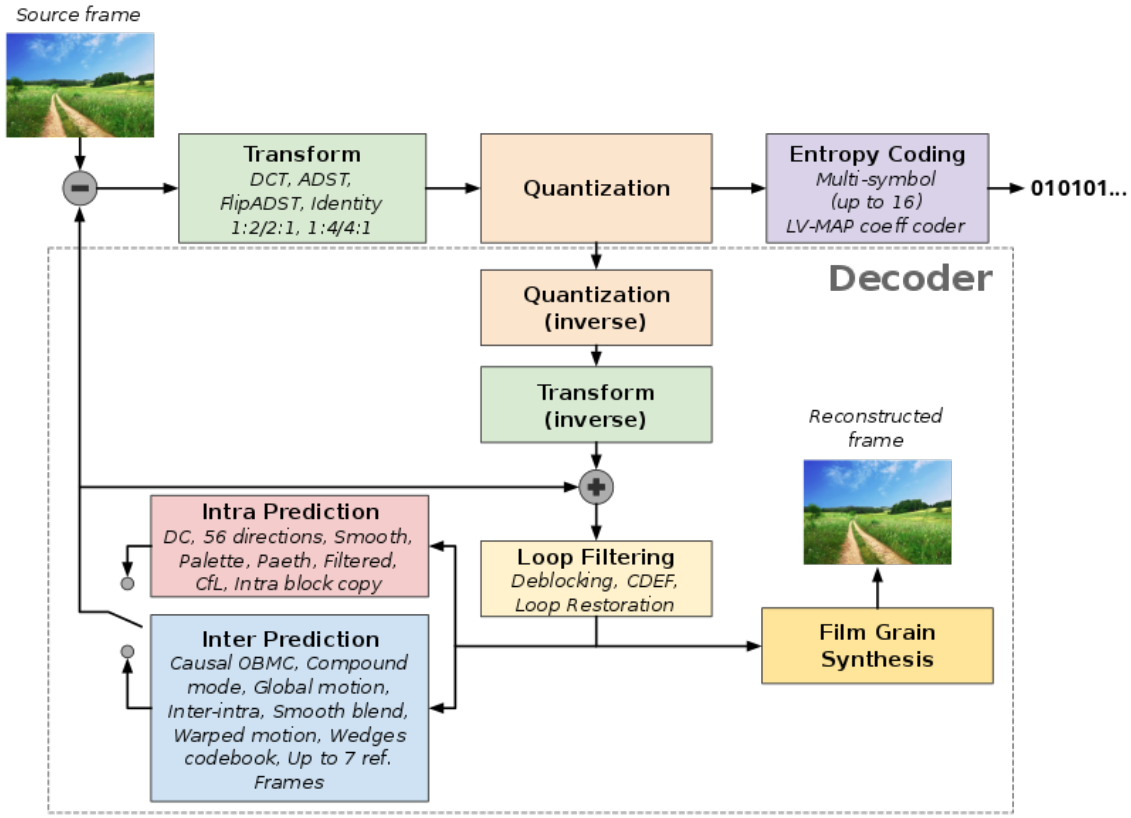

Figure 2.2: Processing stages of an AV1 encoder with relevant technologies associated with each stage. [6]

All these improvements allows AV1 to perform better than VP9. [5]
In order to test the performance and compare it with VP9 and HEVC, several videos of different types and size are used.
The results are collected in the following tables(2.3):

TABLE I
BDRATE(%) OF AV1 IN COMPARISON WITH LIBVPX VP9 ENCODER

Metric \ Set	1080p	1080p-screen	720p	360p	Average
PSNR-Y	-26.81	-34.99	-28.19	-26.15	-28.07
PSNR-Cb	-31.27	-45.86	-25.42	-23.77	-30.10
PSNR-Cr	-31.07	-42.18	-27.89	-31.04	-31.80

TABLE II
BDRATE(%) OF AV1 IN COMPARISON WITH X265 HEVC ENCODER

Metric \ Set	1080p	1080p-screen	720p	360p	Average
PSNR-Y	-21.39	-25.97	-25.99	-20.00	-22.75
PSNR-Cb	-40.23	-47.57	-36.87	-34.89	-39.18
PSNR-Cr	-40.13	-41.87	-38.27	-41.16	-40.17

Figure 2.3: Performance comparison of AV1, VP9 and HEVC. [5]

Several studies analyze the performance of AV1 in relation to other codecs. The results show that AV1 improve VP9 by 30% in all planes and it is even better of HEVC [5], moreover, AV1 encoded videos have a better PSNR even if they have a higher encoding time than the others.(2.4) [8]
Moreover, although HEVC is better with lower quality video, AV1 is better with higher quality video. [7]

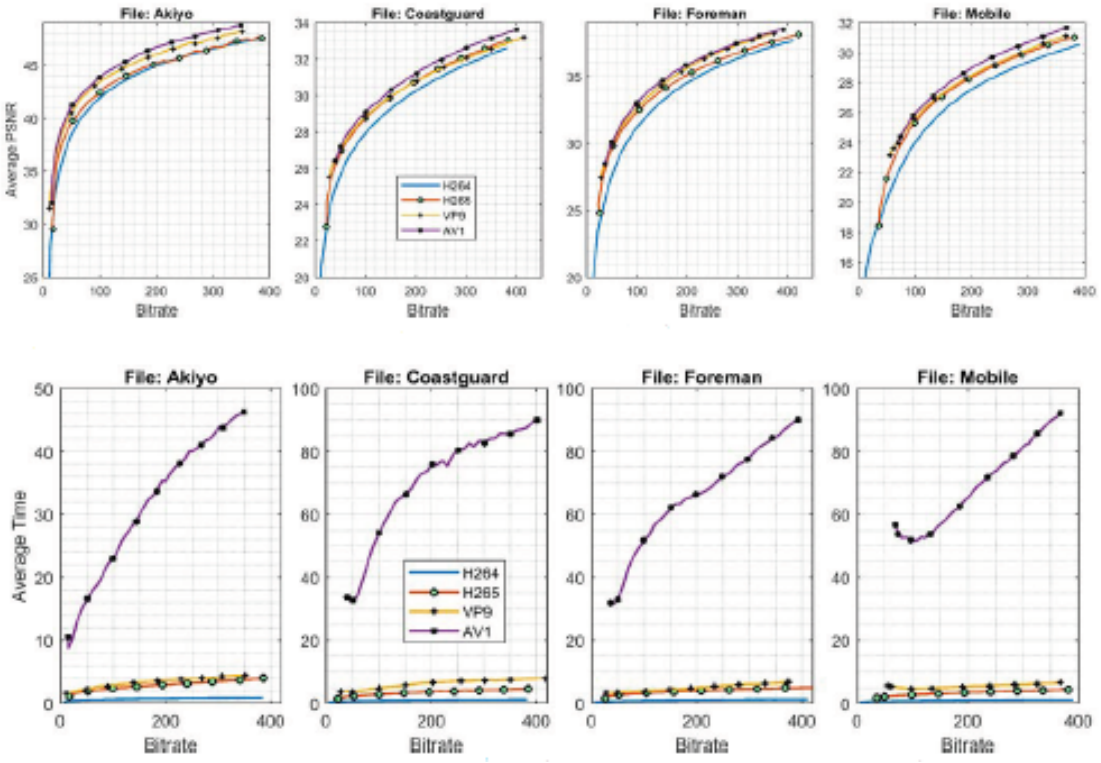

Figure 2.4: Video coding formats average PSNR and encoding time for default parameters. [8]

Chapter 3

Wiener Filter

AV1 is an open source format and thanks to this we were able to do some analysis on the codec. The GPROF tool, allowed us to perform several tests on the codec with different input files and after a careful analysis of the AV1 codec profiling, we decided to focus our attention on in-loop filters, in particular on one of the loop restoration filter: the Wiener Filter.
Several In-Loop filters are applied to a decoded frame. At the beginning, their goal was the improvement of a frame after that it was reconstructed and after that it suffered degradation due to the coding and the transmission.
But this means that the same filter can be used to preserve the information that have to be reconstructed after the trasmission. So the Information given by the filter is sent inside the codified bit-stream and help the decoder to rebuilt the received frame.
In the AV1 codec, after the usual deblocking loop filter, three different switchable loop restoration filters are applied, these filters are separable symmetric Wiener Filters, dual self-guided filters and domain transform recursive filters. [9]

3.1 Separable Symmetric Wiener Filter

After having searched in literature articles about loop filters and after having seen the absence of articles about an hardware architecture for Wiener Filter, it was decided to deepen the analysis for this kind of structure, in order to have all the necessary elements to realize a device able to perform the filter algorithm. In detail, our project is based on the work presented in an article describing the filter algorithm from a mathematical point of view, and then on the code of the AV1 codec that implements the same algorithm in software.

3.1.1 Article

In this section the algorithm that is implemented by the Wiener Filter and which is described in the article **"A switchable loop-restoration with side-information framework for the emerging AV1 video codec"** is analyzed.
The key idea of the Wiener Filter is that all the pixels of a degraded frame can be reconstructed through the pixels surrounding them. In particular we analyze the pixels in a w *x* w window around the pixel that need to be recontructed, where $\boldsymbol{w = 2r + 1}$ with r integer number.
The key elements are:

- $\boldsymbol{H = E[XX^T]}$: the autocovariance of x which can be rewritten as:

$$H = \begin{bmatrix} H_{00} & H_{01} & \dots & H_{0,w-1} \\ H_{10} & H_{11} & \dots & H_{1,w-1} \\ \dots & \dots & \dots & \dots \\ H_{w-1,0} & H_{w-1,1} & \dots & H_{w-1,w-1} \end{bmatrix} \tag{3.1}$$

  in which each $\boldsymbol{H_{ij}}$ element is a matrix of size w *x* w.

- $\boldsymbol{M = E[YX^T]}$: the cross correlation of x with the source y which can be rewritten as:

$$M = \begin{bmatrix} M_0 & M_1 & \dots & M_{w-1} \end{bmatrix} \tag{3.2}$$

  in which each $\boldsymbol{M_i}$ element is a vector of size *1* *x* w.

- $\boldsymbol{F = H^{-1}M}$: a 2D filter taps in column-vectorized form of size w^2x1 which can be rewritten as:
  $\boldsymbol{F = column\_vectorize[ab^T]}$, where **a** and **b** are vertical and horizontal filters of size w *x 1*.

- A P mixing matrix of size w x r.

$$P = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & 1 \\ -2 & -2 & \dots & -2 \\ \dots & \dots & \dots & 1 \\ \dots & \dots & \dots & \dots \\ 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 0 \end{bmatrix} \tag{3.3}$$

In addition several constraints are added to F, in particular F has to be separable, so that horizontal and vertical filtering can be applied separately. In this way:
$a(i) = a(w-1-i)$ and $b(i) = b(w-1-i)$ for $i = 0, 1, ..., r-1$
$\sum a(i) = \sum b(i) = 1$
The key idea is to start from a first guess of horizontal and vertical filters, keeping one of these fixed, we calculate the other one following different steps. Once the filter value has been calculated, it is kept fixed to calculate the other one.
In this way, the overall algorithm can be divided in two main steps that are subdivided into 4 more steps each:
1) **Update a**, fixing b:

- U = $\sum_{i=0}^{w-1} \sum_{j=0}^{w-1} b(i)b(j)H_{ij}P$ of size w x r;
- z = $\sum_{i=0}^{w-1} b(i)M_iP - U_r$ of size *1* x r;
- $\widehat{a}^T = z.(P^TU)^{-1}$ where $\widehat{a}^T = [a(0)a(1)\dots a(r-1)]$
- a = $P\widehat{a} + Z_r$ with Z_r=$[0\dots1\dots0]$ of size *1* x w.

2) **Update b**, fixed a:

- U=VP where V matrix of size w x w, such that $V_{ij} = a^TH_{ij}a$;
- $z = tP - U_r$ of size 1 x r, where $t_i = M_ia$ of size *1* x w.

- $\widehat{b}^T = z.(P^T U)^{-1}$ where $\widehat{b}^T = [b(0)b(1)\dots b(r-1)]$

- b = $P\widehat{b} + Z_r$ with Z_r=$[0\dots1\dots0]$ of size *1 x w*. [9]

3.1.2 Code

In the file pickrst.c of the AV1 codec, in function **search_wiener**, the algorithm described in the previous section is implemented.
The code that implements the Wiener Filter will be analyzed, but of the whole code we have analyzed only the main functions that will be useful for the realization of the hardware architecture. Several design choices have been adopted to try to realize as faithfully as possible the algorithm described in the article, in detail:

- Instead of using matrices, vectors of different sizes are implemented and used as matrices by means of loops with appropriate indexes.
- The type of the different vectors are **int32_t** and **int64_t**, this means that these dimension are selected to avoid overflow and also that the code works only with integer numbers.
- In order to work with integer numbers, we use a scale factor equal to $\mathbf{2^{16}}$
- The approximations are executed by means of division to power of two, that are equal to shift and so truncation of the final bits. This implies that we have a loss of informations, but thanks to the scale factor, the losses are not relevant.

In order to better understand the code, the constant values defined at the beginning of the code, used to parameterize the code itself, are shown.

- **WIENER_HALFWIN** = 3
- **WIENER_WIN** = 2 * WIENER_HALFWIN + 1
- **WIENER_WIN2** = (WIENER_WIN) * (WIENER_WIN)
- **WIENER_HALFWIN1** = WIENER_HALFWIN + 1
- **WIENER_FILT_PREC_BITS** = 7
- **WIENER_FILT_STEP** = 1 << WIENER_FILT_PREC_BITS
- **WIENER_FILT_TAP0_MIDV** = 3

- **WIENER_FILT_TAP1_MIDV** = -7
- **WIENER_FILT_TAP2_MIDV** = 15
- **WIENER_FILT_TAP3_MIDV** = WIENER_FILT_STEP - 2 * (WIENER_FILT_TAP0_MIDV + WIENER_FILT_TAP1_MIDV + WIENER_FILT_TAP2_MIDV)
- **WIENER_TAP_SCALE_FACTOR** = (int64_t)1 << 16
- **NUM_WIENER_ITERS** = 5

After the computation of the matrices **H** and **M**, the main function starts with several functions to compute the final vectors **a** and **b**.
The main sub-functions of **search_wiener** are analyzed below:

wiener_decompose_sep_sym

This is the starting function in which the two vectors **a** and **b** are initialized with a starting values. In addition it is also created the reference of the wanted values of **H** and **M**. After this initializing part, the algorithm start with a loop in which vectors **a** and **b** are calculated in an iterative way through the two main function **update_a_sep_sym** and **update_b_sep_sym**.

```
static int wiener_decompose_sep_sym(int wiener_win, int64_t *M, int64_t
    *H, int32_t *a, int32_t *b) {
  static const int32_t init_filt[WIENER_WIN] = {
    WIENER_FILT_TAP0_MIDV, WIENER_FILT_TAP1_MIDV, WIENER_FILT_TAP2_MIDV
   ,WIENER_FILT_TAP3_MIDV, WIENER_FILT_TAP2_MIDV,
   WIENER_FILT_TAP1_MIDV,WIENER_FILT_TAP0_MIDV};
                .   .   .
  for (i = 0; i < wiener_win; i++) {
    a[i] = b[i] =
        WIENER_TAP_SCALE_FACTOR / WIENER_FILT_STEP * init_filt[i +
   plane_off];
  }
  for (i = 0; i < wiener_win; i++) {
    Mc[i] = M + i * wiener_win;
    for (j = 0; j < wiener_win; j++) {
      Hc[i * wiener_win + j] =
          H + i * wiener_win * wiener_win2 + j * wiener_win;
    }
  }
  iter = 1;
  while (iter < NUM_WIENER_ITERS) {
    update_a_sep_sym(wiener_win, Mc, Hc, a, b);
```

```
19      update_b_sep_sym(wiener_win, Mc, Hc, a, b);
20      iter++;
21    }
22    return 1;
23  }
```

update_a_sep_sym

This is one of the two main functions of the code. We can divide this function in four different parts:

- **Vectors A and B are computated**
  In detail, **A** is a vector of 4 positions, the value of each position is calculated iteratively by adding the previous value of the same position to the new one. The different values are calculated as the product, properly scaled, of a certain value of **M** and the corresponding value of the initial vector **b**.
  The vector **B**, of size 16, is also calculated in the same way, but now, the product is between a value of matrix **H** and two values of the initial vector **b**.

- **Normalization enforcement**
  Vectors **A** and **B**, previously computed, are normalized using a sequence of operations involving the values of the vectors themselves. The goal of these operations is to compress the two vectors in order to adapt them to the linear system of equations. So, in the end, the final sizes of the vectors **A** and **B** are 1x3 and 3x3 respectively[1].

- **Linsolve function**
  The **linsolve function** is called, in this function vector **S** is computated and returned at the main function. Vector **S** is the first half of the final vector **a**.

- **Final a computation** Starting from vector **S**, vector **a** is calculated respecting the symmetry constraints and the scale factor.

```
1  static AOM_INLINE void update_a_sep_sym(int wiener_win, int64_t **Mc,
     int64_t **Hc, int32_t *a, int32_t *b) {
2    int i, j;
3    int32_t S[WIENER_WIN];
4    int64_t A[WIENER_HALFWIN1], B[WIENER_HALFWIN1 * WIENER_HALFWIN1];
5    const int wiener_win2 = wiener_win * wiener_win;
6    const int wiener_halfwin1 = (wiener_win >> 1) + 1;
7    memset(A, 0, sizeof(A));
8    memset(B, 0, sizeof(B));
```

[1]Even if A and B are defined as vectors, they are used as if they were matrices using indices in an appropriate way.

```
  for (i = 0; i < wiener_win; i++) {
    for (j = 0; j < wiener_win; ++j) {
      const int jj = wrap_index(j, wiener_win);
      A[jj] += Mc[i][j] * b[i] / WIENER_TAP_SCALE_FACTOR;
    }
  }
  for (i = 0; i < wiener_win; i++) {
    for (j = 0; j < wiener_win; j++) {
      int k, l;
      for (k = 0; k < wiener_win; ++k) {
        for (l = 0; l < wiener_win; ++l) {
          const int kk = wrap_index(k, wiener_win);
          const int ll = wrap_index(l, wiener_win);
          B[ll * wiener_halfwin1 + kk] +=
              Hc[j * wiener_win + i][k * wiener_win2 + l] * b[i] /
              WIENER_TAP_SCALE_FACTOR * b[j] / WIENER_TAP_SCALE_FACTOR;
        }
      }
    }
  }
  // Normalization enforcement in the system of equations itself
  for (i = 0; i < wiener_halfwin1 - 1; ++i) {
    A[i] -=
        A[wiener_halfwin1 - 1] * 2 +
        B[i * wiener_halfwin1 + wiener_halfwin1 - 1] -
        2 * B[(wiener_halfwin1 - 1) * wiener_halfwin1 + (
    wiener_halfwin1 - 1)];
  }
  for (i = 0; i < wiener_halfwin1 - 1; ++i) {
    for (j = 0; j < wiener_halfwin1 - 1; ++j) {
      B[i * wiener_halfwin1 + j] -=
          2 * (B[i * wiener_halfwin1 + (wiener_halfwin1 - 1)] +
               B[(wiener_halfwin1 - 1) * wiener_halfwin1 + j] -
               2 * B[(wiener_halfwin1 - 1) * wiener_halfwin1 +
                     (wiener_halfwin1 - 1)]);
    }
  }
  if (linsolve_wiener(wiener_halfwin1 - 1, B, wiener_halfwin1, A, S)) {
    S[wiener_halfwin1 - 1] = WIENER_TAP_SCALE_FACTOR;
    for (i = wiener_halfwin1; i < wiener_win; ++i) {
      S[i] = S[wiener_win - 1 - i];
      S[wiener_halfwin1 - 1] -= 2 * S[i];
    }
    memcpy(a, S, wiener_win * sizeof(*a));
  }
}
```

update_b_sep_sym

This is the second main function. It is very similar to the **update a** function, but, in input, it has the vector **a** previously calculated. The subdivision into four parts is the same, each part performs the same operations, also the **linsolve function** is the same. The only change is that in products, instead of multiplying by different values of **b**, it multiplies by different values of **a**. So at the end the final vector **b** is computated.

```
static AOM_INLINE void update_b_sep_sym(int wiener_win, int64_t **Mc,
    int64_t **Hc, int32_t *a, int32_t *b) {
  int i, j;
  int32_t S[WIENER_WIN];
  int64_t A[WIENER_HALFWIN1], B[WIENER_HALFWIN1 * WIENER_HALFWIN1];
  const int wiener_win2 = wiener_win * wiener_win;
  const int wiener_halfwin1 = (wiener_win >> 1) + 1;
  memset(A, 0, sizeof(A));
  memset(B, 0, sizeof(B));
  for (i = 0; i < wiener_win; i++) {
    const int ii = wrap_index(i, wiener_win);
    for (j = 0; j < wiener_win; j++) {
      A[ii] += Mc[i][j] * a[j] / WIENER_TAP_SCALE_FACTOR;
    }
  }
  for (i = 0; i < wiener_win; i++) {
    for (j = 0; j < wiener_win; j++) {
      const int ii = wrap_index(i, wiener_win);
      const int jj = wrap_index(j, wiener_win);
      int k, l;
      for (k = 0; k < wiener_win; ++k) {
        for (l = 0; l < wiener_win; ++l) {
          B[jj * wiener_halfwin1 + ii] +=
              Hc[i * wiener_win + j][k * wiener_win2 + l] * a[k] /
              WIENER_TAP_SCALE_FACTOR * a[l] / WIENER_TAP_SCALE_FACTOR;
        }
      }
    }
  }
  // Normalization enforcement in the system of equations itself
  for (i = 0; i < wiener_halfwin1 - 1; ++i) {
    A[i] -=
        A[wiener_halfwin1 - 1] * 2 +
        B[i * wiener_halfwin1 + wiener_halfwin1 - 1] -
        2 * B[(wiener_halfwin1 - 1) * wiener_halfwin1 + (
    wiener_halfwin1 - 1)];
  }
  for (i = 0; i < wiener_halfwin1 - 1; ++i) {
    for (j = 0; j < wiener_halfwin1 - 1; ++j) {
      B[i * wiener_halfwin1 + j] -=
```

```
            2 * (B[i * wiener_halfwin1 + (wiener_halfwin1 - 1)] +
                 B[(wiener_halfwin1 - 1) * wiener_halfwin1 + j] -
                 2 * B[(wiener_halfwin1 - 1) * wiener_halfwin1 +
                       (wiener_halfwin1 - 1)]);
      }
    }
    if (linsolve_wiener(wiener_halfwin1 - 1, B, wiener_halfwin1, A, S)) {
      S[wiener_halfwin1 - 1] = WIENER_TAP_SCALE_FACTOR;
      for (i = wiener_halfwin1; i < wiener_win; ++i) {
        S[i] = S[wiener_win - 1 - i];
        S[wiener_halfwin1 - 1] -= 2 * S[i];
      }
      memcpy(b, S, wiener_win * sizeof(*b));
    }
}
```

linsolve_wiener

At this point we have this linear system of equations[2]:

$$\begin{pmatrix} A_0 & A_1 & A_2 \\ A_4 & A_5 & A_6 \\ A_8 & A_9 & A_{10} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \tag{3.4}$$

In order to solve it and to find vector of solutions **x**, Gauss elimination method was chosen. We can divide the algorithm into three parts:

- **Partial pivoting**: The goal is to bring the row with the largest pivot to the top of the matrix, so it compare the pivot of the rows and if one pivot is bigger than the other, it swap the corresponding rows.
- **Forward elimination**: Here we convert **A** and **b** to row-echelon form. These two steps are performed alternately twice.
- **Back substitution and store x**: This is the last step, here we calculate the **x** solution properly scaled through back substitution, typical of linear equation system solutions.

```
static int linsolve_wiener(int n, int64_t *A, int stride, int64_t *b,
    int32_t *x) {
  for (int k = 0; k < n - 1; k++) {
    // Partial pivoting: bring the row with the largest pivot to the
    top
```

[2]Note that the real A and b are 4x4 and 1x4 size respectively, but we only use the 3x3 size A submatrix and the 1x3 size b subvector.

```
    for (int i = n - 1; i > k; i--) {
      // If row i has a better (bigger) pivot than row (i-1), swap them
      if (llabs(A[(i - 1) * stride + k]) < llabs(A[i * stride + k])) {
        for (int j = 0; j < n; j++) {
          const int64_t c = A[i * stride + j];
          A[i * stride + j] = A[(i - 1) * stride + j];
          A[(i - 1) * stride + j] = c;
        }
        const int64_t c = b[i];
        b[i] = b[i - 1];
        b[i - 1] = c;
      }
    }
    // Forward elimination (convert A to row-echelon form)
    for (int i = k; i < n - 1; i++) {
      if (A[k * stride + k] == 0) return 0;
      const int64_t c = A[(i + 1) * stride + k];
      const int64_t cd = A[k * stride + k];
      for (int j = 0; j < n; j++) {
        A[(i + 1) * stride + j] -= c / 256 * A[k * stride + j] / cd * 256;
      }
      b[i + 1] -= c * b[k] / cd;
    }
  }
  // Back-substitution
  for (int i = n - 1; i >= 0; i--) {
    if (A[i * stride + i] == 0) return 0;
    int64_t c = 0;
    for (int j = i + 1; j <= n - 1; j++) {
      c += A[i * stride + j] * x[j] / WIENER_TAP_SCALE_FACTOR;
    }
    // Store filter taps x in scaled form.
    x[i] = (int32_t)(WIENER_TAP_SCALE_FACTOR * (b[i] - c) / A[i * stride + i]);
  }
  return 1;
}
```

Chapter 4

Basic Wiener Filter

Starting from the concepts seen in the previous chapter, it was decided to create a basic hardware architecture of the Wiener Filter. A basic architecture means a structure that simply executes the basic algorithm of the Wiener Filter. This implies you will work without taking into account the limitations of available hardware and without evaluating the critical paths and combinatorial paths. This structure will be used as a starting point to later realize a hardware architecture for a Wiener Filter that works at high frequency. In this chapter the various blocks that compose the architecture will be analyzed and the design choices will be discussed in detail.

4.1 Initial Design Choices

Looking carefully at the code, it was observed that it is structured to be more easily adaptable to a hardware algorithm. So it was decided to faithfully follow the choices made in the codec. In detail, the parallelism of the inputs has been maintained, the maximum parallelism of the internal operations of the algorithm has been fixed at **64 bit**, as in the code. In addition, to avoid working with decimal digits and therefore with fixed or floating point numbers, the scale factor of $\mathbf{2^{16}}$ has been maintained. Finally, as concerns rounding, several solutions have been adopted to obtain the same results as in the codec and therefore have negligible information losses. Finally, as mentioned above, the algorithm calculates the vectors a and b iteratively, so it was decided to create a architecture that would implement only one iteration, in order to execute the core of the algorithm.

4.2 Top Level

For dimensioning problems, we decided to have as inputs to the architecture the **M** matrix and the appropriate $\boldsymbol{H_{ij}}$ matrices, all of size 7x7 and 64 bit for each element,

all computed by the software, and the initial vector **b** of 7 elements, each of 32 bits. The outputs instead are the two new vectors **a** and **b** both of 7 elements of 32 bits and a done signal that indicate the availability of the output data for the next iteration. The whole architecture(4.1) is divided into two main blocks:

- **Update a**: it calculates the new vector **a** and provides it to both the output buffer and the next block.
- **Update b**: it receives the new vector **a** and provides the new vector **b** to the output buffer.

Once completed, the done signal enables the two tri-state buffers that provide the two vectors to the output. The main FSM(4.2) is very simple, it will start the **Update a** block, once it will provide the "done" signal, the FSM will start the **Update b** block. At the end of the operation it will provide its done signal and then the FSM will provide the main done in output.

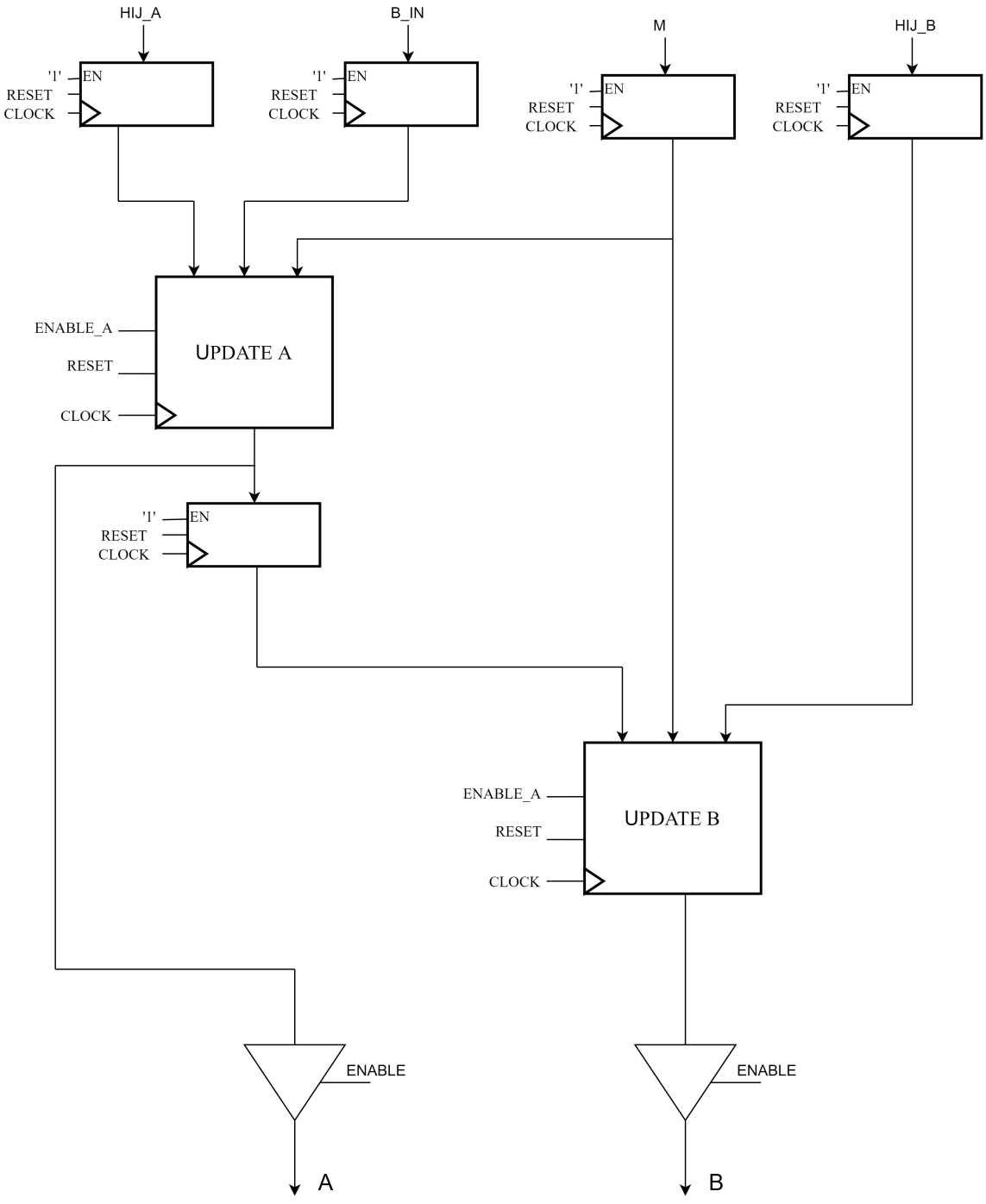

Figure 4.1: Top Level Execution Unit of Wiener Filter architecture.

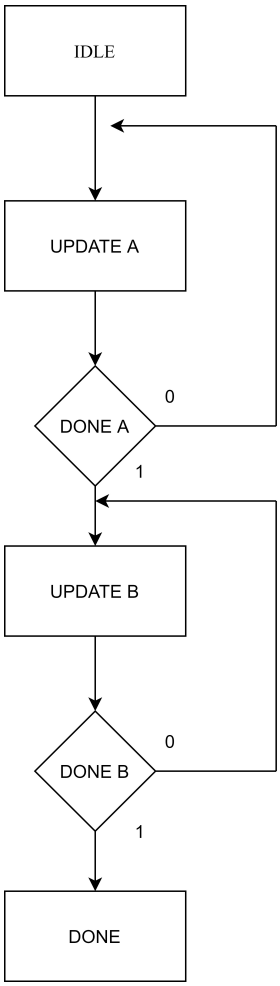

Figure 4.2: Top Level FSM of Wiener Filter architecture.

4.3 Update a

This block(4.3) is one of the two main blocks that compose the device and is divided into several sub-blocks that interact among each other. The inputs are:

- The starting vector **b** composed of 7 elements of 32 bits each.
- The $\boldsymbol{H_{ij}}$ square matrix, a proper sub-matrix of the **H** matrix, of 49 elements, each of 64 bits.
- The whole square matrix **M**, of 49 elements, each of 64 bits.

The output is the new vector **a** of 7 elements, each of 32 bits. First of all, to select the correct values of the vector **b**, of the matrix **M** and for the correct working of

the FSM cycles, two counters of 3 bits each are instantiated. The block algorithm can be divided into two parts that work in parallel. In the first part, the partial square matrices **B** of 16 elements of 64 bits are calculated as the product between the elements of the matrix $\boldsymbol{H_{ij}}$ and two elements of the vector **b**, which we can summarize in $\boldsymbol{H_{ij}b_ib_j}$. At the end of the computation, each partial matrix **B** is accumulated to the sum of the previous ones by means of the **MATRIX SUM** block, to obtain a single matrix **B**. In the second part, the partial vectors **A** composed of 4 elements of 64 bits each are calculated as the product between the elements of the vector $\boldsymbol{M_i}$, selected by the **M SELECTION** block, and an element of the vector **b**, which we can summarize in $\boldsymbol{M_ib_j}$. At the end of each calculation, the partial vector **A** is accumulated at the sum of the previous ones, by means of the **SUM VECTOR** block to obtain a single vector **A**. Once the calculation of the matrix **B** and the vector **A** is finished, they are processed in the **Enforcement** block to obtain a square matrix **B** of 9 elements each of 64 bits and a vector **A** of 3 elements each of 64 bits. At this point we make a change in the nomenclature, as happens in the code, to have the same of a linear system $\boldsymbol{Ax = B}$. So now we have a 3x3 matrix **A** and a 1x3 vector **B**. In the following steps, which will be analyzed in detail later, the linear system is solved to obtain the vector of solutions **x**. The blocks of **PIVOTING** and **FORWARD ELIMINATION** are used twice, on different rows of the matrix and on different elements of the vector, thanks to the use of two multiplexers placed at the entrance of the two blocks. At the end of the two cycles, through the **BACK-SUBSTITUTION STORING** block, the vector of the solutions **x** of 3 elements, each of 32 bits, is calculated. Finally, to obtain the final vector **a** of 7 elements, each of 32 bits, the symmetry constraints are applied to the vector **x** to obtain the last three values and the equality condition and scale factor to obtain the central element. The FSM (4.4) is managed by the two counter signals. After the block enable signal, we find a loop, the execution unit performs the operations of the **SUM J** state until counter **J** reaches 6. Once the first cycle is finished, we find two nested loops. The FSM goes into the **SUM I** state and a further **SUM J** state, at this point the FSM remains in the **SUM J** state until the Counter-J reaches the value of 6, at that point, if the Counter-I has not yet reached the value of 6, the second cycle of **SUM J** starts again. Once the two nested loops are completed, the remaining states are executed in succession, without further conditions. In detail, the **ENABLE ENFORCEMENT** state enables enforcement block operations, the **K0 SEL0** and **K1 SEL1** states direct multiplexers for correct operation and input, and after the execution of pivoting and forward elimination blocks, the **DONE A** state signals that the output data is correct and the **ENABLE B** state enables the next block.

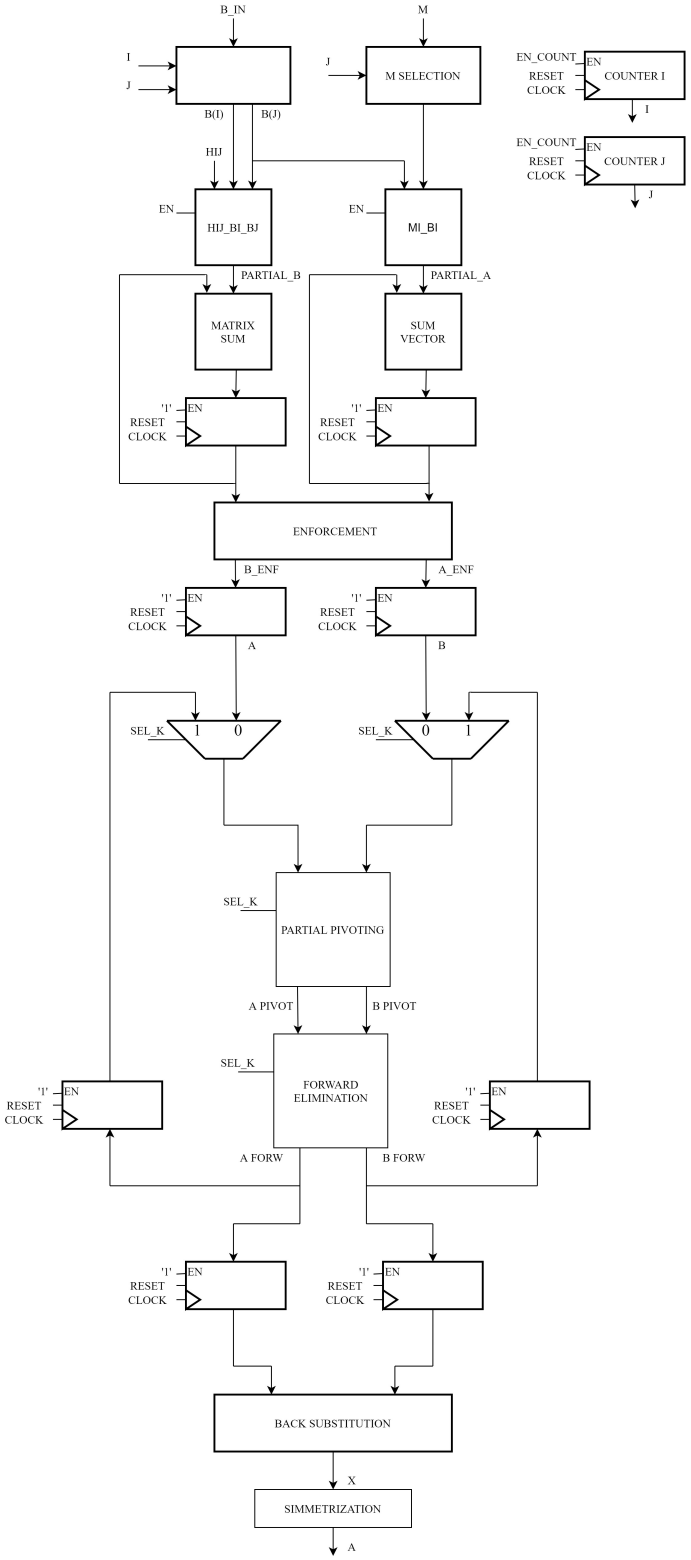

Figure 4.3: Update a Execution Unit.

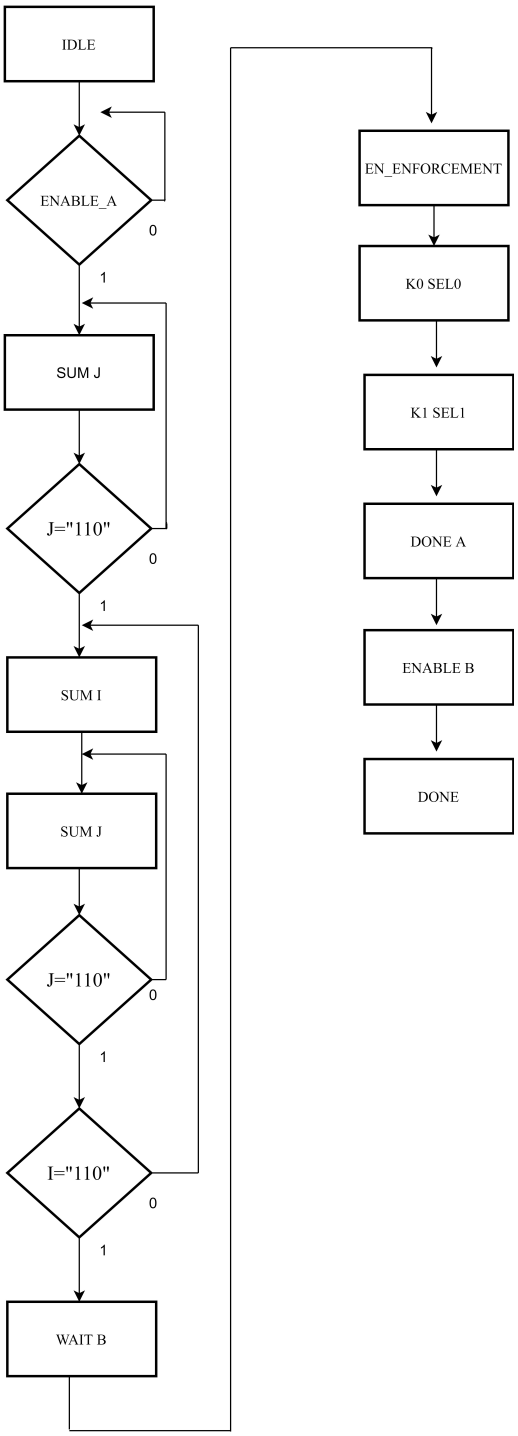

Figure 4.4: Update a FSM.

4.4 $H_{ij}b_ib_j$

The product between the elements of the $\boldsymbol{H_{ij}}$ matrix and the two elements of vector **b** is computed in this block.(4.5) The inputs are the $\boldsymbol{H_{ij}}$ matrix, and the two elements of the **b** vector already selected. Before proceeding with the operations, since later right shifts will be made and that involve truncation with the loss of LSB, the absolute values of the two input **b** values are computed. In this way, at the end of the operations, we will obtain the same approximate values of the codec, then to the final result will be applied the correct sign evaluating the signs of the two values of **b**, applying the two's complement in case of negative sign. In the matrix of the partial products each product between the element $\boldsymbol{H_{ij}(j,i)}$ (matrix elements are selected by scanning the matrix column by column) and the two absolute values of the elements of the input vector **b** is saved. After each multiplication of the element of **b**, a right shift of 16 positions (division by $\mathbf{2^{16}}$) is made to respect the coherence with the scale factor and to avoid overflow.

```
1 FOR I IN 0 TO 6 LOOP
2   FOR J IN 0 TO 6 LOOP
3 PP_ABS(I,J)<=(shift_right(((shift_right((H_IN(J,I)*ABS_BIN1),16)(63
      DOWNTO 0))*ABS_BIN2),16));
```

Each element of the square output matrix consisting of 16 elements of 64 bits is calculated by the sum of specific elements of the partial product matrix.

```
1 B0<=PP(0,0)(63 DOWNTO 0)+PP(0,6)(63 DOWNTO 0)+PP(6,0)(63 DOWNTO 0)+
      PP(6,6)(63 DOWNTO 0);
2 B4<=PP(0,1)(63 DOWNTO 0)+PP(0,5)(63 DOWNTO 0)+PP(6,1)(63 DOWNTO 0)+
      PP(6,5)(63 DOWNTO 0);
3 B8<=PP(0,2)(63 DOWNTO 0)+PP(0,4)(63 DOWNTO 0)+PP(6,2)(63 DOWNTO 0)+
      PP(6,4)(63 DOWNTO 0);
4 B12<=PP(0,3)(63 DOWNTO 0) + PP(6,3)(63 DOWNTO 0);
5
6 B1<=PP(1,0)(63 DOWNTO 0)+PP(1,6)(63 DOWNTO 0)+PP(5,0)(63 DOWNTO 0)+
      PP(5,6)(63 DOWNTO 0);
7 B5<=PP(1,1)(63 DOWNTO 0)+PP(1,5)(63 DOWNTO 0)+PP(5,1)(63 DOWNTO 0)+
      PP(5,5)(63 DOWNTO 0);
8 B9<=PP(1,2)(63 DOWNTO 0)+PP(1,4)(63 DOWNTO 0)+PP(5,2)(63 DOWNTO 0)+
      PP(5,4)(63 DOWNTO 0);
9 B13<=PP(1,3)(63 DOWNTO 0) + PP(5,3)(63 DOWNTO 0);
10
11 B2<=PP(2,0)(63 DOWNTO 0)+PP(2,6)(63 DOWNTO 0)+PP(4,0)(63 DOWNTO 0)+
      PP(4,6)(63 DOWNTO 0);
12 B6<=PP(2,1)(63 DOWNTO 0)+PP(2,5)(63 DOWNTO 0)+PP(4,1)(63 DOWNTO 0)+
      PP(4,5)(63 DOWNTO 0);
13 B10<=PP(2,2)(63 DOWNTO 0)+PP(2,4)(63 DOWNTO 0)+PP(4,2)(63 DOWNTO 0)
     + PP(4,4)(63 DOWNTO 0);
14 B14<=PP(2,3)(63 DOWNTO 0) + PP(4,3)(63 DOWNTO 0);
15
```

```
16 B3<=PP(3,0)(63 DOWNTO 0) + PP(3,6)(63 DOWNTO 0);
17 B7<=PP(3,1)(63 DOWNTO 0) + PP(3,5)(63 DOWNTO 0);
18 B11<=PP(3,2)(63 DOWNTO 0) + PP(3,4)(63 DOWNTO 0);
19 B15<=PP(3,3)(63 DOWNTO 0);
20                                    . . .
21 H_IJ_BIJ_OUT<=((B0,B1,B2,B3),(B4,B5,B6,B7),(B8,B9,B10,B11), (B12,
      B13, B14, B15));
```

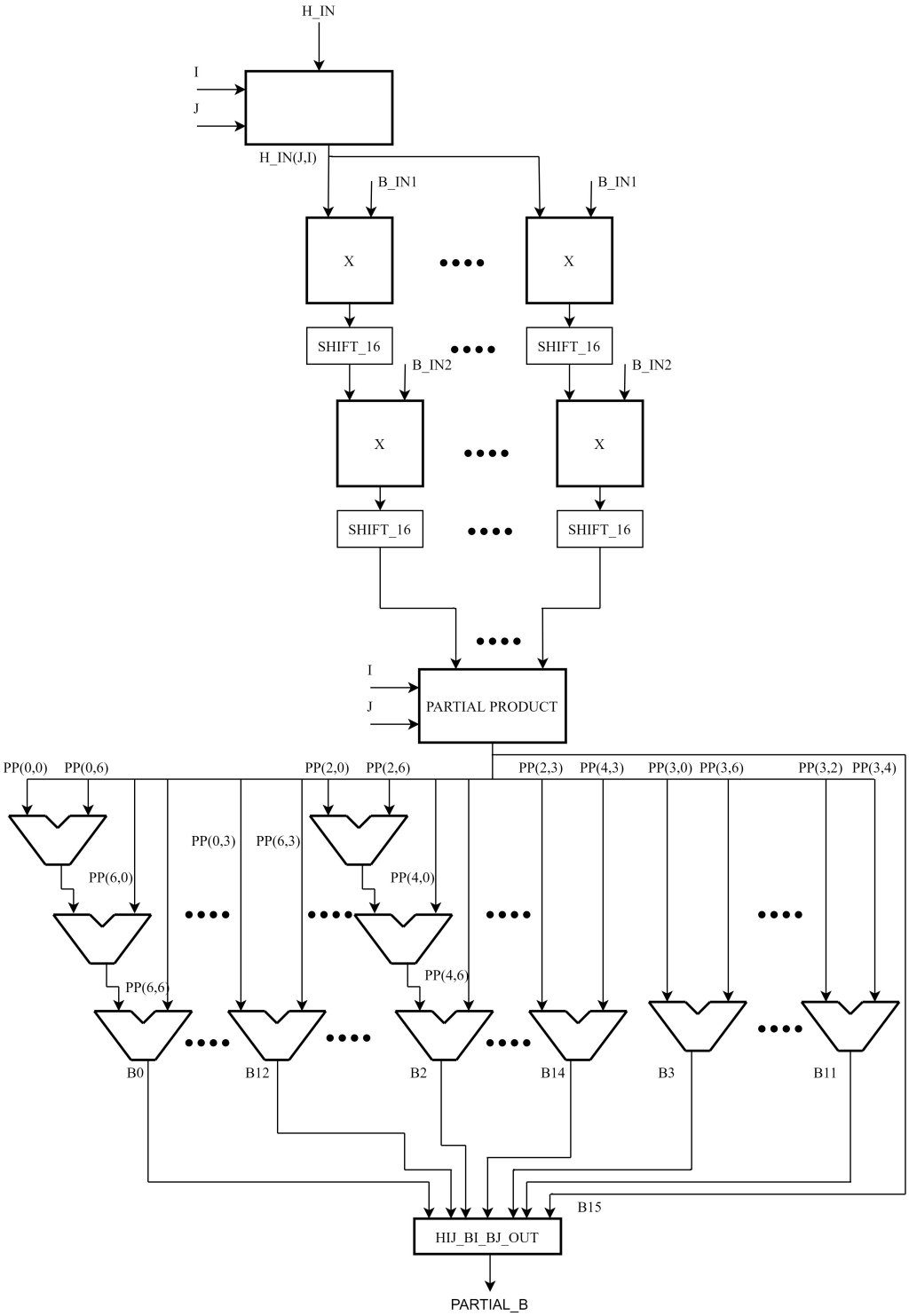

Figure 4.5: $H_{ij}b_ib_j$ Execution Unit.

4.5 $M_i b_i$

This block (4.6) follows the same procedure as the previous block. The inputs are the $\boldsymbol{M_i}$ vector and an element of the **b** vector, previously selected. Again, the absolute value of the $\boldsymbol{b_i}$ value is computed for the same approximation as in the previous block. Each product between a value of the vector $\boldsymbol{M_i}$ and $\boldsymbol{b_i}$, properly shifted, are saved in a matrix of partial products. After applying the correct signs to the partial products, the 4 elements of the output vector, of 64 bits, are calculated by summing the elements of the matrix of partial products.

```
1 FOR I IN 0 TO 2 LOOP
2   PP_ABS(I)<=shift_right((M_IN(I)*ABS_B),16);
3   PP_ABS(6-I)<=shift_right((M_IN(6-I)*ABS_B),16);
4             .   .   .
5   MIJ_BI_OUT(I)<=PP(I)(63 DOWNTO 0) + PP(6-I)(63 DOWNTO 0);
6 END LOOP
7 PP_ABS(3)<=shift_right((M_IN(3)*ABS_B),16);
8             .   .   .
9 MIJ_BI_OUT(3)<=PP(3)(63 DOWNTO 0);
```

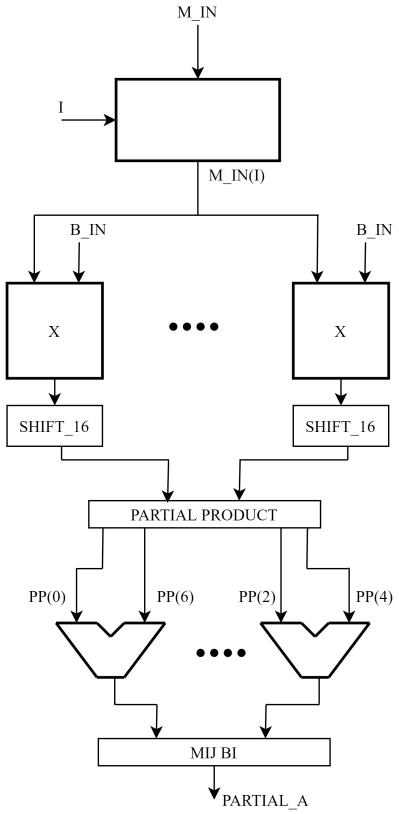

Figure 4.6: $M_i b_i$ Execution Unit.

4.6 Enforcement

The block **Enforcement**(4.7), has in input the vector **A** and the square matrix **B** previously calculated. Through simple operations of addition, subtraction and shift between the same elements of the two inputs, we obtain the same amount of elements as output, but with different dimensions, in detail we have the vector **A ENF** of 3 elements of 64 bit and the 3x3 matrix **B ENF** of which each element has a size of 64 bit.

```
1 FOR I IN 0 TO 2 LOOP
2 A_ENF(I)<=A_IN(I)-shift_left(A_IN(3),1) - B_IN(I,3) + shift_left(
    B_IN(3,3),1);
3 END LOOP;
4
5 FOR I IN 0 TO 2 LOOP
6   FOR J IN 0 TO 2 LOOP
7     ARGUMENT(I,J)<=B_IN(I,3) + B_IN(3,J) - shift_left(B_IN(3,3),1);
8     B_ENF(I,J)<=B_IN(I,J) - (shift_left(ARGUMENT(I,J),1));
9   END LOOP;
10 END LOOP;
```

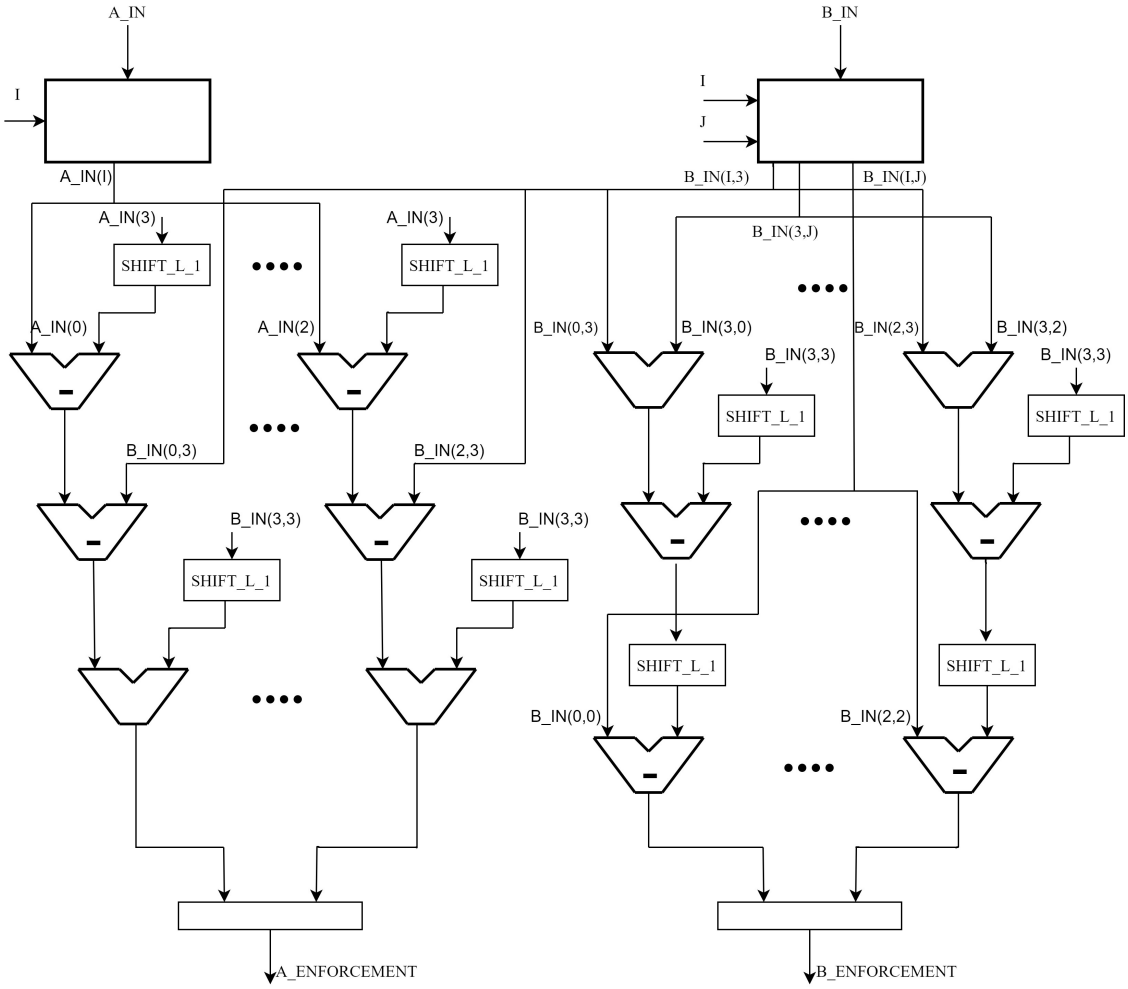

Figure 4.7: Enforcement Execution Unit.

4.7 Solution of Linear System

At this point, through a variable change, we will call the vector **A ENF**, **B** and the matrix **B ENF**, **A**. In this way we have to solve the following linear system:

$$\begin{pmatrix} A_0 & A_1 & A_2 \\ A_4 & A_5 & A_6 \\ A_8 & A_9 & A_{10} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} \tag{4.1}$$

that can be simple rewritten as:

$$Ax = B \tag{4.2}$$

We can follow two approaches to find the vector of solutions **x**. The first one consists in solving the equation:

$$x = BA^{-1} \tag{4.3}$$

Moving in this direction, we have to calculate the inverse of matrix **A**. Assuming you want to implement a block that performs matrix **A** inversion, you would have matrix **A** at the input. The calculation of the inverse matrix implies the computation of the $\boldsymbol{C_{ij}}$ coefficients. Since each element of matrix **A** is 64 bits, each $\boldsymbol{C_{ij}}$ coefficient will be 128 bits size. After that we proceed to calculate the determinant of the matrix **A**, through the Rule of Sarrus, obtaining a value of 192 bits. At this point, once the matrix of coefficients has been transposed, an **Adj** matrix is obtained. Finally, the division between the **Adj** matrix and the determinant should be performed. This results in an $\boldsymbol{A^{-1}}$ 3x3 matrix of 192 bits.

```
1  C00<=(IN_MATRIX(1,1)* IN_MATRIX(2,2)) - (IN_MATRIX(2,1)*IN_MATRIX
       (1,2));
2  C01<=-((IN_MATRIX(1,0)* IN_MATRIX(2,2)) - (IN_MATRIX(2,0)*IN_MATRIX
       (1,2)));
3  C02<=(IN_MATRIX(1,0)* IN_MATRIX(2,1)) - (IN_MATRIX(2,0)*IN_MATRIX
       (1,1));
4  C10<=-((IN_MATRIX(0,1)* IN_MATRIX(2,2)) - (IN_MATRIX(2,1)*IN_MATRIX
       (0,2)));
5  C11<=(IN_MATRIX(0,0)* IN_MATRIX(2,2)) - (IN_MATRIX(2,0)*IN_MATRIX
       (0,2));
6  C12<=-((IN_MATRIX(0,0)* IN_MATRIX(2,1)) - (IN_MATRIX(2,0)*IN_MATRIX
       (0,1)));
7  C20<=(IN_MATRIX(0,1)* IN_MATRIX(1,2)) - (IN_MATRIX(1,1)*IN_MATRIX
       (0,2));
8  C21<=-((IN_MATRIX(0,0)* IN_MATRIX(1,2)) - (IN_MATRIX(1,0)*IN_MATRIX
       (0,2)));
9  C22<=(IN_MATRIX(0,0)* IN_MATRIX(1,1)) - (IN_MATRIX(1,0)*IN_MATRIX
       (0,1));
10
11 C<=((C00,C01,C02),(C10, C11,C12),(C20,C21,C22));
12
13 DETERMINANT<= (IN_MATRIX(0,0)*C00)+(IN_MATRIX(0,1)*C01)+(IN_MATRIX
       (0,2)*C02);
14
15 TRANSPOSITION : TRANSPOSITION_3X3 PORT MAP(IN_MATRIX=>C, OUT_MATRIX
       =>ADJ);
16 RES : DIVISION PORT MAP (DIVIDEND => ADJ, DIVISOR => DETERMINANT,
       QUOTIENT => OUT_MATRIX);
```

As can be observed, this choice implies the execution of operations with elements of excessive size and in particular when performing the division, this also implies the implementation of a divider of incompatible dimensions with the entire system.

Moreover, since this choice is not made by the codec, we are not able to guarantee that every data can be correctly represented on 64 bits, so we are forced to extend the number of bits every time we perform a multiplication operation. To overcome the problem of the size of the divider, an alternative solution could be to avoid the division and use of fixed or floating point representations, using approximate results by applying the following approach. Taking advantage that dividing by powers of two simply means applying a shift to the right of n positions on the data, we make the following considerations:

$$\frac{Adj}{det} = 1 \Rightarrow det = Adj$$
$$\frac{1}{2} < \frac{Adj}{det} \leq 1 \Rightarrow Adj \leq det < 2Adj$$
$$\frac{1}{2^2} < \frac{Adj}{det} \leq \frac{1}{2} \Rightarrow 2Adj \leq det < 2^2 Adj$$
$$\frac{1}{2^3} < \frac{Adj}{det} \leq \frac{1}{2^2} \Rightarrow 2^2 Adj \leq det < 2^3 Adj$$
and so on, more in general:
$$\frac{1}{2^n} < \frac{Adj}{det} \leq \frac{1}{2^{n-1}} \Rightarrow 2^{n-1} Adj \leq det < 2^n Adj$$

Once the range of values to which the $\frac{Adj}{det}$ ratio belongs has been established, it is approximated to the largest reciprocal of the power of two, then the power exponent is saved in an output matrix. Finally, the product $\boldsymbol{BA^{-1}}$ can be performed simply by shifting the values of the vector $\mathbf{B}$, since the $\boldsymbol{A^{-1}}$ matrix is composed only by reciprocal of powers of two.

```
1  FOR i IN 0 TO 2 LOOP
2    FOR j IN 0 TO 2 LOOP
3
4    ADJ_SHIFT(i,j,0)<=(ADJ(i,j));
5    ADJ_SHIFT(i,j,1)<=(shift_left(ADJ(i,j),1));
6    ADJ_SHIFT(i,j,2)<=(shift_left(ADJ(i,j),2));
7    ADJ_SHIFT(i,j,3)<=(shift_left(ADJ(i,j),3));
8    ADJ_SHIFT(i,j,4)<=(shift_left(ADJ(i,j),4));
9    ADJ_SHIFT(i,j,5)<=(shift_left(ADJ(i,j),5));
10                    .   .   .
11   IF(DET = ADJ_SHIFT(i,j,0)) THEN
12   OUT_MATRIX(i,j)<="00000000";
13   ELSIF (DET>=(ADJ_SHIFT(i,j,1)) AND DET<(ADJ_SHIFT(i,j,2))) THEN
14   OUT_MATRIX(i,j)<="00000001";
15   ELSIF (DET>=(ADJ_SHIFT(i,j,2)) AND DET<(ADJ_SHIFT(i,j,3))) THEN
16   OUT_MATRIX(i,j)<="00000010";
17   ELSIF (DET>=(ADJ_SHIFT(i,j,3)) AND DET<(ADJ_SHIFT(i,j,4))) THEN
18   OUT_MATRIX(i,j)<="00000011";
19   ELSIF (DET>=(ADJ_SHIFT(i,j,4)) AND DET<(ADJ_SHIFT(i,j,5))) THEN
20   OUT_MATRIX(i,j)<="00000100";
21   ELSIF (DET>=(ADJ_SHIFT(i,j,5)) AND DET<(ADJ_SHIFT(i,j,6))) THEN
22   OUT_MATRIX(i,j)<="00000101";
23                    .   .   .
```

It can be seen that this approach has two main problems. The first is the need to use a memory large enough to contain all the cases under examination and, secondly, we can see that as **n** increases, the range of values in which the determinant is included grows, since **n** can reach a value of 192 because the determinant is on 192 bit, the approximation is not acceptable. In order to avoid these problems, it was decided to follow the second direction, which is the same as the codec: solve the linear system by applying the Gaussian Elimination method. This allows us to guarantee that each data is correctly represented on 64 bits, and this allows us to avoid the extention of the number of bits needed to represent the information and also allows us to obtain the same results provided by the codec.

4.8 Gaussian Elimination method

To perform Gaussian Elimination method, we have to work on matrix **A** and vector **B** iteratively. The goal is to obtain a matrix in reduced row echelon form. Since it is a matrix with 3 rows, we will need two iterations. So both the partial pivoting block and the forward elimination block will be used twice with different inputs selected by two multiplexers.

4.8.1 Pivoting 1

In the first pivoting step(4.8), the absolute values of the pivots (first non-zero element of the row) of the three rows of the matrix **A** are compared, the same rows and also the values of vector **B** are sorted according to the increasing order of the pivots. In this case the pivots are $\boldsymbol{A_0}$, $\boldsymbol{A_4}$ and $\boldsymbol{A_8}$, the first elements of each line.

```
1  IF(SELECTION='0') THEN --K=0
2       IF(ABS_A4<ABS_A8) THEN
3               IF(ABS_A0<ABS_A8) THEN
4                OUT_MATRIX<=((IN_MATRIX(2,0),IN_MATRIX(2,1),IN_MATRIX
     (2,2)),(IN_MATRIX(0,0),IN_MATRIX(0,1), IN_MATRIX(0,2)), (
     IN_MATRIX(1,0),IN_MATRIX(1,1),IN_MATRIX(1,2)));
5                OUT_VECTOR<=(IN_VECTOR(2), IN_VECTOR(0), IN_VECTOR(1)
     );
6                ELSE
7                OUT_MATRIX<=((IN_MATRIX(0,0), IN_MATRIX(0,1),
     IN_MATRIX(0,2)),(IN_MATRIX(2,0),IN_MATRIX(2,1),IN_MATRIX(2,2)),
     (IN_MATRIX(1,0),IN_MATRIX(1,1),IN_MATRIX(1,2)));
8                OUT_VECTOR<=(IN_VECTOR(0), IN_VECTOR(2), IN_VECTOR(1)
     );
9             END IF;
10       ELSIF (ABS_A0<ABS_A4) THEN
```

```
11              OUT_MATRIX<=((IN_MATRIX(1,0),IN_MATRIX(1,1),IN_MATRIX
    (1,2)),(IN_MATRIX(0,0),IN_MATRIX(0,1), IN_MATRIX(0,2)), (
    IN_MATRIX(2,0),IN_MATRIX(2,1),IN_MATRIX(2,2)));
12              OUT_VECTOR<=(IN_VECTOR(1), IN_VECTOR(0), IN_VECTOR(2));
13         ELSE
14              OUT_MATRIX<=IN_MATRIX;
15              OUT_VECTOR<=IN_VECTOR;
16         END IF;
```

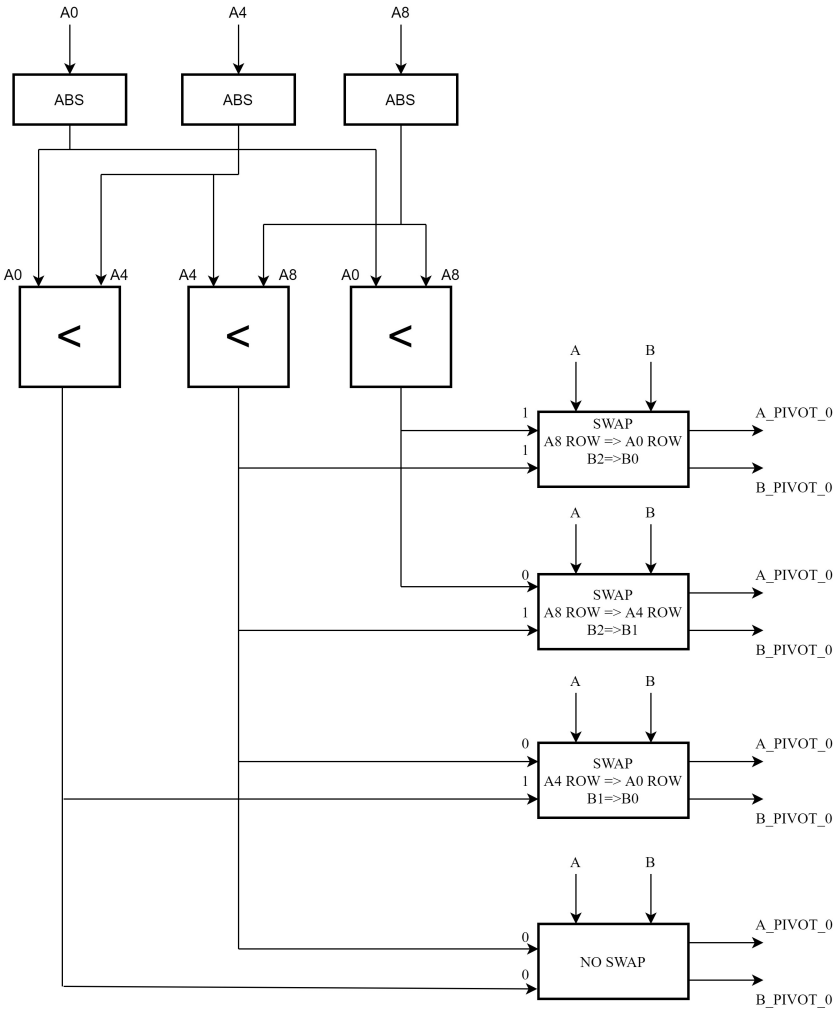

Figure 4.8: Execution Unit of first part of Partial Pivoting block.

4.8.2 Forward Elimination 1

After sorting the rows, the first step of forward elimination(4.9) is taken. The goal is to annul the pivots of the second and third line. To do this, we subtract from each element of the second and third rows of the matrix **A** and from the element of vector **B**, an appropriate value obtained by multiplying the pivot value of the relative row with the corresponding value of the first row and dividing the result by the pivot of the first row. For example, for the second row, we have:

$$A_4 = A_4 - \frac{A_4 A_0}{A_0}$$
$$A_5 = A_5 - \frac{A_4 A_1}{A_0}$$
$$A_6 = A_6 - \frac{A_4 A_2}{A_0}$$

$$b_1 = b_1 - \frac{A_4 b_0}{A_0}$$

```
1  DIVIDEND4<=shift_right(IN_MATRIX(1,0),8)*IN_MATRIX(0,0);
2  DIV4 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0), DIVIDEND=>
       DIVIDEND4(63 DOWNTO 0), QUOTIENT=>QUOTIENT4);
3  A4<=IN_MATRIX(1,0)- shift_left(QUOTIENT4,8);
4  DIVIDEND5<=shift_right(IN_MATRIX(1,0),8)*IN_MATRIX(0,1);
5  DIV5 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
       DIVIDEND5(63 DOWNTO 0),QUOTIENT=>QUOTIENT5);
6  A5<=IN_MATRIX(1,1)- shift_left(QUOTIENT5,8);
7  DIVIDEND6<=shift_right(IN_MATRIX(1,0),8)*IN_MATRIX(0,2);
8  DIV6 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
       DIVIDEND6(63 DOWNTO 0),QUOTIENT=>QUOTIENT6);
9  A6<=IN_MATRIX(1,2)- shift_left(QUOTIENT6,8);
10 DIVIDEND8_K0<=shift_right(IN_MATRIX(2,0),8)*IN_MATRIX(0,0);
11 DIV8_K0 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
       DIVIDEND8_K0(63 DOWNTO 0),QUOTIENT=>QUOTIENT8_K0);
12 A8_K0<=IN_MATRIX(2,0)- shift_left(QUOTIENT8_K0,8);
13 DIVIDEND9_K0<=shift_right(IN_MATRIX(2,0),8)*IN_MATRIX(0,1);
14 DIV9_K0 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
       DIVIDEND9_K0(63 DOWNTO 0),QUOTIENT=>QUOTIENT9_K0);
15 A9_K0<=IN_MATRIX(2,1)- shift_left(QUOTIENT9_K0,8);
16 DIVIDEND10_K0<=shift_right(IN_MATRIX(2,0),8)*IN_MATRIX(0,2);
17 DIV10_K0 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
       DIVIDEND10_K0(63 DOWNTO 0),QUOTIENT=>QUOTIENT10_K0);
18 A10_K0<=IN_MATRIX(2,2)- shift_left(QUOTIENT10_K0,8);
19 B_DIVIDEND1 <= IN_MATRIX(1,0)*IN_VECTOR(0);
20 B_DIV1 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
       B_DIVIDEND1(63 DOWNTO 0),QUOTIENT=>QUOTIENTB1);
21 B1<=IN_VECTOR(1) - QUOTIENTB1;
22 B_DIVIDEND2_K0 <= IN_MATRIX(2,0)*IN_VECTOR(0);
23 B_DIV2_K0 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
       B_DIVIDEND2_K0(63 DOWNTO 0),QUOTIENT=>QUOTIENTB2_K0);
24 B2_K0<=IN_VECTOR(2) - QUOTIENTB2_K0;
25 IF (SELECTION='0') THEN
26    OUT_MATRIX<=((IN_MATRIX(0,0),IN_MATRIX(0,1), IN_MATRIX(0,2)), (
       A4,A5,A6), (A8_K0,A9_K0,A10_K0));
27    OUT_VECTOR<=(IN_VECTOR(0), B1, B2_K0);
```

To maintain the correct value and avoid overflow, we apply a right shift of 8 positions to the first term of multiplication (A_4 in the example) and then a left shift of 8 positions on the term to be subtracted, following the same method applied in the codec.

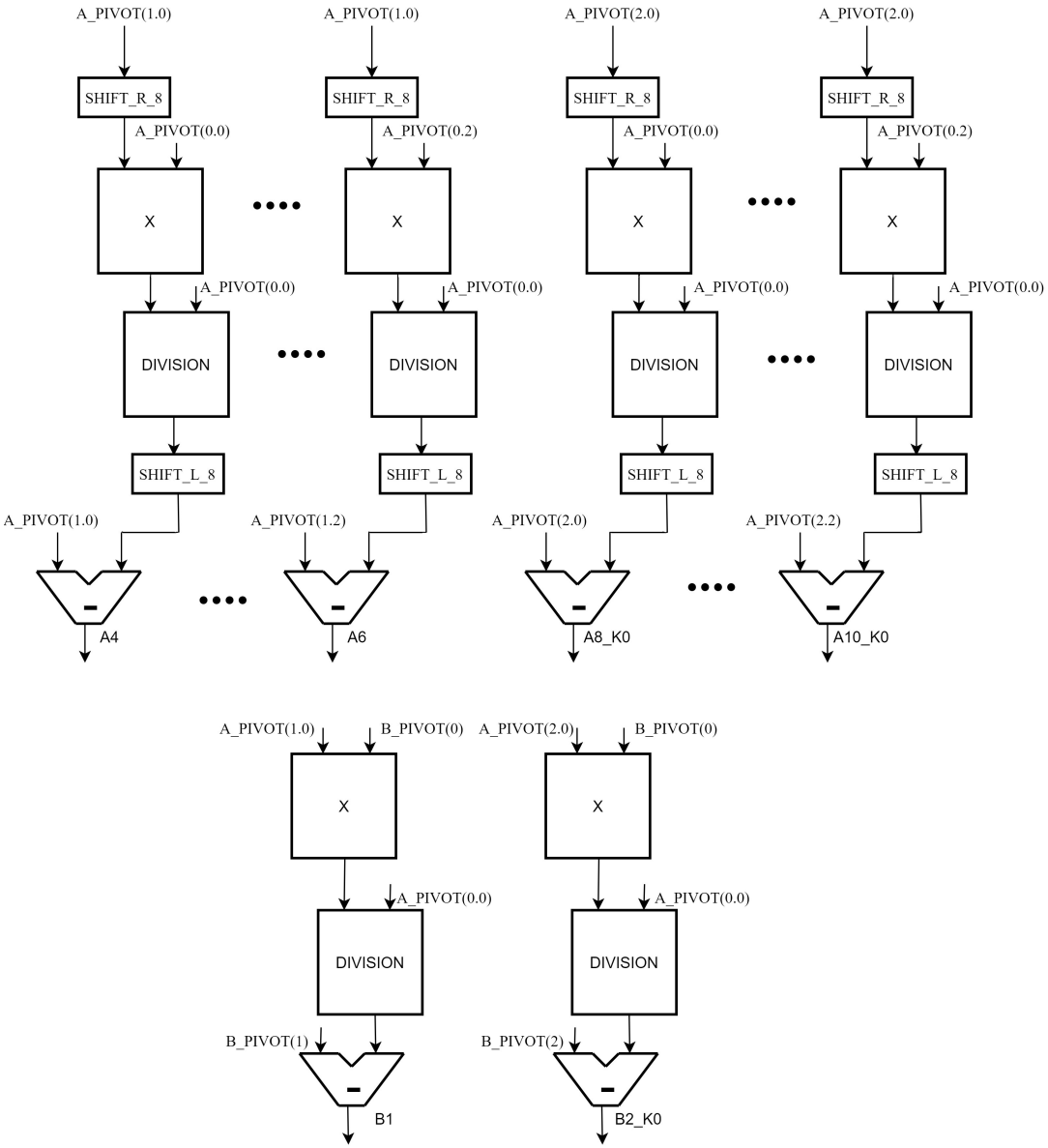

Figure 4.9: Execution Unit of first part of Forward Elimination block.

4.8.3 Pivoting 2

In the second pivoting step(4.10), since the initial values of the first two lines are already null, we consider as pivot the A_5 and A_9 values and, as before, we swap the two rows and the vector values if, evaluating the absolute values, A_9 is bigger than A_5.

```
1 ELSE --K=1
```

```
2     IF(ABS_A5<ABS_A9) THEN
3     OUT_MATRIX<=((IN_MATRIX(0,0), IN_MATRIX(0,1), IN_MATRIX(0,2)),(
      IN_MATRIX(2,0),IN_MATRIX(2,1),IN_MATRIX(2,2)), (IN_MATRIX(1,0),
      IN_MATRIX(1,1),IN_MATRIX(1,2)));
4     OUT_VECTOR<=(IN_VECTOR(0), IN_VECTOR(2), IN_VECTOR(1));
5     ELSE
6    OUT_MATRIX<=IN_MATRIX;
7    OUT_VECTOR<=IN_VECTOR;
8   END IF;
9 END IF;
```

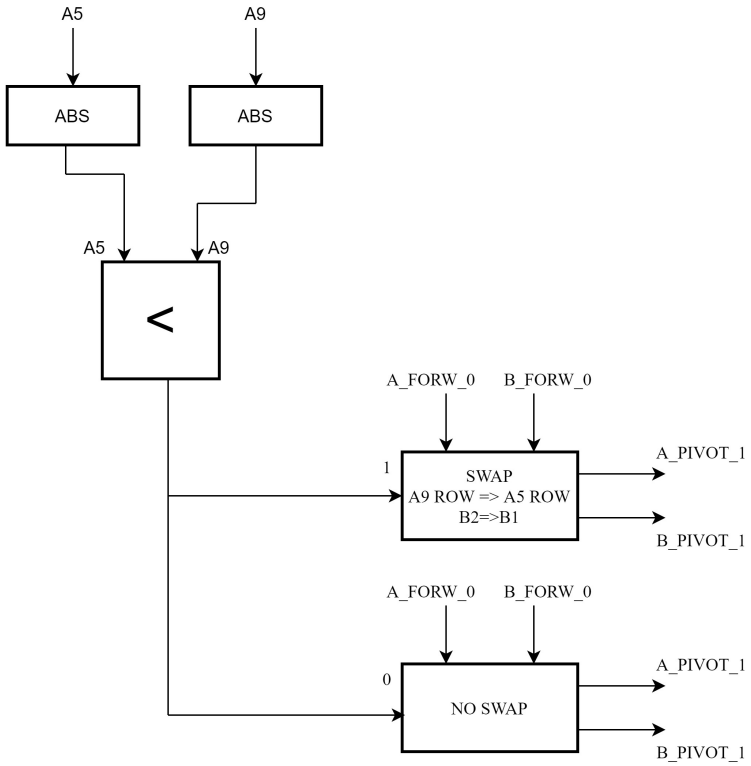

Figure 4.10: Execution Unit of second part of Partial Pivoting block.

4.8.4 Forward Elimination 2

Finally, in the second step of forward elimination(4.11), to get the final matrix in reduced row echelon form, we perform the same operations seen in the first step, but now, considering only the last two rows and performing the operations only on the elements of the last row. Also in this case, we perform shift operations to avoid overflow.

$$A_8 = A_8 - \frac{A_9 A_4}{A_5}$$

$$A_9 = A_9 - \frac{A_9 A_5}{A_5}$$
$$A_{10} = A_{10} - \frac{A_9 A_6}{A_5}$$
$$b_1 = b_1 - \frac{A_9 b_1}{A_5}$$

```
DIVIDEND8_K1<=shift_right(IN_MATRIX(2,1),8)*IN_MATRIX(1,0);
DIV8_K1 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(1,1),
              DIVIDEND=>DIVIDEND8_K1(63 DOWNTO 0),
              QUOTIENT=>QUOTIENT8_K1);
A8_K1<=IN_MATRIX(2,0)- shift_left(QUOTIENT8_K1,8);

DIVIDEND9_K1<=shift_right(IN_MATRIX(2,1),8)*IN_MATRIX(1,1);
DIV9_K1 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(1,1),
              DIVIDEND=>DIVIDEND9_K1(63 DOWNTO 0),
              QUOTIENT=>QUOTIENT9_K1);
A9_K1<=IN_MATRIX(2,1)- shift_left(QUOTIENT9_K1,8);

DIVIDEND10_K1<=shift_right(IN_MATRIX(2,1),8)*IN_MATRIX(1,2);
DIV10_K1 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(1,1),
              DIVIDEND=>DIVIDEND10_K1(63 DOWNTO 0),
              QUOTIENT=>QUOTIENT10_K1);
A10_K1<=IN_MATRIX(2,2)- shift_left(QUOTIENT10_K1,8);

B_DIVIDEND2_K1 <= IN_MATRIX(2,1)*IN_VECTOR(1);
B_DIV2_K1 : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(1,1),
              DIVIDEND=>B_DIVIDEND2_K1(63 DOWNTO 0),
              QUOTIENT=>QUOTIENTB2_K1);
B2_K1<=IN_VECTOR(2) - QUOTIENTB2_K1;
```

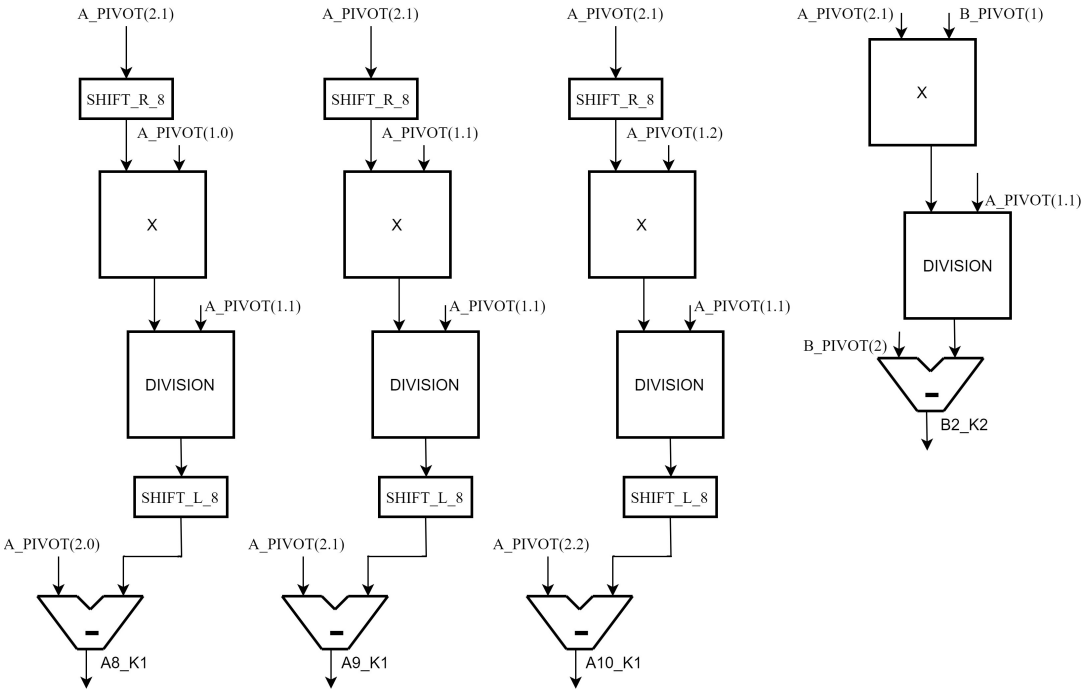

Figure 4.11: Execution Unit of second part of Forward Elimination block.

4.8.5 Back Substitution

This is the last step to get the linear system solution. Starting from the matrix in reduced row echelon form:

$$\begin{pmatrix} A_0 & A_1 & A_2 \\ 0 & A_5 & A_6 \\ 0 & 0 & A_{10} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} \tag{4.4}$$

we proceed with the backwards substitution(4.12). In detail, the following operations are performed:

$$\begin{aligned} x_2 &= \frac{B_2}{A_{10}} \\ x_1 &= \frac{B_1 - x_2 A_6}{A_5} \\ x_0 &= \frac{B_0 - (x_1 A_1 + x_2 A_2)}{A_0} \end{aligned}$$

To obtain the correct value, we divide the value to be subtracted by the scaling factor and finally multiply the result again by the scaling factor 2^{16}. In addition, to get the same rounding of the codec, every time you have to perform a right shift that involves a loss of bits, it is done on the absolute value, to have the same result of the codec.

```
1  DIVIDEND_1<=shift_left(IN_VECTOR(2),16);
2  B2_A10_DIV : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(2,2),DIVIDEND=>
       DIVIDEND_1,QUOTIENT=>X2);
3  PRODUCT_1<=X2*IN_MATRIX(1,2);
4  PROCESS(PRODUCT_1,ABS_PROD_1, C_1_SHIFT)
5  BEGIN
6  IF (PRODUCT_1(127)='1') THEN
7  ABS_PROD_1<=NOT(PRODUCT_1)+1;
8  ELSE
9  ABS_PROD_1<=PRODUCT_1;
10 END IF;
11 C_1_SHIFT<=shift_right(ABS_PROD_1,16);
12 IF (PRODUCT_1(127)='1') THEN
13 C_1<=NOT(C_1_SHIFT)+1;
14 ELSE
15 C_1<=C_1_SHIFT;
16 END IF;
17 END PROCESS;
18 B1_MENO_C<=IN_VECTOR(1)-C_1(63 DOWNTO 0);
19 DIVIDEND_2<=shift_left(B1_MENO_C,16);
20 B1_A5_DIV : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(1,1),DIVIDEND=>
       DIVIDEND_2,QUOTIENT=>X1);
21 PRODUCT_2<=X1*IN_MATRIX(0,1);
22 PROCESS(PRODUCT_2,ABS_PROD_2, C_2_SHIFT)
23 BEGIN
24 IF (PRODUCT_2(127)='1') THEN
25 ABS_PROD_2<=NOT(PRODUCT_2)+1;
26 ELSE
27 ABS_PROD_2<=PRODUCT_2;
28 END IF;
29 C_2_SHIFT<=shift_right(ABS_PROD_2,16);
30 IF (PRODUCT_2(127)='1') THEN
31 C_2<=NOT(C_2_SHIFT)+1;
32 ELSE
33 C_2<=C_2_SHIFT;
34 END IF;
35 END PROCESS;
36 PRODUCT_3<=X2*IN_MATRIX(0,2);
37 PROCESS(PRODUCT_3,ABS_PROD_3, C_3_SHIFT)
38 BEGIN
39 IF (PRODUCT_3(127)='1') THEN
40 ABS_PROD_3<=NOT(PRODUCT_3)+1;
41 ELSE
42 ABS_PROD_3<=PRODUCT_3;
43 END IF;
44 C_3_SHIFT<=shift_right(ABS_PROD_3,16);
45 IF (PRODUCT_3(127)='1') THEN
46 C_3_TEMP<=NOT(C_3_SHIFT)+1;
47 ELSE
```

```
48 C_3_TEMP<=C_3_SHIFT;
49 END IF;
50 END PROCESS;
51 C_3<=C_2+C_3_TEMP;
52 DIVIDEND_3<=shift_left((IN_VECTOR(0)-C_3(63 DOWNTO 0)),16);
53 B0_A0_DIV : DIVISION PORT MAP (DIVISOR=>IN_MATRIX(0,0),DIVIDEND=>
      DIVIDEND_3,QUOTIENT=>X0);
54 OUT_VECTOR<=(X0(31 DOWNTO 0), X1(31 DOWNTO 0), X2(31 DOWNTO 0));
```

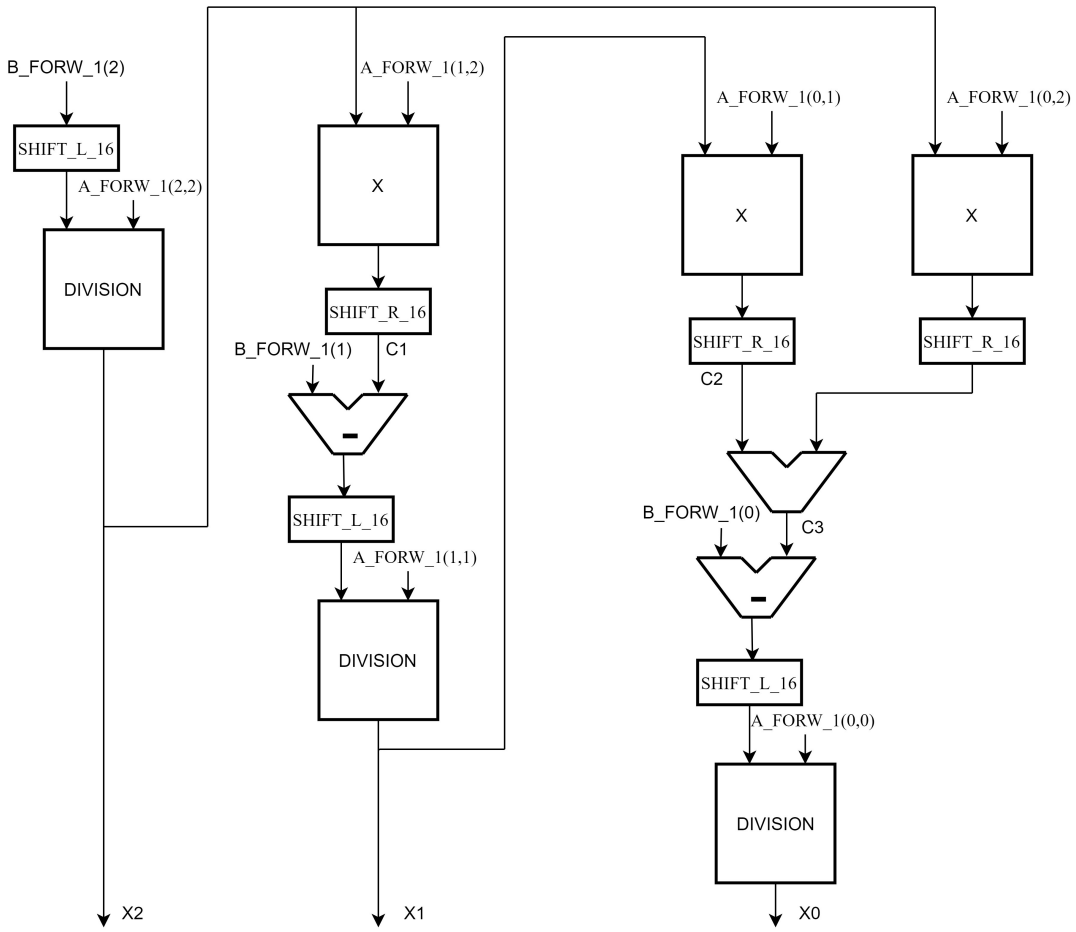

Figure 4.12: Execution Unit of Back Substitution block.

4.9 Update b

This block(4.13) is the second main block that compose the architecture and is very similar to the previous one. It is also divided into several sub-blocks that interact among each other. The inputs are:

- The vector **a** given by **Update a** block, composed of 7 elements of 32 bits each.
- The $\boldsymbol{H_{ij}}$ square matrix, a proper sub-matrix of the **H** matrix, of 49 elements, each of 64 bits.
- The whole square matrix **M**, of 49 elements, each of 64 bits.

The output is the new vector **b** of 7 elements, each of 32 bits. Also in this case, to select the correct values of the vector **b**, of the matrix **M** and for the correct working of the FSM cycles, two counters of 3 bits each are instantiated. The block algorithm can be divided again into two parts that work in parallel. In the first part, the partial vectors **B** of 64 bits are calculated as the product between the elements of the matrix $\boldsymbol{H_{ij}}$ and two elements of the vector **a**, which we can summarize in $\boldsymbol{H_{ij}a_ia_j}$. At the end of the computation, each partial vector **PARTIAL_B** is stored into a partial matrix. In order to obtain the matrix **B** each value of the partial matrix is added with other values of the matrix itself properly selected and the result is stored in the correct position of the final **B** matrix.

```
1  B_MATRIX_PARTIAL(INDEX_I_INT,INDEX_J_INT)<=PARTIAL_B;
2  B_FINAL(0,0)<=B_MATRIX_PARTIAL(0,0)+B_MATRIX_PARTIAL(0,6)+
       B_MATRIX_PARTIAL(6,0)+B_MATRIX_PARTIAL(6,6);
3  B_FINAL(0,1)<=B_MATRIX_PARTIAL(0,1)+B_MATRIX_PARTIAL(0,5)+
       B_MATRIX_PARTIAL(6,1)+B_MATRIX_PARTIAL(6,5);
4  B_FINAL(0,2)<=B_MATRIX_PARTIAL(0,2)+B_MATRIX_PARTIAL(0,4)+
       B_MATRIX_PARTIAL(6,2)+B_MATRIX_PARTIAL(6,4);
5  B_FINAL(0,3)<=B_MATRIX_PARTIAL(0,3)+B_MATRIX_PARTIAL(6,3);
6  B_FINAL(1,0)<=B_MATRIX_PARTIAL(1,0)+B_MATRIX_PARTIAL(1,6)+
       B_MATRIX_PARTIAL(5,0)+B_MATRIX_PARTIAL(5,6);
7  B_FINAL(1,1)<=B_MATRIX_PARTIAL(1,1)+B_MATRIX_PARTIAL(1,5)+
       B_MATRIX_PARTIAL(5,1)+B_MATRIX_PARTIAL(5,5);
8  B_FINAL(1,2)<=B_MATRIX_PARTIAL(1,2)+B_MATRIX_PARTIAL(1,4)+
       B_MATRIX_PARTIAL(5,2)+B_MATRIX_PARTIAL(5,4);
9  B_FINAL(1,3)<=B_MATRIX_PARTIAL(1,3)+B_MATRIX_PARTIAL(5,3);
10 B_FINAL(2,0)<=B_MATRIX_PARTIAL(2,0)+B_MATRIX_PARTIAL(2,6)+
       B_MATRIX_PARTIAL(4,0)+B_MATRIX_PARTIAL(4,6);
11 B_FINAL(2,1)<=B_MATRIX_PARTIAL(2,1)+B_MATRIX_PARTIAL(2,5)+
       B_MATRIX_PARTIAL(4,1)+B_MATRIX_PARTIAL(4,5);
12 B_FINAL(2,2)<=B_MATRIX_PARTIAL(2,2)+B_MATRIX_PARTIAL(2,4)+
       B_MATRIX_PARTIAL(4,2)+B_MATRIX_PARTIAL(4,4);
13 B_FINAL(2,3)<=B_MATRIX_PARTIAL(2,3)+B_MATRIX_PARTIAL(4,3);
14 B_FINAL(3,0)<=B_MATRIX_PARTIAL(3,0)+B_MATRIX_PARTIAL(3,6);
15 B_FINAL(3,1)<=B_MATRIX_PARTIAL(3,1)+B_MATRIX_PARTIAL(3,5);
16 B_FINAL(3,2)<=B_MATRIX_PARTIAL(3,2)+B_MATRIX_PARTIAL(3,4);
17 B_FINAL(3,3)<=B_MATRIX_PARTIAL(3,3);
```

In the second part, the partial vectors **A** of 64 bits each are calculated as the product between the elements of the vector $\boldsymbol{M_i}$, selected by the **M SELECTION** block,

and an element of the vector **a**, which we can summarize in $\boldsymbol{M_i a_i}$. At the end of each calculation, the partial vector **A** is stored into a partial array. In order to obtain the array **A** each value of the partial array is added with other values of the array itself properly selected and the result is stored in the correct position of the final **A** array.

```
1 A_ARRAY_PARTIAL(INDEX_J_INT)<=PARTIAL_A;
2 A_FINAL(0)<=A_ARRAY_PARTIAL(0)+A_ARRAY_PARTIAL(6);
3 A_FINAL(1)<=A_ARRAY_PARTIAL(1)+A_ARRAY_PARTIAL(5);
4 A_FINAL(2)<=A_ARRAY_PARTIAL(2)+A_ARRAY_PARTIAL(4);
5 A_FINAL(3)<=A_ARRAY_PARTIAL(3);
```

Once the calculation of the matrix **B** and the vector **A** is finished, they are processed in the **Enforcement** block, that is exactly the same sub-block of **Update a** block, to obtain a square matrix **B** of 9 elements each of 64 bits and a vector **A** of 3 elements each of 64 bits. At this point we make a change in the nomenclature, as happens in the code, to have the same of a linear system $\boldsymbol{Ax = B}$. So now we have a 3x3 matrix **A** and a 1x3 vector **B**. The next steps are exactly the same as the **Update a** block and so also all the following blocks are the same. The linear system is solved to obtain the vector of solutions **x**. Also in this case, the blocks of **PIVOTING** and **FORWARD ELIMINATION** are used twice, thanks to the use of two multiplexers placed at the input of the two blocks. At the end of the two cycles, through the **BACK-SUBSTITUTION STORING** block, the vector of the solutions **x** of 3 elements, each of 32 bits, is computed. Finally, to obtain the final vector **b** of 7 elements, each of 32 bits, the symmetry constraints are applied to the vector **x** to obtain the last three values and the equality condition and scale factor to obtain the central element. Also the FSM (4.14) is very similar and it is managed by the two counter signals. After the block enable signal, we find a loop, the execution unit performs the operations of the **SUM J** state until counter **J** reaches 5. Once the first cycle is finished, we find two nested loops. The FSM goes into the **SUM I** state and a further **SUM J** state, at this point the FSM remains in the **SUM J** state until the Counter-J reaches the value of 5, at that point, if the Counter-I has not yet reached the value of 6, the second cycle of **SUM J** starts again. Once the two nested loops are completed, the remaining states are executed in succession, without further conditions. In detail, the **ENABLE ENFORCEMENT** state enables enforcement block operations, the **K0 SEL0** and **K1 SEL1** states direct multiplexers for correct operation and input, and after the execution of pivoting and forward elimination blocks, the **DONE** state signals that the output data is correct.

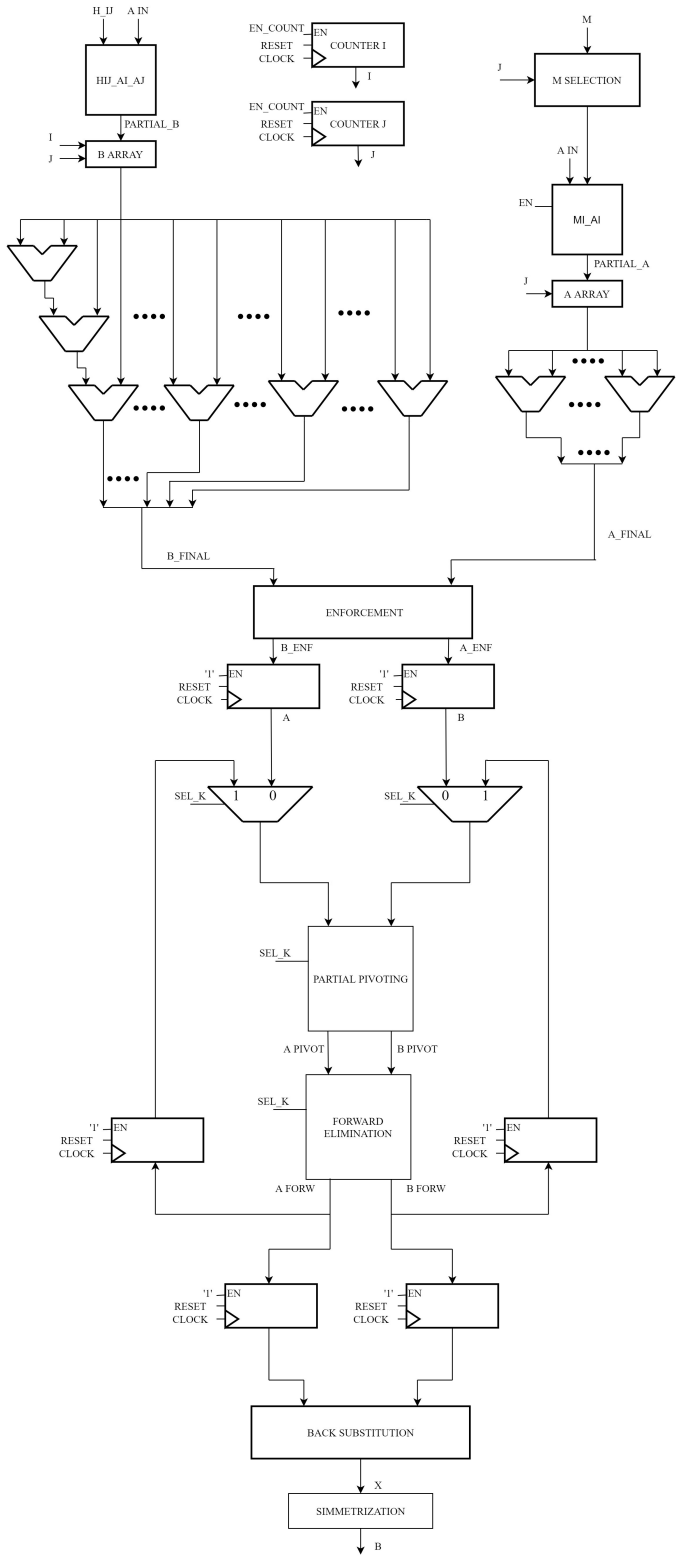

Figure 4.13: Update b Execution Unit.

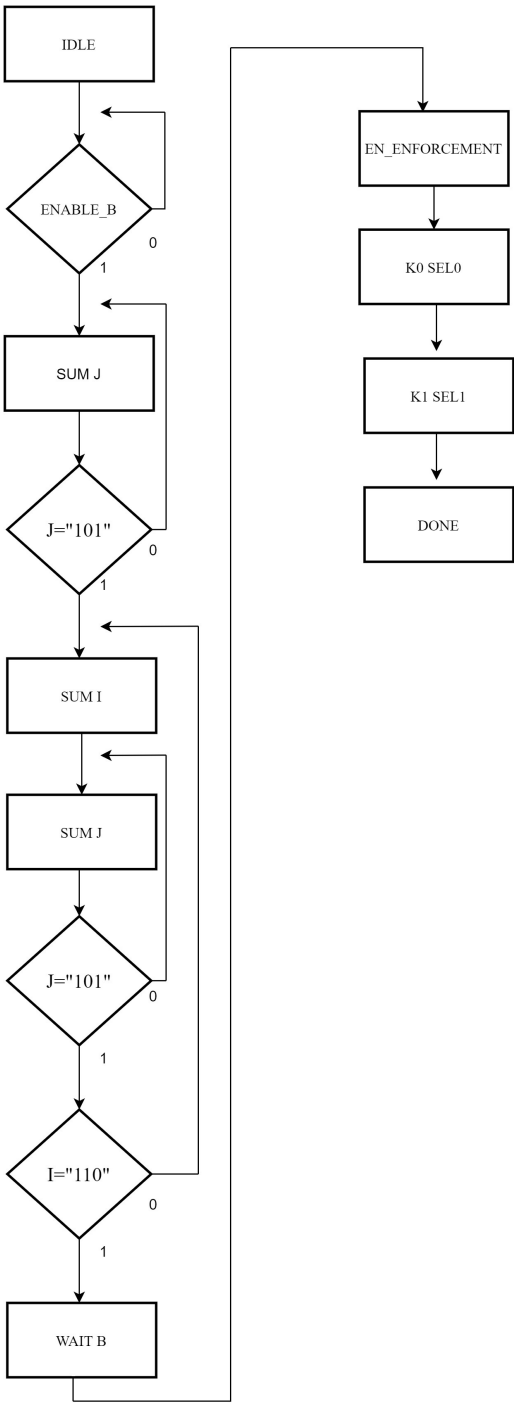

Figure 4.14: Update b FSM.

4.10 $H_{ij}a_ia_j$

As in the $\boldsymbol{H_{ij}b_ib_j}$ block, the product between the elements of the $\boldsymbol{H_{ij}}$ matrix and the two elements of vector **a** is computed.(4.15) The inputs are the $\boldsymbol{H_{ij}}$ matrix, and the **a** vector. Before proceeding with the operations, since later right shifts will be made and that involve truncation with the loss of LSB, the absolute value of the **a** vector is computed. In this way, at the end of the operations, we will obtain the same approximate values of the codec, then to the final result will be applied the correct sign evaluating the signs of the two values of **a**, applying the two's complement in case of negative sign. At the end of computation, each partial products between the element $\boldsymbol{H_{ij}(j,i)}$ (matrix elements are selected by scanning the matrix column by column) and the two absolute values of the elements of the input vector **a** is added to the sum of the previous ones. After each multiplication of the element of **b**, a right shift of 16 positions (division by $\mathbf{2^{16}}$) is made to respect the coherence with the scale factor and to avoid overflow.

```
1  FOR I IN 0 TO 6 LOOP
2    FOR J IN 0 TO 6 LOOP
3    B_ABS:=(shift_right(((shift_right((H_IN(J,I)*A_ABS(I)),16)(63
       DOWNTO 0))*A_ABS(J)),16));
4      IF(A_IN(I)(31)=A_IN(J)(31)) THEN
5        B_TEMP:=B_ABS;
6      ELSE
7        B_TEMP:=NOT(B_ABS)+1;
8      END IF;
9    SUM_B:=SUM_B+B_TEMP(63 DOWNTO 0);
10 END LOOP;
11 END LOOP;
12 B_OUT<=SUM_B;
```

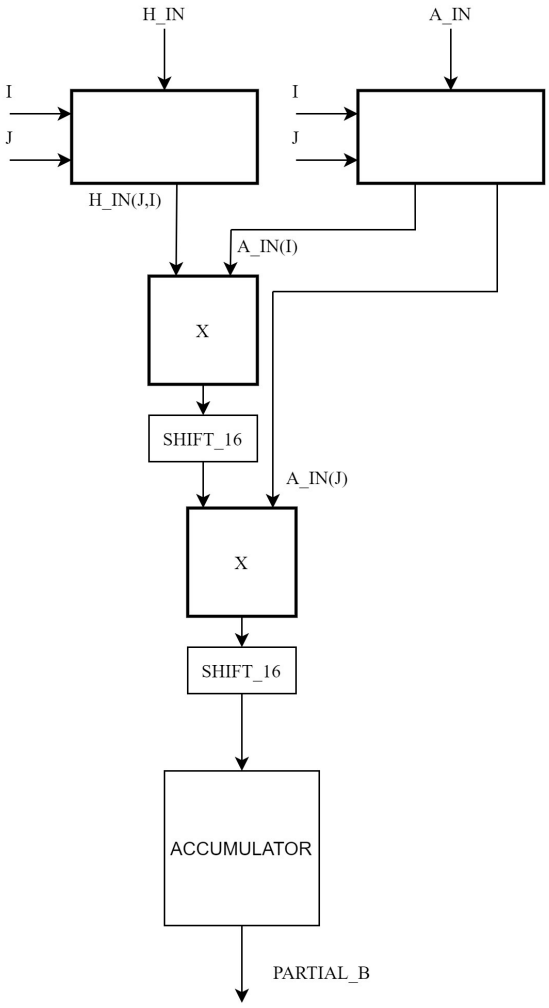

Figure 4.15: $H_{ij}a_ia_j$ Execution Unit.

4.11 M_ia_i

This is a very simple block(4.16) that follows the same procedure of the previous block. The inputs are the $\boldsymbol{M_i}$ vector and the $\mathbf{a}$ vector. Also in this case, the absolute value of $\mathbf{a}$ vector is computed for the same approximation as in the previous block. Each product between a value of the vector $\boldsymbol{M_i}$ and $\boldsymbol{a_i}$, properly shifted, are added to the sum of the previous partial products,after applying the correct signs to the current partial products.

```
1 FOR I IN 0 TO 6 LOOP
2   A_ABSV:=shift_right(M_IN(I)*A_ABS(I),16);
3     IF(A_IN(I)(31)='0') THEN
```

```
4         A_TEMP:=A_ABSV;
5       ELSE
6         A_TEMP:=NOT(A_ABSV)+1;
7       END IF;
8          SUM_A:=SUM_A+A_TEMP(63 DOWNTO 0);
9   END LOOP;
10  A_OUT<=SUM_A;
```

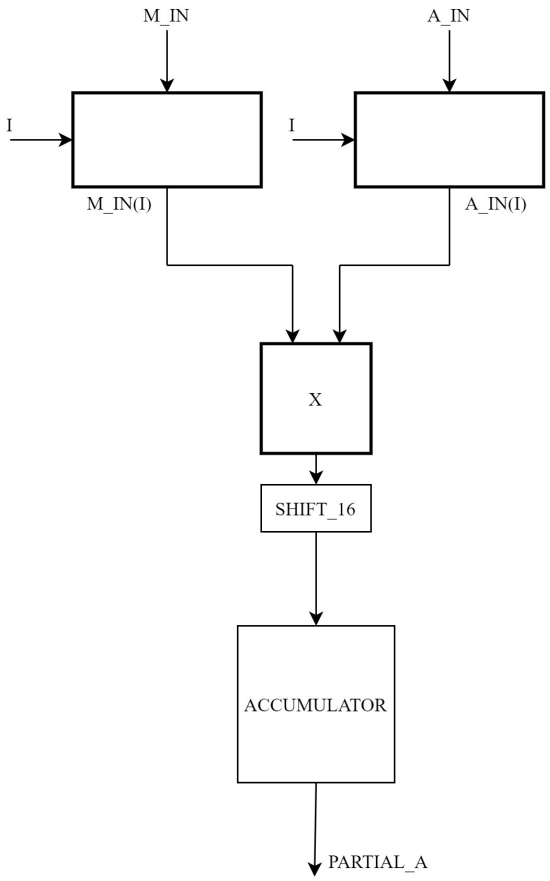

Figure 4.16: $M_i a_i$ Execution Unit.

4.12 Simulation and Performance

Once the construction of the whole architecture was completed, simulations were performed to verify its correct functionality. By running the codec on the test file **Flowervase_416x240_30.yuv** it was possible to extract different inputs thanks to which we could test the architecture under different conditions. Also the intermediate values and output values have been extracted in order to check step by step the

correct working of each block and, at the end, the final values of vectors **a** and **b**. Different simulations have been performed, we report one of them as an example. The extracted inputs and outputs are as follows:

- The starting vector **b** composed of 7 elements of 32 bits each.

$$b = (1536, -3584, 7680, 54272, 7680, -3584, 1536)$$

- The whole square matrix **H** of 2401 elements (49 x 49), each of 64 bits.

- The whole square matrix **M**, of 49 elements, each of 64 bits.

$$M = \begin{pmatrix} 4999060 & 5025613 & 5050784 & 5073739 & 5094182 & 5111263 & 5126710 \\ 5084193 & 5110307 & 5134816 & 5156655 & 5175379 & 5190825 & 5204412 \\ 5183739 & 5209022 & 5234019 & 5255837 & 5272390 & 5284822 & 5296317 \\ 5277779 & 5302305 & 5330294 & 5354683 & 5367976 & 5374936 & 5384627 \\ 5333327 & 5356248 & 5384184 & 5408160 & 5420160 & 5424455 & 5433045 \\ 5371176 & 5392191 & 5417990 & 5440158 & 5451263 & 5454868 & 5462529 \\ 5425034 & 5444236 & 5468202 & 5488891 & 5499249 & 5501899 & 5508557 \end{pmatrix} \tag{4.5}$$

- The final vector **a** composed of 7 elements of 32 bits each.

$$a = (982, -4862, 9525, 54246, 9525, -4862, 982)$$

- The final vector **b** composed of 7 elements of 32 bits each.

$$b = (566, -1995, 3719, 60956, 3719, -1995, 566)$$

In the following picture(4.17) we can observe the main relevant steps of our algorithm. Highlighted in red we see the evolution of the block **Update a** ending when vector **a** is updated, in blue we see the evolution of the block **Update b** ending when vector **b** is updated. Once the the block **Update b** is finished, the tri-state buffers, highlighted in yellow, are enabled and provide the new updated vectors as output. You can also see the main control signals that handle the evolution of the states.

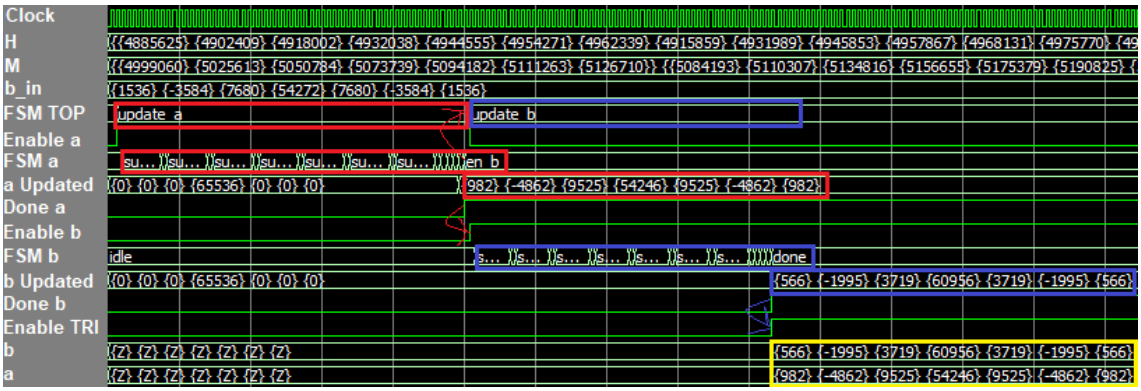

Figure 4.17: Behavioural simulation of the circuit.

Once the correct working of the architecture was verified, it was synthesized using the Synopsys tool and it was possible to obtain reports that gave details about the characteristic of the architecture and its performance. In detail, this thesis work explores in depth the analysis of architectural performance. Studying the report timing you can see that, giving a clock of 10 ns, you get a negative SLACK of -221.70 ns. From this we can derive that the architecture can work at a maximum frequency of 4.3 MHz ($T_{clk} = 232ns$). Moreover, from the report timing extract, it is highlighted which is the critical path. From these indications we can understand where to operate to improve performance in terms of speed.

```
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : BASIC_WIENER_FILTER
Version: M-2016.12
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

  Startpoint: UPDATE_B_TOPLEVEL_PORTING/DP/REG_A_OUT/PORTING12/REG_OUT_reg[0]
              (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Endpoint: B[3][27] (output port clocked by MY_CLOCK)
  Path Group: MY_CLOCK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
```

```
WIENER_FILTER      5K_hvratio_1_1        NangateOpenCellLibrary

Point                                                          Incr       Path
-----------------------------------------------------------------------------
clock MY_CLOCK (rise edge)                                     0.00       0.00
clock network delay (ideal)                                    0.00       0.00
UPDATE_B_TOPLEVEL_PORTING/DP/REG_A_OUT/PORTING12/REG_OUT_reg[0]/CK (DFFR_X1)
                                                               0.00 #     0.00 r
UPDATE_B_TOPLEVEL_PORTING/DP/REG_A_OUT/PORTING12/REG_OUT_reg[0]/QN (DFFR_X1)
                                                               0.07       0.07 r
UPDATE_B_TOPLEVEL_PORTING/DP/REG_A_OUT/PORTING12/U2/ZN (INV_X1)
                                                               0.03       0.10 f
UPDATE_B_TOPLEVEL_PORTING/DP/REG_A_OUT/PORTING12/DATA_OUT[0] (REG_64BIT_4)
                                                               0.00       0.10 f
UPDATE_B_TOPLEVEL_PORTING/DP/REG_A_OUT/OUT_MATRIX[2][2][0] (REG_3X3_64BIT_1)
                                                               0.00       0.10 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/IN_MATRIX[2][2][0]
(BACK_SUB_STORING_1)
                                                               0.00       0.10 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/DIVISOR[0] (DIVISION_3)
                                                               0.00       0.10 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/B[0]
(DIVISION_3_DW01_sub_65)
                                                               0.00       0.10 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U187/ZN
(NOR2_X1)
                                                               0.06       0.15 r
                    .  .  .

UPDATE_B_TOPLEVEL_PORTING/DP/B_OUT[3][27] (UPDATE_B)
                                                               0.00     231.35 r
UPDATE_B_TOPLEVEL_PORTING/B[3][27] (UPDATE_B_TOPLEVEL)
                                                               0.00     231.35 r
B_TRI/IN_VECTOR[3][27] (BUFFER_TRI_STATE_1)                    0.00     231.35 r
B_TRI/OUT_VECTOR_tri[3][27]/Z (TBUF_X1)                        0.05     231.40 r
B_TRI/OUT_VECTOR[3][27] (BUFFER_TRI_STATE_1)                   0.00     231.40 r
B[3][27] (out)                                                 0.02     231.42 r
data arrival time                                                       231.42

clock MY_CLOCK (rise edge)                                   232.00     232.00
clock network delay (ideal)                                    0.00     232.00
clock uncertainty                                             -0.07     231.93
output external delay                                         -0.50     231.43
data required time                                                      231.43
-----------------------------------------------------------------------------
data required time                                                      231.43
data arrival time                                                      -231.42
-----------------------------------------------------------------------------
slack (MET)                                                              0.01
```

Chapter 5

High Speed Wiener Filter

In this chapter we analyze the improvements that have been made to the basic architecture to increase speed. In detail, from the timing reports you can see several critical issues. The main one is related to the division operation, the others depend on the combinational paths of the various sub-blocks. To improve these critical issues, a new block that performs the division operation has been designed and pipeline registers have been added within the sub-blocks to reduce the length of the combinational paths. Starting from the basic architecture, only the blocks to which changes have been made will be analyzed.

5.1 Division

In the main architecture timing report, it is highlighted that the critical paths are those that include division operations. This led us to analyze this operation, studying its report timing.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : BASIC_DIVISION
Version: M-2016.12
****************************************

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

  Startpoint: REG1/REG_OUT_reg[2]
              (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Endpoint: REG3/REG_OUT_reg[11]
            (rising edge-triggered flip-flop clocked by MY_CLOCK)
```

```
Path Group: MY_CLOCK
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
DIVISION           5K_hvratio_1_1        NangateOpenCellLibrary

Point                                                    Incr       Path
-----------------------------------------------------------------------------
clock MY_CLOCK (rise edge)                               0.00       0.00
clock network delay (ideal)                              0.00       0.00
REG1/REG_OUT_reg[2]/CK (DFFR_X1)                         0.00       0.00 r
REG1/REG_OUT_reg[2]/QN (DFFR_X1)                         0.07       0.07 r
REG1/U14/ZN (INV_X1)                                     0.02       0.09 f
REG1/DATA_OUT[2] (REG_64BIT_0)                           0.00       0.09 f
sub_add_65_b0/B[2] (DIVISION_DW01_sub_65)                0.00       0.09 f
sub_add_65_b0/U107/ZN (OR2_X1)                           0.06       0.15 f
sub_add_65_b0/U110/ZN (NOR2_X1)                          0.04       0.19 r
sub_add_65_b0/U114/ZN (AND2_X1)                          0.05       0.24 r
sub_add_65_b0/U116/ZN (AND2_X1)                          0.04       0.28 r
         .  .  .
sub_add_89_b0/U328/ZN (XNOR2_X1)                         0.03      66.35 f
sub_add_89_b0/DIFF[11] (DIVISION_DW01_sub_67)            0.00      66.35 f
U1495/ZN (AOI22_X1)                                      0.04      66.39 r
U1496/ZN (INV_X1)                                        0.02      66.41 f
REG3/DATA_IN[11] (REG_64BIT_1)                           0.00      66.41 f
REG3/U23/Z (MUX2_X1)                                     0.06      66.48 f
REG3/REG_OUT_reg[11]/D (DFFR_X1)                         0.01      66.49 f
data arrival time                                                  66.49

clock MY_CLOCK (rise edge)                              66.60      66.60
clock network delay (ideal)                              0.00      66.60
clock uncertainty                                       -0.07      66.53
REG3/REG_OUT_reg[11]/CK (DFFR_X1)                        0.00      66.53 r
library setup time                                      -0.04      66.49
data required time                                                 66.49
-----------------------------------------------------------------------------
data required time                                                 66.49
data arrival time                                                 -66.49
-----------------------------------------------------------------------------
slack (MET)                                                        0.00
```

The divider used in the basic architecture is realized only using combinational paths, since the divisions are realized on a high number of bits, this means having a negative SLACK of -56.58 ns having used a clock of 10 ns, so the divider can operate at a maximum frequency of 15.15 MHz ($T_{clk} = 66.6ns$). So we decided to use a **Restoring Divider** (5.1), a totally different architecture that minimizes the combinational paths, using several registers. In particular, two shift registers, two

registers, a counter and a subtractor are used to run the algorithm. In the lower part of a shift register of double size compared to the register of operators, i.e. 128 bit, the value of the dividend is loaded, after which at each clock, it is subtracted from the value of the upper half of the shift register, the divisor, if the result is positive, a '1' is saved in the quotient shift register and the partial reminder is loaded in the upper half of the 128 bit shift register, otherwise, if the result is negative, a '0' is saved in the quotient shift register and the partial reminder is not loaded. At each clock, the shift registers shift data to the left of one position.

```
  BEGIN
   IF(DIVISOR(63)='1') THEN
ABS_DIVISOR<=(NOT(DIVISOR))+1;
ELSE
ABS_DIVISOR<=DIVISOR;
END IF;
IF(DIVIDEND(63)='1') THEN
ABS_DIVIDEND<=(NOT(DIVIDEND))+1;
ELSE
ABS_DIVIDEND<=DIVIDEND;
END IF;
END PROCESS;
INVERSION<=DIVISOR(63) XOR DIVIDEND(63);
COUNTER_64BIT : COUNTER_64 GENERIC MAP (N=>6)
               PORT MAP (EN=>EN_CNT,
                     CLOCK=>CLOCK,
                     RST=>RESET,
                     CNT=>COUNT);
COUNT_OUT<=COUNT;
EXT_DIVIDEND(127 DOWNTO 64) <=(OTHERS => '0');
EXT_DIVIDEND(63 DOWNTO 0) <= ABS_DIVIDEND;

PART_REM: SHIFT_REG_128BIT PORT MAP (CLOCK=>CLOCK,
                  RESET=>RESET,
                  EN_SHIFT=>EN_SHIFT_REM,
                  LOAD_REM=>LOAD_REM,
                  LOAD_Z=>LOAD_Z,
                  DATA_IN_Z=>EXT_DIVIDEND,
                  DATA_IN_REM=>PAR_SUM,
                  DATA_OUT=> DATA_OUT_REM_REG);
QUOT: SHIFTER PORT MAP (    CLOCK=>CLOCK,
               RESET=>RESET,
               SHIFT=> EN_SHIFT_QUO,
               DATA_IN=>SIGN_BIT_REG,
               DATA_OUT=>PARTIAL_QUOTIENT);
DIV : REG_64BIT PORT MAP (CLOCK=>CLOCK,
               RESET=>RESET,
               EN=>LOAD_DIV,
               DATA_IN=>ABS_DIVISOR,
```

```
40                DATA_OUT=>DIVISOR_REG);
41 PAR_SUM <= DATA_OUT_REM_REG - DIVISOR_REG;
42 SIGN_BIT_REG<= NOT(PAR_SUM(63));
43 SIGN_BIT<= SIGN_BIT_REG;
44 PROCESS(INVERSION,PARTIAL_QUOTIENT)
45   BEGIN
46 IF(INVERSION='0') THEN
47 QUOTIENT<=PARTIAL_QUOTIENT;
48 ELSE
49 QUOTIENT<=(NOT(PARTIAL_QUOTIENT)+1);
50 END IF;
```

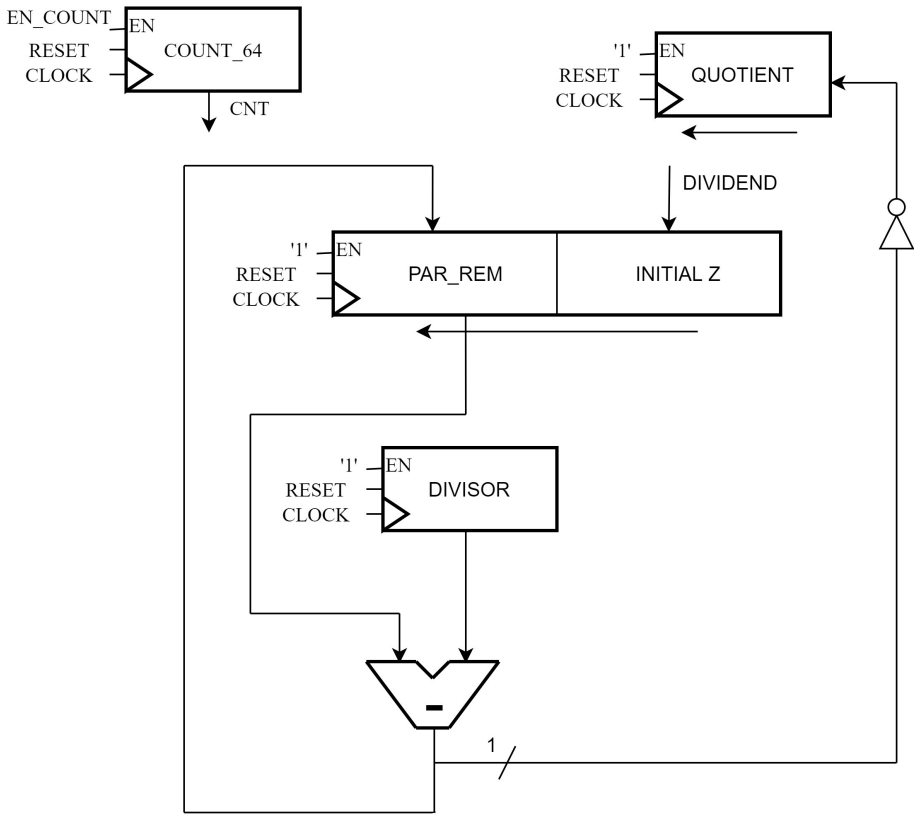

Figure 5.1: Restoring Divider Execution Unit.

Looking at the timing report of this new architecture, we can see a significant improvement, starting from a clock of 10 ns, we obtain a positive SLACK of 3.49 ns, this means that the maximum operating frequency obtained for the Divider is 153.6 MHz ($T_{clk} = 6.51ns$).

```
****************************************
Report : timing
```

```
        -path full
        -delay max
        -max_paths 1
Design : RESTORING_DIVIDER
Version: M-2016.12
*************************************
Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top
  Startpoint: DP/PART_REM/TEMP_reg[64]
              (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Endpoint: DP/PART_REM/TEMP_reg[64]
            (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Path Group: MY_CLOCK
  Path Type: max
  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  DIVISION           5K_hvratio_1_1        NangateOpenCellLibrary

  Point                                                   Incr       Path
  ---------------------------------------------------------------------------
  clock MY_CLOCK (rise edge)                              0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  DP/PART_REM/TEMP_reg[64]/CK (DFFR_X1)                   0.00       0.00 r
  DP/PART_REM/TEMP_reg[64]/Q (DFFR_X1)                    0.09       0.09 f
  DP/PART_REM/DATA_OUT[0] (SHIFT_REG_128BIT)              0.00       0.09 f
  DP/sub_80/A[0] (DIVISION_DP_DW01_sub_0)                 0.00       0.09 f
  DP/sub_80/U5/ZN (INV_X1)                                0.03       0.12 r
  DP/sub_80/U4/ZN (NAND2_X1)                              0.03       0.15 f
  DP/sub_80/U2_1/CO (FA_X1)                               0.09       0.24 f
  DP/sub_80/U2_2/CO (FA_X1)                               0.09       0.33 f
              . . .
  DP/PART_REM/U240/ZN (OAI221_X1)                         0.06       6.39 f
  DP/PART_REM/TEMP_reg[64]/D (DFFR_X1)                    0.01       6.40 f
  data arrival time                                                  6.40

  clock MY_CLOCK (rise edge)                             10.00      10.00
  clock network delay (ideal)                             0.00      10.00
  clock uncertainty                                      -0.07       9.93
  DP/PART_REM/TEMP_reg[64]/CK (DFFR_X1)                   0.00       9.93 r
  library setup time                                     -0.05       9.88
  data required time                                                 9.88
  ---------------------------------------------------------------------------
  data required time                                                 9.88
  data arrival time                                                 -6.40
  ---------------------------------------------------------------------------
  slack (MET)                                                        3.49
```

5.2 Forward Elimination

The **Forward Elimination** block has been divided into two independent blocks (5.2)(5.3) and pipeline registers have been added to each of these two blocks to make the combinational paths shorter. In this way, starting from a critical path that included both a multiplier and a divisor, the new critical path includes only one multiplier.

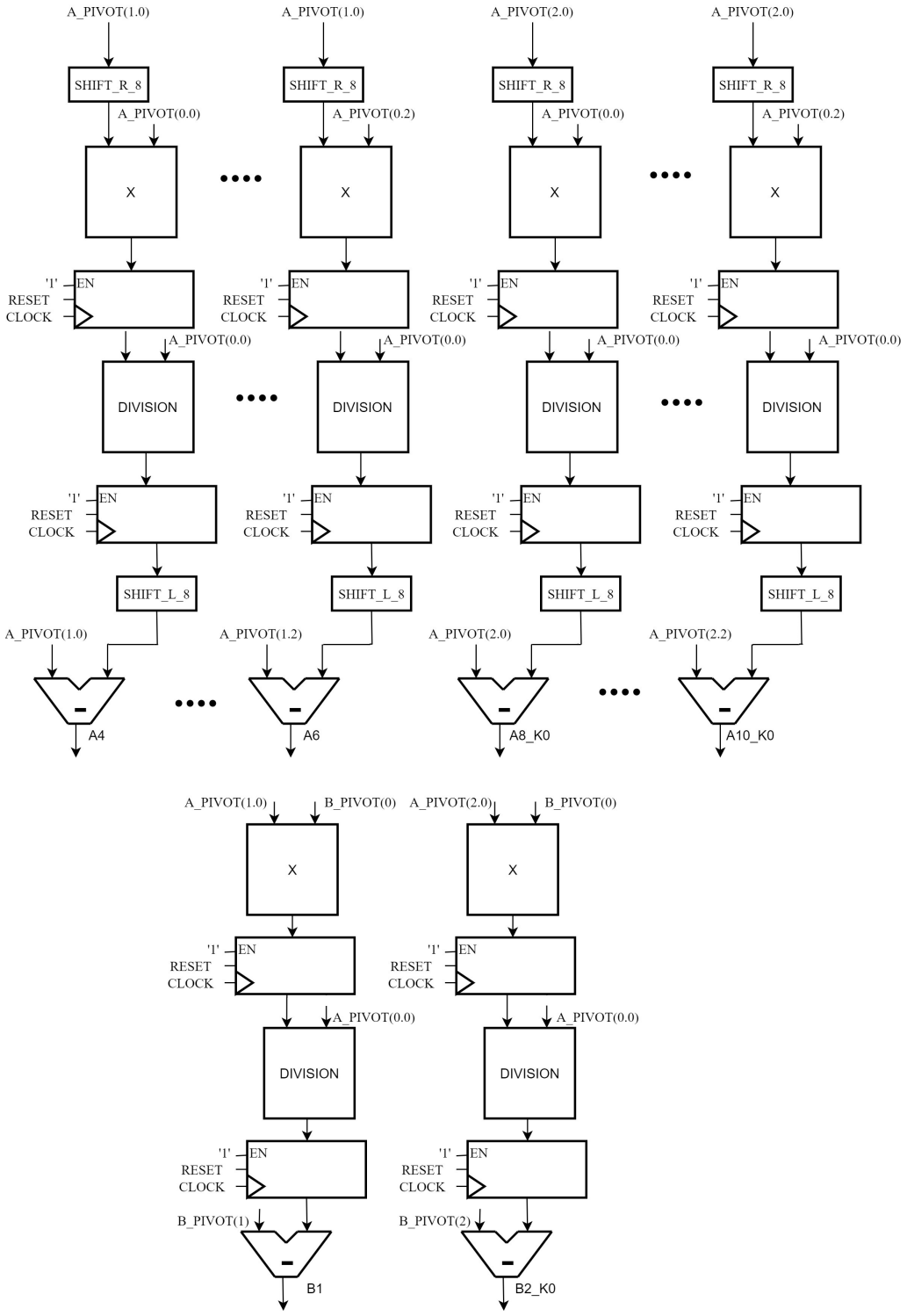

Figure 5.2: Forward Elimination 1 High Speed Execution Unit.

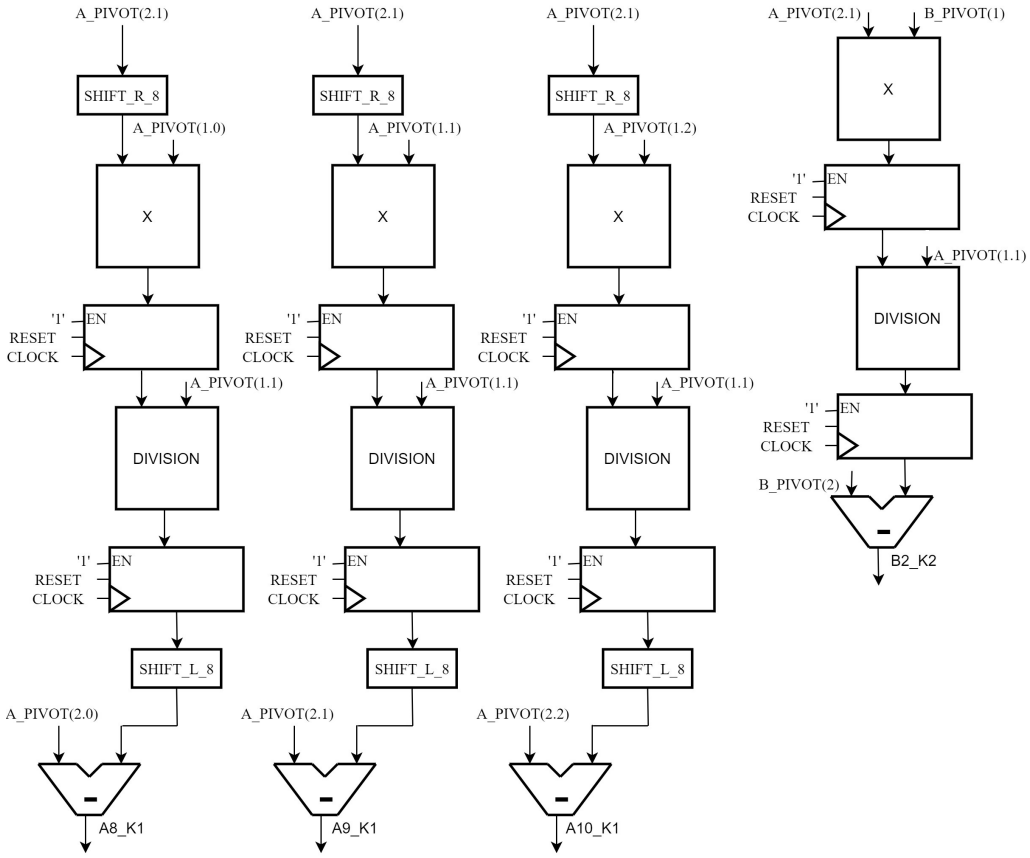

Figure 5.3: Forward Elimination 2 High Speed Execution Unit.

5.3 Back Substitution

Pipeline registers are also inserted in the **Back Substitution** block (5.4) to reduce the combinational paths. Starting from a critical path that included 3 divisors, 2 multipliers, 2 subtractors and a adder we were able to obtain a critical path that included only one multiplier.

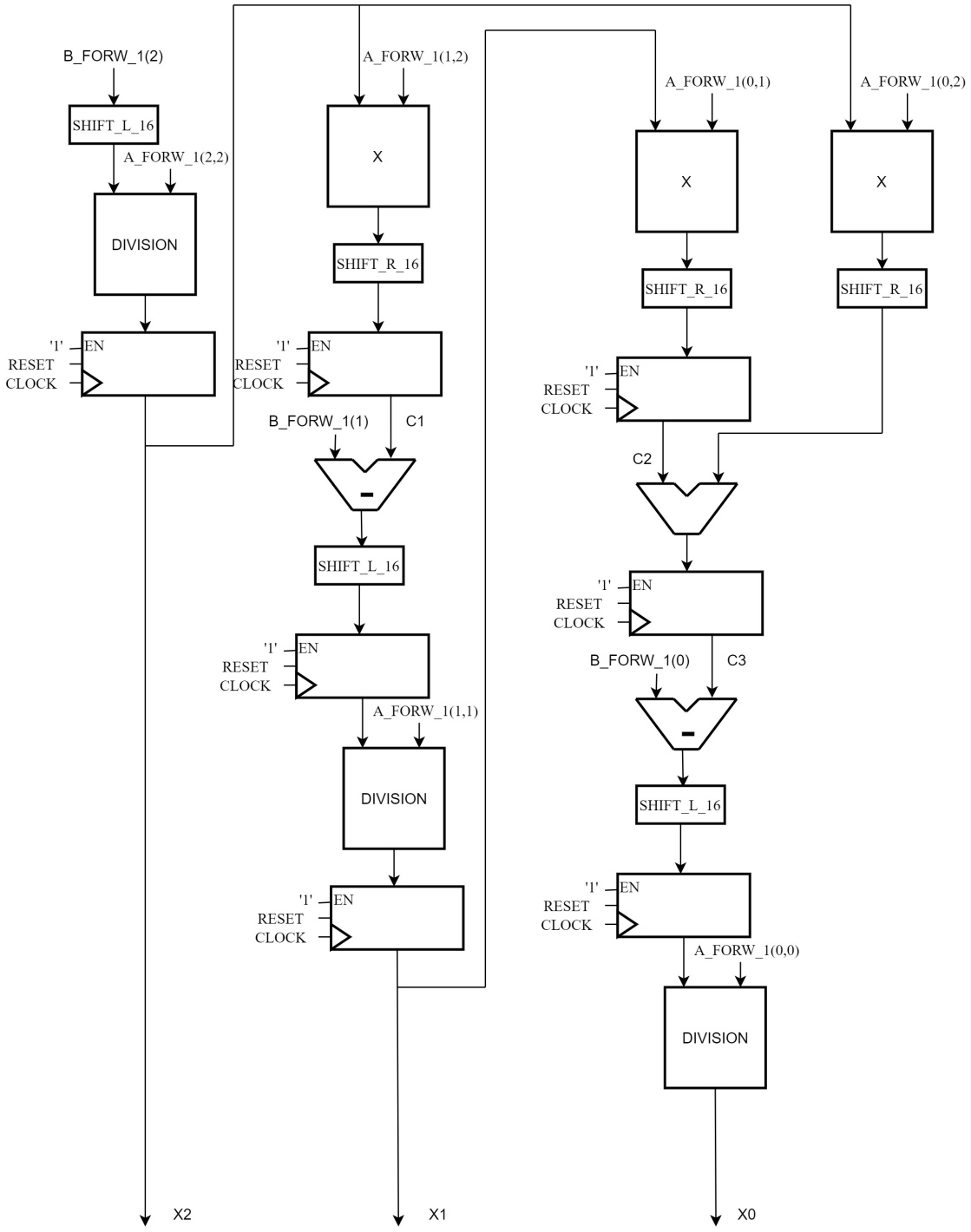

Figure 5.4: Back Substitution High Speed Execution Unit.

5.4 Update a and Update b

Finally, after also dividing the partial pivoting block into two independent blocks, both in **Update a** (5.5) and **Update b** (5.6), pipeline registers are inserted between the different sub-blocks in order to further reduce the length of the combinational paths.

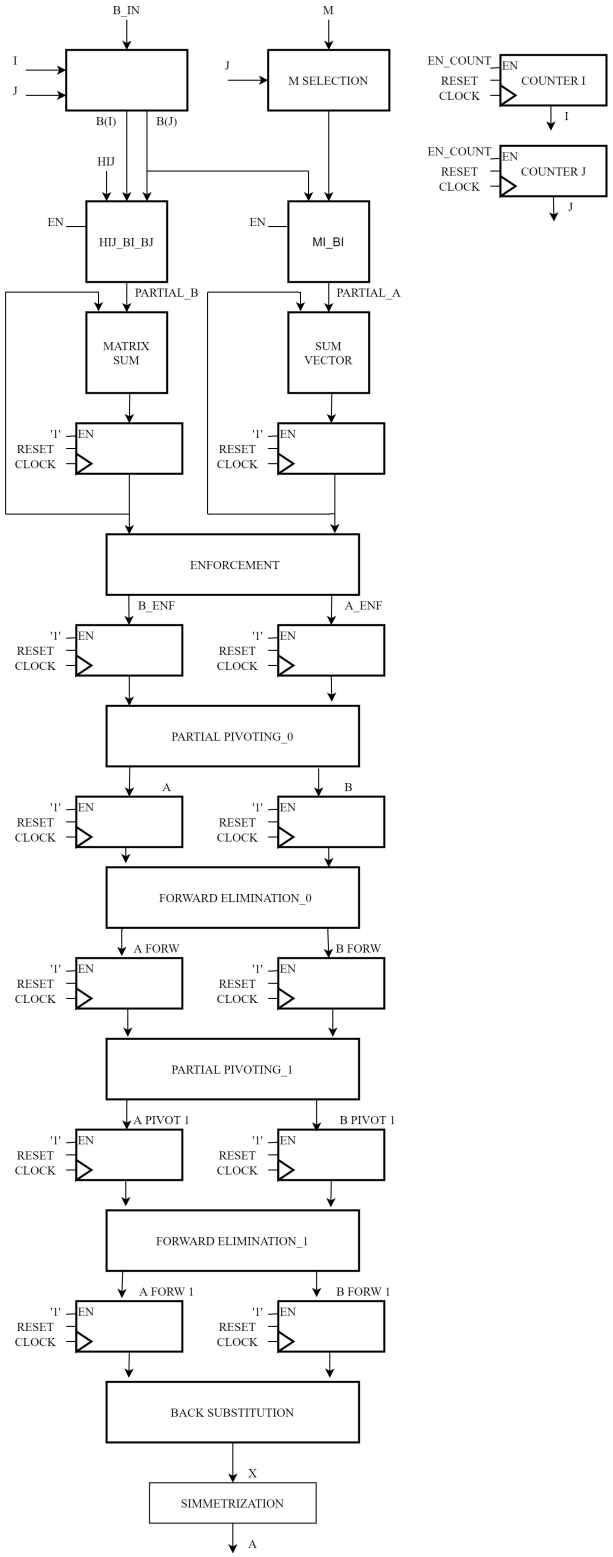

Figure 5.5: Update a High Speed Execution Unit.

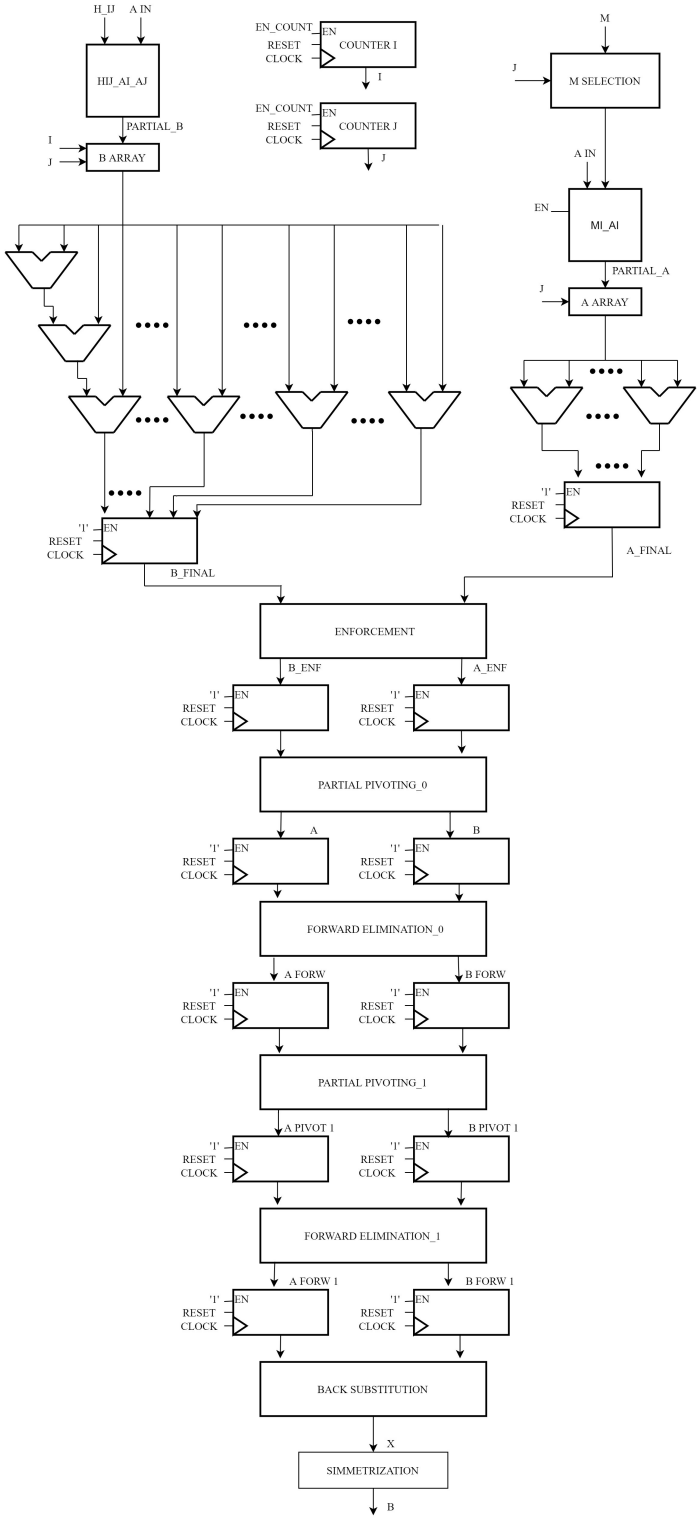

Figure 5.6: Update b High Speed Execution Unit.

Of course, the FSMs (5.7)(5.8) of the two main blocks have been modified, adding wait states to synchronize operations, taking into account the addition of pipeline registers and the new divider, which needs 64 clocks to complete its operation. In this way, even if the FSMs are more complex, we get the same results as the main architecture.

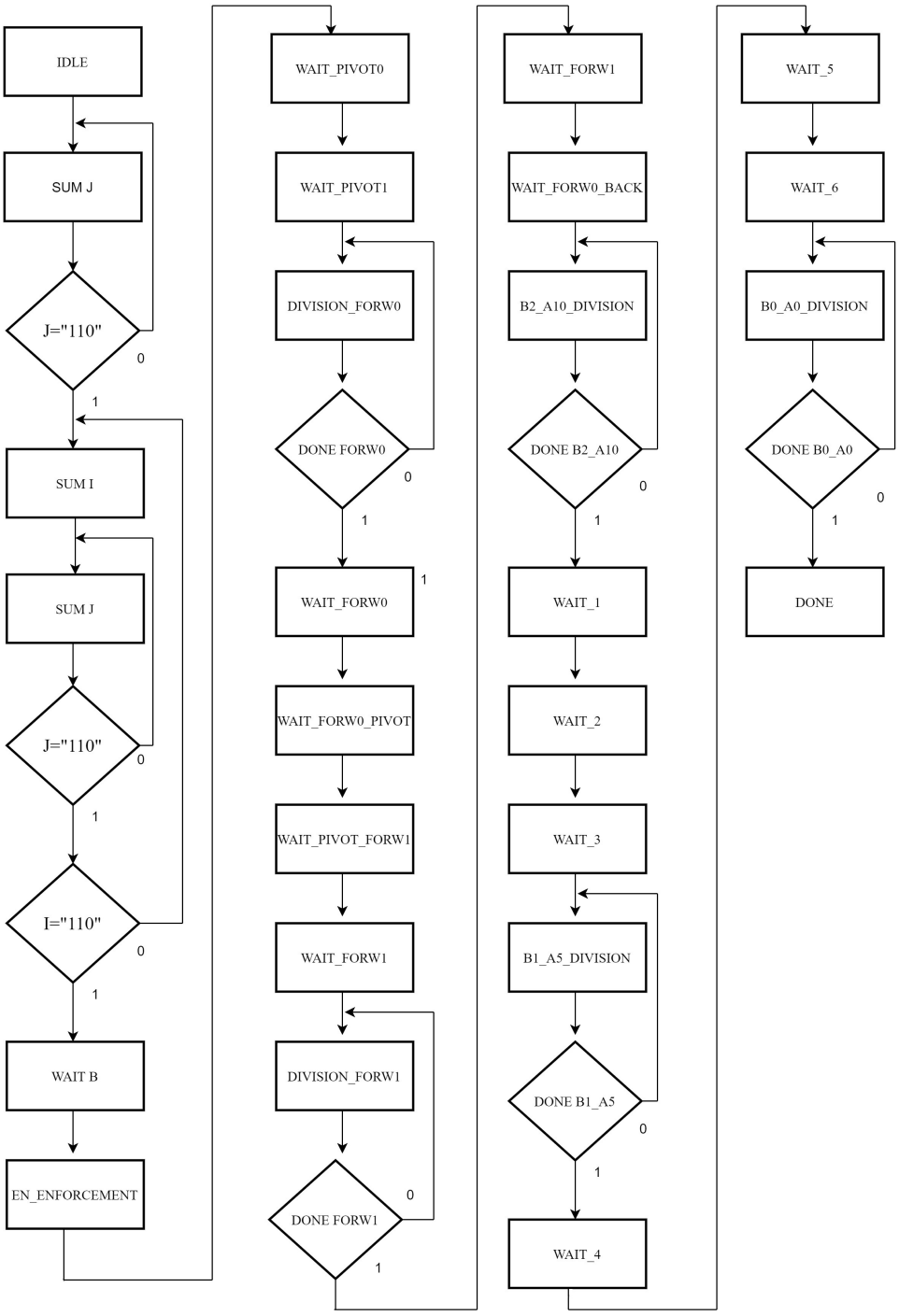

Figure 5.7: Update a High Speed Finite State Machine.

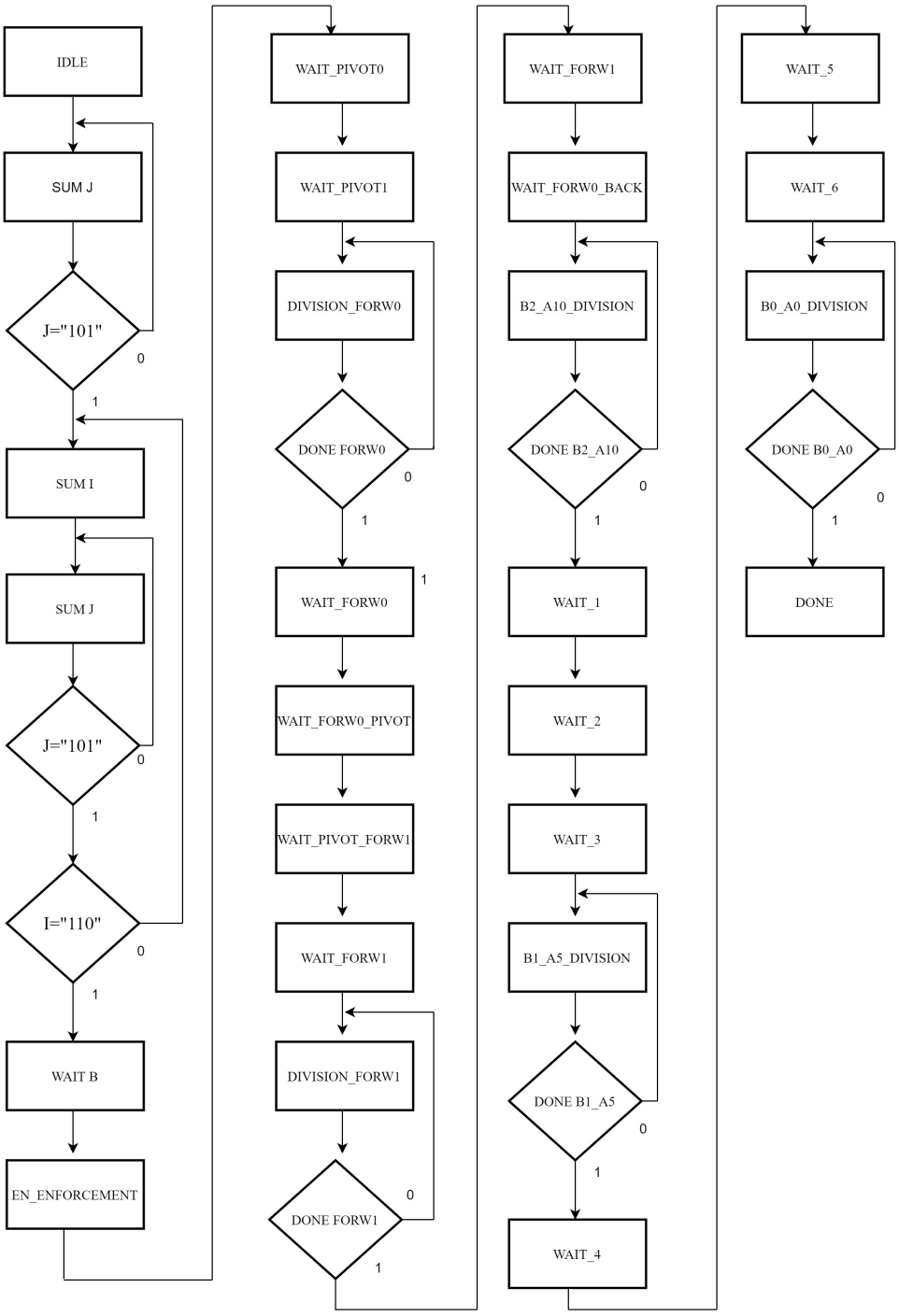

Figure 5.8: Update b High Speed Finite State Machine.

5.5 Simulation and Performance

Once the improvements have been implemented, simulations (5.9) with the same input patterns as the basic architecture have been performed to ensure that the new architecture works properly. As we can see from the simulation, we obtain the same results as the basic architecture, even if we have a different evolution of the states and, since we have added many pipeline registers and inserted the new divider, which requires several clocks to complete the division operation, the new architecture will require much more clocks than the basic architecture.

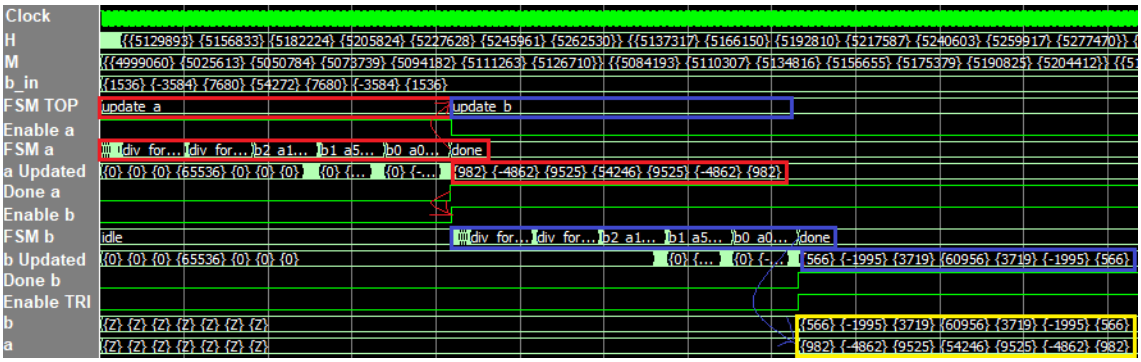

Figure 5.9: Behavioural simulation of the new circuit.

Once the behavioral simulations were completed, the new architecture was synthesized and we obtained the timing report so that we could compare the performance of the new architecture and evaluate its improvements compared to the basic architecture. We have synthesized two versions of the high-speed architecture, one with the pipeline registers, but with the old divider and one complete, both with the pipeline registers and the restoring divider. This is to understand what kind of impact had the two kinds of changes applied.

```
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : WIENER_FILTER
Version: M-2016.12
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top
```

```
Startpoint: UPDATE_B_TOPLEVEL_PORTING/DP/REG_FORW_1_A/PORTING12/REG_OUT_reg[0]
            (rising edge-triggered flip-flop clocked by MY_CLOCK)
Endpoint: UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/REG_DIV3/REG_OUT_reg[63]
          (rising edge-triggered flip-flop clocked by MY_CLOCK)
Path Group: MY_CLOCK
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
WIENER_FILTER      5K_hvratio_1_1        NangateOpenCellLibrary

Point                                                        Incr       Path
------------------------------------------------------------------------------
clock MY_CLOCK (rise edge)                                   0.00       0.00
clock network delay (ideal)                                  0.00       0.00
UPDATE_B_TOPLEVEL_PORTING/DP/REG_FORW_1_A/PORTING12/REG_OUT_reg[0]/CK
(DFFR_X1)
                                                             0.00 #     0.00 r
UPDATE_B_TOPLEVEL_PORTING/DP/REG_FORW_1_A/PORTING12/REG_OUT_reg[0]/Q
(DFFR_X1)
                                                             0.09       0.09 f
UPDATE_B_TOPLEVEL_PORTING/DP/REG_FORW_1_A/PORTING12/DATA_OUT[0]
(REG_64BIT_11)
                                                             0.00       0.09 f
UPDATE_B_TOPLEVEL_PORTING/DP/REG_FORW_1_A/OUT_MATRIX[2][2][0]
(REG_3X3_64BIT_1)
                                                             0.00       0.09 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/IN_MATRIX[2][2][0]
(BACK_SUB_STORING_1)
                                                             0.00       0.09 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/DIVISOR[0]
(DIVISION_3)
                                                             0.00       0.09 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/B[0]
(DIVISION_3_DW01_sub_65)
                                                             0.00       0.09 f
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U16/ZN
(NOR2_X1)
                                                             0.06       0.15 r
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U69/ZN
(AND2_X1)
                                                             0.05       0.20 r
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U17/ZN
(AND2_X1)
                                                             0.05       0.25 r
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U10/ZN
(AND2_X1)
                                                             0.05       0.30 r
UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U11/ZN
```

```
  (AND2_X1)
                                                          0.05       0.35 r
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U9/ZN
  (AND2_X1)
                                                          0.05       0.40 r
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/B2_A10_DIV/sub_add_45_b0/U14/ZN
  (AND2_X1)
                                                          0.05       0.45 r
                              .   .   .

 UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/sub_156/U1/ZN (INV_X1)
                                                          0.03      82.46 r
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/sub_156/U101/ZN (OAI21_X1)
                                                          0.03      82.49 f
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/sub_156/U99/ZN (XNOR2_X1)
                                                          0.05      82.55 f
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/sub_156/DIFF[47]
  (BACK_SUB_STORING_1_DW01_sub_0)
                                                          0.00      82.55 f
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/REG_DIV3/DATA_IN[63]
  (REG_64BIT_1)
                                                          0.00      82.55 f
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/REG_DIV3/U71/Z (MUX2_X1)
                                                          0.07      82.61 f
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/REG_DIV3/REG_OUT_reg[63]/D
  (DFFR_X1)
                                                          0.01      82.62 f
  data arrival time                                                 82.62

  clock MY_CLOCK (rise edge)                             83.00      83.00
  clock network delay (ideal)                             0.00      83.00
  clock uncertainty                                      -0.07      82.93
  UPDATE_B_TOPLEVEL_PORTING/DP/X_COMPUTATION/REG_DIV3/REG_OUT_reg[63]/CK
  (DFFR_X1)
                                                          0.00      82.93 r
  library setup time                                     -0.04      82.89
  data required time                                                82.89
  --------------------------------------------------------------------------
  data required time                                                82.89
  data arrival time                                                -82.62
  --------------------------------------------------------------------------
  slack (MET)                                                       0.27
```

Analyzing the report timing of the architecture with the pipeline registers, but with the old combinatorial divider, we notice that the critical path is still the one that includes the divider, but, with a clock of 10 ns we can have a negative SLACK of -72.73 ns, which implies a maximum operating frequency of 12.08 MHz ($T_{clk} = 82.73ns$), that is an improvement of an order of magnitude compared to the

original architecture.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : WIENER_FILTER_HS
Version: M-2016.12
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

  Startpoint: UPDATE_A_TOPLEVEL_PORTING/DP/COUNTER_I/Q_reg[0]
              (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Endpoint: UPDATE_A_TOPLEVEL_PORTING/DP/SUM_REG_B/PORTING2/REG_OUT_reg[63]
            (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Path Group: MY_CLOCK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  WIENER_FILTER      5K_hvratio_1_1        NangateOpenCellLibrary

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock MY_CLOCK (rise edge)                              0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  UPDATE_A_TOPLEVEL_PORTING/DP/COUNTER_I/Q_reg[0]/CK (DFFR_X1)
                                                          0.00 #     0.00 r
  UPDATE_A_TOPLEVEL_PORTING/DP/COUNTER_I/Q_reg[0]/Q (DFFR_X1)
                                                          0.11       0.11 r
  UPDATE_A_TOPLEVEL_PORTING/DP/COUNTER_I/CNT[0] (COUNTER_N3_0)
                                                          0.00       0.11 r
  UPDATE_A_TOPLEVEL_PORTING/DP/U65/ZN (INV_X1)            0.03       0.15 f
  UPDATE_A_TOPLEVEL_PORTING/DP/U49/ZN (NAND3_X1)          0.03       0.18 r
  UPDATE_A_TOPLEVEL_PORTING/DP/U431/ZN (OAI22_X1)         0.03       0.21 f
  UPDATE_A_TOPLEVEL_PORTING/DP/U456/ZN (AOI221_X1)        0.09       0.30 r
  UPDATE_A_TOPLEVEL_PORTING/DP/U457/ZN (OAI211_X1)        0.05       0.35 f
  UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/B_IN1[1] (HIJ_BI_BJ)
                                                          0.00       0.35 f
  UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/sub_add_24_b0/B[1]
  (HIJ_BI_BJ_DW01_sub_1)
                                                          0.00       0.35 f
  UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/sub_add_24_b0/U141/ZN (INV_X1)
                                                          0.04       0.38 r
  UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/sub_add_24_b0/U104/ZN (AND2_X1)
```

```
                                                          0.04       0.43 r
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/sub_add_24_b0/U59/ZN (AND2_X1)
                                                          0.04       0.47 r
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/sub_add_24_b0/U56/ZN (AND2_X1)
                                                          0.05       0.52 r
                                . . .
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/add_2_root_add_0_root_add_52_3/SUM[63]
(HIJ_BI_BJ_DW01_add_29)
                                                          0.00      10.93 r
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/add_0_root_add_0_root_add_52_3/A[63]
(HIJ_BI_BJ_DW01_add_27)
                                                          0.00      10.93 r
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/add_0_root_add_0_root_add_52_3/U1_63/S
(FA_X1)
                                                          0.11      11.04 f
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/add_0_root_add_0_root_add_52_3/SUM[63]
(HIJ_BI_BJ_DW01_add_27)
                                                          0.00      11.04 f
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/U8960/ZN (AND2_X1)
                                                          0.04      11.09 f
UPDATE_A_TOPLEVEL_PORTING/DP/HIJ_B_I_B_J/H_IJ_BIJ_OUT[0][1][63] (HIJ_BI_BJ)
                                                          0.00      11.09 f
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_B/IN_MATRIX1[0][1][63] (MATRIX_SUM)
                                                          0.00      11.09 f
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_B/add_22_I2_I1/A[63]
(MATRIX_SUM_DW01_add_14)
                                                          0.00      11.09 f
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_B/add_22_I2_I1/U1_63/S (FA_X1)
                                                          0.14      11.22 r
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_B/add_22_I2_I1/SUM[63]
(MATRIX_SUM_DW01_add_14)
                                                          0.00      11.22 r
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_B/OUT_MATRIX[0][1][63] (MATRIX_SUM)
                                                          0.00      11.22 r
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_REG_B/IN_MATRIX[0][1][63] (REG_4X4_64BIT_0)
                                                          0.00      11.22 r
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_REG_B/PORTING2/DATA_IN[63] (REG_64BIT_251)
                                                          0.00      11.22 r
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_REG_B/PORTING2/U4/ZN (NAND2_X1)
                                                          0.03      11.25 f
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_REG_B/PORTING2/U3/ZN (NAND2_X1)
                                                          0.02      11.27 r
UPDATE_A_TOPLEVEL_PORTING/DP/SUM_REG_B/PORTING2/REG_OUT_reg[63]/D (DFFR_X1)
                                                          0.01      11.28 r
data arrival time                                                   11.28

clock MY_CLOCK (rise edge)                               11.40      11.40
clock network delay (ideal)                               0.00      11.40
clock uncertainty                                        -0.07      11.33
```

```
  UPDATE_A_TOPLEVEL_PORTING/DP/SUM_REG_B/PORTING2/REG_OUT_reg[63]/CK (DFFR_X1)
                                                            0.00      11.33 r
  library setup time                                       -0.03      11.30
  data required time                                                  11.30
  ---------------------------------------------------------------------------
  data required time                                                  11.30
  data arrival time                                                  -11.28
  ---------------------------------------------------------------------------
  slack (MET)                                                         0.02
```

Analyzing the report timing of the complete architecture, i.e. with the pipeline registers and the new restoring divider, we can see that the critical path is no more the one that includes the divider, moreover with a clock of 10 ns we can have a negative SLACK of -1.38 ns, which implies a maximum operating frequency of 87.87 MHz ($T_{clk} = 11.38ns$), that is an improvement of a further order of magnitude compared to the original architecture. Analyzing the critical path highlighted by this last report timing we can see how it involves the $\boldsymbol{H_{ij}b_ib_j}$ block, so, as it was done for the divider, we decided to synthesize the block individually in order to perform detailed analysis.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : HIJ_BI_BJ
Version: M-2016.12
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

  Startpoint: REG_B1/REG_OUT_reg[1]
              (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Endpoint: REG_OUT/PORTING7/REG_OUT_reg[63]
            (rising edge-triggered flip-flop clocked by MY_CLOCK)
  Path Group: MY_CLOCK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  HIJ_BI_BJ          5K_hvratio_1_1        NangateOpenCellLibrary

  Point                                                    Incr       Path
  ---------------------------------------------------------------------------
  clock MY_CLOCK (rise edge)                               0.00       0.00
```

```
clock network delay (ideal)                                0.00       0.00
REG_B1/REG_OUT_reg[1]/CK (DFFR_X1)                         0.00 #     0.00 r
REG_B1/REG_OUT_reg[1]/Q (DFFR_X1)                          0.11       0.11 r
REG_B1/DATA_OUT[1] (REG_32BIT_0)                           0.00       0.11 r
sub_add_65_b0/B[1] (HIJ_BI_BJ_DW01_sub_1)                  0.00       0.11 r
sub_add_65_b0/U44/ZN (NOR2_X1)                             0.03       0.14 f
sub_add_65_b0/U49/ZN (AND2_X2)                             0.04       0.19 f
sub_add_65_b0/U37/ZN (AND2_X1)                             0.04       0.23 f
               .   .   .
add_1_root_add_0_root_add_99_3/SUM[63] (HIJ_BI_BJ_DW01_add_16)
                                                           0.00      10.56 f
add_0_root_add_0_root_add_99_3/B[63] (HIJ_BI_BJ_DW01_add_15)
                                                           0.00      10.56 f
add_0_root_add_0_root_add_99_3/U1_63/S (FA_X1)             0.14      10.70 r
add_0_root_add_0_root_add_99_3/SUM[63] (HIJ_BI_BJ_DW01_add_15)
                                                           0.00      10.70 r
U15773/ZN (AND2_X1)                                        0.04      10.74 r
REG_OUT/IN_MATRIX[1][2][63] (REG_4X4_64BIT)                0.00      10.74 r
REG_OUT/PORTING7/DATA_IN[63] (REG_64BIT_10)                0.00      10.74 r
REG_OUT/PORTING7/U130/ZN (NAND2_X1)                        0.02      10.76 f
REG_OUT/PORTING7/U129/ZN (NAND2_X1)                        0.02      10.79 r
REG_OUT/PORTING7/REG_OUT_reg[63]/D (DFFR_X1)               0.01      10.80 r
data arrival time                                                    10.80

clock MY_CLOCK (rise edge)                                10.90      10.90
clock network delay (ideal)                                0.00      10.90
clock uncertainty                                         -0.07      10.83
REG_OUT/PORTING7/REG_OUT_reg[63]/CK (DFFR_X1)              0.00      10.83 r
library setup time                                        -0.03      10.80
data required time                                                   10.80
-------------------------------------------------------------------------
data required time                                                   10.80
data arrival time                                                  -10.80
-------------------------------------------------------------------------
slack (MET)                                                         0.00
```

Observing then, the report timing of the $\boldsymbol{H_{ij}b_ib_j}$ block we can see that the new critical path includes 2 multipliers and 1 adder. By trying to reduce the length of this path by inserting pipeline registers between the operators, we could reduce the combinatorial path to a single multiplier, allowing us to reach a maximum frequency of 117 MHz for the block. But inserting the pipeline registers means an excessive increase in latency. This is due to the fact that the block $\boldsymbol{H_{ij}b_ib_j}$ is used 49 times and each time it makes 49 multiplications. Since the increase in latency compared to the frequency gain is excessive, we have decided not to make any changes to the $\boldsymbol{H_{ij}b_ib_j}$ block, maintaining the maximum operating frequency at 87.87 MHz.

Chapter 6

Conclusions

Video coding is a fast-growing sector and will continue to be in the future due to the increasing diffusion of video applications and the demand for higher quality. This means that encoding and decoding processes need to be further improved. To do this, an excellent solution is to create dedicated hardware that executes the various algorithms that compose the entire codec.
In this thesis work, we have presented a hardware architecture that allows us to run one of these algorithms and a version of it that works at high speed. Being a completely new architecture, we are sure that further improvements can be made later by those who will have the opportunity to continue the work we started. In particular, the basic architecture is an excellent starting point to be able to create a new architecture that satisfies specific design needs, as it has been done for the high-speed architecture.

Bibliography

[1] Visual Networking Index Cisco (VNI Cisco). *Cisco Annual Internet Report 2018-2023*, March 2020.

[2] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2006.

[3] M. Jacobs, J. Probell. *A Brief History of Video Coding*, January 2009.

[4] Alliance for Open Media. *http://aomedia.org*

[5] Yue Chen et al. *An Overview of Core Coding Tools in the AV1 Video Codec*. In Picture Coding Symposium (PCS), June 2018.

[6] Wikipedia. *https://en.wikipedia.org/wiki/AV1*

[7] Andreas S. Panayides, Marios S. Pattichis, Marios Pantziaris, Anthony G. Constantinides, Constantinos S. Pattichis. *The Battle of the Video Codecs in the Healthcare Domain - A Comparative Performance Evaluation Study Leveraging VVC and AV1*. In IEEE Access, January 2020.

[8] Md Abu Layek, Ngo Quang Thai, Md Alamgir Hossain, Ngo Thien Thu, Le Pham Tuyen, Ashis Talukder, TaeChoong Chung, Eui-Nam Huh. *Performance analysis of H.264, H.265, VP9 and AV1 video encoders*. In 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2017

[9] Debargha Mukherjee, Shunyao Li, Yue Chen, Aamir Anis, Sarah Parker, James Bankoski *A switchable loop-restoration with side-information framework for the emerging AV1 video codec*. In IEEE International Conference on Image Processing (ICIP), 2017.