POLITECNICO DI TORINO

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

Master's Thesis

VLSI architecture of a low-complexity Wiener Filter for video coding

Supervisor:
Prof. Maurizio Martina

Candidate:
Giorgio Armanno

October 12, 2020

Acknowledgements

I would like to thank, first of all, Professor Maurizio Martina who gives me the chance to develop this work increasing my interest on the VLSI design subject that i have taken during all my course of study. I thank him for supporting me, helping me during the critical part of the work and sharing its precious knowledge with me. I decided to write the following part in Italian to be better understood by the people who this personal section is dedicated to.

Ci tengo a ringraziare il mio amico e collega Sandro che ha contribuito alla realizzazione di questa tesi e con cui ho condiviso la maggior parte della mia giornata negli ultimi mesi. Lo ringrazio per aver messo tutto il suo impegno e le sue conoscenze. Ringrazio i miei ex-colleghi ed ora amici Ico e Fè, che mi hanno aiutato ad ambientarmi in una città nuova diventando un punto di riferimento per me. Ringrazio il mio amico Jack, conosciuto per caso ed insostituibile compagno di avventure. Ringrazio Ciccio, non solo amico, ma coinquilino, confidente, fratello, senza il quale probabilmente non sarei arrivato fin qui. Ringrazio Simone, cugino acquisito ma vero, per esserci sempre stato. Ringrazio il mio amico Pupo, compagno di giochi e di chiacchiere.

Ringrazio l'amore della mia vita Nadia, che mi ha tenuto la mano negli ultimi 10 anni, che mi è stata accanto in ogni momento di difficoltà e di debolezza, che non ha mai dubitato di me , accettando le mie ansie e i 1522 km di distanza. La ringrazio per avermi aspettato senza mai lamentarsi e per emozionarsi sempre al posto mio. Ringrazio tutta la sua famiglia e le sue "mie" sorelle.

Ringrazio la mia famiglia per il supporto che non è mai mancato: i miei fratelli tutti, 3 padri per me. I loro figli per la gioia che portano in casa. Mia nonna, per cui non basterebbe un libro intero per esprimere la graditudine che nutro nei suoi confronti e gli altri nonni che sarebbero stati tanto felici di esserci. I miei zii e il resto della famiglia per la loro stima.

Infine i miei genitori, anche se non basta un grazie per ripagare il sostegno, morale prima che economico, e la forza che mi hanno dato in questo percorso. Li ringrazio per essersi privati per me, per averci sempre creduto e per avermi messo sempre in cima a tutto.

Ringrazio chiunque abbia creduto anche un minimo in me.

Abstract

The aim of presented thesis work is to provide a special purpose hardware implementation of AOMedia AV1 Wiener Filter. The basic idea was to start from the analisys of the complete AV1 codec and then focus on a very specific part based on the *profiling* results of the codec, in order to understand the percentage of use for each function and evaluate which one needed of more attention. The work presented here concerns the design, the implementation, the analysis and a possible optimization of an Hardware architecture for the Wiener Filter of AV1. The reasons why the attention was focused on the Wiener filter are multiple: first of all, the lack of informations about this kind of process in literature. Moreover, the importance of the *in-loop filters* in AV1 codec mechanism of improving the quality of the output images and, finally, the considerations about its usage in the coding process that will be better explained in thesis development. The first attempt to implement the architecture was to try to start from the source C code (available on AOMedia Website using GitHub) and build an equivalent model based on matrix calculus. This approach has been left because of many drawbacks related to the size of involved data that will be explained during the following chapters. So the choice was to follow completely the C code to maintain consistency : the basic implementation derives from an algorithm to architecture mapping in terms of data, operations, flow and parallelism. The basic idea of the presented work is not only to show the steps and the design choices to obtain a working VLSI implementation starting from some binding inputs, but also how it is possible to improve it based on the application to develop. So, once obtained a working architecture, it has been synthesized with Synopsys to evaluate the critical points and understand the direction to follow to improve that performances. Due to the huge dimension of many internal component of the filter and the very high parallelism, the *complexity* has been evaluated like one of the most critical parameter to optimize. Starting from the basic implementation and by using deeply a *folding* approach on the bigger component inside the architecture, a **low-complexity** version has been implemented, obtaining an area reduction of about 90%.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 A brief introduction to Video Codec

According to a Cisco Research [1],video content composes more than 70% of internet traffic today, and a growth of more than 80% worldwide is expected by 2021 as well as the demand of a greater resolution both for images and videos is increased. Nowadays for each electronic device, such us smartphone, tablet or TV a Full HD resolution is available and 4K technology is quite common, leading to a constant need of more memory space to store these bigger informations and, moreover, of a more efficient data compression system, able to remove spatial and temporal redundancy to improve images and video processing.

The key component of a video processing is the codec. It is used to compress and decompress a digital media and it is mainly composed by two part: encoder and decoder. The first one performs the compression (or *encoding*), while the second performs the decompression (or *decoding*). Typically, the compression is lossy, since it causes a loss of informations from the uncompressed video, damaging video quality. Obviously, the accuracy of reconstruction of the original uncompressed video changes according to the specific codec used and can be determined by considering many relevant parameters, such as the amount of data used to represent the video (called *bit-rate*), the sensitivity to be "lossy", the complexity of encoding and decoding algorithm and so on.

The video codec behavior can be summarize in few passages: video is just a sequence of pictures, and the encoder takes one of that at a time and compresses it.The already encoded pictures are used to make a prediction of the actual one: the coding algorithm computes the difference between the predicted and the real current picture, and then processes it by means of a DCT (*Discrete Cosine Transform*) unit or other transform coding algorithm. After a quantization step, the resultant bit-stream is released as output and sent to decoder.

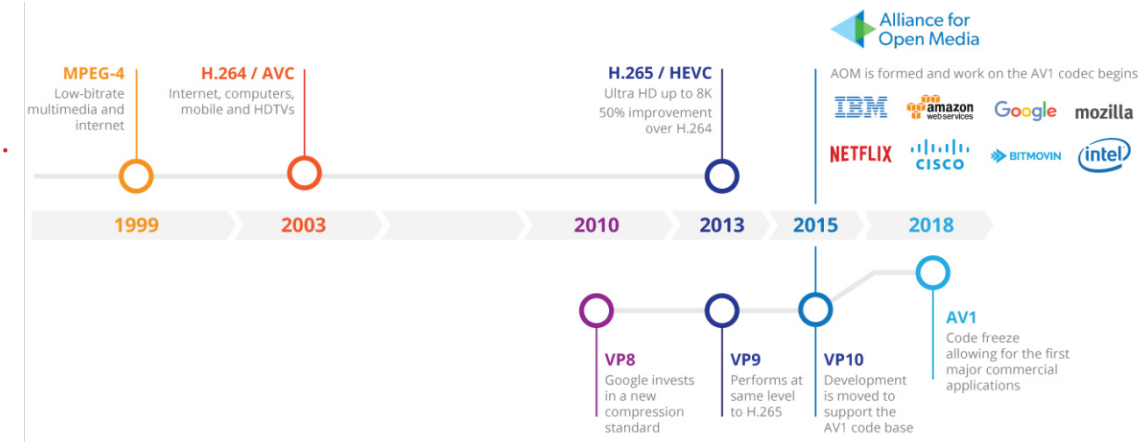


Figure 1.1: The historical delevopment of video codecs [5]

1.2 The next generation *open-media* codecs : AOMedia AV1

The most important companies all over the world is spending much time and money trying to improve, even in small quantity, the file sizes or the image quality because this leads to many benefits both for video creators and consumers. In the last years the need for an *open media* codec has increased with the growing of internet video contents, because many companies spent ten or thousand millions to compress and decode video files and, also, because the triumph of the internet is founded on the fact that the basic technologies (like browsers, operating system..) are open and available to be freely used. The combination of those needs led several big companies to create some alternatives to avoid paying very expensive licensing fees a year: for example, in 2013 Mozilla created a functional prototype of *Daala Project* while Google was promoting *VP9*. The main goal was to create a new generation of video coding, able to share video fast, easy and at low cost. In this panorama Mozilla, Google and Cisco, with Amazon and Netflix and some hardware vendors like AMD and Intel, founded AOMedia in 2015 that, in 2018, published the first version of AV1 [2], a video codec largely based on VP9 but including many significant improvement, primarily the full compatibility with *W3C Patent Policy* [3]: essentially it can be fully implemented with a royalty-free licensing requirements. This helps the companies to broadcast in an efficient way video over the internet, respecting the required resolution and quality. An overview of the evolution of video codec technology is reported at figure 1.1

The powerful innovation came with AV1 is the compatibility among all web technological support, like browsers. Apart from Google and Mozilla that, being AOMedia members obviously support AV1 playback, also many other web browsers

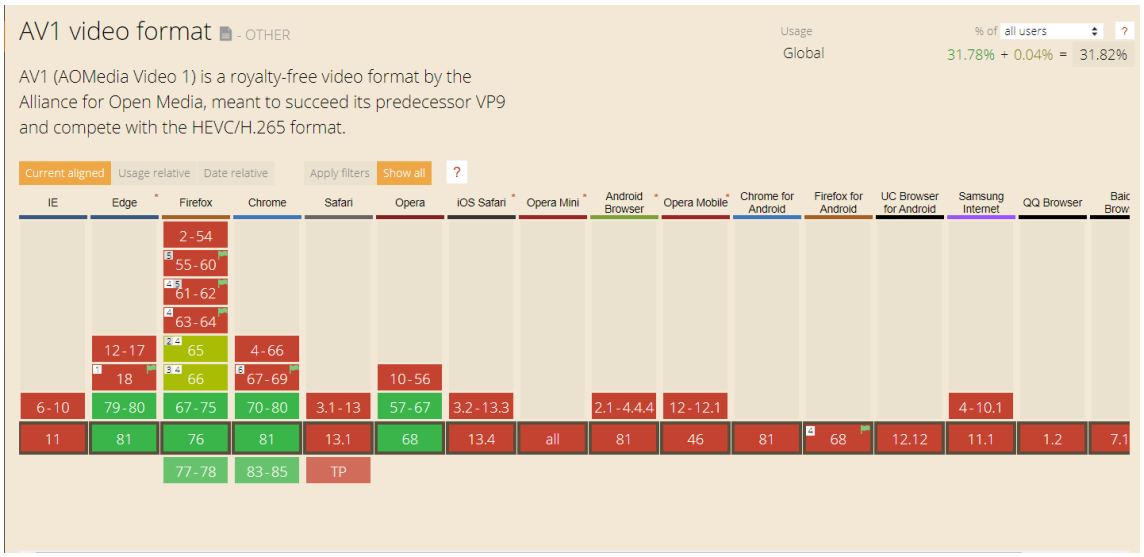


Figure 1.2: AV1 usage percentage of all browsers

play AV1 providing high quality video content reducing bandwidth. Nowadays, according to "Can I Use" [4], AV1 is used in 31.82% of all browsers, like in figure 1.2.

1.3 AV1 Video Coding

The interest in the analysis of several aspects of AOMedia AV1 derives from the innovation that it represents due to the perpetual attempts to improve the codec mechanism by testing and proposing new coding tools. Starting from an initialization set almost equal to the VP9 one, nowadays AV1 is in a final-phase where very special purpose features have been designed and an high-level syntax has been reached.

1.3.1 A comparison between AV1 and other video codecs

It is necessary to do a premise before concentrating on how AV1 is different from other video codecs. As a matter of fact, making an objective comparison between video codecs is very difficult because of the difference between video coding standards and the different implementations for the related encoders: the point is that standards can not be simulated, only their implementation can be used. So, even if two different encoder implementations produce comparable bit-streams using the same standard, they can be very different. This is why at the *Picture Coding Symposium* (PCS), in 2018, were presented different works of comparison between video

coding standards. In particular, for a practical *Over-the-top* (OTT) streaming application AV1 was compared to standard video codecs like VP9 and HEVC using a proper input data set suitable for that applications and able to cover a range of video sequences with different properties [5]. The AV1 average bitrate has been observed to be reduced of 13% respect to VP9 and 17% respect to HEVC and its BD-rate (Bjontegaard rate) reduction increases to a range of 22%-27% respect to VP9 and 30%-40% respect to HEVC. So, for an almost standard input sequence, AV1 outperforms both VP9 and HEVC. This kind of result is confirmed also by other researches, that report an improvement up to 30% of the average bitrate than VP9, accepting to pay obviously a reasonable increases in encoding complexity [6]. These results are useful only to give an idea of the video coding performances and an indication on the historical reasons that led to the rise of AV1, so it should be interpreted carefully without forgetting the initial premise. However, the discussions on the comparison between video codecs performances produced also completely different results: some work reports that HEVC performs better than AV1 [7] with plausible conclusions. It seems strange that opposite minds appear correct at same time, indeed the point is that it depends on the choice of codec implementation as mentioned before, configuration, metrics and test sequences, since each different codec works differently depends on the type of content it has to perform.

1.3.2 AV1 block diagram

A reference implementation of decoder and encoder is published by AOMedia and available on its official website [8]. By analyzing their description in C language it is possible to build a block diagram concerning the main processing stages of AV1 codec algorithm, and it is represented in figure 1.3.

The coding process starts taking one frame at a time from the initial sequence of pictures. Each current frame is divided into different size parts before being encoded. Then, it is processed by intra prediction or inter prediction based on the kind of frame: in particular, *intra prediction* is employed to exploit spatial redundancy and correlation within pixels inside the same frame while *inter prediction* is used to exploit temporal redundancy between more subsequent frames by employing the *motion estimation*. What is elaborated from prediction blocks is subtracted to the original input frame to compute the error made by prediction,then processed by means of transformation and quantization algorithm and finally coded with a non-binary *entropy coding* to be transmitted. In addition to the encoding operation, the encoder enables a decoder to improve prediction quality by using the previous reconstructed frame: the quantized data is inversely quantized and transformed to be summed to the prediction signal. AV1 uses also several *loop filtering* and post-processing tools to the processed frame, like *Film Grain Synthesis*, to improve reconstruction quality.

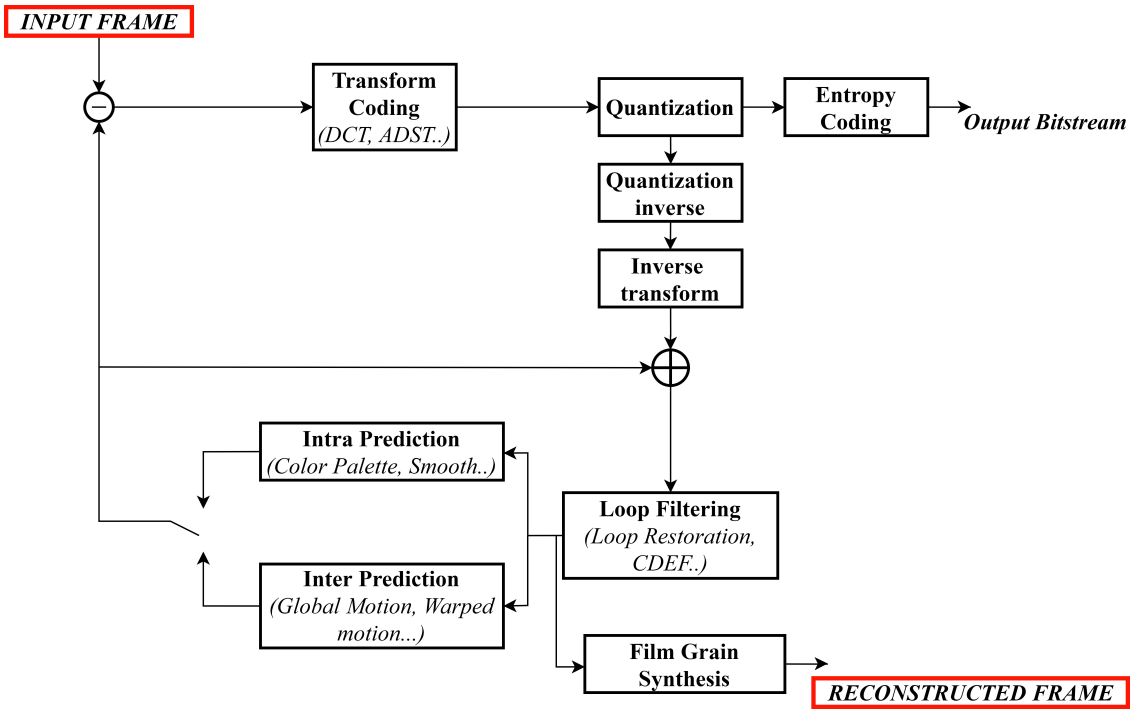


Figure 1.3: AV1 block diagram

1.3.3 AV1 Coding Technique

- *Transform Coding*: as mentioned before, AV1 allows partitioning in units of different sizes, differently from VP9. In particular, it supports squares (2:1 and 1:2) and rectangles (4:1 and 1:4) sizes, going from a 4 x 4 to 64 x 64 pixels. AV1 exploits also a set of *transform kernel*, where DCT, ADST, flipADST and IDTX are involved and applied both in horizontal and vertical directions.

- *Coding Partition*: frame is separated into adjacent same-sized blocks called "superblocks" but, differently from VP9, AV1 use 10 structures of partition starting from a 128 x 128 or 64 x 64 frame because it includes also rectangular partitions absent in VP9. The recursive partitions is maintained until reaching 4x4 pixels.

- *Intra Prediction*: AV1 updates VP9's intra prediction modes including several tools developed as follows:

  – *Enhanced Directional Intra Prediction*: the directional intra mode angle has been improved to have a better resolution. Starting from 8 nominal angles, a step size of 3 degree is introduced in order to obtain an overall angle given by the sum between the nominal angle and a "delta angle",

that add a contribute that belong to the interval from 0 to 3 times the step size.

- *Non-directional Smooth Intra Predictors*: a quadratic interpolation in both horizontal and vertical directions is employed by means of three new smooth predictors : SMOOTH_V, SMOOTH_H and SMOOTH.
- *Recursive-filtering-based Intra Predictor*: five FILTER_INTRA modes are designed to remove spatial redundancy. Each mode is represented like 7-pixels filter able to exploit correlation between pixels in a 4 x 2 block and 7 neighbours. Each block is predicted based on the different weight filters gives to the neighbours.
- *Chroma Predicted from Luma(CfL)*: chroma pixels are processed as linear function of reconstructed luma pixels without using decoder, but just determining proper parameters from original chroma pixels and then signaling them in the bitstream [9].
- *Color Palette as a Predictor*: to simplify the process of artificial videos, many blocks is approximated by using a small number of single colors (up to 8).
- *Intra Block Copy*: Intra predictor uses informations about reconstructed blocks in the same frame, as happens with Inter predictor between subsequent frames. This optimizes the prediction process especially from the videos characterized by a repeated content within the same frame.

- *Inter Precition*: differently from VP9, where a maximum of 2 references between 3 possible ones are used for motion compensation module, in AV1 intra coder is more powerful since it extends the set of reference frames motion vectors. So, the prediction is improved by means of several algorithms:

  - *Extended Reference Frames*: the number of references for each frame is extended from 3 (VP9) to 7. VP9 used:
    - "LAST" frame, that is the nearest past one;
    - "GOLDEN" frame, that is the distant past one;
    - "ALTREF" frame, that is the temporal filtered future one;

    AV1 adds two past frames, called "LAST2" and "LAST3" and two future frames, called "BWDREF" and "ALTREF2". What is created is a multilayer model of prediction where frames are shared alternatively and can be used individually or combined in a pair to increase the number of possible combination of references.
  - *Dynamic Spatial and Temporal Motion Vector Referencing*: AV1 develops a motion vector coding incredibly improved, since it exploits both spatial

and temporal candidates. It means that, starting from a certain frame, the motion vector consult neighborhood to generate the spatial references set, but exploits also a temporal motion estimation to generate temporal references.

The motion estimation works in three steps:

* Motion vector buffering;
* Motion trajectory creation;
* Motion vector projection.

The extimation procedure can be explained as follows: a reference frame index and the correspondent motion vectors are stored for each coded frames and, before decoding, the possible motion trajectories going from one frame to another one are examined and recorded. Once all the reference frames have been determined, the motion extimated frame are derived by applying the motion trajectories to the desired reference frames. At this point, temporal informations are included and then "filtered" until reaching a maximum of 4 final references.

– *Overlapped Block Motion Compensation (OBMC)*: a combination of predictions is made by using smoothing filters in both horizontal and vertical filter to decrease prediction errors, especially in block edges.

– *Warped Motion Compensation*:it is exploited by both global and local warped motion compensation.

* The local warped motion describes each model parameters at the block level by motion vectors assigned to neighborhood to analyze the local motion variation with a "minimal overhead"
* The global warped motion analyze the motion models from the current frame and the reference.

The real advantage of AV1 warping is the possibility to be implemented in an efficient way just shearing horizontally and then vertically, using proper "8-tap interpolation filters" for each shear.

– *Advanced Compound Prediction*: AV1 includes several new tools for compound prediction, that can be generalized by the formula:

$$p_f(i,j) = m(i,j)p_1(i,j) + (1 - m(i,j))p_2(i,j) \tag{1.1}$$

where, for a pixel (i,j),p_1 and p_2 are two predictors, p_f is the final predictor and $m(i,j)$ are the weighting coefficients belonging to range [0-1]

- *Entropy Coding*: as mentioned before, AV1 uses a non-binary entropy coding, based on a multi-symbol arithmetic to reach an higher precision than the

maximum achievable with binary configuration. Each syntax element belongs to a precise alphabet of elements.

- *Enhancement Filter*: AV1 is characterized by in-loop and post-processing tools. These technique don't have necessarily to do with efficiency or encoding improvement , their aim is just to make the output image look better. To be more precise, AV1 uses a combination of pre-processing filters, post-processing filters and in-loop filter, that belongs to the coding process. A basic structure of an AV1 coding loop filters is presented at 1.4

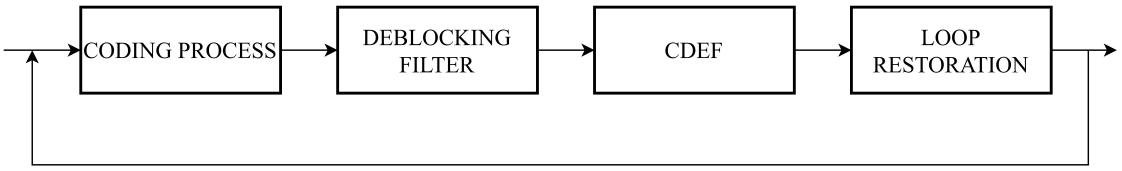


Figure 1.4: AV1 coding loop filters

It is possible to identify three different kind of filters. The first is called *deblocking filter*: it remove artifacts at the edges of the codec blocks, mainly due to DCT that, even if works very well to compact energy, it tends to accumulate error at the edges. The second one is called *Constrained Direction Enhancement Filter (CDEF)* and removes particularly noise formed at sharper edge. Since it is directional it can be used directly in the edges and it can follow them instead of filtering in every possible directions as happens in most other filters. The third filter, called Loop Restoration filter is composed by two configurable filters: a *Wiener Filter* and a *Self-Guided Filter*.This is that's the thesis work focuses on and it is better explained in Chapter 2

  - *Self-Guided Filter*: as the name suggests, in this case the guide picture is the same as the imagine to be filtered. The decoder employees two filters of this type, a 3x3 and 5x5 and then their outputs are weighting combined obtain the final restored version.
  - *Wiener Filter*: the filtering is exploited by using a 7x7 separable Wiener filter, combining symmetry and normalization constraints to reduce the number of filtered parameters sent.

1.4 Thesis organization

The thesis work is organized as follows:

- This is the introductive *Chapter 1*, where a brief and concise overview about video codec and more in details AV1 tools has been presented to illustrate the panorama in which this work has been developed and give the basics to better understand what is displayed on the next chapters.
- The *Chapter 2* focuses on a more precise presentation about Wiener Filter concepts and a detailed explanation of its C implementation for AOMedia AV1 video codec, available on AOMedia website through GitHub as mentioned.
- The *Chapter 3* is about the development of a basic Hardware implementation for the filter including all the basic blocks involved: starting from the C development, each mapped Hardware component is presented reaching the final version of the VLSI architecture. The analysis of obtained results in terms of speed and complexity is proposed, underlining the reasons why a complexity optimization is necessary and which is the best way to reach it.
- The *Chapter 4* concerns the development of a low-complexity implementation, illustrating which blocks have been modified and how.
- In the *Chapter 5*, the final one, a comparison between the previous two architectures is presented in order to understand the level of improvement reached.

Chapter 2

Wiener Filter overview

2.1 AV1 Loop Restoration Unit

The field of the image restoring is complex since it involves an huge number of different technique to increase image or video quality, working for example on blurs, banding, noise or contrast. The interest on this topic is increased when these tools started to be used inside image and video codec both to recover some information lost from compression and make them visibly more pleasant. The problem to overcome regards the integration of these schemes inside a video codec process because of their high-complexity : implementing a restoration tool able to work on input contents that could be high-resolute could requires a very high computational cost, expecially if binding constraint have to be respect. The complexity have been reduced in modern video codecs by optimizing restoration paradigms and working on data transmission from encoder to the decoder.

The AV1 Loop Restoration Unit is composed by two switchable filters:

- *Self-Guided Filter*: as the name suggests, in this case the guide picture is the same as the imagine to be filtered. The decoder employees two filters of this type, a 3x3 and 5x5 and then their outputs are weighting combined obtain the final restored version.
- *Wiener Filter*: the filtering is obtained by using a 7x7 separable Wiener coefficient, combining symmetry and normalization constraints to reduce the number of filtered parameters sent.

2.2 Wiener filter behavior

In video coding process, Wiener Filter is used to reconstruct a degraded frame by means of a non-causal filter where each pixel is considered in a w x w window

around it, where w is n odd number such that $w = 2r + 1$, with r an integer number representing the radius of involved window [6]. It is worth to noting that the filtering block doesn't work with the nominal value of w^2 input taps, but it receives a processed version of the taps contained in the matrices H and M. In particular:

- H is given by
$$H = E[XX^T] \tag{2.1}$$
that is the autocovariance of X, the column-vectorized version of the w^2 input taps.

- M is given by
$$M = E[YX^T] \tag{2.2}$$
that is the cross correlation between of X and the source pixel Y.

Clearly this approach requires to transmit w^2 values for each filtered pixel and increments both bit rate and decoding complexity. For this reason many constraints needs to be imposed [6]:

- The resultant filter has to be separable: it means that filtering can be implemented separately for horizontal and vertical w taps.

- Each horizontal and vertical filter have to be symmetric.

- Horizontal and vertical filter coefficients can not assume all possible values, but such that their sum is exactly "S" for both filters, where S is a constant value that, for the AV1 implementation is equal to 2^{16}.

These constraints make the filtering very powerful since they allow to send for each filter just r values instead of w. Moreover, since the filter operates only to compute of the first r elements, the implementation complexity is reduced considering that both the vertical and horizontal filter, hereinafter called a and b respectively, can be derived as follows:

$$a(i) = a(w - 1 - i), i = 0, 1, ..r - 1 \tag{2.3}$$

$$a(r) = S - 2\sum_{i=0}^{r-1} a(i) \tag{2.4}$$

$$b(i) = b(w - 1 - i), i = 0, 1, ..r - 1 \tag{2.5}$$

$$b(r) = S - 2\sum_{i=0}^{r-1} b(i) \tag{2.6}$$

The filtering process follows a simple iterative scheme: it starts with an initial value of horizontal and vertical filters and optimize (a in this case) one of them while the other is kept fixed (b_{in}). Once the first the *r-taps* version of the filter is obtained, it is reconstructed using the equations 2.3, 2.4, 2.5 and 2.6 and then used as input for the other filter processing. The Wiener filter process is represented at figure 2.1.

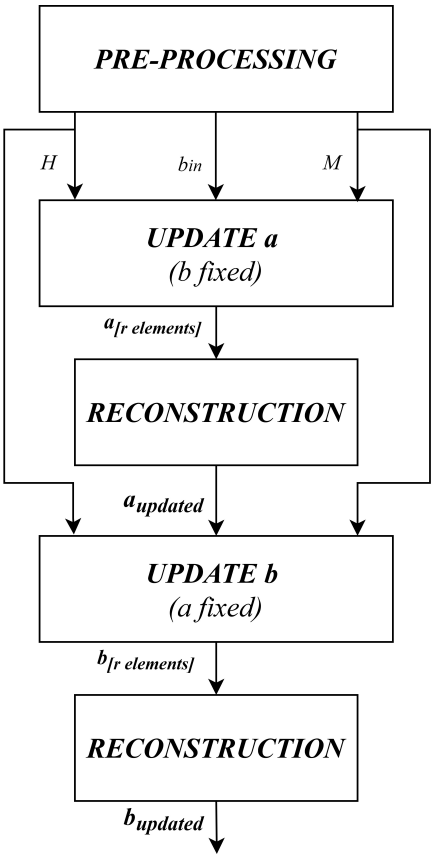


Figure 2.1: Wiener Filter process

2.3 C model from AOMedia AV1 Codec Library

As mentioned before, AV1 library source code is available online at [8]. The software reference for Wiener Filter is reported in *"pickrst.c"* function inside *"search_wiener"* source code. Before going on it is necessary to clarify some parameters that is often used inside C model implementation:

- In AV1 implementation, $r = 3$ and so $w = 7$.

- H is w x w matrix of w x w matrix H_{ij}, so it has a dimension of w^2 x w^2 (that is a 49x49).

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} & H_{04} & H_{05} & H_{06} \\ H_{10} & H_{11} & H_{12} & H_{13} & H_{14} & H_{15} & H_{16} \\ H_{20} & H_{21} & H_{22} & H_{23} & H_{24} & H_{25} & H_{26} \\ H_{30} & H_{31} & H_{32} & H_{33} & H_{34} & H_{35} & H_{36} \\ H_{40} & H_{41} & H_{42} & H_{43} & H_{44} & H_{45} & H_{46} \\ H_{50} & H_{51} & H_{52} & H_{53} & H_{54} & H_{55} & H_{56} \\ H_{60} & H_{61} & H_{62} & H_{63} & H_{64} & H_{65} & H_{66} \end{bmatrix}$$

- M is a 1 x w vector of 1 x w vector M_i, so it is a 1 x w^2 vector, but during the elaboration it is considered like a w x w matrix to handle it easier.

$$M = \begin{bmatrix} M_0 & M_1 & M_2 & M_3 & M_4 & M_5 & M_6 \end{bmatrix}$$

- As explained , the sum S of components of the output vector is equal to 2^{16}.

In the following pieces of code is used:

- WIENER_WIN to indicate a dimension of 7.
- WIENER_WIN2 to indicate a dimension of 49.
- WIENER_HALFWIN1 to indicate the middle position component of a 7-element vector, the fourth one.
- WIENER_FILT_SCALE to indicate a scaling factor of 2^7
- WIENER_TAP_SCALE_FACTOR to indicate 2^{16} value. The name clarify the reason why this value has been used: it represents a scaling factor necessary to avoid to work with high parallelism data.
- NUM_WIENER_ITERS to indicate how many times the Wiener filtering process is called and it is equal to 5;

2.3.1 Pre-processing stage

The pre-processing stage is used to evaluate both H and M matrices, and it is done by the function *wiener_decompose_sep_sym*, which code is reported below:

```
static int wiener_decompose_sep_sym(int wiener_win, int64_t *M, int64_t
    *H, int32_t *a, int32_t *b) {
        int k=0;

  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,
  };
  int64_t *Hc[WIENER_WIN2];
  int64_t *Mc[WIENER_WIN];
  int i, j, iter;
  const int plane_off = (WIENER_WIN - wiener_win) >> 1;
  const int wiener_win2 = wiener_win * wiener_win;
  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);
    update_b_sep_sym(wiener_win, Mc, Hc, a, b);
    iter++;
  }

  return 1;
}
```

Both H and M and initial value for filters a and b are derived by a constant vector *init_filt*. For each iteration, the *Update a* process occurs by calling *update_a_sep_sym*function and providing all the needed inputs, that are H,M and b_{in}; once finished, it returns the updated a filter value, used then by the function *update_b_sep_sym* to obtain in the same way the updated b filter. The entire

process will be repeated until the limit value of the number of iterations equal to 5 is reached.

2.3.2 *update_a_sep_sym* function

The function is defined as follows:

```
static AOM_INLINE void update_a_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;
```

- B is a 4x4 matrix and each (i, j) element is derived as:

$$B = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} H_{ij} \cdot b(i) \cdot b(j) \tag{2.7}$$

- A is a 1x4 vector and each $i - th$ element is derived as:

$$A = \sum_{i=0}^{w-1} M_i \cdot b(i) \tag{2.8}$$

The whole computation is reported in the following extract of C model:

```
  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;

        } } }  }
```

WIENER_TAP_SCALE_FACTOR is used to to avoid to reach a large parallelism because of the multiplications without increasing the complexity, since it is just a logic shifting: whenever a and b vector are used as operand in any complex operation, their components are properly scaled. This will obviously involve an approximation since the codec will never consider floating point number, so each fractional part is removed in scaling.

wrap_index function is used to index input matrices and select for each cycle the correct component for both A and B . It is done such that the operation of multiplication results completely symmetric. Let's consider, for example, the first 7x7 sub-matrix H_{00}: each element is multiplied step by step by b filter components and B matrix will be obtained by adding each single contribution symmetrically. The process is based on the idea to obtain the output filter by solving a linear system of equation in which:

- Matrix B is the coefficients matrix
- Vector A is the column-matrix of constant terms

The system solution represents the output value of the filter a and since, as explained in paragraph 2.2 only $r = 3$ filtered taps have to be sent, it is necessay to process these structures by an *enforcement* block to reduce the matrices size. It works as follows:

```
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)]);
    }
  }
```

It is possible to notice that the loop cycle terminates at $wiener\_halfwin1 - 1 = 3$, so the process involves only the 3 x 3 sub-matrix in the top left corner for B and the first 3 components for A. The system resolution is done by calling the function *linsolve_wiener*.

2.3.3 *linsolve_wiener* function

The function is declared as follows:

```
static int linsolve_wiener(int n, int64_t *A, int stride, int64_t *b,
    int32_t *x)
```

The B matrix is hereinafter called A and the A vector is called b. The parameter *stride* is equal to $wiener\_halfwin1 = 4$, while x is the 32-bit solution vector of 3 elements.

To solve the linear system, AV1 uses Gauss-elimination method instead of the direct method that would have involved to reverse the coefficient matrix and so its determinant. Obviously this solution would have required a much higher computational cost because each involved component is a 64-bit data and a dramatically high parallelism would have been reached. Moreover, this kind of implementation would not allow to scale properly input data. Problems related to the inversion operation will be better explained in 3.1.1.

The used Gauss-elimination method is based on the following steps:

1. creating the matrix [A|b];
2. applying Partial pivoting;
3. applying Forward Elimination;
4. applying Back-Substitution.

Partial pivoting is used to bring the element of a matrix with the largest pivot to the top: for pivot it is intended the first non-zero (and possibly far from it) element of a row . The technique used to reach this kind of configuration is called *pivoting* and in this case, since it swaps only the rows without involving the columns, it is defined *partial*. Then the Forward Elimination operation helps to produce a matrix in a *row echelon form*, that means that:

- If there are some rows containing only zeroes, they are at the bottom.
- The pivot of a non-zero row have to be to the right of the pivot of the row above it

It means that, in the proposed case of Wiener Filter, the matrix [A|b] is:

$$[A|b] = \begin{bmatrix} A_{00} & A_{01} & A_{02} & b_0 \\ A_{10} & A_{11} & A_{12} & b_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{bmatrix}$$

Being a 3x3 coefficient matrix, 2 steps are necessary to get it into an *upper triangular form*. A first stage of *Partial Pivoting* is applied to first column elements and the row with the largest between A_{00}, A_{10}, A_{20} is brought to the top. Then, by means of *elementary row operations* (for example multiplying by a constant or combining different rows adding an integer multiple of a row to another one) *Forward Elimination* is applied and the matrix is modified into:

$$[A|b] = \begin{bmatrix} A'_{00} & A'_{01} & A'_{02} & b'_0 \\ 0 & A_{11'} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{bmatrix}$$

A second step of *Partial Pivoting* is now applied on $A_{11'}$ and A'_{21}, swapping the correspondent row if $A'_{21} > A_{11'}$. Then, reusing *Forward Elimination* on the last two rows it is possible to replace the lower pivot with 0, obtaining the following structure where the pivots are A'_{00} , $A_{11''}$ and A''_{22}:

$$[A|b] = \left[\begin{array}{ccc|c} A'_{00} & A'_{01} & A'_{02} & b'_0 \\ 0 & A_{11''} & A''_{12} & b''_1 \\ 0 & 0 & A''_{22} & b''_2 \end{array}\right]$$

This algorithm is implemented as follows:

```
for (int k = 0; k < n - 1; k++) {
        //n=2 : 2 cycles of partial pivoting and forward elimination
    // Partial pivoting: bring the row with the largest pivot to the
       top
    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];
      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;
    }
  }
```

It is important to note that *Forward Elimination* algorithm produce neither an exact value of 0 before pivots nor an exact value of non-zero elements because obviously it uses an approximation to integer number, since the algorithm performs a logic scaling instead of a punctual division . A small error is introduced to the system but hereinafter it will be consider negligible. To solve the upper triangular system, is sufficient to use a simple *Back-substitution* method and then store the filter taps in x variable as follows:

```
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]);
         }}
```

To maintain data consistency, it is necessary that the updated version of filter a have the same 32-bit parallelism of its initial value because this vector will be used as input for the *Update b* process, this is why only the first 32 bit are taken from each element of x.

The last step is the reconstruction of x vector to 7 elements by applying a very simple operation of symmetry that is made by *update_a_sep_sym* as follows:

```
    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));
```

2.3.4 *update_b_sep_sym* function

This function works similarly to *update_a_sep_sym*. It is defined as:

```
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;
```

The only difference compared with *Update a* is the algorithm that produces B matrix and A vector:

```
 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;
        }
      }
    }}
```

As explained A and B are derived using the a vector previously updated. Even if they are derived in a different way, they maintain the same dimension they have in *Update a* process

$$B = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} H_{ij} \cdot a(i) \cdot a(j) \tag{2.9}$$

$$A = \sum_{i=0}^{w-1} M_i \cdot a(i) \tag{2.10}$$

The multiplication remains symmetric but the function *wrap_index* operates in a different way. For example, let's consider a 7x7 sub-matrix H_{ij} (same concepts can be obviously applied to a generic sub-vector M_i to generate A):

- In *Update a* process, each sub-matrix H_{ij} produces an entire matrix B, that is back-added to the previous B contribution given by sub-matrix $H_{i,j-1}$
- In *Update b* process, each sub-matrix H_{ij} produce just a single partial value P_{ij} stored in a 7 x 7 P matrix. Once filled, its components are added symmetrically to produce the 4x4 output matrix B.

The steps of *enforcement* and the solution of linear system of equation are exactly the same than the one analyzed in paragraph 2.3.2, since the same functions of *linsolve_wiener* (explained at paragraph 2.3.3) is used.

Chapter 3

A basic architecture of AV1 Wiener Filter

The idea of designing a hardware architecture starting from the reference specifications of AV1 is quite complex because it is necessary to take into account many constraints and considerations outside from software panorama: several parameters like area, speed, power consumption are involved and a good trade-off must be found in order to design a well-performing architecture. So this thesis presents the design of a special purpose architecture by knowing the basic algorithm and it means to focus particularly on other additional performance parameters (latency, throughput and so on) to obtain an architecture coherent with fixed design choices. The main advantage of a dedicated architecture is the possibility to reach very high time or energy efficiency because each design choice is properly customized. On the other hand, there is a flexibility deficit and the only way to make the architecture working is to use a fixed algorithm and adapt it to a system where to process the well-known input data.

The starting idea is to implement a basic VLSI implementation using VHDL language, without any kind of optimization. Indeed the first architecture is developed just like an Hardware translation of C model implementation, paying attention only to ensure its proper behavior and, once obtained a working architecture, it is analysed to evaluate its performances respect to a specific clock period T_{CK}. In particular, the following parameters are considered:

- *Critical Path* [T_{CP}], defined as the longest path delay. Namely it is the amount of time spent to propagate data along the longest path but T_{CP} is very important because it determines if the architecture could work or not, since have to be $T_{CP} < T_{CK}$. The critical path is the first parameter under exam since from it depend depend both the speed evaluation in terms of working frequency and the power consumption.

- *Time per data item* $[T]$, defined as the amount of time elapsed between two subsequent output data items.
- *Throughput* $[tp]$, defined as $1/T$ and represents the number of sample generated per second. It gives an idea of architecture speed.
- *Latency* $[L]$, defined as the number of clock cycle the architecture takes from a data entering into the circuit until it is released as output.
- *Area*$[A]$ gives an idea of system complexity and it is measured by considering the size and the number of each involved components such as ports, nets and nets.

3.1 Algorithm to Architecture Mapping

The main problem of the algorithm mapping is to deal with very large data size. For example, each H or M component is a 64-bit data, while the parallelism of initial vector a and b is 32 bit. Of course their elaboration with elementary operations, above all multiplications and divisions, would have meant reaching too high parallelism: this is why, following the C model implementation, each intermediate critical result has been truncated to 64 bit except from the 32-bit output filtered vector as mentioned at paragraph 2.3.3.

Another critical point is the scaling operation of negative number. Each time a data must be scaled, a shifter is employed. Let's consider, for example, to operate a 2^8 scaling using a 8-bit shifter, with an initial bit sequence of 011000011010100000 that corresponds to a decimal value of 100000.

Remembering that all fractional parts will be truncated,in positive case:

$$N_{shifted} = \frac{100000}{2^8} = 390,625... = 390 \tag{3.1}$$

Scaling bit sequence the same result is obtained:

$$011000011010100000_{bin} = 100000_{dec} \rightarrow 0110000110_{bin} = 390_{dec} \tag{3.2}$$

In this case the shifter works properly. What happens in negative case is that the final result is always approximated to lower number but, being negative, it is the the larger in magnitude:

$$N_{shifted} = \frac{-100000}{2^8} = -390,625... = -390 \tag{3.3}$$

Scaling bit sequence:

$$100111100101100000_{bin} = -100000_{dec} \rightarrow 1001111001_{bin} = -391_{dec} \tag{3.4}$$

A considerable error is introduced inside the process whenever a negative shifter is implemented. In order to avoid this problema still using shifters, the scaling is performed on the absolute value of the negative data, complementing it later again:

```
PROCESS(DATA_IN)
BEGIN
        IF (DATA_IN(63) = '1') THEN
                DATA_IN_ABS <= NOT (DATA_IN)+1;
        ELSE
                DATA_IN_ABS <= DATA_IN;
        END IF;
END PROCESS;

DATA_IN_SHIFTED_ABS <= shift_right(DATA_IN,8);

PROCESS(DATA_IN)
BEGIN
        IF (DATA_IN (63) ='1') THEN
                DATA_IN_SHIFTED <= NOT(DATA_IN_SHIFTED_ABS)+1;
        ELSE
                DATA_IN_SHIFTED <= DATA_IN_SHIFTED_ABS;
        END IF;
END PROCESS;
```

3.1.1 Hardware design implementation of *Update a*

The aim of *Update a* is to map the algorithms of *update_a_sep_sym* and *linsolve_wiener* functions illustrated in the Chapter 2: it processes as input the matrices H and M and the input fixed vector b_{in} and it computes the update version of filter a. The evolution of the whole algorithm is handled by using two counters for every block where they are needed that select properly, at each step, which component of H, M or b has to be processed. The *counter enable* is provided by an FSM that controls the correct flowing of algorithm flows.

Computation of vector A The component designed to derive A vector maps the following instructions:

```
  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;
    }
  }
```

It receives as inputs, for each cycle:

- M_{in}, a 7 elements sub-vector of M correspondent to the i-th cycle properly selected by the external block "*M selection*" such that:

$$M_{in} = \begin{bmatrix} M_0 & M_1 & M_2 & M_3 & M_4 & M_5 & M_6 \end{bmatrix}$$

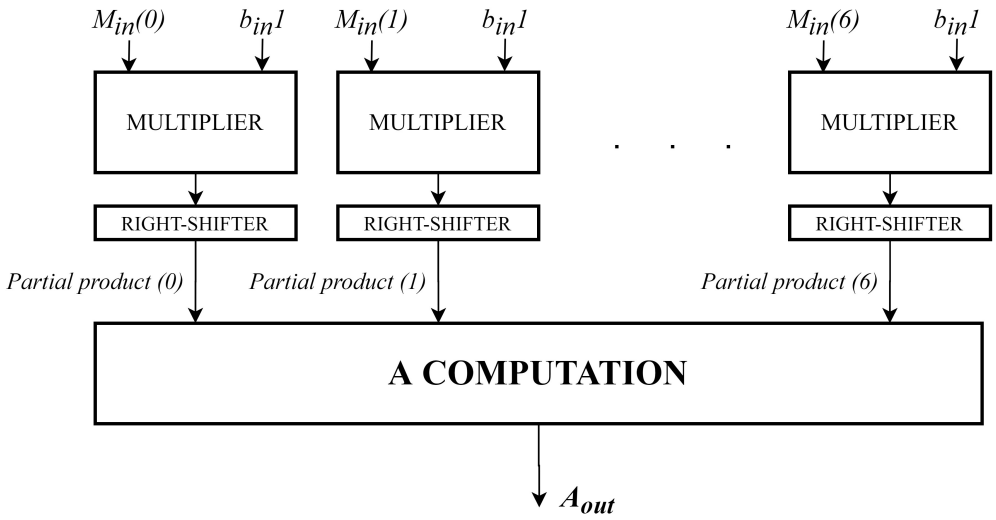


Figure 3.1: M_{ij} x b_i data path

where each M_x is a 64-bit element.

- $b_{in}1$, the i-th element of initial vector b_{in}.

The block, called "$\boldsymbol{M_{ij}}$ **x** $\boldsymbol{b_i}$", generates a vector of partial products and then computes the output vector A_{out} by processing it as reported in the data path in figure 3.1

The behavior of *A computation* block is described below:

```
FOR I IN 0 TO 2 LOOP
A_OUT(I)<=PP(I)(63 DOWNTO 0) + PP(6-I)(63 DOWNTO 0);
END LOOP;
A_OUT(3) <= PP(3) (63 DOWNTO 0);
```

Computation of matrix B This component is quite similar to the previous one and it is used to implement the following instructions:

```
  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;

        } } }  }
```

It receives for each cycle:

- H_{in}, the 7x7 sub-matrix of H correspondent to (i, j) cycle properly selected from the external block "*H Selection*" such that

$$H_{in} = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} & H_{04} & H_{05} & H_{06} \\ H_{10} & H_{11} & H_{12} & H_{13} & H_{14} & H_{15} & H_{16} \\ H_{20} & H_{21} & H_{22} & H_{23} & H_{24} & H_{25} & H_{26} \\ H_{30} & H_{31} & H_{32} & H_{33} & H_{34} & H_{35} & H_{36} \\ H_{40} & H_{41} & H_{42} & H_{43} & H_{44} & H_{45} & H_{46} \\ H_{50} & H_{51} & H_{52} & H_{53} & H_{54} & H_{55} & H_{56} \\ H_{60} & H_{61} & H_{62} & H_{63} & H_{64} & H_{65} & H_{66} \end{bmatrix}$$

  where each H_{xx} is a 64-bit element.

- $b_{in}1$, the i-th element of initial vector b.

- $b_{in}2$, the j-th element of initial vector b.

This block called "$\boldsymbol{H_{ij}}$ **x** $\boldsymbol{b_i}$ **x** $\boldsymbol{b_j}$" provides as output a 4x4 matrix B_{out} related to (i, j) step. Its datapath is represented at figure 3.2 and it is described by the following VHDL code where the absolute value done to implement properly the shifting operation has not been reported:

```
PROCESS(H_IN, BIN1, BIN2,)
  BEGIN
FOR I IN 0 TO 6 LOOP
        FOR J IN 0 TO 6 LOOP
PP(I,J)<=(shift_right(((shift_right((H_IN(J,I)*BIN1),16)(63 DOWNTO 0))*BIN2),16));
END LOOP;
END LOOP;
END PROCESS;
B0<=PP(0,0)+PP(0,6) +PP(6,0) + PP(6,6) ;
B4<=PP(0,1) +PP(0,5) +PP(6,1) + PP(6,5) ;
B8<=PP(0,2) +PP(0,4) +PP(6,2) + PP(6,4) ;
B12<=PP(0,3)  + PP(6,3) ;

B1<=PP(1,0) +PP(1,6) +PP(5,0) + PP(5,6) ;
B5<=PP(1,1) +PP(1,5) +PP(5,1) + PP(5,5) ;
B9<=PP(1,2) +PP(1,4) +PP(5,2) + PP(5,4) ;
B13<=PP(1,3)  + PP(5,3) ;

B2<=PP(2,0) +PP(2,6) +PP(4,0) + PP(4,6) ;
B6<=PP(2,1) +PP(2,5) +PP(4,1) + PP(4,5) ;
B10<=PP(2,2) +PP(2,4) +PP(4,2) + PP(4,4) ;
B14<=PP(2,3)  + PP(4,3) ;

B3<=PP(3,0)  + PP(3,6) ;
B7<=PP(3,1)  + PP(3,5) ;
B11<=PP(3,2)  + PP(3,4) ;
B15<=PP(3,3) ;
```

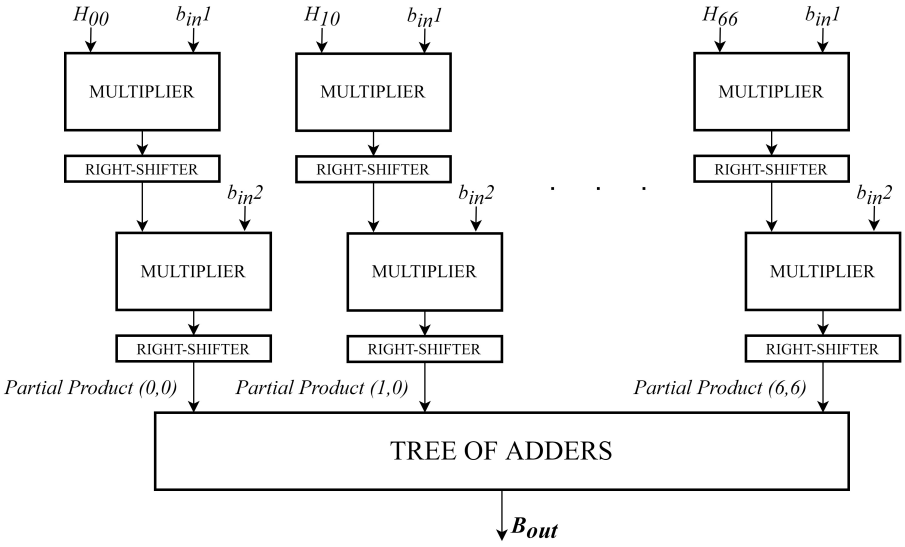


Figure 3.2: H_{ij} x b_i x b_j data path

In order to process the proper component of the matrix H_{in} other two internal indices have been employed. Calling them z and k, the block generates, performing two steps of multiplication and shift, a 7x7 matrix of partial product such that its (z, k) element is derived by processing the (z, k) element of H_{in}. Then, a tree of adders is implemented to compress the matrix dimension in a symmetric way like reported in the VHDL code above.

Enforcement The enforcement block acts to compress the input matrices adapting them to the linear system of equation. It receives as inputs:

- The 4x4 B matrix:
$$B = \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$
- The 1x4 A vector:
$$A = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \end{bmatrix}$$

The algorithm described at paragraph 2.3.2 is mapped to Hardware using the following VHDL code:

```
PROCESS(A_IN, B_IN, ARGUMENT)
BEGIN

FOR k IN 0 TO 2 LOOP
A_ENF(k)<=A_IN(k)-shift_left(A_IN(3),1) - B_IN(k,3) + shift_left(B_IN(3,3),1);
END LOOP;

FOR k IN 0 TO 2 LOOP
        FOR z IN 0 TO 2 LOOP
                ARGUMENT(k,z)<=B_IN(k,3) + B_IN(3,z) - shift_left(B_IN(3,3),1);
                B_ENF(k,z)<=B_IN(k,z) - (shift_left(ARGUMENT(k,z),1));
        END LOOP;
END LOOP;

END PROCESS;
```

The implementation can be divided in two parts: the first involves A , the other B. The output vector $A_{enforced}$ is computed as represented at figure 3.3, producing the 1x3 output vector:

$$A = \begin{bmatrix} A'_0 & A'_1 & A'_2 \end{bmatrix}$$

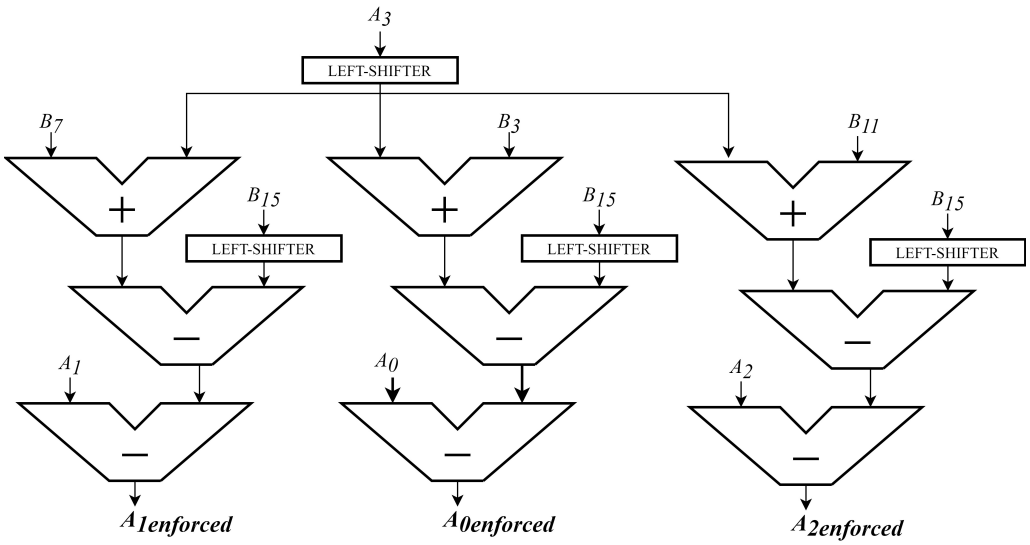


Figure 3.3: A enforcement data path

The elaboration of B matrix is very similar but, for reason of space, the figure 3.4 reports only the first and the last part of the implementation:

The output is the 3 x 3 matrix:

$$B = \begin{bmatrix} B'_0 & B'_1 & B'_2 \\ B'_4 & B'_5 & B'_6 \\ B'_8 & B'_9 & B'_{10} \end{bmatrix}$$

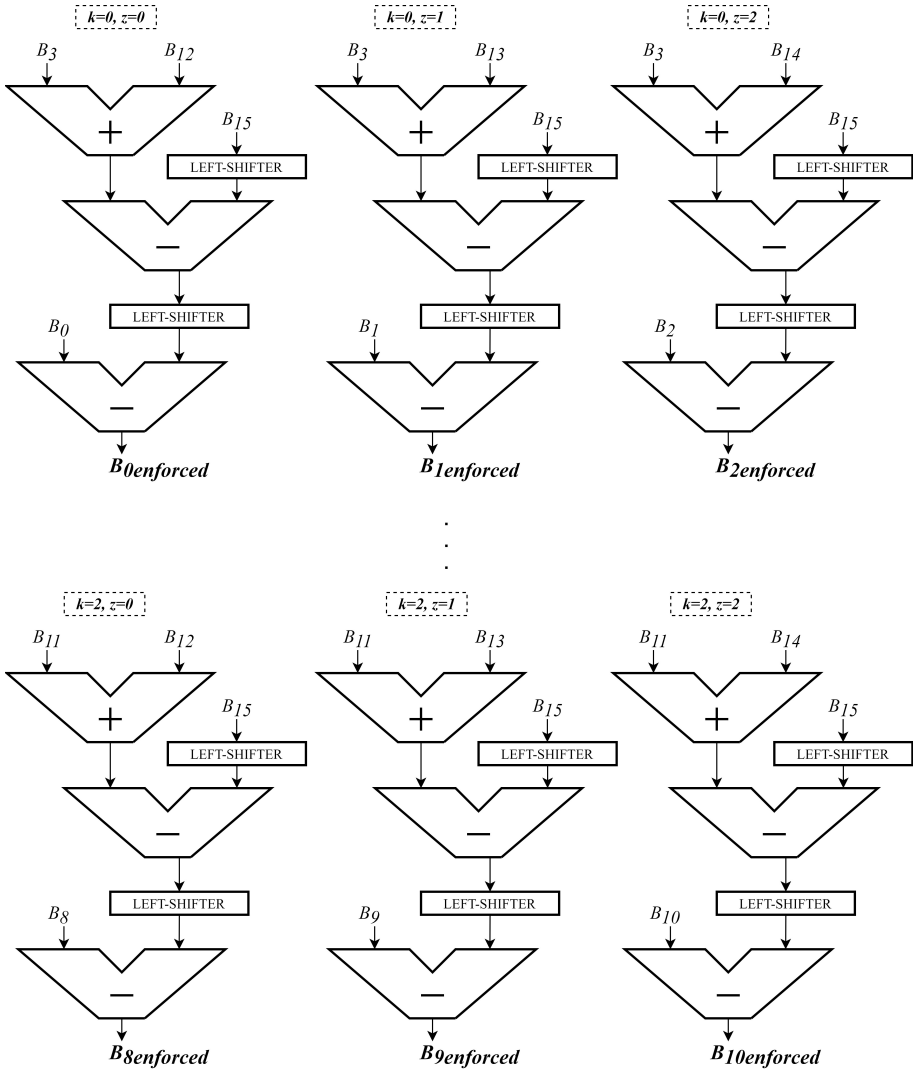


Figure 3.4: B enforcement data path

To follow the C model terminology and as mentioned in 2.3.3, hereinafter B matrix will be called A and A vector will be called b. After the enforcement block both coefficient matrices and known terms vector have been obtained so it is possible to introduce the design choices for the resolution of the linear system

$$A \cdot a = b \tag{3.5}$$

where the a vector solution is the output vertical filter.

System resolution: inverse method for linear system The first attempt was to implement a solution using the *inverse method for linear system*. Starting

from the system

$$\begin{cases} A_0 \cdot a_0 + A_1 \cdot a_1 + A_2 \cdot a_2 = b_0 \\ A_4 \cdot a_0 + A_5 \cdot a_1 + A_6 \cdot a_2 = b_1 \\ A_8 \cdot a_0 + A_9 \cdot a_1 + A_{10} \cdot a_2 = b_2 \end{cases}$$

the solution can be expressed as:

$$a = b \cdot A^{-1} \tag{3.6}$$

The main advantage is to reduce the solution to a row-column product, paying the cost of the inversion operation that include the computation of the *Cofactor* matrix whose each element is:

$$Cof\{A_{ij}\} = (-1)^{i+j} \cdot M_{ij} \tag{3.7}$$

where M_{ij} is the *minor* related to A_{ij} that is the *determinant* of the smaller matrix obtained cutting down from A the i-th row and j-th column. From Cofactor matrix, the *adjugate* matrix [ADJ] is computed by transposing it and the inverse matrix is equal to:

$$A^{-1} = \frac{ADJ}{DET} \tag{3.8}$$

where DET is the determinant of A matrix.

In this special purpose architecture, the matrix inversion represents a big trouble for two reasons:

- A_{ij} is a 64 bit data, so the calculation of the adjugate matrix and the determinant requires two and three multiplication respectively. This means that ADJ elements have a parallelism of 128 bit and determinant reaches even 196 bit. The implementation of a 196-bit division is unsustainable not only for its huge computational cost, but also because it would require a higher number of consecutive combinatorial block in the critical path, worsening the performances in terms of speed.

- DET is always bigger than each elements of ADJ because it is obtained from one more multiplication. Their ratio always belongs to range [0-1] and, since fractional part is always truncated, the result would be just a matrix of zeroes.

A possible solution might be to use a different numerical representation, for example floating-point or fixed-point, but it would have changed all the algorithm behavior presented in previous chapter making the hardware mapping impossible. For these reasons , following the C model approach, the linear system has been solved using the Gauss-elimination algorithm described at paragraph 2.3.3 and the following paragraphs focus on the design of *Partial pivoting*, *Forward elimination* and *Back − substitution and storing* based on their C model behavior.

Partial Pivoting From a computational point of view, this is the simplest block inside the whole architecture because it consists only in a swap rows operation. A unique block has been implemented for both processing stage: each step is individuated by a value of the parameter k, so $k = 0$ identifies the first step while $k = 1$ the second one. The architecture at figure 3.5 is described by the following VHDL extract:

```
PROCESS(A,b,k, ABS_A0,ABS_A4,ABS_A5,ABS_A8,ABS_A9)
BEGIN
IF(k='0') THEN --k=0 : first computational stage
            IF(ABS_A4<ABS_A8) THEN
                  IF(ABS_A0<ABS_A8) THEN
OUT_MATRIX<=((A(2,0),A(2,1),A(2,2)),(A(0,0),A(0,1), A(0,2)), (A(1,0),A(1,1),A(1,2))
    );
OUT_VECTOR<=(b(2), b(0), b(1));
                ELSE
OUT_MATRIX<=((A(0,0), A(0,1), A(0,2)),(A(2,0),A(2,1),A(2,2)), (A(1,0),A(1,1),A(1,2)
    ));
OUT_VECTOR<=(b(0), b(2), b(1));
            END IF;

                ELSIF (ABS_A0<ABS_A4) THEN

OUT_MATRIX<=((A(1,0),A(1,1),A(1,2)),(A(0,0),A(0,1), A(0,2)), (A(2,0),A(2,1),A(2,2))
    );
OUT_VECTOR<=(b(1), b(0), b(2));
       ELSE

OUT_MATRIX<=A;
OUT_VECTOR<=b;
       END IF;

ELSE --k=1:second computational stage

         IF(ABS_A5<ABS_A9) THEN
OUT_MATRIX<=((A(0,0), A(0,1), A(0,2)),(A(2,0),A(2,1),A(2,2)), (A(1,0),A(1,1),A(1,2)
    ));
OUT_VECTOR<=(b(0), b(2), b(1));
     ELSE
OUT_MATRIX<=A;
OUT_VECTOR<=b;
         END IF;
END IF;

END PROCESS;
```

- In the first step, the absolute values of A_0, A_4 and A_8 are compared two by two to find the largest: *Swap Rows* block makes the first row the one with the largest pivot and eventually change the position of b elements. The output matrix A_{pivot} and the output vector b_{pivot} are ready for the *Forward Elimination* step.

- In the second stage, the inputs come from the first stage of *forward elimination*. What remains to compare is the absolute value of pivots of second and

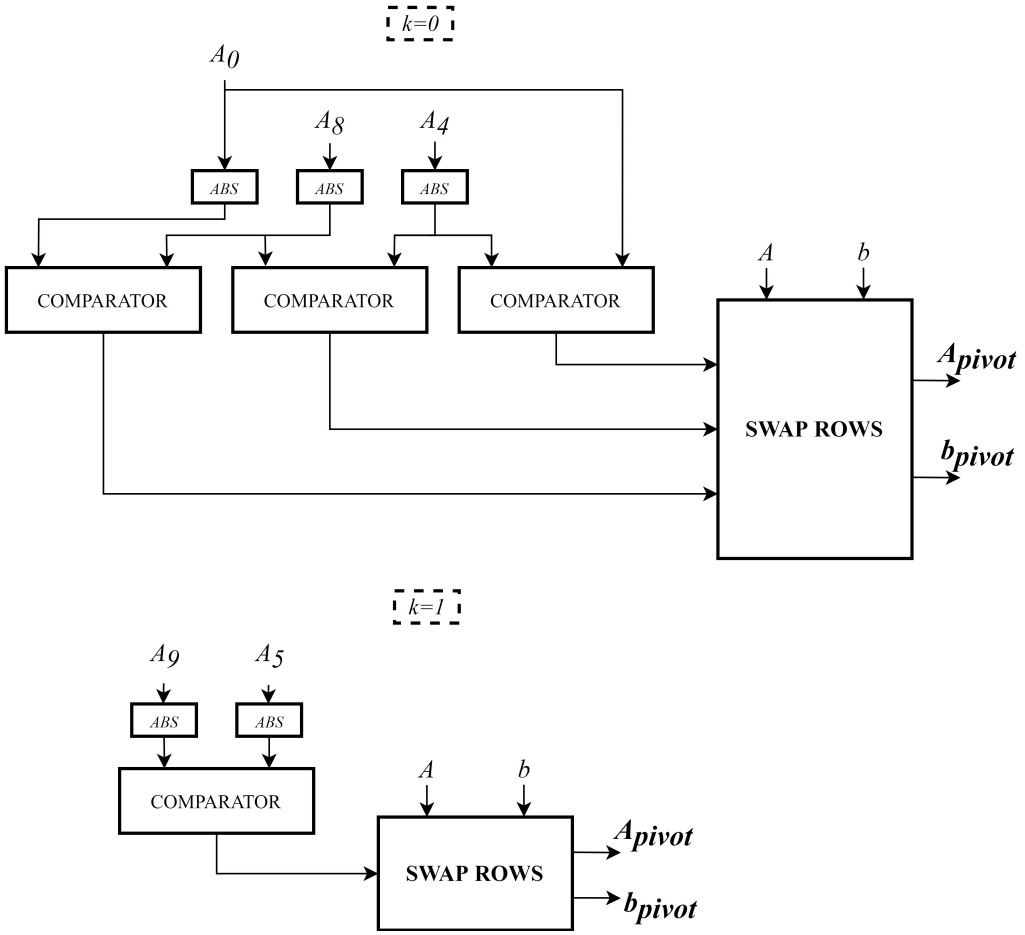


Figure 3.5: Partial Pivoting Data Path

third rows, A_5 and A_9. If $A_9 > A_5$ the two relative rows and the relative b components are swapped. Otherwise, both input structures are released as outputs and ready for the second stage of *Forward elimination*.

Forward elimination This is the mathematical step of the linear system resolution: it performs multiplications, divisions and subtractions to combine properly two rows and, finally, transform the matrix as closest as possible to an upper triangular form to be able to find the solution of the system. Similarly to Partial Pivoting, a unique block has been implemented for both processing stages. It is necessary to distinguish how each stage works since different elements of input structure are involved.

First stage The first stage occurs after the first Partial pivoting operation, so it is ensured that the first rows contains the largest pivot. This allows to involve only the second and the third rows of A matrix in the computation. At the same time, the meaning of the system must be unaltered and this means that also the second and the third elements of b vector must be processed exactly in the same way. To derive each element of the output matrix A_{FE} the following VHDL code has been implemented (only $A_{4_{FE}}$ is reported as an example)and the related architecture is reported at figure 3.6.

```
--Code is referring to A4_FE computation but operations
--    are the same for all the others components
DIVIDEND <= shift_right(A4,8) * A0;
DIV      : DIVISION PORT MAP (DIVISOR=>A0,
                              DIVIDEND=>DIVIDEND(63 DOWNTO 0),
                              QUOTIENT=>QUOTIENT);
A4_FE <= A4- shift_left(QUOTIENT,8);
```

The same approach is used also to modify the b vector in order to find b_{FE}: the VHDL description is reported below and its architecture is at figure 3.7.

```
--Code is referring to b1_FE computation but operations
--                       are the same for b2_FE
B_DIVIDEND <= A4*b0;
B_DIV : DIVISION PORT MAP (DIVISOR=>A0,
                                                DIVIDEND=>B_DIVIDEND(63 DOWNTO 0)
                                                ,
                                                QUOTIENT=>QUOTIENT);
b1_FE<=b1 - QUOTIENT;
```

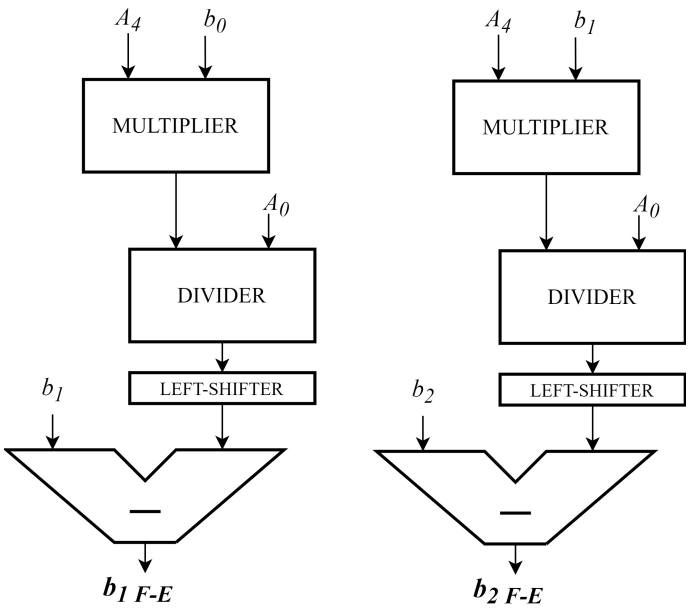


Figure 3.7: First stage of Forward Elimination:Data Path for b_{FE}

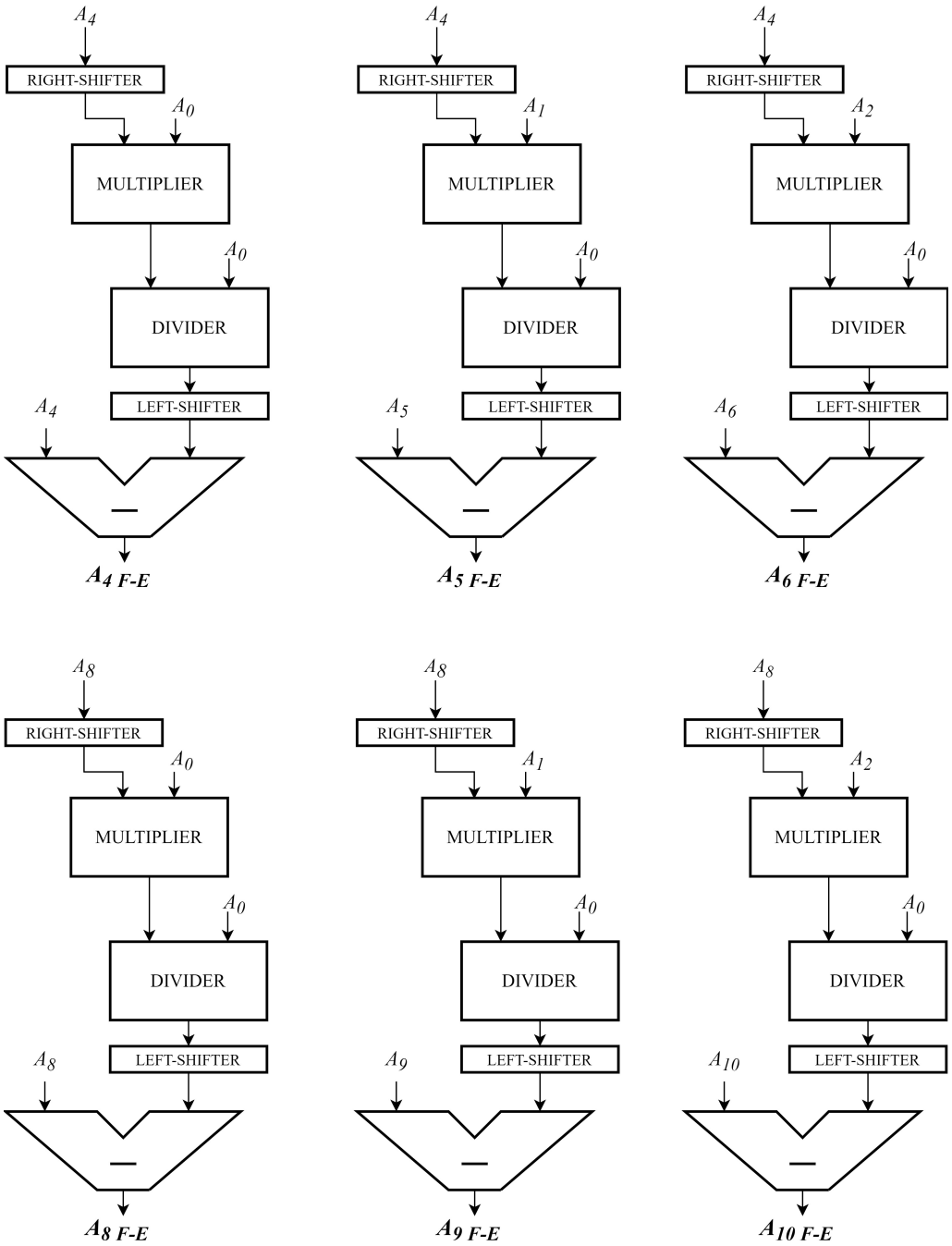


Figure 3.6: First stage of Forward Elimination: Data Path for A_{FE}

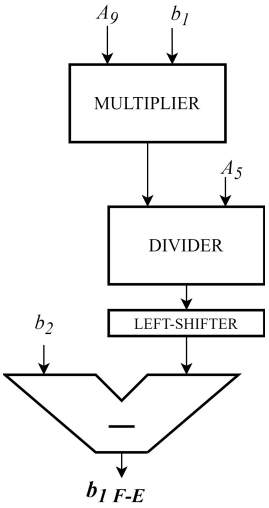


Figure 3.8: Second stage of Forward Elimination: Data Path for b_{FE}

Second stage The second stage Forward Elimination is very similar to the first one in terms of behavior. The relevant difference is, being after the last stage of Partial pivoting, the matrix and the vector are already ordered so it works only on the last element of b vector and the last A matrix row to reach the upper triangular form. The sequence of operations for both structures is exactly the same to first stage, so they have been described by the same VHDL extract and their architectures are reported at figure 3.8 and 3.9 respectively.

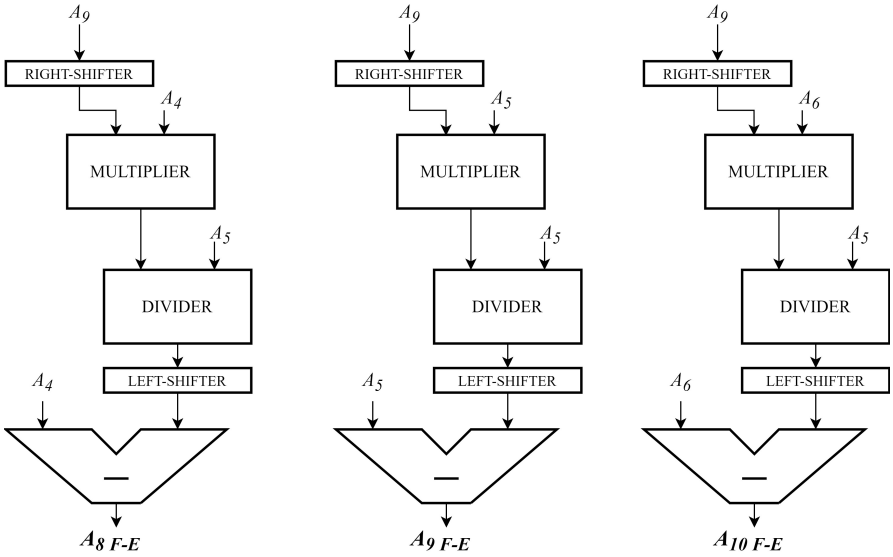


Figure 3.9: Second stage of Forward Elimination: Data Path for A_{FE}

Divider implementation Before going on, a special mention has to be done for division operation because it is a big critical point due to its complex integration. A basic divider could be an huge and slow operator, the worst to implement a good design. In particular, for this 64-bit case, a bad designed divider could be dramatic because the algorithm involves several divisions. As mentioned at the beginning of the current chapter, the aim is to start building a no-optimized architecture to analyse its critical points and, finally, apply a proper optimization. Even knowing that the divider implementation will be definitively the first point to consider to build a more efficient design, in this basic architecture a simple function based on the *Restoring Division Algorithm* is used. It performs an unsigned division between two 64-bit numbers A and B by doing a 64-bit addition and a 64-bit subtraction at each step: after all, each division will cost 128 operations and the integration of 128 64-bit adder. The function has been reported in VHDL using the *variable* construction.

```
FUNCTION DIVISION (A : UNSIGNED; B : UNSIGNED) RETURN UNSIGNED IS
VARIABLE A1 : UNSIGNED(63 DOWNTO 0):=A;
VARIABLE B1 : UNSIGNED(63 DOWNTO 0):=B;
VARIABLE P1 : UNSIGNED(63 DOWNTO 0):= (OTHERS => '0');
VARIABLE I : INTEGER:=0;

BEGIN
FOR I IN 0 TO 63 LOOP
P1(63 DOWNTO 1) := P1(62 DOWNTO 0);
P1(0) := A1(63);
A1(63 DOWNTO 1) := A1(62 DOWNTO 0);
P1 := P1-B1;
IF(P1(63) ='1') THEN
A1(0) :='0';
P1 := P1+B1;
ELSE
A1(0) :='1';
END IF;
END LOOP;
RETURN A1;

END DIVISION;
```

Back-substitution and storing Once obtained an upper triangular linear system in a row echelon form, the last remaining operation is to solve it. This block aims to find the 3-taps filtered vector $x = [x_0, x_1, x_2]$ by solving the following system:

$$\begin{cases} A_0 \cdot x_0 + A_1 \cdot x_1 + A_2 \cdot x_2 = b_0 \\ A_5 \cdot x_1 + A_6 \cdot x_2 = b_1 \\ A_{10} \cdot x_2 = b_2 \end{cases}$$

This is not the final output of the filter, since it is necessary to make a symmetry to obtain a 7-taps vector: this operation is done by an external block. The designed architecture have to map the following algorithm:

```
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]);
         }}
```

From a computational point of view, this block is really complex since:

- it involves several expensive operators like dividers and multipliers. Their integration requires an increment of the complexity;
- the output is the results of lots of subsequent operations along the critical path and this widely affects the speed performances.

The design choice was to implement a purely combinatorial block even if it is not a good choice in terms of data flow control and output results checking: the simulation depends on many external factors like the simulator or the timing constrains. Despite that the choice to avoid to break the critical path employing registers is done to ensure that final analysis of the performances will be made considering a complete no-optimized version of the architecture, in order to be able to better understand the critical points and optimize them. The related architecture is reported at figure 3.10. An extract of the VHDL description is reported below. For each instruction has been reported to which (i, j) cycle it occurs:

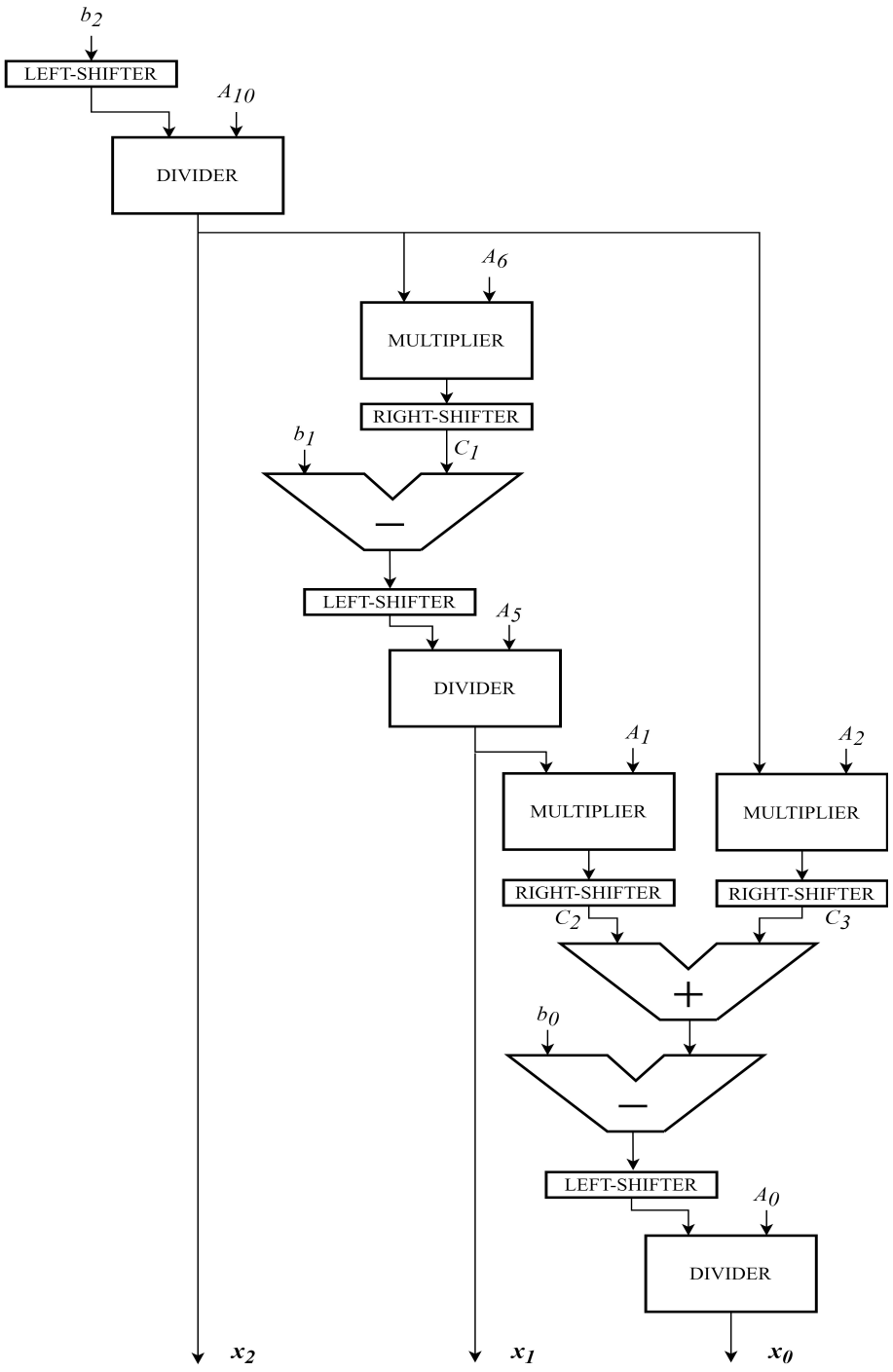


Figure 3.10: Combinational Data Path for Back-Substitution

```
--I=2
DIVIDEND_1<=shift_left(b(2),16);
B2_A10_DIV : DIVISION PORT MAP (DIVISOR=>A(2,2),
                                DIVIDEND=>DIVIDEND_1,
                                QUOTIENT=>X2);
--I=1
        --J=2
PRODUCT_1<=X2*A(1,2);

C1<=shift_right(PRODUCT_1,16);
DIVIDEND_2<=shift_left(b(1)-C1(63 DOWNTO 0),16);
B1_A5_DIV : DIVISION PORT MAP (DIVISOR=>A(1,1),
                                DIVIDEND=>DIVIDEND_2,
                                QUOTIENT=>X1);
--I=0
        --J=1
PRODUCT_2<=X1*A(0,1);
C2<=shift_right(PRODUCT_2,16);
PRODUCT_3<=X2*A(0,2);
C3_SHIFT<=shift_right(PRODUCT_3,16);
C3<=C_2+C3_SHIFT;
DIVIDEND_3<=shift_left((b(0)-C_3(63 DOWNTO 0)),16);
B0_A0_DIV : DIVISION PORT MAP (DIVISOR=>A(0,0),
                                DIVIDEND=>DIVIDEND_3,
                                QUOTIENT=>X0);

OUT_VECTOR<=(X0(31 DOWNTO 0), X1(31 DOWNTO 0), X2(31 DOWNTO 0));
```

Only the first 32 bit of x_0, x_1 and x_2 are taken to maintain data coherence with the following filtering operation.

Top-level architecture The top-level architecture is reported at figure 3.11. It is necessary to make a premise regarding the multiplicative blocks H_{ij} x b_i x b_j and M_{ij} x b_i. As mentioned, each of them receives as input the proper sub-structures correspondent to the (i, j) working cycle from external selection components, but in two different way:

- For M_{ij} x b_i an *M Selection* block is implemented such that it processes the whole 7x7 M matrix releasing as output the 1x7 sub-vector using only the j counter: since for each cycle an entire row has to be selected, just one counter is needed, this means that the whole selection occurs when $i = 0$.So, for the first seven cycle, where $i = 0$ and j is incremented by the counter from 0 to 6, M_{ij} corresponds to the j-th row of the input matrix M. As a consequence,M_{ij} x b_i terminates its process in 7 clock cycles. *M Selection* has been integrated inside the top-level architecture due to the reasonable dimension of the involved input.

- The block H_{ij} x b_i x b_j has been indexed in a different way because, because in this case both counters are needed to properly select the wanted sub-matrix H_{ij}. It was not possible to integrate the *H selection* inside the architecture

because of the huge dimension of input H matrix. The selection from a 49x49 matrix of 64-bit elements would have been too expensive during the synthesis. This is why sub-matrix H_{ij} is provided directly implementing the proper selection in the testbench avoiding in this way its synthesis.

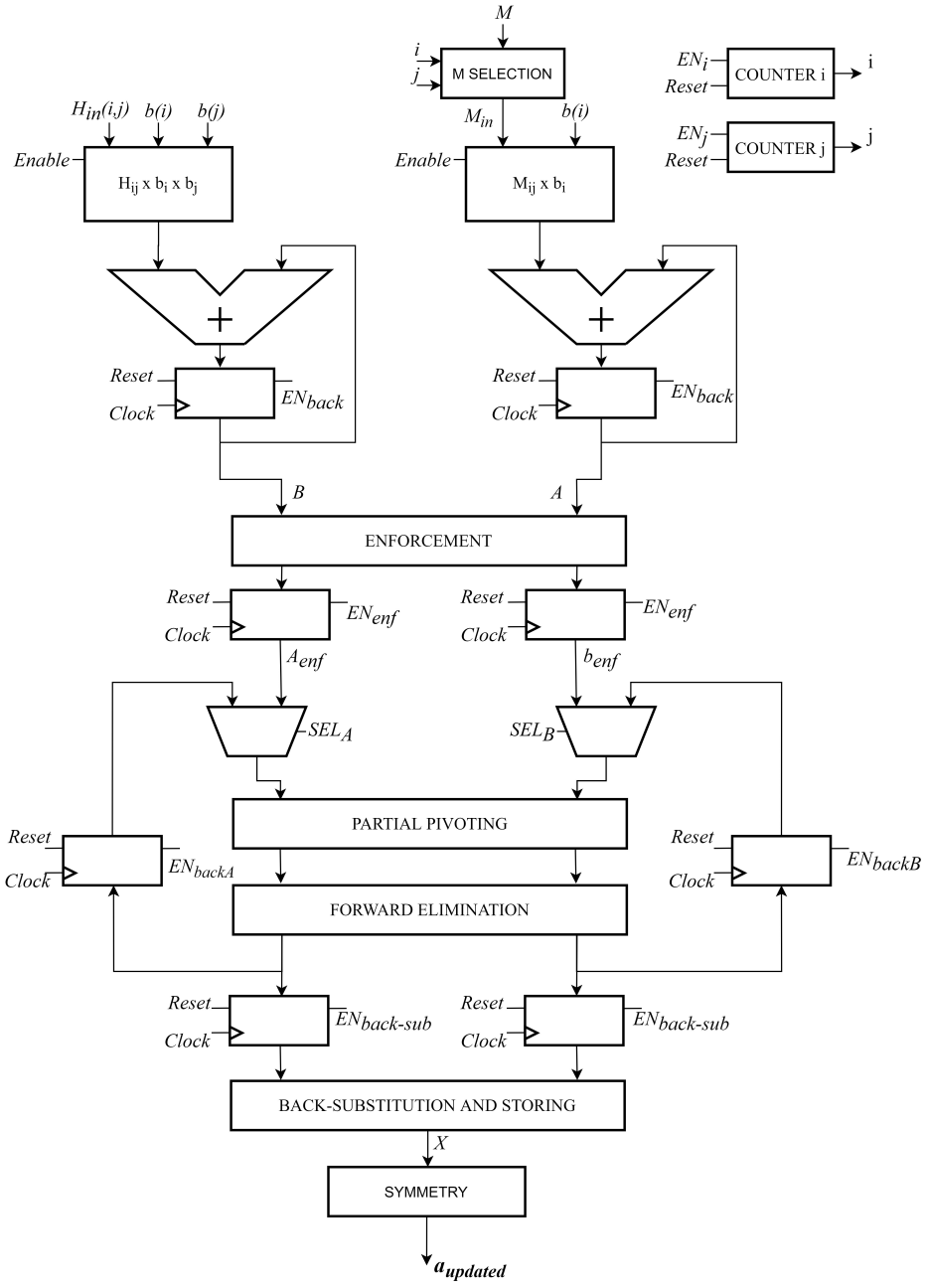


Figure 3.11: Top-level FSM for *Update a*

To keep under control the instructions flow each signal is handled by using the FSM described at figure 3.12. For example, the multiplicative blocks are enabled only when they have to perform some kind of operations to be sure they work with correct data input.

At the beginning, when the reset signal is released, both i and j are equal to 0. In the first seven clock cycles where only j is incremented [**Enable Counter j** state], H_{ij} x b_i x b_j and M_{ij} x b_i work in parallel. In particular, for each $(0, j)$ cycle they generate a partial B matrix and A vector respectively that are added using a feedback register to the previous contribution computed at cycle $(0, j-1)$. When $j = 6$, M_{ij} x b_i ends its process, so the related feedback register is disabled [**Disable Ret Register** state] and have to wait the H_{ij} x b_i x b_j completion that lasts other 42 clock cycles in which every partial B matrix produced at cycle (i, j) continue to be back-added to previous contribution of cycle $(i, j-1)$. Once obtained the final value of matrix B, one clock cycle is spent to wait that the last [$(6, 6)$] partial matrix B passes through the feedback register [**Wait Last B Storing** state]. A and B are processed as discussed by the *Enforcement* block [**Enable Enforcement** state]. To solve the system, two processing step of *Partial Pivoting* and *Forward-Elimination* are computed . Each stage is selected using a 2-to-1 multiplexer such that:

- In **k=0: first processing stage** state its output are the matrices coming from *Enforcement* block;
- In **k=1: first processing stage** state its output are the matrices coming from textitForward-Elimination block by means of a feedback register;

.

The upper triangular system is then solved by *Back-Substitution and Storing* at **Enable Update_B** state. The input register of this block are enabled only in this state to ensure to not have overusing by processing useless intermediate structures. Finally, the following symmetry is employed to the 3-tap vector X to obtain the final output vector $\boldsymbol{a_{updated}}$, the vertical filter derived by applying Wiener Filter to input pixel:

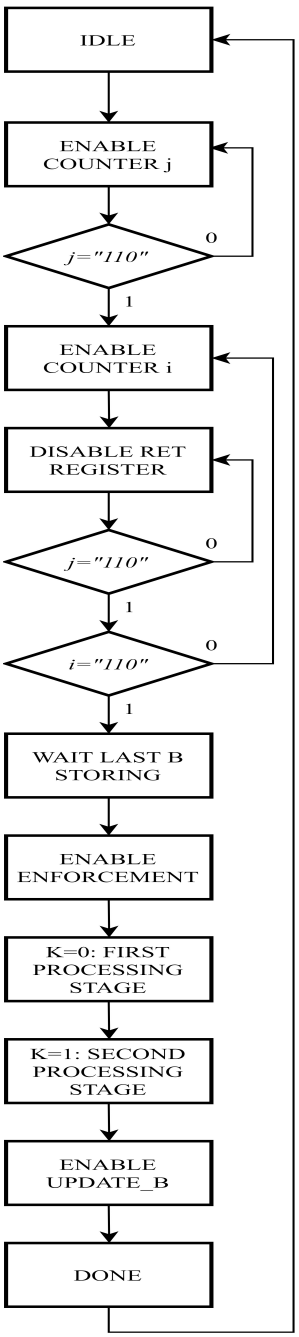


Figure 3.12: Top-level FSM for *Update a*

```
A(0)<=X(0);
A(1)<=X(1);
A(2)<=X(2);
A(4)<=X(2);
A(5)<=X(1);
A(6)<=X(0);
SHIFT_IN <=X(0)+X(1)+X(2);
A(3)<= shift_left(TO_SIGNED(1, 32),16) --2^16 =WIENER_TAP_SCALE_FACTOR
        - shift_left(SHIFT_IN,1);
```

3.1.2 Hardware design implementation of *update b*

The architecture designed to derive the output horizontal filter is quite similar to the vertical one since the same *linsolve_wiener* function is used for both of them. What changes is the computation of B matrix and A vector that consists in the mapping of *update_b_sep_sym* function and leads to a quite different top-level implementation.

This component receives as inputs the matrices H and M and a_{in}, the updated version of the vertical filter vector that will be kept fixed during all the process.

Computation of vector A Differently from the block M_{ij} x b_i, this component $\boldsymbol{M_{ij}}$ **x** $\boldsymbol{a_i}$ produces a single output value A_{out} correspondent to each iteration (i, j) that will be back-added to the contribution related to the previous cycle. The algorithm to map is the following:

```
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;
        }
}
```

The inputs are:

- M_{in}, a 7 elements sub-vector of M correspondent to the i-th cycle properly selected again by the external block "*M selection*" such that:

$$M_{in} = \begin{bmatrix} M_0 & M_1 & M_2 & M_3 & M_4 & M_5 & M_6 \end{bmatrix}$$

  where each M_x is a 64-bit element.

- a_{in}, the updated vertical filter.

The j-th step is described by the architecture at figure 3.13 where the *Index Selection* block picks the j-th component of both vector $M_{in}$and a_{in} .

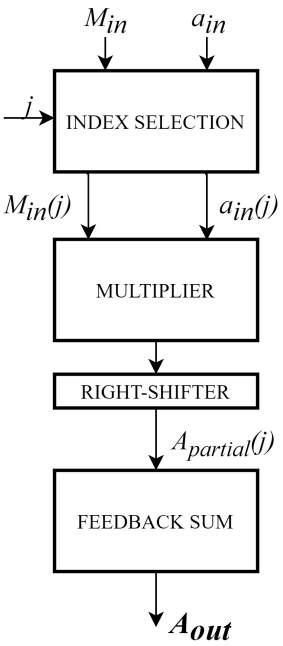


Figure 3.13: M_{ij} x a_i data path

The VHDL *variable* construct has been preferred to the "adder and feedback register" to perform the feedback sum because the register would have required a internal reset signal when the last element of M_{in}, the i-th row of the M matrix, is processed; otherwise the process related to the first element of the $i+1$ row vector would have had an offset equal to $A_{partial}$: it has been chosen to follow a design implementation in which only the top-level reset is considered and so, even if variables represent a kind of approach very close to software world, in this specific application they are used because they could be reset very easily:

```
PROCESS(M_IN,A)
VARIABLE A_TEMP : SIGNED(95 DOWNTO 0):=TO_SIGNED(0,96);
VARIABLE SUM_A: SIGNED(63 DOWNTO 0):=TO_SIGNED(0,64);
BEGIN
SUM_A:=TO_SIGNED(0,64); -- Reset of variable SUM_A
A_TEMP:=TO_SIGNED(0,96); -- Reset of variable A_TEMP

FOR j IN 0 TO 6 LOOP
        A_TEMP:=shift_right(M_IN(j)*A(j),16);
    SUM_A:=SUM_A+A_TEMP(63 DOWNTO 0);
END LOOP;

TEMP_A<=SUM_A; --Necessary because the signal instance have to be inside the
    process
END PROCESS;
A_PARTIAL<=TEMP_A;
```

Computation of B matrix Similarly to what explained in the previous paragraph, the block $\boldsymbol{H_{ij}}$ **x** $\boldsymbol{a_i}$ **x** $\boldsymbol{a_j}$ receives as input

- H_{in}, the 7x7 sub-matrix of H correspondent to (i, j) cycle properly selected from the external block *H Selection* such that

$$H_{in} = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} & H_{04} & H_{05} & H_{06} \\ H_{10} & H_{11} & H_{12} & H_{13} & H_{14} & H_{15} & H_{16} \\ H_{20} & H_{21} & H_{22} & H_{23} & H_{24} & H_{25} & H_{26} \\ H_{30} & H_{31} & H_{32} & H_{33} & H_{34} & H_{35} & H_{36} \\ H_{40} & H_{41} & H_{42} & H_{43} & H_{44} & H_{45} & H_{46} \\ H_{50} & H_{51} & H_{52} & H_{53} & H_{54} & H_{55} & H_{56} \\ H_{60} & H_{61} & H_{62} & H_{63} & H_{64} & H_{65} & H_{66} \end{bmatrix}$$

  where each H_{xx} is a 64-bit element.

- a_{in}, the updated vertical filter

The algorithm to map is the following:

```
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;
      }
    }
  }}
```

The developed architecture is at figure 3.14. The *Index Selection* block selects the (i, j) component from the input sub-matrix and the i-th from a_{in}. It is important to clarify that both counters i and j are internal to this block and used to scan the 7x7 matrix H_{in}.They must not be confused with the ones used from *H Selection* and *M Selection* to define properly the sub-matrix and sub-vector that have to be processed. One value of $B_{partial}$ is produced for each (i, j) cycle: similarly to the computation of A matrix, a *Feedback Sum* is needed to combine all subsequent contributions.

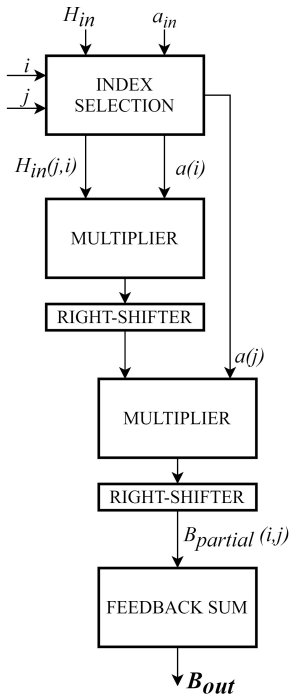


Figure 3.14: H_{ij} x a_i x a_j data path

This is done by using the *variable* construct approach like follows:

```
PROCESS(H_IN,A)
VARIABLE B : SIGNED(95 DOWNTO 0):=TO_SIGNED(0,96);
VARIABLE SUM_B: SIGNED(63 DOWNTO 0):=TO_SIGNED(0,64);
BEGIN
SUM_B:=TO_SIGNED(0,64);-- Reset of variable SUM_B
B:=TO_SIGNED(0,96); -- Reset of variable B

FOR I IN 0 TO 6 LOOP
        FOR J IN 0 TO 6 LOOP
        B:=(shift_right(((shift_right((H_IN(J,I)*A(I)),16)(63 DOWNTO 0))*A(J)),16))
            ;
        SUM_B:=SUM_B+B(63 DOWNTO 0);
END LOOP;
END LOOP;

TEMP_B<=SUM_B;

END PROCESS;
B_PARTIAL<=TEMP_B;
```

Top-level architecture The top-level implementation is really similar to *Update a* architecture, including the considerations done for the input multiplication blocks: the same *M selection* block is integrated, while the sub-matrix H_{ij} is provided directly from the testbench.

The final top-level architecture is presented at figure 3.15.

The flow is controlled by an FSM exactly equal to the one reported at figure 3.12. Apart from the already described multiplication blocks, another difference between the two configurations regards the storing algorithm to derive the matrix B and the vector A.

For the first seven clock cycle where $i = 0$ and only j is incremented, M_{ij} x a_i produces a value of $A_{partial}$ stored in the j-th position of a 7-elements vector for each step $(0, j)$. Once filled it, their components are processed by a tree of adder to perform the final value of A vector:

```
PARTIAL_A_ARRAY(j)<=PARTIAL_A;
--Once filled the array:
A(0)<=PARTIAL_A_ARRAY(0)+PARTIAL_A_ARRAY(6);
A(1)<=PARTIAL_A_ARRAY(1)+PARTIAL_A_ARRAY(5);
A(2)<=PARTIAL_A_ARRAY(2)+PARTIAL_A_ARRAY(4);
A(3)<=PARTIAL_A_ARRAY(3);
```

At the same time, H_{ij} x a_i x a_j computes the value $B_{partial}(0, j)$ storing it in a 7 x 7 matrix. From the 8^{th} clock cycle, when the whole vector A has been computed and stored, it is necessary to wait the completion of B disabling the store operation for A, similarly to what done for feedback registers in **Disable Ret Register** state. For each subsequent (i, j) step, $B_{partial}(i, j)$ is stored into (i, j) elements of the partial matrix.

The final value for B is computed once filled the matrix and it requires 49 total clock cycles.

```
PARTIAL_B_MATRIX(i,j)<=PARTIAL_B;
-- Once filled the matrix:
B(0,0)<=PARTIAL_B_MATRIX(0,0)+PARTIAL_B_MATRIX(0,6)+PARTIAL_B_MATRIX(6,0)+
    PARTIAL_B_MATRIX(6,6);
B(0,1)<=PARTIAL_B_MATRIX(0,1)+PARTIAL_B_MATRIX(0,5)+PARTIAL_B_MATRIX(6,1)+
    PARTIAL_B_MATRIX(6,5);
B(0,2)<=PARTIAL_B_MATRIX(0,2)+PARTIAL_B_MATRIX(0,4)+PARTIAL_B_MATRIX(6,2)+
    PARTIAL_B_MATRIX(6,4);
B(0,3)<=PARTIAL_B_MATRIX(0,3)+PARTIAL_B_MATRIX(6,3);
B(1,0)<=PARTIAL_B_MATRIX(1,0)+PARTIAL_B_MATRIX(1,6)+PARTIAL_B_MATRIX(5,0)+
    PARTIAL_B_MATRIX(5,6);
B(1,1)<=PARTIAL_B_MATRIX(1,1)+PARTIAL_B_MATRIX(1,5)+PARTIAL_B_MATRIX(5,1)+
    PARTIAL_B_MATRIX(5,5);
B(1,2)<=PARTIAL_B_MATRIX(1,2)+PARTIAL_B_MATRIX(1,4)+PARTIAL_B_MATRIX(5,2)+
    PARTIAL_B_MATRIX(5,4);
B(1,3)<=PARTIAL_B_MATRIX(1,3)+PARTIAL_B_MATRIX(5,3);
B(2,0)<=PARTIAL_B_MATRIX(2,0)+PARTIAL_B_MATRIX(2,6)+PARTIAL_B_MATRIX(4,0)+
    PARTIAL_B_MATRIX(4,6);
B(2,1)<=PARTIAL_B_MATRIX(2,1)+PARTIAL_B_MATRIX(2,5)+PARTIAL_B_MATRIX(4,1)+
    PARTIAL_B_MATRIX(4,5);
B(2,2)<=PARTIAL_B_MATRIX(2,2)+PARTIAL_B_MATRIX(2,4)+PARTIAL_B_MATRIX(4,2)+
    PARTIAL_B_MATRIX(4,4);
B(2,3)<=PARTIAL_B_MATRIX(2,3)+PARTIAL_B_MATRIX(4,3);
B(3,0)<=PARTIAL_B_MATRIX(3,0)+PARTIAL_B_MATRIX(3,6);
B(3,1)<=PARTIAL_B_MATRIX(3,1)+PARTIAL_B_MATRIX(3,5);
B(3,2)<=PARTIAL_B_MATRIX(3,2)+PARTIAL_B_MATRIX(3,4);
B(3,3)<=PARTIAL_B_MATRIX(3,3);
```

All the steps necessary to derive and solve the linear system of equations are exactly the same explained during the analizys of vertical filter since it consists only on the application of a fixed mathematical solution and it doesn't depend on the unknown variables processed. This is why the C model exploits the same *linsolve_ wiener* function to compute both horizontal and vertical filter.

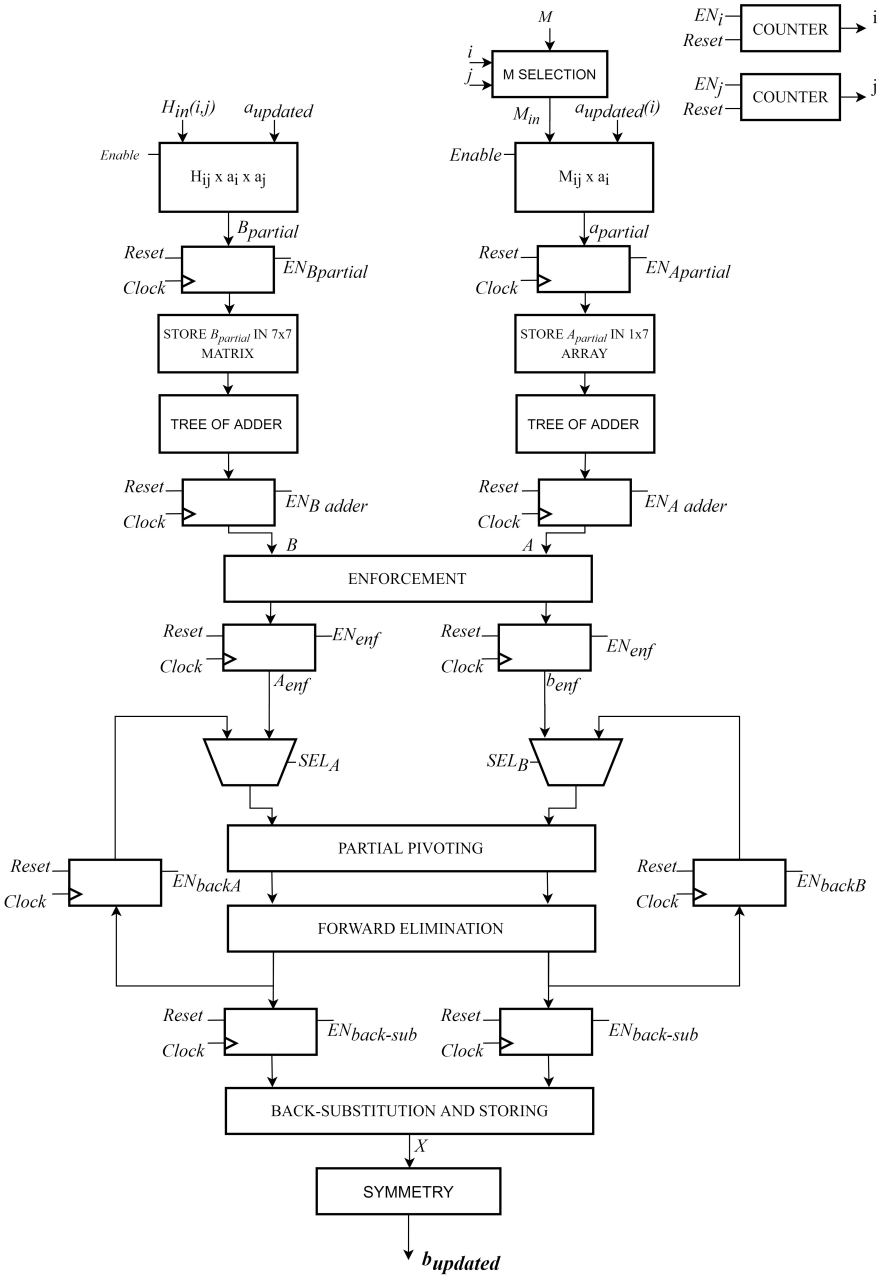


Figure 3.15: Top-Level Data path for *update b*

3.2 Implementation analysis: results and performances

The last step of every kind of design must be the evaluation of the results, not only from a numerical point of view but also in terms of performances reached. It is very interesting at the end to discover if the designed architecture is working properly and which is the cost to pay to obtain the correct results. To do that both *Simulation* and *Synthesis* operation have been performed on the final architecture reported at figure 3.16.

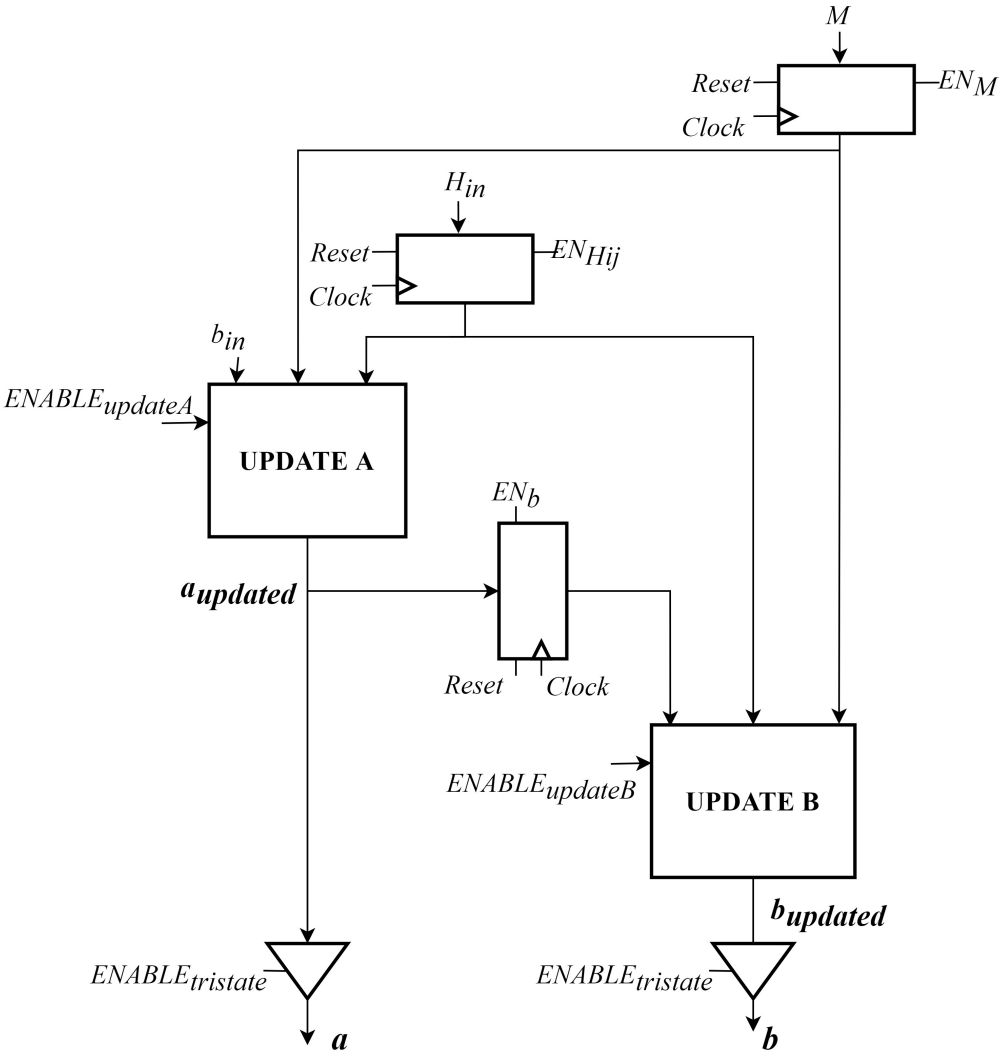


Figure 3.16: Top-Level Data Path for Wiener Filter

To enable properly *Update a* and *Update b* the FSM at figure 3.17 has been designed.

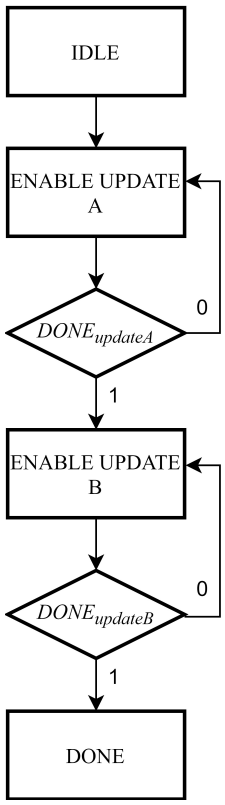


Figure 3.17: Top-Level FSM for Wiener Filter

These few states are necessary to control the correct data flowing and in particular to ensure that the horizontal filter b is computed only after the completion of *Update a* using the signal ($Done_{updateA}$), otherwise it would receive a wrong value of input updated filter. Obviously the possibility to making both filtering architectures working in parallel would be efficient in order to decrease by a factor 2 the processing time but, unfortunately, the filter algorithm doesn't allow that due to data dependencies.

Two mentions have to be done for what concerns this architecture:

- In order to ensure a good timing behavior, each input is processed after being stored in a register. Placing a series of input registers is a good choice in general for every digital design, but especially in this particular case where the structure is inside a loop because it gives the possibility to enable the architecture to read the input only when some valid are available avoiding to work with useless data.

- This architecture has not to be intended stand-alone because its output became the input of the following filtering process. This loop behavior forces the

employment of two output buffer tri-state to ensure that both outputs are available only when the last of two has been completed, otherwise wrong partial results will be inserted inside the loop. To do that a signal $Done_{updateB}$ is exploited to drive both buffers tristate and the outputs are released only when it is equal to 1.

3.2.1 Simulation with *Modelsim*

The only way to verify if the architecture works properly is to compare the results obtained by executing the software model of Wiener Filter with the ones obtained simulating the Hardware architecture.

Using a *.yuv* input file called *"Flowervase_832_480_30.yuv"* , the encoder "aomenc" by *libaom* AV1 library has been executed by using *cmake* command, printing on screen the most significant data, above all:

- input matrix H;
- input vector M;
- input vector b_{in};
- indexes from *"wrap_index"* function;
- coefficient matrix and known factors of linear system of equation;
- output value for a and b.

Once derived all the necessary input values, the *testbench* has been developed in the following way:

- H, M and b_{in} have been provided by hand using process statement;
- the Wiener Filter Top-Level has been instantiated;
- the component related to *H selection* has been included providing the proper value of counter i and j directly from the Wiener Filter Top-Level component. Therefore, no counters has been developed in the testbench;

The simulation results is reported with the Modelsim extract of figure 3.18.

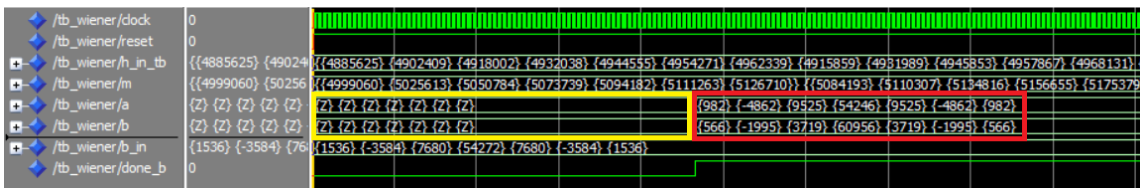


Figure 3.18: Modelsim simulation for basic Wiener Filter

Until the signal *done_b* became '1', both output lines are high-impedence [yellow rectangle]. The signal drives output buffers to release both horizontal and vertical filter at the same time [red rectangle].

3.2.2 Synthesis with *Synopsys*

Once verified the behavior of the basic architecture testing it on Modelsim , it has been synthesized using Synopsys. Hereinafter the attention will be focused mainly on two parameters:

- Speed: the minimum clock period is derived by the evaluation of critical path from which depends the maximum value of working frequency;
- Area: the dimension of both combinatorial and not combinatorial cells involved has been established.

The synthesis has been performed with a *top-down* approach: after having elaborated the top-level and all its components, the design is compiled providing directly all the proper timing constraints starting from the higher level architecture and leaving the compiler to transmit timing information to every internal components. An example of a typical used script is reported below:

```
set_host_options -max_cores 3
analyze -f vhdl -lib WORK ../src/MATRIX_PACK.vhd
analyze -f vhdl -lib WORK ../src/DIVISION.vhd
analyze -f vhdl -lib WORK ../src/COUNTER.vhd
analyze -f vhdl -lib WORK ../src/M_SELECTION.vhd
analyze -f vhdl -lib WORK ../src/REG_1X3_64BIT.vhd
analyze -f vhdl -lib WORK ../src/REG_1X4_64BIT.vhd
analyze -f vhdl -lib WORK ../src/REG_1X7_32BIT.vhd
analyze -f vhdl -lib WORK ../src/REG_3X3_64BIT.vhd
analyze -f vhdl -lib WORK ../src/REG_4X4_64BIT.vhd
analyze -f vhdl -lib WORK ../src/REG_7X7_64BIT.vhd
analyze -f vhdl -lib WORK ../src/REG_32BIT.vhd
analyze -f vhdl -lib WORK ../src/REG_64BIT.vhd
analyze -f vhdl -lib WORK ../src/MATRIX_SUM.vhd
analyze -f vhdl -lib WORK ../src/SUM_VECTOR.vhd
analyze -f vhdl -lib WORK ../src/PIVOTING.vhd
analyze -f vhdl -lib WORK ../src/MUX_A.vhd
analyze -f vhdl -lib WORK ../src/MUX_B.vhd
analyze -f vhdl -lib WORK ../src/MIJ_BI.vhd
analyze -f vhdl -lib WORK ../src/MIJ_AI.vhd
analyze -f vhdl -lib WORK ../src/HIJ_BI_BJ.vhd
analyze -f vhdl -lib WORK ../src/HIJ_AI_AJ.vhd
analyze -f vhdl -lib WORK ../src/FORWARD_ELIMINATION.vhd
analyze -f vhdl -lib WORK ../src/ENFORCEMENT.vhd
analyze -f vhdl -lib WORK ../src/buffer_tri_state.vhd
analyze -f vhdl -lib WORK ../src/BACK_SUB_STORING.vhd
analyze -f vhdl -lib WORK ../src/FSM.vhd
analyze -f vhdl -lib WORK ../src/FSM_B.vhd
analyze -f vhdl -lib WORK ../src/FSM_TOP.vhd
analyze -f vhdl -lib WORK ../src/UPDATE_A.vhd
analyze -f vhdl -lib WORK ../src/UPDATE_A_TOPLEVEL.vhd
```

```
analyze -f vhdl -lib WORK ../src/UPDATE_B.vhd
analyze -f vhdl -lib WORK ../src/UPDATE_B_TOPLEVEL.vhd
analyze -f vhdl -lib WORK ../src/WIENER_FILTER.vhd
elaborate WIENER_FILTER -arch BEHAVIOR -lib WORK
create_clock -name MY_CLOCK -period 10 CLOCK
set_dont_touch_network MY_CLOCK
set_clock_uncertainty 0.07 [get_clocks MY_CLOCK]
set_input_delay 0.5 -max -clock MY_CLOCK [remove_from_collection [all_inputs]
    MY_CLOCK]
set_output_delay 0.5 -max -clock MY_CLOCK [all_outputs]
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set_load $OLOAD [all_outputs]
compile
report_timing> ./report_timing.txt
report_area> ./report_area.txt
```

Timing performances [T_{cp}]: The *report_timing* command is typed to extract the most relevant timing informations of the compiled architecture . The compiler maps each single component defining the critical path using pre-defined cells from the "NangateOpenCellLibrary": the time spent to propagate data along the longest path, called T_{cp}, is derived by means of a parameters called $SLACK$ that is defined like the difference between the constrained time and the actual time of a timing path: it represent the quantity of the clock period not used to transmit data along the critical path .Let's consider a generic clock period T_{ck}. It is:

$$T_{cp} = T_{ck} - SLACK \tag{3.9}$$

This relation gives informations on the architecture reliability. In particular:

- $SLACK > 0$ means that the design has a working frequency such that the processed data is able to travel all the subsequent combinatorial blocks inside the clock period defined by the provided constraint;
- $SLACK < 0$ means that the actual clock period is not enough to propagate data along the whole critical path and the architecture can not work at this frequency. The only way to solve it is to make the constraint softer.

From the file "report_timing.txt" it has been noticed that the critical path of synthesized architecture is given by the block *Back-Substitution*, and in particular by the computation of *x(0)*.

$$T_{cp} = 3 \cdot T_{division} + 2 \cdot T_{multiplication} + 2 \cdot T_{subtraction} + T_{adder} + 4 \cdot T_{shifter} \tag{3.10}$$

Clock Period[T_{ck}]	SLACK	Critical Path Delay [T_{cp}]
10 ns	-221.43 ns	231.42 ns
232 ns	+0.01	231.42 ns

Table 3.1: Critical Path delay for basic implementation

The first attempt was to provide a clock constraint of $10ns$ but the related SLACK was strongly negative. The lowest SLACK of $+0.01ns$ is obtained by considering a 232 ns clock period, from which 231.43 ns are available to propagate data since it is necessary to include also a fixed *clock uncertainty* parameters equal to 0.07 ns and an output external delay of 0.50 ns. The obtained working frequency is:

$$f_s = \frac{1}{T_{ck}} = \frac{1}{232ns} \simeq 4.31MHz \tag{3.11}$$

As expected, the combinatorial nature of the presented basic implementation is the cause of a bad result in terms of frequency. To make the execution speeder, the architecture should be strongly pipelined in order to split as much as possible the critical paths. This kind of modification will not be discussed in this thesis because the aim is to present a low-complexity optimization approach.

Area performances: The command *report_ area* allows to analyse each contribution of area of every involved component. InFrom the file "report_area.txt" it is possible to identify the main results that are reported in the table below:

Number of ports	1245024
Number of nets	5016113
Number of cells	3568398
Number of combinational cells	3539633
Number of sequential cells	22044
Number of buf/inv	862746

Table 3.2: *report_ area* results of basic implementation

The total area requirement is given by two contributions:

Contribution	Area [µm^2]
Combinational Area	4132550.177
Non-combinational area	80061.746

Table 3.3: Total area of basic design implementation

The total area occupied by the architecture is almost equal to 4.22 mm^2, that is definitely too much high to be just a little component of the entire video codec environment.

3.3 Elaboration for a real-time video sequence

To better understand the provided results and to better focus on the impact they have on the codec execution it is necessary to define some target application and analyse their elaboration related to the performances reached with the proposed basic architecture.

In particular, it is possible to consider some real-time video sequences and evaluate for each of them how the design works. By evaluating the implementations throughput, intended as the number of samples processed per second, it is possible to define for a certain application how many frames can be elaborated in a real-time processing. The most known video format have been chosen like target application:

- Standard Definition [SD]: there are different kind of SD sequences, based on their resolution. The most used are:
  - 480i, with a resolution of 720x480 pixels;
  - 576i, with a resolution of 720x576 pixels;
- High Definition [HD]: it is the most used nowadays and it is divided into:
  - 720p, with a resolution of 1280x720 pixels;
  - 1080p, with a resolution of 1920x1080 pixels.

For each resolution, the related approximated *fps* (frames per second) is reported at table 3.4.

Application	Resolution [pixels x pixels]	*fps* [Basic Architecture]
SD [480i]	720 x 480	**12**
SD [576i]	720 x 576	**10**
HD [720p]	1280 x 720	**4**
HD [1080p]	1920 x 1080	**2**

Table 3.4: Video sequences fps for the basic architecture

It is clear how the designed basic architecture can not sustain high frame rates neither for SD or HD definition. This is due to its combinatorial nature that increase the time necessary to process each pixel.

3.4 From basic to optimized implementation: the need of a complexity reduction

The main reason why the attention is focused on the complexity optimization is that it is challenging not only didactically speaking but also from a technological point of view. In the last 30 years, the need of reaching an heterogeneous integration led to a technological scaling of dimension. In particular, reducing transistor size, it was possible to integrate more functionality inside the same chip and this is the reason why the maximum dimension of an integrated circuit, and so the maximum area that can be exploited for system design is increased by 50% every year. It is really important to keep under control the complexity in VLSI design because it determines also the manufacture costs, the quality of production process and power consumption, expecially in the presented situation where the analysis is limited to a small part of the whole architecture.

- **Cost:** the total cost of one die depends strongly from die area with a proportionality that has been extimated to be about the fourth power [10]:

$$cost\ of\ die = f(die\ area)^4 \tag{3.12}$$

  Since the area can be controlled by designer, the aim must be to produce gates smaller and smaller to reach higher integration density and smaller die size.

- **Production quality:** the area influences also the quality of die production through the *yield* parameter, defined like the ratio between the number of working die inside a wafer respect to the total one. From a statistical point of view, larger is the area lower is the yield because less die fit on the wafer. It is very important to control this technological parameter because many devices requires very strict constraint on it.

- **Power consumption:** The power dissipation of a circuit is given by:

$$P_{avg} \propto f_{clk} \cdot C_L \cdot V_{DD}^2 \tag{3.13}$$

  Reducing the complexity of involved operator the total switching capacitance C_L is reduced as well.

Chapter 4

Low complexity architecture of AV1 Wiener Filter

The basic idea to develop a low complexity architecture is to work with a *bottom to top* approach. The optimization has been approached following a precise methodology described in the steps below that will be deepen in this chapter:

1. Individual analysis and synthesis of each involved block designed for the basic architecture.
2. Identification of the most critical in terms of complexity.
3. Modification and optimization to reach a low-complexity version of each component.
4. Design of the top-level architecture for both *Update a* and *Update b*.
5. Integration of the optimized implementations in the top-level architectures.

The synthesis results obtained by performing the *report_ area* of each component one at a time are reported in the table 4.1. This analysis was useful to give an idea of what to focus on to reduce the complexity of the basic architecture. In particular:

- The *divider* contribution is highlighted because it is really crucial in that phase of research. This is the reason why the whole optimization mechanism began by analysing the divider: the implementation used for the basic design and explained at paragraph 3.1.1 is unsustainable for a low-complexity purpose because it involves a huge number of operators. The massive usage of the divider component expecially in already complex blocks such as *Forward Elimination* or *Back-substitution* lead to a further increment of their complexity and, as a consequence, to the need for a new optimized version.

Hardware component	Area[µm^2]
M_{ij} x b_i	77730.520
M_{ij} x a_i	76808.298
H_{ij} x b_i x b_j	1033883.482
H_{ij} x a_i x a_j	1034442.881
Enforcement	10275.846
Pivoting	4164.495
Forward Elimination	709903.197
Back-substitution	193676.729
Divider	**87258.107**

Table 4.1: Area occupation of each component for basic implementation

- The multiplicative blocks are relevant in terms of area requirement because, even if they don't employee divider, they need for several multipliers.
- The central part of the elaboration composed by *Enforcement* and *Pivoting* is not crucial since they perform few number of basic operations, like swapping.

The critical elements have been strongly modified using a *folding* approach in order to share hardware components. What really changes respect to the basic implementation is the design approach: it is not possible to use unlimited resources, so the design is based on the idea of reducing at minimum the number of exploited components. Basically, the same operator is used whenever possible providing step by step the correct inputs by means of *multiplexers* in order to perform the related operation; every time a data dependency occurred a certain number of register is placed to ensure the correct algorithm execution. On the other hand, as it will be explained better during the following sections, the cost is a considerable increment of latency since several clock cycles are required to complete each operation.

For the reasons explained above, the optimization for a *low-complexity* version of AV1 Wiener Filter has been developed for both *Update a* and *Update b* following these steps:

1. optimization of divider algorithm;
2. optimization of most complex components using *folding*;
3. implementation of FSM for each block to handle data flow;
4. integration of re-designed component into the top-level architecture;
5. implementation of a specific top-level FSM to ensure the algorithm execution accuracy.

It is necessary clarify that this proposed way for the design optimization, that is valid for both the horizontal and vertical filter design, works only because this is a dedicated architecture and the basic algorithm to map extracted from the C model is fixed and well known. This is really important to underline how each kind of implementation reported in this work is adapted on each single line of the code and many architectural choices have been made knowing exactly what expected from the execution. A very efficient optimization would have required to change the algorithm description in some way maintaining the same working behavior, since obviously the provided source code has not been developed for any kind of optimization purpose . The algorithm instructions to map block by block are exactly the same of the ones available at Chapter 3 and will not be reported in this section.

4.1 Divider algorithm optimization

From a purely computational point of view, the main problem of the previous implementation for the division is the employment of 64 adder and 64 subtractors to perform one single operation. The main goal of the optimization is to reduce as much as possible the number of hardware components even paying in terms of latency: in fact, the only way to reduce the complexity of the division is to distribute the whole operation in several clock cycle end re-use the same operators. The implemented architecture follows again a **"Restoring divider algorithm"** and it is reported at figure 4.1.

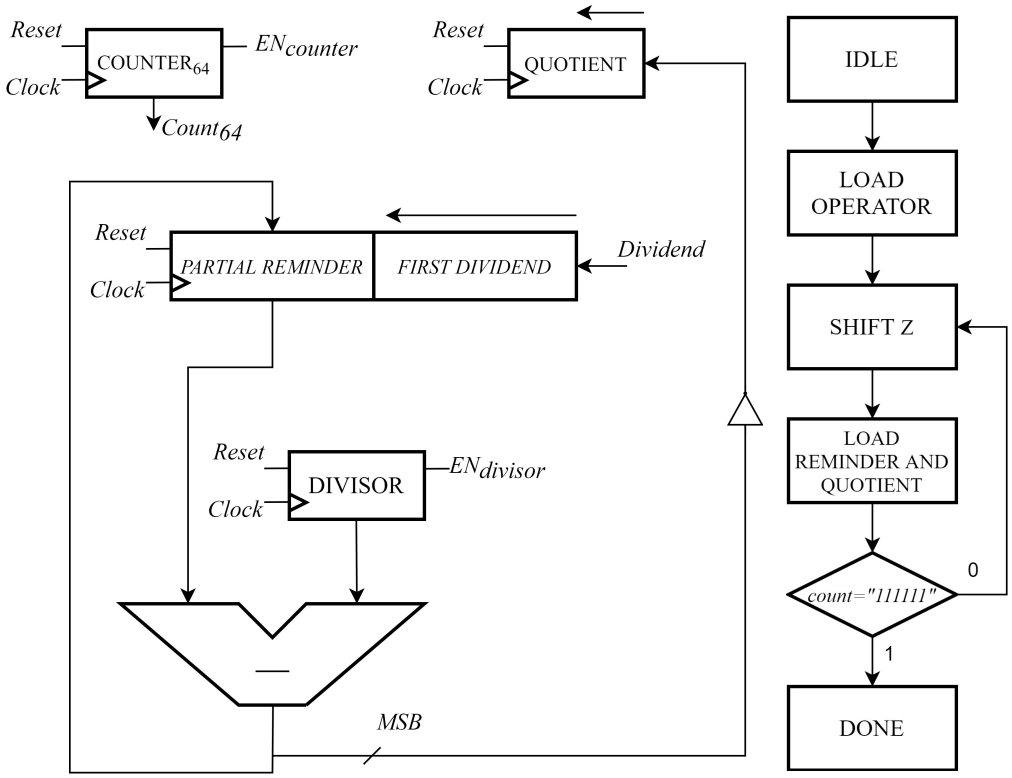
The used components are:

- a 128 bit shift-register, where the 64 LSBs are occupied by the dividend, while the 64 MSBs contain the value of partial reminder computed cycle by cycle;
- a 64 bit register where the divisor has been stored;
- a 64 bit counter to stop the iteration at 64^{th} cycle;
- a 64 bit shift-register to store the quotient.

For each i-th cycle, the value of partial reminder is computed as:

$$s^{(i)} = 2s^{(i-1)} - q_{64-i} \cdot (2^{64}d) \tag{4.1}$$

The algorithm is that of typical of a Restoring Divider: at each cycle i, the first 64 MSBs from the 128 bit shift-register are taken and from which the divisor is subtracted(the 2^{64} shifting is made taking only the first 64 MSBs): if the result is positive a $'1'$ is stored in the Quotient shift register and the Partial Reminder is loaded into 128 bit shift register starting from the MSB, otherwise a $'0'$ is stored



(a) Optimized division data path

(b) Optimized division FSM

Figure 4.1: Optimized Restoring Divider

into the quotient shift register without loading the partial reminder. At cycle $i+1$, the value taken from 128 bit shift-register depends on the subtraction result and it will be:

- $2 \cdot s^{(i)}$ if it was positive;
- $2 \cdot s^{(i-1)}$ if it was negative.

The proper value of quotient is stored inverting the sign bit of the partial reminder, in order to save 1 every time it is positive and 0 when negative. The algorithm requires several control signals for store and load operations: for this purpose, the FSM at figure 4.1 has been implemented.

- The **Load operator** state enables the storing of both dividend and divisor in correspondent registers;
- The **Shift z** state enables the shift operation of both shift-registers at the end of each cycle;

- The **Load reminder and quotient** state provide a load enable signal for partial reminder equal to the opposite of its sign bit and enables the quotient storing;

- When the value of counter reach 64, the operation is concluded.

This algorithm is very powerful because, apart from the fact that it is very easy to implement, it requires just one subtractor and 4 registers to perform the division but, on the other side, each operation lasts 64 clock cycles.

Making a comparison between the previous implementation:

Implementation	Area[µm^2]
Basic divider	87258.107
Optimized divider	3426.346

Table 4.2: Complexity comparison between dividers

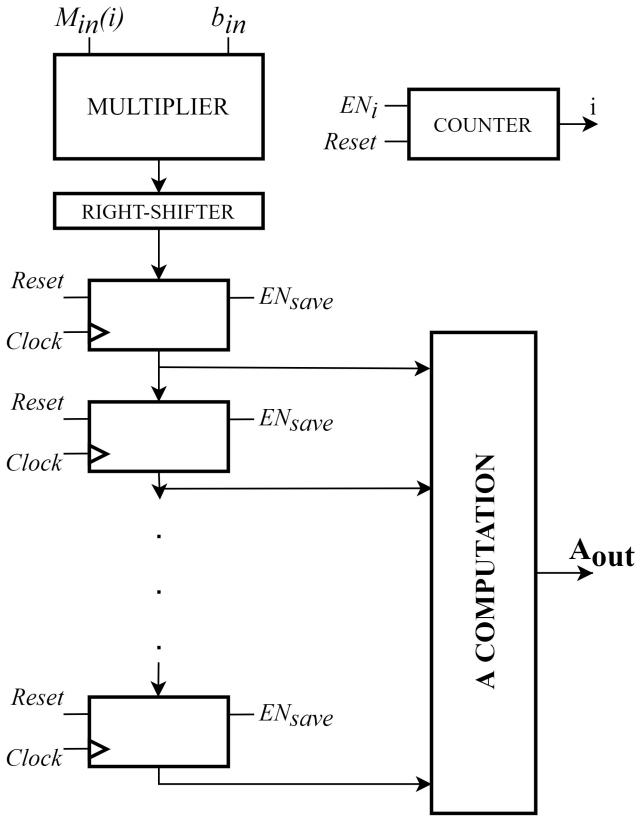
The complexity has been reduced of a factor close to **26**.

4.2 Hardware architecture of optimized *update a*

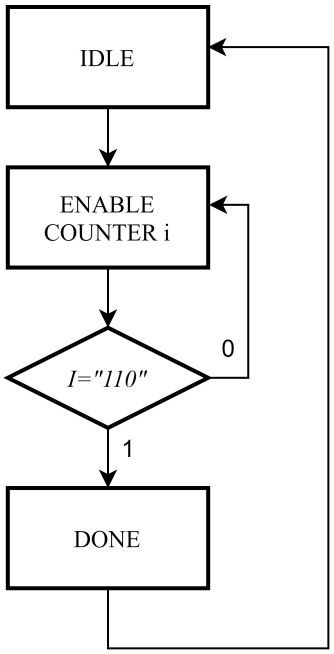
The architecture designed for each block is the product of several implementation attempts: it is no longer enough to develop a working architecture, but it is necessary to reach the minimum level of complexity The mapped algorithm describing the Wiener Filter is obviously the same reported at Chapter 2. All the considerations presented here concern only the integrated blocks without including any kind of optimization for the top-level architecture to underline how it is possible to achieve a great improvement changing only the behavior of its components. The fixed point in the design is to exploit just one operator for each kind whenever allowed by data dependency constraints. Every modification aimed to the architecture optimization has been designed in order to obtain a balanced structure, in order to make always a trade off between latency and complexity: for this reason the lower-area components have not been modified because optimizing an already low complex elements would not have introduced a relevant improvement in terms of area, but would have increased too much the latency. That's why the intermediate block of *Enforcement* and *Partial Pivoting* have not been optimized and will not be presented in this section.

4.2.1 Computation of vector A

This block, called again "M_{ij} x b_i", receives the same input of the basic implementation and reported at paragraph 3.1.1. It has been designed considering that the employment of 7 different multipliers which exploit the same fixed operand b_{in} is critical in terms of area occupation: this is why one single multiplier is shared between the 7 components of input vector M_{in}. The architecture is described by the data path and the FSM reported at figure 4.2



(a) Data Path M_{ij} x b_i



(b) FSM M_{ij} x b_i

Figure 4.2: Hardware implementation forM_{ij} x b_i

The first multiplier operand is provided by selecting at each cycle the i-th component of input vector M_{in} through the output of the *Counter* , enabled at state **Enable counter i**. To store the partial product produced step by step, a chain of seven registers have been employed (e.g. at the i-th cycle, the i-th value of partial product is stored in first register while the second contains the one related to cycle $i-1$, the third the one of cycle $i-2$ and so on). At the 6^{th} step, the operation is completed and all the values stored inside the registers are combined in the A

computation block that works as follows (the registers are enumerated from the top to the bottom):

```
B_OUT(0)<= OUT_REG_1 + OUT_REG_7;
B_OUT(1)<= OUT_REG_2 + OUT_REG_6;
B_OUT(2)<= OUT_REG_3 + OUT_REG_5;
B_OUT(3)<= OUT_REG_4;
```

In this case 4 different adders have been implemented even if it was possible to reduce more the complexity by using just one and performing the operations one at a time: this solution has not been considered because the computational cost of an adder is quite negligible respect to the multiplier, so the complexity would have been decreased of a little factor paying a further deteriorating of latency performances. The optimization led to a complexity reduction of a factor about 7.7.

Implementation	Area[µm^2]
Basic M_{ij} x b_i	77730.520
Optimized M_{ij} x b_i	10104.542

Table 4.3: Complexity comparison between M_{ij} x b_i

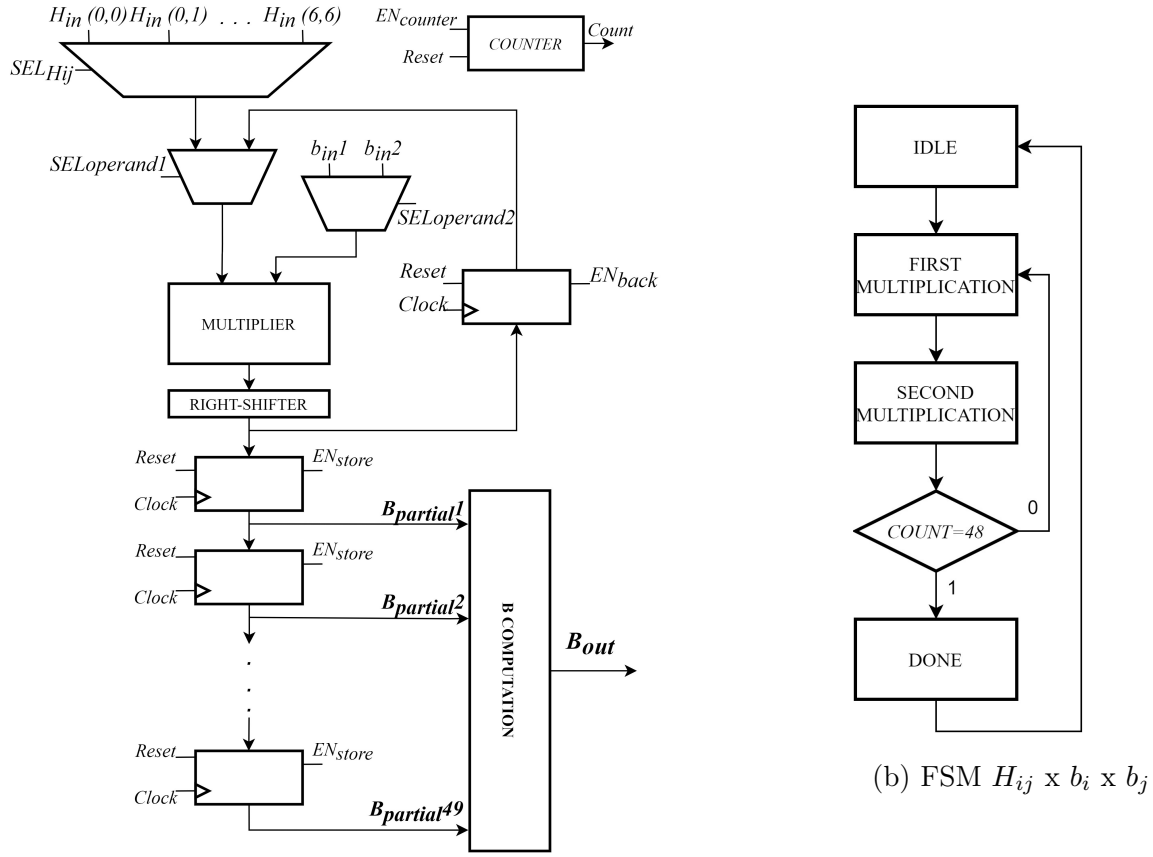
4.2.2 Computation of matrix B

The considerations on the component "$\boldsymbol{H_{ij}}$ **x** $\boldsymbol{b_i}$ **x** $\boldsymbol{b_j}$ take a crucial role in the whole optimization since it is one of the most expensive of the architecture. From a computational point of view, the multipliers are the bottleneck since the algorithm requires 98 multiplications. Analysing better the basic structure reported at paragraph 3.1.1 it is possible to notice that, having the same inputs:

- The first set of multipliers receives a fixed operand equal to $b_{in}1$ and one element of 7 x 7 sub-matrix H_{in}.
- The second set of multiplier uses the result of previous multiplication and another fixed operand equal to $b_{in}2$

These operations have been replaced by using:

1. one single multiplier;
2. a 49-to-1 multiplexer selecting the proper element of H_{in};
3. a 2-to-1 multiplexer that selects the fixed operand from $b_{in}1$ and $b_{in}2$;
4. a 2-to-1 multiplexer to select which set of multiplication have to be performed.



(a) Data Path H_{ij}x b_ix b_j

(b) FSM H_{ij} x b_i x b_j

Figure 4.3: Hardware implementation forH_{ij} x b_i x b_j

The architecture is reported at figure 4.3. The selection signals are provided in **First Multiplication** and **Second multiplication** state as follows:

- In *"First Multiplication "* the multiplier is used to compute $H_{in}(i,j) \cdot b_{in}1$, so:
  - SEL_{Hij} is exploited to scan the 7 x 7 matrix horizontally and vertically by using the value of *count* signal. It is updated in the range $(0-48)$ and the selected element is extract sorting the matrix from left to right:

$$\begin{bmatrix} H_0 & H_1 & H_2 & H_3 & H_4 & H_5 & H_6 \\ H_7 & H_8 & H_9 & H_{10} & H_{11} & H_{12} & H_{13} \\ H_{14} & H_{15} & H_{16} & H_{17} & H_{18} & H_{19} & H_{20} \\ H_{21} & H_{22} & H_{23} & H_{24} & H_{25} & H_{26} & H_{27} \\ H_{28} & H_{29} & H_{30} & H_{31} & H_{32} & H_{33} & H_{34} \\ H_{35} & H_{36} & H_{37} & H_{38} & H_{39} & H_{40} & H_{41} \\ H_{42} & H_{43} & H_{44} & H_{45} & H_{46} & H_{47} & H_{48} \end{bmatrix}$$

So, step by step, the value H_{count} comes out from the input matrix.

- $SEL_{operand1}$ selects as first operand the data coming from the matrix multiplexer providing to multiplier $H_{in}(i, j)$.
- $SEL_{operand2}$ provides $b_{in}1$ as second operand.

- In "Second Multiplication" state:
  - SEL_{Hij} is neglected since the matrix element has already been used.
  - $SEL_{operand1}$ selects as first operand data coming from the feedback register, providing to multiplier the shifted result of the previous calculation.
  - $SEL_{operand2}$ provides $b_{in}2$ as second operand.

After each second stage of multiplication a partial value is stored in a register chain similarly to the *"M_{ij} x b_i"* architecture, but employing 49 registers. It is interesting to notice how the reduction of huge components corresponds to a strongly increment of registers number that have very little influence on the complexity of the system being low-area components.

The derivation is completed by combining all the 49 partial values obtained in the *B Computation* component:

```
B0<=OUT_REG_1+OUT_REG_7+OUT_REG_43+ OUT_REG_49;
B4<=OUT_REG_2+OUT_REG_6+OUT_REG_44+ OUT_REG_48;
B8<=OUT_REG_3+OUT_REG_5+OUT_REG_45+ OUT_REG_47;
B12<=OUT_REG_4 + OUT_REG_46;

B1<=OUT_REG_8+OUT_REG_14+OUT_REG_36+ OUT_REG_42;
B5<=OUT_REG_9+OUT_REG_13+OUT_REG_37+ OUT_REG_41;
B9<=OUT_REG_10+OUT_REG_12+OUT_REG_38+ OUT_REG_40;
B13<=OUT_REG_11+ OUT_REG_39;

B2<=OUT_REG_15+OUT_REG_21+OUT_REG_29+ OUT_REG_35;
B6<=OUT_REG_16+OUT_REG_20+OUT_REG_30+ OUT_REG_34;
B10<=OUT_REG_17+OUT_REG_19+OUT_REG_31+ OUT_REG_33;
B14<=OUT_REG_18 +OUT_REG_32;

B3<=OUT_REG_22+ OUT_REG_28;
B7<=OUT_REG_23+ OUT_REG_27;
B11<=OUT_REG_24+ OUT_REG_26;
B15<=OUT_REG_25;
```

The advantage obtained is a complexity reduction of a factor close to **25**.

Implementation	Area[µm^2]
Basic H_{ij} x b_i x b_j	1033883.482
Optimized H_{ij} x b_i x b_j	41748.966

Table 4.4: Complexity comparison between H_{ij} x b_i x b_j

Once obtained B matrix and A vector, it is necessary to apply the same *Enforcement* step described at paragraph 3.1.1 to fit the dimensions and then solve the linear system of equation by using the same Gauss-elimination algorithm exploiting *Partial Pivoting*, *Forward Elimination* and *Back-substitution*. As mentioned, the architecture for *Enforcement* and *Partial Pivoting* are not explained in this section and their basic version described at paragraph 3.1.1 and 3.1.1 respectively are used. The only difference in the pivoting operation is that, differently from the unique structure presented in the basic implementation, it is now divided in two parts, one for each processing stage in order to adapt it to the *Forward Elimination* operation that have to be necessarily processed in two separated step since the derivation of the architecture differs depending on the processing stage and each of them will be handled with different Finite State Machines: adopting two blocks ensures a better synchronization.

4.2.3 First stage of Forward Elimination

After the first step of *Partial Pivoting* , a **Forward Elimination** operation on the last two rows (6 elements) is performed again to start the conversion into an upper triangular form. The basic version described at paragraph 3.1.1 uses for each input, above all, one multiplier, one divider and one subtractor. The data flow has been compacted to use the same operators to process all the inputs deriving the architecture at figure 4.4.
The same algorithm has been replicated. It is not necessary to implement counters because the number of iterations to perform is well-known and equal to 8 (the last two rows of matrix $[A|b]$). The execution starts only when the block is enabled by $EN_{forward}$ signal; after that, the operations is managed by FSM states and, for each one, the selection signals pick proper inputs. For example, to update A_4 (the sequence of operation is reported at pag. 34):

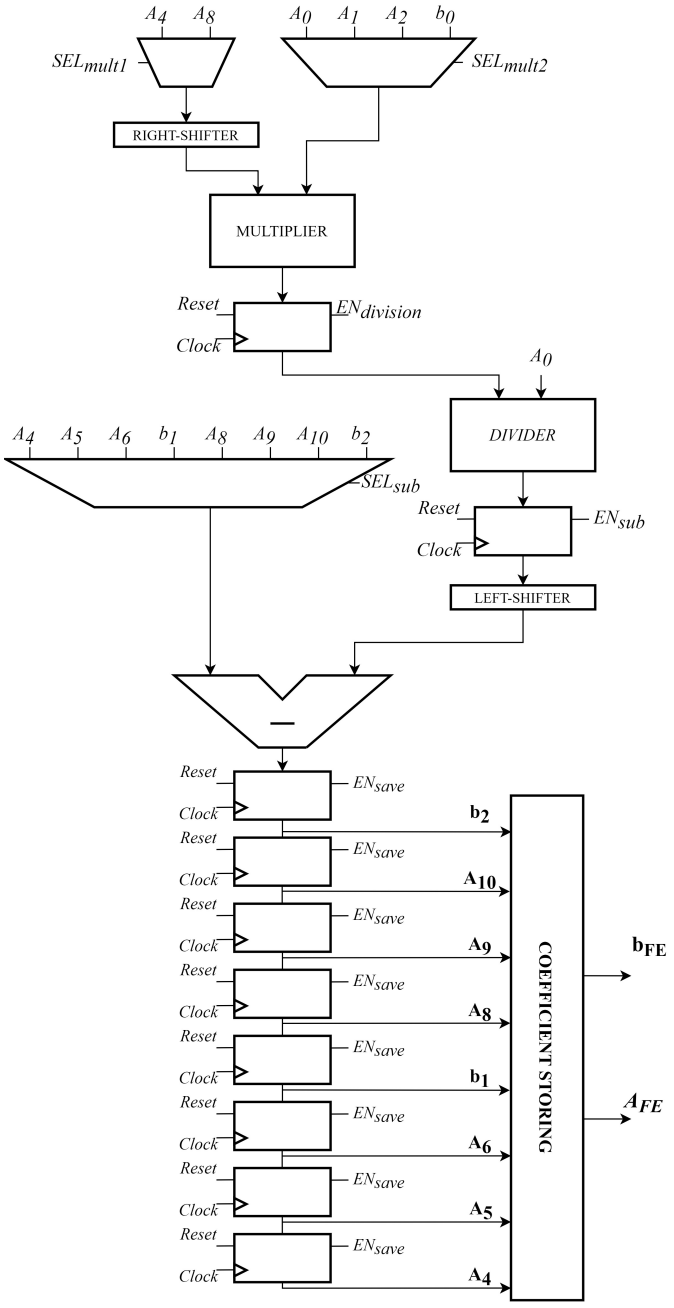
- **1^{st} multiplication** state selects the input A_4 from the multiplexer at the left and the input A_0 from the right one. The multiplier compute the operation

```
DIVIDEND <= shift_right(A4,8) * A0;
```

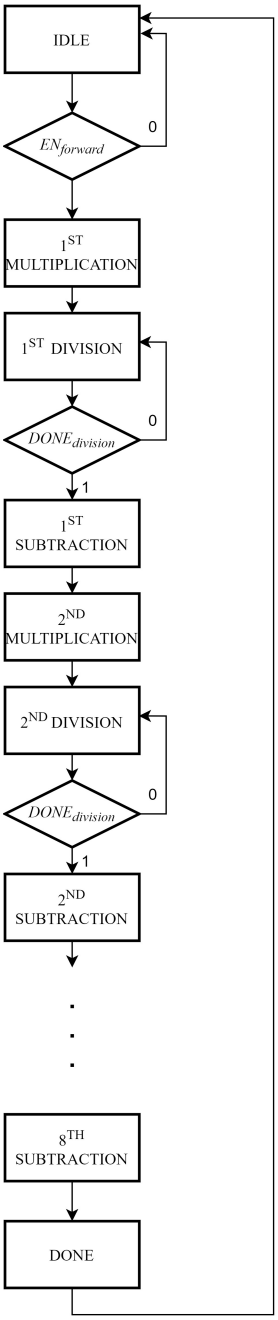
- **1^{st} division** state provides the result of previous operation to the divider by means of a register and it is divided by A_0.

- **1^{st} subtraction** state provides SEL_{sub} signal to pick A_4 and perform the operation

```
A4_FE <= A4- shift_left(QUOTIENT,8);
```



(a) Data Path for first stage of Forward Elimination



(b) FSM for first stage of Forward Elimination

Figure 4.4: Hardware implementation for first stage of Forward Elimination

Every time a division is executed, the flow is stopped waiting its completion. The output elements are derived neatly starting from the first of the second row A_4 and are stored consecutively in an 8 registers chain . Once terminated, the output A matrix and b vector are filled with the proper updated coefficients.

4.2.4 Second stage of Forward Elimination

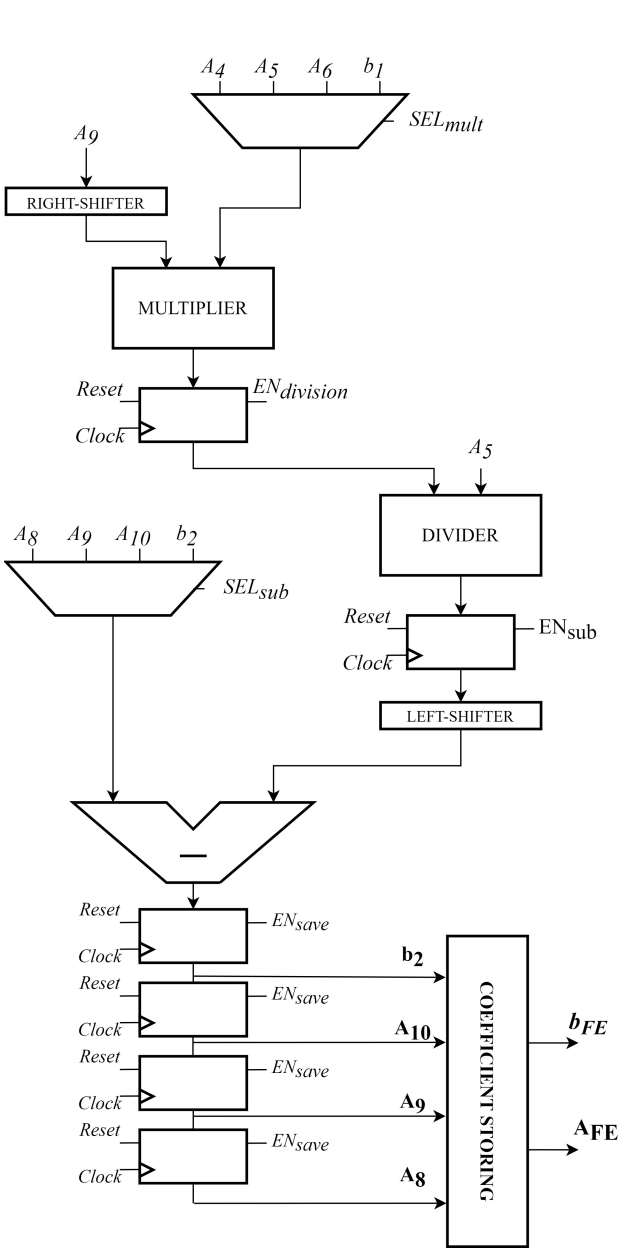
This elaboration is similar to the first stage, but involves only the last row of A matrix and the last element of b vector. Even if the number of computational blocks is lower, it is necessary to pay attention on this component since it contains expensive operator like divider. For that reason the same concepts are applied again to share the resources (figure 4.5).

It is possible to notice how the architecture is really similar to the one at figure 4.4. The mechanism is exactly the same apart from the distribution of input data inside the structure. The computed elements are cycle by cycle stored in a 4 registers chain and, after all, the output A matrix and b vector are filled.

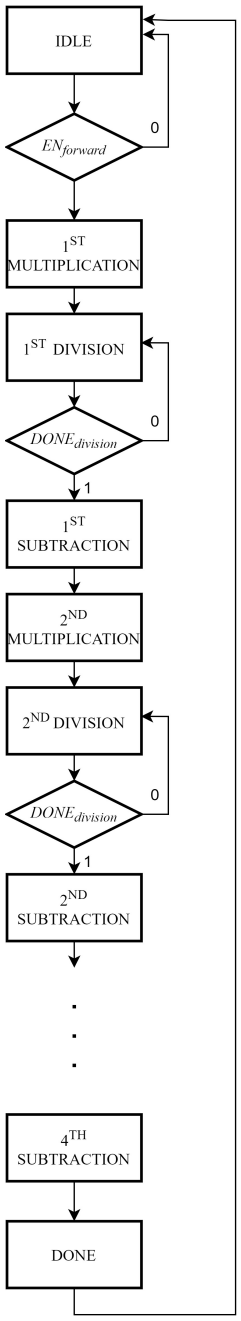
It is worth to notice that the whole *Forward Elimination* operation has been realized using just two multipliers, two dividers, two subtractors and 12 registers. This allows to reach a complexity reduction of a factor of about **19**.

Implementation	Area [µm^2]
Basic *Forward Elimination*	709903.197
Optimized *First stage of Forward Elimination*	19256.272
Optimized *Second stage of Forward Elimination*	18274.200
Total optimized *Forward Elimination*	37350.472

Table 4.5: Complexity comparison between *Forward Elimination*



(a) Data Path for second stage of Forward Elimination



(b) FSM for second stage of Forward Elimination

Figure 4.5: Hardware implementation for second stage of Forward Elimination

4.2.5 Back-substitution and storing

The **Back-substitution and storing** block presented at paragraph 3.1.1 is one of the least complex inside the whole architecture: it employees just 3 dividers, 3 multipliers and 3 adders to execute the algorithm. Despite this, the presented optimization leads to a further complexity improvement and at the same time split the critical path and, even if that's not the task that it has been designed for, it helps to relax the clock period constraints and decrease the critical path computed by the analysis of basic implementation. The designed architecture is reported at figure 4.6. The algorithm is mapped now in a quite more complex way, that is explained below:

- In **1st division** state, the output $\boldsymbol{x(2)}$ is computed by dividing the component b_2 selected by SEL_{div1} properly shifted and A_{10} selected by SEL_{div2} and it is stored in the first register of the chain below the divider.
- In **"1st multiplication** state, the multiplier is used to compute the operation $x(2) \cdot A_6$. The result is then right-shifted to derive C_1.
- In **1st subtraction** the difference from b_1 and C_1 is made and the result is the dividend to provide to divider for the next division.
- In **2nd division** state the output $\boldsymbol{x(1)}$ is computed by dividing the difference obtained at previous point and the coefficient A_5. To store the result, as made for $x(2)$, the chain of registers is enabled and so, at the next clock cycle, $x(1)$ will be available at the output of first register while $x(2)$ will be shifted on the second one.
- In **2nd multiplication** state, C_2 is derived exactly as C_1 handling the multiplexers control signals to provide x(1) and A_1 to multiplier.
- In **3rd multiplication** state, C_3 is obtained again repeating the same multiplication and shifting operation, providing as operands x(2) and A_2. At the same time C_2 is stored in the sum addend register.
- **"Sum"** state is used to perform the addition of C_2 available at the output of addend register and C_3.
- In **2nd subtraction** the subtractor is again used to provide the dividend to perform final division making the difference between b_0 and the sum $C_2 + C_3$ derived at previous step.
- In **Last division** state, the divider is exploited to compute the division between A_0 and the dividend provided from previous point deriving the last

output $x(0)$ that, by enabling again the chain of register, is properly stored. The final output configuration is the one reported at figure 4.6.

The number of resources has been reduced to one instance for each kind of operation and this leads to a reduction complexity of about **8**.

Implementation	Area [µm^2]
Basic *Back-substitution and storing*	193676.729
Optimized *Back-substitution and storing*	24587.444

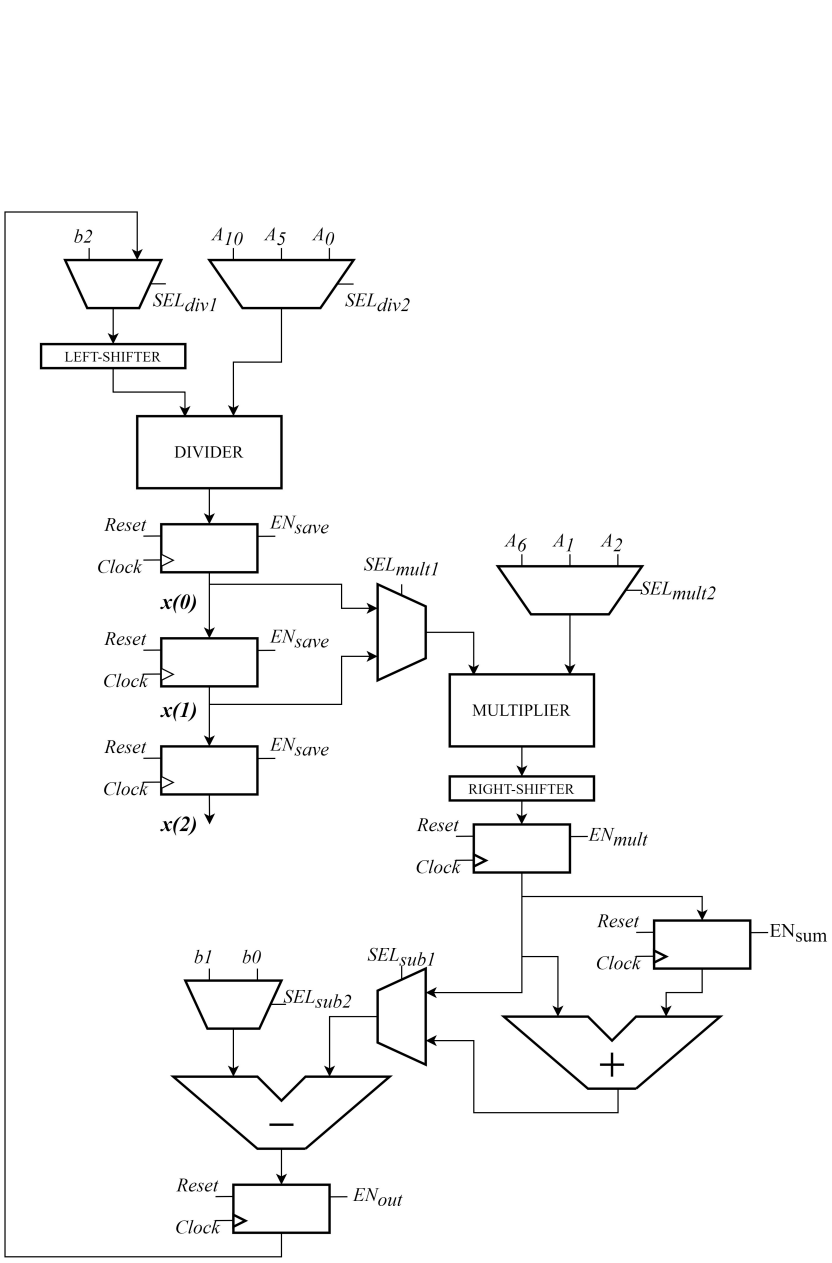
Table 4.6: Complexity comparison between *Back-substitution and storing*

4.2.6 Top-level

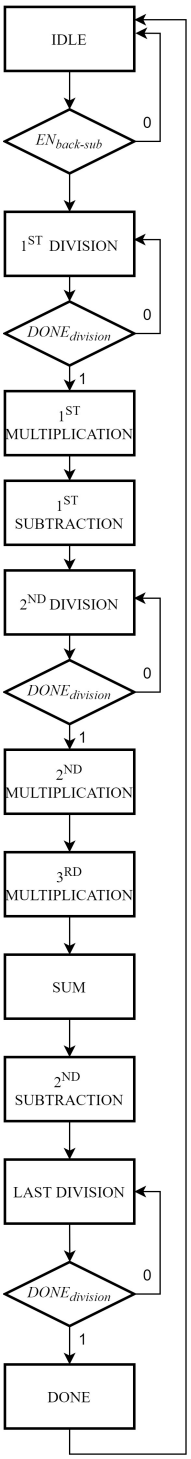
The design has been implemented with the idea of maintaining the same top-level architecture modifying the related FSM to adapt the new features of optimized components. The control of the execution flow is the most critical point because an optimization aimed to complexity reduction is more or less independent from timing behavior, but the only way to organize the algorithm development is to use an FSM scanning the design cycle by cycle. Differently from the basic combinatorial implementation where it was ensured that each involved component completed all the operations in one single clock cycle, this design is characterized by a clock cycles request that depends both on the number of operations to performs and on the type of involved operators. For that reason it is important to underline that, even if the block diagram and the mapped algorithm are exactly the same, the design behavior is completely different and it is managed by a more complex dedicated FSM as can be noticed at figure 4.7. As a consequence of that, also the register enabling is more difficult since they must store data, cycle by cycle, only after the related operation completion: this is why many enabling control state have been inserted. The design execution follows these steps:

- The mechanism that provide inputs is exactly the same than the one described for basic architecture handling *M Selection* and *H Selection* in the same way. When the inputs are available, both multiplicative blocks are enabled. Differently from basic implementation it is not possible to control each operation completion by using counters since each one needs to different clock cycles: each block provide a *done* signal when it completes the operations.

- The computation of each partial vector of a requires the elaboration of the 7 elements of sub-vector M_{in} provided by *M Selection* block. In order to



(a) Optimized *Back-substitution and storing* data path



(b) Optimized *Back-substitution and storing* FSM

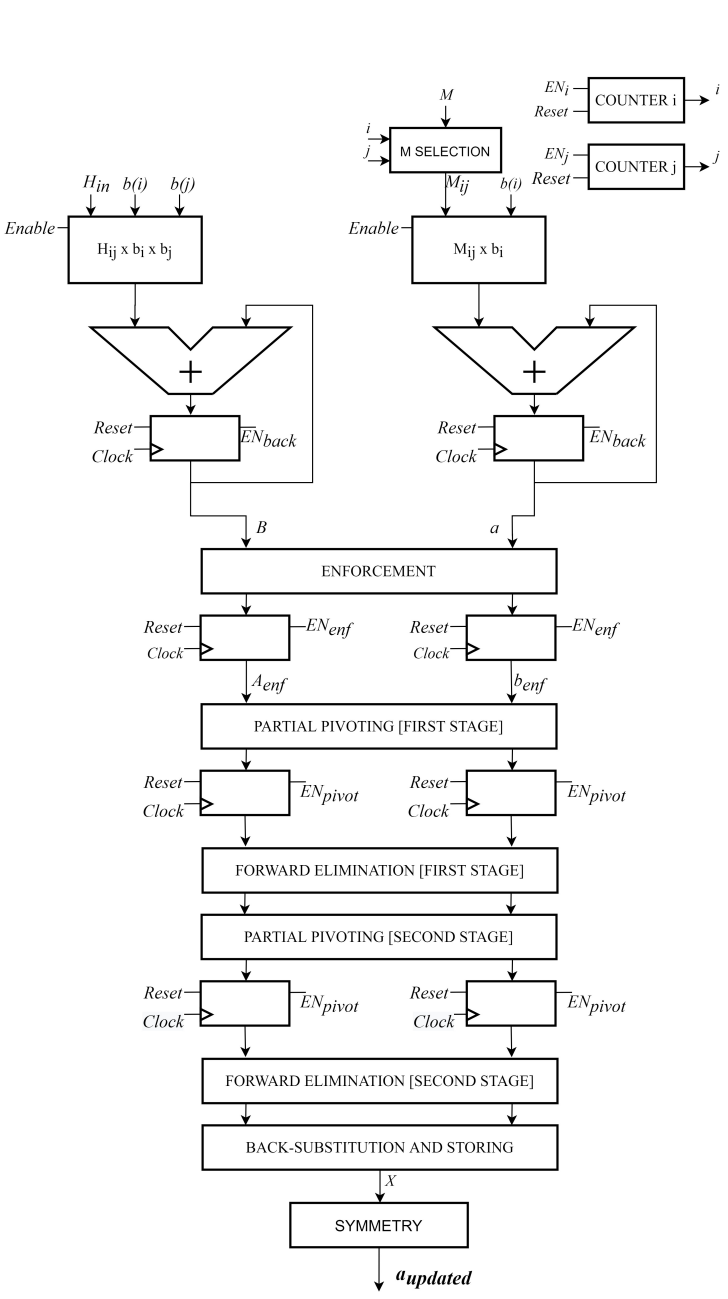
Figure 4.6: Hardware implementation for optimized *Back-substitution and storing*

synchronize the operations it is important to notice how the blocks H_{ij} x b_i x b_j and M_{ij} x b_i complete the respective operations with a different number of clock cycle since each B partial matrix is the result of an elaboration of 49 elements of H_{in} one for each clock cycle. On the contrary each a partial vector is produced by elaborating only 7 input elements of M_{in}. Both H_{ij} x b_i x b_j and M_{ij} x b_i are enable at the same time: since the a partial vector is produced earlier, when the component signals its completion with the signal $DONE_{M_{ij}xb_j}$, the result is stored using a feedback register. The subsequent a partial product is elaborated only after B partial matrix is produced and available to be stored. This operation is repeated for the first six cycles when all the partial values of a have been back-added and the final value of the vector is provided out from the feedback register.

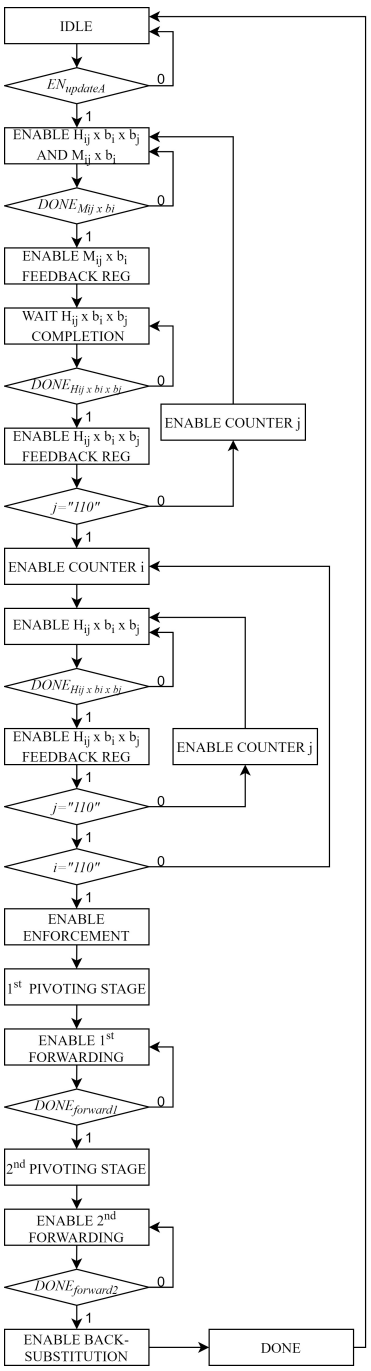
- Between the 6^{th} and the 48^{th} cycle(where $i = j = 6$), the derivation of B partial matrix continues as for the previous step until the final value is reached.

- Once obtained and then enforced both final value for a and B the system $A \cdot a = b$ is solved by repeat the same execution flow implemented for the basic implementation but enabling each operation only after the previous one is completed. Nevertheless,*Enforcement* and *Pivoting* block does not have any *done* because, not being optimized, it is ensured that the execution ends in one single clock cycle.

The vertical filter $\boldsymbol{a_{updated}}$ is taken by selecting only the first 32 bit of each component of the symmetrized output vector.



(a) Data Path for optimized *update a*



(b) FSM for optimized *update a*

Figure 4.7: Hardware implementation for *update a*

4.3 Hardware implementation of optimized *update b*

The concept is the same of the one applied for the basic implementation: executing the same algorithm means using the same *linsolve_wiener* function and as a consequence just the derivation of B matrix and a vector changes from *update a* design. From table 4.1 it can be noticed an analogy from the correspondent computation of a and B in terms of complexity and so the steps adopted to optimize them are similar to what explained in previous section.

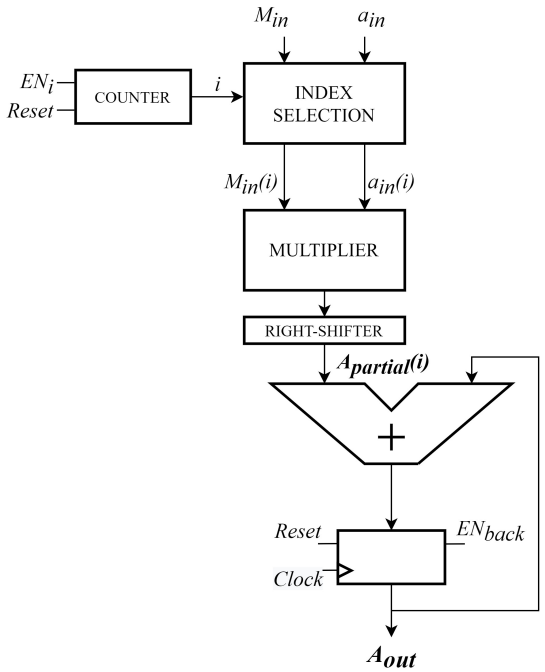
4.3.1 Computation of vector A

The optimized $\boldsymbol{M_{ij}}$ x $\boldsymbol{a_i}$ generates again one single output value for each input sub-vector M_{in}. While in the basic implementation case the execution was based on the "*variable*" syntax to perform the feedback sum, in the optimized one the only way to reduce the area request is to give up this method adopting a classic adder with feedback register, even if it involves a new local reset signal. Moreover, a further optimization is done on the multiplication numbers: each operation $M_{in}(i) \cdot a_{in}(i)$ is completed exploiting the same multiplier, instead of instancing seven. This means that while in basic architecture a *"for"* loop is used and all the multiplications were done in parallel, in the architecture reported at figure 4.8 the block *Index selection* provides proper component for M_{in} and a_{in} related to the i-th cycles according to the counter value enabled by the FSM state **Enable counter i**. Every time the i-th operands are provided to multiplier the feedback register will sum the contribution $i-1$ and, after the 6^{th} clock cycle, the value of $A_{partial}$ related to the sub-vector M_{in} is released as output.

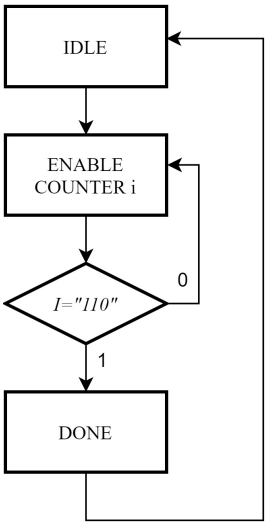
The obtained architecture completes the operation using just one counter, one multiplier, one adder and one register allowing a complexity reduction of a factor close to **10**.

Implementation	Area[µm^2]
Basic M_{ij} x a_i	76808.298
Optimized M_{ij} x a_i	7736.876

Table 4.7: Complexity comparison between M_{ij} x a_i



(a) Optimized M_{ij} x a_i data path



(b) Optimized M_{ij} x a_i FSM

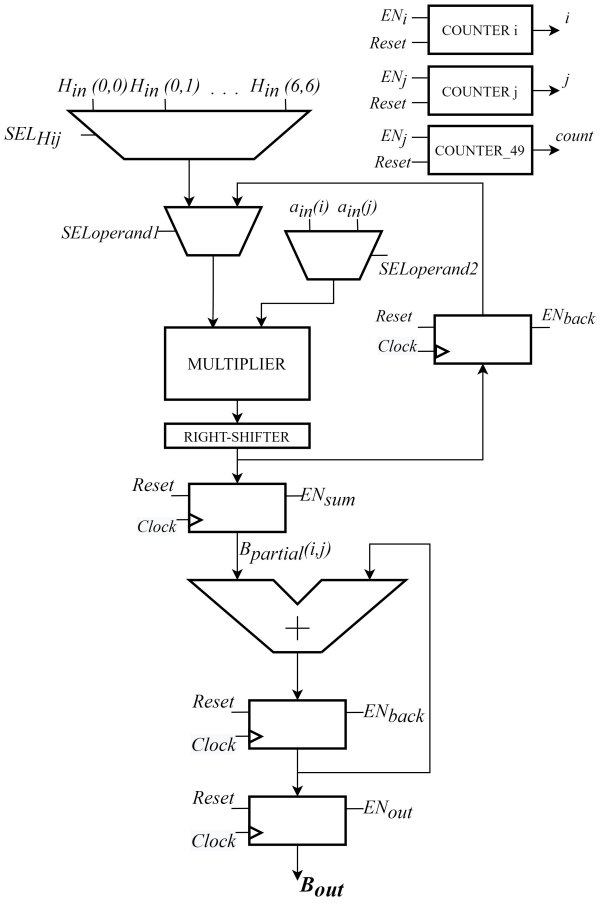
Figure 4.8: Hardware implementation for optimized M_{ij} x a_i

4.3.2 Computation of matrix B

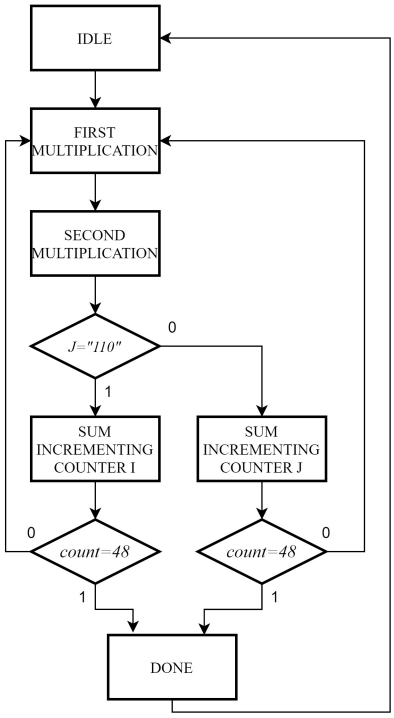
The optimization for $\boldsymbol{H_{ij}}$ **x** $\boldsymbol{a_i}$ **x** $\boldsymbol{a_j}$ is quite more complex than the one explained for A vector because it must process the 49 elements of a 7x7 sub-matrix of H. Its basic implementation reported at paragraph 3.1.2 needs for 2 multiplication, 2 shifting and a feedback sum for each processed input. The adopted way to share resources is similar to the one exploited in H_{ij} x b_i x b_j: in particular, the first set of multiplier performs the multiplication between the (i, j) element of sub-matrix H_{in} and the i-th element of input vector a. The second one computes the previous result multiplying it with the j-th element of a. The complexity has been reduced to the following components:

1. one single instance for multiplier;
2. a 49-to-1 multiplexer selecting the proper element of H_{in};
3. a 2-to-1 multiplexer that selects the operand coming from input vector a.
4. a 2-to-1 multiplexer to select which set of multiplication have to be performed.

The output storing algorithm is similar to the basic implementation because again just one value is produced instead of a whole matrix: a feedback adder is exploited. The final architecture is reported at figure 4.9



(a) Optimized H_{ij} x a_i x a_j data path



(b) Optimized H_{ij} x a_i x a_j FSM

Figure 4.9: Hardware implementation for optimized H_{ij} x a_i x a_j

The design requires the employment of three counters:

- *Counter i* and *Counter j* are exploited to select respectively i-th and j-th component of input vector a_{in};
- *Counter_49* is used to select the proper component of input matrix H_{in}

The selection signals are provided in *"First Multiplication "* and *"Second multiplication"* state as follows:

- In **First Multiplication** the multiplier is used to compute $H_{in}(i,j) \cdot a_{in}(i)$, so:
  - SEL_{Hij} is exploited to scan the 7 x 7 matrix horizontally and vertically by using the value of *count* signal. It is updated in the range $(0-48)$ and the selected element is extract sorting the matrix from left to right:

$$\begin{bmatrix} H_0 & H_1 & H_2 & H_3 & H_4 & H_5 & H_6 \\ H_7 & H_8 & H_9 & H_{10} & H_{11} & H_{12} & H_{13} \\ H_{14} & H_{15} & H_{16} & H_{17} & H_{18} & H_{19} & H_{20} \\ H_{21} & H_{22} & H_{23} & H_{24} & H_{25} & H_{26} & H_{27} \\ H_{28} & H_{29} & H_{30} & H_{31} & H_{32} & H_{33} & H_{34} \\ H_{35} & H_{36} & H_{37} & H_{38} & H_{39} & H_{40} & H_{41} \\ H_{42} & H_{43} & H_{44} & H_{45} & H_{46} & H_{47} & H_{48} \end{bmatrix}$$

    So, step by step, the value H_{count} comes out from the input matrix.
  - $SEL_{operand1}$ selects as first operand data coming from the matrix multiplexer providing to multiplier $H_{in}(i,j)$.
  - $SEL_{operand2}$ provides $a_{in}(i)$ as second operand
- In "Second Multiplication" state:
  - SEL_{Hij} is neglected since the matrix element has already been used.
  - $SEL_{operand1}$ selects as first operand data coming from the feedback register, providing to multiplier the shifted result of the previous calculation.
  - $SEL_{operand2}$ provides $a_{in}(j)$ as second operand.

After each second stage of multiplication a partial value is produced and it is summed to the contribution computed at previous clock cycle. For that operation it is necessary to distinguish two cases:

- If $j < 6$ it means that the scan of whole a_{in} is not completed, so the subsequent sum have to be exploited incrementing only the j counter to move in the same row.
- If $j = 6$ the whole vector has been scanned. In that case the counter i have to be incremented and counter j reset to move on to the next row and start again the cycles of 6 iterations. In both cases, the process ends when the total number of iterations is equal to 49 and so when *counter_49* reach a value of 48.

The execution is performed used only 2 expensive component: one adder and one multiplier, that compared with the 96 multipliers and the "variable" feedback sum exploited in the basic design allows a reduction of complexity of a factor of about **85** paying just some additional multiplexers.

Implementation	Area[µm^2]
Basic H_{ij} x a_i x a_j	1034442.881
Optimized H_{ij} x a_i x a_j	12443.214

Table 4.8: Complexity comparison between H_{ij} x a_i x a_j

4.3.3 Top-level

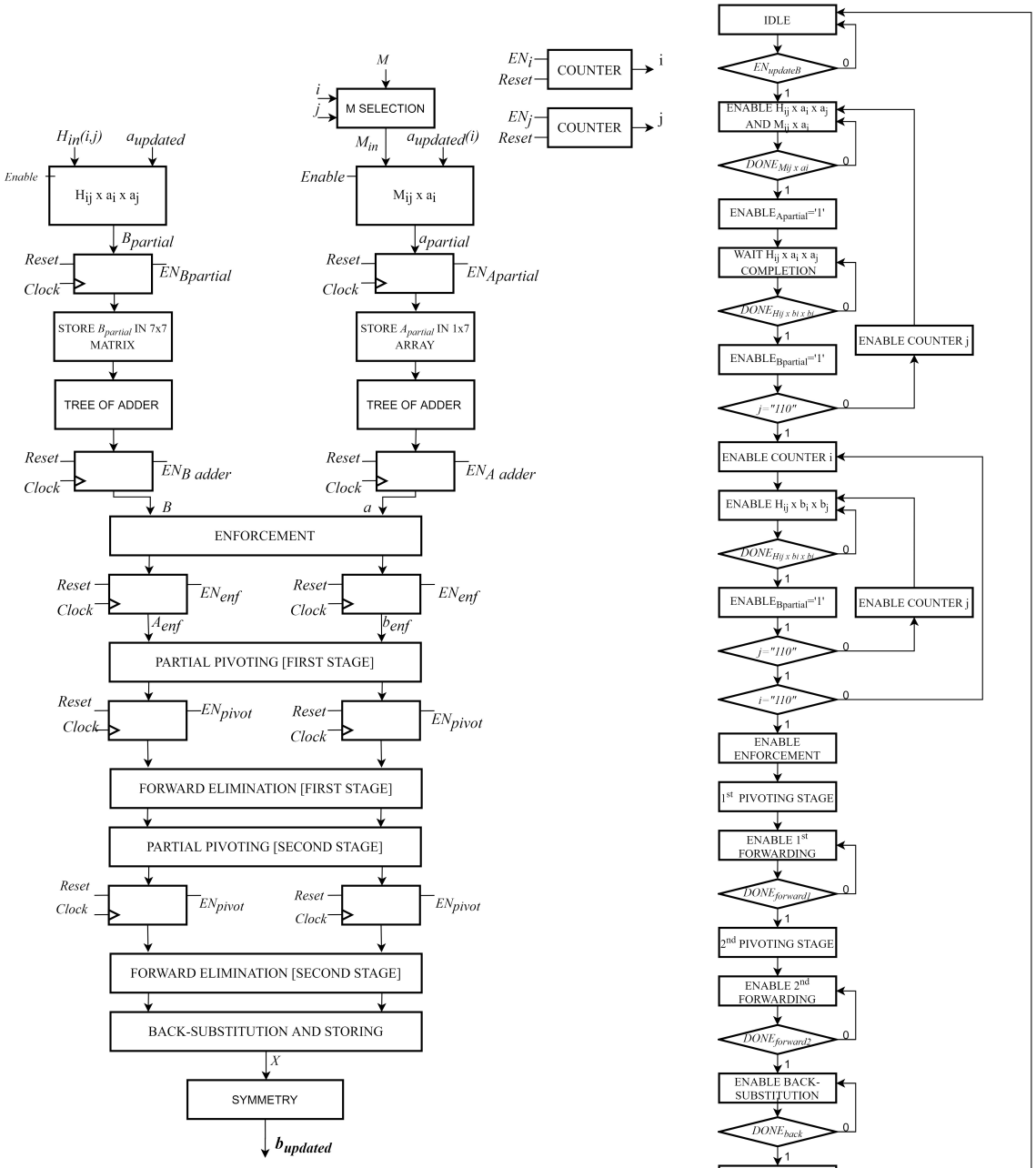
The top-level architecture for the optimized horizontal filter is derived from the same design choices that is again summarized below:

- Each sub-vector of M is provided by an *M selection* instance based on the counters values;

- The sub-matrix H_{in} is provided directly from the testbench. This solution has become even more powerful in the optimized implementation since *H selection* block would have required an huge area occupation without any possibility to compress it.

- The only difference compared with the optimized version of the *Update a* design is about the storing. In fact the feedback adder has been replaced by the same trees of adder of the basic *Update b* implementation. Of course from a computational point of view it might seems a contrast choice but it isn't: once filled the partial structures both for B and a, optimizing the storing algorithm would have meant to exploit just one adder to see a significant reduction of complexity and then perform one operation for each clock cycle. The already critical latency would have increased more of a large factor. In this case ,paying the employment of some more adders, the design is surely more balanced.

- All the steps from the *Enforcement* are repeated exactly as done for vertical filter *a*. It is possible to notice how a further complexity reduction could be obtained shari this architectural section between both *Update a* and *Update b*, with perhaps a proper selection signal. This step is skipped in this presentation to underline, block by block, the complexity contribution and the improvement related to same top-level architecture.



(a) Data Path for optimized *update b*

(b) FSM for optimized *update b*

Figure 4.10: Hardware implementation for *update b*

The final designed architecture is reported at figure 4.10.

The execution is controlled by the FSM that, how it can be easily noticed, it is very similar to the one exploited at figure 4.7. The only difference consists in the storing enable operation: differently from *Update a* where it was necessary to control a feedback register, in this case the partial value for both B and a is stored in a 7x7 matrix and a 1x7 vector respectively only when the related input register is enabled in the state "$\boldsymbol{Enable_{Bpartial}}$" and "$\boldsymbol{Enable_{apartial}}$".After the last cycle $i = j = 6$ the execution proceed solving the linear system as explained and producing the updated value of filter b taking the first 32 bit of each component of the symmetrized output vector.

Chapter 5

Results and Synthesis

In order to analyse optimized architecture of the Wiener Filter in terms the same top-level architecture reported at figure 3.16 controlled by related FSM at figure 3.17 has been exploited, in order to define the amount of improvements introduced with the low-complexity optimization. The design analysis has been performed in three steps:

1. Simulation of the optimized architecture using *Modelsim* to verify if it works properly;

2. Synthesis of the working architecture using *Synopsys* to derive whole design complexity informations ;

3. Comparison between optimized and basic architecture complexity.

5.1 Optimized design simulation

The basic idea was to reuse significant data already extracted from basic design simulation and then verify if the results obtained by the new implemented architecture were equals to the ones derived from software execution of *.yuv* file *"Flower-vase_832_480_30.yuv"* . This choice allows to provide the same inputs extracted from the software analysis also for this optimized design and make the results comparison easier. The simulation is performed again using a *Modelsim test-bench* where:

- Both input matrices H and M and vector b_{in} is provided by process statement.

- *H selection* block is instanced to avoid its synthesis providing it the counter i and j taken from Wiener Filter top-level instance.

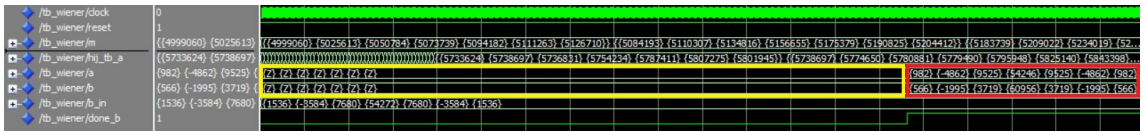


Figure 5.1: Modelsim simulation for optimized Wiener Filter

- The provided clock period is equal to the one used in the basic design, in order to be able to quantify the latency increment.

The relative *Modelsim* simulation screen is reported at figure 5.1.

The screen above is extracted with the same horizontal resolution of the one at figure 3.18 related to the basic architecture, and what turns out immediately is that the output buffers stay at High-Impedance state for longer time because the signal $Done_b$ is released later: as expected the latency parameter is increased a lot. Both the output 7-elements of horizontal and vertical filter are equal to the ones derived in the basic implementation.

5.2 Optimized design synthesis

Differently from the basic implementation where the synthesis was used to trace the optimization field, this one aims to evaluate gives an idea on the achievement of the objectives set. Hereinafter only the performances in terms of complexity will be discussed since timing evaluation is out from the optimization goal. Nevertheless this kind of improvement probably leads to a reduction of critical path delay because the insertion of many registers have the effect of splitting the critical path easing time constraints. Obviously this consideration have to be related to latency huge increment to evaluate properly the total time spent to finish processing.

To analyse the area occupation the command *report_area* is typed again and the results are reported in the table below:

Number of ports	242670
Number of nets	482942
Number of cells	193866
Number of combinational cells	163017
Number of sequential cells	30051
Number of buf/inv	29469

Table 5.1: *report_area* results of optimized implementation

The total occupied area is given adding both combinational and noncominational contribution:

Contribution	Area [µm^2]
Combinational Area	253727.824
Non-combinational area	147411.884
Total area	401139.709

Table 5.2: Total area of top-level optimized design implementation

5.3 Elaboration for a real-time video sequence

The evaluation on the behavior of the proposed optimized architecture for the elaboration of real-time application is reported considering the same standard video sequences used for the basic implementation. This kind of analysis deviates from the low-complexity purpose because the improvement of the elaboration speed would have required a different type of optimization, focused on the throughput improvement inserting properly pipeline registers. Even if this proposed design relaxes the timing constraints decreasing critical path with the inclusion of folding registers, the results should be measured carefully because of the latency.

For sake of completeness, the results related to the maximum *fps* reachable from the low-complexity optimized architecture will be reported and compared with the basic implementation, even if its throughput is not indicative of speed performances.

Application	Resolution [pixels x pixels]	*fps* [Basic Architecture]	*fps* [Low-comlexity]
SD [480i]	720 x 480	**12**	**317**
SD [576i]	720 x 576	**10**	**264**
HD [720p]	1280 x 720	**4**	**119**
HD [1080p]	1920 x 1080	**2**	**52**

Table 5.3: Video sequences fps comparison

5.4 Results comparison

The table below summarizes the main results obtained comparing the reports for both the implementation:

Report data	Basic design	Low area design	Reduction Factor
Number of ports	1245024	242670	5.1
Number of nets	5016113	482942	10.4
Number of cells	3568398	193866	18.4
Number of combinational cells	3539633	163017	21.8
Number of sequential cells	22044	30051	0.73
Number of buf/inv	862746	29469	29.2

Table 5.4: Reports contribution comparison

It is worth to notice that the reduction factor confirms clearly what expected from the design choice: it turns out easily how the contribution given by the *combination cells* is one of the most reduced because the optimization is based on a drastic abandonment of the combinatorial nature of basic design. To obtain that, a step-by-step employment of several registers to save properly computed data was needed: this is the reason why the contribution related to *sequential cells* has increased even for a small quantity. To quantify combinatorial and non-combinational trend both contributions are compared in the following table:

Contribution	Basic design[μm^2]	Low area design[μm^2]	Reduction Factor
Combinational Area	4132550.177	253727.824	16.3
Non-combinational area	80061.746	147411.884	0.54

Table 5.5: Combinational and sequential contributions comparison

What emerges is that:

- A massive usage of folding technique leads to a very high hardware resources sharing. For that reason the area required by combinatorial block is decreased of a factor **16.3**.
- The mentioned increment of storing components to complete folding approach leads to a little complexity rise and the consequent area request from non combinatorial block became about the double.

The main relevant comparison concerns the total occupied area:

Contribution	Basic design [µm^2]	Low area design [µm^2]	Reduction Factor
Total Area	4212611.924	401139.709.824	10.5

Table 5.6: Total complexity comparison

The total complexity has been reduced of about **90%**.

Chapter 6

Conclusions

The continuous increment of video applications diffusion lead to the need of improving as much as possible the coding mechanism, in order to be able to process high resolution videos in a very efficient way. The concept of efficiency is the key point of this presented treatment: of course a special purpose hardware implementation helps to manage properly some critical point increasing the processing quality. As mentioned many times before, even if the main objective of the presented architecture is to reduce the complexity, it is necessary to make the design reliable, reaching good value also for throughput or latency. This is why, even reaching a reduction of complexity of about 90%, it is possible to further improve it by those who will continue this work. For example, an increment of folding degree would reduce more the complexity, but it is necessary to take more into account the latency issue. Just to give an idea, this level of optimization produce a latency increment of a factor close to 100 measured by simulating both designs with the same clock period and obtaining the correspondent amount of clock cycle spent to produce the output. Anyway this parameters doesn't consider that the complexity optimization make the design faster reducing the minimum clock period. Considering the amount of time spent to release the output for each implementation and evaluating the **Total Time Increment Factor** like

$$TTIF = \frac{T_{ck}(basic) \cdot N_{cycles}(basic)}{T_{ck}(optimized) \cdot N_{cycles}(optimized)} \tag{6.1}$$

the expected value will be much smaller.

In this panorama the obtained results can be considered very powerful because not only they record a good improvement, but it is a good starting point for further optimization in the same direction like:

- Redesigning every involved operators (multipliers above all) building a resource sharing algorithm for each one;

- Optimizing also the already low complex component;
- Optimizing storing algorithm exploiting it in several clock cyle;
- Designing a different way to implement the inputs distribution avoiding to store the whole matrix structures and provide for each clock cycle only the needed input vector;
- Sharing all the components mapping the *linsolve_wiener* function since they are in common for both *Update a* and *Update b*.

Bibliography

[1] Online available at : *https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html*

[2] Online available at: *https://research.mozilla.org/av1-media-codecs/*

[3] Online available at: *https://www.w3.org/Consortium/Patent-Policy-20040205/*

[4] Online available at: *https://caniuse.com/#feat=av1*

[5] Feldmann C.: "Multi-Codec DASH Dataset: An Evaluation of AV1,AVC,HEVC and VP9 – Bitmovin".

[6] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, Ching-Han Chiang, Yunqing Wang, Paul Wilkins, Jim Bankoski,Luc Trudeauy, Nathan Eggey, Jean-Marc Valiny, Thomas Daviesz, Steinar Midtskogenz, Andrey Norkinx and Peter de Rivaz: "An Overview of Core Coding Tools in the AV1 Video Codec", *2018 Picture Coding Symposium (PCS)*,San Francisco, 2018

[7] Laude T.; Adhisantoso Y.G.; Voges J.; Munderloh M.; Ostermann J.: "A comparison of JEM and AV1 with HEVC: coding tools, coding efficiency and complexity", *Picture Coding Symposium (PCS)*, San Francisco, 2018

Online available at : *https://bitmovin.com/av1-multi-codec-dash-dataset/*, 2018

[8] Online available at: *https://aomedia.googlesource.com/aom/*

[9] L. N. Trudeau, N. E. Egge, and D. Barr: “Predicting chroma from luma in AV1”, Data Compression Conference, 2018.

[10] Jan M. Rabaey : "Digital Integrated Circuits – A Design Perspective", University Press, 2004