



Technical University Of Crete

Department of Electronic and Computer Engineering

Microprocessor and Hardware Laboratory (MHL)

**Diploma thesis : “*Novel memory systems
tailored to vision algorithms*”**

Author : Farantos Georgios

Committee

Supervisor : Assistant Professor Yannis Papaefstathiou

Professor Apostolos Dollas

Professor Michalis Zervakis

Abstract

Nowadays, one of the most important challenges in the field of Computer Vision and the target of all the engineers that are associated with this, is the design of embedded systems that will execute real-time algorithms with big accuracy and low cost [2]. This effort meets a lot of difficulties as the algorithms that have already been applied “suffer” by bottlenecks, especially as far as that concerns the memory accesses.

In this thesis, we deal with a novel embedded system, which implements the OpenTLD algorithm and can be applied at a broad-range of algorithms based on the generalized Viola and Jones framework. Viola and Jones framework, despite the fact that is hardware friendly in resource-limited devices (FPGAs), the way it access the memory causes certain problems and parallelism is not easy to be applied on it. This is mainly due to the fact that it is a memory bound problem and its memory access pattern has certain characteristics that make it hard to take advantage of the conventional memory hierarchies. Our system accelerates the bottleneck of the algorithm with a high bandwidth distributed memory sub-system which is independent of the various software parameters [1].

Thus, it was of high interest, the effort to try this embedded system on an algorithm out of Viola and Jones framework and study the result. That is the second part of the thesis. The first one is to optimize the first edition of the system in order to be more efficient and easier as concerns the communication with different algorithms.

Είναι μακριά απο μας τα άστέρια
μακριά - μακριά, πολύ - πολύ μακριά.
Ανάμεσα στ' αστέρια η γή μας είναι μια κουκκίδα
μια τόση δα κουκκίδα,
και η Ασία το ένα πέμπτο είναι της γής μας.
Μια χώρα της Ασίας είναι οι Ινδίες,
μες τις Ινδίες μια πόλη είναι η Καλκούτα.
Ο Μπεναρτζή δεν είναι παρά μοναχά ένας άνθρωπος μες τη Καλκούτα.

Και να τι θέλω τώρα να σας πώ:
Μές τις Ινδίες, μέσα στη πόλη της Καλκούτας
φράζαν το δρόμο ενός ανθρώπου,
αλυσοδέσαν ενα άνθρωπο που βάδιζε.
Νάτο λοιπόν γιατί δεν καταδέχομαι
να υψώσω το κεφάλι στα αστροφώτιστα διαστήματα.
Θα πείτε τα άστρα είναι μακριά
κ' η γή μας τόσο δα μικρή.

Ε το λοιπόν ότι κι αν είναι τ' άστρα
εγώ τη γλώσσα μου τους βγάζω.
Για μένα το λοιπόν,
πιό εκπληκτικό και πιο επιβλητικό
και πιο μυστηριακό και πιο μεγάλο
είναι ένας άνθρωπος που τον εμποδίζουν να βαδίζει
είναι ένας άνθρωπος που τον αλυσοδένουν

Nâzım Hikmet, "Μικρόκοσμος"
Απόδοση: Γιάννης Ρίτσος

Special thanks to

- ***Antonis Nikitakis for the cooperation, the support and his patience***
- ***The professors : Ioannis Papaefstathiou, Apostolos Dollas, Michael Zervakis, for their comprehension in deadline subjects and their ability in make me thinking and be organized as an engineer.***
- ***Marios for make me handling mechanical problems in a different way***
- ***Postgraduate students Panos and Aggelos for their support in Parts Based Detector Algorithm***
- ***My family for total support during studies***

Table of Contents

1	Chapter - Introduction.....	7
1.1	Computer Vision.....	7
1.2	Embedded Systems	8
1.3	Viola – Jones object detection framework.....	9
1.4	Memory access and object detection algorithms	11
1.5	Related work.....	12
2	Chapter – The embedded Sub – System	13
2.1	A general approach	13
2.1.1	The four modules	13
2.1.2	Computation core.....	14
2.1.3	The Memory module	14
2.1.4	Collision Detect module	15
2.1.5	Loop Decoding module.....	15
2.2	Structure and functionality of each part	16
2.2.1	Loop Decode Module	16
2.2.2	Collision Detect Core	17
2.2.3	Memory	19
2.2.4	Computation Core	20
2.2.5	Top Level.....	21
2.3	Conclusions.....	22
3	Chapter – Optimizations on the complete system.....	23
3.1	Use of 32 Collision Detect Cores	23
3.2	Handling sets of 32 addresses	25
3.3	The duplication of 32 cores and the ability of 64-address-queries.....	25

3.4	Optimization of Collision Detect Core’s FSM – The two configurations	26
3.5	Use of FIFOs – A permanent supply of addresses	31
3.6	Generating pseudo-random addresses	32
3.7	Use of delay registers	33
3.8	Total system control	33
3.9	Performance evaluation	35
4	Chapter – Crossing the border of Viola – Jones Framework	36
4.1	General Description of Parts Based Detector Algorithms – The “Face Detection, Pose Estimation, and Landmark Localization in the Wild” Algorithm	36
4.2	The bottleneck and the parallelization.....	37
4.2.1	The bottleneck.....	37
4.2.2	Parallelization	38
4.3	Vivado - High Level Synthesis	39
4.3.1	The C code	39
4.3.2	The VHDL code	40
4.4	The Embedded System tailored to “Face Detection, Pose Estimation, and Landmark Localization in the Wild” Algorithm.....	41
4.4.1	Implementation of filter	42
4.4.2	Multiplications and Additions in Computation Core	43
4.4.3	Top Level.....	44
4.5	Measurements – results.....	45
4.5.1	Vivado’s report	45
4.5.2	The simple specialized sub - system.....	46
4.5.3	The effect of scrambling function	47
4.5.4	Use of cache	48
4.6	Performance	49
4.7	Conclusion	49
5	References	50

1 Chapter - Introduction

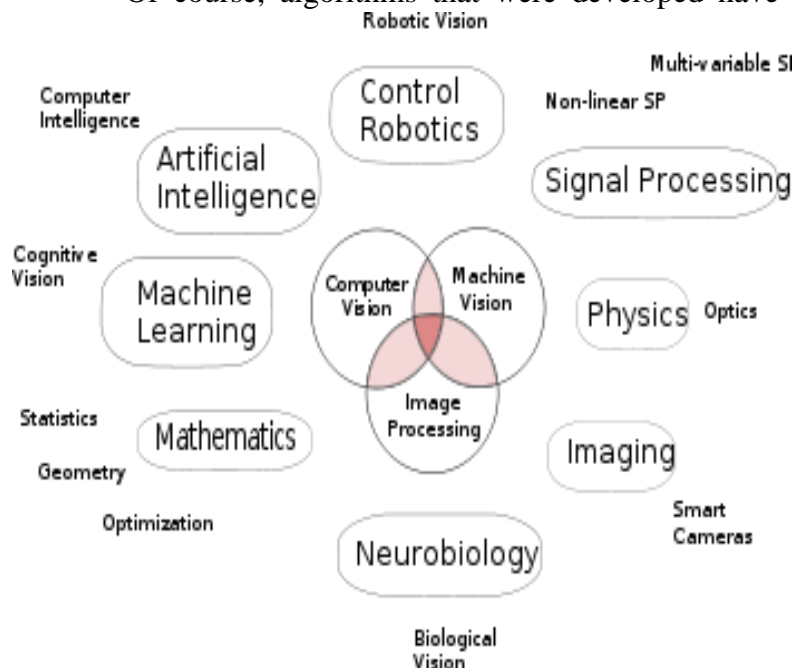
This is a very important part, as it introduces as with the main concepts of this thesis and describe subjects like computer vision science, embedded systems, Viola and Jones framework and algorithms that the researcher or the common reader of this report should be informed.

1.1 Computer Vision

Computer vision is a field that includes methods for acquiring, processing, analyzing and understanding images and that enables machines to solve particular tasks, like detecting, identifying and tracking objects or make decisions by extracting information from those images. The goal of computer vision is to design computer systems that are capable of performing these tasks both accurately and efficiently at real-time and with low consume of energy while using limited resources, especially in embedded environments[1][4].

Sub-domains of computer vision include scene reconstruction, event detection, video tracking, object recognition, motion estimation and image restoration. Object recognition, the classical problem in computer vision and image processing is that of determining whether or not the image data contains some specific object, feature, or activity. At motion estimation, an image sequence is processed to produce an estimate of the velocity and the direction of a subject. As regard as scene reconstruction, it aims at computing a 3D model of the scene, given one or (typically) more images of a scene, or a video [4]. Those were only a small sample of what computer vision deal with.

Of course, algorithms that were developed have a wide application range.



Examples of applications of computer vision include systems for: controlling processes (e.g. an industrial robot), navigation (e.g. an autonomous vehicle or a mobile robot), detecting events (e.g. for visual surveillance or people counting), modeling objects or environments (e.g. medical image analysis or topographical modeling), automatic inspection (e.g. in

Figure 1.1 Relation between computer vision and various other fields

manufacturing applications)[4].

In this thesis we deal, generally, with object detection and in the second part, more specifically, with face detection. Although, it will not concern us the way (the algorithm) in which we detect objects that may appear in an image, but how exactly we will face the multiple and continuous accesses in memory that are necessary for the completion of the algorithm.

1.2 Embedded Systems

It is true, that somebody could describe an embedded system in many different but also valid ways. “A system that is specialized” or “any system which does a fixed number of predefined set of tasks with the deadlines” or “A system designed to achieve certain application. Get some particular input and perform some particular active” could be some of them [5]. Thus, a specific and maybe strict definition of that term would be “unfair”. However, we can provide a general description to cover all the aspects of the theme.

An embedded system is a computer system with a dedicated function, which is included in a bigger mechanical or electrical system. Its main characteristics are the real – time implementation, the low – power consumption and the small size. Furthermore, they usually are computational, networked as they may use information from the net, reactive at the speed of the environment, heterogeneous as they may combine hardware and software and they interact with the external world (use of sensors or actuators) [5][6].

Embedded systems have an enormous spread, nowadays, and are used in numerous systems and activities by million people even it is not always noticeable.

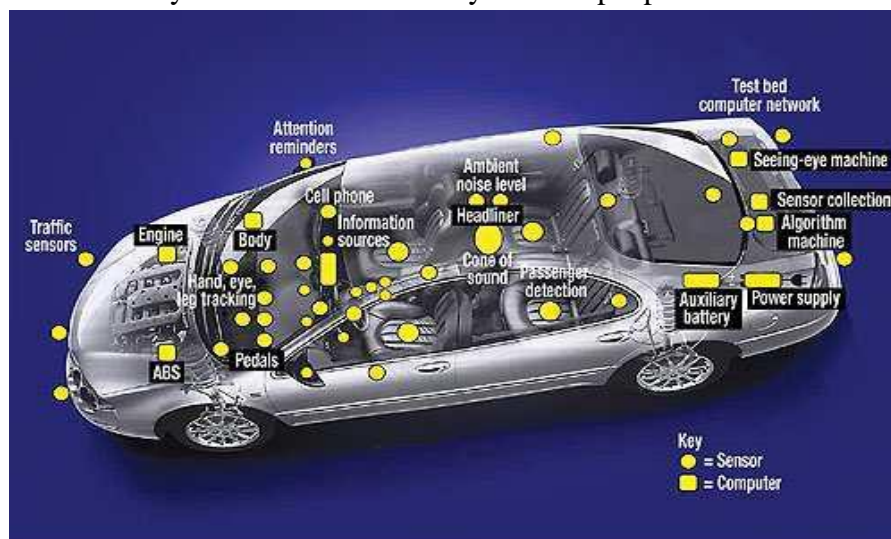


Figure 1.2 Modern cars rely on few embedded systems

machines and dishwashers (Figure 1.3). Medical equipment uses embedded systems to appear vital signs in monitors, accomplish medical examinations and act non-invasive internal inspections [6].

At the factor of electronics we meet PDAs, mp3 players, mobile phones, GPS receivers etc.

Many households use microwave ovens, washing



Figure 1.2 Embedded systems are... everywhere

Our embedded system is planned to speed up vision algorithms of the Viola – Jones family and not only, based on a smart use of memory parallelism.

1.3 Viola – Jones object detection framework

The Viola–Jones object detection framework has been the first object detection framework which provided competitive object detection rates in real-time. It was proposed in 2001 by Paul Viola and Michael Jones. Although it can be trained to detect a variety of object classes, it was motivated primarily by the problem of face detection [3]. Being a popular approach that somebody may find in the literature, it is adopted in numerous embedded systems that concern face detection, eyes detection, pedestrian detection and many more applications under the open computer vision library (OpenCV)[\[1\]\[3\]](#).

The framework uses various feature types that have a common characteristic. They rely on more than one rectangular area that includes the sums of image pixels

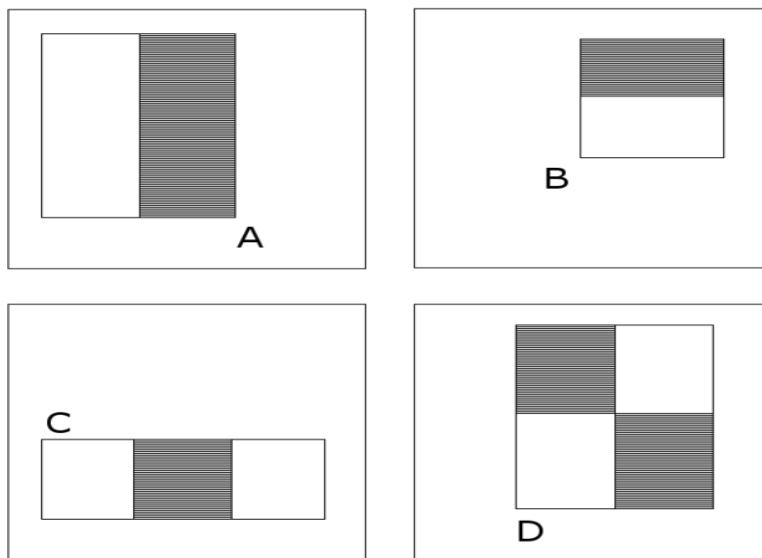


Figure 1.3 Feature types used by Viola and Jones

within. [Figure 1.4](#) shows the different types of feature that are used by Viola – Jones. The value of any given feature is always simply the sum of the pixels within clear rectangles subtracted from the sum of the pixels within shaded rectangles. Observing the figure, we can imagine that they are

sensitive to horizontal and vertical features.

The key characteristic, although, is the use of an image representation called “integral image”. With its use rectangular features can be evaluated in constant time, something that gives speed advantage. At integral image, the value of a point (x, y) is the sum of all the pixels above and left of it [7]. The equation is in [figure 1.5](#).

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y')$$

Figure 1.5 Integral image pixels' equation

At time we have already computed the integral image, any rectangular can be accomplished, only with the use of four parameters, the square tops. Let's call them A, B, C, D. The rectangular is the simple sum $D - B - C + A$. It is easily noticed how powerful the integral image is, as with the concerned image saved in a memory, we need only four accesses and three additions to get to the result.

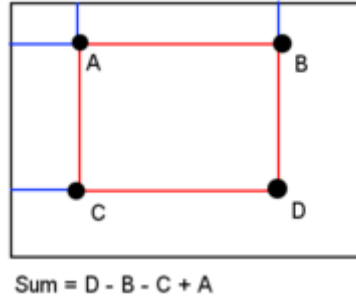


Figure 1.6 A description of computing a sum in the Integral Image

An example of such an algorithm is SURF. Speeded Up Robust Features is a robust local feature detector that can be used in object recognition or 3D reconstruction. It uses an integer approximation to the determinant of Hessian blob detector, which can be computed extremely quickly with an integral image (3 integer operations). For features, it uses the sum of the Haar wavelet response around the point of interest. Again, these can be computed with the aid of the integral image [8]. Except from SURF we can mention Open TLD algorithm that is based too, on the use of the integral image. These simple examples present the importance and the numerous facilities that the integral image offers to computer vision field.

1.4 Memory access and object detection algorithms

It is time to discuss the main problem that object detection suffers from and that was the motivation of the system we discuss. It is the delay of memory structures at the time of data fetching. Object detection algorithms treat an image in several ways to decide, at last, if it includes the desired object.

Filtering an image is a very common and yet characteristic example of how often a memory access is necessary at such tasks. The number of filters used, differs among algorithms but that is of minor importance as even an only filter demands thousand even million of memory accesses to be applied at the whole image. This is what we call “bottleneck”. Think of a 640x480 image, whose corresponding integral image is stored at memory, and an algorithm that convolves 5x5 filters with it ([figure 1.7](#)). A simple serial code will demand $(640-5+1)*(480-5+1)*25 = 636*476*25 = 7.568.400$ memory accesses. The completion time depends on the memory speed, the use of cache and others, but, even so the delay is enormous.

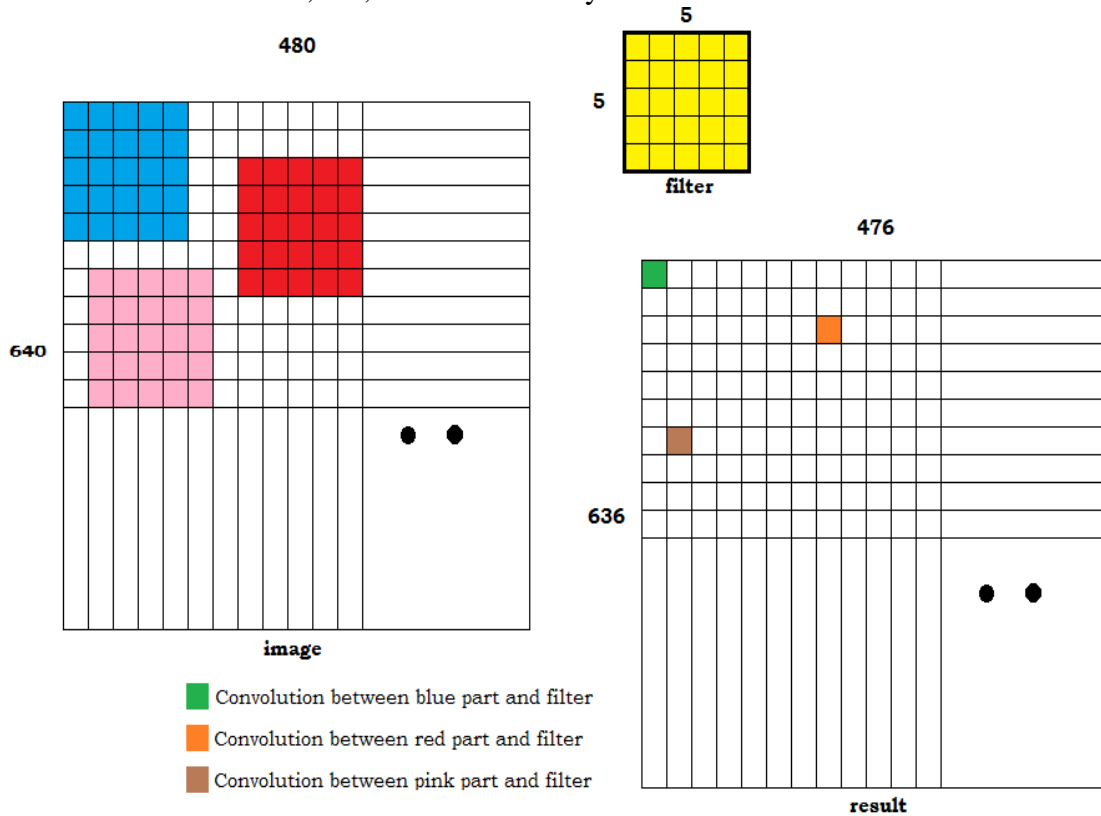


Figure 1.7 Image and filter convolution

The acceleration of the above accesses and the proper parallelism of memory is what our embedded system does.

1.5 Related work

In [15], the authors present a novel approach of the acceleration of the Haar-classifier based face detection algorithm with the use of a FPGA. They achieve real-time performance, very high detection rate and low false positives, using highly pipelined micro-architecture and utilizing abundant parallel arithmetic units in the FPGA. The design is also flexible toward the limited resources that a FPGA provides and gives a general view on implementing non-systolic based vision algorithm acceleration with FPGAs. The chip they use is a Xilinx XC5VLX110T FPGA, included in a HiTech Global PCIe card.

In [16] the authors are deal with the edge detection field which is used by numerous vision algorithms during their process. They use it in order to locate changes in luminosity or intensity that caused by big changes that a picture may appear. Furthermore, they proposed an FPGA implementation of their algorithm tailored to mobile robotic systems. More specifically, their hardware implementation uses the Altera Cyclone EP1C60240C8 and can perform the algorithm on a grey-scale image 360x280 in 2.5ms clocked at 27MHz.

In [17] the authors deal with a People – Detection System designed on an FPGA. They capture images from a network camera by using JPEG – compressed frames and they process the detection on a Virtex-II 2V1000. A MicroBlaze processor running at 75 MHz, controls the whole system behavior as well as the communication with dedicated hardware which is based on FSL links. The system detects people accurately at a rate of 2.5 frames per second.

In [18] the authors present an object classification that is able to classify objects in real-time using an FPGA implementation based on memory invariants and Kohonen neural networks. The two faces were implemented separately; the classification phase in hardware, while the Kohonen network in software. The hardware part implemented along with a set of sixteen parallel Kohonen neurons for the classification of an unknown object, demonstrating a possible real-time solution for object classification.

2 Chapter – The embedded Sub – System

The first part of this thesis is, as reported above, the optimization of a High Throughput Run Time Adaptable Memory Sub-system in order to be more efficient when faces algorithms of various species and various demands and, furthermore, to make it an autonomous unit which may easily be used in cooperation with other systems.

2.1 A general approach

The sub – system that will occupy us was, initially, designed by Nikitakis and Papaefstathiou, in order to solve some of the memory problems we described [1]. In general, it uses parallelism of memory and to serve, in a faster way, the accesses of memory that an object detection algorithm demands, during its implementation. Furthermore, it can handle situations, where parallelism could not be feasible, in the best possible way.

The first algorithm that was tested on the system was openTLD [2][19]. It accelerated the bottleneck of the algorithm and when implemented on a modern FPGA, the measurements showed that it is more than 23 times faster than a state-of-the art Intel CPU and even faster than a highly parallel Graphical Processing Unit (GPU). In addition it is at least 40x more energy efficient than both the Intel CPU and the GPU [2].

2.1.1 The four modules

Totally, it is composed by four different parts (figure 2.1), the loop decoding core, the collision detect core, the memories and the computation core which we analyze more below.

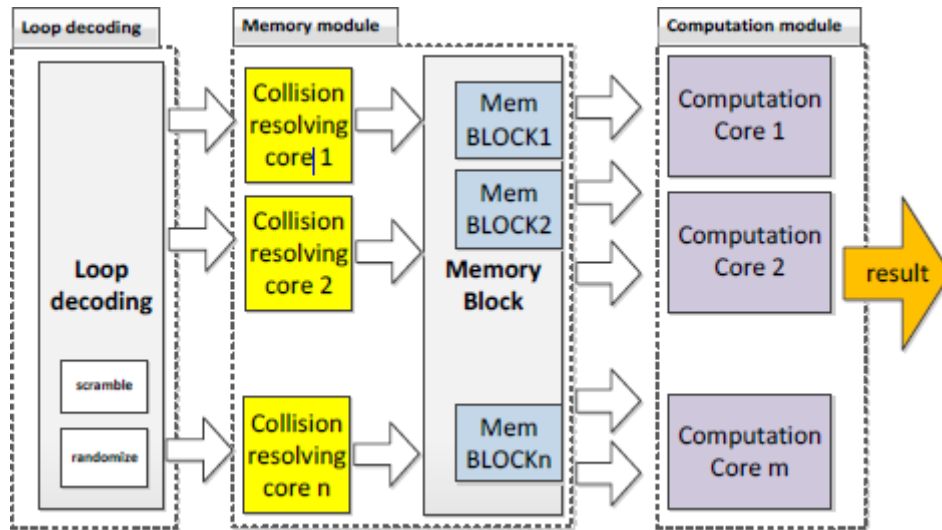


Figure 2.1 The four parts of the memory sub – system [2]

2.1.2 Computation core

Beginning upside down, for our better comprehension, we find the computation core. This is the part where, as its name reveal, our system do the math. The kind of the math may differs from algorithm to algorithm, as each of them processes images in its own way and, consequently, demands different treatment for its completion. The project in its “home” edition, as it is easy to deduce, uses the computation core to accomplish the $A - B - C + D$. These data come direct from the memory and once they are available the core calculate the sum. There is no sense to talk about this part more for now as we describe it in detail below and because it is a part that changes for the needs of the algorithm this thesis discuss.

2.1.3 The Memory module

Beside the computation core we find the memories that keep the data of the integral image and supply the core. Here is where the principle of parallelism is exploited. Imagine that we have a picture of dimensions 640x480. Each of these pixels needs to be stored in memory, so that they can be used by the algorithm whenever it is necessary. Having one big memory of 307200 (640x480) addresses, would gave us the data of only one pixel per clock cycle (10 ns). Instead, we divide our memory in 16 parts that contain equal number of pixels ($307200 / 16 = 19200$). As you can see the first block of memory contains the pixels 1 to 19200, the second one the pixels 19201 to 38400 and continuing in the same manner, the last one the pixels 288000 to 307200.

The ideal scenario, in that way, is to take 32 different data (we have 16 dual port memories) per clock cycle and this happens when each pixel corresponds to different block from the others. However, this is very difficult to happen due to spatial locality. If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future [\[9\]](#). And this is easy to understand if we think that we apply a filter at a part of the image. Even it is of dimensions 10x10, algorithm is going to demand a hundred pixels from memory that belongs to the same block! That means 100 clock cycles that is not very efficient. A way to solve this problem is to use a one-to-one scrambling function that will be implemented twice (see chapter [2.1.4](#)).

2.1.4 Collision Detect module

It is the part which distributes the block of addresses, born at the loop decode core, to the memories. Its task is very important as it is built to send each address to the correct block of memory and, furthermore, to serialize those addresses that correspond to the same block (that is what we call collision detect).

Each memory is connected with its own core, so if we adopt the 16 memories that we talked above, we will need 16 identical collision detect cores with different ID. Each core receives all the addresses and according to the value creates a signal that in the end is compared to the ID. If it matches, the core sends the address to the memory block. Thus, we can meet three different cases; no address matches the block, one address matches the block and several addresses match the block. The second one is the ideal one, as if it happens for all of the blocks, the sixteen addresses will be served in only one clock cycle. The worst scenario is when all addresses are driven to the same core – block (16 collisions), so they need to be serialized. The last one will pass through memory after 16 clock cycles, while other cores are idle.

With purpose to decrease the number of the collisions and those idle cores, we need to use a clever addressing in the memory and that's why we use the scrambling function we mentioned before. It effects to the addresses in such a way that the memory accesses are allocated to different sub-blocks. This function is implemented again to re-map the actual addresses to the scrambled ones, so to receive the correct data [\[1\]](#).

This part was suitable for optimization that would make the whole system more efficient and would give us useful conclusions on the question whether the extended use of hardware causes enough acceleration or not. In other words, there is a trade off. The more we increase spatial complexity, the more we decrease time complexity.

2.1.5 Loop Decoding module

We reached the level where the blocks of addresses are created. Its form depends on the kind of the algorithm but, in general, it follows the movement of the sliding window on the image (e.g. while a filter is convolved), the scale change and how the feature sampling is performed [\[1\]](#). Actually, it is designed to un-roll those loops in the algorithm that relate the above characteristics.

At home implementation the loop decode core creates 2 blocks of 16 addresses per loop iteration, in order to fill the sixteen dual-port memories but this is going to change in the final version of the sub-system.

2.2 Structure and functionality of each part

In this chapter, we are going to describe each of the parts we mentioned above, with more details. We grapple with the inputs of each unit, in what ways they are exactly processed (FSMs, control signals etc) and finally, what signals they lead to the rest system (outputs).

2.2.1 Loop Decode Module

In the beginning, we describe the way in which the addresses are born. In general, as we mentioned before it is built to unroll the loops of the target algorithm that causes big delay in memory.

In [2] the authors describe exactly its functionality. We are informed that after the decoding of the 5 inner loop of the algorithm the core produces 16 memory accesses per cycle and this is achieved with the use of an 8-state Finite State Machine combined with 4 lookup tables (which are connected with 4 fixed-point multipliers. The core produces, then, sixteen outputs and a bit that validates them.

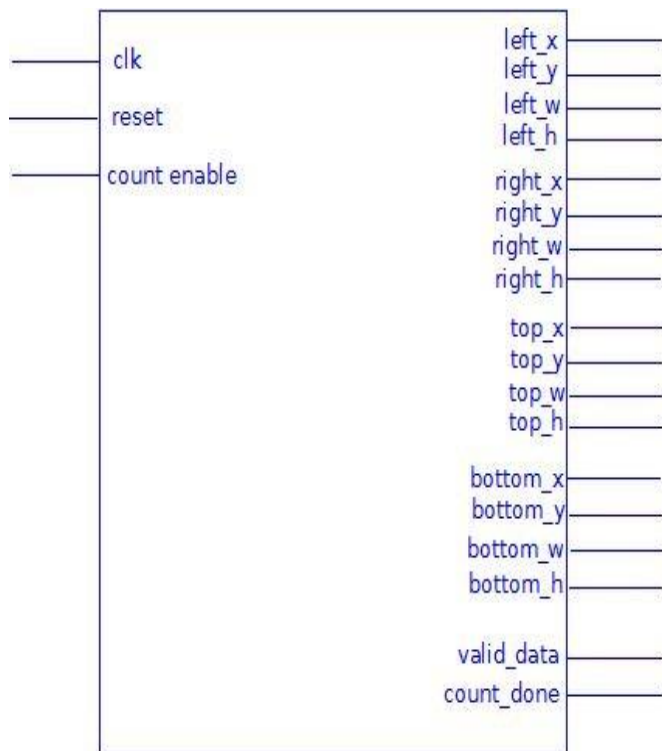


Figure 2.2 Loop Decoding Block Diagram

These mentioned just for our better comprehension. We are not going to get deeper here as this part is going to be totally different in the final version and there was no reason to change it.

2.2.2 Collision Detect Core

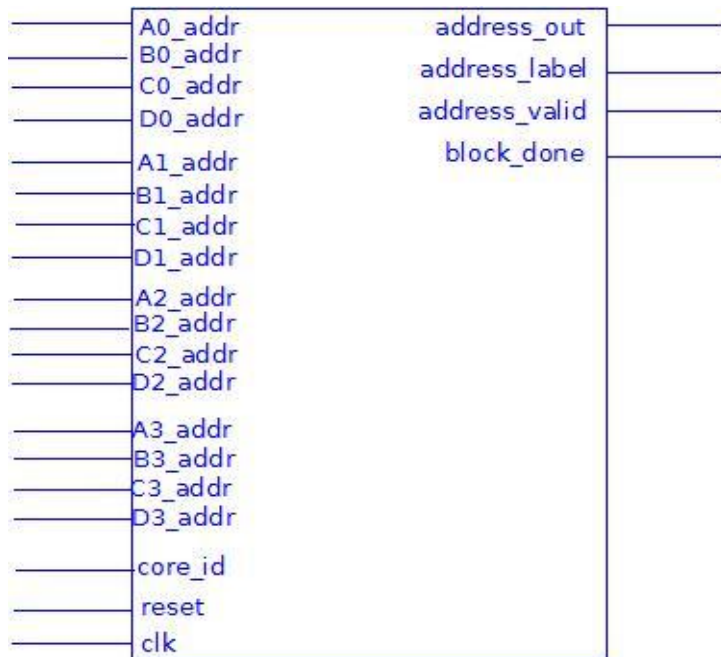


Figure 2.2 Collision Detect Block Diagram

Here is the most important part of the sub – system. That is the core where the sixteen memory accesses, that the previous core produced, are driven. So we have sixteen inputs that refers to those addresses and one input for the core id that is unique for each copy of the core that is used in the top level and four outputs. These inputs are named from A to D and from 0 to 3 (A0, A1, ..A3, ..B2, ..C4...D3).

As we have divided the image into sixteen different parts, it is time to see where each input belongs. Thus, we create sixteen signals similar to core id declaring in which block each address should be sent. For example, if A0 address is 11000 it belongs to the first memory block, as it is between 0 and 19200, and the A0_dest_sig takes the value “0000001”. It is easy to understand that the possible values for these signals are from “0000001” to “0010000”.

After finishing the definition of these signals we have to specify which of the addresses are forced to get through the specific core. We need to drive out all the addresses that their signal is equal to the id (serialization). For this reason, we use two FSMs of 8 states each. We split it in two parts, in order to avoid one huge FSM of 16 states. In here, we compare each signal with the core id.

At the first state S1 of first FSM we compare the A0_dest_sig. If it matches we drive out the 15 Least Significant Bits of the A0_address (see [2.2.3](#)), its label and the valid bit. These are some of the outputs of the core. The label describes each address (A0-D3) in a unique way and the valid bit declares if the exported data are correct or wrong. Afterwards, we check the next one (B0) and if it matches too, we transfer to case S2 where we follow the same procedure (put signals to output and check for the rest signals). If not we check C0 and do the same. If in S1, A0 does not match we check B0 and we act as before. Have in mind that each state corresponds to

the service of different address. When we visit a case X, means that its corresponding address Y is going to get to the output.

As far as, FSM 1 is done we get to state S0 that activate the second one, which treats the signals from A2-D3 in the same way, by disabling the reset. This act transfers us in the first state of the second FSM and the procedure continues until all signals are checked. If so we enable the block done signal, which is the last output of the core. This alert the sub – system that the core has finished with the process of the present block and that is ready for the next one. Have in mind that, the existence of two FSMs forces us to use different copies of the same signals. For example, address valid signal exists as “address_valid_a” in the first FSM and as “address_valid_b” in the second one. We choose the appropriate signal according to which FSM is enabled.

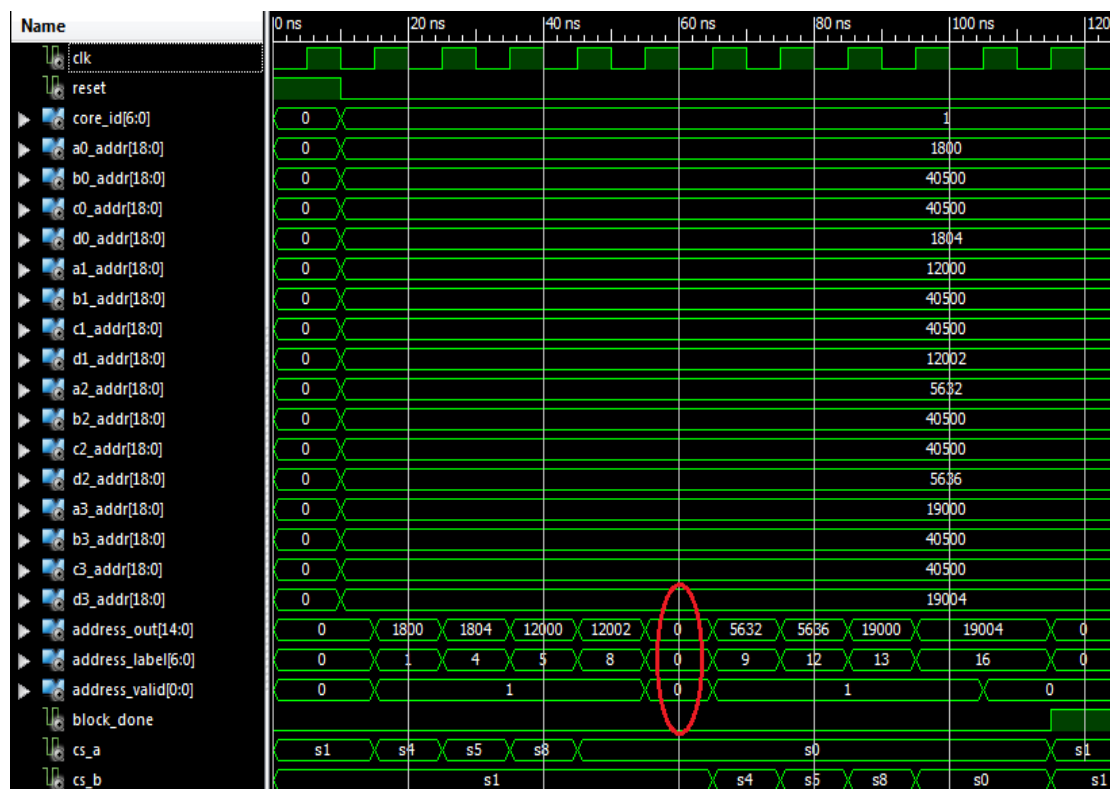


Figure 2.2 Collision Detect Core Simulation - the lost clock cycle

The way that FSMs are constructed satisfy all the possible scenarios and that make us sure that all the addresses will follow the right path to the memory blocks. In [figure 2.4](#) appears a characteristic simulation of the module. Having 1 as core id, we expect to the output only those addresses between 0 and 19200. Indeed, we see that these addresses are serialized and we take one result per clock cycle. The red cycle show the point where we move from one FSM to another and. It is a lost cycle for which we are going to discuss in [3.4](#).

2.2.3 Memory

This is the place that are stored the data of the Integral image. We use a dual port RAM of depth 19200 (number of pixels in each block) and of width 26 (bits needed to describe a pixel). In order to express those 19200 addresses we need only 15 bits from the 19-bit addresses that handles the collision detect core, so we use the LSBs. It is important to mention here why there is no loss after the removal of the 4 MSB and even the 15 bits are enough for our purpose.

Imagine that we separate the address into these two parts. With 4 bits we can refer to 16 different items while with 15 bits we may refer to 19200 items and to exact 32767 of them. Having this in mind, we can deduce that the 4 MSBs show us the core, while the 15 LSBs show us the position in memory.

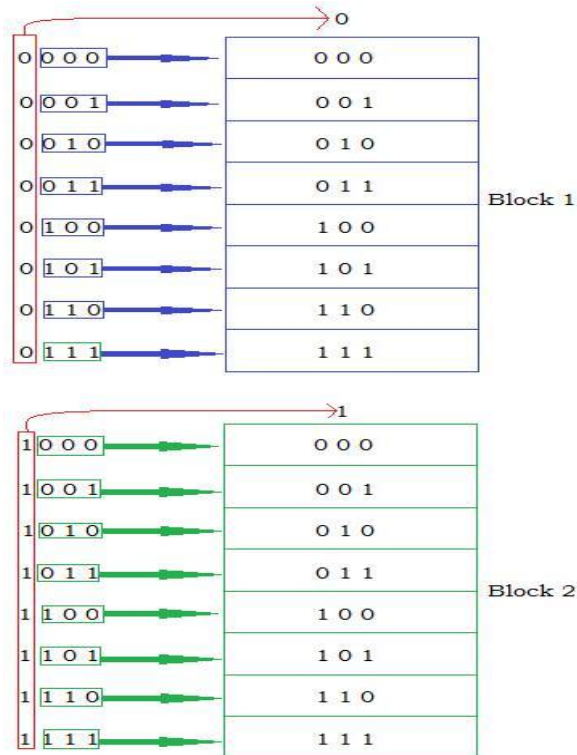


Figure 2.5 Memory addressing

Think of this; Similar LSBs, but different MSBs show the same address of memory but in different blocks, while different LSBs, but similar MSBs show the block but different addresses. By the time we reach to memory, we have already decided the correct block and so, the LSBs are what exactly we need to get the correct data. The [figure 2.5](#) shows what we just described on a smaller scale, for our better understanding. It has sixteen addresses distributed in two blocks. In this example, Collision detect core that exists beside block 1, handling the address 0101, would sent only the 101 to the block 1 exporting the right data. On the other side, Collision detect core that exists beside block 2, would recognize that 0101 does not belong to block 2 and sends nothing.

2.2.4 Computation Core

All the signals that are until now “open” in our description are plugged in this part of our sub – system. They are none other than the data came out from the memory and the label and valid that are accompanying them and came out from the computation core. Therefore, we have $3 \times 16 = 48$ inputs and we use them to calculate 4 sums; $A0-B0-C0+D0$, $A1-B1-C1+D1$, $A2-B2-C2+D2$, $A3-B3-C3+D3$. When all of them are complete the core enables the block done signal to inform the whole system.



Figure 2.2 Computation Core Block Diagram

In order to begin its process, the core needs to store the elements that come from the memory to the correct signals. Thus, it is checking the label and the valid signal that come from the computation core. If valid of some block is equal to ‘1’ it puts the data from the corresponding memory to the signal that the label indicates, else it puts ones. Twenty seven bits of ‘1’ is very convenient, as a number that is not being used so can show us that we have no data.

The main processing takes place in an FSM of three active states that handles the signals where the inputs will be stored and those control signals which declare if they are stored or not. In the reset state all these signals are initialized to zero. Next, we get to state 1, where we check those signals that have no value yet. We stay here until all values are caught. In parallel, we have 4 signals that hold the sums by adding those exact signals that the FSM processes. In the case where all signals are caught in the FSM, it is sent to state S3 where the block done signal is enabled.

In the simulation that is appeared in [figure 2.7](#) we try to show a representative example. For our easy understanding, we arrange all sums to be the same (8) and all signals to be caught in on the first cycle except two. The data of B0 while be caught on second clock cycle while data of A3 on third cycle. Thus, we stay in case s1 until all of them are caught and the computation of the corresponding sums is delayed. The red arrows show that when the last remaining signal is brought, the sum is calculated on the next cycle.

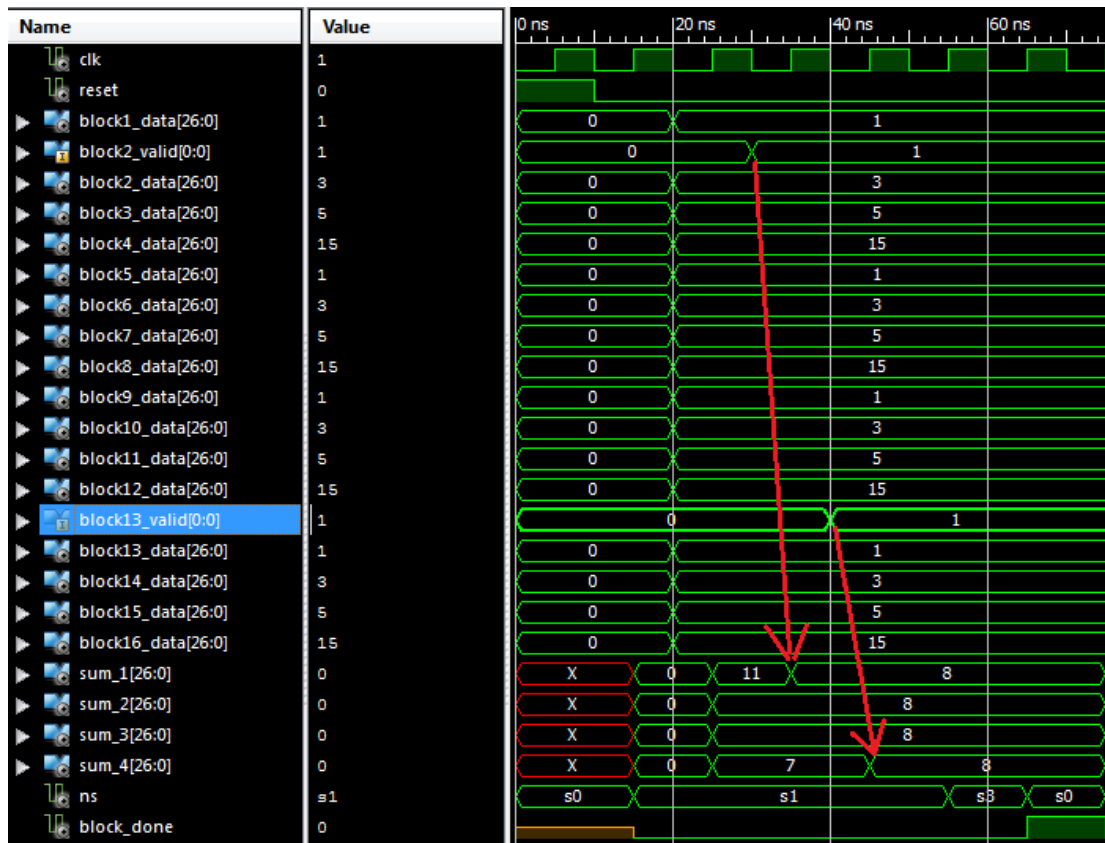


Figure 2.2 Computation Core Simulation

2.2.5 Top Level

We reached, so far, the level that includes and controls all the above units. A universal image is what we need to complete the puzzle.

In the beginning we use one Loop Decoding Core. The sixteen addresses it products are driven to, also, sixteen Collision Detect Cores, each of which has its own id. Mention that, as you may have already understand, there is no relation between the two “sixteen”. We could have 32 addresses processed by 16 cores or something like that. After each Collision Detect Core we have a memory block and at the end two Computation cores. The first one is going to receive the signals by the A ports of the memories, while the second one those from ports B. Although, as we send the same addresses to both A and B (home edition), the two cores calculate the same result.

Of course, we need to control all these units in order to work synchronized and we use a FSM for that reason. Having received valid addresses from the first Core, the FSM disables the reset from the Collision Detect Core and “freezes” the address production. Then we get to state S2 where we stay until the Collision Detect process is done. If so, we are ready to receive the next block of addresses, so we activate the Loop Decode again.

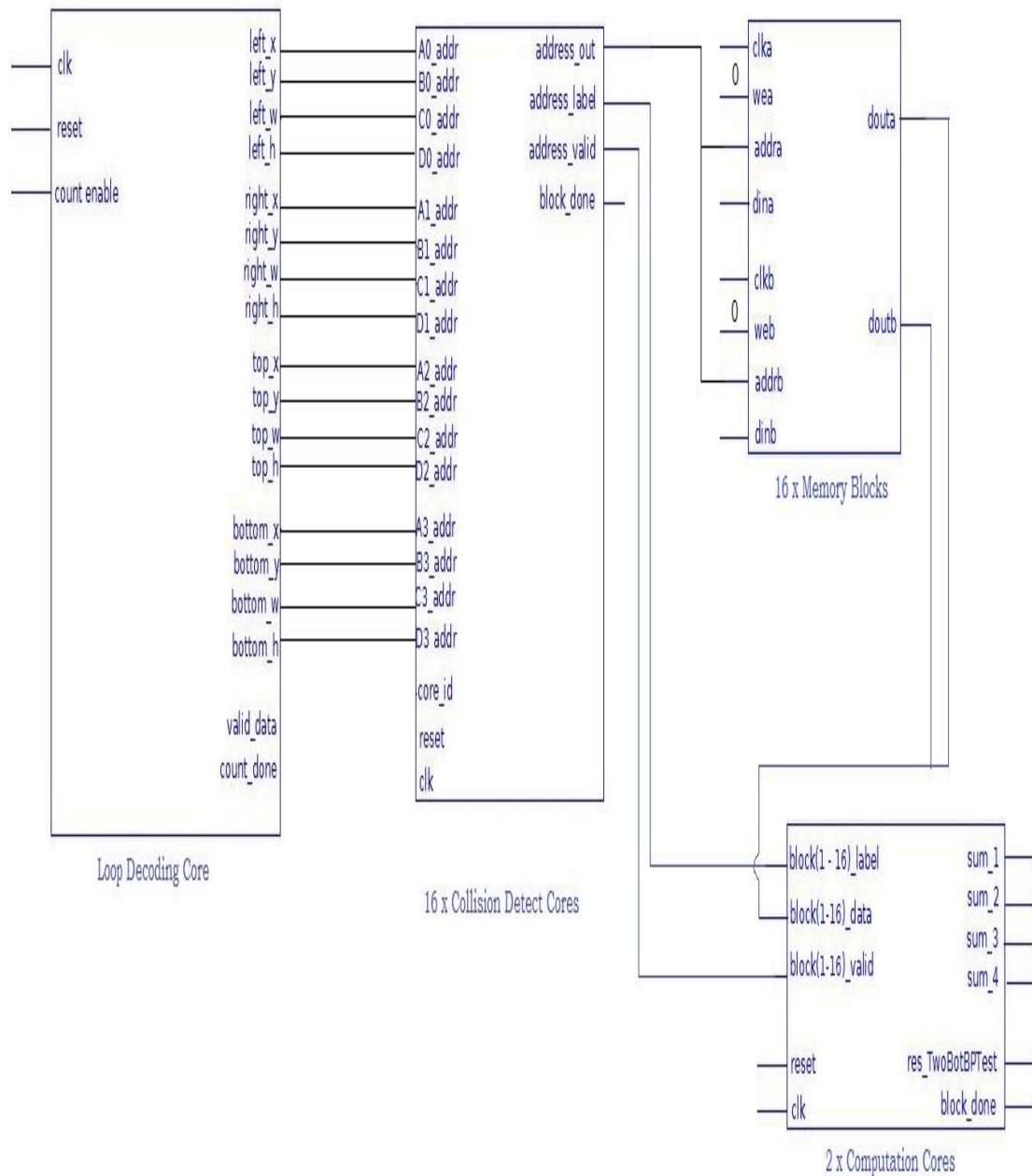


Figure 2.2 Top Level Block Diagram

2.3 Conclusions

The sub – system we just described was the base on which this thesis was developed. The authors in [2] had spotted some things that was need to be changed and that there were a lot of optimizations to be done within the border of our research of how much faster this sub – system could become and how much would expand the range of the concerned algorithms. This is where our job begins and the theme of the next chapter.

3 Chapter – Optimizations on the complete system

A deep study on the code and its description on the previous part of this report would bring an experienced scientist in front of critical problems that need to be fixed and various transforms that could be applied on our embedded sub – system. In this chapter we are going to describe all the optimizations we applied on it. At the end of this part, our system will be able to process 32 addresses each time, with the use of 32 Collision Detect Cores and 32 memory blocks, while duplicating the team of those cores will be feasible the process of 64 addresses using both the ports of memories. Furthermore, especially the Collision Detect Core, will be able to use either the previous optimized FSM or a new FSM with 32 states. Finally, the generation of the address block will be continuous by a new module that produces random addresses and the use of FIFOs will make them available to the system at any time needed. Of course, it is necessary the creation of a new FSM that controls the whole system.

3.1 Use of 32 Collision Detect Cores

The first change we attempted was the increase of the number of Collision Detect Core and inevitably the number of memory blocks from 16 to 32. Its purpose was a further decrease of the collisions by a percentage. The size of memories halved, as we have to distribute the 307200 pixels of the image into double blocks ($307200 / 32 = 9600$). Thus, for the data mining we need 14 bits at the input of the memories. As concerns the Collision Detect Core, we changed the range whereby we were initializing the destination signals. As they represent the target block of each address, they should follow the halving done to the memory. At the computation core we add the corresponding inputs that come from the extra memories and cores, as they can, obviously, be data providers and have to include them at the computation of signals. At top level, we just increased the number of port maps for both components from 16 to 32, adding the proper signals too.

In this way, we managed to eliminate the collision of addresses that were belonging to the same block before. For example we shall refer to addresses 8000 and 16000. In home edition, they should both be driven to block 1 as they are between 0 and 19200. Their process would finish in two clock cycles by the same block. Now that the separate line is 9600, the first one will be served by block one, while the second one by block two and all these in only one clock cycle ([figure 3.1](#)).

We can easily reach to the conclusion that this optimization can accelerate our system by 50% at best case. The worst one is, if each address belongs already to different block so we do not have any acceleration. Of course there are intermediate cases too, such as the serving of ten addresses by one block that after the change will separate into 3 and 7. Reducing the clock cycles from ten to seven we have 30% acceleration.

Having to handle million of addresses during the execution of the algorithm, where time is critical, this acceleration may save as thousand clock cycles. A further increase of the blocks would achieve a higher acceleration but also requires more

hardware. The question is where is the best balance, which gives us higher performance in low cost.

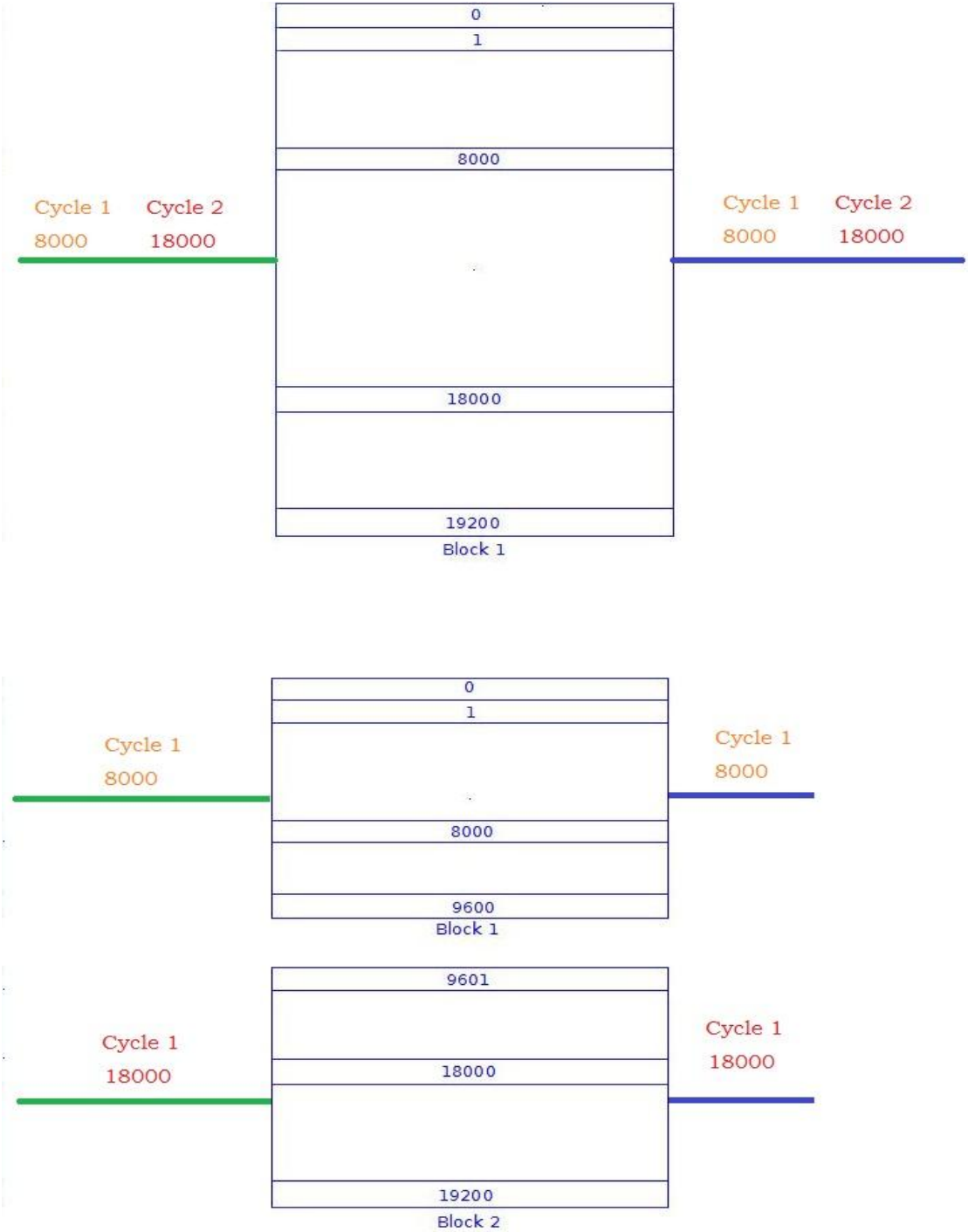


Figure 3.1 Advantage of using more Collision Detect Cores

3.2 Handling sets of 32 addresses

The sixteen addresses, as said before, are used for the computation of four values via the operation $A - B - C + D$. We could easily prepare the system in order to process double number of them, again for speed reasons. The way we produce them will be analyzed below, in chapter 3.

Until now, we used variables A0 – D3, but it is time to add those from A4 – D7 that will be the new inputs of Collision Detect Core. In here, we treat them in the same way as the old ones as concerns the creation of the destination signals. Although, it is necessary the creation of two more FSMs, one per eight signals, that will be, obviously, a copy of the previous ones. The difference lies in the block done signal which is enabled by the end of the fourth FSM's process. We can tell that the present fourth FSM is similar to the previous second one. Despite this, we move among the FSMs in the same way as before.

At the computation core, we add the appropriate signals with purpose to catch the values of the new variables and we make a copy of the existing FSM which handles them. The block done signal is activated as soon as both of the FSMs have finished and, then, the core will supply the system with eight sums.

The adding of these addresses allow us to compute four more rectangles of the Integral image in only one “repetition” of the sub – system. The worst scenario is the 32-cycles delay in the Collision Detect Core, but it is less possible with the optimization in [3.1](#). Again, we face the question of what is the ideal number of addresses that should be provided to the system.

3.3 The duplication of 32 cores and the ability of 64-address-queries

By now, we have not take advantage of the second port of the memories that gives us the opportunity of exporting the data of two variables in only one clock cycle and that because we were sending the same address at both inputs. Therefore, as said in [2.2.5](#), the two units of the computation core calculate the same eight rectangles (after [3.2](#)). It is time to change this situation by duplicate the team of the existing 32 cores. In that way we create twin cores that differs only in the name (Collision_detect_core_1 - Collision_detect_core_1B). The twins have the same id as they refer to the same block of memory, but their outputs are driven to the two different ports.

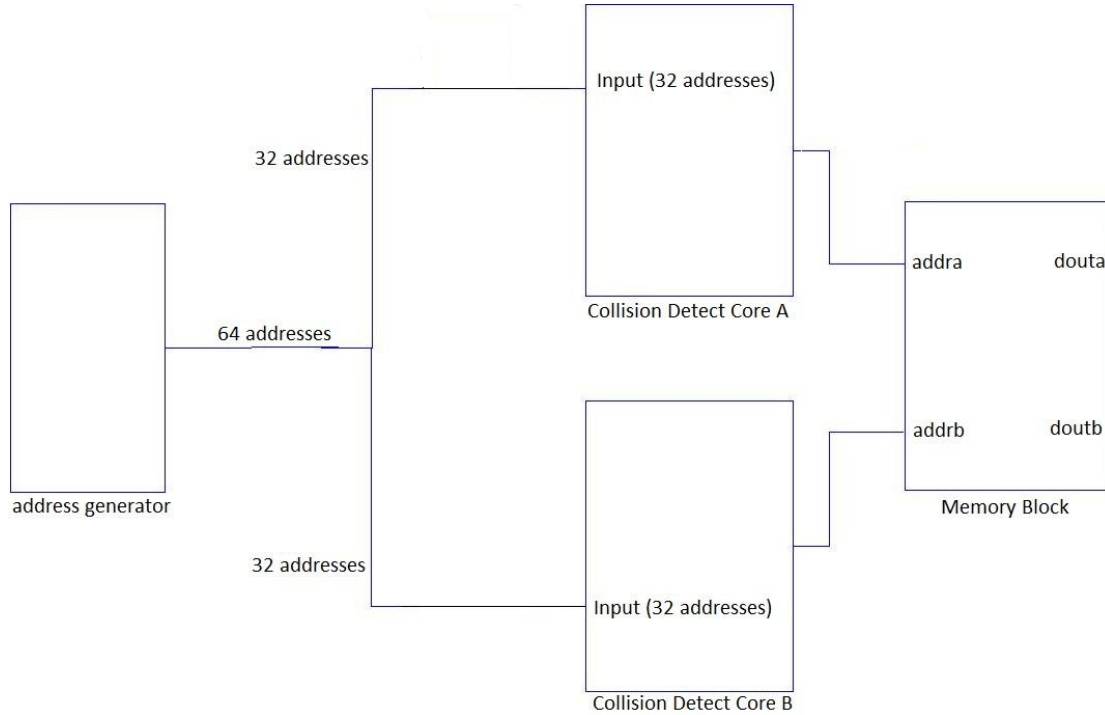


Figure 3.2 Supply of 64 addresses - Use of the memory's second port

We are allowed, now, to supply the system with 64 different queries that will be created in the loop decoding core. The first team of 32 is sent to A cores, while the second team to B cores ([figure 3.2](#)). At the end, we benefit of sixteen different sums, which means sixteen different rectangular areas that were asked by the algorithm.

3.4 Optimization of Collision Detect Core's FSM – The two configurations

In [2.2.2](#) we identified, with the help of [figure 2.4](#), that the functionality of the two connected FSMs, is characterized by a weak point. If the requested addresses need to use both of them, an empty clock cycle is appeared which consists a useless delay. Now, we need to understand better the transitions made during their process.

When the external reset is enabled we are in reset case of the first FSM. In there we activate the reset of the second FSM and we hold it to '1' while we handle addresses from A0 to D1. When we are done, we move to state S0 that is a pause state and no data are leaving. We check, also, if we have already gone to second FSM and if not, we activate it immediately. Thus, we are in case S1 and continue with the process of signals A2 to D3. The lost clock cycle is, obviously the pause state and there is need to be eliminated, as it could be lead to a huge universal delay. There are proposed two solutions with advantages and disadvantages.

The first one is the modification of the present FSMs in order to erase this idle state. So, when get there, we just activate the second FSM by setting the reset b to zero without touch the other relative signals. Having reset at the sensitivity list, the FSM will begin its process but with value reset = 1 as it is going to change to 0 on the

next clock cycle. We exploit exactly this specificity to start checking signals at this very clock cycle. And as you can see in [figure 3.3](#) it works fine!

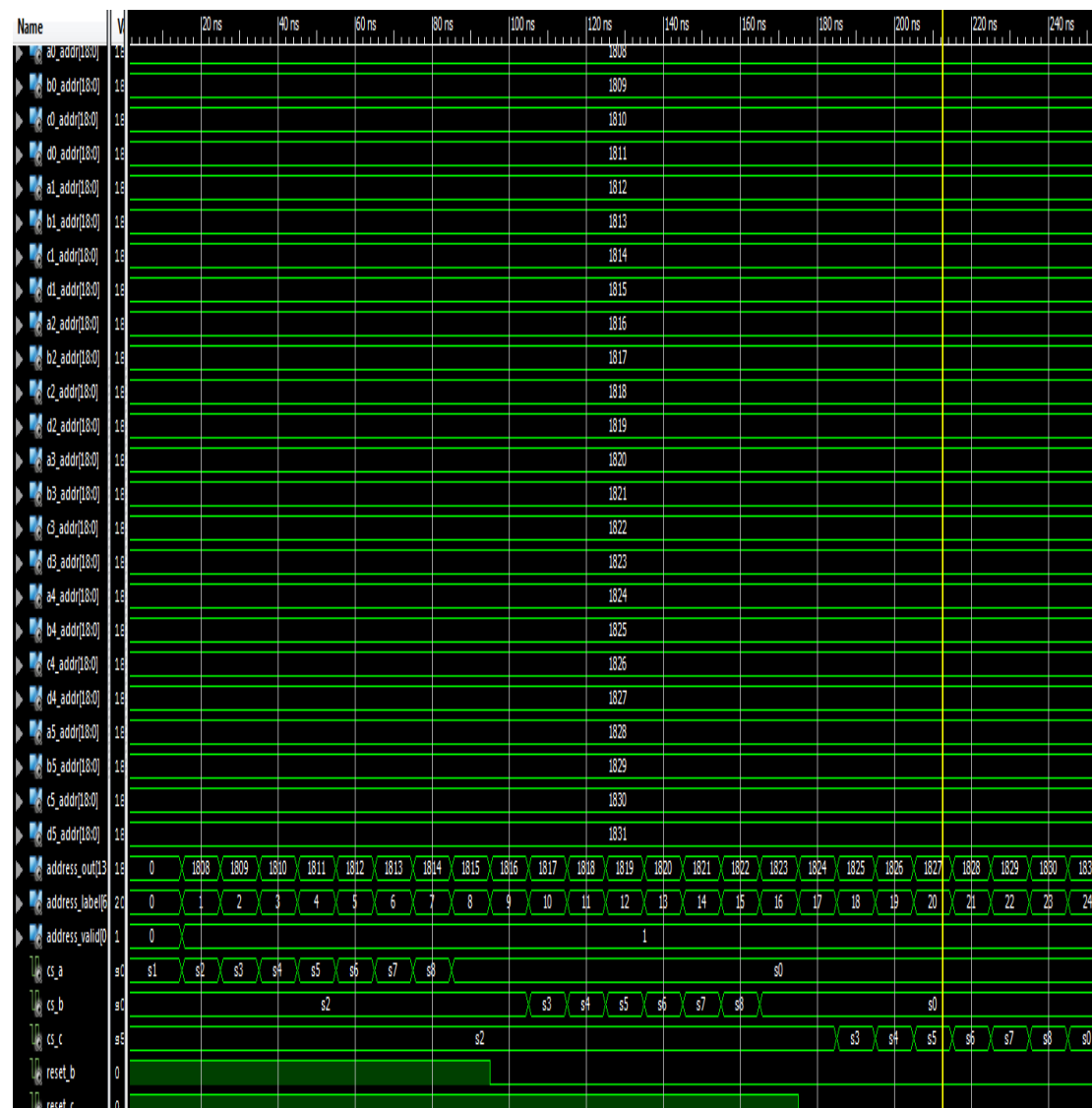


Figure 3.3 Collision Detect Core Simulation - Corrected four-part FSM

In this figure we present only 24 signals and the functionality of 3 FSMs for space reasons. As you can see when the first FSM finishes, it closes the Reset b and at on the next clock cycle the next one gives to the output the value of the first address it process (1816). Afterwards, it checks for the others and, as soon as, it is done it activates FSM number 3 in the same way.

The disadvantage of this implementation appears at the version of four linked FSMs ([3.2](#)) and when there is no use of the intermediate ones. There we need at least one clock cycle per FSM and if no address matches this cycle is lost. You can see an example in [figure 3.4](#). We are in core 1 se we handle addresses form 0 to 9600. The second FSM is not used while the addresses it processes do not belong to it. Thus, it is stable to state s0 and then gives command to third FSM to begin.

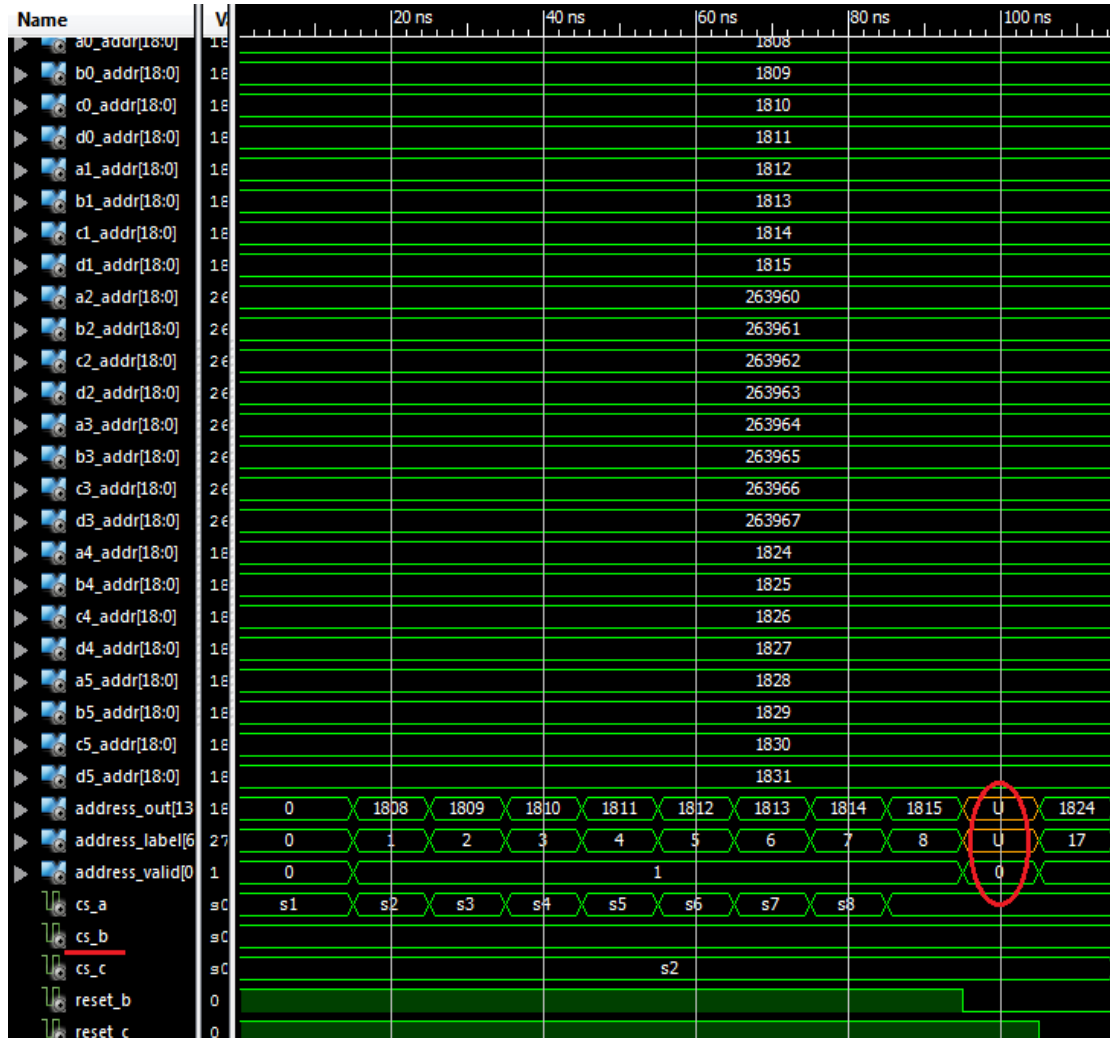


Figure 3.4 Collision Detect Core Simulation - Idle intermediate FSMs

The second solution is the construction of one huge FSM of 16 or 32 states, depending on the version we use. While we get rid of the signals that were connecting the previous FSMs, each new state is composed by much more controls in order to cover all the inputs and furthermore no state is similar to one another as before. The advantage of this implementation is that there is no loss of clock cycle at any case and that any team of signals is going to use it as long as it is necessary. Figures [3.5](#), [3.6](#) and [3.7](#) show three different possibilities that have interest to study.

The [first one](#) shows the serialization of all the signals as all of them belong to the certain core. Once again we present only 24 inputs as there was not enough space for the other 8. Although, it is clear enough the way in which the FSM works as signals like “valid” and “CS_a” that declares the movement, are visible.

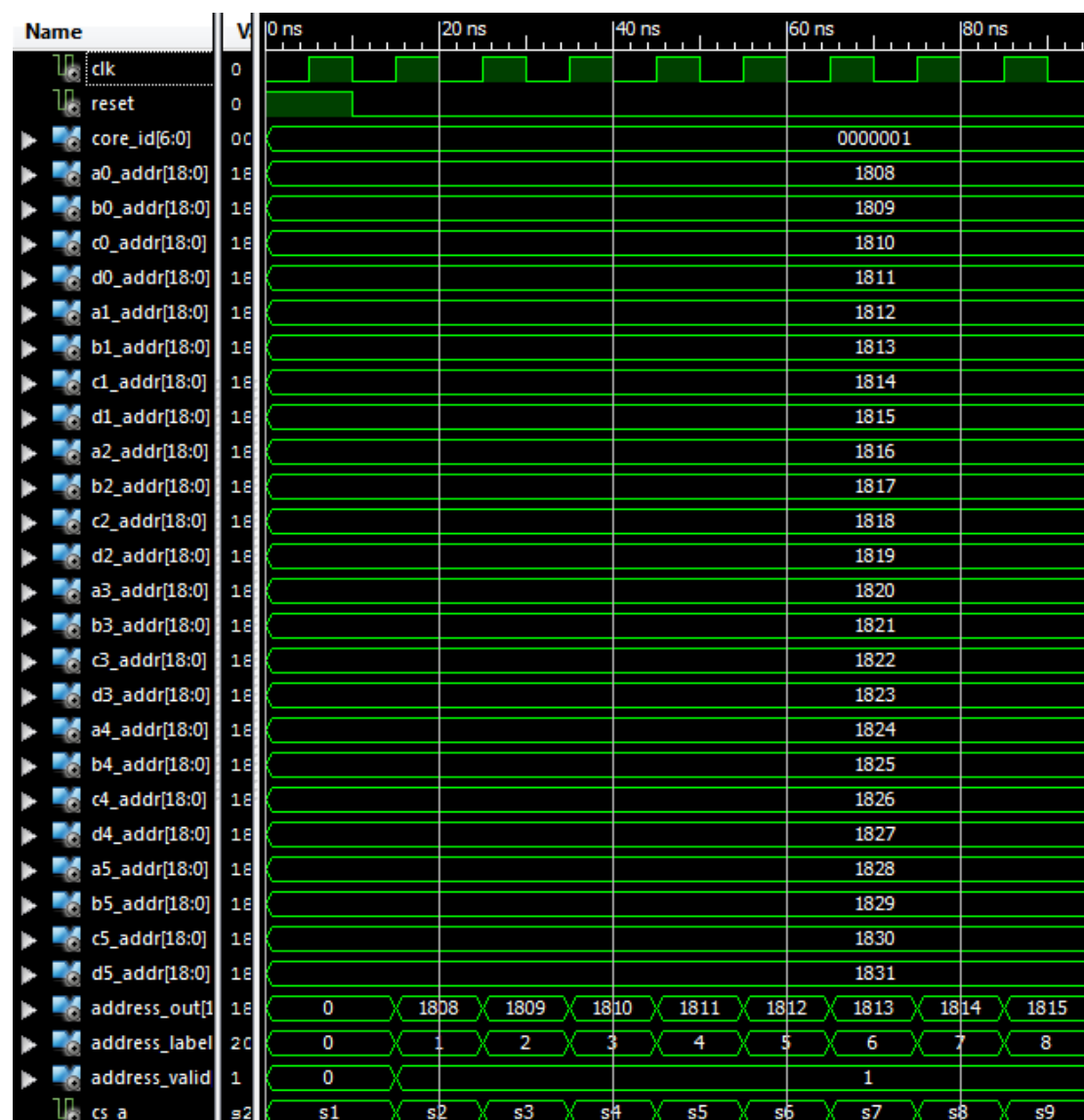


Figure 3.5 Collision Detect Core - 1 Huge FSM Simulation - Case 1

The second one is the presentation of the case we lose with the previous implementation. In [figure 3.6](#) we choose to pass through the core, two signals; one from FSM 1 and one from FSM 4. As predicted, there is no loss of any clock cycle as there is no such reason. We display only few non – taken addresses with the taken ones.

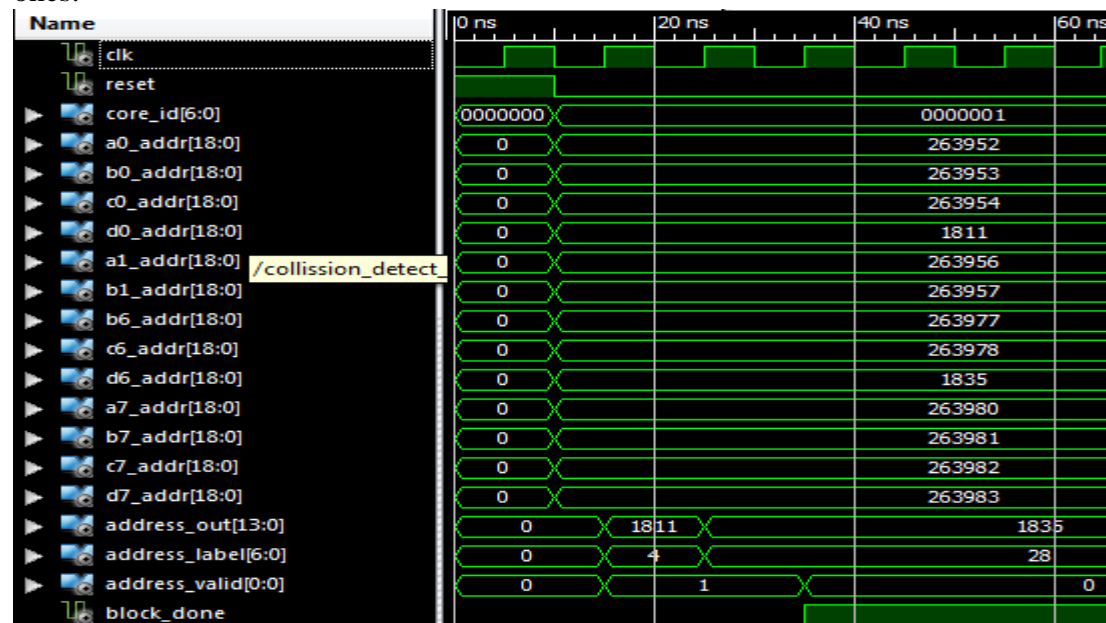


Figure 3.6 Collision Detect Core - 1 Huge FSM Simulation - Case 2

In [third case](#), we simulate the behavior of the FSM when no address belongs to the certain block. We need only one clock cycle to find out that no address is going to get to the output and enable the block done signal.

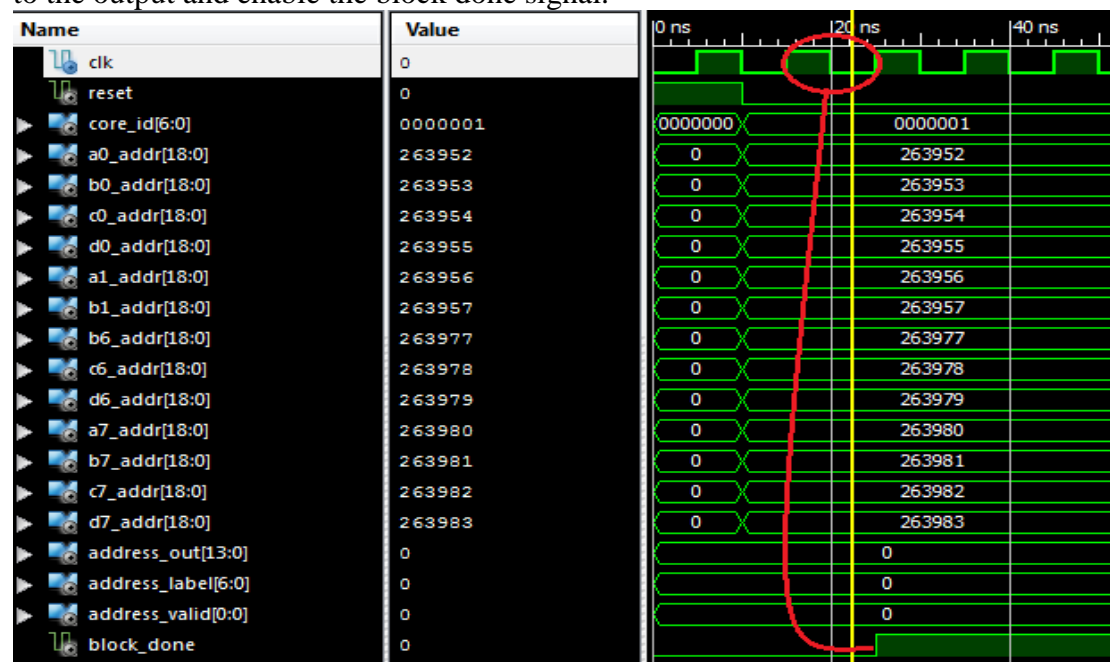


Figure 3.7 Collision Detect Core - 1 Huge FSM Simulation - Case 3

3.5 Use of FIFOs – A permanent supply of addresses

The control we described at [2.2.5](#) was activating the production of an address block as soon as the Collision Detect Core had finished its job. That would start and stop the Loop Decoding Core too many times during the execution of the algorithm and considering that a clock cycle is needed per activation that means too many lost clock cycles.

The best possible way to get rid of those cycles is to use FIFOs. FIFO (First In First Out) is a method for organizing and manipulating a data buffer, where the oldest (first) entry, or 'head' of the queue, is processed first [\[10\]](#). We use, then, as much FIFOs as the number of the addresses in a produced set. Each of them has space for few addresses and all of them are synchronized by the same signals.

Thus, in general, we store all the produced sets of addresses into FIFOs and when the Collision Detect Core demands a new one it is ready for use. The first unit is activated to produce new sets when there is free space in FIFOs but this will not cost a clock cycle anymore as it is taking place while Collision Detect Core is busy. In [figure 3.8](#) you can see a typical simulation on a FIFO.

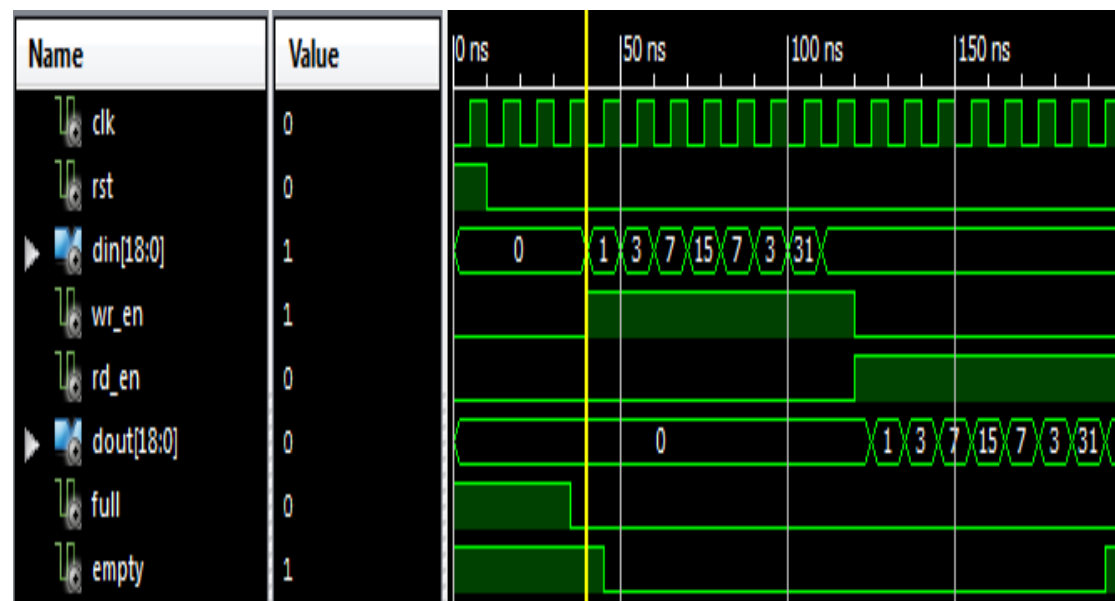


Figure 3.8 FIFO Simulation

We put seven values to the input, while we enable them to be written, as soon as we are informed that the FIFO is empty. Then we start read these values and when we have done the “empty” signal is enabled again.

3.6 Generating pseudo-random addresses

The insertion of these units has made our system more autonomous and its units more “discrete”, as we separated the unit of production with the unit of consumption. It is now easier to change the first one, depending on which algorithm we implement. For now, we are going to use a unit that produces 32 addresses that have no relation with the reality. It was created just for reasons of measuring and recording the behavior of the complete sub – system. As it has not some special functionality, it could be totally omitted but for the report reasons here it is.

When the unit is reset we create 32 addresses of some value that have one bit less than expected. When the unit is enabled we add 32 in each address at every cycle and put the result to the output including the missing bit (MSB). The reason of this bizarreness is to hold the values of the addresses into the limits (307200). For typical reasons we simulate the functionality of this unit in [figure 3.9](#).

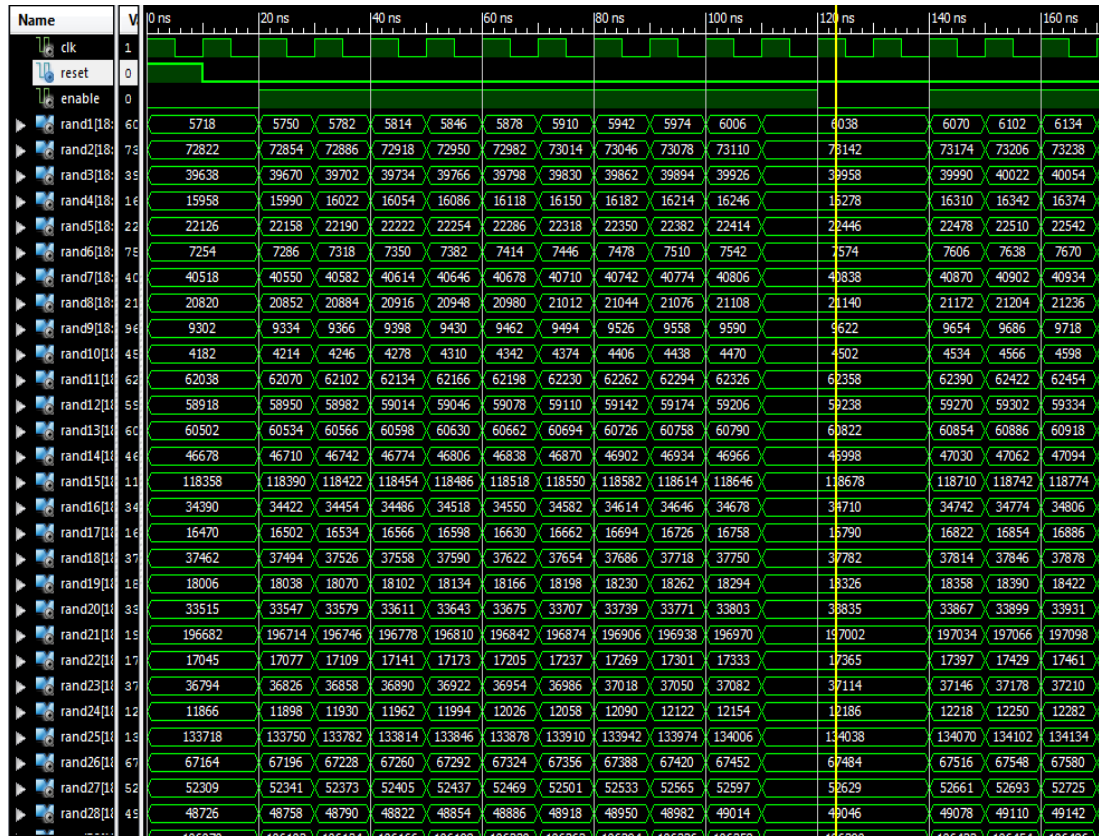


Figure 3.9 Random address generator Simulation

When enabled the system starts the production of the addresses in every clock cycle. Then we disable it, and see that it stays stable. Afterwards, it resumes from the appropriate address. This is very important because in future Loop Decoding Cores it is necessary to get all the addresses in the correct order.

3.7 Use of delay registers

The three signals that the Computation Core receives for each address was, until now, not synchronized. While the “address label” and “address valid” signals were driven direct to it, the “address out” is sent to the memory which has one clock cycle delay. For this reason we use a register to delay the other two in order to get to the unit together ([figure 3.10](#)).

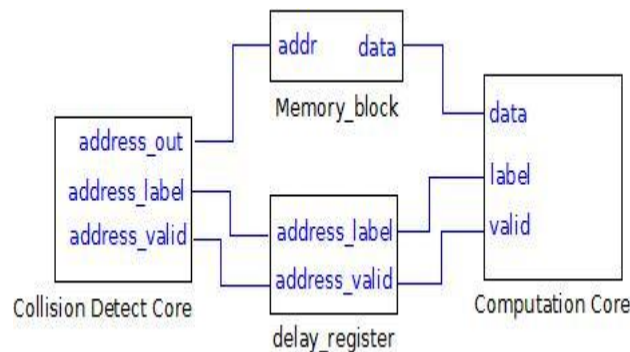


Figure 3.10 The use of delay register

3.8 Total system control

This renewed sub – system needs a different control system from the previous to work correct and have all its units synchronized. In this paragraph we are going to describe how we handle the “vital” signals of each unit using a control FSM in the Top Level and not only. You can see the below description represented in [figure 3.11](#).

In the beginning we put the system’s reset to the address generator and FIFOs. The inverted “full” signal from FIFOs is connected to address generator’s “enable” signal and to FIFOs’ “write enable”. In this way, we ensure that as soon as the FIFOs have free space the address generator will supply them with sets of addresses. These signals have no need of the FSM but it is not the same for the next ones.

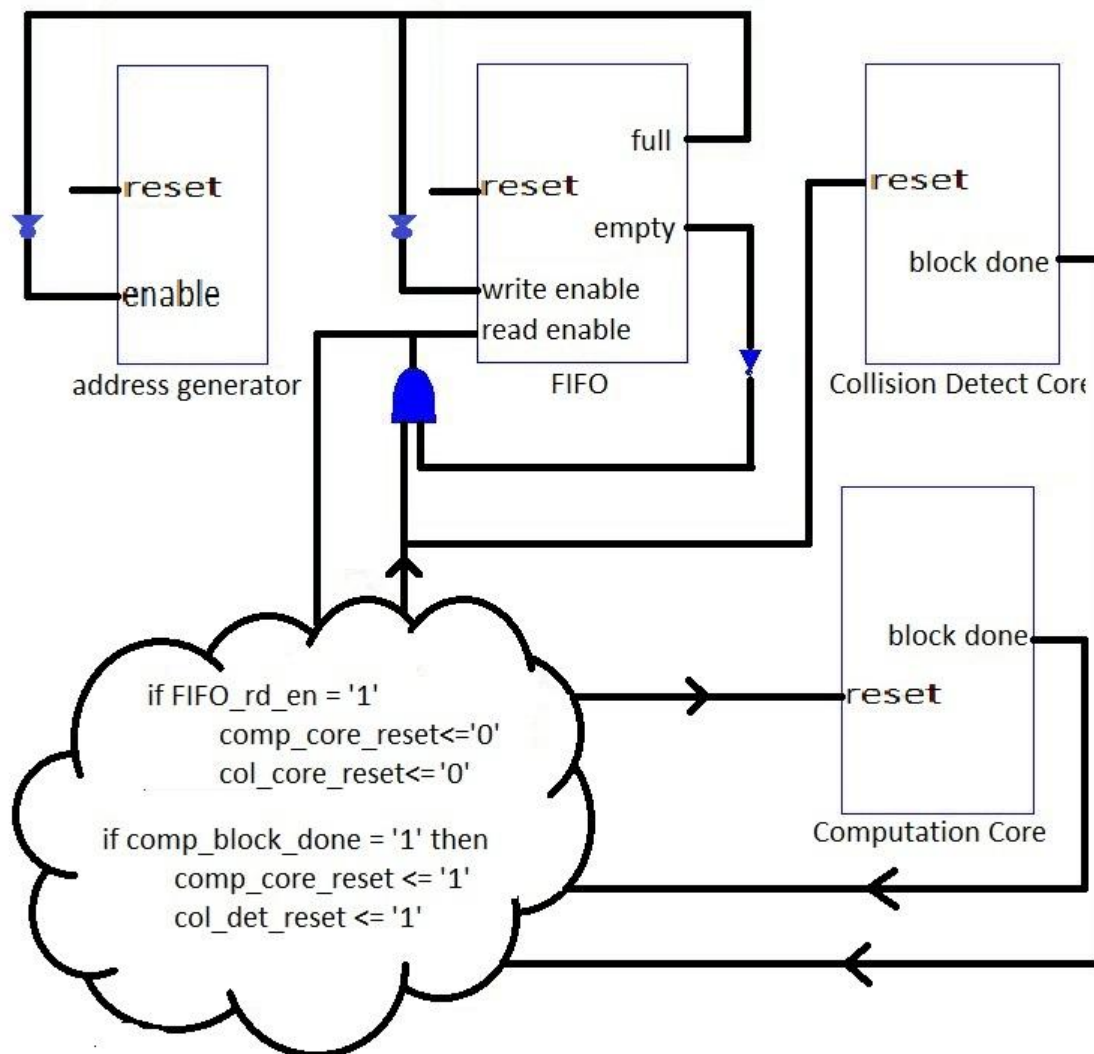


Figure 3.11 Top Level Block Control

In order to read data from FIFOs we need two things. The first is that the FIFOs are not empty, so we use the inverted “empty” output. The second is the Collision Detect Core reset that is handled by the FSM, like the Computation Core Reset. When the System’s reset is enabled we activate them all and we are ready to transfer at state S0. We stay here until the FIFO “read enable” signal is ‘1’ and, having reset already ready, this is going to happen when the FIFOs gets data. At this very moment we put the two main units to work and we get to state S1. We check, now, if the Computation Cores have finished their job. If so, we activate again the reset signals and we are ready to receive the next set of addresses from FIFO while moving to state S0. Otherwise, we stay here until this happens.

Being sure of the right functionality of the control unit and the whole sub system, we can use it to face anyone algorithm belongs to Viola – Jones framework and we are ready to test on it different kind of algorithms and see the results.

3.9 Performance evaluation

Using an FPGA of Zynq family, the XC7Z045, we synthesize and implement our design in order to see its requirements and if we can download it correctly. By using the version with the four linked FSMs we achieve a minimum period of 4.932 ns and a maximum frequency of 202.751 MHz, while using the version with the one huge FSM we have 4.947 ns as minimum period and a maximum frequency of 202.136 MHz.

It is encouraging that we achieved and overcame the threshold of 200 MHz in which the initial sub – system was working. That means that, in general, our optimizations did not affect the clock period and thus can be characterized as successful.

4 Chapter – Crossing the border of Viola – Jones Framework

The second part of this thesis begins when the requirement of testing our sub – system on algorithms that do not belong in Viola – Jones framework, becomes imperative. It is of high interest to find out how flexible it can be proved as for the changes to be made and to what extent will it accelerate some tested algorithm. For that reason, we are studying the “Face Detection, Pose Estimation, and Landmark Localization in the Wild” algorithm which is specialized to face detection while processing an image. The results are utterly interesting.

4.1 General Description of Parts Based Detector Algorithms – The “Face Detection, Pose Estimation, and Landmark Localization in the Wild” Algorithm

Part-based models refer to a broad class of detection algorithms, in which various parts of the image are used separately in order to determine if and where an object of interest exists [13]. In this thesis we examine “Face Detection, Pose Estimation, and Landmark Localization in the Wild” algorithm that was created by Xiangxin Zhu and Deva Ramanan [14]. The general idea is that the algorithm tries to detect certain parts of an object, in our case of a face, using special filters. When it is done it uses a mixture of trees in order to check the locality of the detected parts and if it responds to the real object. For example, for face detection, it will search for eyes, mouth, nose or ears and as soon as it finds them it will check if they are on the right position too and not randomly scattered in the image.

The concept is to find the portion of the code in which we spend the most of the time for data processing (bottleneck). Then, we are going to try its parallelization by implementing it in VHDL and combine it with our sub – system. If the results are encouraging, our sub - system can be easily connected to the remaining algorithm, as a production engine that consumes and produces data in real – time and at low cost, behaving as what exactly it is; an embedded system.

4.2 The bottleneck and the parallelization

Searching the algorithm from edge to edge in order to find out the part we described above, we meet the numerous access memories and computations that happen when we apply the special filters on the image. No doubt, it is a point we need to study thoroughly in order to understand its functionality and use it properly to achieve our purpose. Let's begin with a general description and come, finally, to what exactly we are going to try to parallelize.

4.2.1 The bottleneck

The certain algorithm takes as input an image and creates a pyramid of not-constant height, composed by different resized versions of the same image; the most we move up to the pyramid the smaller the image is. Afterwards, it calculates the Histogram of Oriented Gradients (HOG) for all of them. HOG is a technique that counts occurrences of gradient orientation in localized portions of an image. The thought behind it, is that local object appearance and shape within an image can be described by the distribution of intensity gradients or edge directions [11]. The specific HOGs we describe are 3D and more specifically $X \times Y \times 32$, where $X \times Y$ is the size of each image.

On this HOGs we are going to apply some filters of dimensions $5 \times 5 \times 32$. As you see the depth of the filters is the same to this of pictures. The number of filters to be applied is 99 and corresponds to all the parts we need to locate in Face Detection. So, each filter is going to be convolved with each HOG in this way; the first level of the picture is convolved with the first level of a filter, the second level of picture with the second of the filter and so on. The procedure of convolution is shown, in details, at [figure 4.1](#). As soon as we have finished with all levels we add all of them pixel by pixel and we produce a 2D data structure. This happens with all filters.

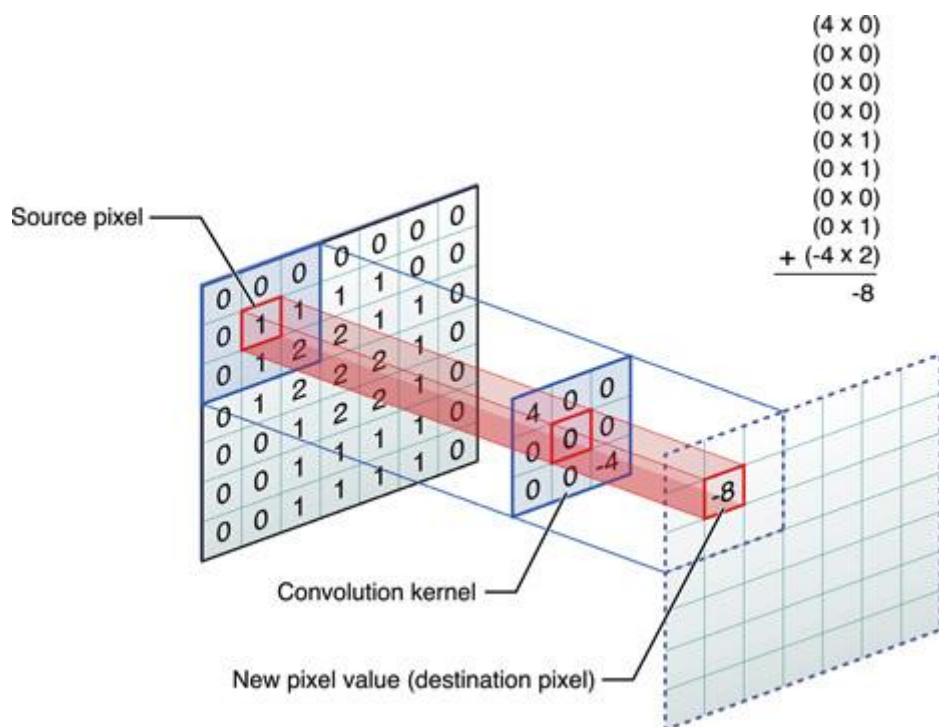


Figure 4.1 Detailed convolution between image and filter [20]

But, let's see the code that performs the above acts. Firstly, there is the function;

```
_Vector<_Array<T>*> *featureResponse(_Vector<_Array<T>*> *response,
_Array<T> *feat, _Vector<_filter<T>> *filters),
```

which has, as arguments, the HOG (feat) and all the filters. It calls the function “convolve” for each filter separately, as the second one takes a HOG and a filter and returns a 2D response;

```
_Array<T> * convolve(_Array<T> *response, _Array<T> *feat, _Array<T>
*filter)
```

The convolve function with its turn, uses a nested for of depth 5 in order to produce the response. This could be translated as follow;

```

For each one of the 32 levels (the depth of the HOG and filter)
    For each row of the pixels
        For each column of the pixels
            For each row of the 5x5 sub image and sub filter
                For each column of the 5x5 sub image and sub filter
                    Do the mul and add
```

This part demands from memory the values of HOGs and filters in order to accomplish two mathematical operations per iteration. This behavior causes million of memory accesses that delay the total process and there is need of being parallelized.

4.2.2 Parallelization

In this try we faced a lot of problems as it was not clear, in the beginning, how exactly this could be done. Having in mind the structure of our system we see that we cannot store a bigger image than of 640x480 dimensions as our memory has space for exactly 307200 pixels. Consequently, we cannot store more than one pictures in memory and this, automatically allow us to process only one level of the total 32 (we left outside the outer loop for now).

Watching carefully the two inner loops that concern the sub filter and the sub image that are convolved, we notice that, when unrolling them, they create blocks of 25 addresses. Automatically, we remember the 32 addresses that our system receives and think that with the suitable changes it could handle the 25 too. Although, this is not enough, as we need to take care of the movement that the two remaining for-loops causes to the filter. In other words, to make the sets of the 25 addresses which are going to be sent to the Collision Detect Core more specific.

Now, we can propose a total idea of how our system is going to cope with this algorithm. The most important is the design of the Loop Decoding Core. In here, we

have to create all the sets of 25 queries that are needed to complete the convolution between a 640x480 image and a 5x5 filter that is going to give us a 636x476 table. Then we store the image to memory and we use one more to store the corresponding filter. For every set we receive from Loop Decoding, we calculate its convolution with the filter in the Computation Core. When its process finishes it demands the next set of 25. When the total process is finished our embedded system informs the software to load on memories the next level of the image and filter and we start again.

The next step is to transfer the address generation to VHDL, in other words, to create the new Loop Decoding Core. For this purpose we are going to take advantage of Xilinx's tool "Vivado High Level Synthesis" for which we discuss in the next paragraph.

4.3 Vivado - High Level Synthesis

High-level synthesis is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. The goal of HLS is to let hardware designers efficiently build and verify hardware, by giving them better control over optimization of their design architecture, and through the nature of allowing the designer to describe the design at a higher level of tools while the tool does the RTL implementation [12]. Vivado is Xilinx's High Level Synthesis tool which we are going to use.

4.3.1 The C code

As designers, in this case, we need to create a C code that unrolls the two inner loops, follows the movement of the filter and generates 25 addresses or 50 ones if we want to use both the ports of the memories. This on its own was quite easy, but we have to follow a specific plan of organization to cover Vivado's requirements.

In order to give a solution at our implementation, Vivado needs the source file .c or .cpp and a testbench file in which we run the main function that calls the function we designed. So, in the first one, we did the main functionality using a struct and some if statements, while, in the second one we have just the call. After several experimentations we reached the conclusion that we need to take care of the enable signal, which is vital for the module that is going to be created, exactly here and so we act.

4.3.2 The VHDL code

After debugging, running C code and choosing a device for the solution, we are ready to synthesize it. Vivado completes its process in a few seconds, giving the .vhd files and a report that includes information about what exactly it used to design the solution. The produced .vhd files correspond to the functions we had in C design. We are going to check the gen.vhd to make sure that it produces the addresses as we wish. Indeed, as we can see in [figure 4.2](#) the results are correct.

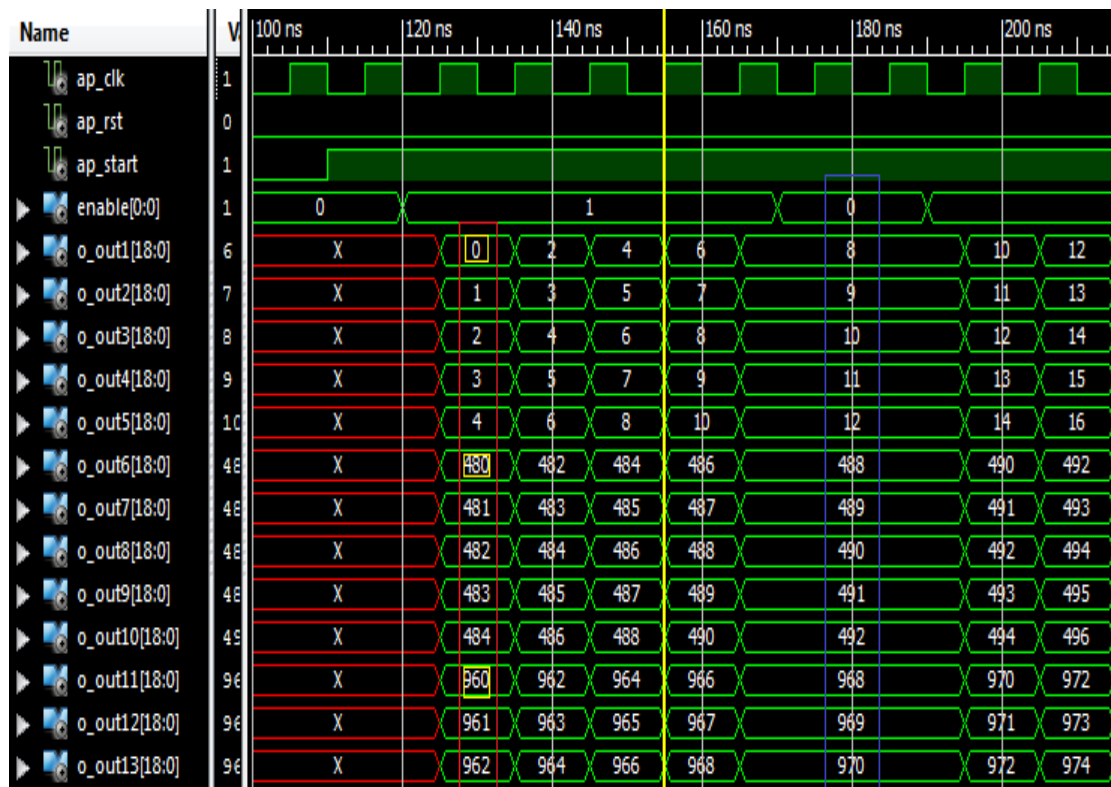


Figure 4.2 Simulation of Loop Decoding Core

In the red rectangle we show part of the first set of addresses. The smaller yellow ones declare the change of the row in the sub-image. As we have a 640x480 picture, to the change the line we need to add 480. Below it, we see that the next set begins with address 1 as the filter moves one position to the right to be re-convolved. The blue rectangle is to confirm the correct functionality of the enable. When it is disabled the address generation freezes and continues when it is enabled again.

4.4 The Embedded System tailored to “Face Detection, Pose Estimation, and Landmark Localization in the Wild” Algorithm

By now, we have completed the first part of the transformation we attempt on the sub-system in order to cope with the algorithm requirements and it is time to consummate it. The memory accesses, that the convolve function is demanding, have been calculated and sent to the system. What remains is the way in which we are going to use them. The main part that will concern us is the Computation Core, but minor changes were done almost everywhere.

The number of FIFOs is now fifty as many as the addresses we produce. The first team of 25 sends the outputs to the Collision Detect A Cores whose number remains as before (32), while the second team to Collision Detect B Cores (32) (see [figure 3.2](#) for help).

To resolve the collisions we use the previous huge FSM, but with 25 states, while the inputs reduced too. [Figure 4.3](#) presents its simulation for 25 collisions, but there is not anything new to mention here. Have in mind that the names of the inputs have changed to A0 to E4; the letter declares the row, the number declares the column.

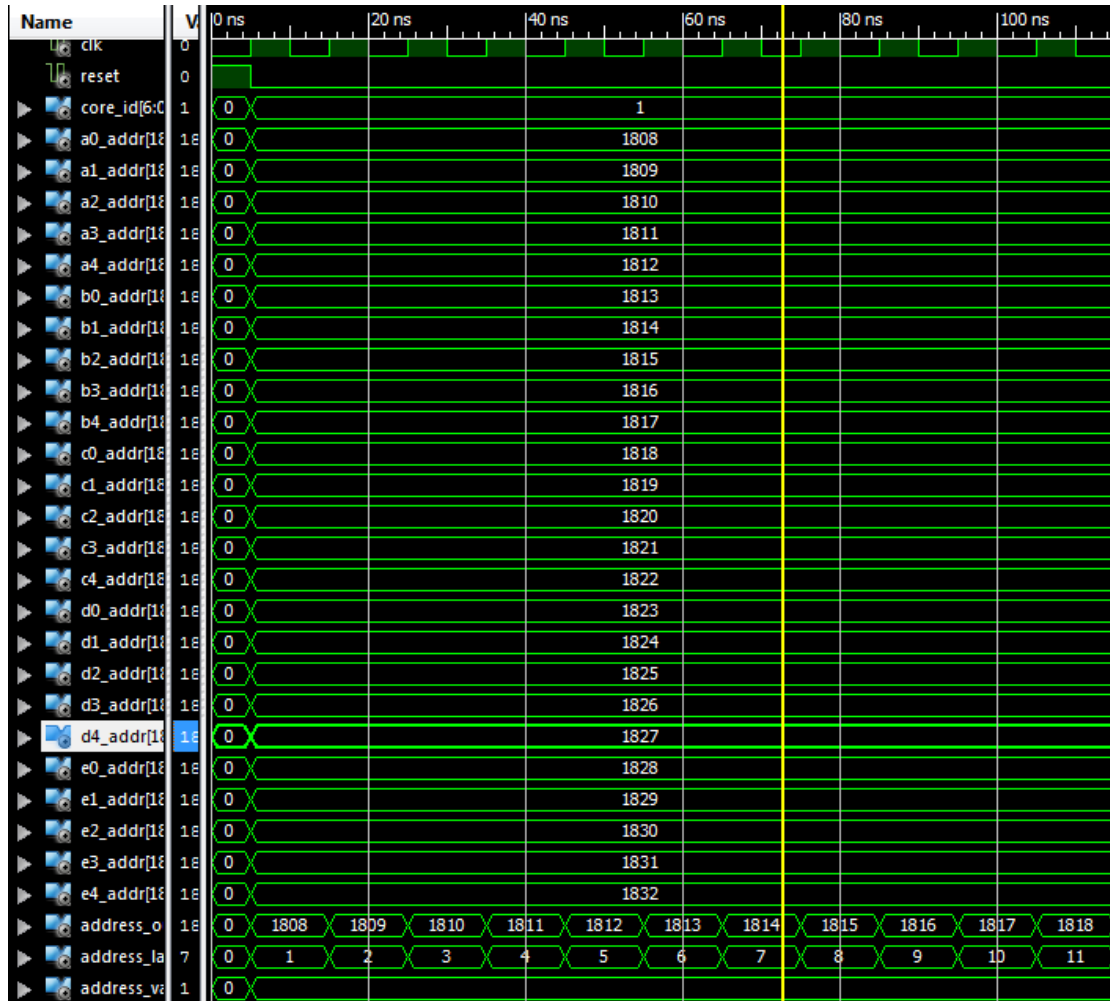


Figure 4.3 Part of simulation of 25 collisions in Collision Detect Core

4.4.1 Implementation of filter

The new part is the insertion of a memory block that holds the filter. It is similar to the blocks of the image with one difference. We have 32 such units with exact the same data; the 25 values of the filter stored in order (A0 – E4). Each of them has two ports and receives data from the same Core as the corresponding Memory Block ([figure 4.4](#)).

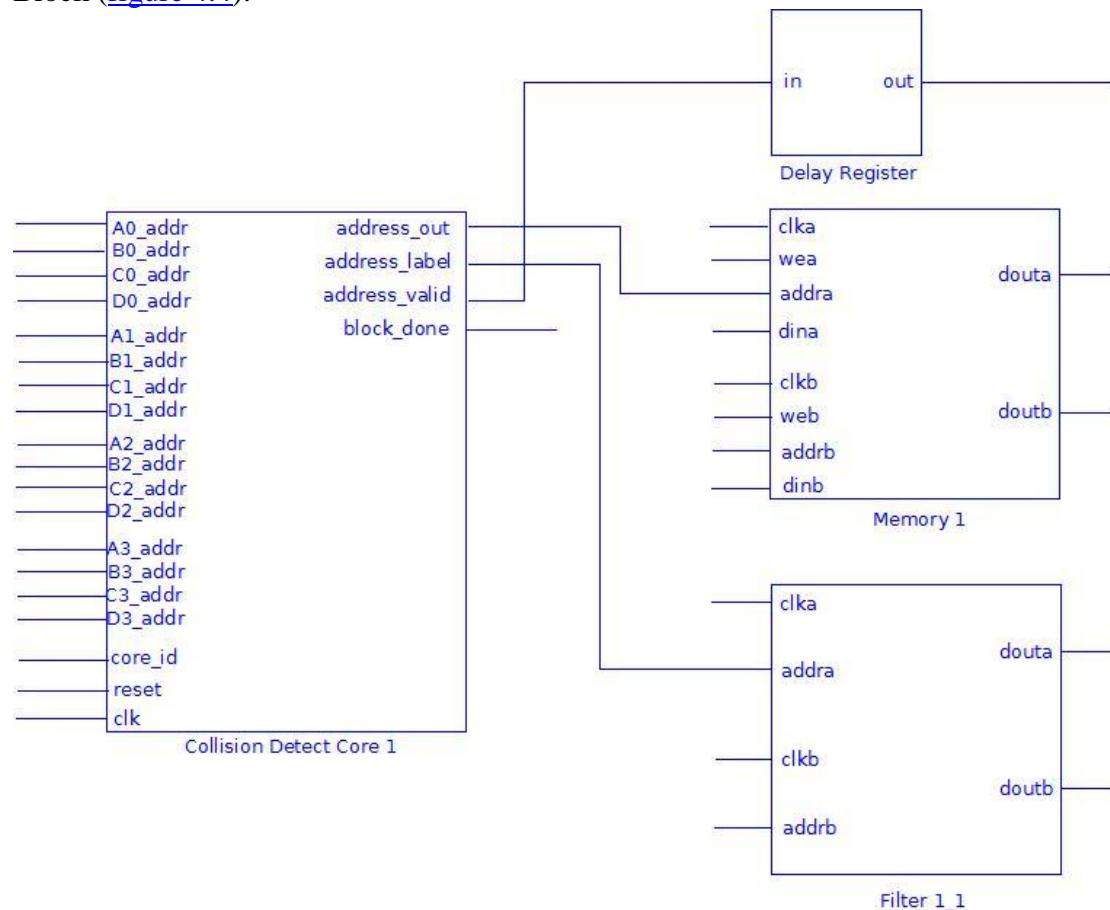


Figure 4.4

Block Diagram - The insertion of filter

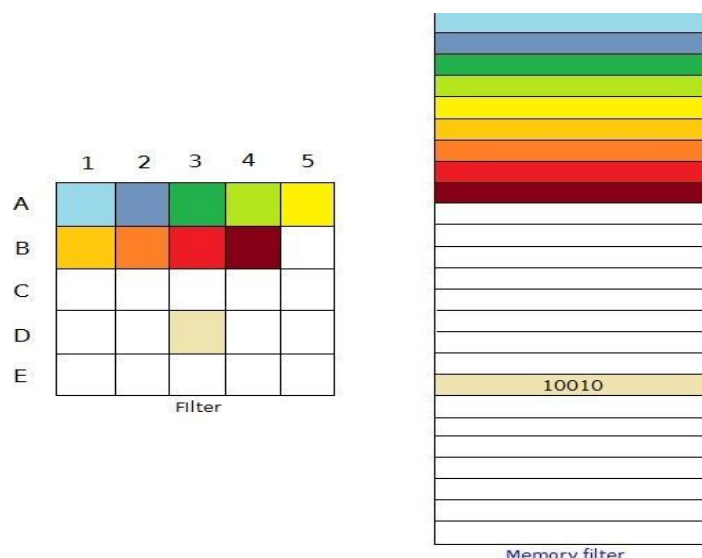


Figure 4.5

Storing filter data to filter memory

Thus, when we demand a number from memory (e.g. data of D3) we use its label to find, simultaneously, the data of this exact place of the filter with which it is going to be multiplied. The label can be used as a pointer while we have predicted to put the data in right order ([figure 4.5](#)). As a result, after a clock cycle we drive to memory both the numbers of the image and the filter in order to be multiplied.

4.4.2 Multiplications and Additions in Computation Core

Despite the previous inputs of this unit, we include the data that come from the 32 copies of the filter. It is very important to calculate all the multiplications in any order and store them somehow in order to make the final addition. For this reason we work with the same logic as before. We classify the data that come from the filters depending on the label and the valid signal, just as we did with image data. When having “0000011” as label from a block, it means that we receive the A2 image data from memory and A2 filter data from filter, simultaneously. Using the previous FSM we store in signals the filter data, too. As soon as, a couple of numbers arrive we keep their multiplication in other signals (named as “A0_mul” etc). The new element is the use of a second FSM that handles the sum. It waits until all the signals have been caught and then it adds the 25 signals that hold the multiplications. In the same clock cycle, it validates the output and informs the system to begin the process of the next set. In the next figure (4.6) we simulate an example of the correct functionality of the new Computation Core.

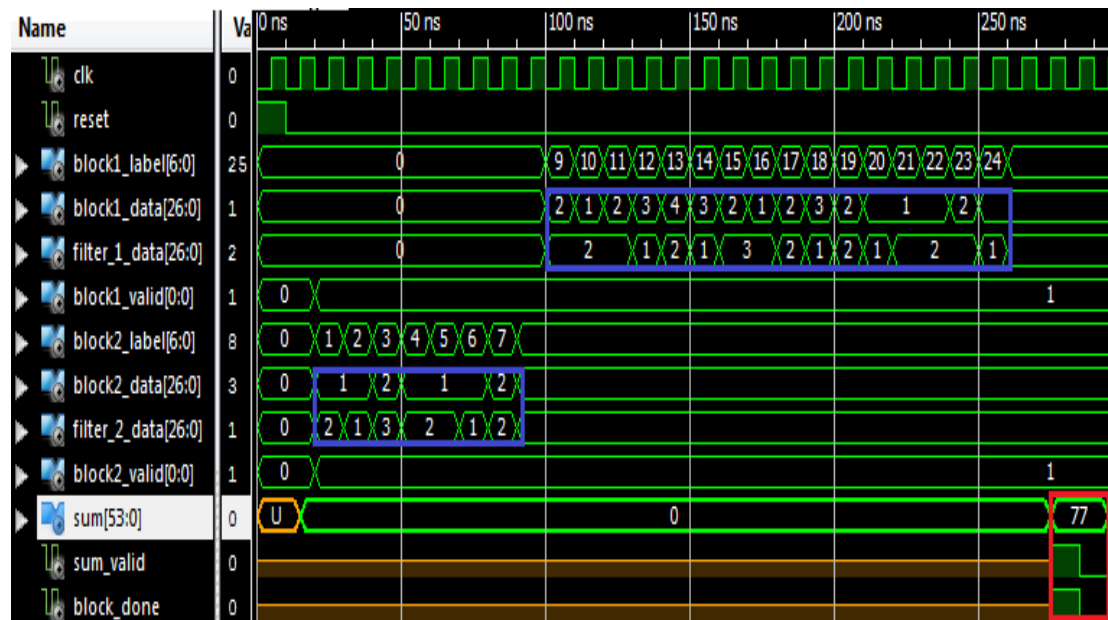


Figure 4.6 Simulation of Computation core

We choose to take all the data from only two blocks, just to prove that the result will be correct whenever we take the data to the input. In the blue rectangles are the numbers that are going to be multiplied in pairs per clock cycle. When all are calculated we export the sum (red rectangle).

4.4.3 Top Level

The new datapath of the system's top level is shown in [figure 4.7](#) and in general it was described above. What about the control unit? It is exactly the same with the previous one ([figure 3.7](#)) as it was designed to satisfy different units, no matter their functionality.

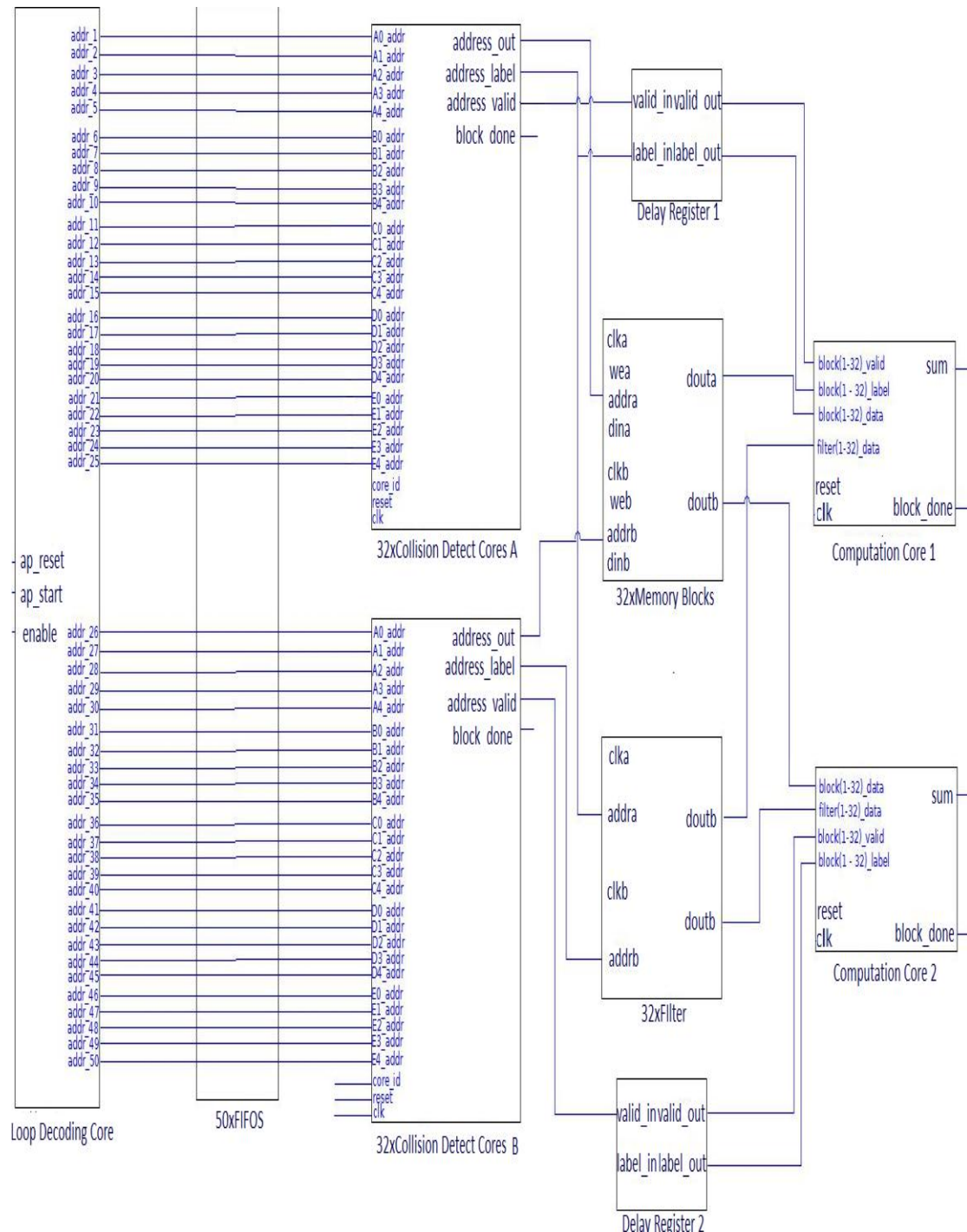


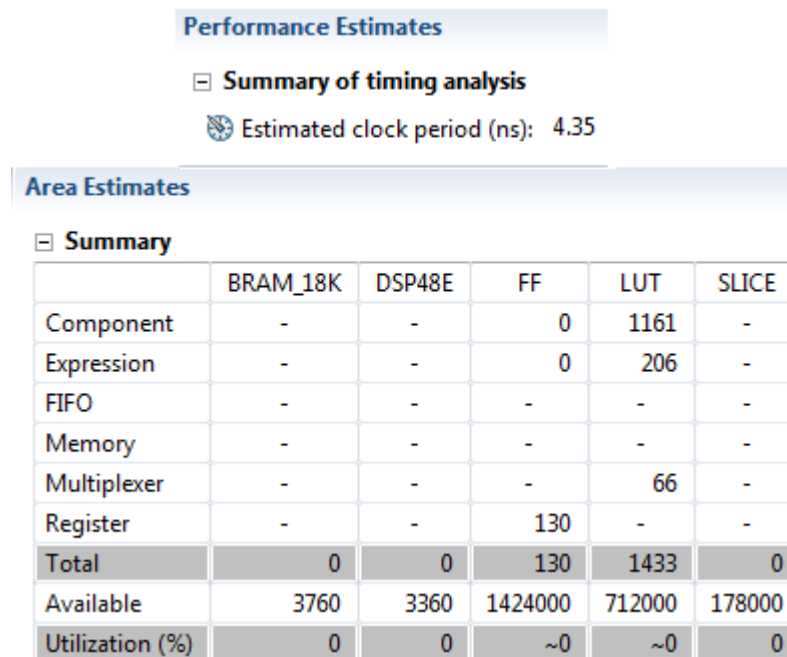
Figure 4.7 New Top Level's Block Diagram

4.5 Measurements – results

When done with the new specialized sub – system, it is time to test its functionality and its performance. Using a simple program in C we take useful measurements, as concerns the average collisions, the memory accesses per cycle and the total memory accesses, we wait during an implementation.

4.5.1 Vivado's report

After synthesis, the Vivado HLS created the following report that estimate the clock period of the hardware it produced and includes the exact elements it used.



The image shows a screenshot of the Vivado HLS report. It includes a 'Performance Estimates' section with a 'Summary of timing analysis' showing an estimated clock period of 4.35 ns. Below this is the 'Area Estimates' section with a 'Summary' table showing resource utilization for BRAM_18K, DSP48E, FF, LUT, and SLICE.

	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	-	0	1161	-
Expression	-	-	0	206	-
FIFO	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	66	-
Register	-	-	130	-	-
Total	0	0	130	1433	0
Available	3760	3360	1424000	712000	178000
Utilization (%)	0	0	~0	~0	0

Figure 4. 8 Vivado's Report

The clock period matches our standards (max 5 ns), thus the synthesis can be called successful.

4.5.2 The simple specialized sub - system

In this part, we are going to test our system without the use of any scrambling function in order to have a first aspect of its behavior. As we can see in [figure 4.9](#), we have 25 addresses corresponding to the same block, and so 25 collisions in the 66% of cases, while 20 collisions in 16.72% and 15 collisions in 16.85%. This is reasonable as the 5x5 filter demands neighboring addresses.

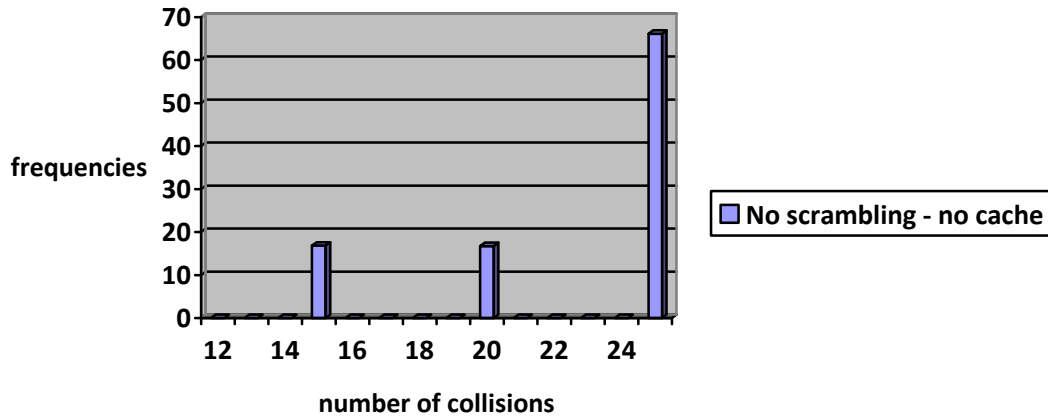


Figure 4.9 No scrambling - no cache

The above statistics have been exported after 302736 total loops, where we have;

total memory accesses (million) : 7.57
average collisions per address query : 22.5
served memory accesses per cycle using dual port : 2.226
average Pixel throughput at 200 Mhz : 445.274 Mega Pixels per sec
average throughput at 200 Mhz : 1.739 GBytes per sec

It seems that too many collisions do not accelerate our system and we need to improve these percentages.

4.5.3 The effect of scrambling function

It is very important to remind here the necessity of the scrambling function. In [1] the authors describe two functions that were tested, the XOR and bit – reordering, giving, also, statistics about how each of them reduces the collisions in each tested algorithm. We are going to use the bit reordering scrambling that works as follows.

When producing an address we choose an optimal bit which will be the reference point, where the LSBs become MSBs and the opposite. In this way we separate the neighboring pixels and send them away into different blocks. While storing the pixels into memory we apply again this function to get the correct data (1-1). In this chapter we are going to search the optimal bit in order to achieve the best result.

In [figure 4.10](#) we choose to reorder at bit 14 and then at bit 4. In first case, we receive 5.94 average collisions per address query and we serve 8.424 memory accesses per cycle using dual port. Furthermore, processing at 200 MHz serves 1684.85 Mega Pixels per sec that means 6.581 GBytes per sec. In second case, we receive 4.75 average collisions per address query and we serve 10.53 memory accesses per cycle using dual port. Furthermore, processing at 200 MHz serves 2106.86 Mega Pixels per sec that means 8.23 GBytes per sec.

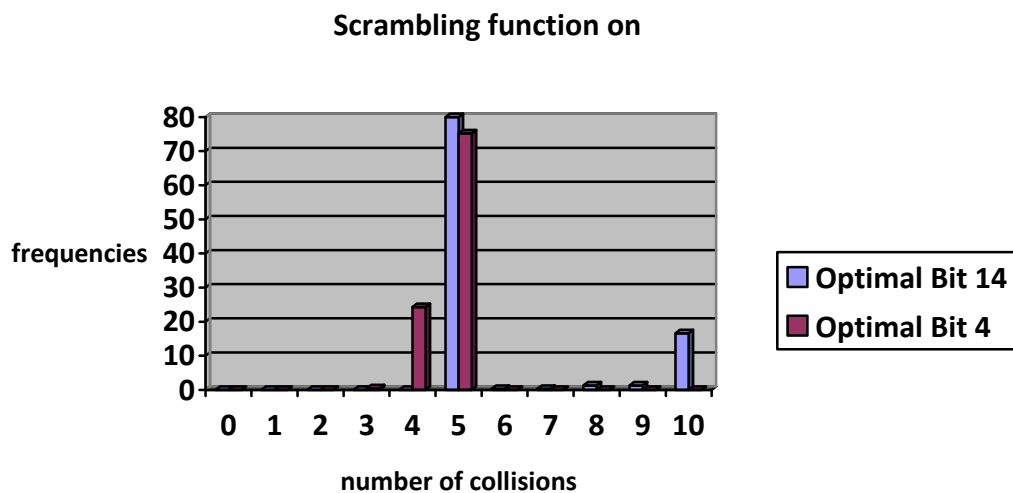


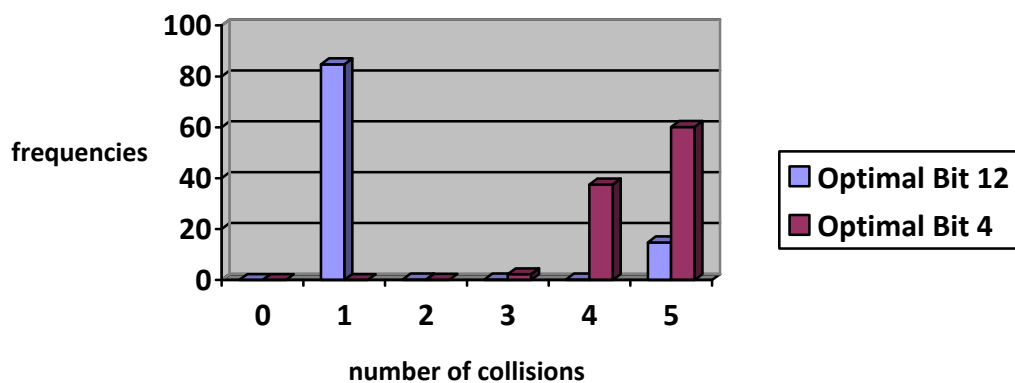
Figure 4. 10 Scrambling function on

As you can see, the biggest percentage of collisions decreases from 25 to 5 that improve the average collisions per address query. When this function is disabled we get an average of 22 collisions, as mentioned before, and this proves that not having collisions is not a product of luck but the result of this function.

4.5.4 Use of cache

Having a second, more careful look at the way the addresses are born for the simple operation of convolution, we notice that the filter moves by a pixel per iteration and as a result, from the total 25 queries almost 20 are the same. Thus, it would be very efficient the use of a simple cache in order to keep these data stored and easily accessed. For the below measurements we use a simple LRU cache that remembers only the previous set of addresses and we test again 4 and 12 as optimal bits for the partial reordering.

Scrambling function on - Cache on



4.11 Scrambling function on - Cache on

In case of optimal bit 4, we have the following results:

- average collisions per 32/25/16 address query : 4.58
- served memory accesses per cycle using dual port : 10.92
- average Pixel throughput at 200 Mhz : 2184.55 MPixels per sec
- average throughput at 200 Mhz : 8.533 GBytes per sec

In case of optimal bit 12, we have:

- average collisions per 32/25/16 address query : 1.6
- served memory accesses per cycle using dual port : 31.21
- average Pixel throughput at 200 Mhz : 6242.57 MPixels per sec
- average throughput at 200 Mhz : 24.39 GBytes per sec

It seems that the cache needs a different optimal bit to work properly but when it happens the results are impressive as in the most of cases we do not have any collisions. This does not means that our system is invalidated by the use of cache but that this is optimal in the specific algorithm that uses convolution.

4.6 Performance

Using the same device as before (Family: Zynq, Device: XC7Z045), the Xilinx report after Synthesis and Implementation gave as the following results:

- Minimum period : 4.867ns (Maximum Frequency: 205.468MHz)
- Minimum input arrival time before clock : 1.304ns
- Maximum output required time after clock : 1.109ns
- Maximum combinational path delay : 0.640ns

Once again, we achieved our purpose of not fall below 200 MHz.

4.7 Conclusion

The creation of the Loop Decoding Core was converted to a piece of cake with the use of Vivado. Building it, directly, in VHDL would, probably, take us a great amount of time to complete and would bring us in front of, too many difficulties. Now, we just designed in few lines of C code the memory pattern and the work was done. That helps us to achieve the general purpose of the system. In other words, we have, only, to find out where each algorithm needs acceleration, due to memory delays, while the memory sub-system is responsible to parallelize the data.

5 References

- [1] ANTONIS NIKITAKIS, IOANNIS PAPAETSATHIOU “Bijective Addressing in the Viola-Jones Framework : A High Throughput Run Time Adaptable Memory Sub-system”
- [2] NIKITAKIS A., PAGANOS T. AND PAPAETSATHIOU I.2014. "A novel Embedded System for Vision Tracking ", Accepted to appear in IEEE International Conference on Design, Automation & Test in Europe Conference & Exhibition (DATE' 2014)
- [3] http://en.wikipedia.org/wiki/Viola-Jones_object_detection_framework
- [4] http://en.wikipedia.org/wiki/Computer_vision#Applications_for_computer_vision
- [5] YANNIS PAPAETSATHIOU “Overview of Embedded Computing Systems”
- [6] http://en.wikipedia.org/wiki/Embedded_system
- [7] http://en.wikipedia.org/wiki/Summed_area_table
- [8] <http://en.wikipedia.org/wiki/SURF>
- [9] http://en.wikipedia.org/wiki/Locality_of_reference
- [10] <http://en.wikipedia.org/wiki/FIFO>
- [11] http://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
- [12] http://en.wikipedia.org/wiki/High-level_synthesis
- [13] http://en.wikipedia.org/wiki/Part-based_models
- [14] Xiangxin Zhu Deva Ramanan “Face Detection, Pose Estimation, and Landmark Localization in the Wild”
- [15] Changjian Gao, Shih-Lien Lu “Novel FPGA based Haar classifier face detection algorithm acceleration”, International Conference on Field Programmable Logic and Applications, 2008. FPL 2008.
- [16] Wenhao He and Kui Yuan, \An Improved Canny Edge Detector and its Realization on FPGA", in Proceedings of the 7th World Congress on Intelligent Control and Automation
June 25 - 27, 2008, Chongqing, China.

- [17] Vinod Nair and Pierre-Olivier Laprise and James J. Clark, "An FPGA-Based People Detection System", in EURASIP Journal on Applied Signal Processing 2005:7, 1-15.
- [18] Deepayan Bhowmik, Balasundram P. Amavasai and Timothy J. Mulroy, "Real-time object classification on FPGA using moment invariants and Kohonen neural networks", Proc. IEEE SMC UK-RI 5th Chapt. Conf. Advances in Cybernetic Systems (AICS 2006), pp. 43-48, 2006.
- [19] BECKER T., LIU Q., LUK W., NEBEHAY G., AND PFLUGFELDER R.. 2011. Hardware-accelerated object tracking. In Proc. Int. Conf. on Field Programmable Logic and Applications (FPL), Sept. 2011.
- [20]
<https://developer.apple.com/Library/ios/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html>