Danmarks Tekniske Universitet

DTU

02203 Design of Digital Systems

Lab 2: Edge Detector HW-Accelerator – Report

Authors

Group 9

Mariana Santos – s233360

December 1, 2023

This project was completely done by the only author listed.

Abstract

Specific hardware is usually used to optimize very specific tasks. This report delves into the design and implementation of a hardware accelerator that is able to detect edges in an image using the **Sobel operator** algorithm. Using VHDL along with XILINX Vivado, various solutions will be explored and discussed. These are programmed on the NEXYS 4 DDR Artix™-7 FPGA board. The hardware description code mentioned in this report can be found on the appendices A-**??**, along with some diagrams.

Contents

1 Introduction

Edge detection is widely used in the image processing field. Essentially, this process works by analysing the each image's pixels, and detecting discontinuities in the brightness. Having only information about the edges, instead of the whole picture, allows for multiple diverse algorithms to process images quickly and efficiently. One of the best well-known applications for this technique is object detection.

1.1 Images

This project aims to explore a hardware implementation of a well-known edge detection technique. The images used in this scope are represented in *PMG* format, and are 352x288 pixels. Each pixel is represented by an 8-bit value – i.e., an integer between 0 and 255, where 0 represents black, 255 white, and the values in between shades of gray.

1.2 Algorithm

The algorithm used in this project is the **Sobel operator**. It successfully identifies the intensity changes in the pixels of an image by a process called convolution. The Sobel operator consists in doing a convolution between a image with two 3x3 kernels/matrices – one for detecting changes in the horizontal direction, and the other for the vertical direction. These are 2 kernels shown below on (1).

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{1}$$

These two kernels are convolved with each single pixel in the original image, and at the end, an image where the "edges" are more pronounced will be obtained.

1.3 Hardware and protocol

The piece of hardware to be designed should act as a accelerator. It should be able to receive a request signal from the CPU, and to be able to access the memory, if it were to be implemented in a real-world scenario. A schematic of how this structure could look like is shown on Figure 1.

In the scope of the project, the `start` input signal is replaced by a button press, and the `finish` will correspond to a light on the board. When the `start` signal is set to active, the image processing will be triggered, and the output signal `finish` will be activated once this processing is finished. Then it will be ready to receive new processing requests.

The image is stored on a RAM memory on the FPGA. This memory has a word size of 32 bits, which means that a word contains 4 pixels.Figure 2a shows how pixels are referred to when addressed in this report. A schematic of the memory, and how the image is stored in it is shown on Figure 2b. Note that this last schema differs from the one on the project

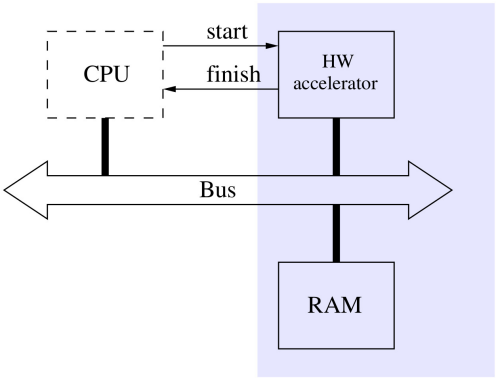
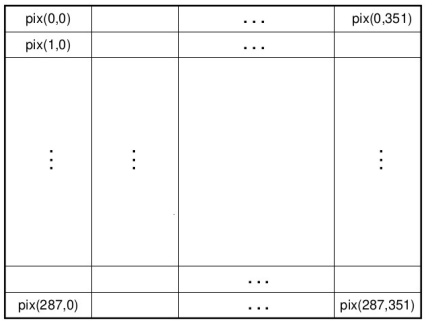
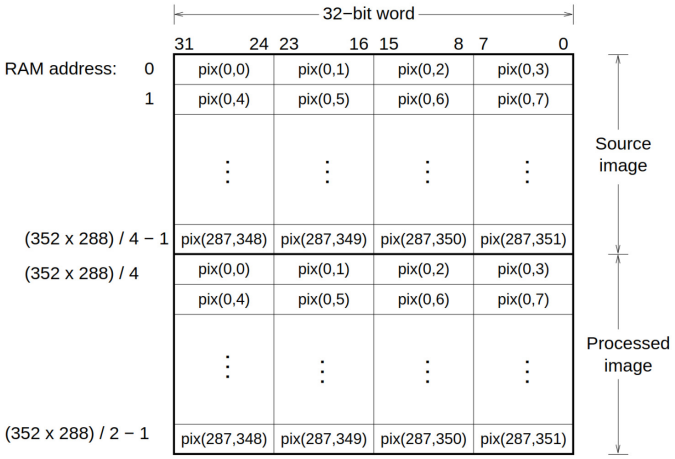


Figure 1: Schematic of how the hardware accelerator could be used in a real-world system. Image taken from the project guide [1].

guide, since it reflects the actual system created (this is in accordance to the project files that we were given).



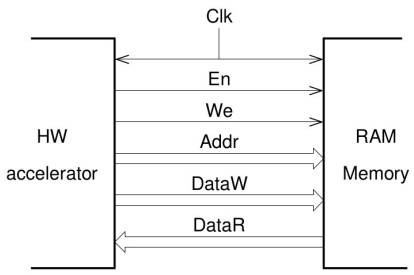
(a) Schema of an image, showing how the pixels are identified. Image taken from the project guide [1].



(b) Diagram depicting how the image pixels are stored on main memory. Base image taken from the project guide [1], but **altered** to reflect the true schema.

Figure 2: Representations of an image and how it is stored in the main memory of the system.

The interface to communicate with the RAM memory is shown on Figure 3. Throughout this report, this memory might also be referred to as *main memory*. `DataW` and `DataR` correspond to 32 bit data – a word. Both reading and writing operations are asynchronous: the reading data is only available on the next clock cycle, and the same happens for data to be written. Figure 4 is a timing diagram showing how communication with this memory looks like.



En	We	Meaning
0	X	No transaction
1	0	Read transaction (load word)
1	1	Write transaction (store word)

Figure 3: Schematic of the RAM memory interface. Image taken from the project guide [1].

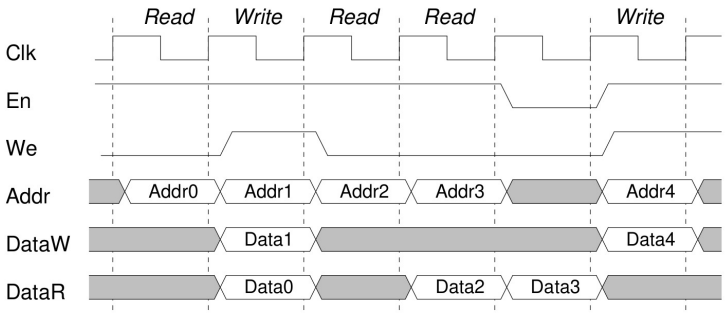


Figure 4: Example timing diagram of a communication situation with the RAM memory. Image taken from the project guide [1].

2 Task 0: Getting familiar with a simpler project

With the goal of getting familiarised with the system's components, a simpler problem was analysed: that of inversing each one of the picture's pixels. That is, for each pixel, perform the operation shown on (2).

$$pixel <= 255 - pixel \tag{2}$$

2.1 State diagram

To tackle this problem, the first step was the design of a state diagram for the Finite State Machine with Datapath (FSMD). Figure 5 shows the final state machine diagram for the task. The RT-operations – such as storing data on register, or indicating output data – are annotated in each step, while the transition annotations aim to indicate which state should be the next one.

Since this is such a simple problem, it can be interesting to go through all the states and explore what happens in each one.

- **S0:** The system is in this state while it is waiting for the `start` signal to be activated. The read address (`addrR`) and the write address (`addrW`) registers are set to their initial values.

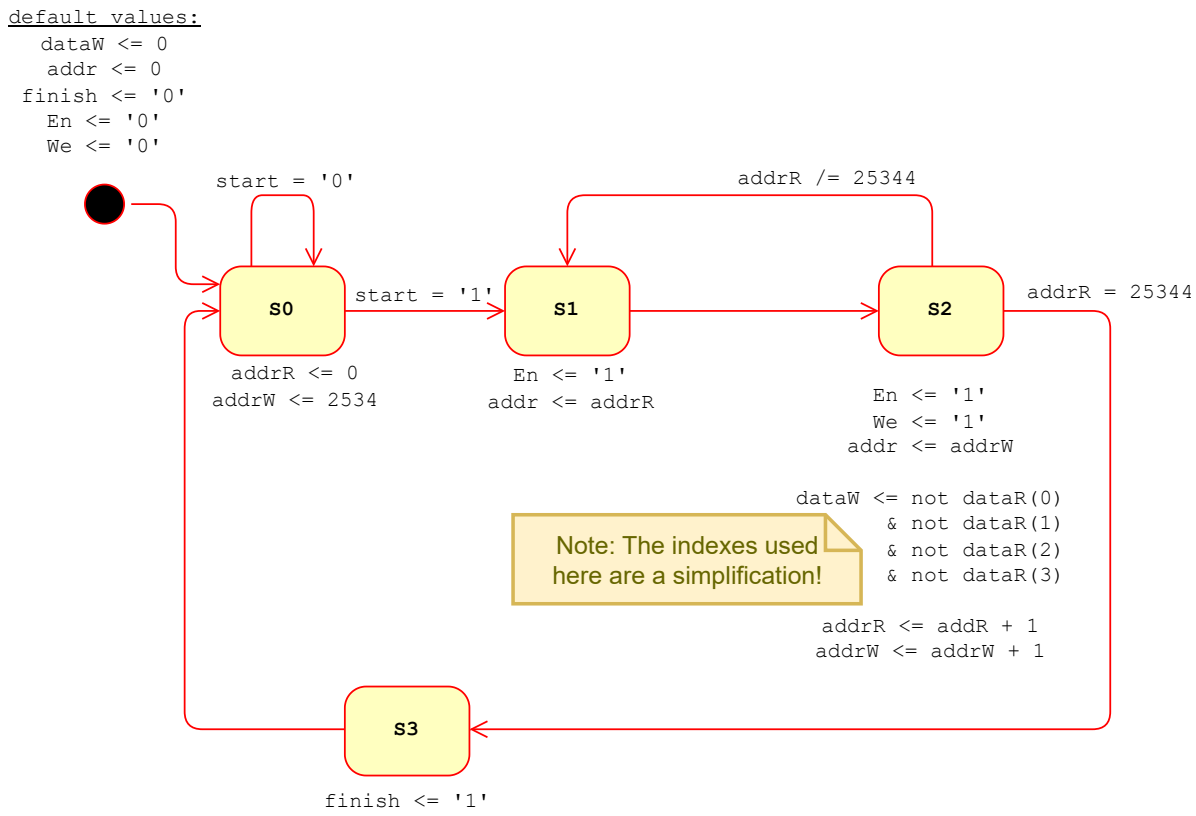


Figure 5: State diagram for the Finite State Machine with Data-path (FSMD).

- **S1:** In this state, the system will send a `read` request to the memory for the address `addrR`.
- **S2:** In this state, the memory will return the data that was requested (`dataR`) – 4 pixels of the image. Then, each one of the pixels is inverted immediately, and sent back to the memory to be written on the write address (`addrW`). Then, the register values of these two addresses are incremented by one. At this point, two things can happen:
  - if the whole output image has been written to memory, then the next state will be `S3` – the final one.
  - otherwise, the next state will be the previous one, `S1`, to allow for more data to be read and written.
- **S3:** Last state before the machine is ready for a new image: the `finish` signal is activated, and the next state is `S0`.

2.2 General considerations

The bottleneck of this design should be accessing the main memory, since 2 cycles are always needed to read and write a 4-pixel word. Since an image is always 288x352, then there are a total of $288 * 352/4 = 25344$ 4-pixel words. As discussed, each word needs 2 cycles to be processed and saved. Additionally, the FSM contains an extra state at the start and end of this process. It is possible to approximate the time it takes to process an image as shown in (3). Here it is assumed that the clock period is 80ns, since that is the default value set in the files made available for the project (`test2.vhd`).

$$t = 80 * \frac{288 * 352}{4} * 2 + 2 = 4055042\text{ns} \tag{3}$$

This means that this design is capable of processing approximately 247 images per second, as demonstrated in (4).

$$num\_imgs\_per\_second = \frac{1}{4055042 * 10^{-9}} \approx 247 \tag{4}$$

In terms of speed, the only way to increase the throughput would be to increase the clock speed of the system, since the main memory is already in use in the most efficient way.

2.3 Block diagram with datapath

A possible schematic of the datapath for the system is shown on Figure 6. It shows the flow of data, and depicts the simplicity of the system: the data only goes through one compute component that will invert the pixel, before it is saved back into the main memory.

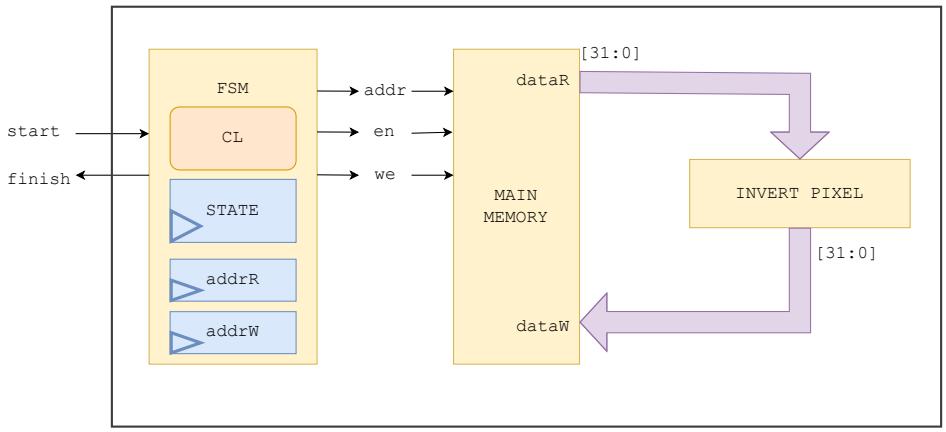


Figure 6: Block diagram with datapath for the inverter project.

2.4 ASMD-chart

The Algorithmic State Machine with a Datapath (ASMD) chart can be found on Appendix A. It depicts both the Finite State Machine operations and states, but also describes clearly the control flow, and the register operations of the datapath. It can be easily translated to actual VHDL code.

2.5 Specifying with VHDL

The VHDL code for this project is fairly straightforward, and can be found in Appendix B.

2.6 Simulating system

Using the files provided in class, the system was simulated. Figure 7 shows a piece of the timing diagram resulting from the simulation of the design. For the portion shown on the image, it is clear that the memory, which is the bottleneck of the system, is always on use: the enable (`en`) sign is always activated, and the address bus is also always changing, with data being read and written constantly.

Running the simulation takes 4055520ns. This value is very close to the value obtained in subsection 2.2.

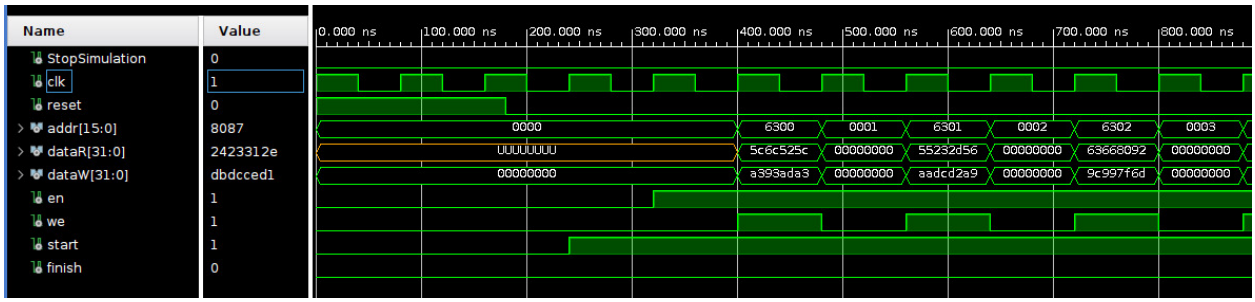


Figure 7: Portion of the timing diagram obtained from simulating the inverter project.

Figure 8 shows the original image used to simulate the system, and the result obtained.

2.7 Synthesizing design

This smaller project was synthesized and tested on the NEXYS 4 DDR Artix™-7 FPGA board. To upload and download the images to the board, the program supplied with the project files was used. Figure 9 shows the input and the image obtained, as well as the interface of this program mentioned.

Figure 8: On the left, there's the original image which served as input for the accelerator. On the right, the image produced by the inverter project.

3 Task 1 & 2: Designing a first version of the hardware edge detector

When tackling the real challenge, the approach taken was the following: first, a draft version should be designed – something that produces the expected output, but that does not cover edge cases. Performance or hardware resource considerations weren't important. In this section, this simpler approach will be discussed.

The assumptions that this draft was based on are as follows:

- The edges of the image shall not be computed correctly. The edges of the output image shall be set with the value 0, so that only 350x286 pixels are actually computed.
- The system shall not cache a big amount of data for processing. An additional FPGA RAM entity should not be used at this point, only registers.

3.1 Block diagram with datapath and procedure

Figure 10 shows a possible block diagram with datapath representation for this system. It clearly shows that there are 3 registers that store data from the main memory. These are 4-byte registers (i.e., each one can store 4 pixels), and each one corresponds to one different line in the image. What is exactly stored on them will be discussed shortly. The `COMPUTE SOBEL` represents the actual computation to perform the convolution. In practice, is a set of additions and multiplication operations. The result is then stored in an output register `OUT`, which will be written back to memory depending on the current FSM state.

The following images describe in more detail the process of getting the data to the registers, and how the output buffer is filled up. Figures 12 to 14 shows steps in the completion of the 4 pixels that are needed to be written on the main memory. The larger table on the left represents the image, and contains information on the data that is stored on the registers, and the data that is being read by memory. The smaller table on the right is a representation of the output image that is being created. **??** shows a legend for the images.

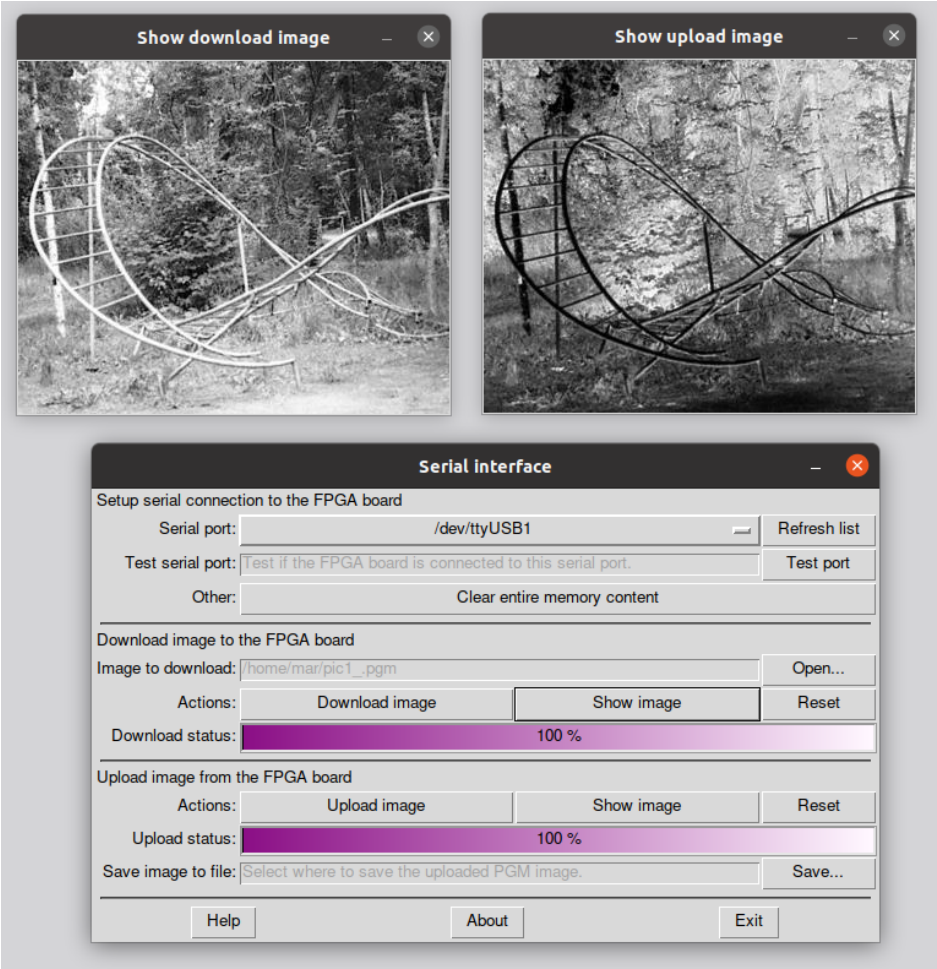


Figure 9: On the top left corner, there's the original image which served as input. On the top right corner, there's the output image as processed by the FPGA board. On the bottom, there's the interface of the program used to upload and download the images to and from the board.

Note that the first row on the output is set with the value 0. All the edges are set with 0 by default, since they will not be calculated, as already stated.

The first image, Figure 12, shows that the first 4 pixels of every line are read, and stored in the respective registers. With this information, it is possible to compute the 2nd and 3rd pixels of the output buffer. Note that last pixel is needed before this buffer can be written in the database.

In order to compute this last pixel, more information is needed. In fact, to compute it, some pixels from the current word, and some from the next word are needed. Therefore, it is necessary to combine the two. Figure 13 shows how this is done. A new word is read from memory, and only the two first pixels are stored in the register. The register will now contain its own last two pixels, along with these 2 new ones. After computing this last pixel, the output buffer is stored on main memory. With the current register information, the first

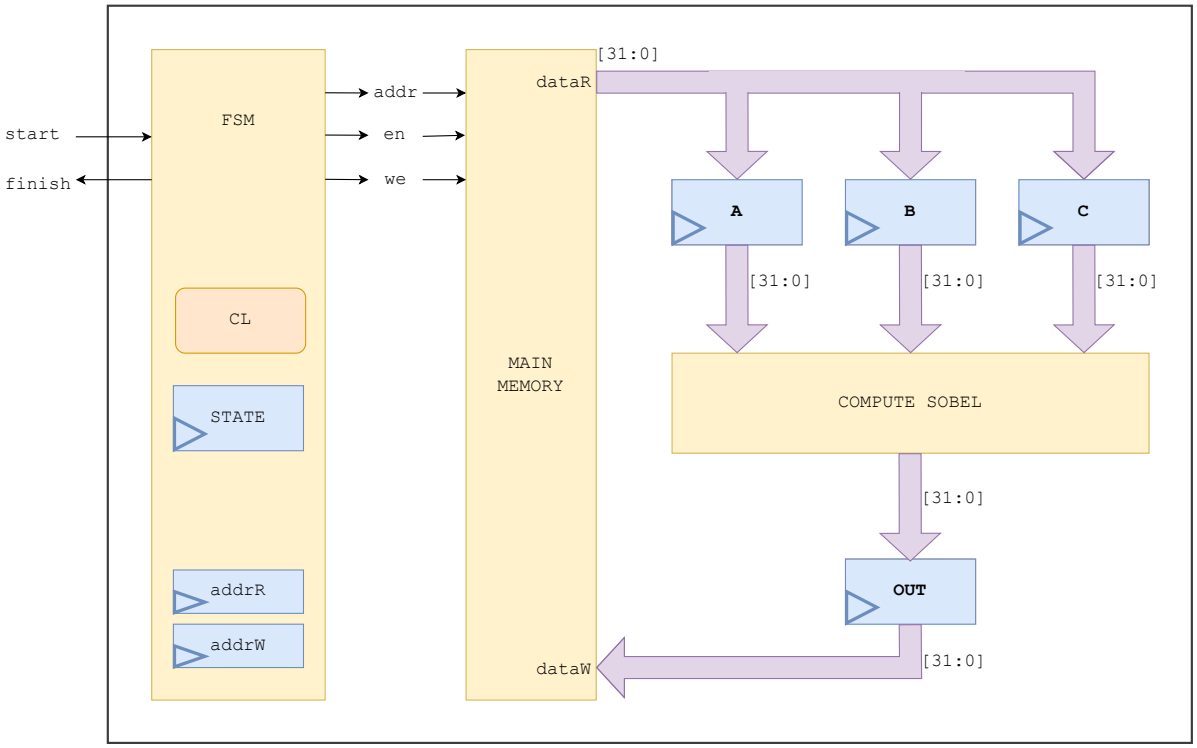


Figure 10: Block diagram with datapath for the first version of the edge detector system.

Figure 11: Legend for the following schematics.

pixel of a new output buffer should also be computed.

After these steps, the situation is very similar to the one in the first step, as shown in Figure 14: the 1st output pixel is already computed, and there is information to compute the 2nd and 3rd ones. This process will be repeated until the end of the line is reached. Then, all will be repeated for the following lines. In average, each line will be read into a register 3 times.

Another thing that is worth mentioning is **truncating**. When computing the Sobel convolution, the maximum value that can be obtained is 1530, which in binary looks like `10111111010`. The problem with this number is that it is not representable with only 8 bits, it needs 11. To solve this problem, some values are truncated, and only the 8 most significant bits are used. Effectively, this number becomes `10111111`, which is 191 in base 10. Even though this is a solution to the problem of representing the value in less bits, it is not a perfect solution. The maximum value of any pixel in the output image will be 191, instead of 255, which will make it appear darker than it should. A better solution could be to use a mapping function to compute the output bit based on a better distribution. However, truncating uses next to no additional hardware, making it an attractive solution

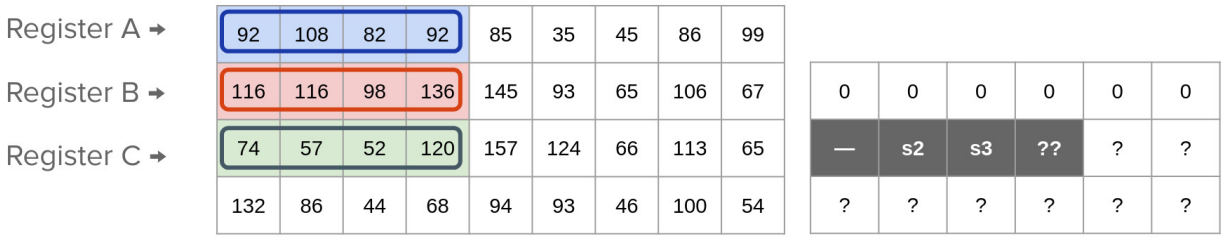


Figure 12: Schema depicting the first phase of the workflow of the first version of the system: a word is read from each line and stored. The 2nd and 3rd pixel of the output buffer are computed.

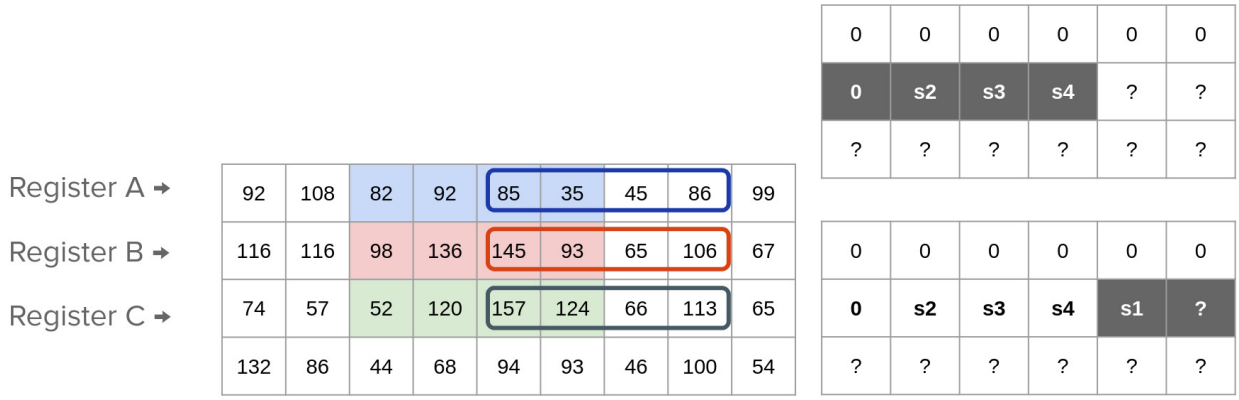


Figure 13: Schema depicting the second phase of the workflow of the first version of the system: a new word is read from each line and only partially stored. The 4th pixel of the output buffer is computed, as well as the 1st of the next one.

as well. This technique was also used in the optimised version of this project.

3.2 ASMD-chart

The ASMD-chart for this design is divided into 3 parts, and can be found in Appendix C.

3.3 General Consierations on design

This time, the memory usage is not optimised, so it cannot be considered to be the bottleneck of this design. In the majority of cases, a new 4-pixel word is written every 11 states – and thus, every 11 clock cycles. Considering the clock cycle to be 80ns as before, and the edge pixels not to be computed, then it should take about 22022000ns to process an image, as shown on (5).

$$t = 80 * \frac{(288-2) * (352-2)}{4} * 11 = 22022000\text{ns} \tag{5}$$

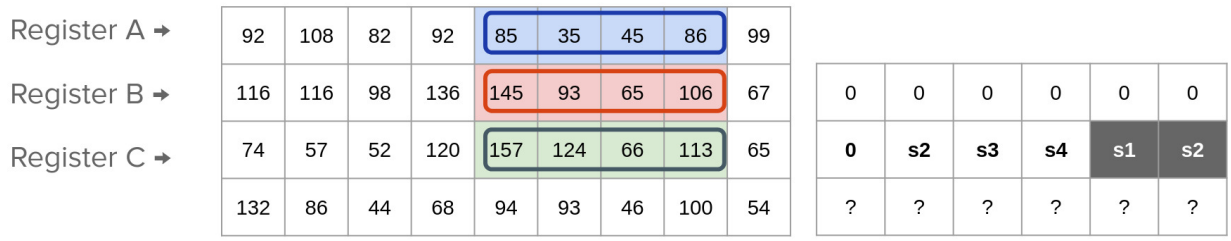


Figure 14: Schema depicting the third phase of the workflow of the first version of the system: this step is similar to the first one, and the process repeats from here on.

Therefore, a hardware accelerator with this design is able to process approximately 45 images per second, as demonstrated in (6).

$$num\_imgs\_per\_second = \frac{1}{22022000 * 10^{-9}} \approx 45 \tag{6}$$

In terms of performance and throughput, it is clear that this design is not appropriate.

3.4 Specifying with VHDL

This design was specified with VHDL in XILINX Vivado with a two-process method. A two-process state machine is a design pattern used to create finite state machines, such as the one being discussed. In short, it involves 2 processes:

1. **combinational logic process**
   Defines how the states should change. Is sensitive to the input and current state, and doesn't need the clock signal.

2. **synchronous process**
   Is responsible for updating the registers, and resetting the system. Uses the clock to ensure this happens at the correct moment.

The code can be found on Appendix D. Due to the additional challenge in this project, some new approaches were taken during its development. The main goal of using these VHDL features is to obtain a more readable hardware description file, and at the same time to reduce the probability of errors during development.

1. **constants**
   Constants were used in the combinational logic process to facilitate the writing of code. Instead of writing the same large numbers over and over, simpler identifiers could be used instead. In this case, constants were used to store the **number of addresses in the image**, and also the **number of addresses related to one line in the image**. It is not possible to change their value after the compilation of the VHDL file. They are simply used to avoid the repetition of the same values in multiple places, since that is a situation prone to result in errors.

2. **pure functions**
   Similarly, functions were used with the goal of allowing for large (or not so large) pieces of code not to be replicated into different parts of the code file. This makes the code much smaller, easier to follow, and less prone to coding errors – if something needs to be fixed, it can be fixed in just one place, instead in all the places where the code block is replicated. The use of functions should not have an impact on syntheses, since the synthesizing tool should translate the function description into hardware just as if it was described without the use of functions. Functions must always return something. In this version, only **pure** functions were used. The difference between *pure* and *impure* functions is discussed further in subsection 4.6.

3. **variables**
   In this implementation, variables were used only inside functions, even though they could be used outside as well. These are sequential elements. In this case, they were also used to make code more readable. They were used as a way to *store* the value of a computation, for example, so that the code could be separated in multiple lines. They could be eliminated by *copying* the code related to each variable, and *pasting* it where the variable is used. Similarly to before, the design is synthesized and implemented as expected.

One operation that was used while it shouldn't be was the `rem` operator, which is the modulus/remainder operator. This topic will be delved further on subsection 6.2.

3.5 Simulating design

The design was successfully simulated with the files provided. A portion of the timing diagram obtained is shown on Figure 15.

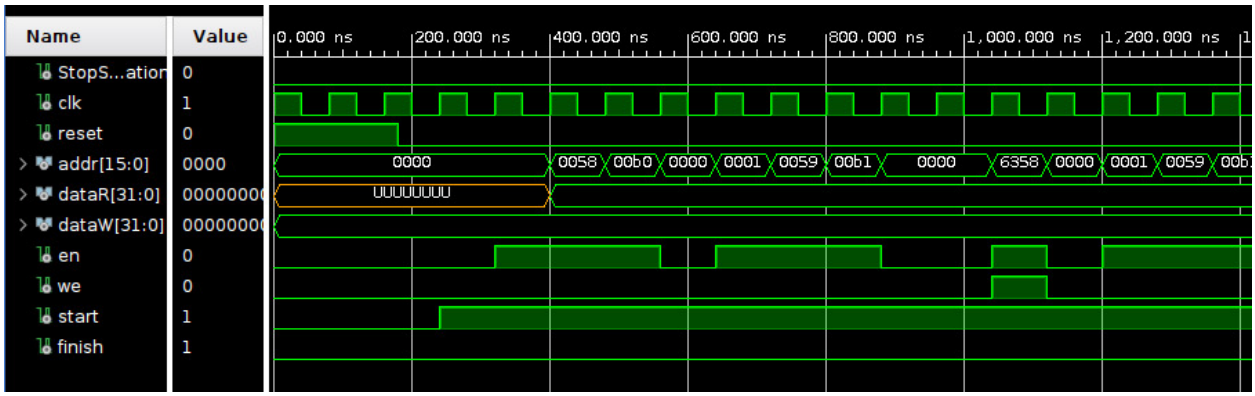


Figure 15: Portion of the timing diagram obtained from simulating the first version of the edge detection project.

From the image, it is clear that the main memory isn't always in use – the `en` signal is often deactivated – which is an obvious sign that the memory resources are not used to the maximum. Furthermore, while not visible on the image, the same words are normally read

more than once during processing. Since the main memory should be the bottleneck of the system, data should be accessed as little as possible, and it should be in use the maximum of time, so that it does not stay idle. This proves the point that this design has some issues. However, it produces a valid and correct image (except the borders, as discussed). Executing the simulation on an image takes 22056640ns, which, once again, is very similar to the expected value.

Figure 16 shows the original image used in the simulation, and the result after processing it with this design.

Figure 16: On the left, there's the original image which served as input for the accelerator. On the right, the image produced by the first version of the edge detection project.

4 Task 4: Optimizing design: computing edges and incorporating a RAM

Having a "simpler" version designed, the next step was to re-design the system without the limitations imposed. This second version of the design should be able to compute the boundary pixels, and be more efficient. A big goal for this part was to explore the use of the FPGA's memory resources.

4.1 How to compute edges

To deal with the boundary conditions, the technique chosen was the one described on the project guide: mirroring. This could be defined as expanding the original image in each edge, mirroring the adjacent boundary pixel. In the end, the resulting image would be 290x354 pixels. When computing the Sobel operator on this new temporary image's pixels, the processed image will be 288x252, which is the size of the original image. Figure 17 illustrates this technique.

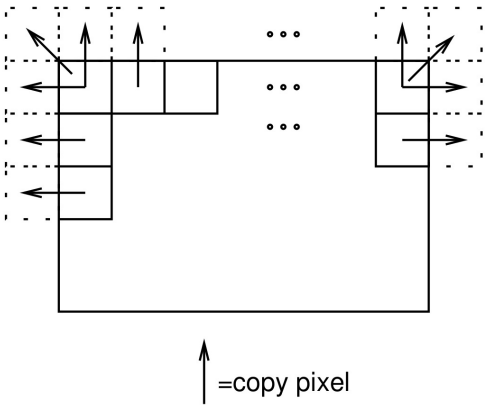


Figure 17: Schema illustrating the mirroring technique to support boundary conditions. Image taken from the project guide [1].

4.2 RAM chosen

For this context, there are two RAM types that could make sense to use: **Distributed RAM** (also known as **LUT RAM**), and **Block RAM** (BRAM). The main characteristic that differentiates between the both options for this project is the fact that the LUT RAM supports *asynchronous reads* while BRAM only supports *synchronous reads*. In both cases, the write operation is *synchronous*. In this project, a distributed RAM was used. The main reason for this is that having *asynchronous reads* would make the circuit simpler to design.

The process of how this RAM came to be is explored in section subsection 6.1. Figure 18 shows a schema of how the RAM designed is organised. It uses 32-bit words. In total, it can hold 360 pixels, but only 354 are stored. The first and last pixel of the row is replicated to accommodate for the computation of the edge cases. The $\boldsymbol{x}$ represents any possible row.

An unusual characteristic of this RAM is that the input and the output data bus have different sizes. The input data bus is the same size of a word – 32 bits. Figure 19a illustrates which bits are altered when a write operation is performed on the address 2, as an example. In fact, the data can be fed from the main memory to the RAM without any processing, with the exception of the first and last pixels.

On the other hand, the read operation isn't so straightforward. This RAM was designed with the goal of using it to compute a whole output 4-pixel buffer at the same time. For this, each RAM needs to be able to return 6 pixels during each read operation. Figure 19b shows how this operation behaves. When a specific address is read, the RAM will return data such as shown in (7).

$$\begin{aligned} &\textit{read } \mathbf{addr} \textit{ from } RAM: \\ &\\ &\textit{out} <= RAM(\mathbf{addr})(7 \textit{ downto } 0) \\ &\quad \& \; RAM(\mathbf{addr}+1)(31 \textit{ downto } 0) \\ &\quad \& \; RAM(\mathbf{addr}+2)(31 \textit{ downto } 24) \end{aligned} \tag{7}$$

	31 – 24	23 – 16	15 – 8	7 – 0
0	*undefined*	*undefined*	*undefined*	pix(x, 0)
1	pix(x, 0)	pix(x, 1)	pix(x, 2)	pix(x, 3)
2	pix(x, 4)	pix(x, 5)	pix(x, 6)	pix(x, 7)
	...			
87	pix(x, 344)	pix(x, 345)	pix(x, 346)	pix(x, 347)
88	pix(x, 348)	pix(x, 349)	pix(x, 350)	pix(x, 351)
89	pix(x, 351)	*undefined*	*undefined*	*undefined*

Figure 18: Schema of the distributed RAM designed. The numbers on the left represent the addresses, and the numbers on top the bits within the 32-bit word.

4.3 Block diagram with datapath and procedure

A schema representing the path of data in the system is shown on Figure 20. It shows that data is loaded into 3 different RAM's. The FSM controls which of the RAMs should be accessed for writing, through the signals `ram_we0`, `ram_we1` and `ram_we2`. Which RAM is chosen depends on the number of the row that is being written. The first and last rows of the image are written into two of the RAM's, allowing us to process the edge cases without changing the computation algorithm.

When the 3 RAMs have data, and every time a new row is written to one of them, the Sobel operator will be computed for a new whole output line. It will then be stored in the main memory, word by word.

Essentially, there is a big reading phase, where a new line will be read into one of the RAMs. Then, there is a big computing and writing phase, where data is fetched from the RAMs, calculations are done, and the result is stored back into main memory. The main memory is never idle with this algorithm. However, its use is not completely optimized. This will be discussed on subsection 4.5.

4.4 ASMD-chart

The ASMD-chart is shown on Appendix E divided into 2 parts.

4.5 General Considerations on design

The memory is not completely used to the best: it reads two times the first word of a row, and the last one as well. It also reads the whole last row twice. In this section, it is also

	31 24	23 16	15 8	7 0
0	*undefined*	*undefined*	*undefined*	pix(x, 0)
1	pix(x, 0)	pix(x, 1)	pix(x, 2)	pix(x, 3)
2	pix(x, 4)	pix(x, 5)	pix(x, 6)	pix(x, 7)
	...			
87	pix(x, 344)	pix(x, 345)	pix(x, 346)	pix(x, 347)
88	pix(x, 348)	pix(x, 349)	pix(x, 350)	pix(x, 351)
89	pix(x, 351)	*undefined*	*undefined*	*undefined*

(a) Schema of the RAM, showing the pixels that are altered when the address 2 is **written**.

	31 24	23 16	15 8	7 0
0	*undefined*	*undefined*	*undefined*	pix(x, 0)
1	pix(x, 0)	pix(x, 1)	pix(x, 2)	pix(x, 3)
2	pix(x, 4)	pix(x, 5)	pix(x, 6)	pix(x, 7)
	...			
87	pix(x, 344)	pix(x, 345)	pix(x, 346)	pix(x, 347)
88	pix(x, 348)	pix(x, 349)	pix(x, 350)	pix(x, 351)
89	pix(x, 351)	*undefined*	*undefined*	*undefined*

(b) Schema of the RAM, showing the pixels that are output when the address 0 is **read**.

Figure 19: Representations of the RAM data during read and write operations.

considered that the clock period is 80ns. An approximation of the time the system uses on reading the pixels and store them in the caches is calculated on (8). The 2 (in $2+\frac{352}{4}$) comes from the fact that two extra cycles are used to read the first and last word in every image line. The 1 (in $1+288$) comes from the fact that the last row is read twice.

$$t\_read = (2 + \frac{352}{4}) * (1 + 288) * 80\text{ns} \tag{8}$$

In (9) is the calculation of the time the system uses on computing the Sobel operator and storing it on memory.

$$t\_write = \frac{352}{4} * 288 * 80\text{ns} \tag{9}$$

Combining these two together should give an approximate value of how long it takes to process a whole image with this design. (10) shows that this value is 4062082ns.

$$t = t\_read + t\_write = 4062082\text{ns} \tag{10}$$

Calculating the number of images that it can process in a second can be done as shown on (11). For this design, this value should be around 246, which is a big improvement in comparison to the previous design. This value is still below what could be achieved, and as thus there are still improvements that could be made to improve this design, but these are not going to be explored further.

$$num\_imgs\_per\_second = \frac{1}{4062082 * 10^{-9}} \approx 246 \tag{11}$$

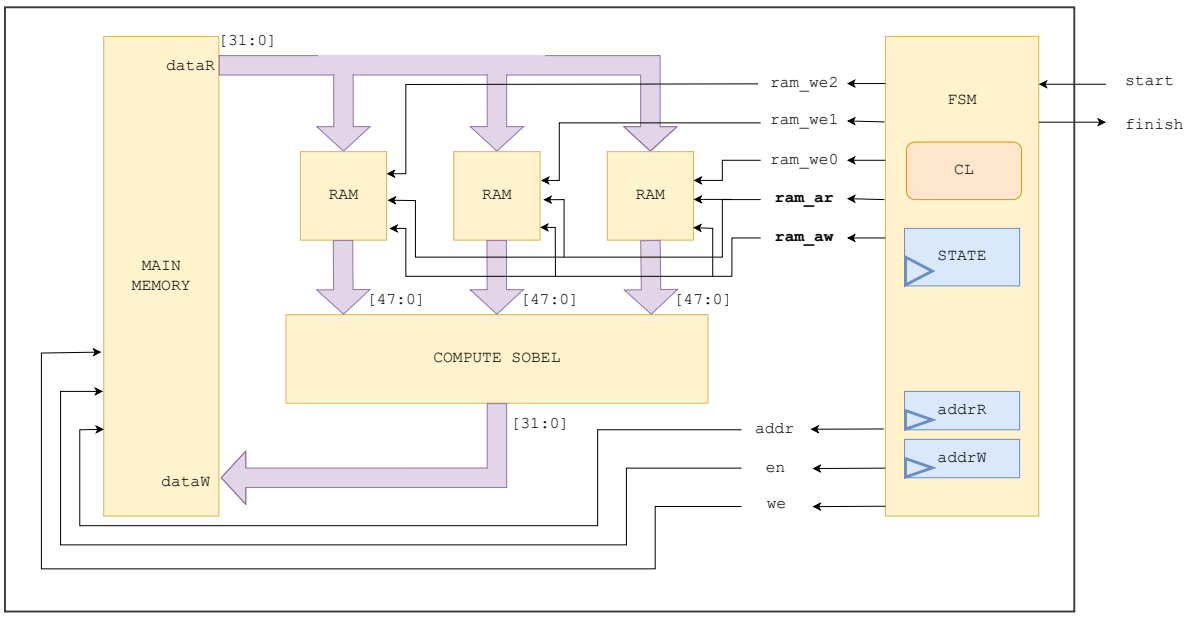


Figure 20: Block diagram with datapath of the optimised design.

4.6 Specifying with VHDL

The two-process method was also used for this design. The biggest change when compared to the previous design is that a new entity was created: the **distributed RAM** entity. 3 instances of it are in use. To understand the best RAM configuration, and how to actually use the RAM resources on the FPGA, some options were tested. This will be discussed further in section subsection 6.1.

In addition to the VHDL features explored on subsection 3.4, some more things were used in this new optimised design:

1. **impure functions**
   Everything written above on subsection 3.4 about **pure functions** applies to **impure functions** as well. The only difference is that these allow for signals to be written, which is not allowed in pure functions. This means that an *impure* function might return a different result if it is executed multiple times, while a *pure* function will **always** return the same result. The impure function in this design computes the convolution to produce the output pixel. The *impure* part comes from the fact that it sets a *read address* on the RAMs, so that the cached pixels can be read. The logic could be changed to make this a pure function, but for simplification purposes, this strategy makes more sense, since there is no disadvantage on using them, if one is careful when programming.

2. **procedures**
   Procedures are extremely similar to impure functions: they can be looked at as being blocks of code that can be placed in each situation where the procedure is called. One

big difference between these two, however, is that procedures don't return any value. In this project, a single procedure is used to selecting the correct RAM and enable writing on it.

4.7 Simulating design

This design was simulated with the same files as the other designs. Figure 21 shows an initial part of the timing diagram produced by Vivado. It shows the first reading phase, where data is read from main memory, and stored in the RAM. To handle the top boundary case, the first line is cached in two of the RAMs, hence the active signal on both `ram_we0` and `ram_we1`. The simulation takes 4131680ns, which is also close to the estimated value.

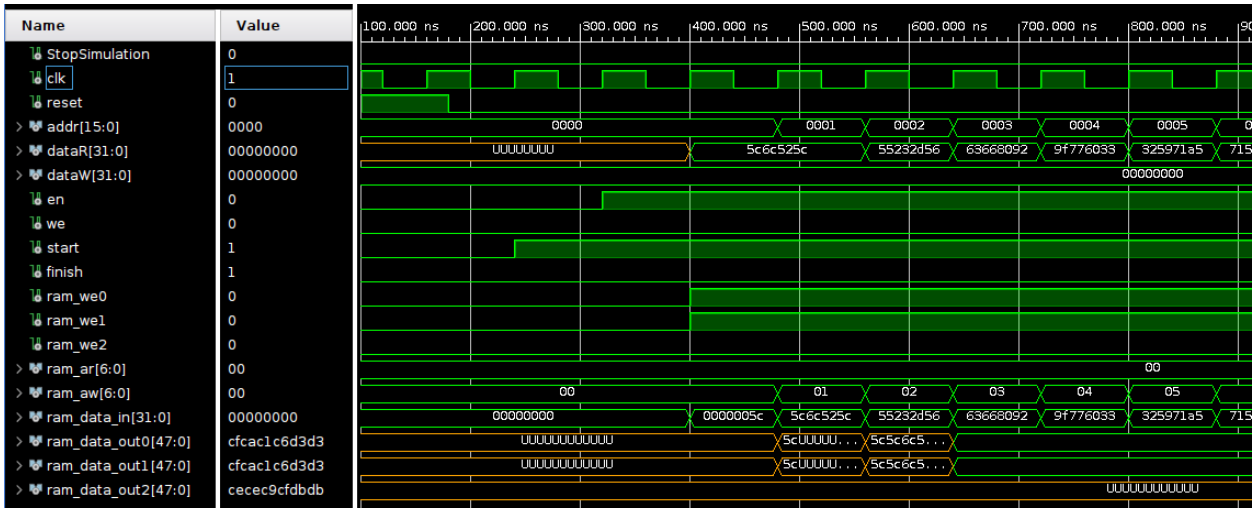


Figure 21: Portion of the timing diagram obtained from simulating the optimised design.

Figure 22 shows the original image and the processed one. These results should be very similar to those on Figure 16, but upon closer inspection it is possible to verify that they differ on the output image's edge pixels.

5 Task 3: Synthesizing and testing design

Similarly as described in subsection 2.7, the design was tested on the NEXYS 4 DDR Artix™-7 FPGA board, and the provided Python script was used to upload and download the images from and to the board. The interface for this program, along with the original image, and the output image can be seen on Figure 23. The execution was shown to a TA during one of the classes.

Due to an error in the files provided, some changes were made made both to the accelerator and the RAM description file. These include changing the order in which pixels are written in main memory, and the order in which they are saved in the RAMs. The top entity provided also needed to be changed to accommodate the new RAMs. These changes in the code can be seen on Appendix G.

Figure 22: On the left, there's the original image which served as input for the accelerator. On the right, the image produced by the optimised edge detection project.

6 Analysing VHDL synthesis: RAM and rem

There are some topics that are worth to cover in more detail in this report, such as:

- the process to understand the best RAM configuration, and how Vivado infers the code to be a RAM
- the `rem` and `mod` operator: remainder/modulus operation

6.1 RAM

Designing a RAM was a process that took a while to understand. It can be described as being an iterative process: in the beginning it wasn't obvious what requisites the code needed to have so that Vivado would infer the entity as being a RAM and use the FPGA's memory resources. The first distributed RAM example was taken from the "Vivado Design Suite User Guide: Synthesis" book [2].

In the following subsections, a few of the iteration phases are described. For this analysis, a new project was created, where the only entity existent is the one being discussed. The goal in this process was to get a RAM that was useful and that actually used the RAM resources on the board. The code used for all these analysis is on Appendix H.

6.1.1 Current/final version

With the final version of the RAM, which was already discussed in subsection 4.2, after synthesizing and obtaining the utilization report, it is clear that Vivado interprets the design as using the FPGA memory LUTs. Figure 24 shows part of the *Slice Logic* table, that indicates the amount of LUTs in use, and their type. It is clear that the memory resources

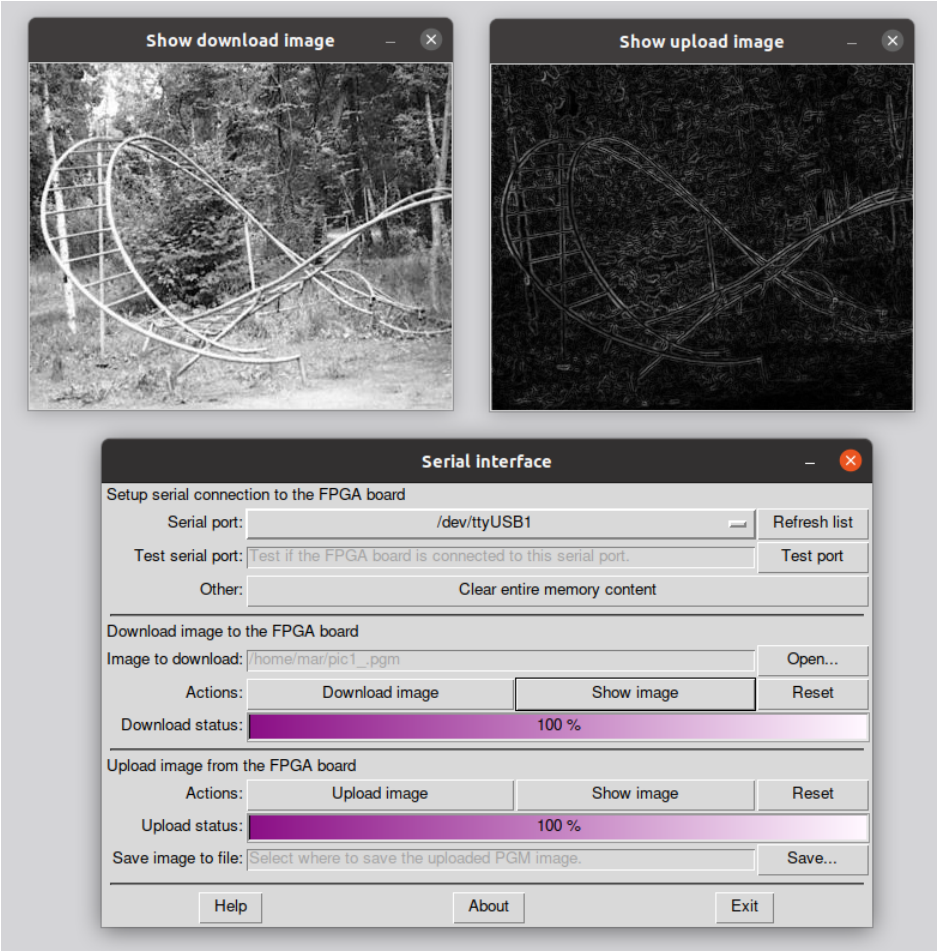


Figure 23: On the top left corner, there's the original image which served as input. On the top right corner, there's the output image as processed by the FPGA board. On the bottom, there's the interface of the program used to upload and download the images to and from the board.

are being used in this design. Note that this is valid even though the input and the output buses have different bit capacity.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	191	0	0	63400	0.30
LUT as Logic	55	0	0	63400	0.09
LUT as Memory	136	0	0	19000	0.72
LUT as Distributed RAM	136	0			
LUT as Shift Register	0	0			
Slice Registers	0	0	0	126800	0.00

Figure 24: Part of *Slice Logic* table, found on Vivado's Synthesis Utilization Report – final version of distributed RAM design

6.1.2 First version

During the brainstorming phase of how the design could be changed to incorporate a caching entity, an idea was to use only one RAM that would store 3 image lines at the same time. During a write operation, there would be a address to select the pixel to be written, as well as an address to select which of the lines should be written. During a read operation, the data from the three lines would be made available synchronously.

With this design, it would be possible to access the exact pixel by an address. In a read operation, 6 pixels would be provided, but it would be possible to chose exactly the first pixel of the set. The writing operation would work in a similar fashion, but with 5 pixels instead of 6.

As shown on the table in Figure 25, upon synthesis, Vivado does not recognise this design as being a RAM, so it doesn't use the RAM resources. This information is shown on the `LUT as Memory` row, which has the `Used` value as 0.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	15518	0	0	63400	24.48
LUT as Logic	15518	0	0	63400	24.48
LUT as Memory	0	0	0	19000	0.00
Slice Registers	8640	0	0	126800	6.81
Register as Flip Flop	8640	0	0	126800	6.81
Register as Latch	0	0	0	126800	0.00
F7 Muxes	3888	0	0	31700	12.26
F8 Muxes	1792	0	0	15850	11.31

Figure 25: *Slice Logic* table, found on Vivado's Synthesis Utilization Report – first version of distributed RAM design

6.1.3 Second version

One hypotheses of why the previous design wasn't inferred correctly is that the entity has three output ports, instead of just one. In this second version, the focus was on removing two of these output buses. To accommodate this change, the address that was used to select which line was to be written was also removed. As visible on Figure 26, this did not produce the desired results, since the Memory resources were still not in use.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	5448	0	0	63400	8.59
LUT as Logic	5448	0	0	63400	8.59
LUT as Memory	0	0	0	19000	0.00
Slice Registers	2880	0	0	126800	2.27
Register as Flip Flop	2880	0	0	126800	2.27
Register as Latch	0	0	0	126800	0.00
F7 Muxes	1112	0	0	31700	3.51
F8 Muxes	504	0	0	15850	3.18

Figure 26: *Slice Logic* table, found on Vivado's Synthesis Utilization Report – second version of distributed RAM design

6.1.4 Third version

Another possibility of why it isn't working is that the input and output buses have different sizes. Therefore, a new design was developed in which the input and output buffers had the same size. Note that in this design, it was still possible to select the first pixel of the word. As shown on the table in Figure 27, this change also didn't produce the desired outcome.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	6242	0	0	63400	9.85
LUT as Logic	6242	0	0	63400	9.85
LUT as Memory	0	0	0	19000	0.00
Slice Registers	2880	0	0	126800	2.27
Register as Flip Flop	2880	0	0	126800	2.27
Register as Latch	0	0	0	126800	0.00
F7 Muxes	1104	0	0	31700	3.48
F8 Muxes	504	0	0	15850	3.18

Figure 27: *Slice Logic* table, found on Vivado's Synthesis Utilization Report – third version of distributed RAM design

6.1.5 Fourth version

Maybe the problem is that multiple array locations were accessed during a writing operation at the same time. To test this hypothesis, the word size was increased to 4 – pixels can only be selected in multiple of 4, similarly to how the main memory works. The code in this phase was very similar to that on the Vivado Synthesis manual [2], so it was expected to work as desired. Figure 28 shows that it is the case, since the value on `LUT as Memory` is no longer 0.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	131	0	0	63400	0.21
LUT as Logic	35	0	0	63400	0.06
LUT as Memory	96	0	0	19000	0.51
LUT as Distributed RAM	96	0			
LUT as Shift Register	0	0			
Slice Registers	0	0	0	126800	0.00
Register as Flip Flop	0	0	0	126800	0.00
Register as Latch	0	0	0	126800	0.00
F7 Muxes	0	0	0	31700	0.00
F8 Muxes	0	0	0	15850	0.00

Figure 28: *Slice Logic* table, found on Vivado's Synthesis Utilization Report – fourth version of distributed RAM design

6.1.6 Fifth version

With the goal of designing a RAM that could adapt to the project's necessities, it could be interesting to go back and make sure that the input and output buses can't have different sizes. Changing the output signal to have 48 bits, while the input signal has only 32, produces a design with the synthesis report on Figure 29. This table shows that this design was correctly inferred as being a RAM, and uses the board's memory resources – proving therefore that the input and output buffer ultimately don't need to have the same size.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	251	0	0	63400	0.40
LUT as Logic	59	0	0	63400	0.09
LUT as Memory	192	0	0	19000	1.01
LUT as Distributed RAM	192	0			
LUT as Shift Register	0	0			
Slice Registers	0	0	0	126800	0.00
Register as Flip Flop	0	0	0	126800	0.00
Register as Latch	0	0	0	126800	0.00
F7 Muxes	0	0	0	31700	0.00
F8 Muxes	0	0	0	15850	0.00

Figure 29: *Slice Logic* table, found on Vivado's Synthesis Utilization Report – fifth version of distributed RAM design

Having understood the constraints that the code must follow, so that the entity is correctly synthesized, it was just a matter of selecting the best configuration for the RAM. From this iterative process, it was possible to understand that:

- the input and output buffer can have different sizes
- the write operation can only access one word in the memory internal array

With this in mind, an appropriate RAM design was reached, as already discussed.

6.2 `rem` operator

In the edge detector designs, the `rem` operator was used. One of the uses involve the selection of the correct RAM to perform caching of data. Another involves understanding if the pixel that is being read/written is the last of a line.

The remainder, or modulus, operation is a computation based on multiplication and subtraction – which means it will never have a good performance on hardware. Even though Vivado can synthesize the code, and even implement and program the FPGA board with it, it is obvious that this element shouldn't be used: during the implementation phase, there's a **Critical Warning** that states that the design failed to meet the timing requirements. This warning is most likely due to the use of this operator. Note that the board behaves as expected once it is programmed, despite the warning.

Similarly to what was done previously, in the analysis of the RAM designs, a new project was created just to test this operator. The code can be found on Appendix I. A simple circuit that mimicks the usage that the operator has on the edge detector design was created. The syntheses schema generated by Vivado is shown on Figure 30. Unfortunately, the tool just synthesizes it to it's own entity, without revealing the lower-level characteristics. A *google* search shows that it is not easy to understand how this operand is synthesized – since it depends on the boards, and on other different factors –, but the consensus seems to be that it is an extremely bad idea to use it. Searching on Vivado manuals, as well as the board's specification didn't return much information.

Therefore, it would have been a good idea to find an alternative for this operator. Two alternatives come to mind:

- implement a **lookup table** using LUTs: a new entity where a specific input signal would provide an output signal with the result of the modulus computation – all *hard coded*
- add registers that are used as counters in the accelerator entity, so that the modulus operation could be replaced by a simple subtraction operation – given that the counters are updated properly

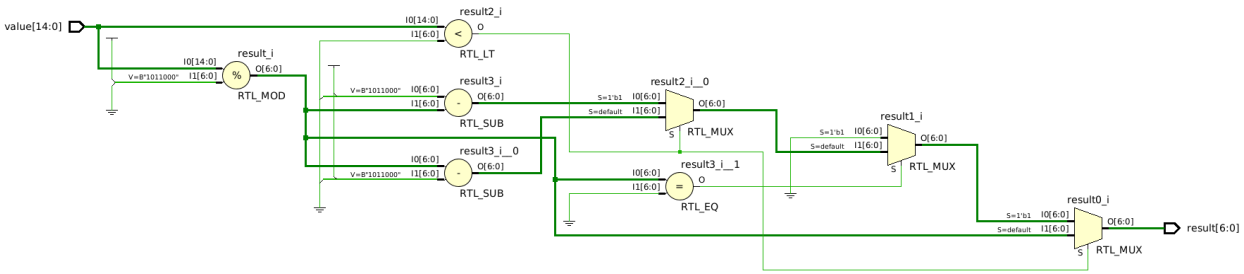


Figure 30: RTL-level schema of `rem` analysis project.

List of Figures

List of Tables

Listings

References

[1] DTU, *02203 Design of Digital Systems (fall 2023) - Edge detection design project*, 2023.

[2] I. Xilinx, *Vivado Design Suite User Guide: Synthesis*. 2020.

Appendix A ASMD-chart of the inverter project

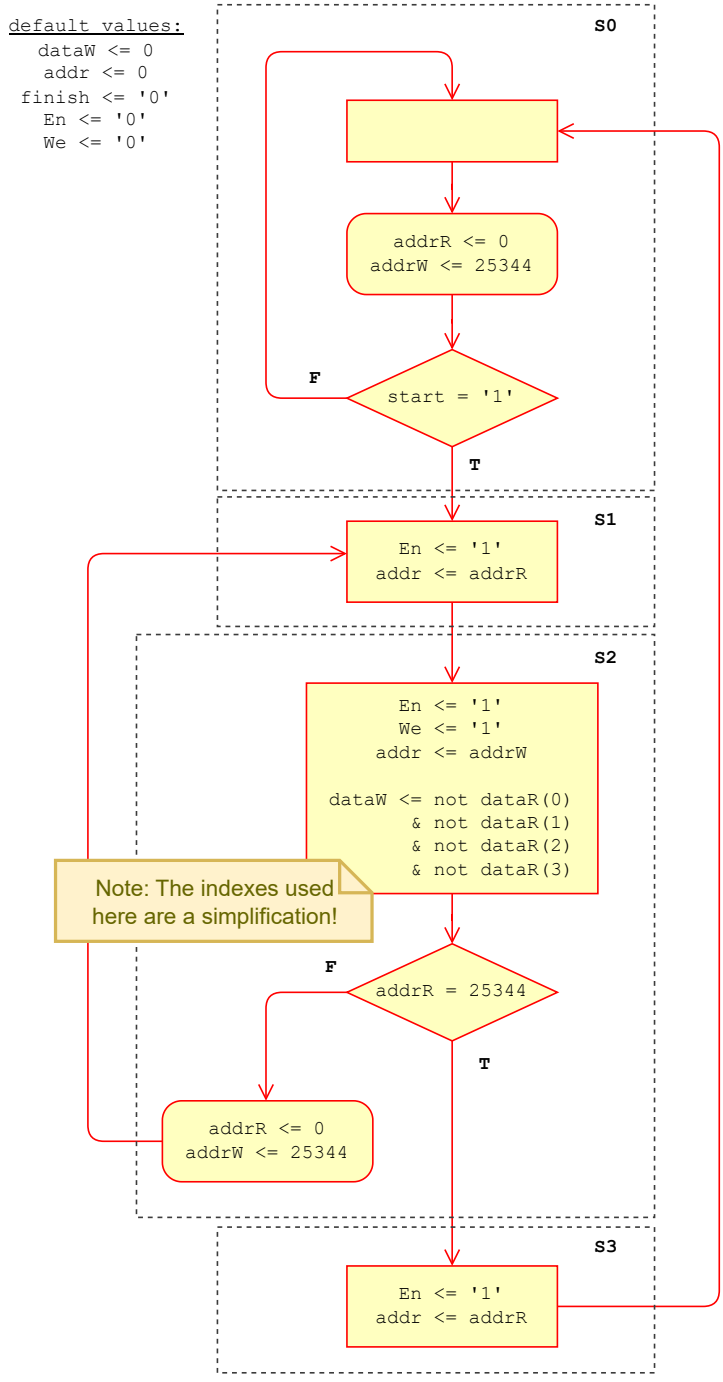


Figure 31: ASMD-chart for the inverter project.

Appendix B VHDL description of the inverter project

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4  USE work.types.ALL;
5  ENTITY acc IS
6    PORT (
7      clk : IN bit_t; -- The clock.
8      reset : IN bit_t; -- The reset signal. Active high.
9      addr : OUT halfword_t; -- Address bus for data.
10     dataR : IN word_t; -- The data bus.
11     dataW : OUT word_t; -- The data bus.
12     en : OUT bit_t; -- Request signal for data.
13     we : OUT bit_t; -- Read/Write signal for data.
14     start : IN bit_t;
15     finish : OUT bit_t
16   );
17 END acc;
18 ARCHITECTURE rtl OF acc IS
19   TYPE state_type IS (S0, S1, S2, S3);
20   SIGNAL state, next_state : state_type;
21   SIGNAL addrR, next_addrR : halfword_t;
22   SIGNAL addrW, next_addrW : halfword_t;
23 BEGIN
24   cl : PROCESS (clk, reset, state, addrR, addrW, start, dataR)
25   BEGIN
26     next_state <= state;
27     next_addrR <= addrR;
28     next_addrW <= addrW;
29     dataW <= (OTHERS => '0');
30     addr <= (OTHERS => '0');
31     finish <= '0';
32     En <= '0';
33     We <= '0';
34     CASE (state) IS
35       WHEN S0 =>
36         -- En <= '0';
37         next_addrR <= halfword_zero;
38         next_addrW <= std_logic_vector(to_unsigned(25344, 16));
39         IF start = '1' THEN
40           next_state <= S1;
41         END IF;
42       WHEN S1 =>
43         En <= '1';
44         next_state <= S2;
45         addr <= addrR;
46       WHEN S2 =>
47         next_addrR <= std_logic_vector(to_unsigned(to_integer(
   unsigned(addrR)) + 1, 16));
48         next_addrW <= std_logic_vector(unsigned(addrW) + 1);
49         En <= '1';
50         We <= '1';
51         addr <= addrW;
52         dataW <= NOT dataR(31 DOWNTO 24)
53            & NOT dataR(23 DOWNTO 16)
54            & NOT dataR(15 DOWNTO 8)
55            & NOT dataR(7 DOWNTO 0);
56
57           IF addrR = std_logic_vector(to_unsigned(25344, 16)) THEN
```

```
58              next_state <= S3;
59            ELSE
60              next_state <= S1;
61            END IF;
62        WHEN S3 =>
63          finish <= '1';
64          next_state <= S0;
65
66        WHEN OTHERS =>
67          next_state <= S0;
68      END CASE;
69
70    END PROCESS cl;
71
72    seq : PROCESS (clk, reset)
73    BEGIN
74      IF reset = '1' THEN
75        state <= S0;
76        addrR <= (OTHERS => '0');
77        addrW <= byte_zero & byte_one;
78      ELSIF rising_edge(clk) THEN
79        state <= next_state;
80        addrR <= next_addrR;
81        addrW <= next_addrW;
82      END IF;
83    END PROCESS seq;
84  END rtl;
```

Listing 1: VHDL code describing the `acc` entity for the inverter project.

Appendix C ASMD-chart of the first edge detector version

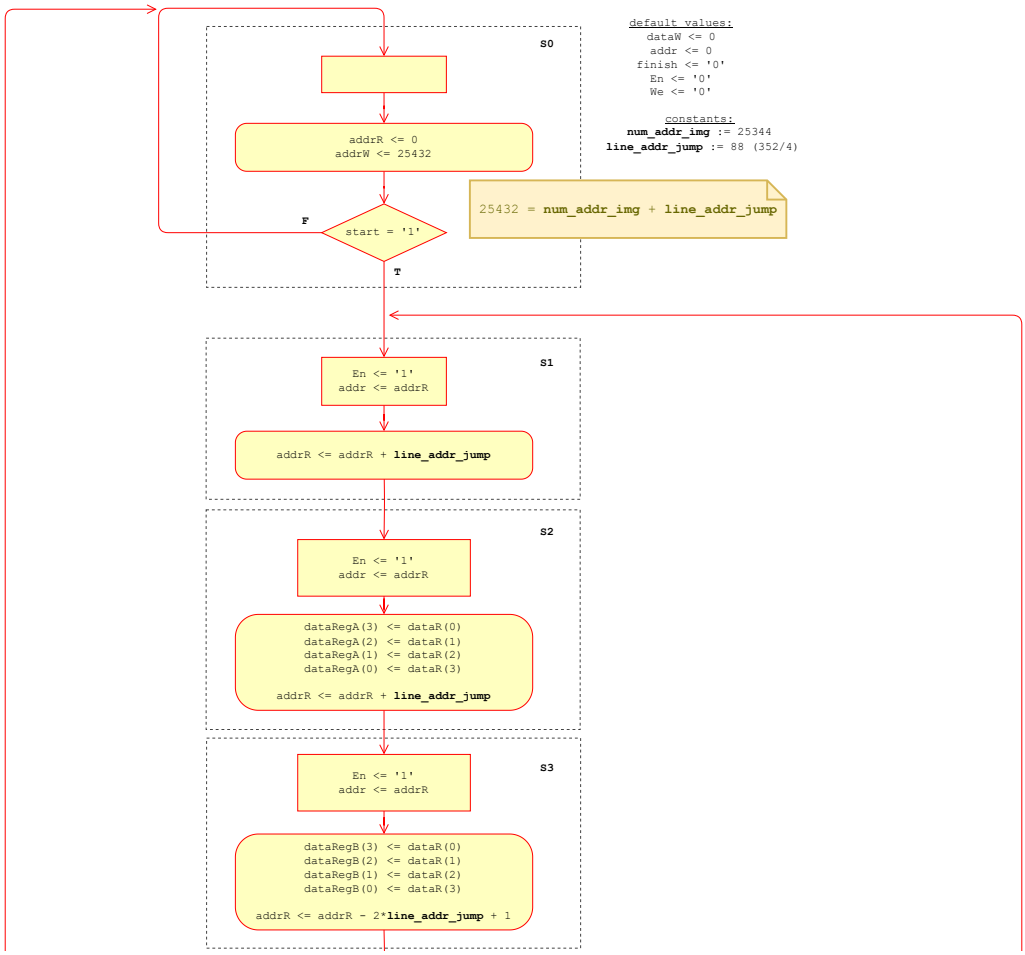


Figure 32: First part of ASMD-chart for the first version of the edge detector.

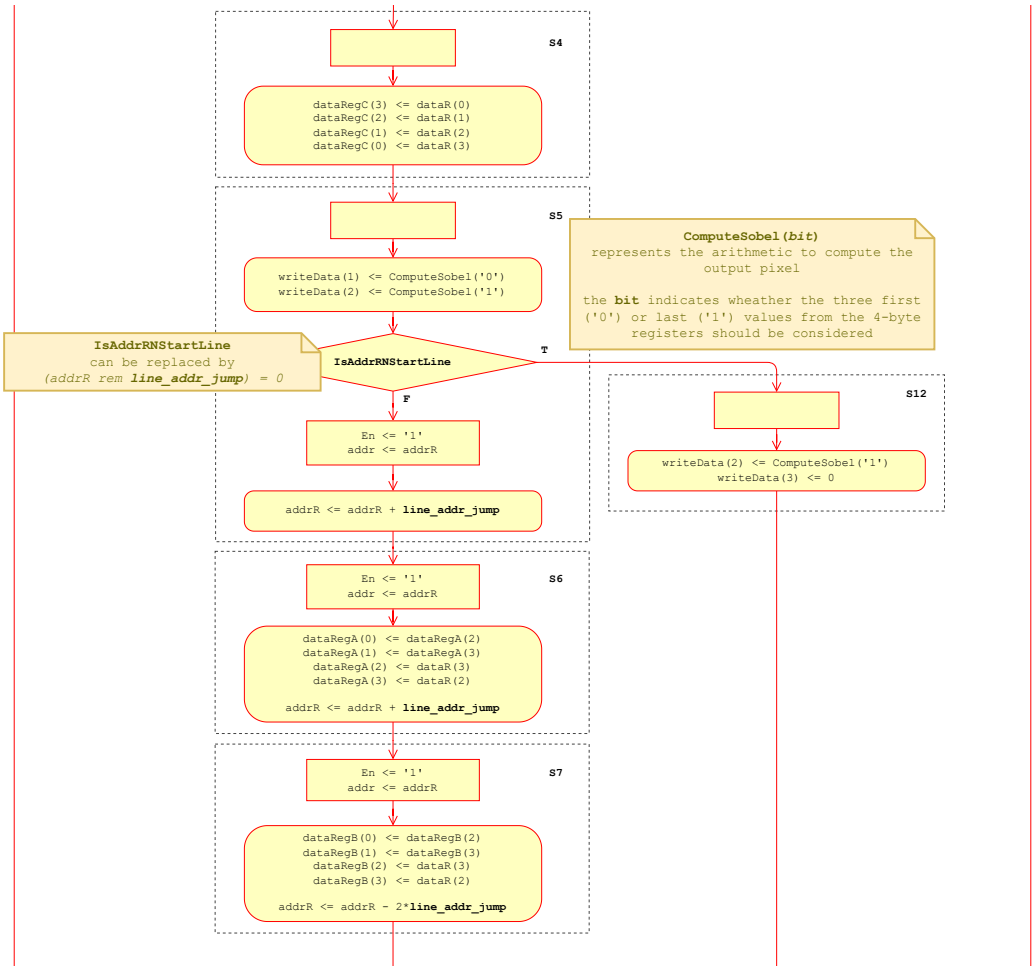


Figure 33: Second part of ASMD-chart for the first version of the edge detector.

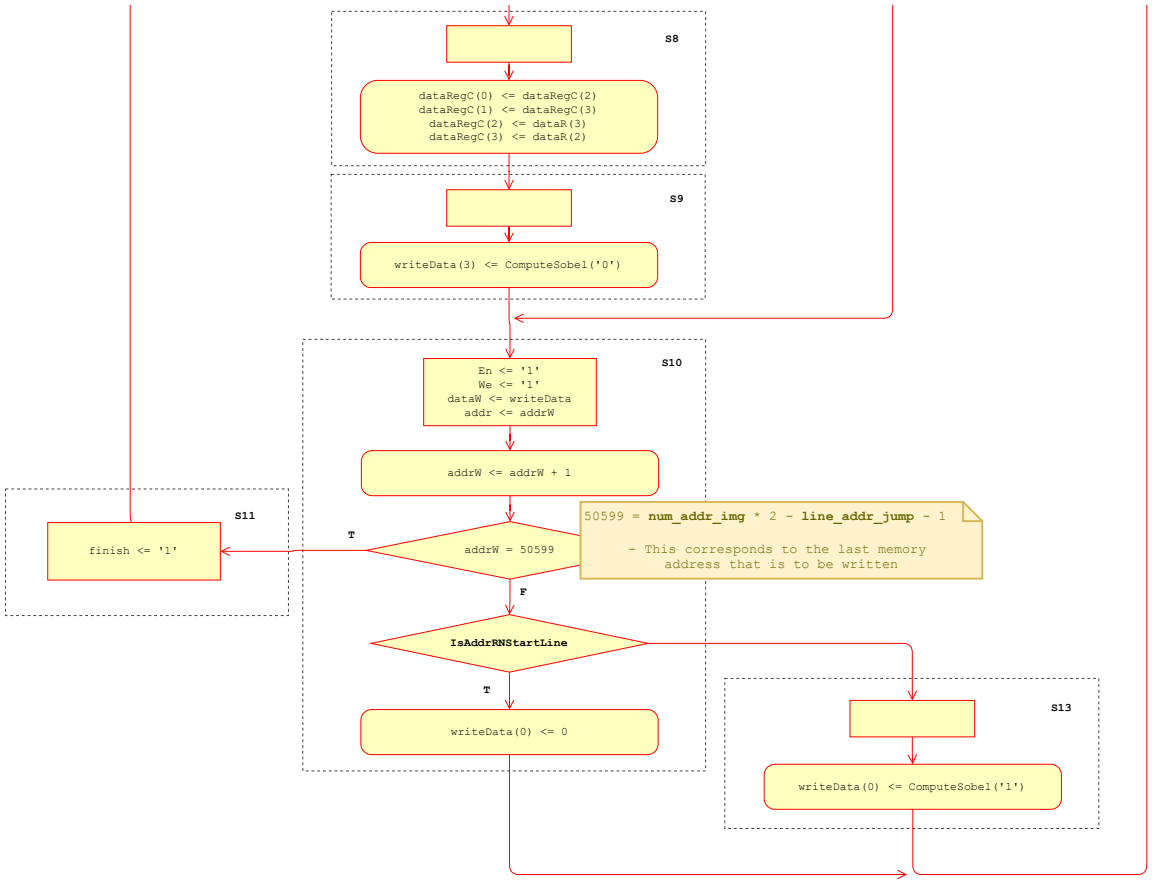


Figure 34: Third part of ASMD-chart for the first version of the edge detector.

Appendix D VHDL description of the first edge detector version

```
1
2  LIBRARY IEEE;
3  USE IEEE.std_logic_1164.ALL;
4  USE IEEE.numeric_std.ALL;
5  USE work.types.ALL;
6  ENTITY acc IS
7    PORT (
8      clk : IN bit_t; -- The clock.
9      reset : IN bit_t; -- The reset signal. Active high.
10     addr : OUT halfword_t; -- Address bus for data.
11     dataR : IN word_t; -- The data bus.
12     dataW : OUT word_t; -- The data bus.
13     en : OUT bit_t; -- Request signal for data.
14     we : OUT bit_t; -- Read/Write signal for data.
15     start : IN bit_t;
16     finish : OUT bit_t
17   );
18 END acc;
19 ARCHITECTURE rtl OF acc IS
20   TYPE state_type IS (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10,
      S11, S12, S13);
21   TYPE line_t IS ARRAY (3 DOWNTO 0) OF byte_t;
22   SIGNAL state, next_state : state_type;
23   SIGNAL addrR, next_addrR : halfword_t;
24   SIGNAL addrW, next_addrW : halfword_t;
25   SIGNAL dataRegA, next_dataRegA : line_t;
26   SIGNAL dataRegB, next_dataRegB : line_t;
27   SIGNAL dataRegC, next_dataRegC : line_t;
28   SIGNAL writeData, next_writeData : line_t;
29 BEGIN
30   cl : PROCESS (clk, reset, state, addrR, addrW, dataRegA, dataRegB,
      dataRegC, writeData, start, dataR)
31     CONSTANT num_addr_img : INTEGER := 25344;
32     CONSTANT line_addr_jump : INTEGER := 352/4;
33
34     FUNCTION IsAddrRNStartLine RETURN BOOLEAN IS
35     BEGIN
36       RETURN (to_integer(unsigned(addrR)) REM line_addr_jump) = 0;
37     END FUNCTION;
38
39     FUNCTION ComputeSobel (
40       flag : BIT) RETURN byte_t IS
41       VARIABLE gx, gy : INTEGER;
42       VARIABLE total : std_logic_vector(10 DOWNTO 0);
43
44       BEGIN
45         IF flag = '0' THEN
46                     gx := -1 * to_integer(unsigned(dataRegA(0))) -2 *
      to_integer(unsigned(dataRegB(0))) -1 * to_integer(unsigned(
      dataRegC(0)))
47                           +1 * to_integer(unsigned(dataRegA(2))) +2 *
      to_integer(unsigned(dataRegB(2))) +1 * to_integer(unsigned(
      dataRegC(2)));
48                     gy := +1 * to_integer(unsigned(dataRegA(0))) +2 *
      to_integer(unsigned(dataRegA(1))) +1 * to_integer(unsigned(
      dataRegA(2)))
```

```
49                          -1 * to_integer(unsigned(dataRegC(0))) -2 *
    to_integer(unsigned(dataRegC(1))) -1 * to_integer(unsigned(
    dataRegC(2)));
50        ELSE
51                      gx := -1 * to_integer(unsigned(dataRegA(1))) -2 *
    to_integer(unsigned(dataRegB(1))) -1 * to_integer(unsigned(
    dataRegC(1)))
52                          +1 * to_integer(unsigned(dataRegA(3))) +2 *
    to_integer(unsigned(dataRegB(3))) +1 * to_integer(unsigned(
    dataRegC(3)));
53                      gy := +1 * to_integer(unsigned(dataRegA(1))) +2 *
    to_integer(unsigned(dataRegA(2))) +1 * to_integer(unsigned(
    dataRegA(3)))
54                          -1 * to_integer(unsigned(dataRegC(1))) -2 *
    to_integer(unsigned(dataRegC(2))) -1 * to_integer(unsigned(
    dataRegC(3)));
55                  END IF;
56        total := std_logic_vector(to_unsigned(ABS(gx) + ABS(gy), 11))
    ; RETURN total(10 DOWNTO 3);
57      END FUNCTION;
58      BEGIN
59        next_state <= state;
60        next_addrR <= addrR;
61        next_addrW <= addrW;
62        next_dataRegA <= dataRegA;
63        next_dataRegB <= dataRegB;
64        next_dataRegC <= dataRegC;
65        next_writeData <= writeData;
66        dataW <= (OTHERS => '0');
67        addr <= (OTHERS => '0');
68        finish <= '0';
69        En <= '0';
70        We <= '0';
71        CASE (state) IS
72          WHEN S0 =>
73            next_addrR <= halfword_zero;
74            next_addrW <= std_logic_vector(to_unsigned(num_addr_img +
     line_addr_jump, 16));
75
76            IF start = '1' THEN
77              next_state <= S1;
78            END IF;
79
80          WHEN S1 =>
81            En <= '1';
82            addr <= addrR; -- next state: read A
83            next_addrR <= std_logic_vector(to_unsigned(to_integer(
    unsigned(addrR)) + line_addr_jump, 16)); -- next address: read B
84            next_state <= S2;
85
86          WHEN S2 =>
87            En <= '1';
88            next_dataRegA(3) <= dataR(7 DOWNTO 0);
89            next_dataRegA(2) <= dataR(15 DOWNTO 8);
90            next_dataRegA(1) <= dataR(23 DOWNTO 16);
91            next_dataRegA(0) <= dataR(31 DOWNTO 24);
92
93            addr <= addrR; -- next state: read B
94            next_addrR <= std_logic_vector(to_unsigned(to_integer(
    unsigned(addrR)) + line_addr_jump, 16)); -- next address: read C
95            next_state <= S3;
```

```
96
97          WHEN S3 =>
98            En <= '1';
99            next_dataRegB(3) <= dataR(7 DOWNTO 0);
100           next_dataRegB(2) <= dataR(15 DOWNTO 8);
101           next_dataRegB(1) <= dataR(23 DOWNTO 16);
102           next_dataRegB(0) <= dataR(31 DOWNTO 24);
103
104           addr <= addrR; -- next state: read C
105           next_addrR <= std_logic_vector(to_unsigned(to_integer(
    unsigned(addrR)) - (2 * line_addr_jump) + 1, 16)); -- next address
    : read following A
106           next_state <= S4;
107
108         WHEN S4 => -- reads C
109           next_dataRegC(3) <= dataR(7 DOWNTO 0);
110           next_dataRegC(2) <= dataR(15 DOWNTO 8);
111           next_dataRegC(1) <= dataR(23 DOWNTO 16);
112           next_dataRegC(0) <= dataR(31 DOWNTO 24);
113
114           next_state <= S5;
115
116         WHEN S5 =>
117           next_writeData(1) <= ComputeSobel('0');
118           next_writeData(2) <= ComputeSobel('1');
119           IF IsAddrRNStartLine THEN
120             -- if next addrR is a new line, then we want to finish
121             -- this line without reading anymore for now;
122             -- the address will continue to be the first pixels in
    the new line
123             next_state <= S12;
124           ELSE
125             -- this is the "normal" situation
126             -- we want to read the new pixels to compute the last
    two output pixels
127             En <= '1';
128             addr <= addrR; -- next state: read A
129             next_addrR <= std_logic_vector(to_unsigned(to_integer(
    unsigned(addrR)) + line_addr_jump, 16)); -- next address: read B
130             next_state <= S6;
131           END IF;
132
133         WHEN S6 =>
134           En <= '1';
135           -- read A and store half
136           next_dataRegA(0) <= dataRegA(2);
137           next_dataRegA(1) <= dataRegA(3);
138           next_dataRegA(2) <= dataR(31 DOWNTO 24);
139           next_dataRegA(3) <= dataR(23 DOWNTO 16);
140           addr <= addrR; -- next state: read B
141           next_addrR <= std_logic_vector(to_unsigned(to_integer(
    unsigned(addrR)) + line_addr_jump, 16)); -- next address: read C
142           next_state <= S7;
143
144         WHEN S7 =>
145           En <= '1';
146           -- read B and store half
147           next_dataRegB(0) <= dataRegB(2);
148           next_dataRegB(1) <= dataRegB(3);
149           next_dataRegB(2) <= dataR(31 DOWNTO 24);
150           next_dataRegB(3) <= dataR(23 DOWNTO 16);
```

```
151             addr <= addrR; -- next state: read C
152             -- next address: read the same A as before;
153             -- we will want to re-read A, B and C once more!
154             next_addrR <= std_logic_vector(to_unsigned(to_integer(
    unsigned(addrR)) - 2 * line_addr_jump, 16));
155             next_state <= S8;
156
157           WHEN S8 =>
158             -- read C and store half
159             next_dataRegC(0) <= dataRegC(2);
160             next_dataRegC(1) <= dataRegC(3);
161             next_dataRegC(2) <= dataR(31 DOWNTO 24);
162             next_dataRegC(3) <= dataR(23 DOWNTO 16);
163
164             next_state <= S9;
165
166           WHEN S9 =>
167             -- write buffer will now be complete
168             next_writeData(3) <= ComputeSobel('0');
169             next_state <= S10;
170
171           WHEN S10 =>
172             -- storing output
173             En <= '1';
174             We <= '1';
175             dataW <= writeData(0) & writeData(1) & writeData(2) &
    writeData(3);
176             addr <= addrW;
177             -- increment write address by 1
178             next_addrW <= std_logic_vector(to_unsigned(to_integer(
    unsigned(addrW)) + 1, 16));
179
180             IF addrW = std_logic_vector(to_unsigned(num_addr_img * 2
    - line_addr_jump - 1, 16)) THEN
181               -- all pixels are finished: go to final state
182               next_state <= S11;
183
184             ELSIF IsAddrRNStartLine THEN
185               -- start reading a new line
186               -- borders are ignored -- output pixel is 0
187               next_writeData(0) <= byte_zero;
188               next_state <= S1;
189             ELSE
190               -- "normal" situation - continue reading other pixels
    in the same line
191               next_state <= S13;
192             END IF;
193
194           WHEN S12 =>
195             -- finish this line
196             next_writeData(2) <= ComputeSobel('1');
197             next_writeData(3) <= byte_zero; -- borders are set to 0
198             -- output buffer is complete: save it on S10
199             next_state <= S10;
200
201           WHEN S13 =>
202             -- compute first pixel of output buffer
203             next_writeData(0) <= ComputeSobel('1');
204             next_state <= S1;
205
206           WHEN S11 =>
```

```
207             finish <= '1';
208             next_state <= S0;
209
210           WHEN OTHERS =>
211             next_state <= S0;
212
213         END CASE;
214
215       END PROCESS cl;
216
217       seq : PROCESS (clk, reset)
218       BEGIN
219         IF reset = '1' THEN
220           dataRegA(0) <= (OTHERS => '0');
221           dataRegA(1) <= (OTHERS => '0');
222           dataRegA(2) <= (OTHERS => '0');
223           dataRegA(3) <= (OTHERS => '0');
224
225           dataRegB(0) <= (OTHERS => '0');
226           dataRegB(1) <= (OTHERS => '0');
227           dataRegB(2) <= (OTHERS => '0');
228           dataRegB(3) <= (OTHERS => '0');
229
230           dataRegC(0) <= (OTHERS => '0');
231           dataRegC(1) <= (OTHERS => '0');
232           dataRegC(2) <= (OTHERS => '0');
233           dataRegC(3) <= (OTHERS => '0');
234
235           writeData(0) <= (OTHERS => '0');
236           writeData(1) <= (OTHERS => '0');
237           writeData(2) <= (OTHERS => '0');
238           writeData(3) <= (OTHERS => '0');
239
240           state <= S0;
241           addrR <= (OTHERS => '0');
242           addrW <= byte_zero & byte_one;
243         ELSIF rising_edge(clk) THEN
244           dataRegA <= next_dataRegA;
245           dataRegB <= next_dataRegB;
246           dataRegC <= next_dataRegC;
247           writeData <= next_writeData;
248           state <= next_state;
249           addrR <= next_addrR;
250           addrW <= next_addrW;
251         END IF;
252       END PROCESS seq;
253 END rtl;
```

Listing 2: VHDL description of the first edge detector version.

Appendix E ASMD-chart of the optimised edge detector

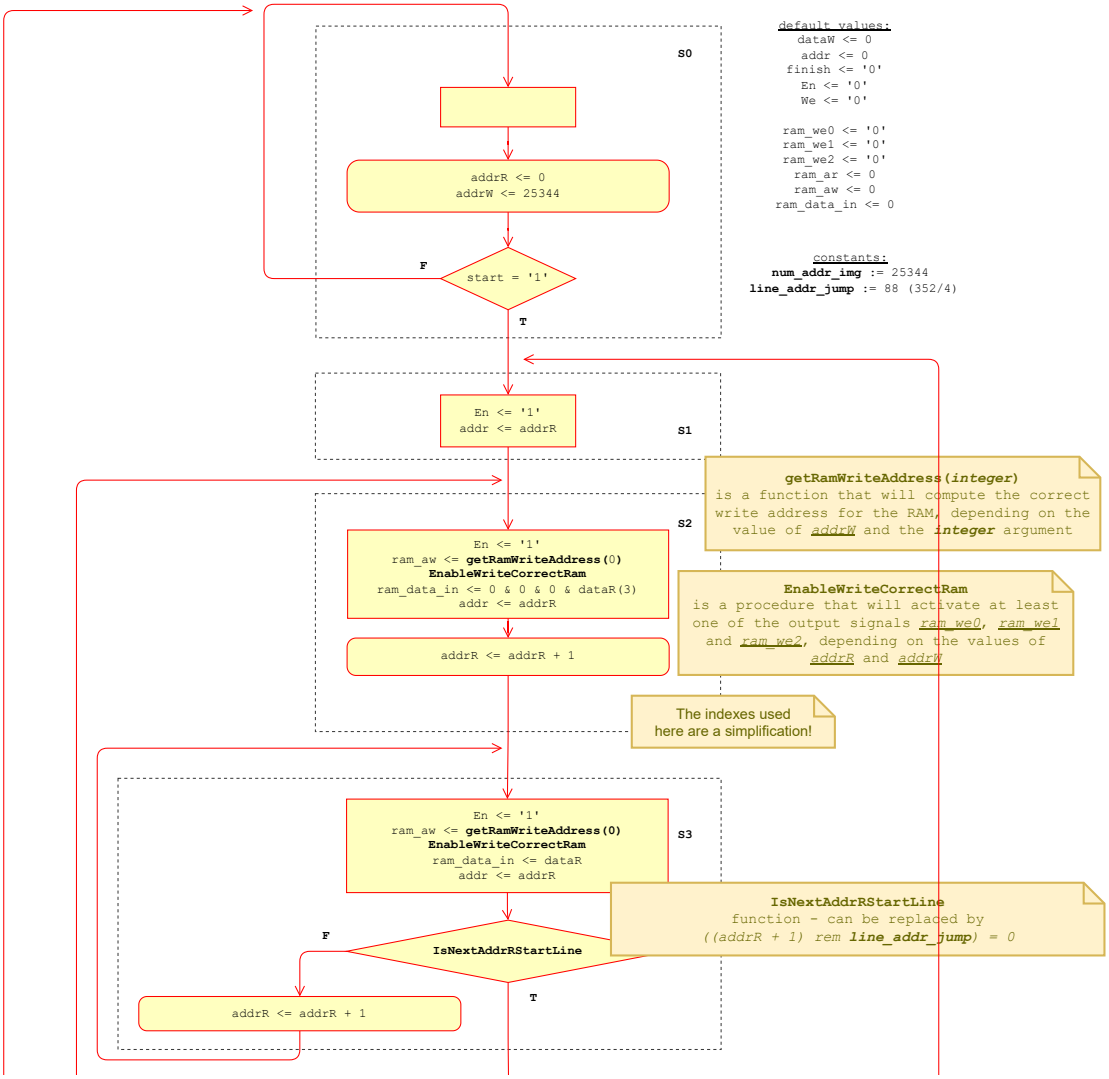


Figure 35: First part of ASMD-chart for the optimised version of the edge detector.

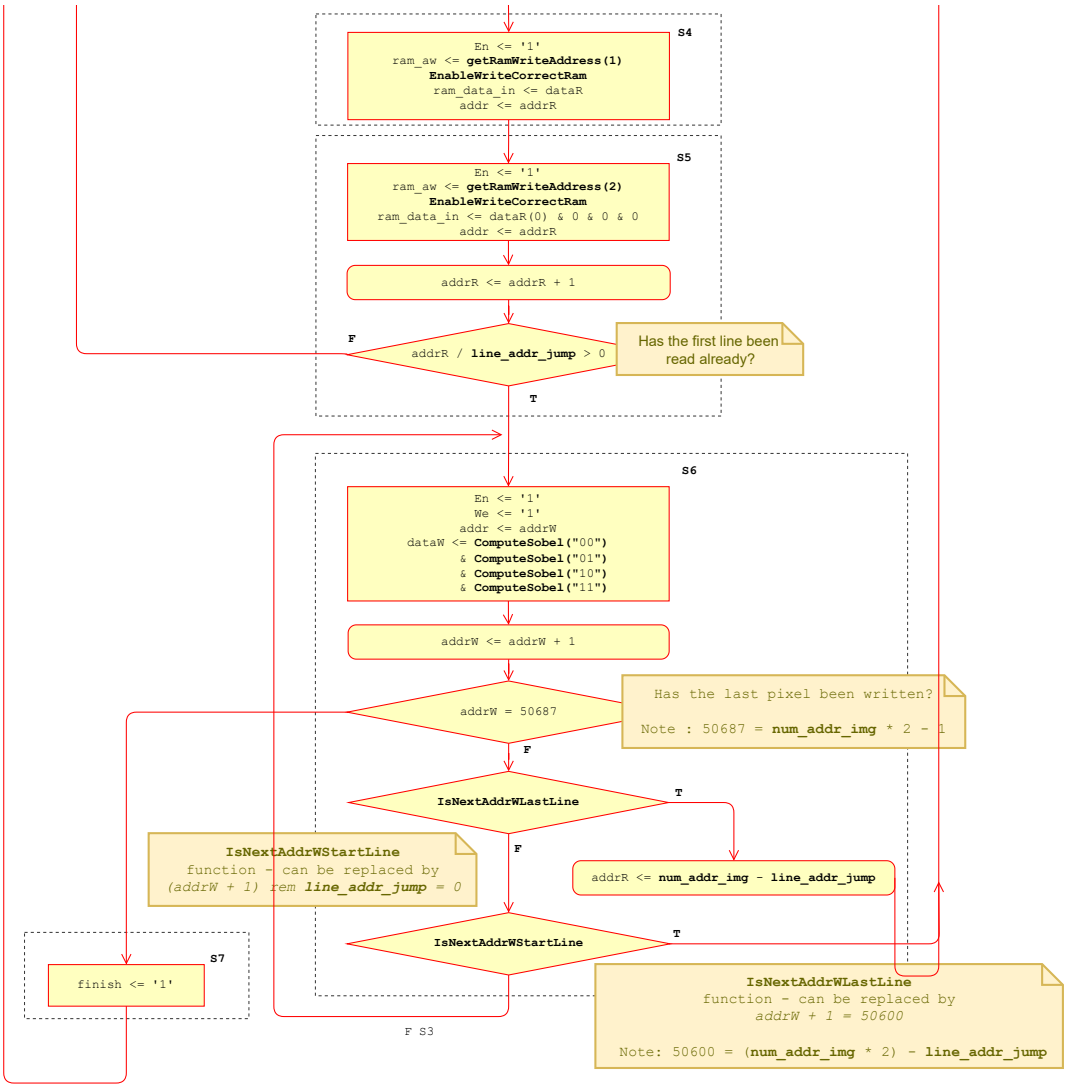


Figure 36: First part of ASMD-chart for the optimised version of the edge detector.

Appendix F VHDL description of the optimised edge detector

```
1
2  LIBRARY IEEE;
3  USE IEEE.std_logic_1164.ALL;
4  USE IEEE.numeric_std.ALL;
5  USE work.types.ALL;
6
7  ENTITY acc IS
8    PORT (
9      clk : IN bit_t; -- The clock.
10     reset : IN bit_t; -- The reset signal. Active high.
11     addr : OUT halfword_t; -- Address bus for data.
12     dataR : IN word_t; -- The data bus.
13     dataW : OUT word_t; -- The data bus.
14     en : OUT bit_t; -- Request signal for data.
15     we : OUT bit_t; -- Read/Write signal for data.
16     start : IN bit_t;
17     finish : OUT bit_t;
18
19     ram_we0 : OUT bit_t;
20     ram_we1 : OUT bit_t;
21     ram_we2 : OUT bit_t;
22     ram_ar : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
23     ram_aw : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
24     ram_data_in : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
25     ram_data_out0 : IN STD_LOGIC_VECTOR(47 DOWNTO 0);
26     ram_data_out1 : IN STD_LOGIC_VECTOR(47 DOWNTO 0);
27     ram_data_out2 : IN STD_LOGIC_VECTOR(47 DOWNTO 0)
28   );
29 END acc;
30
31 ARCHITECTURE rtl OF acc IS
32   TYPE state_type IS (S0, S1, S2, S3, S4, S5, S6, S7);
33
34   TYPE line3_t IS ARRAY (2 DOWNTO 0) OF byte_t;
35   SUBTYPE rem_addr_type IS STD_LOGIC_VECTOR(18 DOWNTO 0);
36   SUBTYPE rem_addr_type_lin IS STD_LOGIC_VECTOR(1 DOWNTO 0);
37   SUBTYPE rem_addr_type_col IS STD_LOGIC_VECTOR(6 DOWNTO 0);
38
39   SIGNAL state, next_state : state_type;
40
41   SIGNAL addrR, next_addrR : halfword_t;
42   SIGNAL addrW, next_addrW : halfword_t;
43
44 BEGIN
45
46   cl : PROCESS (clk, reset, state, addrR, addrW, start, dataR,
      ram_data_out0, ram_data_out1, ram_data_out2)
47
48     CONSTANT num_addr_img : INTEGER := 25344;
49     CONSTANT line_addr_jump : INTEGER := 352/4;
50
51     FUNCTION IsNextAddrRStartLine RETURN BOOLEAN IS
52     BEGIN
53       RETURN ((to_integer(unsigned(addrR)) + 1) REM line_addr_jump) =
       0;
54     END FUNCTION;
```

```
55
56      FUNCTION IsNextAddrWStartLine RETURN BOOLEAN IS
57      BEGIN
58        RETURN ((TO_INTEGER(unsigned(addrW)) + 1)REM line_addr_jump) =
    0;
59      END FUNCTION;
60
61      FUNCTION IsNextAddrWLastLine RETURN BOOLEAN IS
62      BEGIN
63        RETURN to_integer(unsigned(addrW)) + 1 = (num_addr_img * 2) -
    line_addr_jump;
64      END FUNCTION;
65      FUNCTION GetPixels(offset : STD_LOGIC_VECTOR(1 DOWNTO 0); vector
    : STD_LOGIC_VECTOR(47 DOWNTO 0)) RETURN line3_t IS
66        VARIABLE A : line3_t;
67      BEGIN
68        CASE (offset) IS
69          WHEN "00" =>
70            A(0) := vector(47 DOWNTO 40);
71            A(1) := vector(39 DOWNTO 32);
72            A(2) := vector(31 DOWNTO 24);
73
74          WHEN "01" =>
75            A(0) := vector(39 DOWNTO 32);
76            A(1) := vector(31 DOWNTO 24);
77            A(2) := vector(23 DOWNTO 16);
78
79          WHEN "10" =>
80            A(0) := vector(31 DOWNTO 24);
81            A(1) := vector(23 DOWNTO 16);
82            A(2) := vector(15 DOWNTO 8);
83
84          WHEN "11" =>
85            A(0) := vector(23 DOWNTO 16);
86            A(1) := vector(15 DOWNTO 8);
87            A(2) := vector(7 DOWNTO 0);
88
89          WHEN OTHERS =>
90            A(0) := byte_zero;
91            A(1) := byte_zero;
92            A(2) := byte_zero;
93
94        END CASE;
95
96        RETURN A;
97
98      END FUNCTION;
99
100     IMPURE FUNCTION ComputeSobel (
101       offset : STD_LOGIC_VECTOR(1 DOWNTO 0)) RETURN byte_t IS
102       VARIABLE gx, gy : INTEGER;
103       VARIABLE total : STD_LOGIC_VECTOR(10 DOWNTO 0);
104
105       VARIABLE A : line3_t;
106       VARIABLE B : line3_t;
107       VARIABLE C : line3_t;
108
109       VARIABLE address : INTEGER;
110       VARIABLE out_line : INTEGER;
111
112     BEGIN
```

```
113       address := (to_integer(unsigned(addrW)) - num_addr_img);
114       ram_ar <= STD_LOGIC_VECTOR(to_unsigned(address REM
    line_addr_jump, 7));
115
116       out_line := address / line_addr_jump;
117
118       IF (out_line REM 3) = 0 THEN
119         A := GetPixels(offset, ram_data_out0);
120         B := GetPixels(offset, ram_data_out1);
121         C := GetPixels(offset, ram_data_out2);
122       ELSIF (out_line REM 3) = 1 THEN
123         A := GetPixels(offset, ram_data_out1);
124         B := GetPixels(offset, ram_data_out2);
125         C := GetPixels(offset, ram_data_out0);
126       ELSIF (out_line REM 3) = 2 THEN
127         A := GetPixels(offset, ram_data_out2);
128         B := GetPixels(offset, ram_data_out0);
129         C := GetPixels(offset, ram_data_out1);
130       END IF;
131
132       gx := - 1 * to_integer(unsigned(A(0))) - 2 * to_integer(
    unsigned(B(0))) - 1 * to_integer(unsigned(C(0)))
133             + 1 * to_integer(unsigned(A(2))) + 2 * to_integer(
    unsigned(B(2))) + 1 * to_integer(unsigned(C(2)));
134
135       gy := + 1 * to_integer(unsigned(A(0))) + 2 * to_integer(
    unsigned(A(1))) + 1 * to_integer(unsigned(A(2)))
136           - 1 * to_integer(unsigned(C(0))) - 2 * to_integer(unsigned(
    C(1))) - 1 * to_integer(unsigned(C(2)));
137
138       total := STD_LOGIC_VECTOR(to_unsigned(ABS(gx) + ABS(gy), 11));
139       RETURN total(10 DOWNTO 3);
140
141     END FUNCTION;
142
143     PROCEDURE EnableWriteCorrectRam IS
144       VARIABLE readAddr : INTEGER;
145       VARIABLE writeAddr : INTEGER;
146
147     BEGIN
148       readAddr := to_integer(unsigned(addrR));
149       writeAddr := to_integer(unsigned(addrW));
150
151       -- edge case (last)
152       IF writeAddr >= ((num_addr_img * 2) - line_addr_jump) THEN
153         ram_we1 <= '1';
154
155         -- edge case (first)
156       ELSIF readAddr < line_addr_jump THEN
157         ram_we0 <= '1';
158         ram_we1 <= '1';
159
160       ELSIF (((readAddr / line_addr_jump) + 1) REM 3) = 0 THEN
161         ram_we0 <= '1';
162       ELSIF (((readAddr / line_addr_jump) + 1) REM 3) = 1 THEN
163         ram_we1 <= '1';
164       ELSE
165         ram_we2 <= '1';
166       END IF;
167     END PROCEDURE;
```

```
168     FUNCTION getRamWriteAddress(plus : INTEGER) RETURN
    rem_addr_type_col IS
169       VARIABLE readAddr : INTEGER;
170     BEGIN
171       readAddr := to_integer(unsigned(addrR));
172       RETURN STD_LOGIC_VECTOR(to_unsigned((readAddr REM
    line_addr_jump) + plus, 7));
173     END FUNCTION;
174
175   BEGIN
176
177     next_state <= state;
178     next_addrR <= addrR;
179     next_addrW <= addrW;
180     dataW <= (OTHERS => '0');
181     addr <= (OTHERS => '0');
182     finish <= '0';
183
184     En <= '0';
185     We <= '0';
186     -- RAM --
187     ram_we0 <= '0';
188     ram_we1 <= '0';
189     ram_we2 <= '0';
190
191     ram_ar <= (OTHERS => '0');
192     ram_aw <= (OTHERS => '0');
193     ram_data_in <= (OTHERS => '0');
194
195     CASE (state) IS
196       WHEN S0 =>
197
198         next_addrR <= halfword_zero;
199         next_addrW <= STD_LOGIC_VECTOR(to_unsigned(num_addr_img, 16))
    ;
200
201         IF start = '1' THEN
202           next_state <= S1;
203
204         END IF;
205
206       WHEN S1 =>
207         En <= '1';
208         addr <= addrR;
209
210         next_state <= S2;
211
212       WHEN S2 => -- read first 4 pixels for first time
213         En <= '1';
214
215         ram_aw <= getRamWriteAddress(0);
216
217         EnableWriteCorrectRam;
218         ram_data_in <= byte_zero & byte_zero & byte_zero & dataR(31
    DOWNTO 24);
219
220         addr <= addrR; -- next state: read first again
221         next_addrR <= STD_LOGIC_VECTOR(to_unsigned(to_integer(
    unsigned(addrR)) + 1, 16));
222         next_state <= S3;
223       WHEN S3 => -- read first 4 pixels once again, and also all the
    rest
```

```
224         En <= '1';
225
226         -- we will want to save it on ram_aw = addr + 1 (which is
    addrR here)
227         ram_aw <= getRamWriteAddress(0);
228         EnableWriteCorrectRam;
229         ram_data_in <= dataR;
230
231         addr <= addrR; -- next state: read next byte in same row
232         IF IsNextAddrRStartLine THEN
233           next_state <= S4; -- process last "read" and repeat
234         ELSE
235           next_addrR <= STD_LOGIC_VECTOR(to_unsigned(to_integer(
    unsigned(addrR)) + 1, 16));
236
237         END IF;
238       WHEN S4 => -- almost same as before
239         En <= '1';
240
241         -- we will want to save it on ram_aw = addr + 2.... (which is
     addrR+1 here)
242         ram_aw <= getRamWriteAddress(1);
243         EnableWriteCorrectRam;
244         ram_data_in <= dataR;
245
246         addr <= addrR; -- next state: read next byte in same row; don
    't update it
247         next_state <= S5;
248       WHEN S5 => -- read last pixel for the 2nd time
249         En <= '1';
250
251         -- we want to save it on addrR + 2
252         ram_aw <= getRamWriteAddress(2);
253
254         EnableWriteCorrectRam;
255
256         ram_data_in <= dataR(7 DOWNTO 0) & byte_zero & byte_zero &
    byte_zero;
257
258         addr <= addrR; -- next state: read last
259
260         -- next will read the first of the next line
261         next_addrR <= STD_LOGIC_VECTOR(to_unsigned(to_integer(
    unsigned(addrR)) + 1, 16));
262
263         -- if this is at least the 2nd line:
264         IF ((to_integer(unsigned(addrR))) / line_addr_jump) > 0 THEN
265           next_state <= S6; -- compute next
266         ELSE
267           next_state <= S2; -- read more
268         END IF;
269
270       WHEN S6 =>
271         En <= '1';
272         We <= '1';
273         addr <= addrW;
274
275         dataW <= ComputeSobel("00") &
276           ComputeSobel("01") &
277           ComputeSobel("10") &
278           ComputeSobel("11");
```

```
279
280         next_addrW <= STD_LOGIC_VECTOR(to_unsigned(to_integer(
    unsigned(addrW)) + 1, 16));
281         IF to_integer(unsigned(addrW)) = (num_addr_img * 2) - 1 THEN
282           next_state <= S7;
283
284           -- handle last case!
285         ELSIF IsNextAddrWLastLine THEN
286           next_addrR <= STD_LOGIC_VECTOR(to_unsigned(num_addr_img -
    line_addr_jump, 16));
287           next_state <= S1;
288
289         ELSIF IsNextAddrWStartLine THEN
290           next_state <= S1;
291
292         END IF;
293
294       WHEN S7 =>
295         finish <= '1';
296         next_state <= S0;
297       WHEN OTHERS =>
298         next_state <= S0;
299     END CASE;
300   END PROCESS c1;
301   seq : PROCESS (clk, reset)
302   BEGIN
303     IF reset = '1' THEN
304       state <= S0;
305       addrR <= (OTHERS => '0');
306       addrW <= byte_zero & byte_one;
307
308     ELSIF rising_edge(clk) THEN
309       state <= next_state;
310       addrR <= next_addrR;
311       addrW <= next_addrW;
312
313     END IF;
314
315   END PROCESS seq;
316 END rtl;
```

Listing 3: VHDL description of the optimised edge de- tector – entity `acc`

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.numeric_std.ALL;
5  ENTITY rams_dist IS
6    PORT (
7      clk : IN STD_LOGIC;
8      we : IN STD_LOGIC;
9      aw : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
10     ar : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
11
12     di : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
13     do : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
14   );
15 END rams_dist;
16 ARCHITECTURE syn OF rams_dist IS
17
18   TYPE ram_type IS ARRAY (89 DOWNTO 0) OF STD_LOGIC_VECTOR(31 DOWNTO
     0);
```

```
19
20    SIGNAL RAM : ram_type;
21  BEGIN
22    PROCESS (clk)
23    BEGIN
24      IF (clk'event AND clk = '1') THEN
25        IF (we = '1') THEN
26          RAM(to_integer(unsigned(aw))) <= di;
27        END IF;
28      END IF;
29    END PROCESS;
30    do <= RAM(to_integer(unsigned(ar)))(7 DOWNTO 0) & RAM(to_integer(
       unsigned(ar)) + 1)(31 DOWNTO 0) & RAM(to_integer(unsigned(ar)) +
       2)(31 DOWNTO 24);
31  END syn;
```

Listing 4: VHDL description of RAM for the optimised edge de- tector – entity $\texttt{rams}_{dist}$

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE WORK.types.ALL;
4
5  ENTITY testbench IS
6  END testbench;
7
8  ARCHITECTURE structure OF testbench IS
9    COMPONENT clock
10     GENERIC (
11       period : TIME := 80 ns
12     );
13     PORT (
14       stop : IN STD_LOGIC;
15       clk : OUT STD_LOGIC := '0'
16     );
17   END COMPONENT;
18
19   COMPONENT memory2 IS
20     GENERIC (
21       load_file_name : STRING
22     );
23     PORT (
24       clk : IN STD_LOGIC;
25       en : IN STD_LOGIC;
26       we : IN STD_LOGIC;
27       addr : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
28       dataW : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
29       dataR : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
30       dump_image : IN STD_LOGIC
31     );
32   END COMPONENT memory2;
33
34   COMPONENT acc
35     PORT (
36       clk : IN bit_t;
37       reset : IN bit_t;
38       addr : OUT halfword_t;
39       dataR : IN word_t;
40       dataW : OUT word_t;
41       en : OUT bit_t;
42       we : OUT bit_t;
43       start : IN bit_t;
```

```
44        finish : OUT bit_t;
45
46        ram_we0 : OUT bit_t;
47        ram_we1 : OUT bit_t;
48        ram_we2 : OUT bit_t;
49        ram_ar : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
50        ram_aw : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
51        ram_data_in : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
52        ram_data_out0 : IN STD_LOGIC_VECTOR(47 DOWNTO 0);
53        ram_data_out1 : IN STD_LOGIC_VECTOR(47 DOWNTO 0);
54        ram_data_out2 : IN STD_LOGIC_VECTOR(47 DOWNTO 0)
55
56      );
57    END COMPONENT;
58
59    COMPONENT rams_dist IS
60      PORT (
61        clk : IN STD_LOGIC;
62        we : IN STD_LOGIC;
63        ar : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
64        aw : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
65        di : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
66        do : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
67      );
68    END COMPONENT;
69
70    SIGNAL StopSimulation : bit_t := '0';
71    SIGNAL clk : bit_t;
72    SIGNAL reset : bit_t;
73
74    SIGNAL addr : halfword_t;
75    SIGNAL dataR : word_t;
76    SIGNAL dataW : word_t;
77    SIGNAL en : bit_t;
78    SIGNAL we : bit_t;
79    SIGNAL start : bit_t;
80    SIGNAL finish : bit_t;
81    SIGNAL ram_we0 : bit_t;
82    SIGNAL ram_we1 : bit_t;
83    SIGNAL ram_we2 : bit_t;
84
85    SIGNAL ram_ar : STD_LOGIC_VECTOR(6 DOWNTO 0);
86    SIGNAL ram_aw : STD_LOGIC_VECTOR(6 DOWNTO 0);
87    SIGNAL ram_data_in : STD_LOGIC_VECTOR(31 DOWNTO 0);
88
89    SIGNAL ram_data_out0 : STD_LOGIC_VECTOR(47 DOWNTO 0);
90    SIGNAL ram_data_out1 : STD_LOGIC_VECTOR(47 DOWNTO 0);
91    SIGNAL ram_data_out2 : STD_LOGIC_VECTOR(47 DOWNTO 0);
92  BEGIN
93    -- reset is active-low
94    reset <= '1', '0' AFTER 180 ns;
95
96    -- start logic
97    start_logic : PROCESS IS
98    BEGIN
99      start <= '0';
100
101     WAIT UNTIL reset = '0' AND clk'event AND clk = '1';
102     start <= '1';
103
104     -- wait before accelerator is complete before deasserting the
```

```
    start
105   WAIT UNTIL clk'event AND clk = '1' AND finish = '1';
106   start <= '0';
107
108   WAIT UNTIL clk'event AND clk = '1';
109   REPORT "Test finished successfully! Simulation Stopped!" SEVERITY
      NOTE;
110   StopSimulation <= '1';
111 END PROCESS;
112
113 SysClk : clock
114 PORT MAP(
115   stop => StopSimulation,
116   clk => clk
117 );
118
119 Accelerator : acc
120 PORT MAP(
121   clk => clk,
122   reset => reset,
123   addr => addr,
124   dataR => dataR,
125   dataW => dataW,
126   en => en,
127   we => we,
128   start => start,
129   finish => finish,
130   ram_we0 => ram_we0,
131   ram_we1 => ram_we1,
132   ram_we2 => ram_we2,
133   ram_ar => ram_ar, -- the same for 3 rams
134   ram_aw => ram_aw, -- the same for 3 rams
135   ram_data_in => ram_data_in, -- the same for 3 rams
136   ram_data_out0 => ram_data_out0,
137   ram_data_out1 => ram_data_out1,
138   ram_data_out2 => ram_data_out2
139 );
140
141 Memory : memory2
142 GENERIC MAP(
143   load_file_name => "/home/mar/pic1_.pgm"
144 )
145 -- Result is saved to: load_file_name & "_result.pgm"
146 PORT MAP(
147   clk => clk,
148   en => en,
149   we => we,
150   addr => addr,
151   dataW => dataW,
152   dataR => dataR,
153   dump_image => finish
154 );
155
156 RamDist0 : rams_dist
157 PORT MAP(
158   clk => clk,
159   we => ram_we0,
160   ar => ram_ar,
161   aw => ram_aw,
162   di => ram_data_in,
163   do => ram_data_out0
164 );
```

```
165   RamDist1 : rams_dist
166   PORT MAP(
167     clk => clk,
168     we => ram_we1,
169     ar => ram_ar,
170     aw => ram_aw,
171     di => ram_data_in,
172     do => ram_data_out1
173   );
174
175   RamDist2 : rams_dist
176   PORT MAP(
177     clk => clk,
178     we => ram_we2,
179     ar => ram_ar,
180     aw => ram_aw,
181     di => ram_data_in,
182     do => ram_data_out2
183   );
184
185 END structure;
```

Listing 5: VHDL description of the optimised edge de- tector – file `test2`

Appendix G VHDL of the optimised edge detector to be programmed on the board

```
1
2 dataW <= ComputeSobel("11") &
3          ComputeSobel("10") &
4          ComputeSobel("01") &
5          ComputeSobel("00");
6
```

Listing 6: what changed in file acc2.vhd in state 6

```
1
2 do <= RAM(to_integer(unsigned(ar)))(31 downto 24)
3     & RAM(to_integer(unsigned(ar))+1)(7 downto 0)
4     & RAM(to_integer(unsigned(ar))+1)(15 downto 8)
5     & RAM(to_integer(unsigned(ar))+1)(23 downto 16)
6     & RAM(to_integer(unsigned(ar))+1)(31 downto 24)
7     & RAM(to_integer(unsigned(ar))+2)(7 downto 0);
8
```

Listing 7: what changed in the ram file

```
1
2 LIBRARY IEEE;
3 USE IEEE.std_logic_1164.ALL;
4 USE IEEE.numeric_std.ALL;
5 USE work.types.ALL;
6
7 ENTITY top IS
8   PORT (
9     clk_100mhz : IN STD_LOGIC;
10    rst : IN STD_LOGIC;
11    led : OUT STD_LOGIC;
12    start : IN STD_LOGIC;
13    -- Serial interface for PC communication
14    serial_tx : IN STD_LOGIC; -- from the PC
15    serial_rx : OUT STD_LOGIC -- to the PC
16  );
17 END top;
18
19 ARCHITECTURE structure OF top IS
20   COMPONENT rams_dist IS
21     PORT (
22       clk : IN STD_LOGIC;
23       we : IN STD_LOGIC;
24       ar : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
25       aw : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
26       di : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
27       do : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
28     );
29   END COMPONENT;
30   -- The accelerator clock frequency will be (100MHz/
     CLK_DIVISION_FACTOR)
31   CONSTANT CLK_DIVISION_FACTOR : INTEGER := 2; --(1 to 7)
32
33   SIGNAL clk : bit_t;
34   SIGNAL rst_s : STD_LOGIC;
```

```
35
36    SIGNAL addr : halfword_t;
37    SIGNAL dataR : word_t;
38    SIGNAL dataW : word_t;
39    SIGNAL en : bit_t;
40    SIGNAL we : bit_t;
41    SIGNAL finish : bit_t;
42    SIGNAL start_db : bit_t;
43
44    -- ram
45    SIGNAL ram_we0 : bit_t;
46    SIGNAL ram_we1 : bit_t;
47    SIGNAL ram_we2 : bit_t;
48
49    SIGNAL ram_ar : STD_LOGIC_VECTOR(6 DOWNTO 0);
50    SIGNAL ram_aw : STD_LOGIC_VECTOR(6 DOWNTO 0);
51    SIGNAL ram_data_in : STD_LOGIC_VECTOR(31 DOWNTO 0);
52
53    SIGNAL ram_data_out0 : STD_LOGIC_VECTOR(47 DOWNTO 0);
54    SIGNAL ram_data_out1 : STD_LOGIC_VECTOR(47 DOWNTO 0);
55    SIGNAL ram_data_out2 : STD_LOGIC_VECTOR(47 DOWNTO 0);
56    -- ram
57
58    SIGNAL mem_enb : STD_LOGIC;
59    SIGNAL mem_web : STD_LOGIC;
60    SIGNAL mem_addrb : STD_LOGIC_VECTOR(15 DOWNTO 0);
61    SIGNAL mem_dib : STD_LOGIC_VECTOR(31 DOWNTO 0);
62    SIGNAL mem_dob : STD_LOGIC_VECTOR(31 DOWNTO 0);
63
64    SIGNAL data_stream_in : STD_LOGIC_VECTOR(7 DOWNTO 0);
65    SIGNAL data_stream_in_stb : STD_LOGIC;
66    SIGNAL data_stream_in_ack : STD_LOGIC;
67    SIGNAL data_stream_out : STD_LOGIC_VECTOR(7 DOWNTO 0);
68    SIGNAL data_stream_out_stb : STD_LOGIC;
69
70  BEGIN
71    led <= finish;
72
73    clock_divider_inst_0 : ENTITY work.clock_divider
74      GENERIC MAP(
75        DIVIDE => CLK_DIVISION_FACTOR
76      )
77      PORT MAP(
78        clk_in => clk_100mhz,
79        clk_out => clk
80      );
81
82    debounce_inst_0 : ENTITY work.debounce
83      PORT MAP(
84        clk => clk,
85        reset => rst,
86        sw => start,
87        db_level => start_db,
88        db_tick => OPEN,
89        reset_sync => rst_s
90      );
91
92    accelerator_inst_0 : ENTITY work.acc
93      PORT MAP(
94        clk => clk,
95        reset => rst_s,
```

```
96        addr => addr,
97        dataR => dataR,
98        dataW => dataW,
99        en => en,
100       we => we,
101       start => start_db,
102       finish => finish,
103
104       ram_we0 => ram_we0,
105       ram_we1 => ram_we1,
106       ram_we2 => ram_we2,
107       ram_ar => ram_ar, -- the same for 3 rams
108       ram_aw => ram_aw, -- the same for 3 rams
109       ram_data_in => ram_data_in, -- the same for 3 rams
110       ram_data_out0 => ram_data_out0,
111       ram_data_out1 => ram_data_out1,
112       ram_data_out2 => ram_data_out2
113     );
114
115   controller_inst_0 : ENTITY work.controller
116     GENERIC MAP(
117       MEMORY_ADDR_SIZE => 16
118     )
119     PORT MAP(
120       clk => clk,
121       reset => rst_s,
122       data_stream_tx => data_stream_in,
123       data_stream_tx_stb => data_stream_in_stb,
124       data_stream_tx_ack => data_stream_in_ack,
125       data_stream_rx => data_stream_out,
126       data_stream_rx_stb => data_stream_out_stb,
127       mem_en => mem_enb,
128       mem_we => mem_web,
129       mem_addr => mem_addrb,
130       mem_dw => mem_dib,
131       mem_dr => mem_dob
132     );
133
134   uart_inst_0 : ENTITY work.uart
135     GENERIC MAP(
136       baud => 115200,
137       clock_frequency => POSITIVE(100_000_000 / CLK_DIVISION_FACTOR)
138     )
139     PORT MAP(
140       clock => clk,
141       reset => rst_s,
142       data_stream_in => data_stream_in,
143       data_stream_in_stb => data_stream_in_stb,
144       data_stream_in_ack => data_stream_in_ack,
145       data_stream_out => data_stream_out,
146       data_stream_out_stb => data_stream_out_stb,
147       tx => serial_rx,
148       rx => serial_tx
149     );
150
151   memory3_inst_0 : ENTITY work.memory3
152     GENERIC MAP(
153       ADDR_SIZE => 16
154     )
155     PORT MAP(
156       clk => clk,
157       -- Port a (for the accelerator)
```

```
158         ena => en,
159         wea => we,
160         addra => addr,
161         dia => dataW,
162         doa => dataR,
163         -- Port b (for the uart/controller)
164         enb => mem_enb,
165         web => mem_web,
166         addrb => mem_addrb,
167         dib => mem_dib,
168         dob => mem_dob
169       );
170     -- ram
171
172     RamDist0 : rams_dist
173     PORT MAP(
174       clk => clk,
175       we => ram_we0,
176       ar => ram_ar,
177       aw => ram_aw,
178       di => ram_data_in,
179       do => ram_data_out0
180     );
181     RamDist1 : rams_dist
182     PORT MAP(
183       clk => clk,
184       we => ram_we1,
185       ar => ram_ar,
186       aw => ram_aw,
187       di => ram_data_in,
188       do => ram_data_out1
189     );
190
191     RamDist2 : rams_dist
192     PORT MAP(
193       clk => clk,
194       we => ram_we2,
195       ar => ram_ar,
196       aw => ram_aw,
197       di => ram_data_in,
198       do => ram_data_out2
199     );
200 END structure;
201
202
```

Listing 8: file top.vhd after changing it and adding dist rams

Appendix H VHDL code to analyse RAM designs

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.numeric_std.ALL;
5  ENTITY rams_dist IS
6    PORT (
7      clk : IN STD_LOGIC;
8      we : IN STD_LOGIC;
9      a_col : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
10     a_lin : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
11     di : IN STD_LOGIC_VECTOR(39 DOWNTO 0);
12     do0 : OUT STD_LOGIC_VECTOR(47 DOWNTO 0);
13     do1 : OUT STD_LOGIC_VECTOR(47 DOWNTO 0);
14     do2 : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
15   );
16 END rams_dist;
17
18 ARCHITECTURE syn OF rams_dist IS
19
20   TYPE ram_type IS ARRAY (359 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO
       0);
21
22   SIGNAL RAM0 : ram_type := (OTHERS => (OTHERS => '0'));
23   SIGNAL RAM1 : ram_type := (OTHERS => (OTHERS => '0'));
24   SIGNAL RAM2 : ram_type := (OTHERS => (OTHERS => '0'));
25
26 BEGIN
27   PROCESS (clk)
28   BEGIN
29     IF (clk'event AND clk = '1') THEN
30       IF (we = '1') THEN
31
32         IF (a_lin = "00") THEN
33           RAM0(to_integer(unsigned(a_col))) <= di(39 DOWNTO 32);
34           RAM0(to_integer(unsigned(a_col)) + 1) <= di(31 DOWNTO 24);
35           RAM0(to_integer(unsigned(a_col)) + 2) <= di(23 DOWNTO 16);
36           RAM0(to_integer(unsigned(a_col)) + 3) <= di(15 DOWNTO 8);
37           RAM0(to_integer(unsigned(a_col)) + 4) <= di(7 DOWNTO 0);
38         ELSIF (a_lin = "01") THEN
39           RAM1(to_integer(unsigned(a_col))) <= di(39 DOWNTO 32);
40           RAM1(to_integer(unsigned(a_col)) + 1) <= di(31 DOWNTO 24);
41           RAM1(to_integer(unsigned(a_col)) + 2) <= di(23 DOWNTO 16);
42           RAM1(to_integer(unsigned(a_col)) + 3) <= di(15 DOWNTO 8);
43           RAM1(to_integer(unsigned(a_col)) + 4) <= di(7 DOWNTO 0);
44         ELSE
45           RAM2(to_integer(unsigned(a_col))) <= di(39 DOWNTO 32);
46           RAM2(to_integer(unsigned(a_col)) + 1) <= di(31 DOWNTO 24);
47           RAM2(to_integer(unsigned(a_col)) + 2) <= di(23 DOWNTO 16);
48           RAM2(to_integer(unsigned(a_col)) + 3) <= di(15 DOWNTO 8);
49           RAM2(to_integer(unsigned(a_col)) + 4) <= di(7 DOWNTO 0);
50
51         END IF;
52
53       END IF;
54     END IF;
55   END PROCESS;
56
```

```
57    do0 <= RAM0(to_integer(unsigned(a_col))) & RAM0(to_integer(unsigned
       (a_col)) + 1) & RAM0(to_integer(unsigned(a_col)) + 2) & RAM0(
       to_integer(unsigned(a_col)) + 3) & RAM0(to_integer(unsigned(a_col)
       ) + 4) & RAM0(to_integer(unsigned(a_col)) + 5);
58    do1 <= RAM1(to_integer(unsigned(a_col))) & RAM1(to_integer(unsigned
       (a_col)) + 1) & RAM1(to_integer(unsigned(a_col)) + 2) & RAM1(
       to_integer(unsigned(a_col)) + 3) & RAM1(to_integer(unsigned(a_col)
       ) + 4) & RAM1(to_integer(unsigned(a_col)) + 5);
59    do2 <= RAM2(to_integer(unsigned(a_col))) & RAM2(to_integer(unsigned
       (a_col)) + 1) & RAM2(to_integer(unsigned(a_col)) + 2) & RAM2(
       to_integer(unsigned(a_col)) + 3) & RAM2(to_integer(unsigned(a_col)
       ) + 4) & RAM2(to_integer(unsigned(a_col)) + 5);
60
61  END syn;
```

Listing 9: Test – first version of RAM

```
1
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.ALL;
4   USE ieee.numeric_std.ALL;
5   ENTITY rams_dist IS
6     PORT (
7       clk : IN STD_LOGIC;
8       we : IN STD_LOGIC;
9       a_col : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
10      di : IN STD_LOGIC_VECTOR(39 DOWNTO 0);
11      do0 : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
12    );
13  END rams_dist;
14
15  ARCHITECTURE syn OF rams_dist IS
16
17    TYPE ram_type IS ARRAY (359 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO
       0);
18
19    SIGNAL RAM0 : ram_type := (OTHERS => (OTHERS => '0'));
20  BEGIN
21    PROCESS (clk)
22    BEGIN
23      IF (clk'event AND clk = '1') THEN
24        IF (we = '1') THEN
25          RAM0(to_integer(unsigned(a_col))) <= di(39 DOWNTO 32);
26          RAM0(to_integer(unsigned(a_col)) + 1) <= di(31 DOWNTO 24);
27          RAM0(to_integer(unsigned(a_col)) + 2) <= di(23 DOWNTO 16);
28          RAM0(to_integer(unsigned(a_col)) + 3) <= di(15 DOWNTO 8);
29          RAM0(to_integer(unsigned(a_col)) + 4) <= di(7 DOWNTO 0);
30
31        END IF;
32      END IF;
33    END PROCESS;
34
35    do0 <= RAM0(to_integer(unsigned(a_col))) & RAM0(to_integer(unsigned
       (a_col)) + 1) & RAM0(to_integer(unsigned(a_col)) + 2) & RAM0(
       to_integer(unsigned(a_col)) + 3) & RAM0(to_integer(unsigned(a_col)
       ) + 4) & RAM0(to_integer(unsigned(a_col)) + 5);
36
37  END syn;
```

Listing 10: Test – second version of RAM

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.numeric_std.ALL;
5  ENTITY rams_dist IS
6    PORT (
7      clk : IN STD_LOGIC;
8      we : IN STD_LOGIC;
9      a_col : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
10     di : IN STD_LOGIC_VECTOR(47 DOWNTO 0);
11     do0 : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
12   );
13 END rams_dist;
14
15 ARCHITECTURE syn OF rams_dist IS
16
17   TYPE ram_type IS ARRAY (359 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO
      0);
18
19   SIGNAL RAM0 : ram_type := (OTHERS => (OTHERS => '0'));
20 BEGIN
21   PROCESS (clk)
22   BEGIN
23     IF (clk'event AND clk = '1') THEN
24       IF (we = '1') THEN
25         RAM0(to_integer(unsigned(a_col))) <= di(47 DOWNTO 40);
26         RAM0(to_integer(unsigned(a_col)) + 1) <= di(39 DOWNTO 32);
27         RAM0(to_integer(unsigned(a_col)) + 2) <= di(31 DOWNTO 24);
28         RAM0(to_integer(unsigned(a_col)) + 3) <= di(23 DOWNTO 16);
29         RAM0(to_integer(unsigned(a_col)) + 4) <= di(15 DOWNTO 8);
30         RAM0(to_integer(unsigned(a_col)) + 5) <= di(7 DOWNTO 0);
31       END IF;
32     END IF;
33   END PROCESS;
34
35   do0 <= RAM0(to_integer(unsigned(a_col))) & RAM0(to_integer(unsigned
      (a_col)) + 1) & RAM0(to_integer(unsigned(a_col)) + 2) & RAM0(
      to_integer(unsigned(a_col)) + 3) & RAM0(to_integer(unsigned(a_col)
      ) + 4) & RAM0(to_integer(unsigned(a_col)) + 5);
36
37 END syn;
```

Listing 11: Test – third version of RAM

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.numeric_std.ALL;
5  ENTITY rams_dist IS
6    PORT (
7      clk : IN STD_LOGIC;
8      we : IN STD_LOGIC;
9      a : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
10     di : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
11     do0 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
12   );
13 END rams_dist;
14
15 ARCHITECTURE syn OF rams_dist IS
16   TYPE ram_type IS ARRAY (87 DOWNTO 0) OF STD_LOGIC_VECTOR(31 DOWNTO
```

```
    0);
17
18    SIGNAL RAM0 : ram_type := (OTHERS => (OTHERS => '0'));
19  BEGIN
20    PROCESS (clk)
21    BEGIN
22      IF (clk'event AND clk = '1') THEN
23        IF (we = '1') THEN
24          RAM0(to_integer(unsigned(a))) <= di;
25        END IF;
26      END IF;
27    END PROCESS;
28
29    do0 <= RAM0(to_integer(unsigned(a)));
30
31  END syn;
```

Listing 12: Test – fourth version of RAM

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4   ENTITY rams_dist IS
5     PORT (
6       clk : IN STD_LOGIC;
7       we : IN STD_LOGIC;
8       a : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
9       di : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
10      do0 : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
11    );
12  END rams_dist;
13
14  ARCHITECTURE syn OF rams_dist IS
15    TYPE ram_type IS ARRAY (87 DOWNTO 0) OF STD_LOGIC_VECTOR(31 DOWNTO
      0);
16
17    SIGNAL RAM0 : ram_type := (OTHERS => (OTHERS => '0'));
18  BEGIN
19    PROCESS (clk)
20    BEGIN
21      IF (clk'event AND clk = '1') THEN
22        IF (we = '1') THEN
23          RAM0(to_integer(unsigned(a))) <= di;
24        END IF;
25      END IF;
26    END PROCESS;
27
28    do0 <= RAM0(to_integer(unsigned(a))) & RAM0(to_integer(unsigned(a))
        + 1)(15 DOWNTO 0);
29
30  END syn;
```

Listing 13: Test – fifth version of RAM

Appendix I VHDL code to analyse rem operator

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.numeric_std.ALL;
5  ENTITY rem_test IS
6    PORT (
7
8      value : IN STD_LOGIC_VECTOR(14 DOWNTO 0);
9      result : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
10   );
11 END rem_test;
12
13 ARCHITECTURE syn OF rem_test IS
14
15   CONSTANT line_addr_jump : INTEGER := 88;
16
17 BEGIN
18   result <= STD_LOGIC_VECTOR(to_unsigned(to_integer(unsigned(value))
       REM line_addr_jump, 7));
19 END syn;
20
```

Listing 14: code to analise rem operator