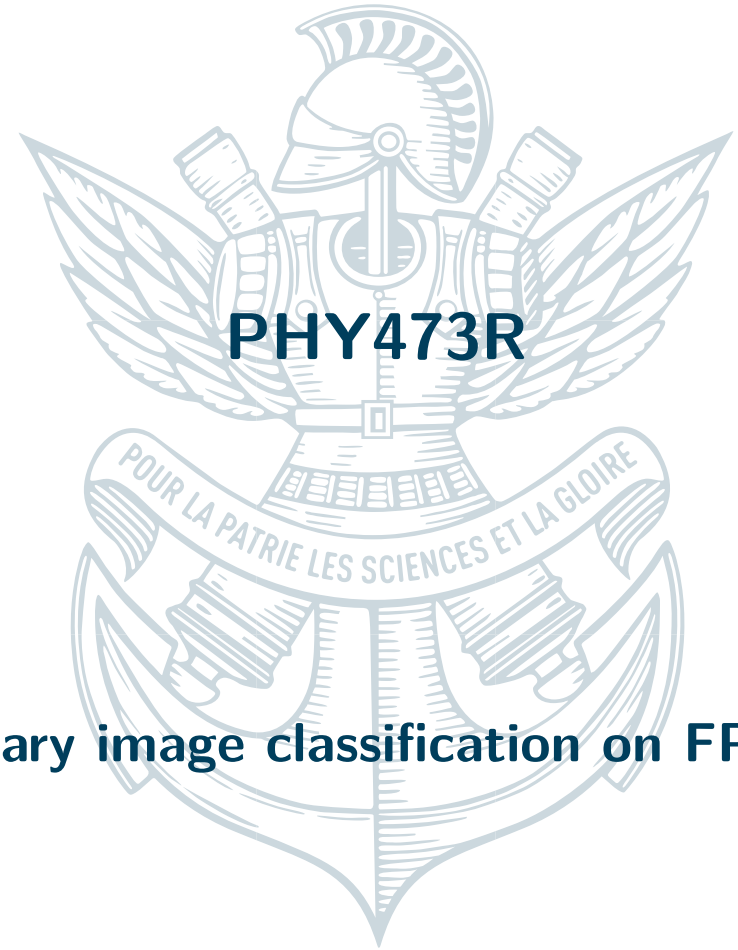


PHY473R

Binary image classification on FPGA

May 2024

Clément Lenoble & Gabriel Pereira de Carvalho

CONTENTS

1 Project overview **3**

2 Definition of the perceptron **4**
2.1 Theoretical functioning 4
2.2 Adaptation to our project 5

3 Training the Model **6**

4 Results **7**

5 User tutorial **9**
5.1 User inputs 9

6 Project design **11**
6.1 State Machine 12
6.2 Datapath components and architecture 13
6.3 Multimodal architecture 16

7 Next steps **16**

8 References **17**

1 PROJECT OVERVIEW

This report presents our project for the course **PHY473R Modal d'Electronique - Circuits logiques programmables (FPGA)** at École polytechnique. The goal of our project is to **perform binary image classification on FPGA**. In order to do this, we used the perceptron, a classical machine learning model for binary classification.. The project was developed on a **Altera DE2-115** board and includes interfaces with a **VGA monitor**, an **LCD** and a **TRDB-DC2 camera** and the training has been done using Python. Our Algorithm performs well with good results both on MNIST dataset and on our real FPGA implementation.

The code can be accessed on the project's Github repository. We have also prepared a video demo available on Youtube.

- Github binary classification: `https://github.com/ArkhamKnightGPC/PHY473R`
- Youtube demo: `https://www.youtube.com/watch?v=SyL3lp6TRCY`

We also worked on **multimodal image classification** (to distinguish 0,1 and 2). A working prototype for this version of the project can be found on our Github. The architecture and logic behind this solution is also described in this report.
The main problem encountered to scale our solution was the memory limitation of the FPGA. The binary classification uses only one perceptron and **40%** of the FPGA's memory. The multimodal classification uses three perceptrons and initially could not fit in the FPGA's memory. We had to reduce the size of the ram used to store the image in order to get this solution to work (each pixel was stored in 4 bits instead of the original 8 bits). Then, the multimodal solution was successfully implemented using **86%** of the FPGA's memory.

- Github multimodal classification: `https://github.com/ArkhamKnightGPC/PHY473R/tree/multimodal`

The following part of this report is dedicated to developing our theoretical motivation problem. We will explain the math behind the perceptron algorithm and how we trained this machine learning model. In the training process, the key element was making a model compatible to our FPGA. Then, we discuss the circuit design, presenting the diagrams we made during conception and how the VHDL code was written. At the end, we provide a section to discuss possible next steps and how the solution could be scaled to classify even more classes given bigger memory constraints.

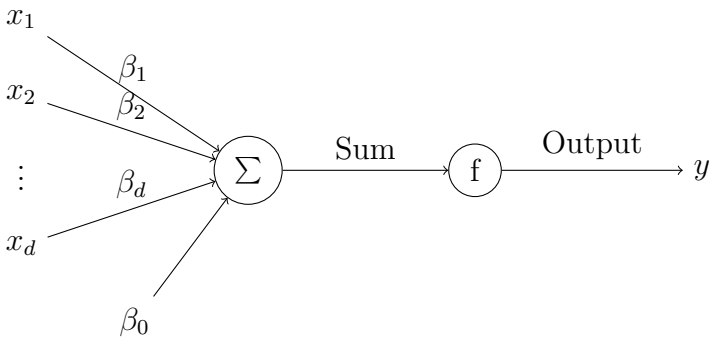
2
DEFINITION OF THE PERCEPTRON

2.1 Theoretical functioning

Developed in 1957 by Frank Rosenblatt, the perceptron is designed to classify datasets that are linearly separable. It takes as input a vector x of the form $x = (x_1, \dots, x_d) \in \mathbb{R}^d$ and then multiplies it by the transpose of the gain vector β, which is determined during training and given by $\beta = (\beta_1, \dots, \beta_d) \in \mathbb{R}^d$. Finally, it adds the bias term β_0 to the sum and applies an activation function, typically a Heaviside function.

Globally, the output of the perceptron is given by the following formula : $y = f(x^\top \beta + \beta_0)$ where $f : \mathbb{R} \to \mathbb{R}$ is the activation function , or also, more intuitively :

$$y = f\left(\sum_{i=1}^{d} \beta_i x_i + \beta_0\right)$$



Let suppose that the data used for training are linearly separable, the aim of the training is to adjust the parameters $(\beta_i)_{i\in\{0,\dots,n\}}$ such that the hyperplane $x^\top \beta + \beta_0 = 0$ separates the two classes of data. Therefore, for all test vector x, y(x) will be either positive or negative and the sign will give us the binary class of the input x.

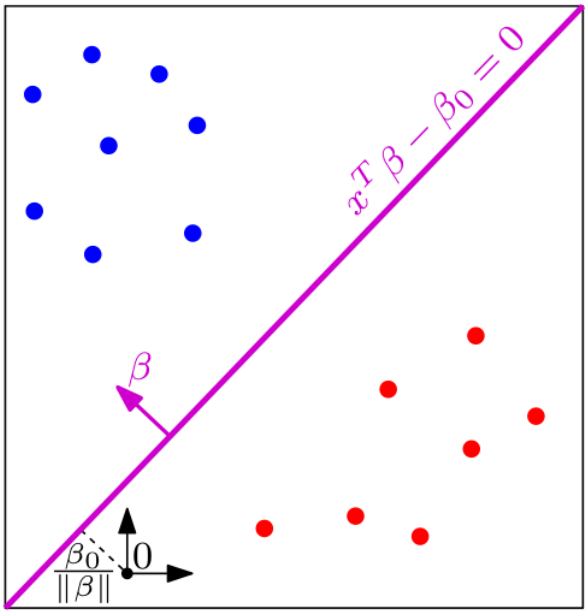


Figure 1: Separating hyperplane on linearly separable data

In our case, the activation function is an the Heavyside function : $H(x) = \begin{cases} 0 & \text{si } x < 0, \\ 1 & \text{si } x \geq 0. \end{cases}$ Hence, the output is either 0 or 1, corresponding to the the binary classification between the different digits evaluated.

2.2 Adaptation to our project

The aim of our project is to classify pictures of digits. Therefore, we have to convert input pictures to vectors, in order to, lastly, give it to the perceptron. As you certainly know, the content of an image can be stored in a matrix of pixel, with several types of pixel-conditioning acording to the image type (vector of $\mathbb{R}^3$ for RGB, real number for gray-level or also bit for black-white). We have chosen to use black and white format because it reduce a lot the memory-size of the image, but with maintaining the important part of the picture's information (for sure, any digit can be expressed in black and white).

Due to technical reasons, the size of the image is 256*256 pixel. Then, the input vector is of the shape $x \in \{0,1\}^d$ where $d = 256 \times 256 = 65536$

The following figure illustrate the transformation of the bit matrix into vector for a 4×4 image

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

Flatten →

0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0

3 TRAINING THE MODEL

To train our model, we used the MNIST dataset. As you can see in figure 2, the MNIST dataset is a large collection of 28x28 handwritten digit images (from 0 to 9). For our binary classification perceptron model we used over 8000 images (split 80% for training and 20% for testing).

We used the python **sklearn** library to train our model, the python script is available on the project's Github repository.

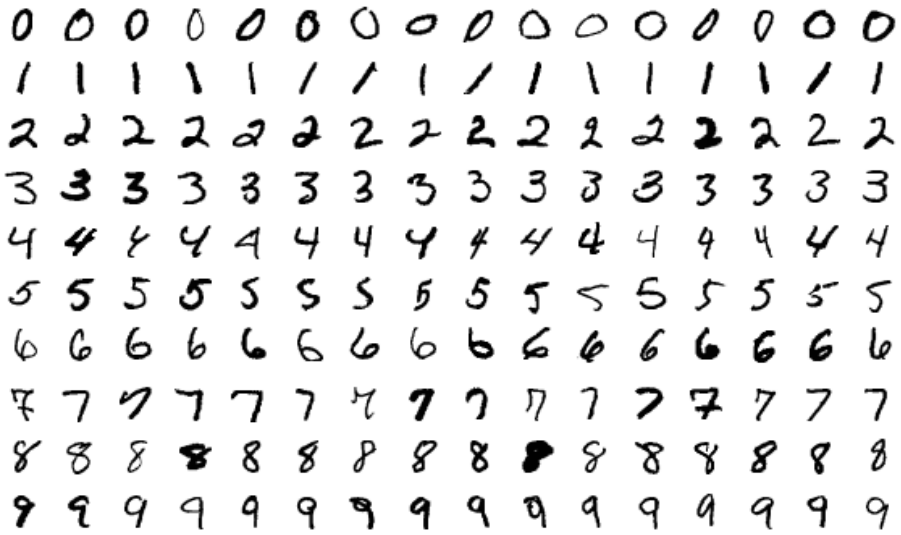


Figure 2: MNIST dataset

To fit the requirements of our FPGA project, we had to:

- Resize the images to 256x256.
- Transform them into black(0) and white(1) with pattern in white and background in black.

- Scale the weights, so we could approximate them with integers.
- Write the weights into a `.mif` file.

Let's discuss the weight scaling. In order to simplify operation with the FPGA, since the `real` data type is not synthesizable and floating point operations are costly we chose to work with only with integers. This means some approximations had to be made in our model.

In order to fit each weight into a 16 bit integer, we had to multiply it by 10000 and take the integer part discarding all decimal digits. In practice, we got good results with our experiments on the FPGA, but this approximation certainly decrease the accuracy of the model a little bit.

4 RESULTS

As discussed in the theoretical section, the perceptron algorithm will assign a weight to each pixel of our `256x256` image that represents the importance of this pixel for the classification. In order to visualise the relative importance of pixels, we can plot heatmaps that clearly show the patterns that our perceptron model is looking for in the image.

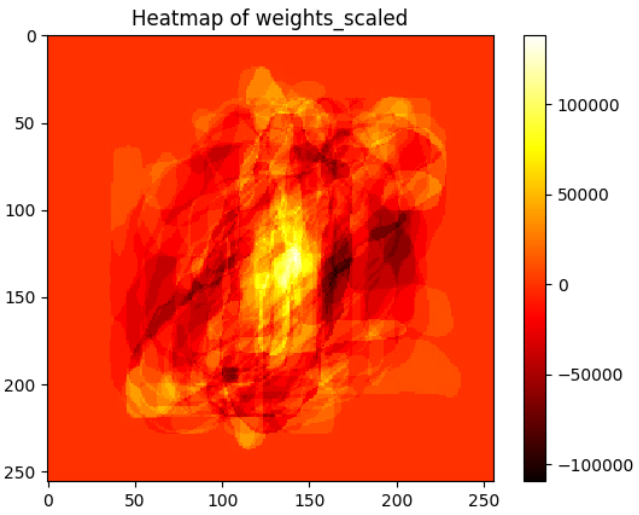


Figure 3: Heatmap for 0/1 perceptron

In figure 3, we see the heatmap of weights for the 0/1 perceptron. Weights were assigned a colormap in order to distinguish which pixels are associated with a one pattern (positive weights) and which pixels are associated with a zero pattern (negative weights). In our python script, this perceptron had a 0.9989 accuracy. However, it is important to understand the

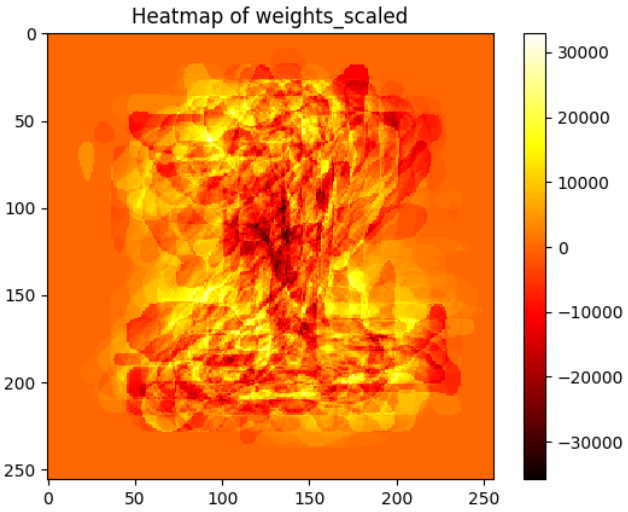


Figure 5: Heatmap for 1/2 perceptron

conditions behind this accuracy. All images in the MNIST dataset are centered on the screen. This means that when testing this model on the FPGA, a good positioning of the digit on the camera image is very important.

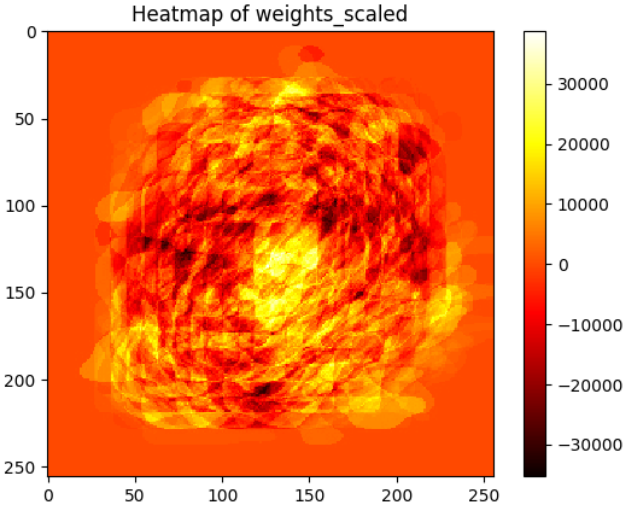


Figure 4: Heatmap for 0/2 perceptron

In figures 4 and 5, we see the heatmaps of weights for the 0/2 and 1/2 perceptrons, respectively. Once again, the models show 0.99 accuracy on the testing data but the good positioning of the digit on the camera image is once again an important point to keep in mind during

experiments on the FPGA.

5 USER TUTORIAL

Before we discurss the details of the circuit's design, let's discuss how a user interacts with the circuit. This procedure explains how to test our project and manipulate the different inputs provided. The goal of this section is to give an overview of the different circuit components before explaining the details.

5.1 User inputs

The user has several inputs to interact with the circuit:

- KEY(0): **reset** button
- KEY(1): **take photo** button
- KEY(2): **start classification** button
- SW(11 downto 0): **exposition time** switches
- SW(17 downto 12): **black and white threshold** switches

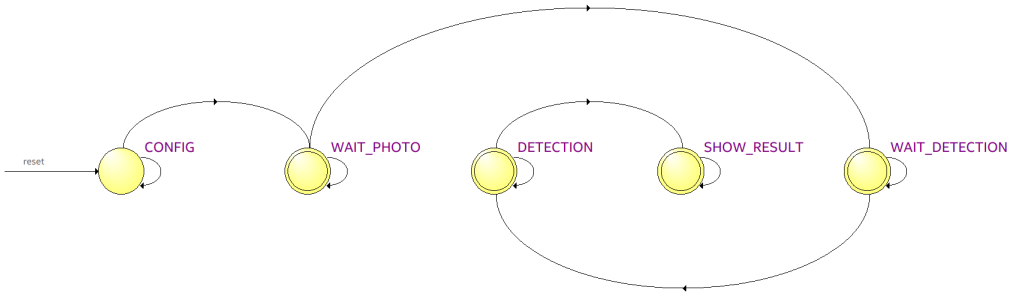


Figure 6: State machine generated by Quartus' state machine viewer

First, let's discuss the initial `CONFIG` state. At this state, we configure registers on the camera via the I2C communication protocol (set exposition time, image dimensions, etc). **In order to change exposition time, the user must modify switches 0 to 11 and press the reset button KEY(0) in order to commit his changes to the camera's registers**.

Then, the system automatically transitions to the `WAIT_PHOTO` state. In the VGA monitor, the user can visualise the image currently stocked in memory.

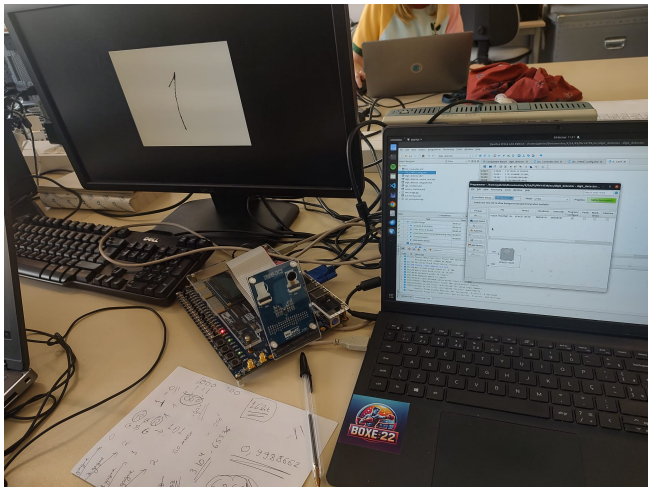


Figure 7: Circuit demo on the lab

Once the handwritten digit is well positioned on the monitor, the user must press the take photo button KEY(1) to move on. From this point on, no further write operations are carried out in the image (in other words, the photo on the screen is now saved in memory).

Then, the system transitions to the `WAIT_DETECTION` state. In this state, the VGA monitor shows the image in black and white format. **In order to change the white intensity threshold, the user must modify switches 12 to 17. The digit must be clearly visible on the monitor before continuing.**

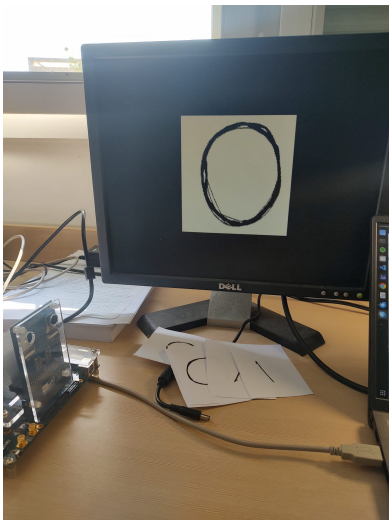


Figure 8: The pattern we want to detect must be clearly visible

If the user is not satisfied with his photo, he can press the reset button to take a new photo. Or if he is satisfied, he must press the KEY(2) button to start classification. Once classification starts, the system transitions to the `DETECTION` state, where

a full sweep of the image RAM and the perceptron RAM will be performed in order to predict the label of the taken photo.

Once the calculation is complete, the system automatically transitions to the `SHOW_RESULT` state. At this state, the LCD displays the detected digit.

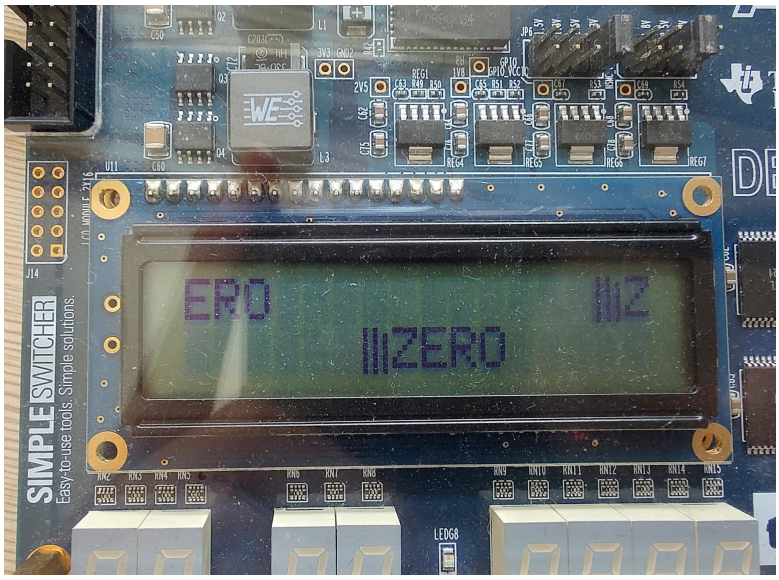


Figure 9: Result message on LCD

6 PROJECT DESIGN

Our architecture follows the **control unit/datapath** paradigm. The project's top level entity `digit_detector` is used as interface with the physical pins listed in our *csv* pin assignment file. The top level entity's architecture uses 2 components: `digit_detector_control_unit` and `digit_detector_datapath`.

The control unit is responsible for implementing the finite state machine that will determine how the system behaves at each clock cycle. It is an essential component in our project. Even though it is smaller than the datapath in terms of lines of code, it completely dictates the datapath's behavior.

6.1 State Machine

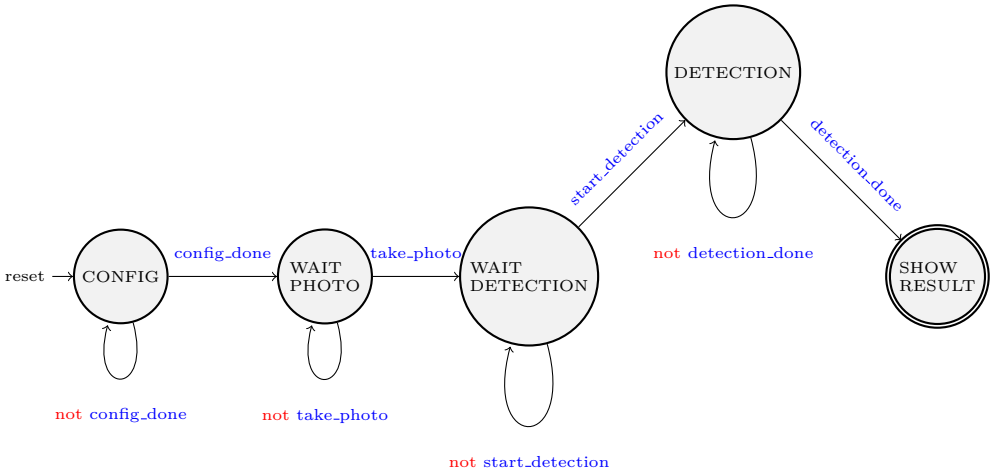


Figure 10: Control unit's state machine

The control unit's finite state machine is represented in figure 10. The initial state is the `CONFIG` state, after we load the program into the FPGA or after we press the reset button this is where the state variable will be.

In the `CONFIG` state, we write several registers in the `TRDB-DC2` camera using the I2C communication protocol and the pins `CMOS_SCLK`, `CMOS_SDAT`. The most important parameters are the image size and the camera's exposition time (controlled with `SW(11 downto 0)`) which will determine how much light the camera collects for the image.

Once the configuration is done, the `config_done` signal will go from 0 to 1 and the transition to the state `WAIT_PHOTO` is automatic. The purpose of the `WAIT_PHOTO` state is to allow the user to position the handwritten digit he wants to test on the camera's field of view. Once he is satisfied with the image he sees on the VGA monitor, he can press the KEY(1) button and activate the `take_photo` signal responsible for disabling write operations to the image RAM. This means the VGA will now read the image saved in memory, and it is this image that will be used for the next stages.

Similarly, the purpose `WAIT_DETECTION` state is to wait for a user input. The pixel input provided to the perceptron is binary (1 for the black pixels and 0 for the white background pixels). In order for the detection to be precise, the black/white threshold (controlled with `SW(17 downto 12)`) must be set according to the user's image. At this state, the same threshold will be applied to the image on the VGA monitor so the user can have feedback on his input. Once he is satisfied with the image he sees on the VGA monitor, he can press the KEY(2) button and activate the `start_detection` signal responsible for triggering the transition to the `DETECTION` state.

In the `DETECTION` state, the datapath will perform a sweep of the image's RAM and the perceptron's RAM using a counter `cnt_detection`. When this counter reaches $256^2 = 65536$,

the perceptron's output is ready for the activation function and the `detection_done` signal triggers the transition to the state `SHOW_RESULT`.

In the `SHOW_RESULT` state we apply the bias to the perceptron's output and the Heaviside step function that will indicate if a zero or a one was detected. In this state, we set the LCD message select signal and the LEDR representing the classification result.

6.2 Datapath components and architecture

The datapath is responsible for interacting with all the other circuit components:

- **I2C controller** for camera configuration
- **VGA interface** to pilot VGA monitor
- **Camera interface** to read camera data
- **RAM image** to write and read image data
- **RAM perceptron** to read perceptron's weights.
- **LCD interface** to write classification result

The components implementing the I2C protocol where provided in the utilities folder of the lab's computer. We made small modifications to fit our project's design:

- we set image dimension to 256×256.
- we set exposition pins as `SW(11 downto 0)`.
- we added `config_done` flag for the state machine transition.

RAM Image and RAM perceptron are provided in the Quartus' IP Catalog (Mega Wizard). We provided a `.mif` file for the RAM perceptron and managed input/output signals.

For simplicity we chose not to implement a separate perceptron entity. The read address is managed by the datapath and the calculation is carried on process `p0`.

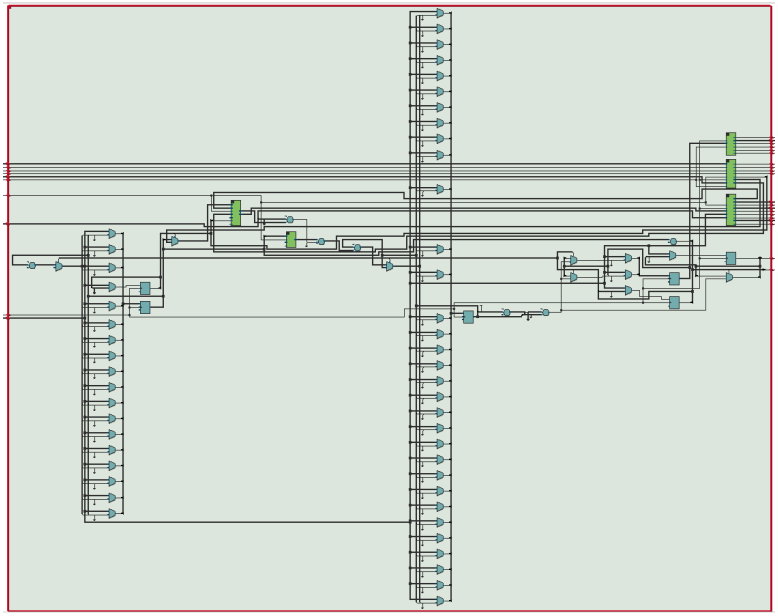


Figure 11: Datapath circuit diagram from Quartus's RTL viewer

As we can see in figure 11, the datapath is quite complex so in this section we will try to give a simplified circuit diagram to show how the components interact and the most important signals.

In our simplified diagram in figure 12, it important to notice that 3 different components must access the RAM image during a run of our project. Therefore, we chose to use a 2 port RAM. One port is dedicated to the VGA and performs only reads operations, this ensures that we always have an image at the monitor in order to guide the user. The second port is shared between the camera and the perceptron.

During detection, the perceptron uses it to read each pixel value in the memory which are them multiplied by the corresponding weight from RAM perceptron and added to an accumulated sum. However, when we're waiting for a photo, the port is used to write the camera's output data into the memory.

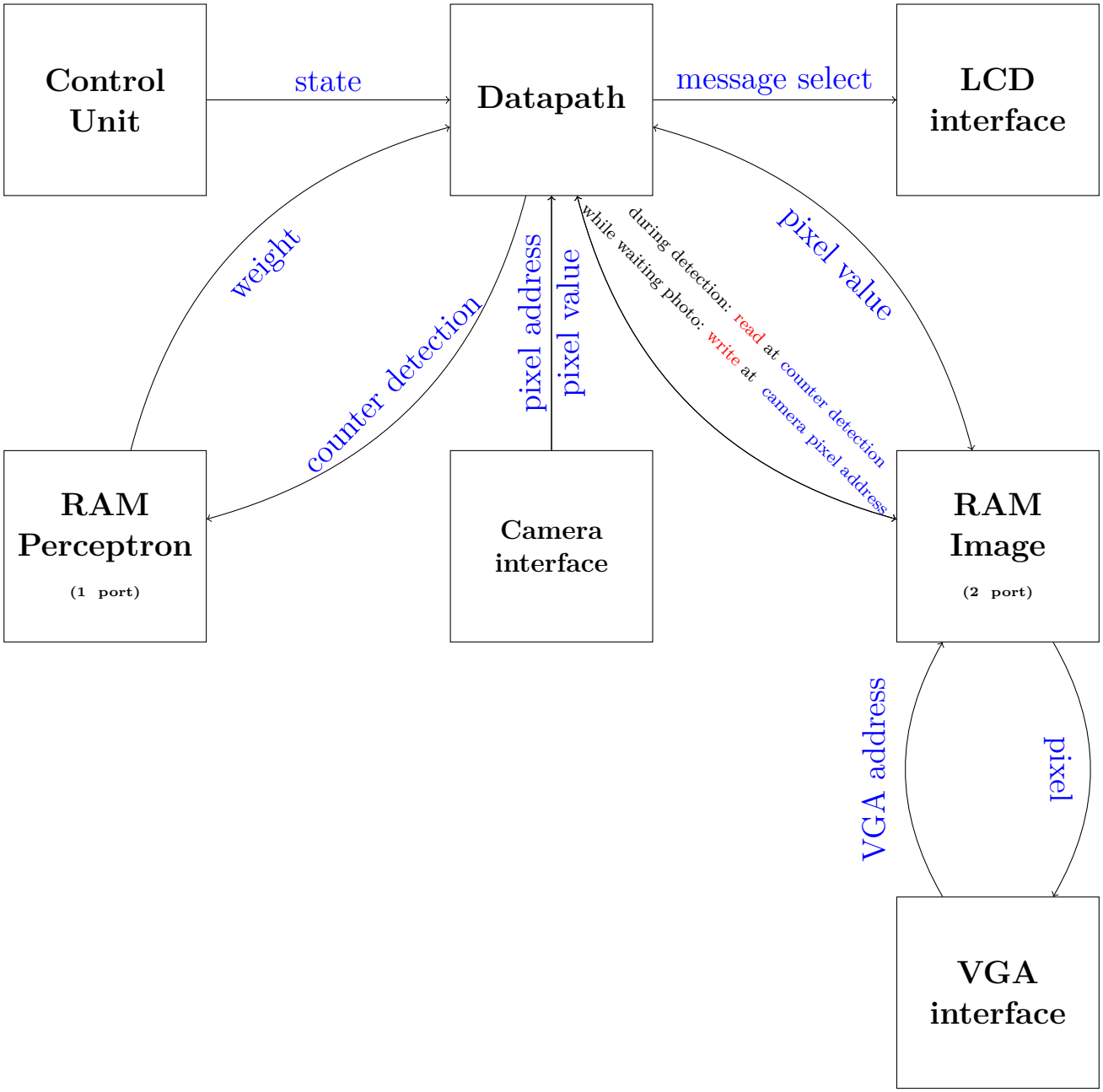


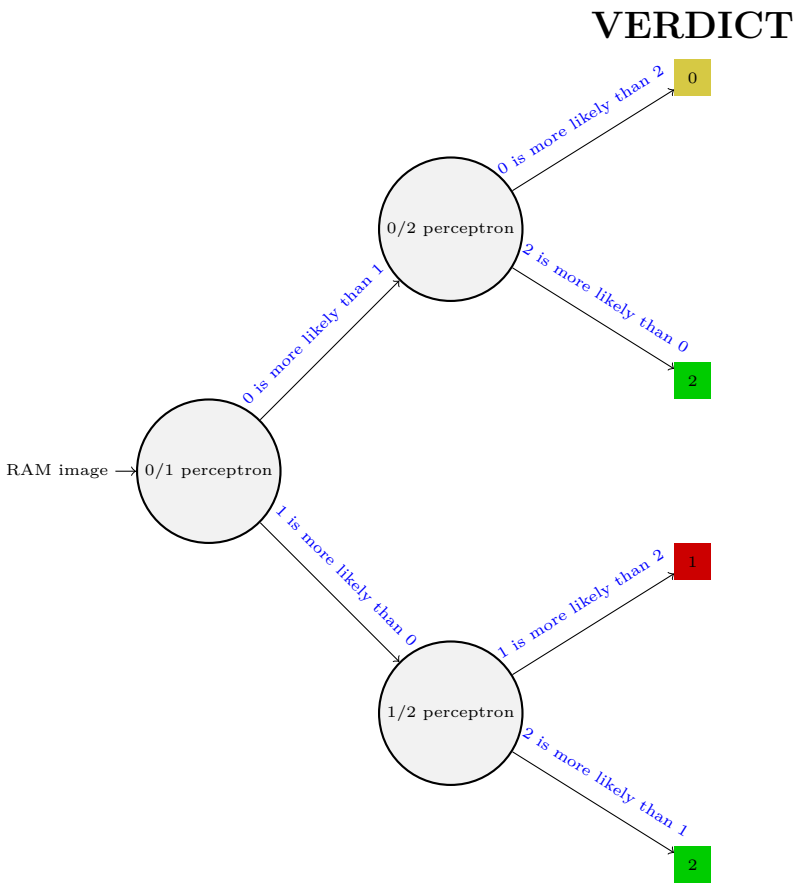
Figure 12: Simplified datapath diagram

The code for each component has been commented for clarity and can provide a detailed view of each component.

6.3 Multimodal architecture

Let's discuss the modifications necessary for multimodal classification. As you can in figure 13, we can reuse our base idea and cascade 3 perceptrons in order to distinguish between three different classes.

The logic of the datapath remains the same. We just add signals to read from 3 perceptron RAMs at a time and compute 3 perceptron outputs.



:

Figure 13: Cascading architecture for perceptrons

7 NEXT STEPS

Our project is currently working and is quite efficient, however, we are exploring ways to enhance our model. We have identified several opportunities for upgrading both its accuracy and features.

- **One-vs-All or One-vs-One** In order to increase the number of classes classified we have implemented a One-vs-One (OVO) method to classify more than two classes. There exists another method to use perceptrons for multi-classification, which is One-vs-All (OvA). In the first, each class is compared every other, it is efficient, acccurate and the convergence is quick but it implies $\frac{n(n-1)}{2}$ classifier, with their 65536 weights each. Due to the lack of memory of the FPGA, this approach might be not easy to implement. Another method is the One-vs-All, which compute, for every classes, the "likelihood" of the class against every others. It implies the implementation of a not binary activation function (to compare) but the number of classifier is only linear in n.

  Hence there is no theoretical clear argument in favour of one, a way of improving our project is to test both and choose the best balance between implementation facility and results.

8 REFERENCES

1. **F. Rosenblatt**. THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN. 1958.

2. **S. Oudot**. Algorithmes pour l'analyse de données en C++ (INF442). École polytechnique, édition 2024.