Digital System Design

NN Image Classification

Tommaso Fava, 1108632

Jean-Baptiste Denoël, 1108427

Group 1

November 4, 2023

Abstract

This report presents the VHDL design of a processing unit that classifies the digits of a 32x32 image using a 3-layer neural network. The images are a binarized subset of the MNIST dataset with size 120, and they are provided through an appropriate coefficient file which initialise a Block RAM. The same applies to the weights for the two fully-connected layers, which are specified in fixed-point representation. The circuit uses a finite-state machine with datapath architecture and it allows a throughput of 4MACs/cycle for both the layers, altough ideas for moving at least to 8MACs/cycle are presented. Input to the board are provided through switches and push-buttons, while the output is shown on the display. Due to an unclear issue, the final accuracy is only around 70% but a possible and very first fix is also presented.

1 Introduction

Neural Network (NN) classification algorithms are becoming more and more popular and their implementation on FPGAs (Field Programmable Gate Arrays) address latency critical application, offering both high performance and optimization.
To start working with NNs, the simplest approach consists on the use of the MNIST (Modified National Institute of Standard and Technology) dataset, which collects 70,000 handwritten digits of size 28x28. However, for this project a subset of 120 binarized (each pixel is either '0' or '1') 32x32 images is used, with the first 60 being shown in Figure 1.

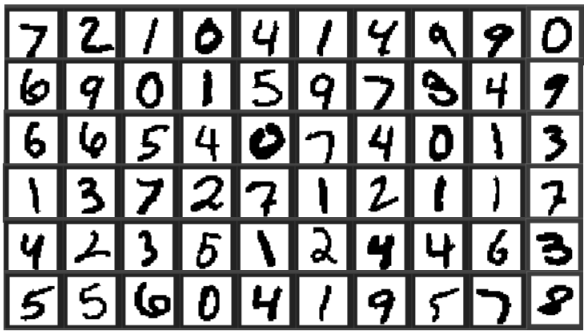

Figure 1: First 60 images of the reduced MNIST dataset.

Those images can be classified by the 3-layer neural network presented in Figure 2, where the fully-connected (FC) layers connect every input to every output. The network works performing dot products between neurons and weights, with the 1024 neurons of the first layer being the image pixels (32x32) and the 10 of the output corresponding to each possible digit (0 to 9).

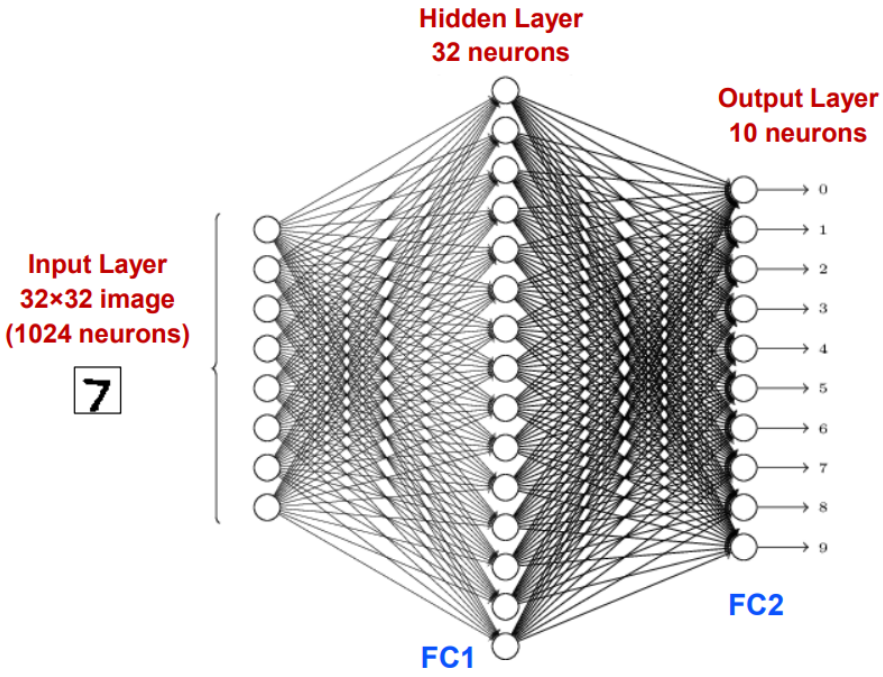

Figure 2: Structure of the 3-layer neural network.

In particular, the first layer takes all the binary pixels 32 different times and multiply them with 32,768 (32x1024) weights, represented as 4-bit fixed-point numbers with scale equal to 2^{-6}. Nevertheless, since they all share the same format, in the design there is no need to consider the scale and so they are always treated as normal 4-bit numbers.
The dot products are calculated as follows:

$$x_1(j) = \sum_{i=1}^{N=1024} (p_i \cdot w_{1_i}) \quad \text{for } j = 0...31$$

A *ReLU activation function* is applied in the neurons of the hidden layer, where it processes the results by keeping positive values and setting negative values to zero :

$$neurons_1(j) = max(0, x_1(j))$$

Regarding the second layer, 320 (10x32) new weights are used to calculate the output neurons. Those weights are represented as 8-bit fixed-point numbers (scale = 2^{-8}), but the same approach explained before is used.
Hence, there are 10 different dot products with no ReLu applied at the end:

$$neurons_2(k) = \sum_{i=1}^{N=32} (neuron_1(i) \cdot w_{2_i}) \quad \text{for } k = 0...9$$

In the output stage, the predicted digit is the index of the neuron with the highest value:

$$prediction = index[max(neurons_2)]$$

The weights and images are specified through appropriate coefficient files (*.coe*) that are used to initialise three different Block RAMs (BRAMs) inside the FPGA (see fig3):

- *images_mem*: 3840x32-bit memory, where each position correspond to 1 image row (32-bit). Each of the 120 images occupy 32 positions.
- *weights1*: 8,192×16-bit memory, where each position correspond to 4 weights (4-bit each).
- *weights2* 80×32-bit memory, where each position correspond to 4 weights (8-bit each).

The three memories are in true dual-port mode, allowing a simultaneous read of two different positions.

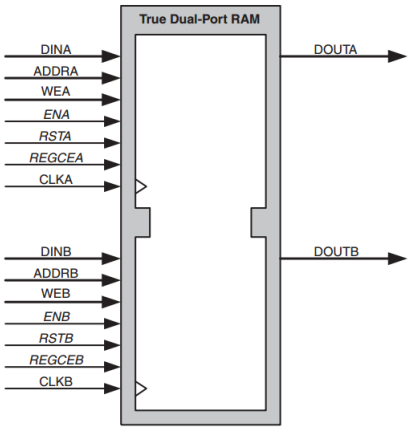

Figure 3: True dual-port BRAM.

2 Circuit Design

The design of the circuit is based on the FSMD (Finite-State Machine with Datapath) architecture and thus two main entities are used: *Control* and *Datapath*. Both of them are structured to follow the sequence of the layers.

2.1 Control Unit

The Control Unit is a Moore Machine with 4 different states: *init*, *layer1*, *layer2* and *finish*.
Differently from the previous project, here the machine set the enables for the registers but not the selectors for the multiplexer, which are totally controlled by the Datapath Unit. Instead, three reset signals are used to make it possible for the user to compute different images.
The behaviour of this unit is quite simple and it is presented in Figure 4. An external reset keeps the state to *init*, where no registers are enables and all the signals stay in their default value. When the reset goes low, the machine starts working. Both states *layers1* and *layers2* set their enable and reset signal to the needed value, and then iterate till the respective *done* signal goes high in the Datapath. These iterations are based on two counters that are responsible for all the activities in the layers, starting from the memory management. When the last counter is done the machine goes to state *finish*, where the prediction is stored on the output register and becomes available to the user.

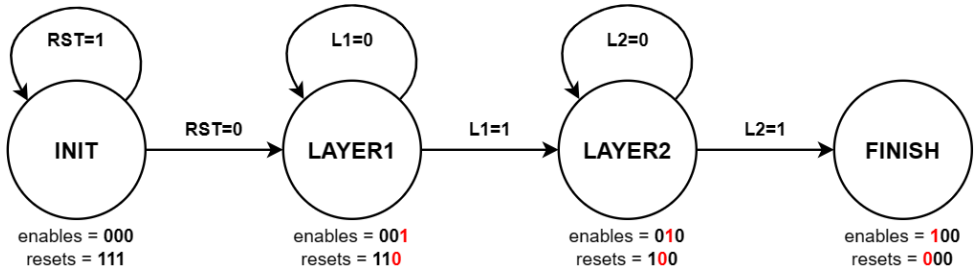

Figure 4: Mealy state machine used for the Control Unit.

When the user wants to use a new image, he just need to reset the circuit, input a number and unset the reset.

2.2 Datapath Unit

As always, the Datapath is the core of the circuit and thus the most interesting part. This time it is not only responsible for the operations on the signals, but it also independently controls the sub-activities within each layer.

2.3 Layer 1

The first layer takes the pixels of the selected images and performs a dot product between them and the first weights. To do it, there is the need to properly read data from the memories, starting by using only one port to make things easier.
To access the images, *images_mem* component is used. Since the positions in the memory are 3840, a 12-bit address is used to select which image the user wants to predict. The process of generating this address is shown in Figure 5.

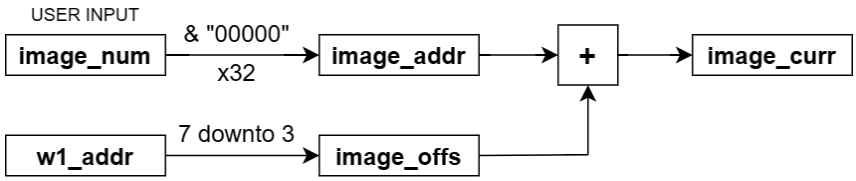

Figure 5: Image address generation.

Basically, the circuit waits for *image_num*, that is a number in range [0,119] directly associated with the image. Then, since every image is contained in 32 consecutive rows (images are 32x32 and each row is 32-bit), that number must be multiplied by 32 so that it can indicate the base address of every image. Being 32 a power of 2, this operations is a simple shift-left by 5 positions. After, one must consider the need to move through all the rows to read all the pixels. This means adding to the base address an offset which has to be in range [0,31]. A broader view of the architecture helps to understand the solution: the entire layer is managed by *w1_addr*, a 13-bit counter than counts till 8191. This is in fact the size of the memory storing the weights (*weights1*) and the number is used to move within all the positions.

Since with a single port 4 weights are read every clock cycle, an entire image row is processed in 8 cycles. Thus, the offset must be started from 0, incremented every 8 cycles and reset to 0 when it reaches 31 (that is 256 cycles, or better 2^8 increments of *w1_addr*). In particular, the reset is needed because each row is used 32 times (1 for every neuron of the hidden layer). If one looks at the sub-vector *w1_addr(7 downto 3)* in Figure 6, that clearly does the job and so there is no need for an additional counter.

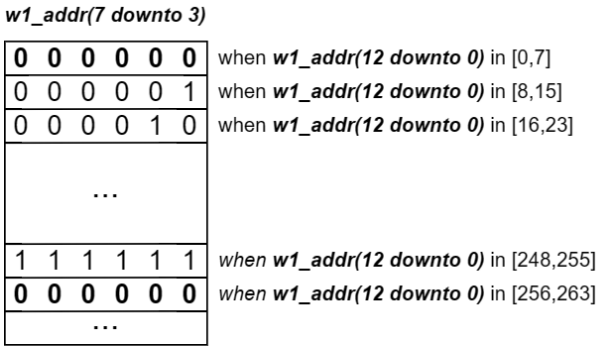

Figure 6: Updates of the offset counter in relation to the updates of *w1_addr*.

Concerning the memory storing the weights, as already said, the address is specified by the entire *w1_addr*. Having both the inputs, they are then processed according to the steps shown in Figure 7. This is a Multiply-Accumulate (MAC) unit with a multiplexer on the pixels side to use only four of them at a time. Since the behaviour of *sel_pixel* is very similar to *image_offs* (here the increment is every cycle and the reset after 8), the sub-vector *w1_addr(2 downto 0)* is used. However, *sel_pixel* must have a delay of one cycle to have the pixels and the weights aligned.

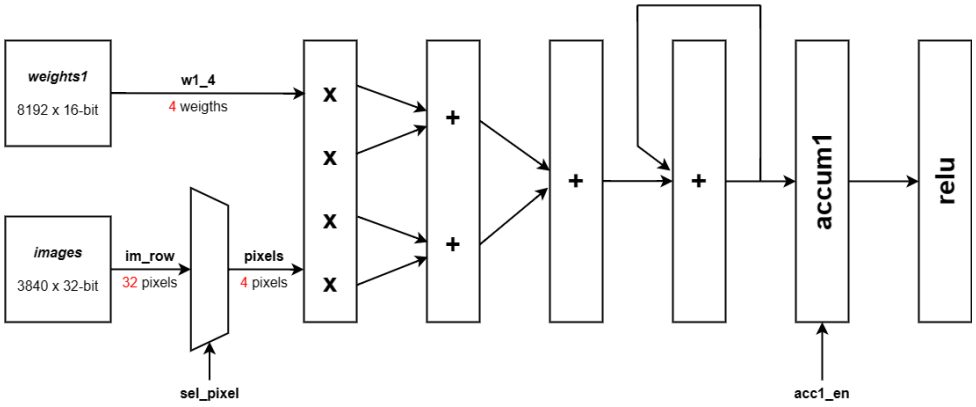

Figure 7: 4 MACs architecture of Layer 1.

Actually, in this layer there is no need for a proper multiplication: one of the input is indeed the binary pixel and this means that the result will be either zero ($p = 0$) or the weight ($p = 1$). Hence, 4 multiplexers are more than enough.
The accumulator and the ReLu units also deserve special attention. The first is an array of 32 vectors specified by the following for-generate statement:

```
    accum_layer1: for i in 0 to 31 generate
    process(clk)
    begin
      if rising_edge(clk) then
        if l1_rst = '1' then
          accum1(i) <= (others => '0');
        elsif acc1_en(i) = '1' then
          accum1(i) <= accum1(i) + add3_l1;
        end if;
      end if;
    end process;
  end generate;
```

All the registers are reset by the signal coming from the Control Unit, while the enable is managed inside the Datapath. The idea is to use a shift register with all 0's and a 1 moving left after 256 cycles, so that only the right accumulator is used every moment. The counter that runs it comes again from the main one.

```
enable1 : process(clk)
begin
  if rising_edge(clk) then
    if init_en1 = '1' then
      acc1_en <= (0 => '1', others => '0');
    elsif shift1 = '1' then
      acc1_en <= acc1_en(30 downto 0) & '0';
    end if;
  end if;
end process;
```

Regarding the latter, the activation function is implemented by just setting the accumulators to 0 when they are negative. An example of the result is presented in Figure 8.

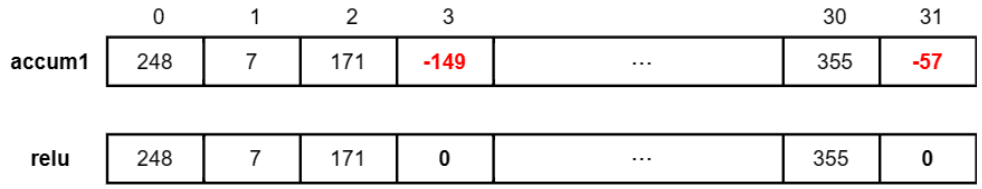

Figure 8: Example of the ReLu activation function.

2.4 Layer 2

The second layer computes new dot products using the output of the ReLu unit together with the weights from *weights2* memory. As shown in Figure 9, the architecture is the same of the previous layer except for the use of pipeline registers and a comparator entity at the end, when there is no ReLu now.

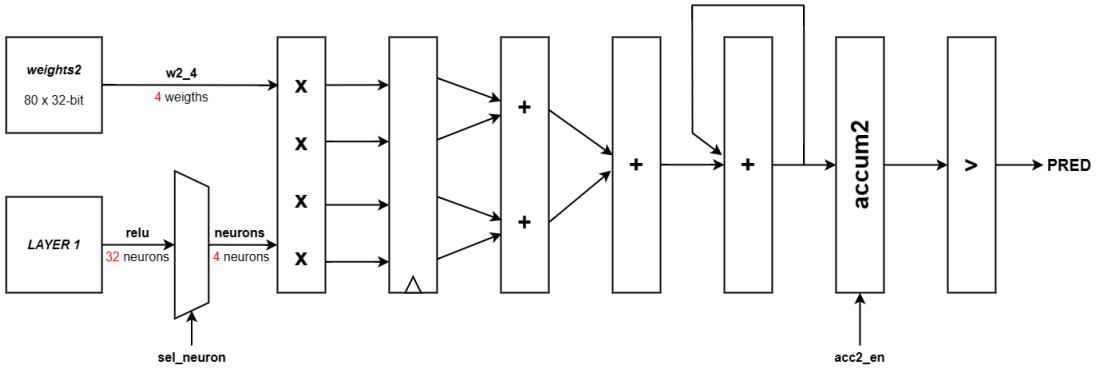

Figure 9: 4 MACs architecture of Layer 2.

The pipeline registers after the multiplications (this time real ones, implemented on DSP blocks) allow the use of a faster clock. Of course, this was not the initial choice and the timing issue found during the implementation could also be solved reducing the frequency from 100MHz to 74MHz. However, pipelining is the best solution.

The Comparator (last unit of the architecture) is instead a separate component (instantiated inside the Datapath) that takes two vectors of size 26 and their positions in the array to output the maximum value and its index. Its role is fundamental to predict the digit, which is indeed the index of the neuron storing the biggest value after Layer 2. Figure 10 presents its architecture, where the inputs come from two multiplexers managed from the Datapath. The final register is needed to use the output as an input for the following comparison.

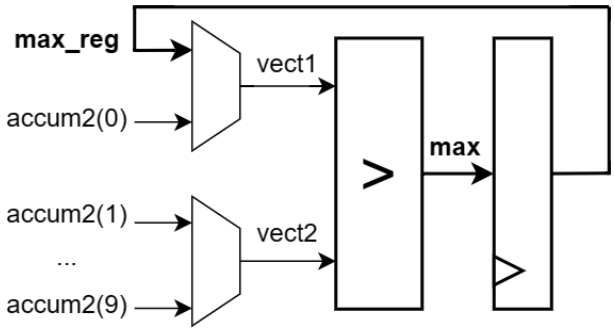

Figure 10: Architecture of the Comparator Unit.

3 Evaluation

Completed the design of both Control and Datapath units, they are instantiated in the Circuit entity and their ports are connected together. From there, a testbench sending the *rst* and *image_num* signals can check the behaviour of the design. In fact, the assignment came with the same NN implemented in a C program that provides the correct predictions and all the possible intermediate values. Comparing them to the simulation outputs evaluates the results.

3.1 Behavioural Simulation

The first step to verify the circuit is performing a behavioural simulation. Figure 11 displays the result of the simulated prediction of image 0 and 1, which are the same received from the script.

Figure 11: Behavioural Simulation of images 0 and 1.

3.2 Synthesis and Implementation

After the behavioural check, synthesis and implementation steps were started. Both of them ended with good results, having no critical warnings in the synthesis report and positive slack after the implementation, as presented in Table 1.

Table 1: Post-implementation timing summary with 10 ns clock period.

Slack	Value (ns)
WNS	0.447
WHS	0.181
WPWS	4.500

Table 2 shows instead the utilization summary, which highlights the complexity of this design when compared with the previous projects, where onky few resources were used.

Table 2: Basys3 utilization summary.

Resource	Utilization	Utilization (%)
LUT	1253	6.02
FF	915	2.20
BRAM	9	18.00
DSP	4	4.44
IO	85	35.85

3.3 Post-Implementation Timing Simulation

Before deploying the circuit on the board, a final post-implementation timing simulation was launched to have a definitive idea of the behaviour.
Unfortunately, as one can notice from Figure 12, the results are not the expected ones. In particular, the figure shows that the predicted digit of images 0 and 1 are still 7 and 2, but the maximum values (from which the predictions depend) are different from before. This happens with all the images, with the result of having only around 70% of correct predictions when simulating all the 120 images. Anyway, since this project hardware and not pure machine learning, talking about percentage of success is not sufficient.

Figure 12: Post-Implementation Timing Simulation of images 0 and 1.

The focus was then to try to identify the issue. Due to internal optimizations, Vivado renames many of the signals after the synthesis and that makes it hard to debug the design. Anyway, the accumulator arrays (*accum1* and *accum2*) can still be loaded on the waveform viewer.
Figure 13 displays the values in the last 20 positions (the viewer window is not big enough to fit 32) of *accum1* for image 0 and they are exactly the ones found in the behavioural simulation, as one can notice also from a comparison with Figure 11. This means that the first layer is processed correctly.

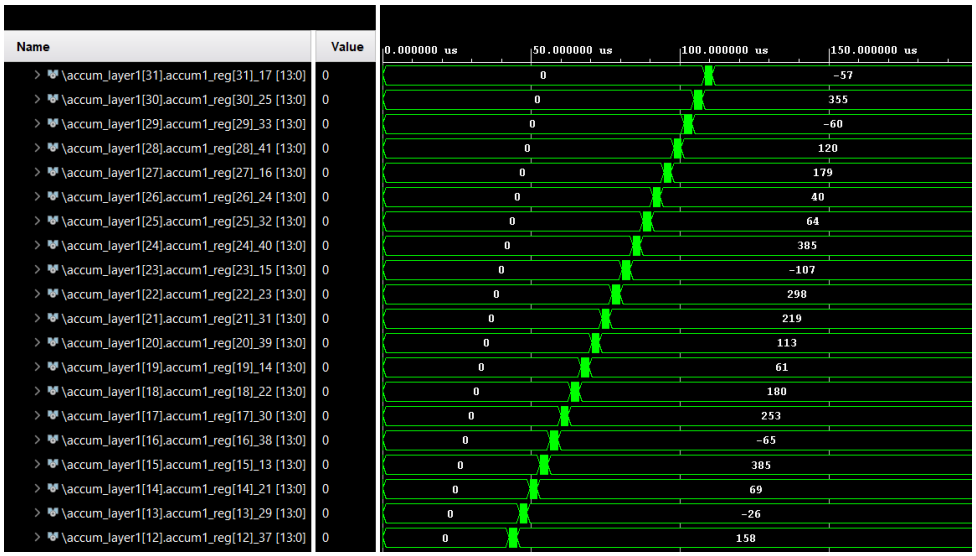

Figure 13: Values in the last 20 positions of *accum1* for image 0.

Looking instead to the second layer as in Figure 14, it is immediately clear that all the values stored in *accum2* (from the very first one of the first neuron to the last) are wrong.

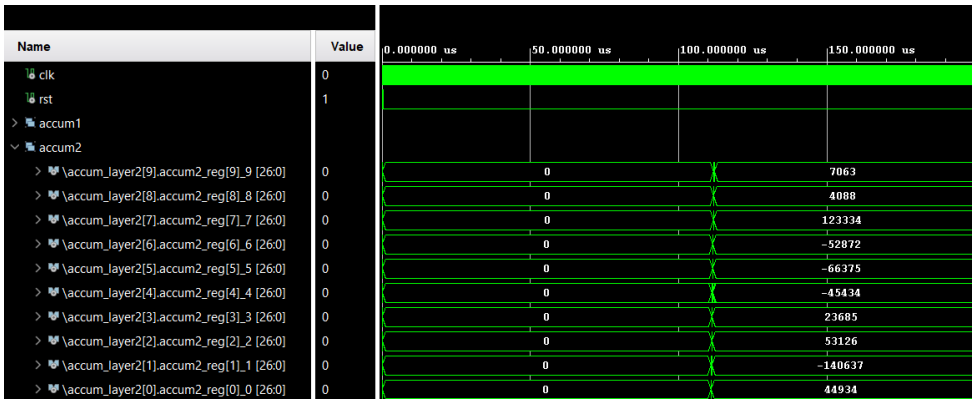

Figure 14: Values in *accum2* for image 0.

Since both the layers share the exact same architecture, it is strange to see such a wrong computation, especially because all the timing constraints are met. The hypothesis of a timing issue was then definitely discarded when the same results were found in a post-synthesis functional simulation, which does not take the delays into consideration.
Hence, it is after the synthesis that the problem arises, but nothing in its report leads to known possible issues. To debug the design, there is the need to access the signals used to compute the values stored in *accum2*, but they are not available after the synthesis.
To make them appear on the viewer, the solution consists in putting them as external ports and the first step was to do it with the signals coming from the four multipliers (*mul1_l2*, *mul2_l2*, *mul3_l2*, *mul4_l2*). Launching a post-implementation timing simulation just after this, the final results suddenly became correct, as can be noticed from Figure 15 (note that *neuron_value* is now in Q13.14 fixed-point representation).

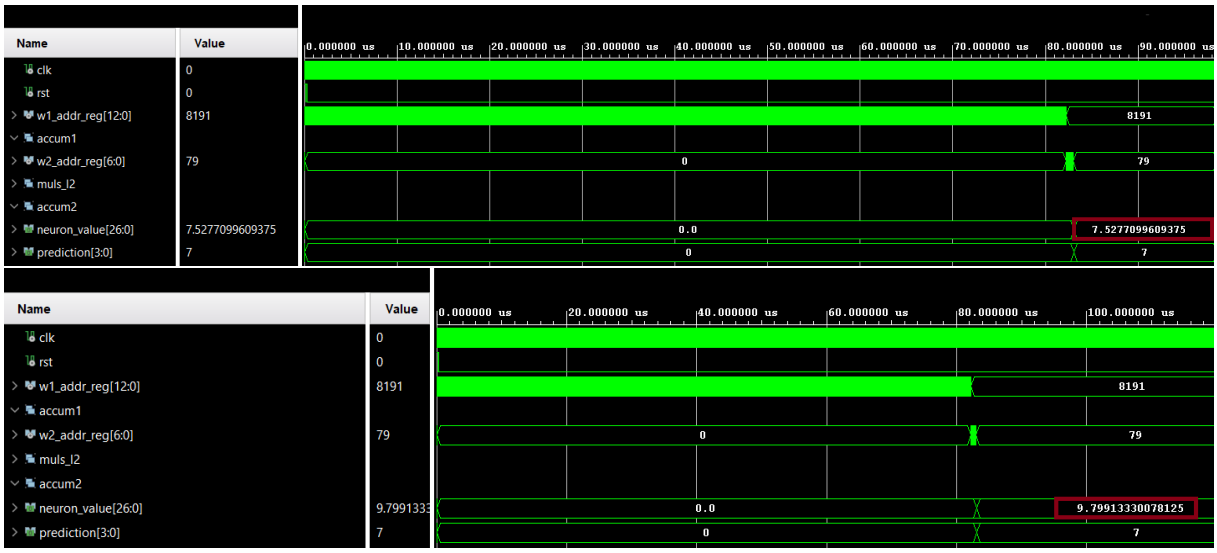

Figure 15: Post-Implementation Timing Simulation of image 0 before and after the modification.

Since the fix is unclear as much as the related issue, the choice was to continue with the initial design and just put here the rest. With more days of work there could certainly be a reasonable explanation for that.

4 User Interface

Before describing the external interface of the circuit, Figure 16 shows the final schematic of the Circuit Unit, which links together Control and Datapath.
Then, to use the circuit on the Basys3 board, there is the need to instantiate not only the Circuit Unit but

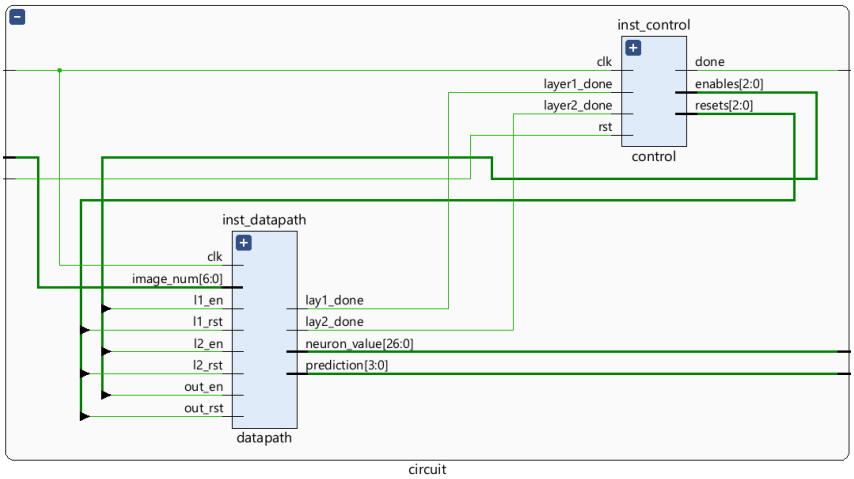

Figure 16: Schematic of the Circuit Unit.

also the controller of the segment display (*disp7*) and the debouncer for the push-buttons (*btn_debounce*), which are standard design files. A top view of the design involving all the needed components is presented in Figure 17.
Moreover, the most important thing is to provide a way for the user to interface with the board. This is done using switches, leds, buttons and the display. In particular, the seven rightmost switches (*sw(6 downto 0)*) are used to input the image number while the leftmost switch (*sw(15)*) acts as the reset for the Control Unit, starting the computation when put to high. The four digits of the display show instead either the predicted digit or the maximum value, switching due to the use of the right push-button (*btnR*). Regarding the maximum number, since it comes in 27-bit size (Q13.14 fixed-point representation) and the digits are only four, it is reduced to 16-bit accepting a loss of precision, as presented in Figure 18. Its MSB represents the sign and thus it is used to control a display point (*dp3*) that works as a minus sign (probably never needed). The point between the third and fourth digits (*dp1*) separate the integer from the fractional part.

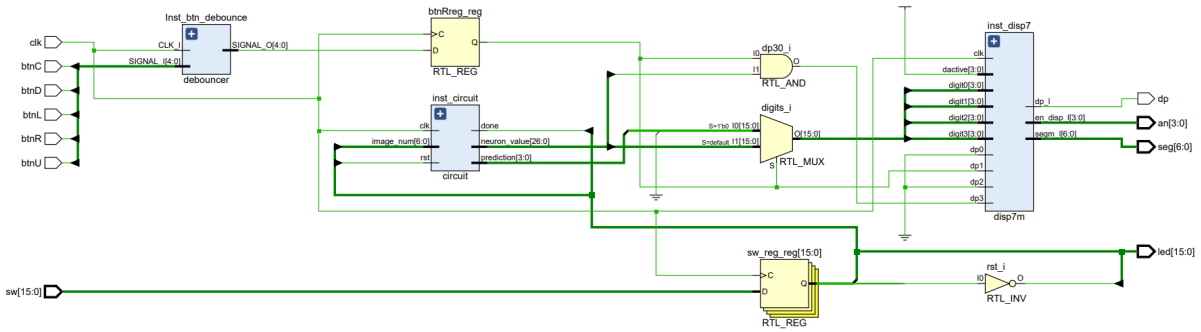

Figure 17: Top view schematic of the design.

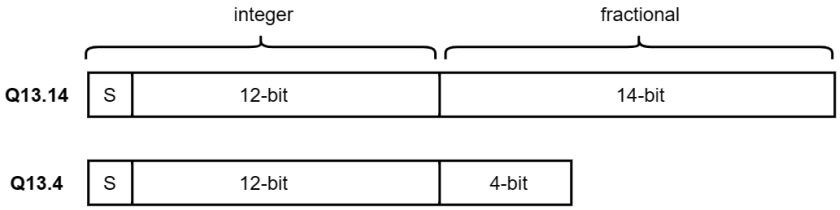

Figure 18: Reduction from Q13.14 to Q13.4.

5 Results and Improvements

The first objective of this project was to write, simulate and implement a fully functional neuron network capable of recognise any image from the given dataset. Even though the behaviour of the circuit presents the issue previously described, this goal can be considered as achieved with the following performance:

- Throughput: 4 MACs/cycle
- Clock Period : 10 ns
- Processing time for image : 111.725 us

Then, the second objective was to improve the performance. Due to the problem faced after the synthesis, at the end there was no time to modify the design but in general this would have made no sense since the main concern was to give an explanation to both the problem and the solution. However, some basic ideas can be presented here.
First, adding more pipeline registers could allow to reduce again the clock period (that already went from 13.5 ns to 10 ns thanks to one register). Looking at the critical path, in Layer 1 or after the first 2 adders of Layer 2 will be a suitable position.
Moreover, the throughput can easily be doubled using a 8 MACs unit, as presented in Figure 19 (where the pipeline registers are not present for a matter of size). Since the current configurations of the BRAMs use only one port, just reading from the second port too will allow to have 8 weights available every cycle, while the image memory already outputs up to 32 pixels with a single port. Hence, with small modifications the number of clock cycles needed for the computation will be halved.

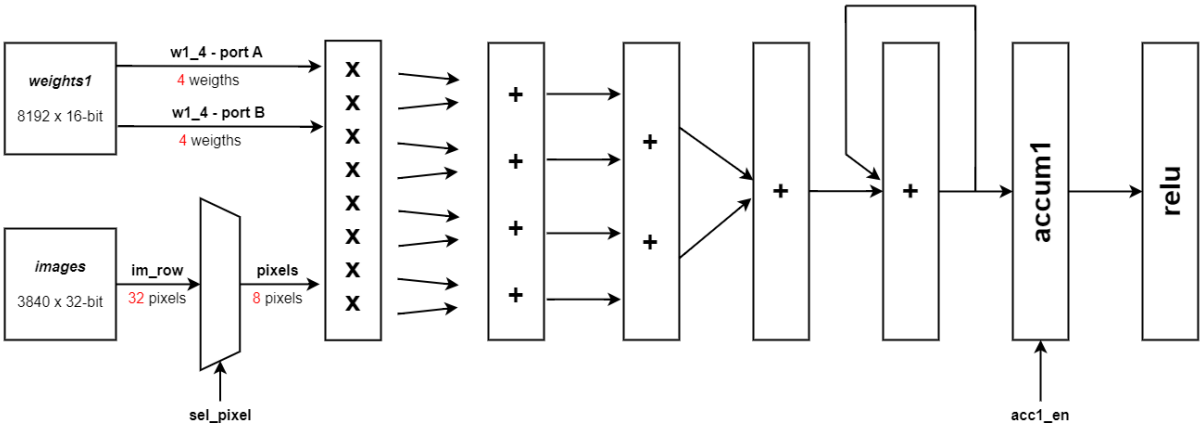

Figure 19: Architecture of MAC8 for the first layer

6 Conclusion

This project was the culmination of learning FPGAs and VHDL design. The goal was to implement a 3-layer neural network capable of determining digits from a 32x32-bit binarized image and unfortunately it was not completely achieved, since for unknown reasons the post-synthesis simulation does not match the correct behavioural simulation. Anyway, a (weird) working solution have been found and in general the effort and the things learned to have a fully functional behavioural simulation can not be underestimated. For the first time memories were introduced in the design and, in general, never before the complexity was that high, having to deal with delays and pipeline registers. This certainly improved the author's hardware knowledge and skills.
This work had to finish with all the ideas implemented, starting from a working 4 MACs unit and finishing with the best possible optimization. Instead, all those things remained on the paper mainly due to the lack of time. It is doubtless frustrating, but the awareness of having put the best effort stays.