École Polytechnique Montreal

ELE8307: Rapport d'étape

Students :
Hossein Askari,
Nathan Heraief,
Alexandre Riviello

Teacher :
Jean-Pierre David

September 12, 2019

Contents

1 Introduction

In this project, we are asked to develop an optimized embedded platform allowing the fast execution of pre-trained ANNs using ternary logic for weights and binary logic for the input values. We already have a hardware platform and a software application provided. To achieve this goal, tasks will be broken down and detailed throughout this report.
We will be using a SocKit board containing a SoC which includes a Cyclone V FPGA and an ARM A9 processor. By executing part of the program on the A9 processor and by accelerating most calculations through hardware, a considerable speed increase is expected. The general architecture of the SoC is illustrated in Figure 1

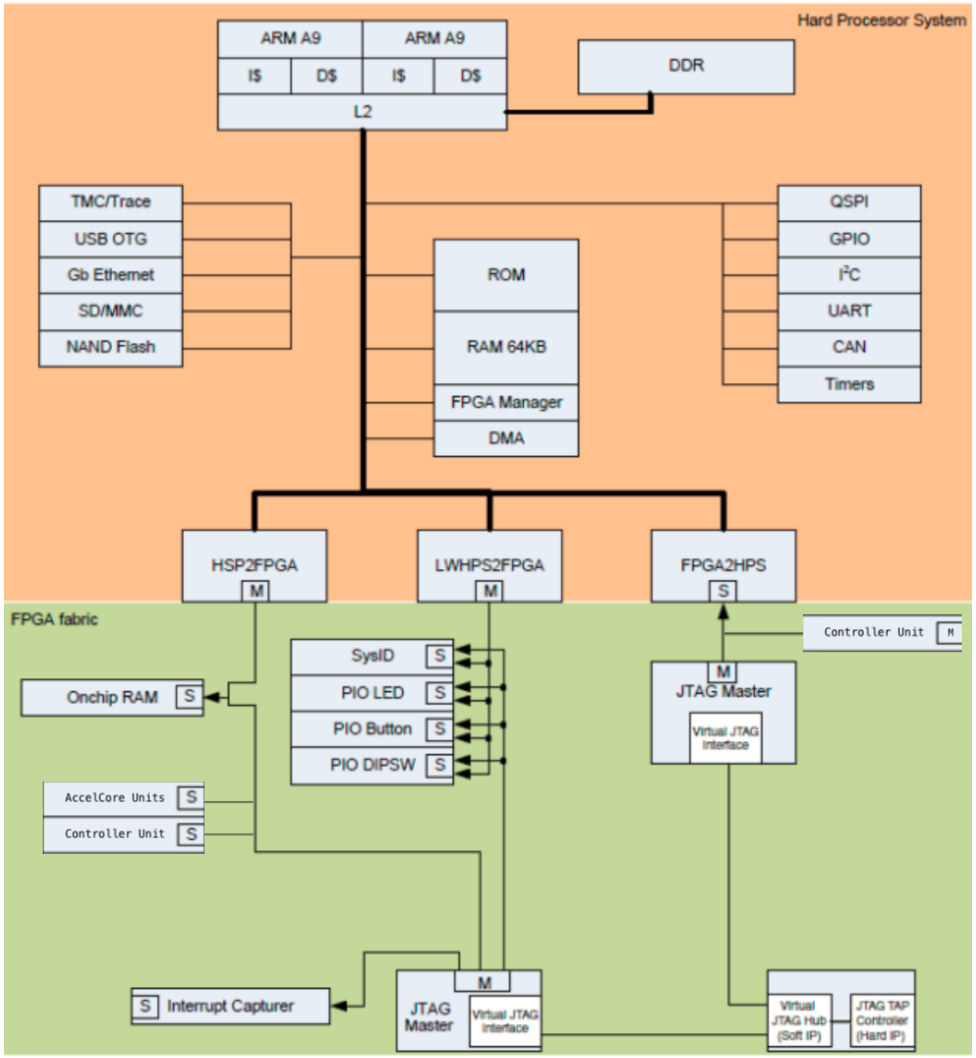

Figure 1: Overall architecture of SoCKit SoC [1]. This fugure illustrates different blocks of both the ARM Processor and the FPGA fabric.

In the following section, the main tasks required to successfully accelerate the forward propagation of ANNs will be explained. The next sections will then go into details of explaining how each section/task will be implemented.

2 Description of tasks

The tasks in this project can be broken down into three categories:

- Designing AccelCore units to accelerate MAC operations.
- Establish communication between the HPS and the FPGA to obtain the required data.
- Designing a controller to manage data and control the AccelCore units.

The first task will include the design of acceleration units and buffers which will accumulate data to feed into the next layer. Each bit of the buffer will hold the output of a single neuron. The second task is to establish the communication between the HPS and the FPGA. The processor will be used to compress the data and to send the information to block RAM within the FPGA. Finally, the third task will be the creation of the controller. The controller is divided into two sub-tasks. The first sub-task is to create a window slider to shift the window when a new convolution needs to be computed. The second sub-task is to create a more general controller which will control all the start and done signals of the AccelCores and buffers in the line of calculations.

3 Task 1: The AccelCore Units

3.1 AccelCore Overview

To accelerate the convolution task, a unit named *AccelCore* (short for Accelerator Core) will be used to compute the mathematical operations. The inputs of this unit will be fed by a controller which will inject weight values and the appropriate image input/output of the previous layer. There will be three AccelCores (one per layer). Each will have a different input size. The first AccelCore will have 256 input values (to represent the max window size), the second AccelCore will have 1024 inputs (the maximum amount of neurons in the first layer) and the last AccelCore will have 64 inputs (the maximum amount of neurons in the second layer). The general structure of the AccelCore does not change between layers.

3.2 Time Analysis

The original program was profiled on the NIOS II processor. However, the final solution will be running on the Cortex A-9 processor. The profiling results are therefore not indicative of the actual timing of the program. To measure the speed increase of the accelerator core, only the MAC operations will be considered. In other words, for this element of the proposed solution, we will not consider the time to fetch data from memory as this will be covered in the section on the data management controller and the HPS-FPGA communication module.

The operations to perform are multiplications and additions. The weights are multiplied by the image or the output of the previous layer and are then added to a buffer which will store the cumulative sum of all previous multiplications. Figure 2 illustrates this segment of the code.

```
for (int j=0; j<n_input; j++) {
    acc += *(cur_weight++) * source[j];
}
```

Figure 2: Multiply-accumulate operations found within the NNLayer::propagate() function.

The fastest potential implementation of these operations on the ARM Cortex A-9 processor would imply the use of the MLA instruction (multiply-accumulate). According to the datasheet [2], this operation requires 2 clock cycles. Assuming the window size of the convolution is 16x16 (the maximum size), it would require 256 MLA instructions to execute a single convolution. It would require approximately 512 clock cycles to compute a single convolution. The calculation is summarized in the following equation.

$$Total\_Cycles = 16 * 16\ instructions * 2\ \frac{cycle}{instruction} = 512\ cycles$$

A more complete analysis of the time consumed (including loading the weights and image inputs) will be explored in the final section of this report.

3.3 Proposed Accelerations

The proposed acceleration for the multiply-accumulate operations will be realized through the implementation of AccelCores. These units will be implemented entirely in hardware. Their inputs will be W bits representing weight values and I bits representing image or output layer values. The weights are ternary (2 bits) and the image/output layer values are binary. Therefore, there are 2 times more weight bits than image/output layer bits. The output of the AccelCore is a single bit representing the result of the entire convolution. Within each AccelCore will be N processing elements (PEs). These units will each be performing 8-bit MAC operations in parallel. The AccelCores will include a 2-stage pipeline to synchronize the flow of data. A `start` and `done` signal will be used to indicate when the process is over. Figure 3 illustrates the general architecture of the AccelCore.

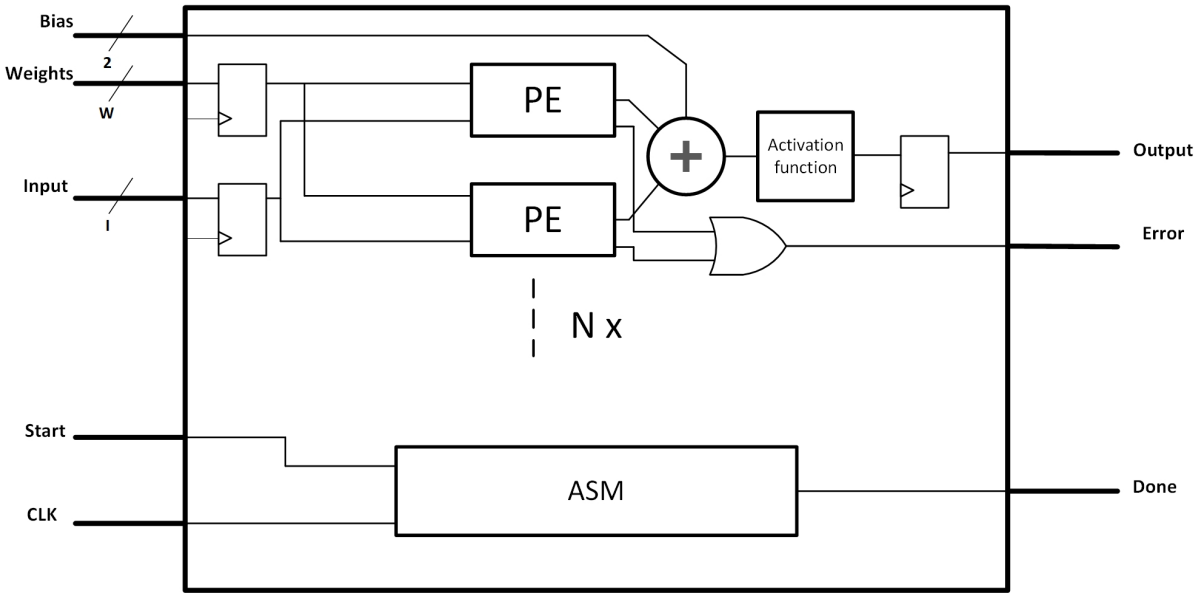

Figure 3: High-Level Representation of an AccelCore Unit.

As previously mentioned, the calculations within the AccelCores are essentially performed by the processing elements. These elements each take 16 bits of weight values and 8 bits of image/output layer values. They each perform 8 multiplication-accumulate operations. An error detection circuit is also included which evaluates if one of the weight values is inaccurate (the weight value can never be -2 (0b10) since they are limited to -1, 0 or 1). Figure 4 illustrates a single processing element.

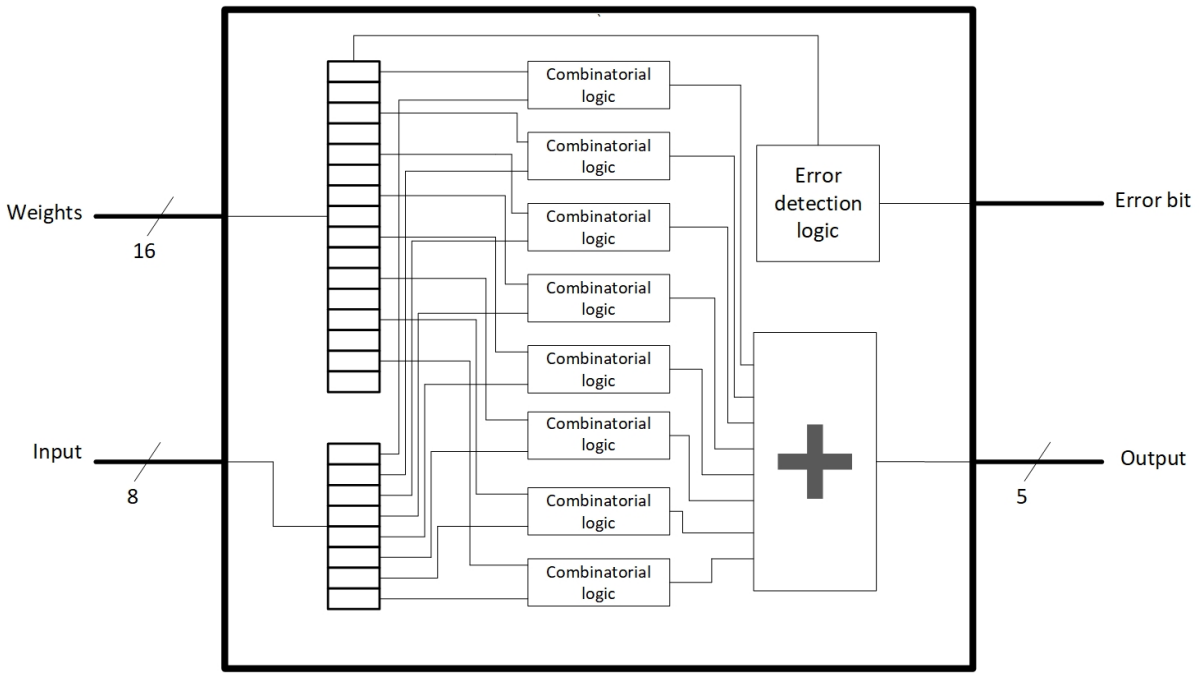

Figure 4: Representation of the Processing Element (PE).

The result of the combinatorial logic is determined by the product of a 2-bit value and a 1-bit value. Eight of these logical circuits are instantiated. The logic is illustrated in Figure 5.

W[1]	W[0]	I	Out[1]	Out[0]
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	-	-
1	1	0	0	0
1	1	1	1	1

Figure 5: Truth table of the combinatorial logic inside the PE.

The resulting logical circuit is illustrated in Figure 6.

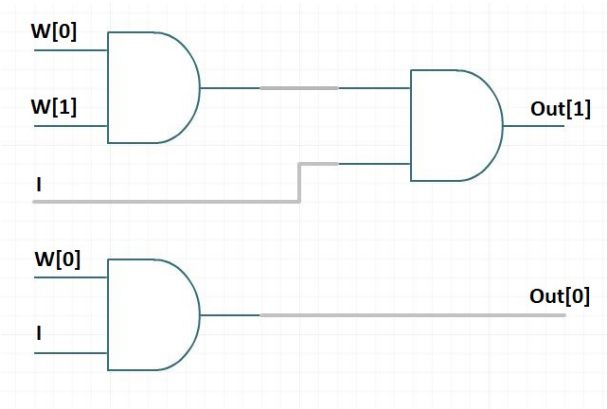

Figure 6: Combinatorial logic inside the PE.

Overall, the AccelCore has a 2-staged pipeline. In 2 clock cycles, it can perform I multiplications in parallel and sum the results together.

3.4 Architecture and ASM

The general architecture of the AccelCore was explained in the proposed acceleration section. The AccelCore has a 2-stage pipeline. A state machine is used to determine when the calculation is complete. Essentially, this state machine outputs a `done` value 2 clock cycles after the `start` bit is equal to 1. Its only purpose is to count 2 cycles. A state diagram representing the state machine is illustrated in Figure 7.

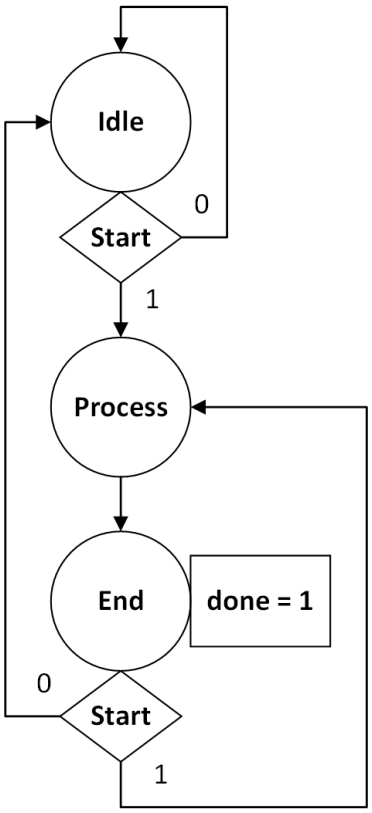

Figure 7: ASM of the AccelCore.

3.5 Test Plan

To make sure that the proposed solution does not alter the calculations in the original code, a PE class and AccelCore class were instantiated in the original software to perform the MAC operations.

```
BYTE AccelCore::apply(BYTE* input, BYTE* weight, BYTE inputSize, BYTE weightSize, BYTE bias)
{
    assert((inputSize%accelCoreSize_) == 0);
    assert((weightSize%accelCoreSize_) == 0);
    assert(inputSize == weightSize);

    BYTE sum = bias;
    for (int i = 1; i <= accelCoreSize_; i++)
    {
        BYTE inputSlice[8];
        BYTE weightSlice[8];
        memcpy(inputSlice, &input[8 * (i - 1)], sizeof(short) * 8);
        memcpy(weightSlice, &weight[8 * (i - 1)], sizeof(short) * 8);
        sum += peVector_[i-1]->apply(inputSlice, weightSlice);
    }
    return nonLinear(sum);
}
```

Figure 8: C++ Equivalent of the AccelCore.

```
BYTE PE::apply(BYTE* image, BYTE* weight)
{
    BYTE result = 0;

    for (int i = 0; i < peSize_; i++)
    {
        result += image[i] * weight[i];
    }
    return result;
}
```

Figure 9: C++ Equivalent of the Processing Element.

The old program and the modified program ran in parallel and the result of the AccelCore matched the original output for the entire application. This confirms that the general logic of the AccelCore does not alter the original implementation. To further test

the units once they are defined in a HDL, a testbench will be performed to evaluate if the expected outputs have been obtained. To accelerate development time, ModelSim will be used on a personal computer. One test bench will be made for a single PE unit. Once the functionally of the PE is confirmed, a second test bench will be created for the entire AccelCore unit.

3.6 Result Buffer for the AccelCore

A buffer is also used to keep in memory the outputs of the AccelCore. After every calculation, the buffer will shift all the values down the buffer. It is essentially a shift register controlled by an ASM. The buffer takes the result of the AccelCore and the `done` signal of the AccelCore as inputs. It outputs all the values contained within it as well as a `valid` signal to indicate when the buffer is full. The state machine is illustrated in Figure 10.

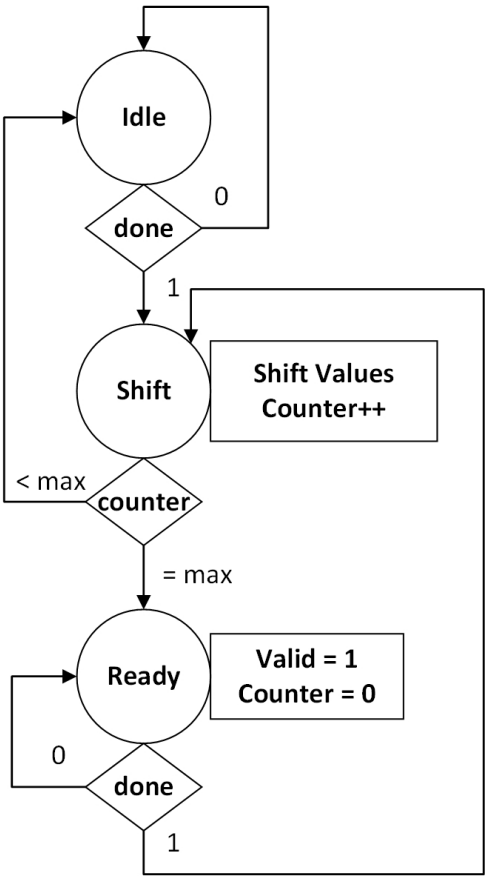

Figure 10: ASM of the Buffer used to store the outputs of the AccelCore.

The importance of the buffer will be more apparent when the complete architecture will be presented in the final section.

4 Task 2: The HPS-FPGA Communication Unit

4.1 Communication Unit Overview

The idea here is to modify where the input, weights and bias are stored. We want to use the HPS in order to manage the data and to communicate with the controller unit. To make our design more flexible, we decided to add a parameter memory. It will be read by the controller to get the image size, number of neurons in each layer and the window size. This gives our design the ability to cope with different types of neural networks and different sizes of inputs. The ARM core will receive inputs and weights from the software initialization and will process the data to then load it into the FPGA bRAM.

We imagine a memory structure which is summed up in the figure below :

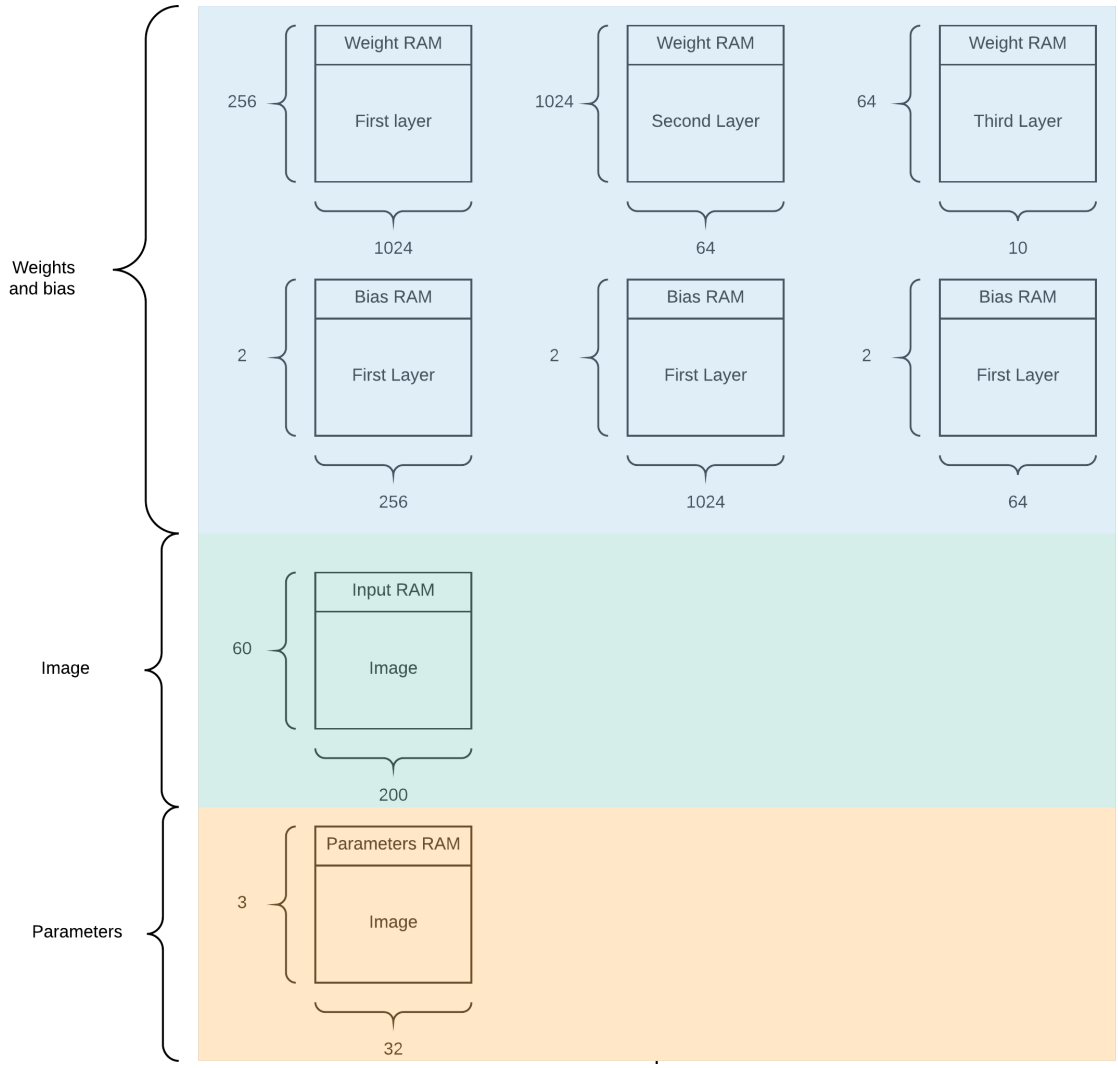

Figure 11: Memory architecture

4.2 SocKit memory constraint

The SocKit board counts around 5140 M10k memory blocks. The memory specifications of a single M10K block are summarized in Figure 12.

Operation Mode	M10K Memory Block Sizes
Single-port and ROM	1K × 8 1K × 10 2K × 4 2K × 5 4K × 2 8K × 1 256 × 32 256 × 40 512 × 16 512 × 20
Dual-port	Write × M / Read × N or W × xY / Read xZ M, N= 1, 2, 4, 8, 16, 32 and Y, Z= 5, 10, 20, 40
True dual-port	Port A xM / port B xN or Port A xY / port B xZ M, N = 1, 2, 4, 8, 16 and Y, Z = 5, 10, 20

Figure 12: Intel block definition

According to this data, we can't store 1024 bits of data into a single memory block. In order to store that much information, we need to use a number N_b of memory blocks all driven by the same controller as we can see below :

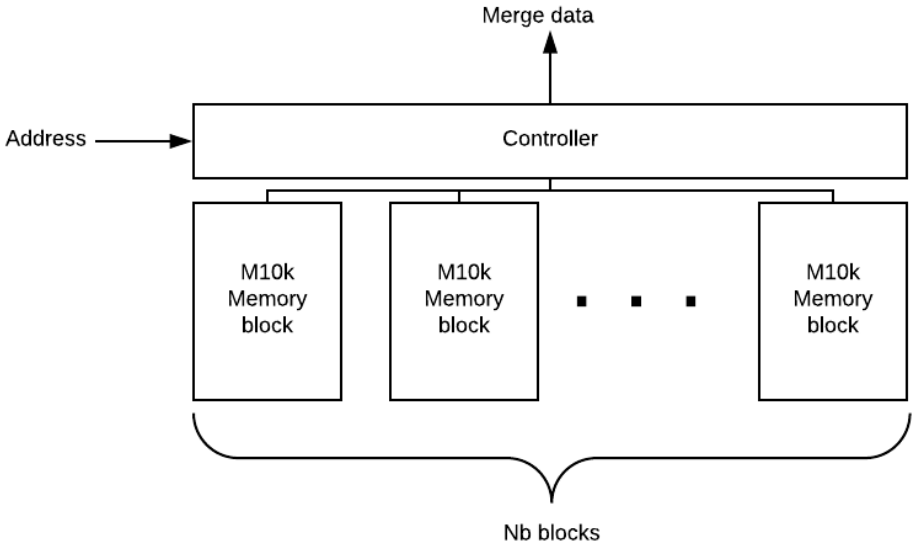

Figure 13: Data size solution

To calculate the number of N_b blocks needed we use the values found in Figure 12. To fit our design in memory blocks we need :

- For the first layer : 32 blocks of 256x32
- For the second layer : 4 blocks of 256x32
- For the third layer : 2 blocks of 256x32

4.3 Communication protocol between HPS and FPGA

The interconnection between the HPS and the FPGA is assured by the AXI protocol.

AXI stands for "Advanced eXtensible Interface" which comes from the AMBA bus family such as AHP. We can configure the AXI width with the Qsys tool (32, 64 and 128 bits). This bus will be connected to all our memory blocks as described in the memory architecture. The AXI bus implementation is done in the Qsys tool like in Figure 14.

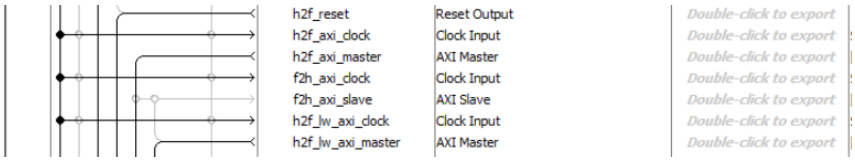

Figure 14: Axi bus in Qsys

4.4 Communication between memory and controller unit

We need to cope with the data and store it at the right place and at the right moment in order to let the controller get fresh data. To be synchronized with the controller unit, we need different signals :

- `s_new_data` : is up for one cycle when new data is sent by the software application.
- `s_load_done_data` : is up for one cycle when new data is stored in the FPGA RAM and available for the controller.

With these signals, we can push a new input in our design and the controller can handle it properly. This is the only synchronization with the software. It's our choice to limit the dependence towards the software application in this project.

4.5 Test Plan

Like every part of this project, it will be cross-tested which means it's not the person who has written the code who is testing the module.

This part is focused on communication between the HPS and the FPGA. We need to test each memory RAM to be sure that the data is stored and that the memory controller is working correctly. We can split the test plan in 3 parts :

- Test the read operation from the software
- Attempt to write to the controller
- Test the signals between the memory and the controller

In a test-bench, we will send inputs and weights to the FPGA bRAM and it will send us data back to compare the read and write process.

5 Task 3: Controller Unit

5.1 Controller Overview

The controller unit is responsible to provide the accelerator with the proper inputs and weights. Our proposed controller consists of the following modules:

- Window Slider: Selects a window from the input image and serializes the window to be used for the accelerator.
- Controller: Based on the status of the accelerator, it will assign control signals to the weight RAMs of each layer.

5.2 Window Slider

As mentioned earlier, this module is responsible to slide and select a window from the input image. Currently, this task is done through software. Figure 15 illustrates the C++ code that serializes the window for the neural network.

```
115  Image * Image::apply_NN(NN * network, int size, int pos) {
116      BYTE* source = new BYTE[size*size];
117      Image * result = new Image(length-size+1,height-size+1); //-size (pour ter
118
119      for (int y=0; y<=height-size; y++) {
120          printf("Processing line %i\r\n",y);
121          for (int x=0; x<=length-size; x++) {
122              /* Appliquer le reseau sur un sous-bloc de l'image */
123              for (int j=0; j<size; j++) {
124                  for (int i=0; i<size; i++) {
125                      source[j*size + i] = (*source_pixel(x + i, y + j)) / 255;
126                  }
127              }
128              network->propagate(source);  // Cette fonction fait les additions
129
130              /* Stocker les bons/meilleurs matchs */
131              unsigned char pixel;
132              pixel = 255*(network->layer[network->n_layer-1].value[pos]);
133              *(result->source_pixel(x,y)) = pixel;
134          }
135      }
136      return result;
137  }
```

Figure 15: C++ code snippet in the original program that shows how a window from the image was originally selected and serialized.

As it can be seen, there are four nested loops, the outer two are responsible to go over width and height of the image (to select a window), the inner two loops are responsible of serializing the image. It is very clear that this process is computationally heavy. Therefore, we decided to implement this computation in hardware. This way, the input image will be loaded once to the FPGA and never again.

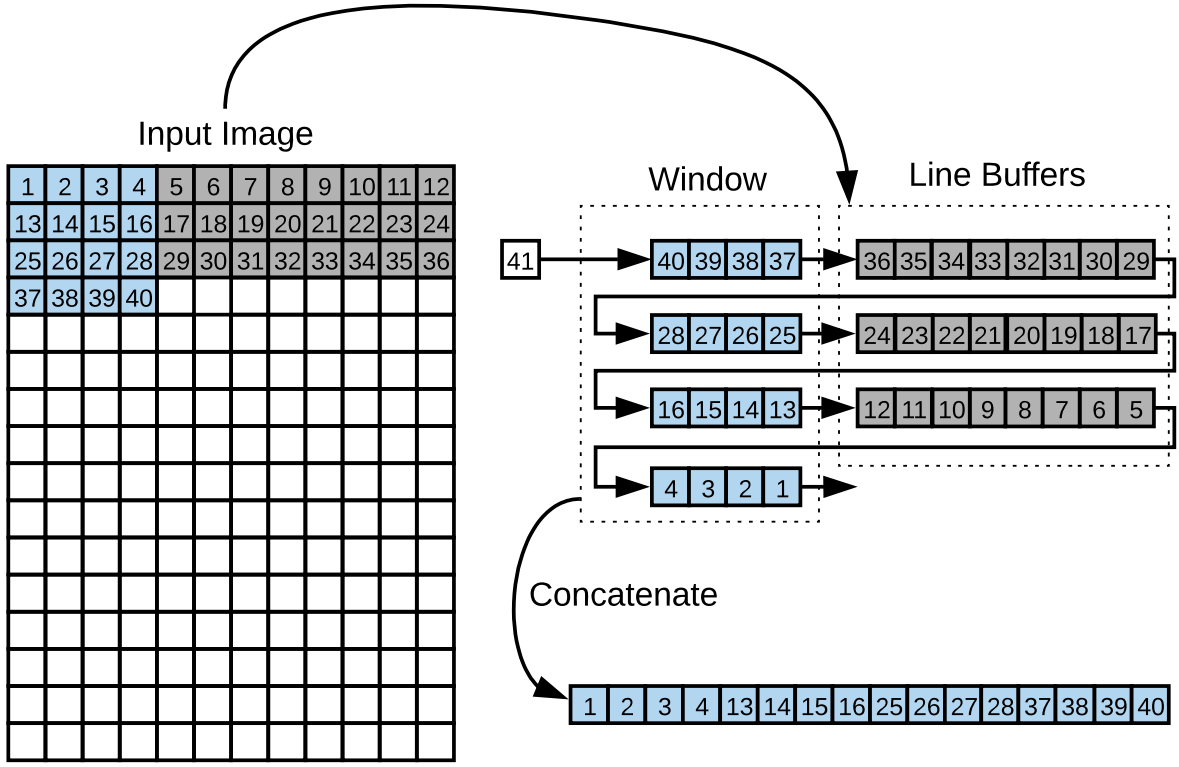

Figure 16: Window Slider module. This module is responsible to select windows from the input image and then concatenate the result to create an array which is ready to be fed to the neural net.

Figure 16 shows an overview of the Window Slider module. This module is responsible of selecting windows from the input image and then concatenating the result to create an array which is ready to be fed to the neural net. As it can be seen, it is a pipe-lined architecture. Assuming the input image is stored line by line in the memory, on every positive edge of `slide`, one pixel is read from the memory and is fed to the pipe-line. The pipe-line consists of `WINDOW_LEN-1` line buffers.A line buffer is a FIFO that stores the image pixels that are not in the current window but that will be used later. Each line buffer has depth of `IMAGE_WIDTH - WINDOW_LEN`. When the pipe-line is full (meaning all line buffers and the Window buffers are fed), the Window Slider can output a new window on every clock cycle. However, the AccelCore cannot compute all this data in one clock cycle. This means that the Window Slider will generate a new window only when it is requested. This request must be sent by a pulse of one clock cycle through the `slide` signal.

One the other hand, not all windows are valid windows. Figure 17 clearly illustrates this phenomenon.

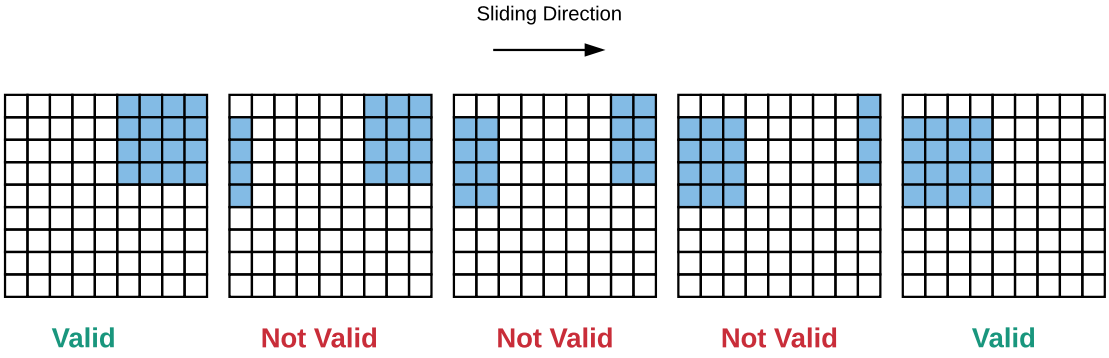

Figure 17: This image shows why not all windows are valid.

Even though we can output a new window every clock cycle, not all of the generated windows are valid. Also, the accelerator cannot consume all the data in a single clock cycle. To improve the management of the Window Slider module, we came up with an ASM so that the `slide` signal is called when needed and the `valid` signal is generated properly.

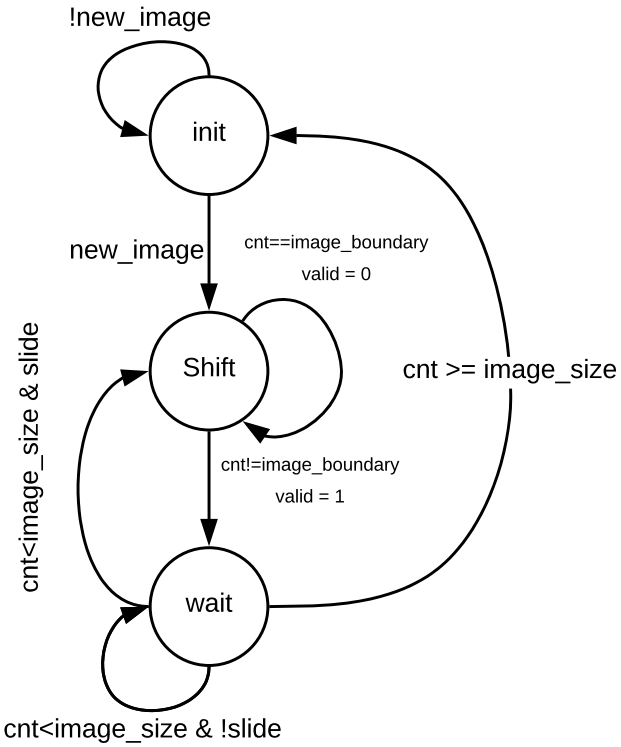

Figure 18: Window Slider ASM. The Window Slider ASM generates a valid signal only when the generated window is within the image boundary. Also, it only generates a valid window when it is requested by the `slide` signal.

Figure 18 illustrates the proposed ASM for the Window Slider module. The Window Slider ASM generates a valid signal only when the generated window is within the image

boundary. Also, it only generates a valid window when it is requested by the `slide` signal.

5.3 Controller

As it was mentioned earlier, we need a controller module to generate control signals to provide correct weights to the accelerators of each layer. The controller also sends control signals to the AccelCores and the Window Slider so that these modules know when to start executing their respective operations.

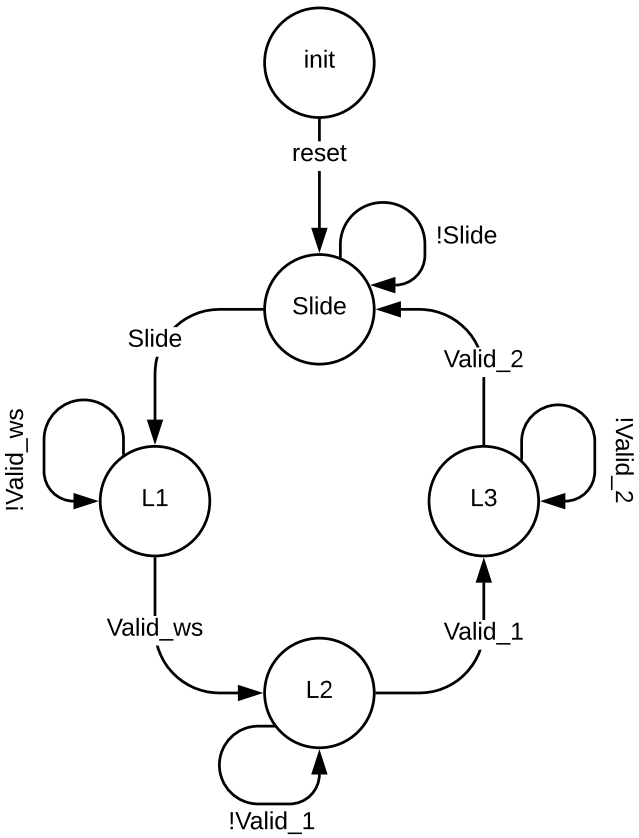

Figure 19: Controller ASM. This figure illustrates the ASM for the proposed controller. In every layer (illustrated by L_x), the controller waits for a valid signal. This valid signal comes from the accelerator cores indicating that the computation of all neurons has been done. Based on the number of the current layer, this will trigger the controller to calculate the next address for the layer's weight RAM.

Figure 19 illustrates the controller ASM. In every layer (illustrated by L_x), the controller waits for a valid signal. This valid signal comes from the accelerator cores indicating that the computation of all neurons is complete. Based on the number of the current layer, this will trigger the controller to calculate the next address for the layer's weight RAM. When we are in the last layer (L_3), the controller activates the slide signal of the Window

Slider module. This signal will trigger the Window Slider to generate a valid window so that it can be consumed by the accelerator. All the signals required to communicate with the input RAM are handled by the controller. The controller will feed pixels to the Window Slider until a correct window is generated.

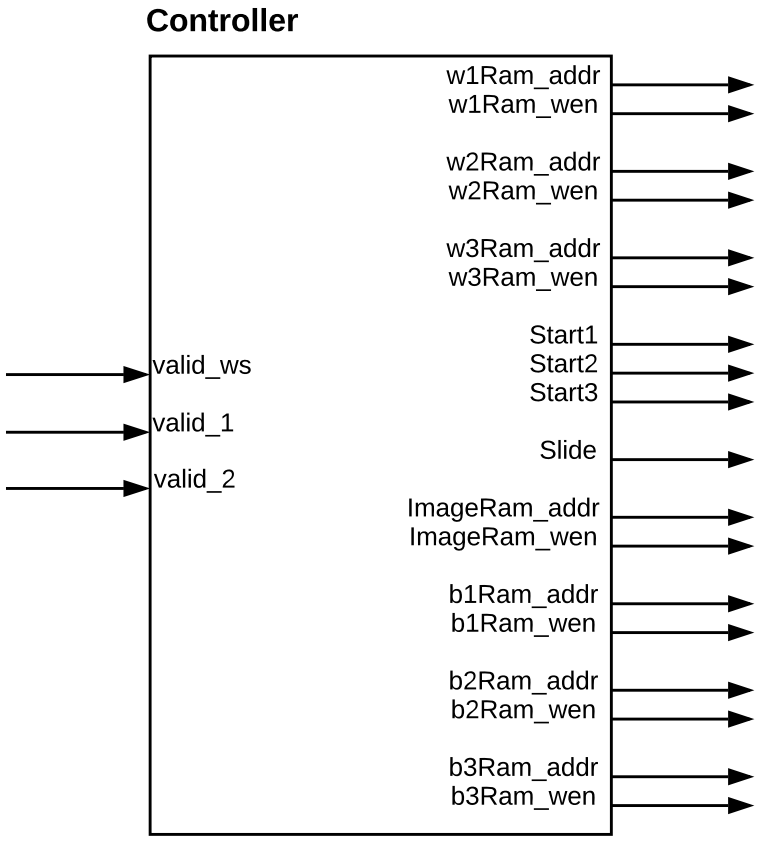

Figure 20: Controller block diagram. This figure shows the signals that are generated and consumed by the controller.

Figure 20 shows the signals for the controller unit. As we discussed in this section, the controller unit manages the data transfer and control signals for the accelerators. It also provides the accelerator with a new window by controlling the Window Slider.

5.4 Test Plan

As a good verification practice and like every part of this project, this module will be cross-tested which means it's not the person who has written the code who is testing the module. To verify the correct functionality of this module, we will be writing two testbenches. One for the Controller unit and one for the Window Slider. The following are the test scenarios that we will test for the controller unit:

- Test if the Window Slider is outputting the correct Window.
- Test if the Window Slider can detect Image boundaries and if the valid signal is generated correctly.
- Test the controller ASM by providing test signals to it's inputs.

6 Test Procedure

Different test methodologies are available that can be used to functionally verify the correctness of our design. As an example, a popular verification methodology that is widely being employed by the industry is UVM or Universal Verification Methodology. UVM is an Accellera standard with support from multiple vendors: Aldec, Cadence, Mentor Graphics, and Synopsys. Figure 21 shows a typical testbench architecture in UVM.

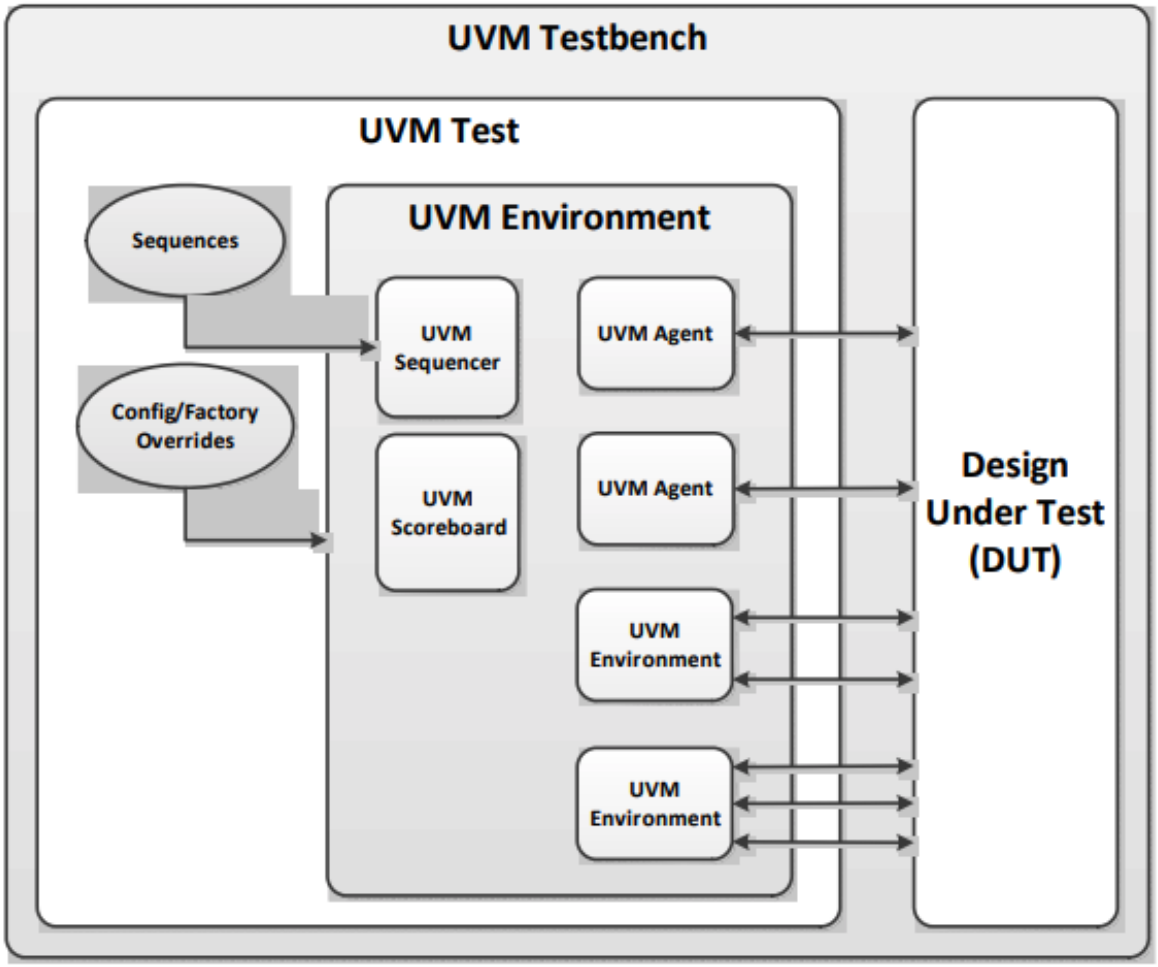

Figure 21: Typical UVM Testbench Architecture [source][5].

While using UVM would give us a lot of leverage for code re-usability, making the environment setup properly to be compatible with the Altera tool chain will need a lot of time. On the other hand, we can re-use some of the UVM concepts when designing our test benches. A simple driver monitor environment is essentially a building block of any UVM testbench. Hence, we will be using this simple but effective testbench architecture in our verification process. Figure 22 illustrates this simple architecture.

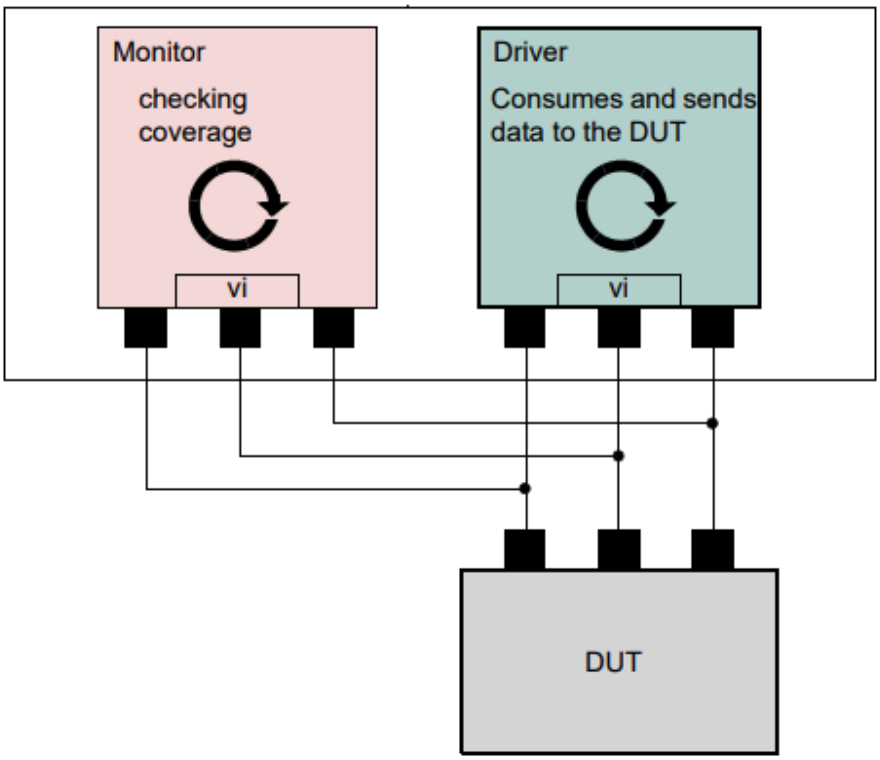

Figure 22: A simple Test Bench architecture taken from commonly used UVM test environments [5].

7 Overall Architecture and Acceleration

The complete architecture of our system is illustrated in Figure 23.

The bits are fed into the Window Slider which fills up until it has enough values to fill an entire feature map. For a 60x200 image, the slider needs to contain at least 916 values to perform the first convolution (15 rows of 60 + 16). This takes 916 clock cycles. Since this is only performed once, this time will be ignored. Every AccelCore requires 2 clock cycles to execute its calculation. The inputs and the weight values are continuously fed into the AccelCore every 2 cycles. This means that the AccelCore is always calculating something. In the worst case scenario, there are 1024 neurons in the first layer (2048 clock cycles), 64 neurons in the second layer (128 clock cycles) and 10 neurons in the last layer (20 clock cycles). Every time something is calculated it is fed directly into the bit buffers meaning that no cycles are lost to retrieve the information calculated by the previous layer. Calculating a single pixel for all 10 images takes 2048 + 128 + 20 = 2196 clock cycles. To cover the entire image, the Window Slider needs to cover 45 x 185 = 8325 different positions. In total, this amounts to approximately 8325 x 2196 = 18.2 million clock cycles. For comparisons's sake, if this code were to run on the NIOS at 50 MHz, this would take approximately 364 ms.

The original code was profiled with 90 neurons in all the layers. The total time to execute the computation was approximately 40 seconds. The bottleneck of the calculation is the fetching of weights and inputs and the MAC operations. More neurons therefore

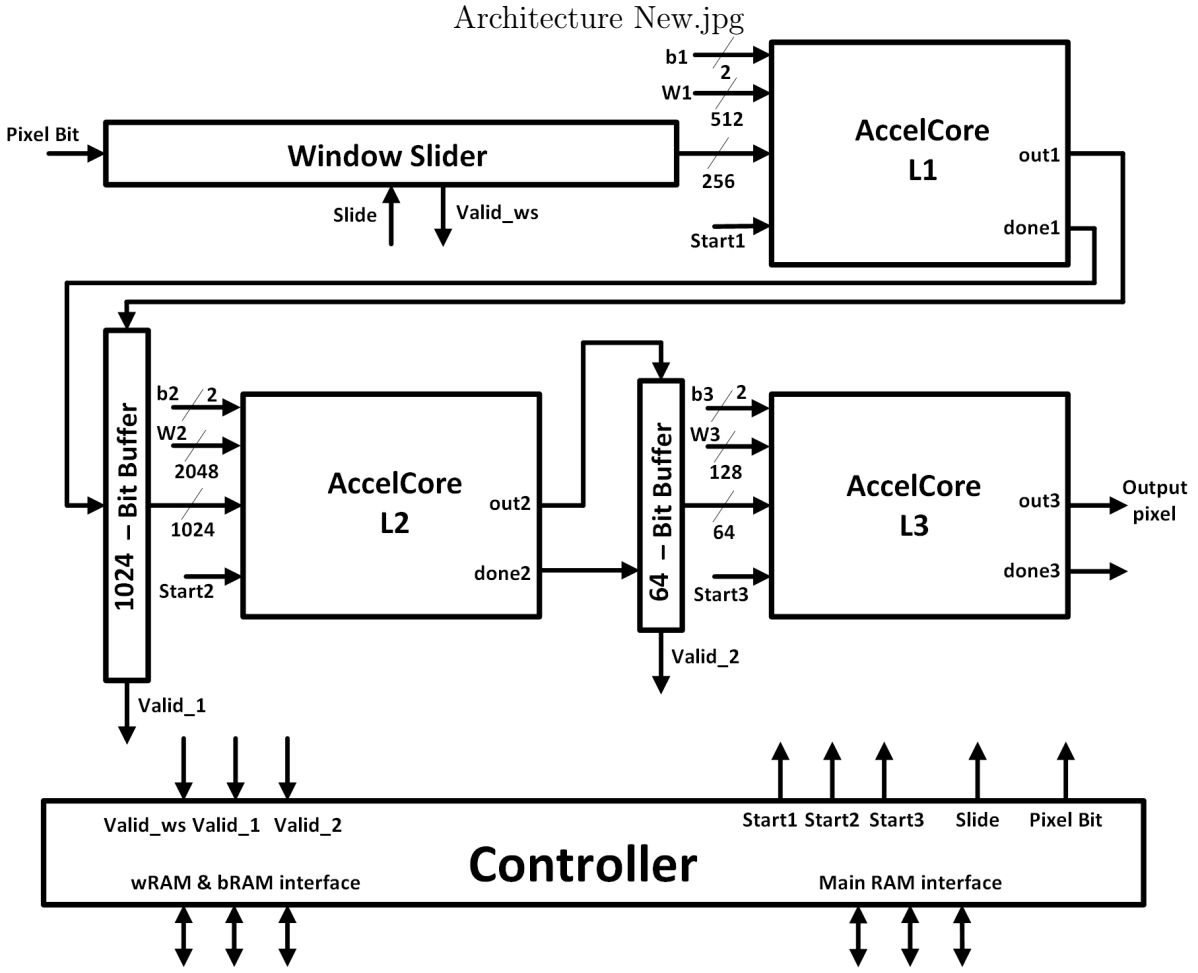

Figure 23: Full architecture of the proposed acceleration.

means longer calculation times. Assuming a linear dependancy, 1172 neurons (the max amount of potential neurons) would take approximately 520 seconds. In the best case scenario, we can therefore expect a 1400x acceleration (520 / 0.364).

In terms of resources, the PEs each consume 8 x 3 logic gates for the multiplications and approximately 8 x 25 gates for the adder. Rounding up, this gives about 250 gates per PE. In total, 168 PEs will be used for a total of 42000 logic gates. The rest of the circuit (controller, slider, adder inside AccelCore, etc.) will consume a very small amount of gates compared to the total PEs. Our solution will therefore consume a bit more than 42000 gates. This should not be a problem for a modern day FPGA.

7.1 5 min vs 24 h case

In the 24 hour case, we can set the hardware parameters to fit the exact size of the neural network. In the 5 minute case, the windows and the image will be zero-padded to fit the size of the network. This will slow down the execution (when comparing to the 5 minute case) since it takes time to load values into memory.

8 Computation Dependency

Although a pipe-lined design would have given us a better throughput, for simplicity and as a first draft of our accelerator, we are proposing to put all the accelerators back to back with a blocking mechanism. Meaning, every clock cycle, only one accelerator of a particular layer is working.

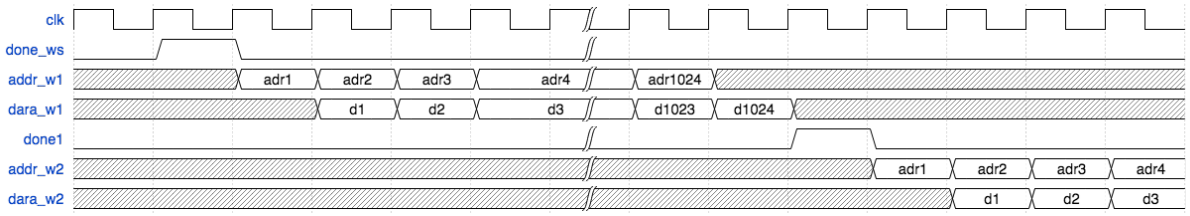

Figure 24: This figure shows how our locking mechanism works. In this example, there are two layers in which each of them are waiting for the previous layers computation to be done (illustrated by `done_x` signal) so that it can start computation.

Figure 24 illustrates our locking mechanism. This figure shows an example were we have two layers. Layer one is waiting for Window Slider to send `done_ws` signal. This is to indicate that the data in the output of Window Slider is valid and can be consumed by the next layer. On the other hand, the controller is waiting for this signal to start providing the AccelCore of the first layer with the correct wights. As it can be seen, the first weight is provided to the AccelCore two clock cycles after the positive edge of `done_ws` . The calculation of the first layer continues until we processed all the weights. To indicate that layer one is done processing, it has to generate a done pulse (`done1`) which is high for one clock cycle. This will trigger layer two to start processing the data provided by the first layer.

9 Conclusion

This project is focused on optimizing a neural network by using a SoC and implemented a hardware/software acceleration. We started from the software application code to understand how it works. We proceeded with small software improvements and profiling the application. This gave us hints on where to work to optimize the execution.

Our proposed solution is to implement 3 different types of units : an AccelCore unit, a HPS-FPGA Communication unit and a general Controller. Each unit is design to reduce the number of cycle to be execute in order to minimize the global time needed for the execution.

To conclude, hardware module are essential to optimize an application. The SoCKit platform allows us to use a hybrid solution where all hot nodes are hardware modules. This solution is tailored to this application but since we haven't forgotten to take care of different sizes of inputs and different neural networks, it is also flexible and dynamic.

10 Bibliography

References

[1] SoCKit HW Lab Instructions, Version 15.0: 2.1 System Architecture

[2] Cortex-A9 Technical Reference Manual, Rev r3p0: Multiplication instruction cycle timings

[3] Quartus II Help v13.1: M10K memory block Definition

[4] Using Qsys with DE1-SoC Cornell ece5760 Qsys Overview: HPS-FPGA bridges in Altera SoCs

[5] http://www.accellera.org/images/downloads/standards/uvm/uvm$_u sers_g uide_1$.2.*pdf*