Faculty of Engineering, Cairo University

Computer Engineering Department

VLSI Course Project

DCNN Accelerator - Phase 1 Report

Hanin Hisham
Dalia Mohamed
Mary Nader
Hagar Haytham

I/O Block Team

Ahmed Khaled
Maryam Shalaby
Zeinab Rabie
Omnia Zakaria

Control Block Team

Ahmed Essam
Kareem Emad
Roba Gamal
Sayed Kotb

Computation Block Team

Contents

1 System Overview

DCNN Accelerator is a system for accelerating inference in a convolutional neural network (CNN). It supports convolution, pooling, and fully-connected layers. Figure 1 shows the main blocks in the designed DCNN Accelerator.
There are three main components: The *I/O block* loads the memory with the data processed from the CPU and performs needed steps for data to be readily available for operation. On completion of loading of the necessary portion of data, the IO block notifies the *Controller* to start its process. The Controller loads the filter and image and decides according to layer type the necessary sequence of operations and programs the computation block accordingly. The computation block makes the necessary computations and the notifies the controller which writes the data back into the memory and proceeds to the next operation (continuing convolution, convolving with a new filter, or entering a new layer).
The IO block is interfaced directly with the CPU and writes its data to the RAM, which the controller takes the data from. Therefore the interface between the controller and the I/O blocks are mostly in the memory and the ready signals. The interface between the controller and the computation block is more involved and direct. The computation block has a small host of buffers for the computation that the controller transfers first before starting the operation. The output data and ready signals are then sent back to the controller when done.

2 I/O Block

2.1 Overview

The I/O block lies at the interface between the CPU and the Accelerator. The data is sent via a script from the CPU to the accelerator and is decompressed and brought onto the on-chip memory of the accelerator.

2.2 Main Sequence

The main sequence of the I/O block's operation is given by Figure 2. The main steps are:

1. The CPU sends the clock and reset signal to the IO controller, and then starts sending the data packet by packet.

2. The I/O interface sends signals the decompression circuit to start, which starts decompressing the CNN's retrieved image info (which is run-length encoded) and saves it onto the on-chip RAM.

3. This process continues until the image is completely sent. Then is repeated with the image information (layers, filters, etc.) until the data is written and a handshake is exchanged with the CPU.

2.3 Design

2.3.1 Interface

The design for the interface between the CPU I/O interface bus and the compression circuit is given in Figure 3. The I/O register holds the data for compression.

2.3.2 Controller

The design of the controller is given in Figure 4.

2.3.3 Decompression

The design of the decompression circuit is given in Figure 5.

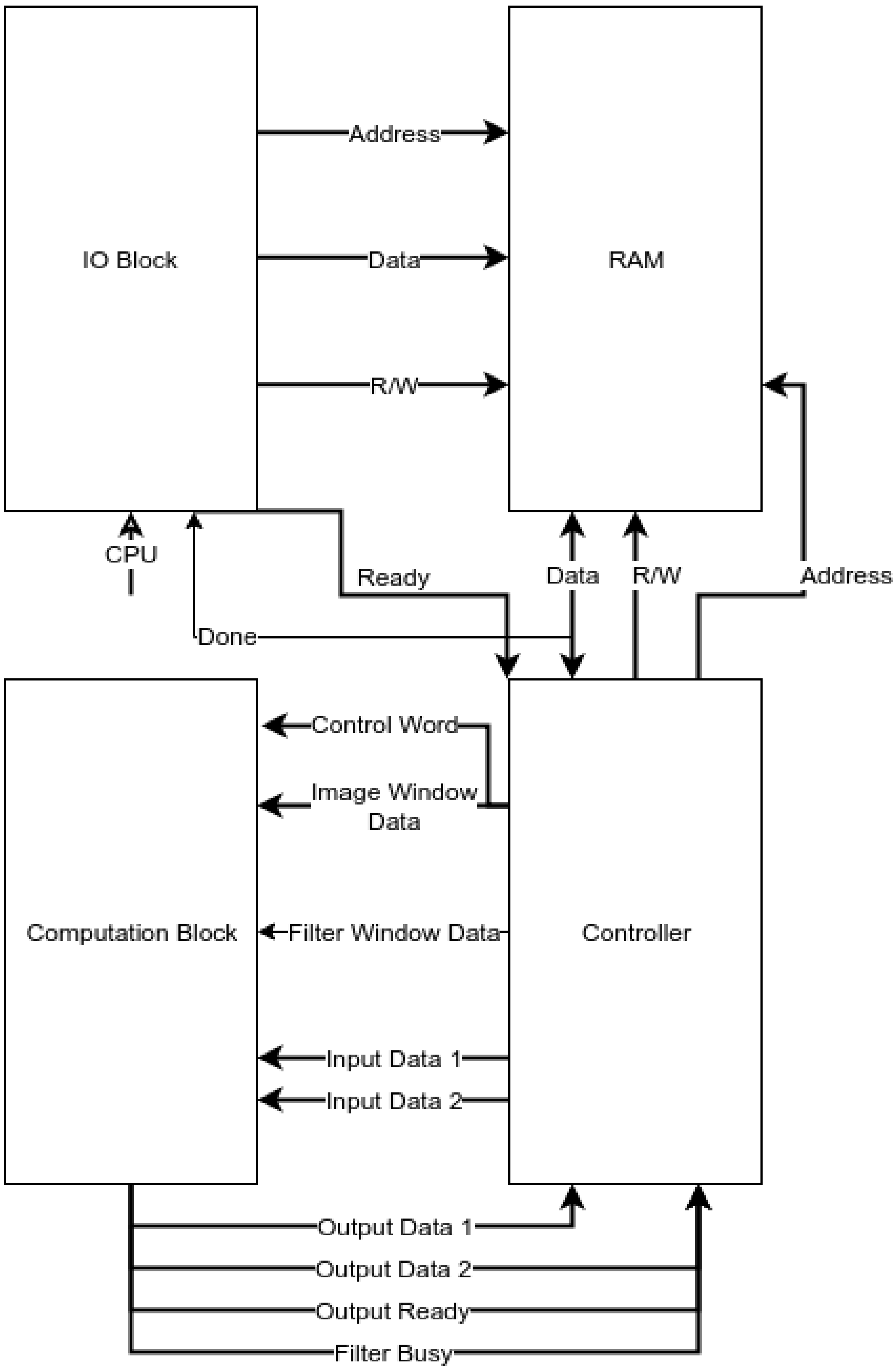

Figure 1: System main components and Interface

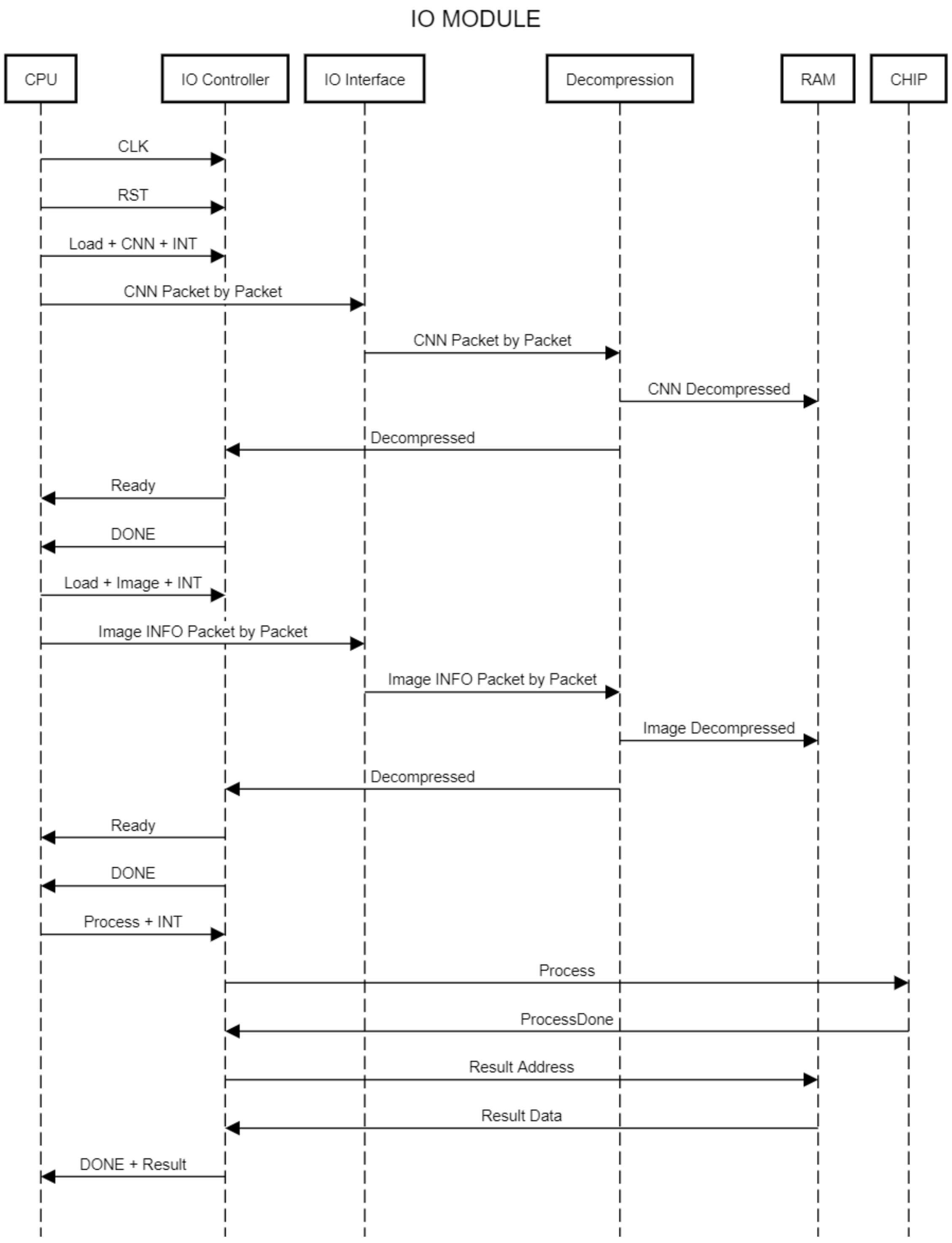

Figure 2: I/O Block Sequence Diagram

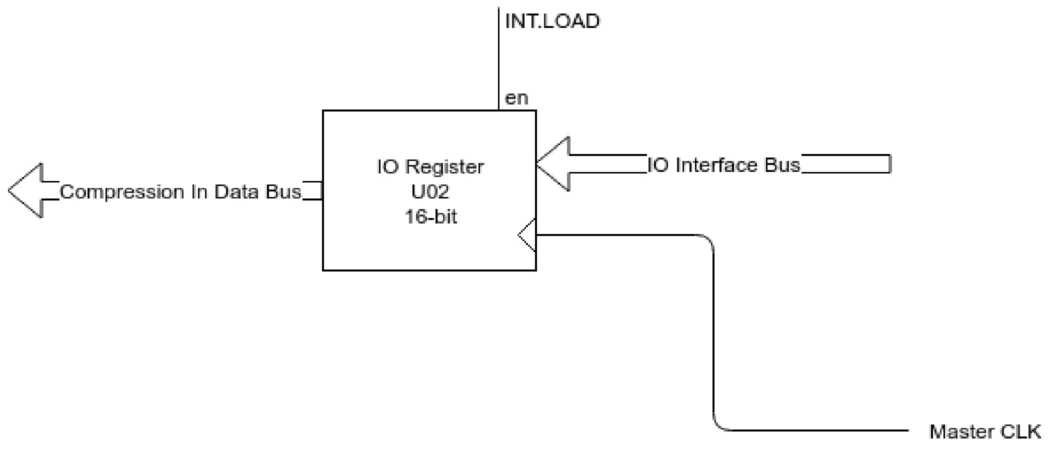

Figure 3: I/O Block Interface

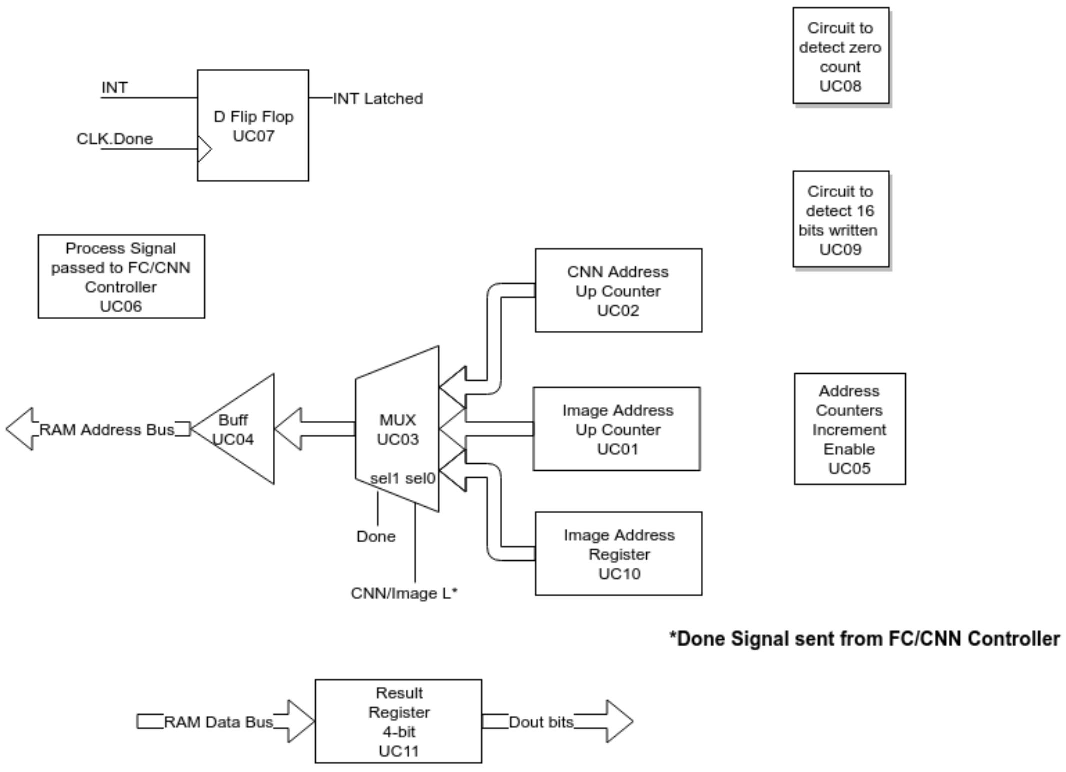
Circuit to detect zero count UC08
INT
CLK.Done
D Flip Flop UC07
INT Latched
Circuit to detect 16 bits written UC09
Process Signal passed to FC/CNN Controller UC06
CNN Address Up Counter UC02
RAM Address Bus
Buff UC04
MUX UC03
sel1 sel0
Image Address Up Counter UC01
Address Counters Increment Enable UC05
Done
Image Address Register UC10
CNN/Image L*
*Done Signal sent from FC/CNN Controller
RAM Data Bus
Result Register 4-bit UC11
Dout bits

Figure 4: I/O Controller

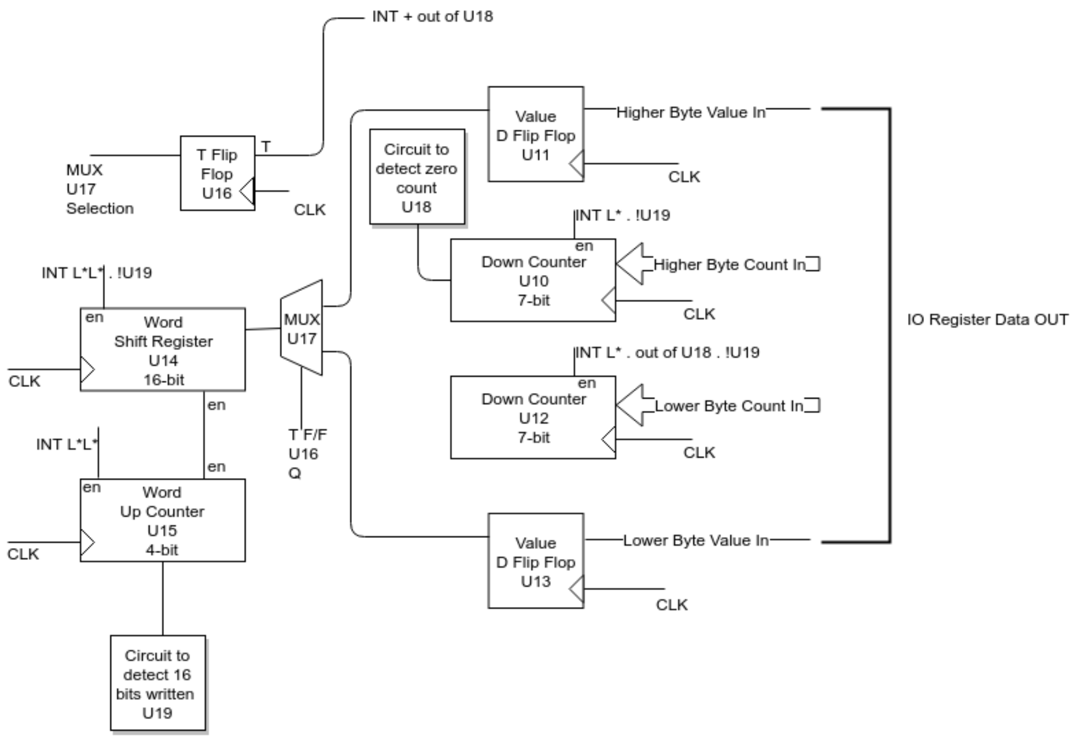

Figure 5: Decompression

2.3.4 Assumptions

1. Selection line **S0** of **MUX01** is **DONE** signal sent by FC.
2. *IO Register* takes one clock cycle to transfer the data onto the data output bus.
3. The stop signal is sent to the CPU by activating **DONE** when sending "1111" on **DOUT**.
4. The **U16** flip flop is added to switch between counters, first INT selects counter1, and when the count of counter1 is zero selects counter2.
5. The word counter retains the full signal only for one cycle, then starts over.

2.3.5 Handshaking

1. When starting transmission an **INT** signal is sent along with setting **LOAD** and the type of data to be sent (**CNN/Image**).
2. The **LOAD** signal is kept high until the last packet is sent.
3. Afterwards the **LOAD** signal is set to zero.

2.3.6 Signals

1. **S0** is a signal sent when processing is completed.
2. **C0** is a signal sent by the I/O controller to activate the decoder.

2.4 RAM

Access to the on-board RAM operates on the basis of priority. Taking in the address / data from both the I/O and the computation units, it only lets the computation unit write if not required by the I/O block. When the I/O block is done writing to the RAM, a signal is sent to the control unit to start computation.

3 Computation Block

The computation block is the brawn of the design, as it's responsible for performing the necessary computations for convolution and pooling. It has two main parameters to configure:

1. operation type (pooling or convolution),
2. and kernel size (3×3 or 5×5).

In the case of convolution, it convolves two windows, one containing the filter, and the other a piece of the image. When pooling is performed, however, the filter window is irrelevant.

If a kernel size of 5×5 is used, both of the windows are considered to be of size 5×5, and both convolution and pooling result in one output. But if a kernel size of 3×3 is used however, the image window is considered to be of size 3×4, and hence convolution and pooling result in two outputs.

3.1 Overview

3.1.1 Interface

In this section we take a look at the inputs and outputs of the computation block. A schematic of the interface is depicted in Figure 6.

1. Image Data (5×16 bit input): A column to be inserted into the image window. The mechanism by which the image window works will be explained in its own section.
2. Filter Data (16 bit input): A single pixel to be inserted into the filter window. The mechanism by which the image window works will be explained in its own section.
3. Output 1 initial value (16 bit input): The first output resulting from pooling or convolution is to be added to this initial value.
4. Output 2 initial value (16 bit input): The second ouput resulting from pooling or convolution is to be added to this initial value.
5. Control Word (2 bits input): Specifies the operation type, and kernel size.
6. Start (1 bit input): An asynchronous signal to latch the initial values, and start the operation.
7. Buffer busy (1 bit output): Specifies whether the image and filter data can be inserted.
8. Output 1 (16 bit output): The first output resulting from convolution/pooling.
9. Output 2 (16 bit output): The second output resulting from convolution/pooling. This is irrelevant when using a window size of 5×5.
10. Busy (1 bit output): Indicates whether the computation block is busy or not. This goes to 0 when it has finished its operation, and goes to 1 when the start signal is sent.

3.1.2 Operation

To program the computation block, both of the windows need to be filled first. This can happen while the buffer busy signal is cleared. The filter window is a queue of 25 words, each word is of size 16 bit, so it needs at least 25 clock cycles to be completely filled. Similarly, the image window is a queue of 25 words, but it takes 5 inputs at a time, so it can be filled in 5 clock cycles. After filling the cache, the control word, and the start signal need to be sent. The computation block will respond with a busy signal, and after a few clock cycles it will clear the busy signal, indicating that the output is ready. The buffer data, and initial values will persist, so no need to update them each time. A sequence diagram of the operation is depicted in Figure 7.

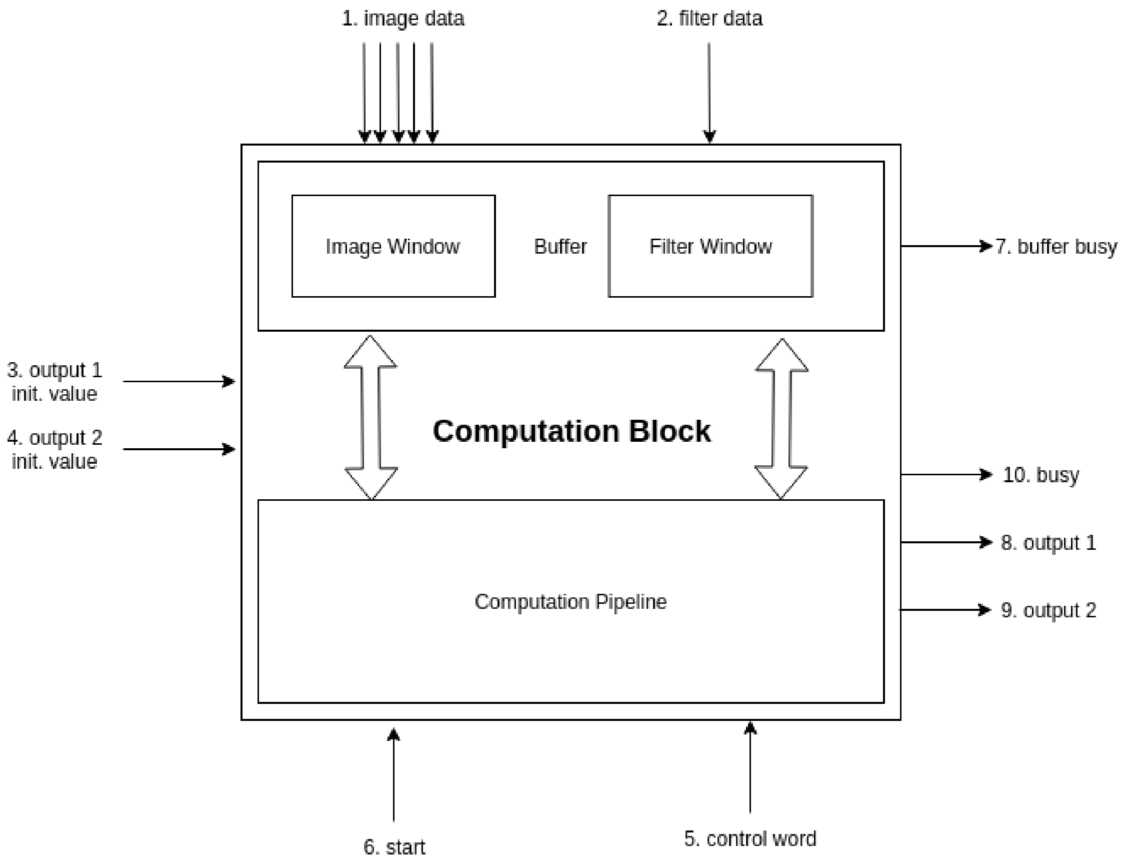

Figure 6: Computation block interface

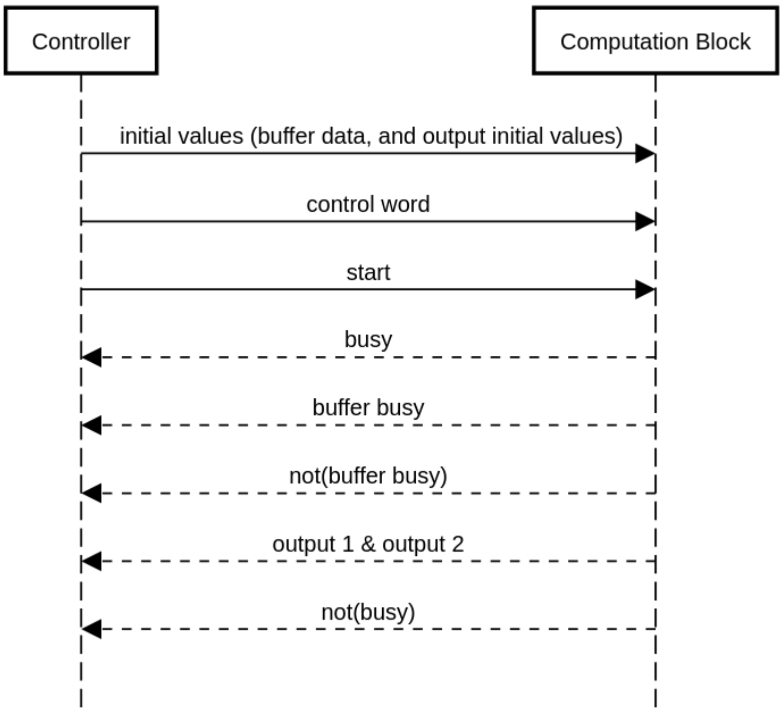

Figure 7: Computation block sequence diagram

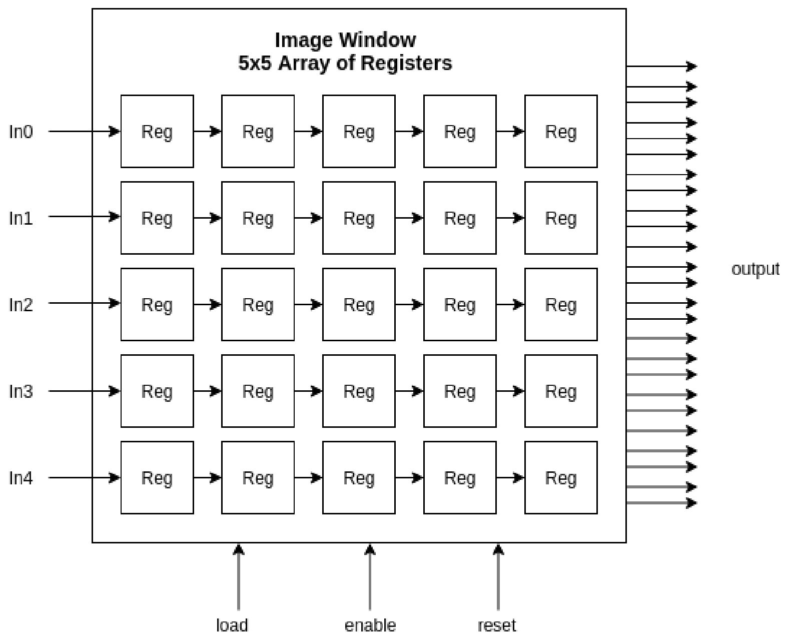

Figure 8: Image Window

3.2 The Buffer

3.2.1 The Image Window

The image window is an array of queues of size 5, where each queue is of size 5, so 25 registers in total. It acts as a horizontally sliding window over the convolved image, since when an input is accepted (by setting the load signal), each column is moved to the one following, and the last column is dejected. The structure of the image window is shown in Figure 8.

3.2.2 The Filter Window

A queue of size 25. No need for a sliding mechanism, and hence it accepts one input at a time.

3.3 Computation Pipeline

The computation process is divided into 5 steps, some can be skipped depending on the type of the operation, and the kernel size. These are:

1. multiplication,
2. addition,
3. merging,
4. normalization,
5. and ReLU.

However, before the input goes to the multiplication phase, it is multiplexed according to the filter size. A schematic of the overall pipeline can be seen in Figure 9.

3.3.1 The Multiplication Step

In this step, 25 multipliers are used in parallel to multiply each value in the image window with the one corresponding in the filter window. So this block takes 50 values as inputs (image$[i, j]$ and filter$[i, j]$ for each $i, j \in [1, 5]$), and outputs 25 products. Notice however that if the kernel size is 5×5 then the result of the convolution is

$$\sum_{i=1}^{5}\sum_{j=1}^{5} \text{image}[i, j] \cdot \text{filter}[i, j]$$

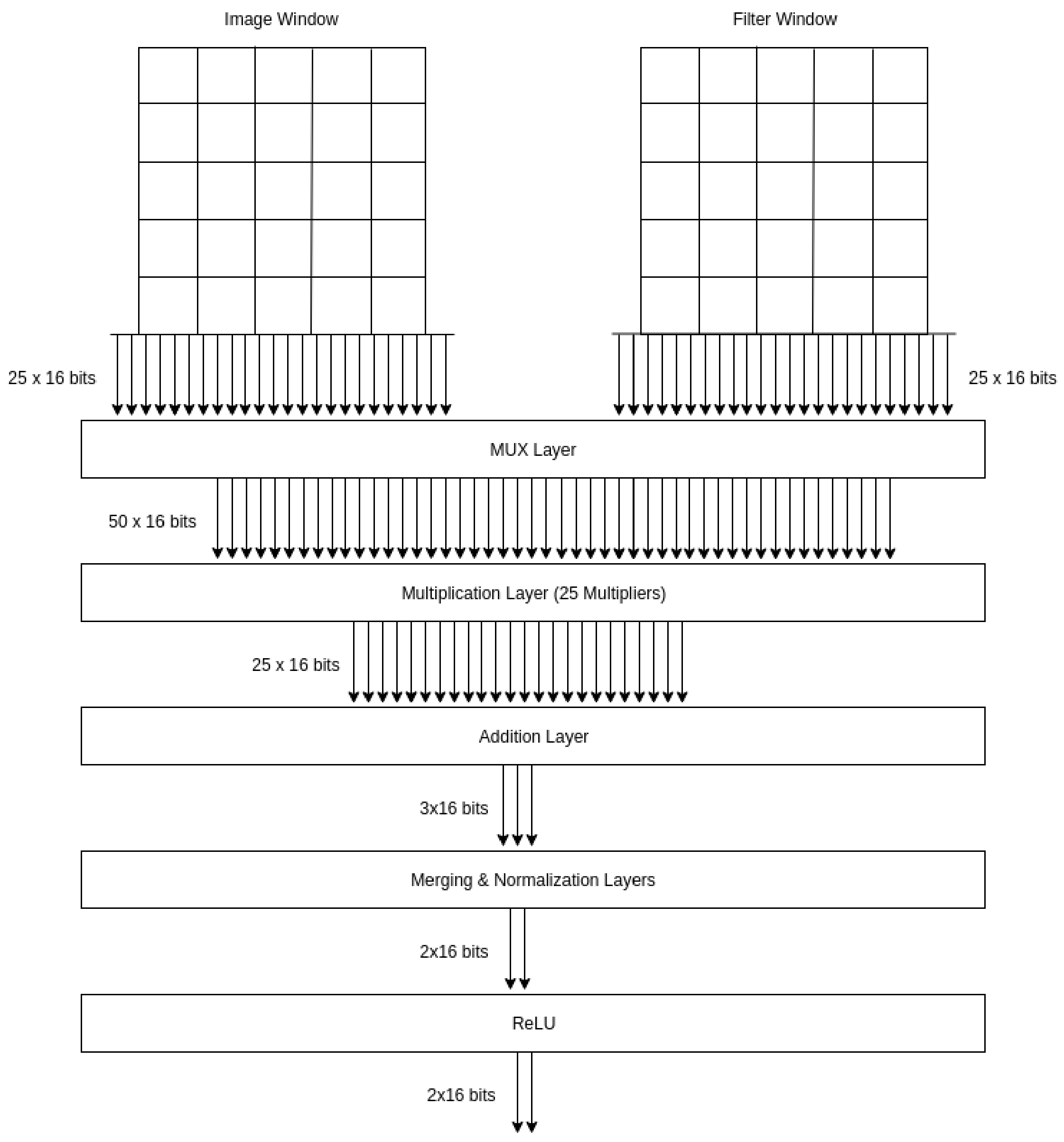

Figure 9: Computation pipeline

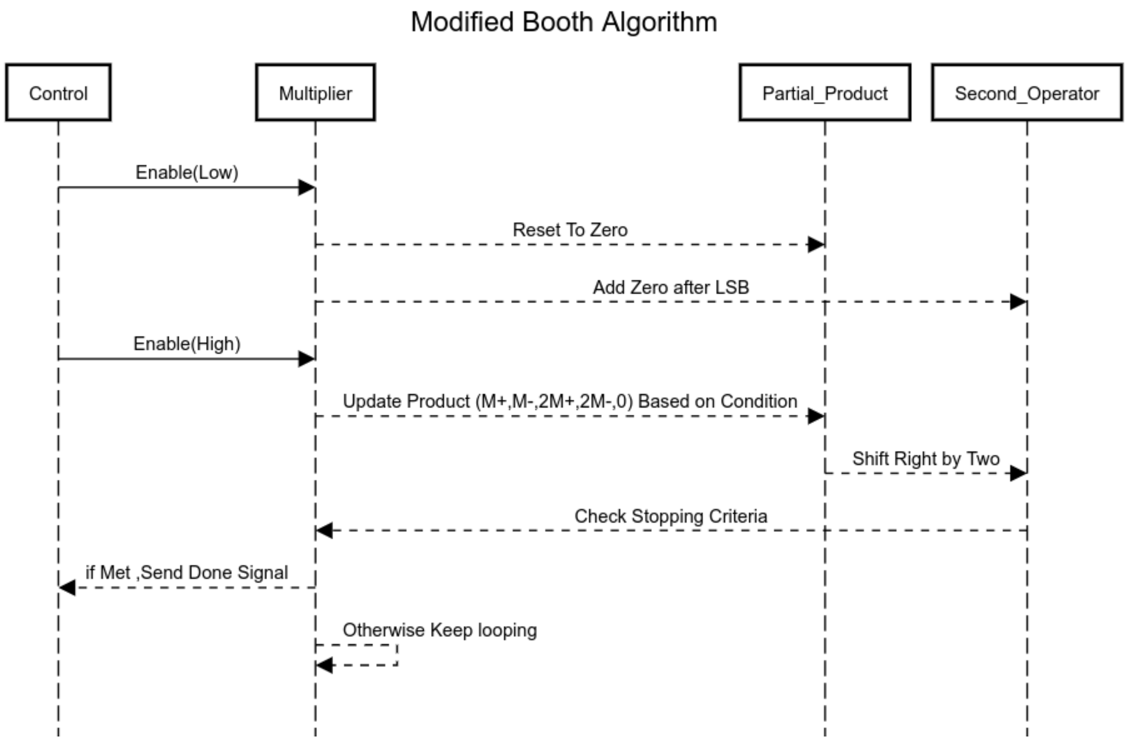

Figure 10: Sequence diagram of the modified booth algorithm

which means that each pixel in the image window is multiplied with one in the filter window in the same position, and we get 100% utilization of the multipliers. But if the kernel size is 3×3, then the wiring is different since we should get the following two outputs from the convolution:

$$\sum_{i=1}^{3}\sum_{j=1}^{3} \text{image}[i,j] \cdot \text{filter}[i,j] \quad \text{and} \quad \sum_{i=1}^{3}\sum_{j=1}^{3} \text{image}[i,j+1] \cdot \text{filter}[i,j],$$

with a utilization of 72%. Thus comes the need of multiplexing the values of the two windows depending on the kernel size. We should note that in the case of 3×3 filter size, the first resulting 9 values concern the first convolution output, and the 9 following values concern the second output, whereas the remaining 7 are garbage.

We use the modified booth multiplication algorithm. Figure 10 shows a sequence diagram of the algorithm.

3.3.2 The Addition Step

The resulting 25 values from the multiplication step are divided into 3 groups of sizes 9, 9 and 7. Each group is an input to an n-operand adder (where n is 9 or 7).

The n-operand adder consists of a register array to hold the operands, and an array of adders that feedback to the registers. Figure 11 depicts the structure of a 9-operand adder. The 7-operand adder has almost the same structure.

3.3.3 The Merging and Normalization Steps

The three results of the previously described n-operand adders will either be merged or kept separate depending on the filter size. In case of a 5×5 filter, the results will be merged by taking the sum. Otherwise, they will be kept separate, and the output is the result of the first 2 trees. Figure 12 shows the internal structure of the merging unit.
To obtain the pooling result, the addition result is then right shifted by either 3 or 5 depending on the filter size. Figure 13 shows the internal structure of the normalization unit.

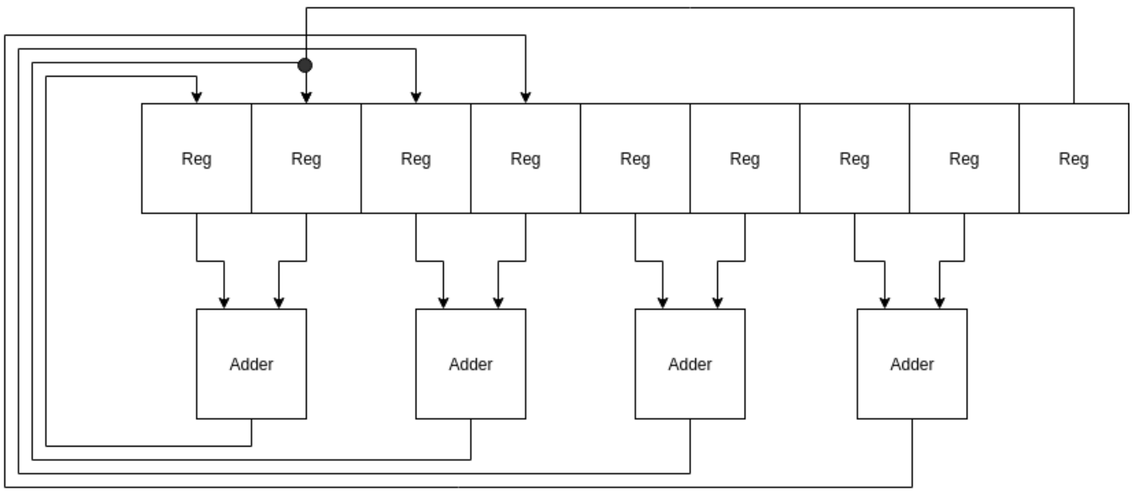

Figure 11: 9-Operand Adder Structure

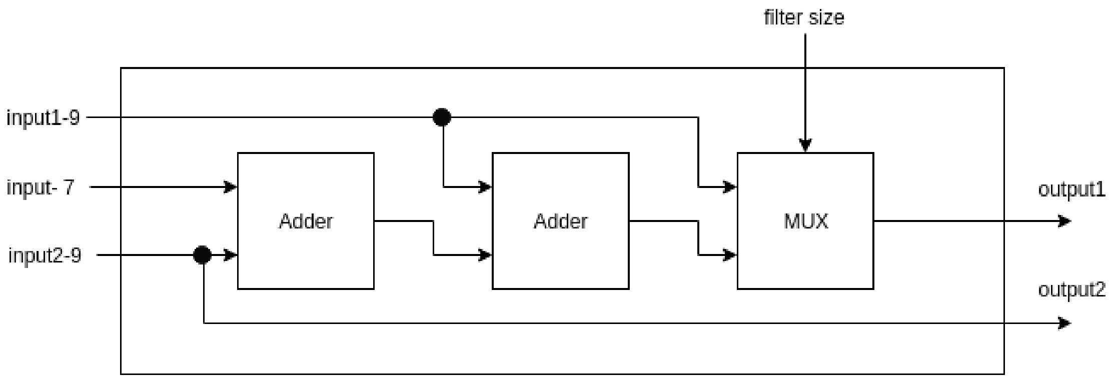

Figure 12: Merging unit

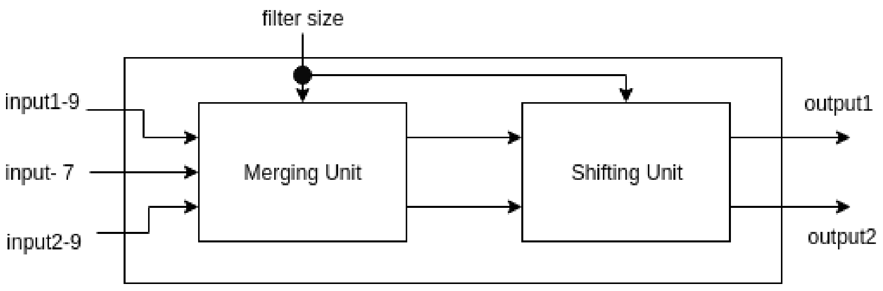

Figure 13: Normalization unit

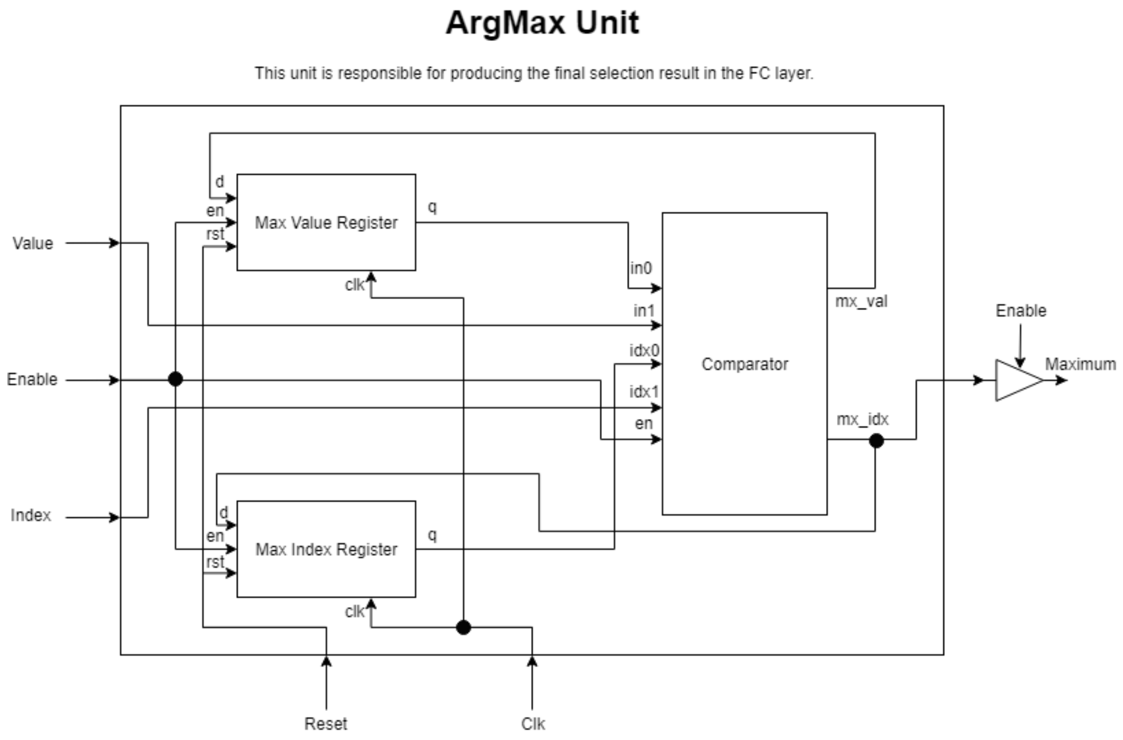

Figure 14: argmax internal structure

3.3.4 The ReLU Step

If convolution is being performed, the final step returns $\max\{0, \text{result1}\}, \max\{0, \text{result2}\}$ where result1 and result2 are the outputs of the previous step. If, however, pooling is performed, this step is not done.

3.4 Argmax Unit

The Argmax unit is responsible for selecting the maximum of all responses produced at the final stage of the fully connected layer which corresponds to the classification result.

3.4.1 Structure

The Argmax unit is comprised of a comparator and 2 registers, one is used for storing the maximum value so far, and the other is storing the index corresponding to the maximum value.

A schematic of the unit's structure can be seen in Figure 14. Also Figure 15 shows the internal workings of the comparator unit used.

3.4.2 Workflow

The ArgMax unit first has to be reset, so the controller starts communication through sending a reset signal that causes the unit to initialize its internal registers. Once the initialization step is done, the reset signal is set to low and the communication begins.

The controller then sends a new value and its corresponding index to the ArgMax unit along with an enable signal that triggers the unit to compare the newly provided value against the current maximum value and updates the maximum value and its corresponding index accordingly.

The index corresponding to the maximum value can be accessed at all times through the output port.

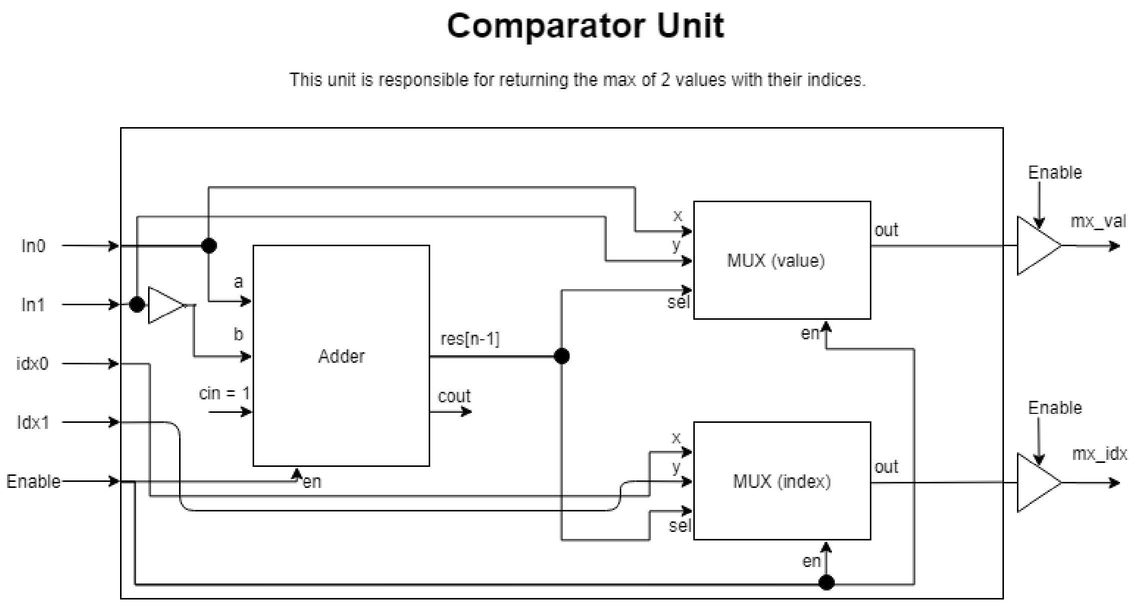

Figure 15: Comparator internal structure

A visual representation of the timeline of events and the communication protocol is shown in Figure 16.

4 Control Unit

4.1 Overview

The control unit is a finite state machine whose aim is to control and generate all the required programming signals for the computation unit as well as read and write data from the memory. The state diagram is shown in Figure 17. The starting state in Initialize Filter Window. A thing to note is that the distinction between fully connected layers and the convolution is made within the states, there is no separate state for convolution and matrix multiplication because the same computation block is used for both.
The general pipeline is as follows: the filter and the image are brought to small and fast caches from the RAM, then the convolution operation is done on a 5x5 block, the result of which is saved back in the memory. Sometimes, it is required to fetch previous values or biases from memory, and on other times the cache should be updated. Finally, the classification phase is done last. Each operation is explained more fully in the pipeline section.

4.2 Pipeline

4.2.1 Initialize Filter Window

The first phase of the pipeline (after being informed that the data is ready) is to start loading the filter window. The filter window, in the computation unit, contains the current filter convolution is done with in the case of convolutional layers and contains a 5x5 subblock of the vector to compute the inner product with in fully-connected layers. In the case of pooling, this phase is skipped.
The layer type (convolution / pooling / fully-connected) is brought in along with the dimensions of the filter (width and height) from predetermined locations in the memory. If the filter is to be fetched, a counter starts counting until either a 3x3 or a 5x5 block is fetched into the filter window, depending on the filter type. There are some edge cases in which padding the filter is required, and these are detected by a small combinational circuit. The basic flow is shown in Figure 18.

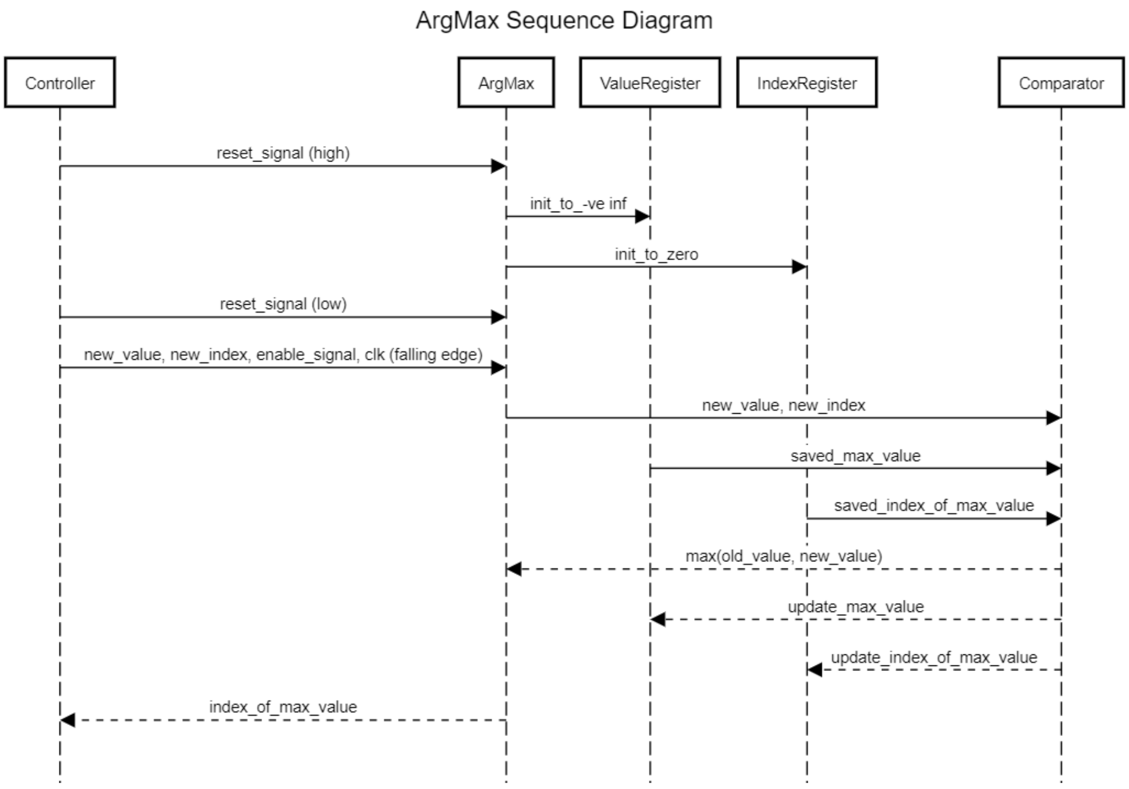

Figure 16: Argmax workflow sequence diagram

4.2.2 Load Into Cache

The second phase of the pipeline is to load the image cache. The image cache, present in the control unit, contains in the case of convolution a 5x28 subblock of the image at any point (perhaps with fewer columns or rows, but with a maximum size of 5x28). The cache is designed to maximize data reuse in convolutional layers (which takes up the majority of computation time in CNNs). More information on the image cache is given later. In the case of a fully-connected layer, the image cache contains a 5x5 sequence of values from the final vectorized output of the previous layers.
We first determine the height and width we are going to bring from the memory and place them in counters. Then we loop over the heights and widths until the image cache is filled. In some edge cases it is required to load zeros instead of pixels into the cache as a form of padding, we detect this and multiplex the input to the cache accordingly. The basic flow is shown in Figure 19.

4.2.3 Initialize Image Window

The third phase of the pipeline is to initialize the image window. The image window, present in the computation unit, contains a 5x5 subblock of the image brought in from the cache. The window is fed the data one column at a time from the cache, in the initialization stage we keep a counter of the current column transferred from the cache and simply transfer the first five columns. The basic flow is shown in Figure 20.

4.2.4 Start Convolution

The fourth phase of the pipeline is to program the computation unit to start the convolution operation. In DCNN, all computations (save for the final argmax classification) are done by the computation unit: pooling, convolution, and fully-connected layers. To the computation unit, the difference between pooling and convolution is established by the presence of an indicator signal for pooling, but there is no difference between convolution and fully-connected layers. It is up to the control unit (in the previous phases) to have fetched the right sequence of vector elements in the right order to do the inner product operation using convolution.

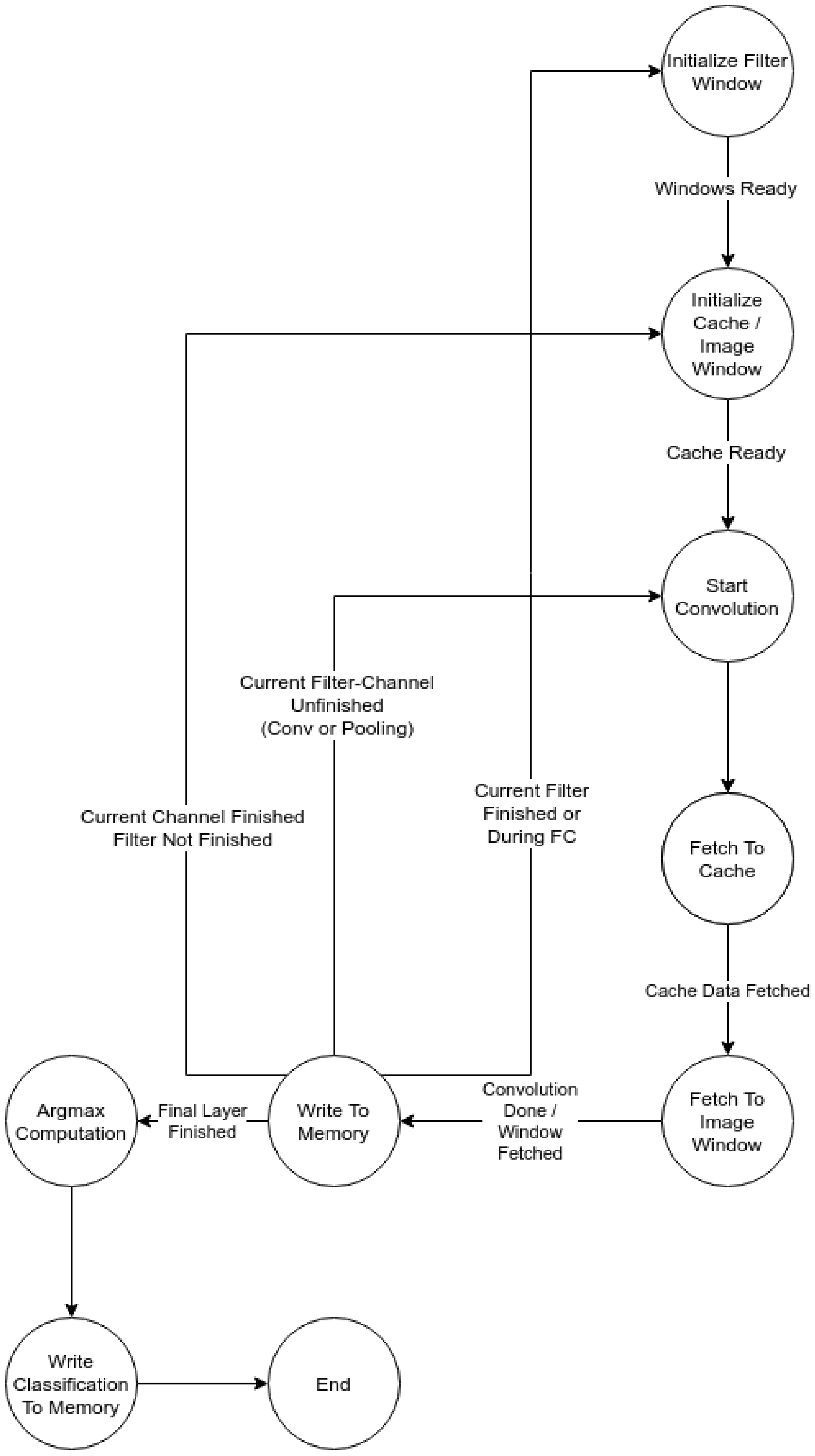
Initialize Filter Window
Windows Ready
Initialize Cache / Image Window
Cache Ready
Start Convolution
Fetch To Cache
Cache Data Fetched
Fetch To Image Window
Convolution Done / Window Fetched
Write To Memory
Current Filter-Channel Unfinished (Conv or Pooling)
Current Channel Finished Filter Not Finished
Current Filter Finished or During FC
Final Layer Finished
Argmax Computation
Write Classification To Memory
End

Figure 17: Overall State Diagram

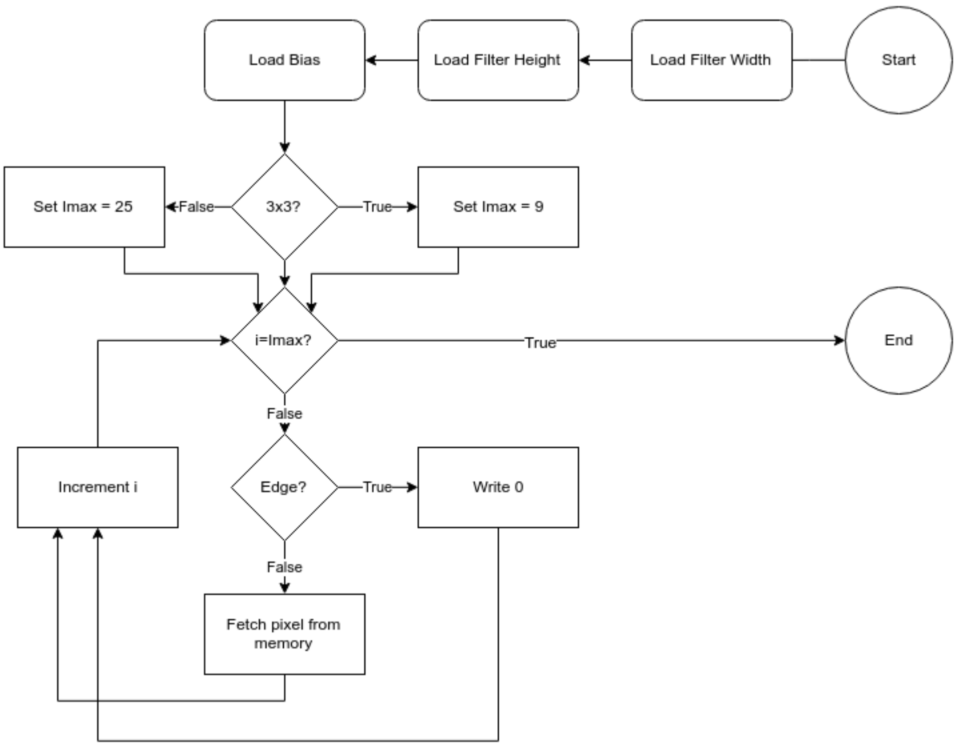

Figure 18: Initialize Filter Window

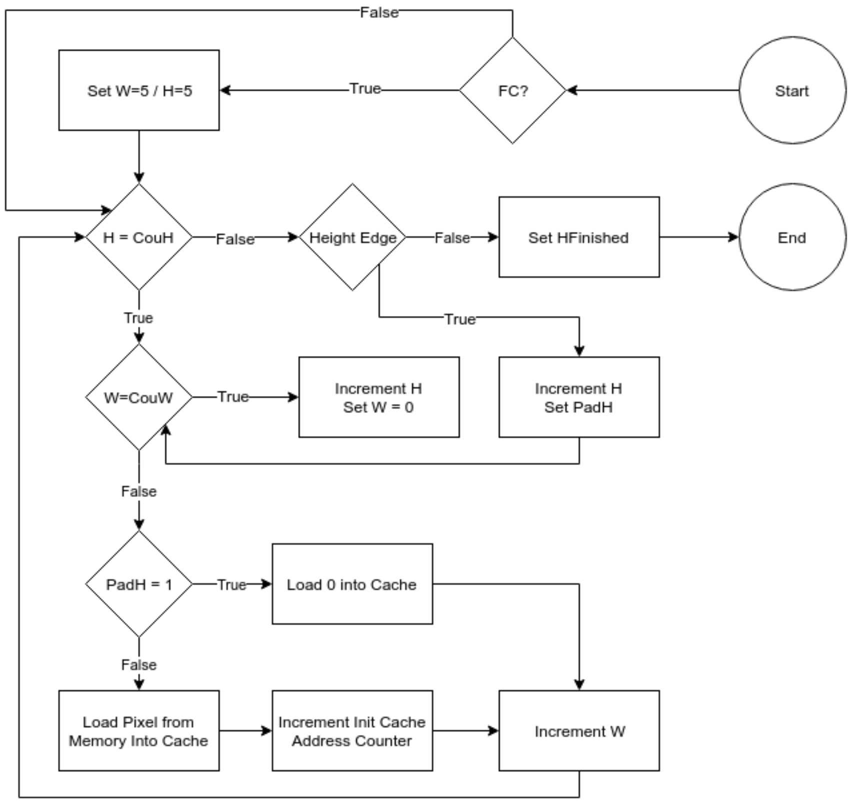

Figure 19: Load Into Cache

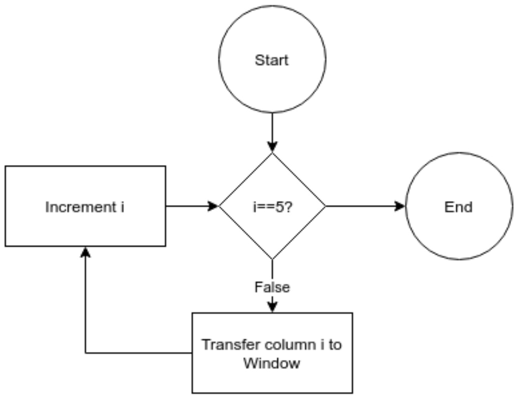

Figure 20: Initialize Image Window

Because the data has been brought in by previous stages of the pipeline and will be written to the memory in stages after this operation, the flow is rather simple. The layer type is first checked: the FC and pooling layers need to fetch nothing and can therefore start right away, however, if a convolution operation is detected then either the bias (at input channel 0) should be fetched or the output of the convolution with the same filter at the previous input channel (i.e. if convolving with an RGB image, then when convolving with the B component the output of accumulation on the R and G components is first brought in), the input is fetched from an address present in the input offset register. In the case of 3x3 convolution, an extra input is brought in (because two convolution operations are done, not one). The fetched input biases are then loaded into the computation unit and a signal is given to start the operation. The basic flow is shown in Figure 21.

4.2.5 Fetch To Cache

This stage happens mid-convolution to insert a pixel into the image cache. There certain edge cases to take into consideration. In case of a 3x3 filter being applied, our architecture entitles pushing 2 zeros rows into the cache at end of the image input for correct operation.
Furthermore, our design makes two calculations at once in the case of a 3x3 filter, thus two pixels have to be inserted into the cache. Figure 22 shows basic implementation flow.

4.2.6 Fetch To Image Window

This stage is responsible for "moving" the image window horizontally by multiplexing the current column from the image cache as input to the image window. In the case of a 3x3 filter, as mentioned above, two columns are shifted into the image window. Figure 23 shows the basic flow.

4.2.7 Write To Memory

This stage is started once a value is ready from the computation unit. The value is written to current out address.
During one layer, the “working space” memory is divided into two sections, a section to write current channel convolution output to and a section to read previous channels from. Since each filter is applied to all channels, first channel pixels are added to filter bias,the rest of channel outputs are accumulated, thus address of current filter output needs to be tracked (output base reg) and address of current value in channel as well (output offset register). After a layer is done the two sections are swapped for maximum memory reuse.
According to current position in network, the next stage is chosen. 5 scenarios are present:

1. Current filter convolution on all image channels ended, or an FC operation is in progress: filter window updated.
2. Current layer is finished: memory output base addresses are swapped and a new filter is fetched.

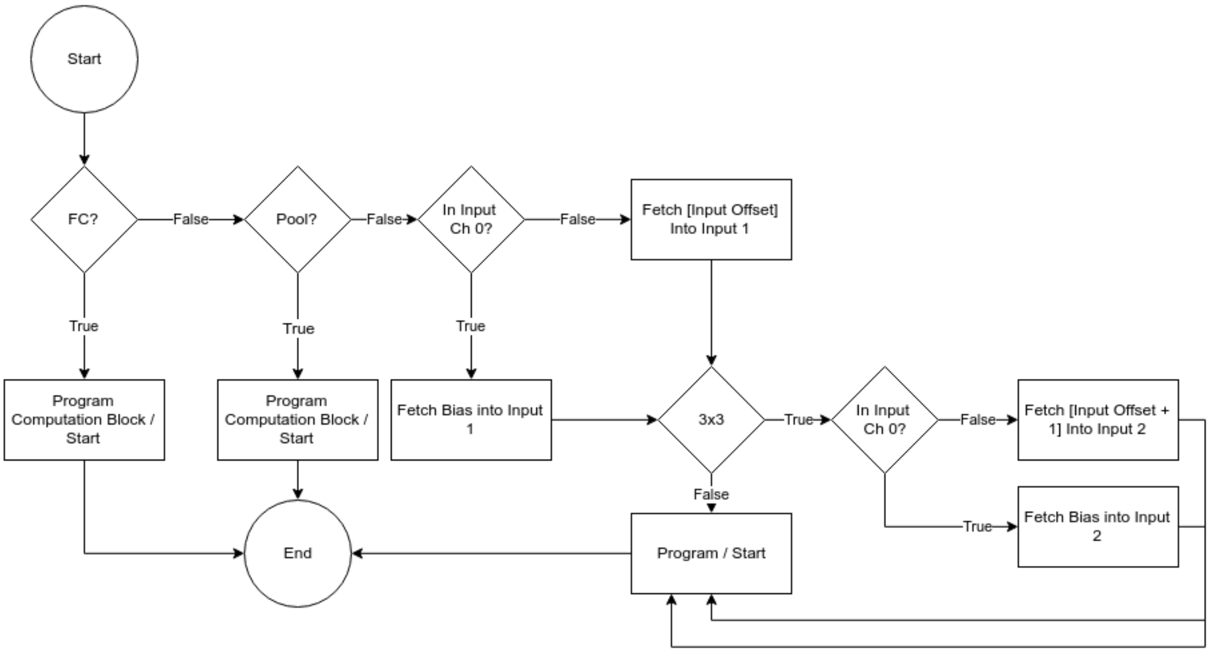

Figure 21: Start Convolution

3. Current filter still in process on image channels: the convolution operation is resumed.
4. Filter convolution on new channel started: output base address is copied to output offset, then cache is initialized with new image channel.
5. Final layer is concluded: Argmax operation started.

Figure 24 shows the basic flow.

4.2.8 Argmax Computation

This stage happens at the very end of the process. Argmax module is initialized with minimum value and then fed FC outputs one by one to maximize on. Since we're classifying digits, it is the maximum of 10 possibilities. Classification is then written to memory for IO to deal with. Figure 25 shows the basic flow.

4.3 Components

4.3.1 Image Cache

The Cache is a block of 28 queues, each queue consists of 5 registers, each one 16 bits, representing a pixel in the image as shown in Figure 26. The input to the image cache is read pixel by pixel,through column inputs, shifting the queue contents vertically with upper most row getting discarded. Thus imitating a vertical slider on the input image. A decoder chooses the columns to insert a pixel into, inserting into all columns shifts the cache an entire row up. A multiplexer selects an entire column to out as input to the image window discussed in the section above.

4.3.2 Registers

The main control registers are given in Figure 27.

4.3.3 Counters

The main counters, divided by purpose, are

1. Loading Image

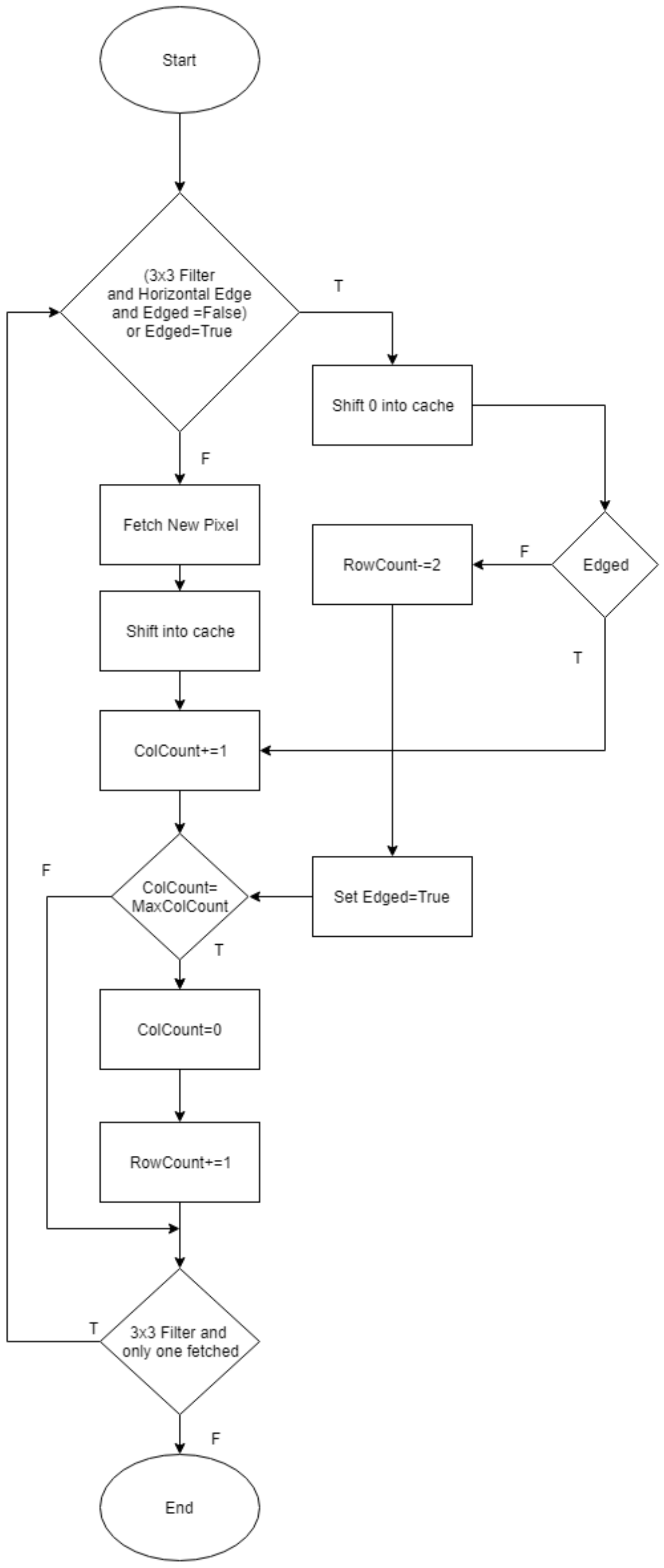

Figure 22: Fetch To Cache

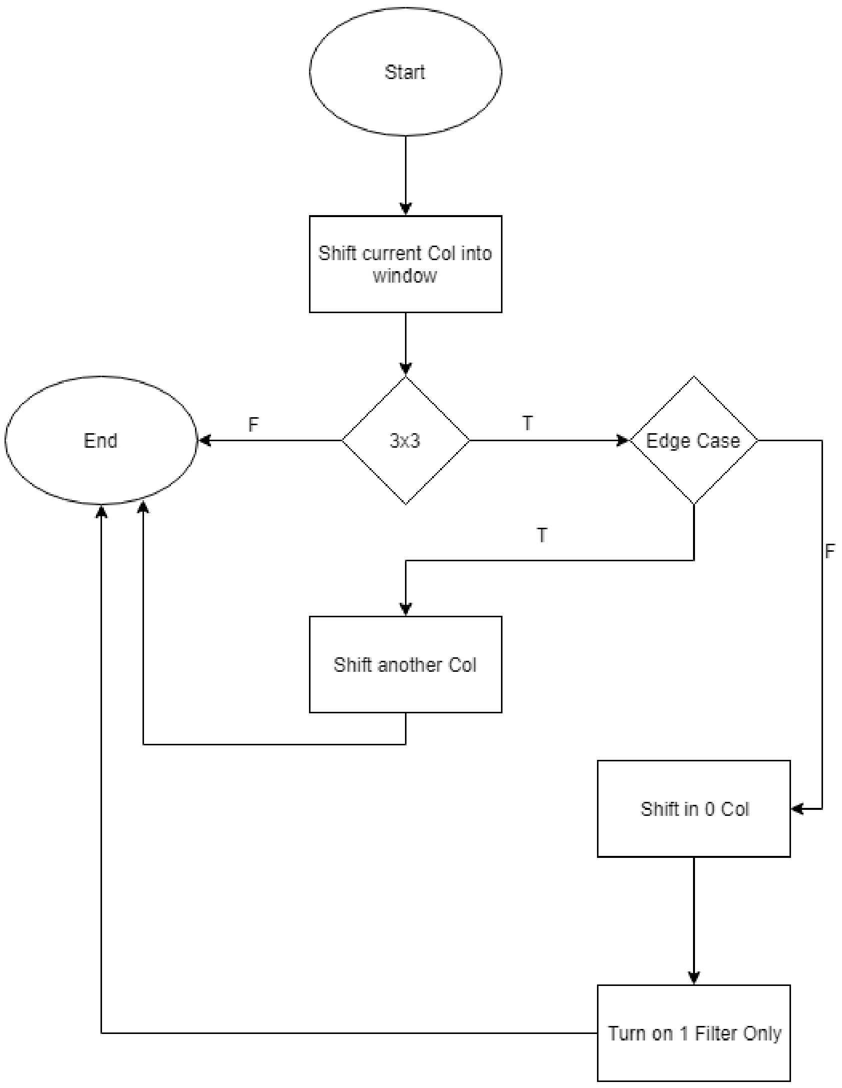

Figure 23: Fetch To Image Window

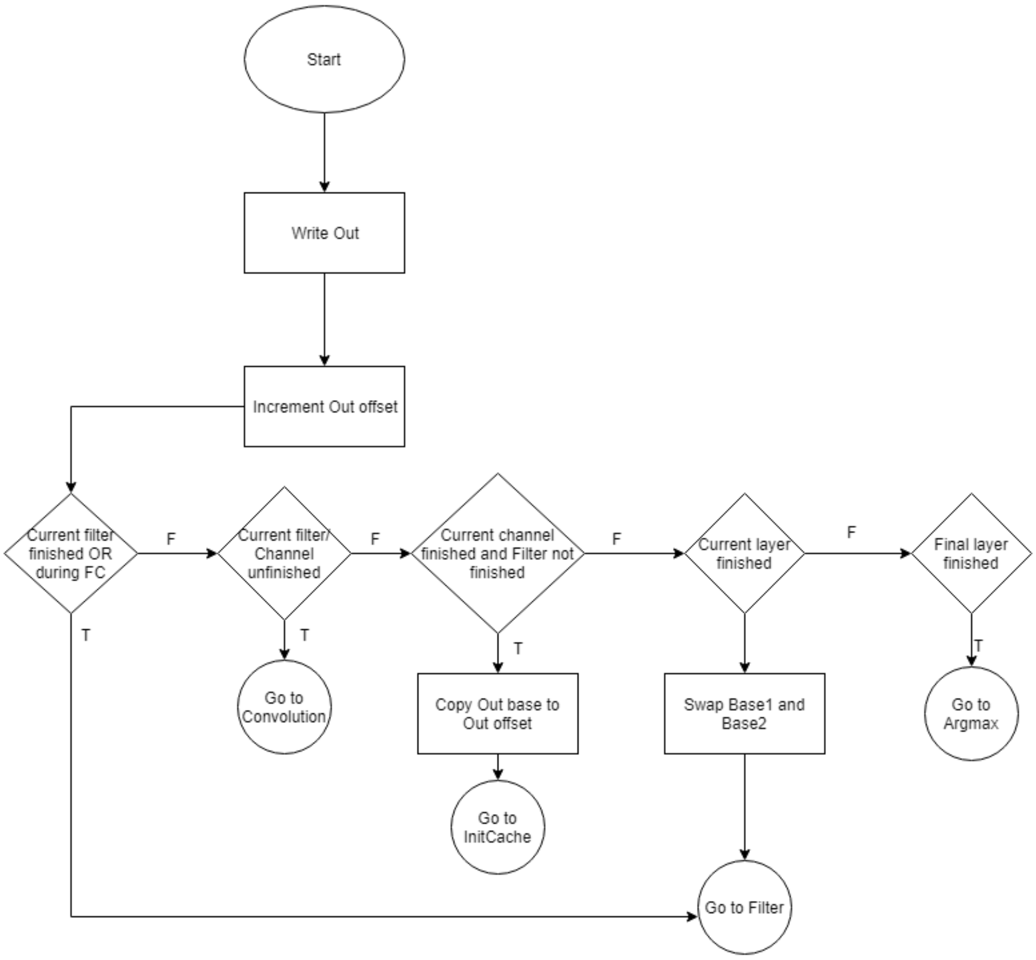

Figure 24: Write To Memory

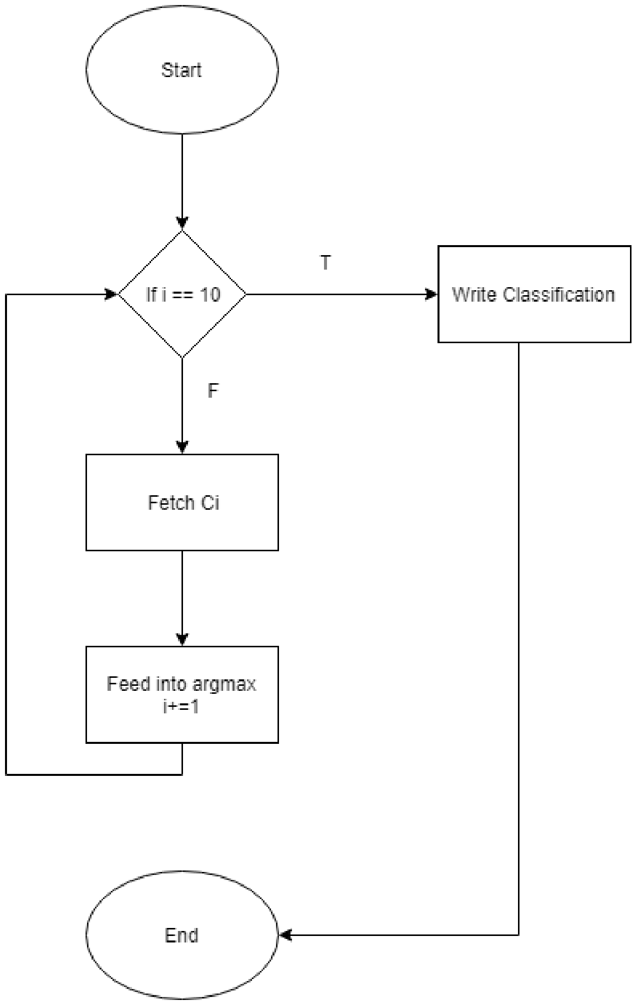

Figure 25: Argmax Computation

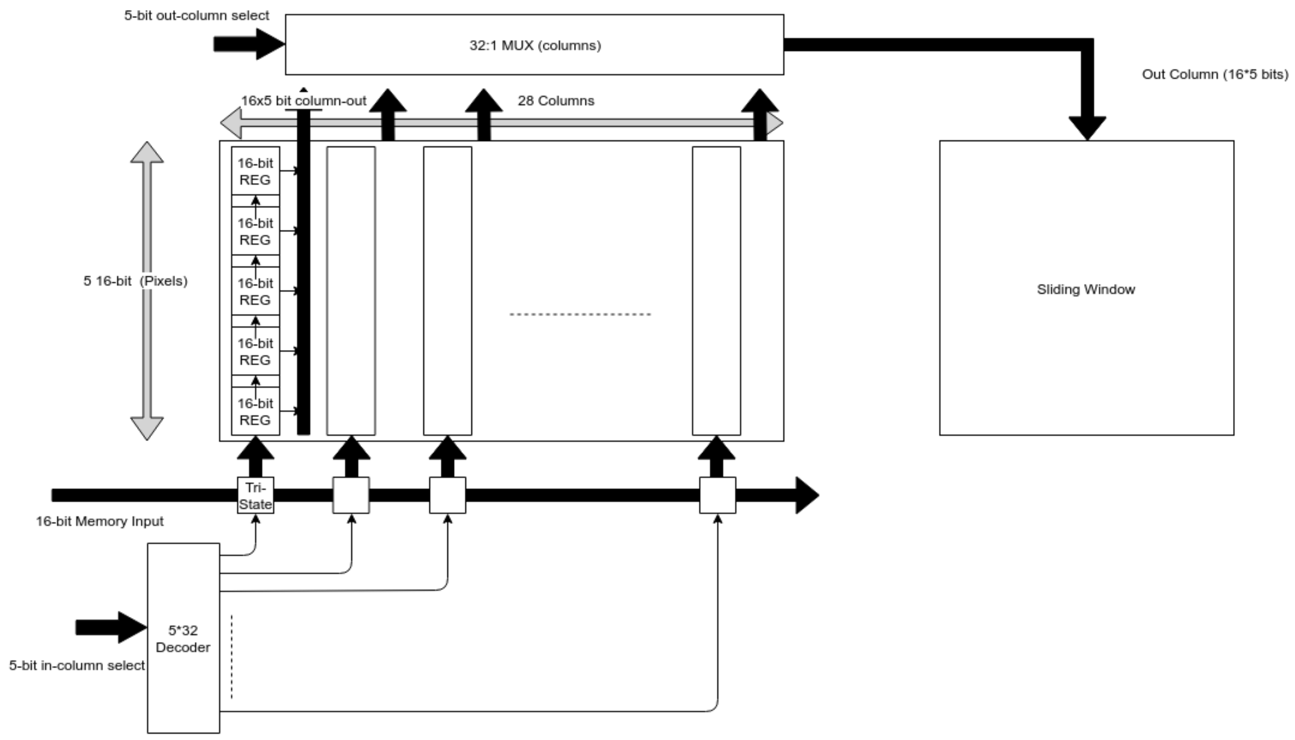

Figure 26: Image Cache

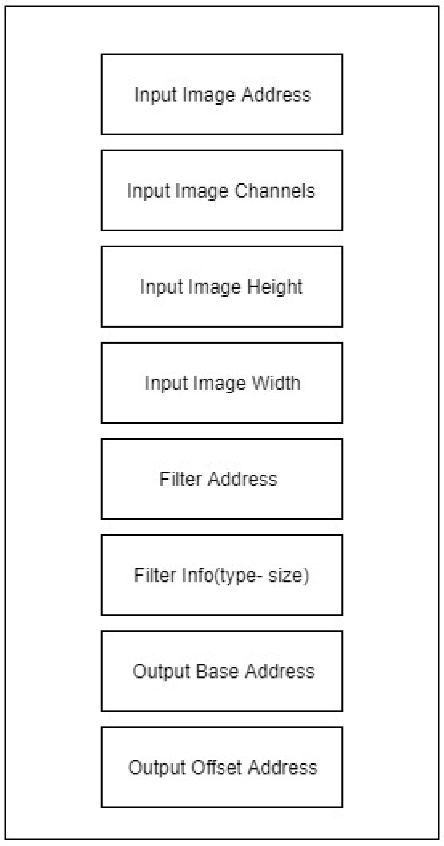

Figure 27: Control Registers

    (a) Channels Counter (3 bits).
    (b) Height Counter (5 bits).
    (c) Width Counter (5 bits).

2. Loading Windows
    (a) Total filter size counter (5 bits).
    (b) Current column counter (3 bits).

3. Computation Control
    (a) Counter for number of words in fully-connected layers (5b).

More counters and registers may be needed in the implementation phase, the design will be adjusted accordingly.