REPORT

Name: Abhiram Dronavalli (adronav@ncsu.edu)
Student ID: 200203685

Delay (ns to run provided example) Clock period: 10 ns # cycles: 13,070	Area: (um 2) Logic: 11027.562135 Memory: N/A	1/(delay.area) (ns -1 .um -2) 6.9382×10^{-10}
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

With the increasing demand for high speed processors to solve the Convolutional Neural Network(CNN), it is a requirement to develop a CNN Application Specific Integrated Chip(ASIC) to repeatedly perform multiply-accumulate(MAC) operations and let the processor handle the rest of the load.

The following project describes the design and synthesis of a 2 layer CNN ASIC. The input to the neural network is an 12x12 matrix of signed 16-bit two's complement numbers and the output generated is an array of 8 16-bit two's complements numbers. The device under test (DUT) is designed in order to read and write the matrix elements from a SRAM located off-chip. This allows the design to update it's computations with the continuously changing input matrix. The design aims to achieve a modular bottom-up approach targeting the scalability of the DUT. The data from the memories is read serially into the MAC IP in a pipelined fashion.

The DUT achieved an area 11027.562135 um^2 and took about 13070 cycles to compute on one set of input data stream with a clock period of 10 ns .

1. Introduction

The following report describes the hardware design and implementation that performs two stages of operations on an input array and generate an output vector. DUT will be implementing two layers, the first layer is a feature extraction from an input and the second layer is a fully connected layer to identify classes.
The design is intended to perform computations faster since MAC calculations are performed in parallel. The completed design achieved the following results:

- Area: 11027.562135 um^2
- Fastest Clock Period possible: 10 ns
- Total Dynamic Power: 412.6 uW

The report has been divided into 6 sections. The micro-architecture shows a high level design of the DUT and explaining each module's data path and control path in detail. The interface specification sections concentrates on the various signals used to synchronize these modules and explain the transfer of data across these modules. Following this, in the technical implementation section, the approach of design is explained and its implementation in this project.
Finally, verification and results section explain the debug of the verilog code and point out the final numbers achieved for the DUT.

2. Microarchitecture

In this section, an overview of the design approach is provided along with the design procedures and it's diagrams.

- **Hardware "algorithmic" approach**

The 12x12 input matrix containing 16-bit two's complement numbers is stored in Data Input Memory (DIM) having a total of 1024x16 bits of storage. This data memory also provides scratchpad to store intermediate value in between computation. Along with this, Filter Vector memory (BVM) having a total of 1024x16 bits of storage is also provided which contains data analogous to CNN weights. Finally, an Output Vector Memory (OVM) with 8x16 bits of storage is provided to store the results obtained from the computation.

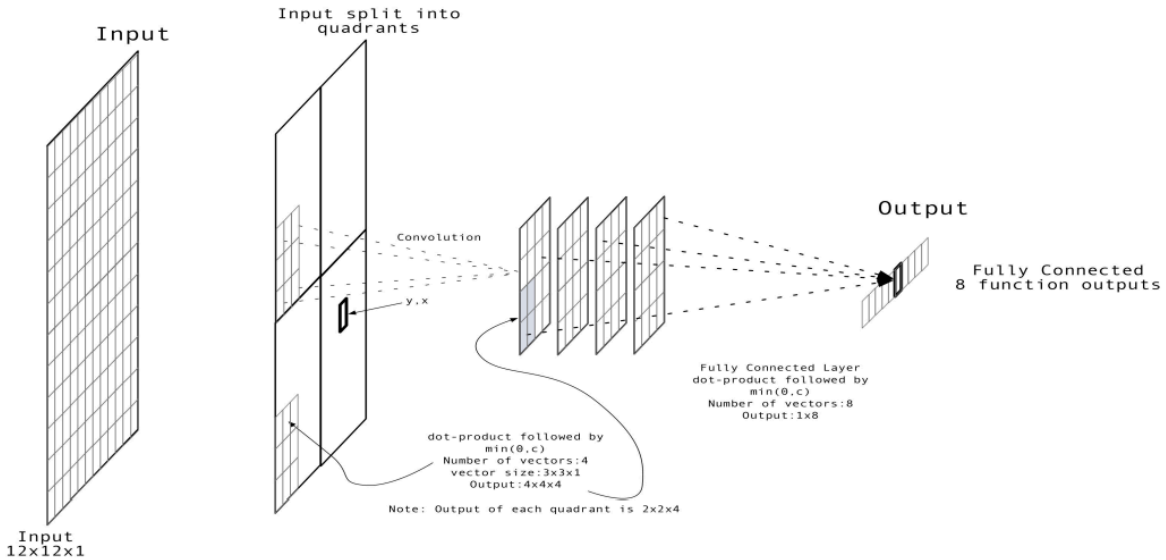

Fig 2.1: Data Flow

The above shown figure describes the flow of data across the DUT. The data flow is divided into 2 major steps as explained below:

1. Step One

The first step is to first perform dot-products on regions of the inputs using four separate step one vectors. We will refer to these as the $[b]_{0..3}$ vectors and they are loaded in the Filter Vector Memory as shown in Fig 2.1. Each of these [b] vectors are 9x1. These vectors are multiplied with data from input matrix which we refer as a(q,i). We will consider the 12x12 input as four 6x6 quadrants and operate on each of these quadrants in an identical manner.

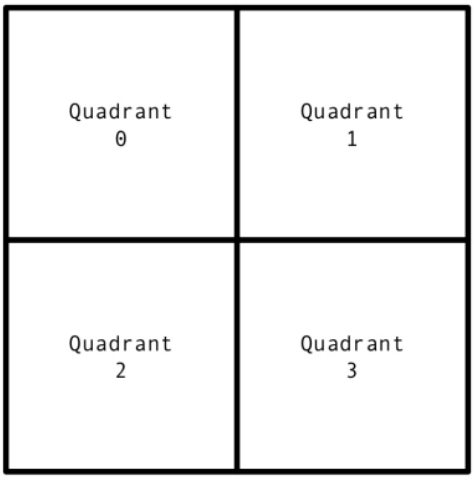

Fig 2.2: Quadrants

We will form four 1x9 vectors, as shown in the Fig 2.3. We will refer to these vectors as [a(q,i)]. We will then perform a dot-product of each of these [a(q,i)] vectors against each of the four supplied step one [b]0..3 vectors.

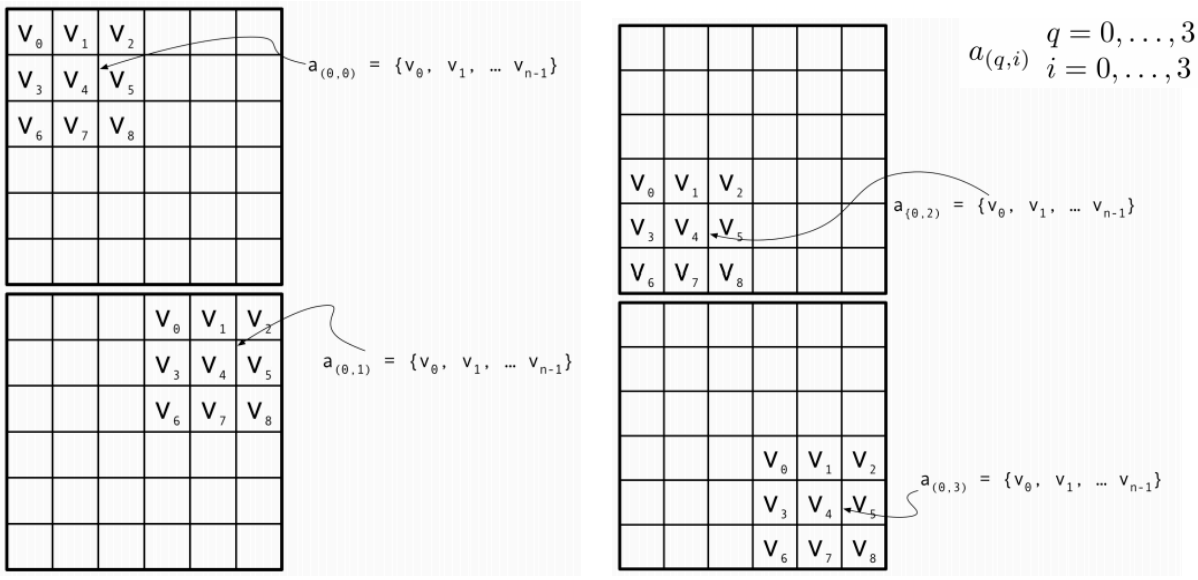

Fig 2.3 Sub quadrant representation

$$b = [b_0, b_1 ... b_8]$$
$$a = [a_0, a_1 ... a_8]$$
$$c = \sum_{n=0}^{8} a_i \cdot b_i$$

Fig 2.5: Dot product

$$z_{i,j}^{b} = f(c_{i,j}^{b}) = \begin{cases} c_i & , c_i \geq 0 \\ 0 & , otherwise \end{cases} \text{ where } b = 0 \ldots 3, i = 0, 1, j = 0, 1$$

Fig 2.6: Threshold Function

This will create four int32's c values which we will form into consider to be a 2x2 array. For each [c], perform the threshold function shown in Fig 2.6 and truncate the result z to 16-bits. Each quadrant will require 16 dot products. This operation on each quadrant will form a 2x2x4 output array as shown in Fig 2.8.

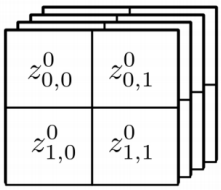

Fig 2.7: Quadrant output

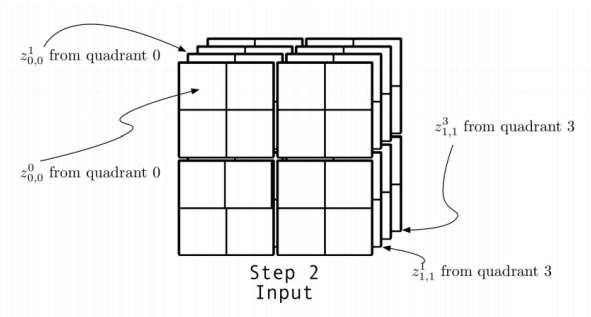

Fig 2.8: Output from Step One

2. Step Two

In step two, we again will perform dot products on this step one output with the eight supplied [m] vectors. These [m] vectors are also supplied in the Filter Vector Memory starting from address 0x040. These eight [m] vectors have 64 elements and you will perform a dot product of each [m_i], i=0..7 vector(s) against a 64-element vector formed from the 4x4x4 output of step one. To reshape the 4x4x4 output of step 1 to create a 1x64 [u] vector, we will extract the elements in row-major fashion starting at the first layer. Again each dot-product will be followed by the threshold function f(x) as shown below.

$$m_i = [m_{i,0}, m_{i,1} ... m_{i,63}]$$
$$u = [u_0, u_1 ... u_6 3]$$
$$w_i = \sum_{n=0}^{63} m_i \cdot u, i = 0, \ldots, 7$$

Fig 2.9: Dot Product

$$O_i = f(w_i) = \begin{cases} w_i & , w_i \geq 0 \\ 0 & , otherwise \end{cases} \text{ where } i = 0 \ldots 7$$

Fig 2.10: Threshold Function

This will result in eight outputs, O and this will be the output of our system. These outputs are then written to output memory. Fig 2.11 shows how data is organized within the memories.

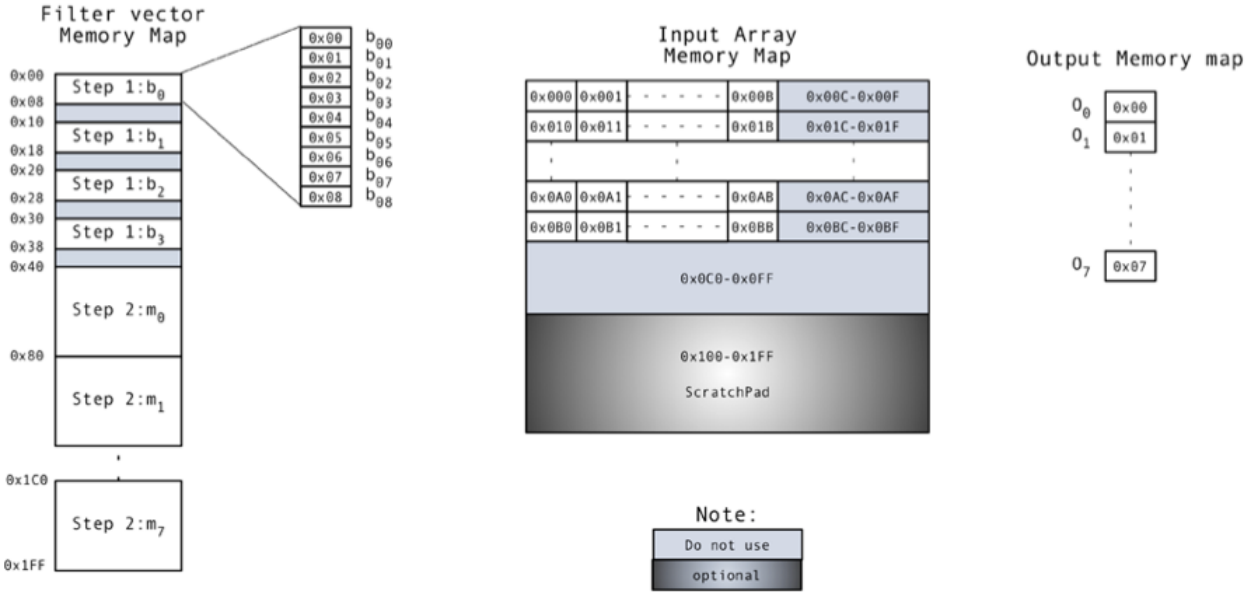

Fig 2.11 Memory Map

- **Hardware design approach**

The design is partitioned into 3 main modules namely quad_module, step2_module and controller. The top level sketch of these modules is shown in Fig 2.12.

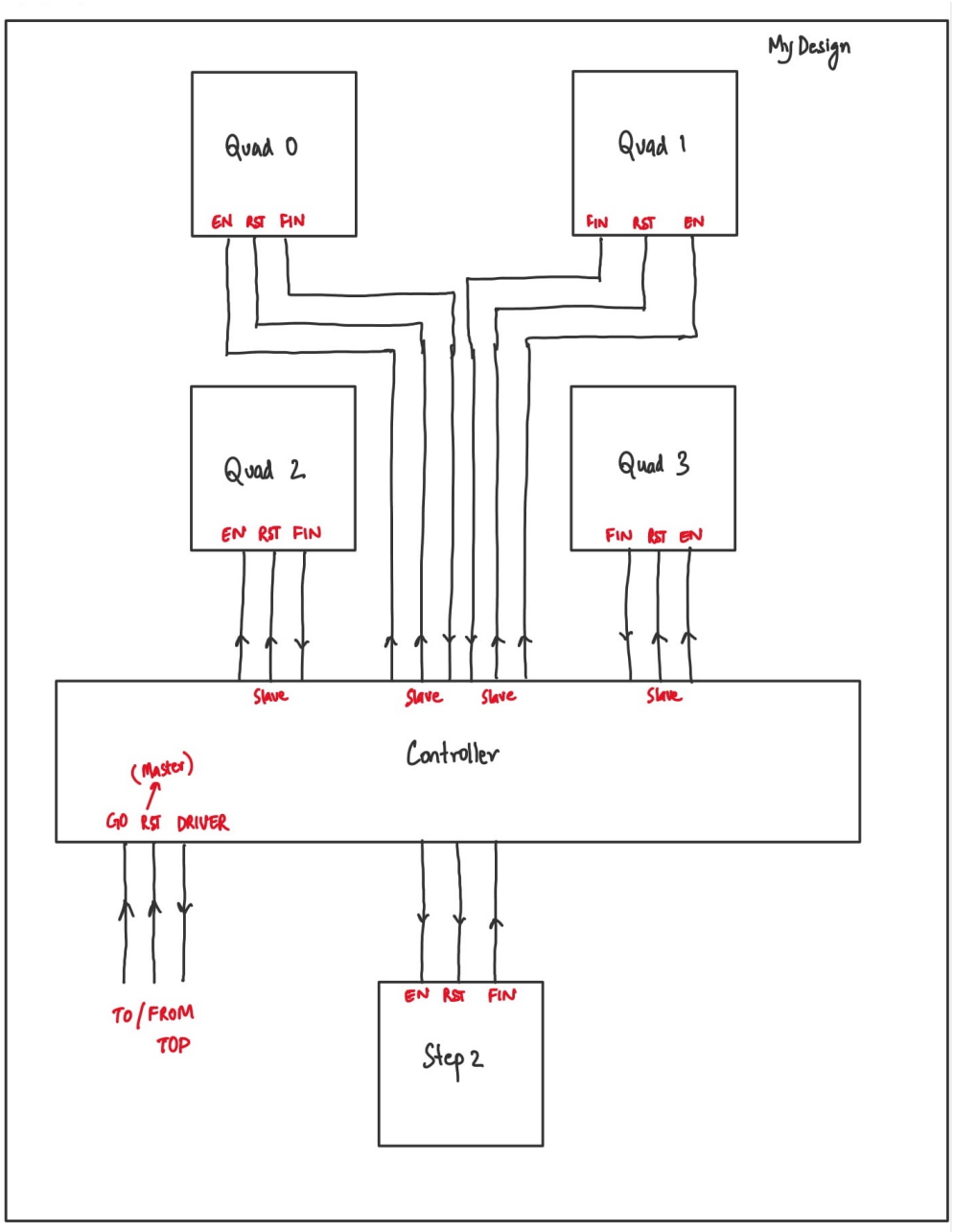

Fig 2.12: MyDesign partitioning

Each of the modules Quad 0, 1, 2, 3 are instantiations of quad_module logic block which perform computation on each of the 4 quadrants of the input matrix. Controller is the heart of the design which sends and receives control signals form the 5 logic blocks and ensures only one block is active at a time since we do not have multiple read ports from the memories. Each of the logic blocks is explained in details below.

1. quad_module Block

The main functions of quad_module block are:

- Generate addresses to read/write data from/to memories
- Fetch data from the memories
- Perform dot-product computation
- Write intermediate results into scratchpad region on memory

Just like every design, the block was created by designing the data path first and the control logic later. The data path consists of address generation logic and data computation logic and similarly separate control logics for address generation and data computation were created. Fig 2.14 and Fig 2.15 gives a complete idea of these data and control logics.

Fig 2.14 shows the design of address generation logic. dim_addr for DIM is generated using counters which correspond to each hexadecimal place of the address. For example, address 0x123 is generated using 3 counters, each of which outputs 1(z-coordinate), 2(y-coordinate) and 3(x-coordinate). Later, these 3 values are concatenated and send to top. We can also see from Fig 2.13 that the counter value for x-coordinate and y-coordinate switches between 2 values using a MUX while the z-coordinate simply outputs 1 or 0. This is because dim_addr takes care of generating both write and read addresses. From Fig 2.11 we notice that input matrix data is located at memory location 0x0<x><x> (<x> indicates don't care) while scratchpad memory is at 0x1<x><x>. Therefore, z-coordinate simply changes between 1 and 0 based on the a_z_enable signal from control logic. While the same signal also ensures that dim_x_addr and dim_y_addr lines take sp_x_addr and sp_y_addr respectively rather than a_x_addr and a_y_addr. This ensures the generation of both read and write address simultaneously.

Similar logic is applied for bvm_addr generation. Since BVM address only reads data therefore no MUX is used to switch address values; a simple counter is used for read address generation.

The control logic controls the timing of enable signals for all the MUXs in the address generation logic based on various external and local signals (status signals from counters and masks). The picture below shows this control logic logic for address path while Fig 2.16 shows for data path.

```
//-------------------------------------Address Path Control-------------------------------------//
// Control logic to control the enables of the counters for address co-ordinate generation
// Masks(Parallel-In Serial-Out Register) used to model the sub-quadrant within the quadrant.

always @ (posedge clk)
begin
    if(reset)
    begin
        a_x_enable <= 1'b0;
        a_z_enable <= 1'b0;
        a_x_mask <= quad_a_x_mask;
        a_y_mask <= quad_a_y_mask;
        b_x_enable <= 1'b0;
        b_y_enable <= 1'b0;
    end //(reset)

    else
    begin
        if(quad_enable)
        begin
            a_x_enable <= 1'b1;
            a_z_enable <= 1'b0;
            b_x_enable <= 1'b1;
            b_y_enable <= 1'b0;

            if(a_y_count == 2'b10 && a_x_count == 2'b10)
            begin
                a_x_mask <= {a_x_mask[0],a_x_mask[3:1]}; // x-coordinate control for sub-quadrant
                a_y_mask <= {a_y_mask[0],a_y_mask[3:1]}; // y-coordinate control for sub-quadrant
                a_x_enable <= 1'b0;
                a_z_enable <= 1'b1;
                b_x_enable <= 1'b0;
            end //(a_y_count == 2'b10 && a_x_count == 2'b10)

            if(a_x_mask[1] && a_y_mask[1] && b_x_addr[3] == 1'b1) b_y_enable <= 1'b1;
        end //(quad_enable)
    end
end

assign a_y_enable = (a_x_count == 2'b10) ? 1'b1 : 1'b0;
```

Fig 2.13 quad_module Address Path Control

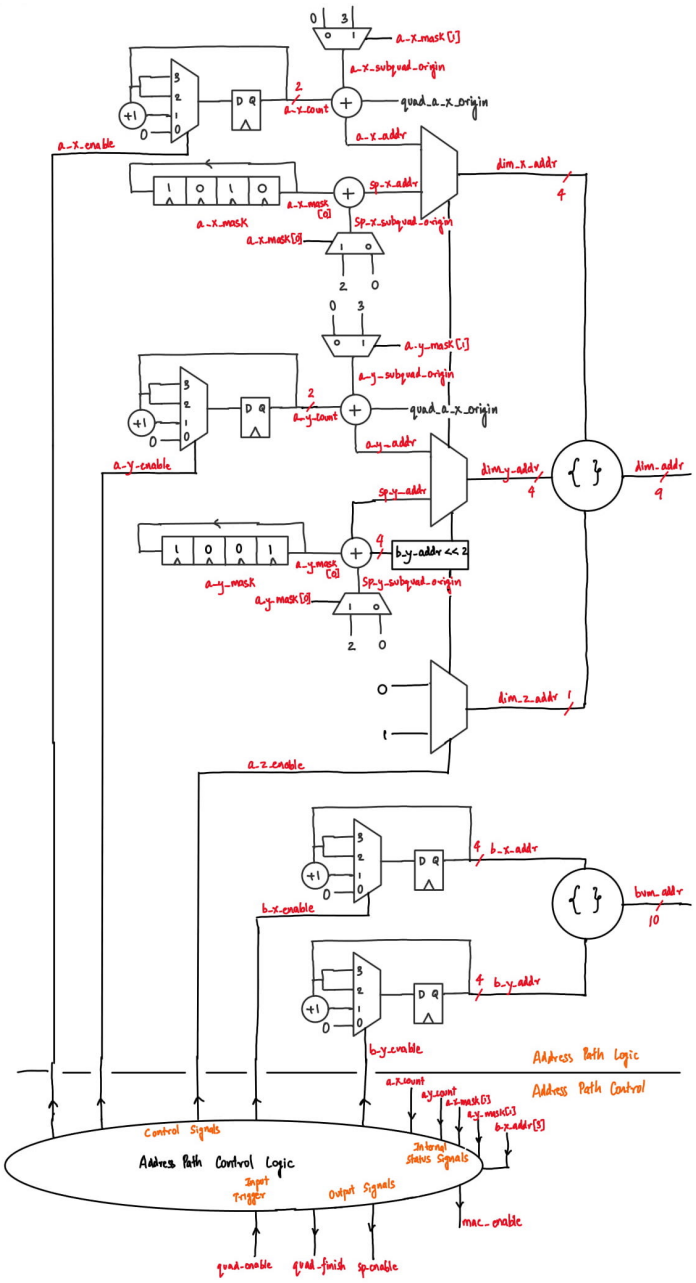

Fig 2.14 quad_module Address Path Logic

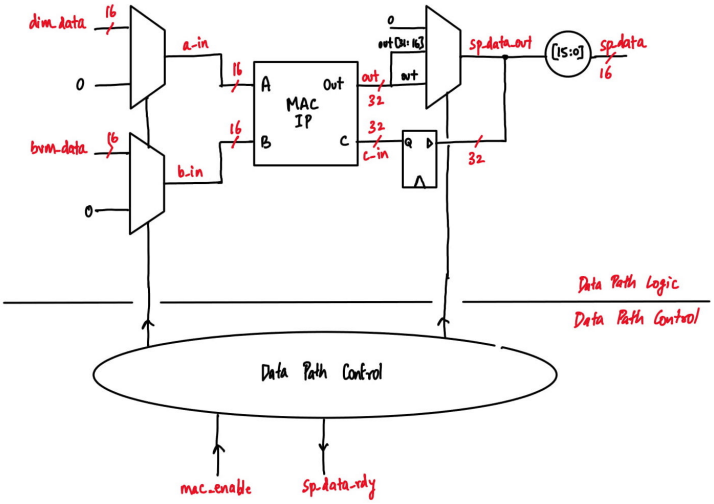

Fig 2.15 quad_module Data Path Logic

```
//--------------------------------------Data Path Control-------------------------------------//
// Control logic for the data flow to the MAC IP

// Shift register to sync data and address control
always @ (posedge clk)
begin
    mac_enable_tmp <= quad_enable;
end

always @ (posedge clk)
begin
    mac_enable_tmp2 <= mac_enable_tmp;
end

always @ (posedge clk)
begin
    mac_enable <= mac_enable_tmp2;
end

// Counter to keep track of input data
always @ (posedge clk)
begin
    if(reset)
    begin
        data_count <= 4'b0;
        TC <= 1'b1;
    end
    else if(mac_enable)
    begin
        if(data_count == 4'b1001) data_count <= 4'b0;
        else data_count <= data_count + 1'b1;
    end //(mac_enable)
end
```

Fig 2.16 quad_module Data Path Control

Fig 2.15 shows the logic for data path. The incoming data is sent to MAC IP by Synopsys DesignWare library in a pipelined fashion. The MUXs on both input and output of the IP allow us the reset the MAC once the computation is complete. Since directly connecting OUT to C can cause timing arcs and combinational logic race condition we add a D-Flip Flop which allows us to both avoid this and provides a stable input to C. The output MUX also takes care of concatenating the output value based on the threshold function.

Data arrives 1 cycle after the address is generated and address generation takes 1 cycle so in total data arrives 2 cycles after address is generated. Due to this a shift register is used to synchronize the enables of both address and data path logics. This can be seen in Fig 2.16. Also, a local counter is created to keep track of input data count.

2. step2_module Block

The main functions of step2_module block are:

- Generate addresses to read/write data from/to memories
- Fetch data from the memories
- Perform dot-product computation
- Write final results into Output Vector Memory (OVM)

step2_module logic is similar to that of quad_module. The address is generated in a similar fashion using counters and later concatenated to form the final address. Unlike quad_module which had same bus to generate both read and write address here we have different busses. Hence, we do not need to switch the counter values using MUXs like we did in quad_module. Also, since we have to compute 64 MAC, we have a larger width counter compared to quad_module. The address path logic is shown in Fig 2.17.

The data path logic is exactly same as that of quad_module. For detailed logic diagram refer Fig 2.15.

The control path logic for address generation and data path logic are also similar to that of quad_module with minor changes. Fig 2.18 and Fig 2.19 show the logic in more detail.

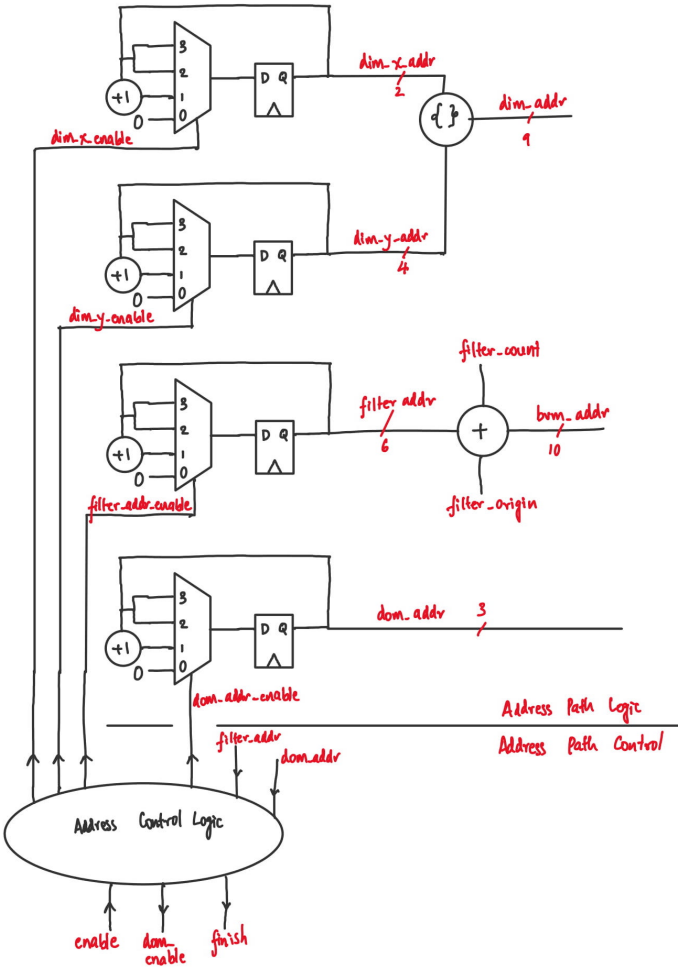

Fig 2.17 step2_module Address Path Logic

```
//--------------------------------------Address Path Control--------------------------------------//
// Control logic to control the enables of the counters for address co-ordinate generation
// Masks(Parallel-In Serial-Out Register) used to model the sub-quadrant within the quadrant.

always @ (posedge clk)
begin
    if(reset)
    begin
        dim_x_enable <= 1'b0;
        filter_addr_enable <= 1'b0;
        dom_addr_enable <= 1'b0;
        filter_count <= 4'b0;
    end
    else
    begin
        dim_x_enable <= 1'b0;
        filter_addr_enable <= 1'b0;
        if(enable)
        begin
            dim_x_enable <= 1'b1;
            filter_addr_enable <= 1'b1;
            if(dom_data_rdy)
            begin
                dom_addr_enable <= 1'b1;
                filter_count <= filter_count + 3'b100;
            end
            else dom_addr_enable <= 1'b0;
        end
    end
end

assign dim_y_enable = (dim_x_addr == 2'b11);
```

Fig 2.18 step2_module Address Path Control

```
//-------------------------------------Data Path Control-------------------------------------//
// Control logic for the data flow to the MAC IP

// Shift register to sync data and address control
always @ (posedge clk)
begin
    mac_enable_tmp <= enable;
end

always @ (posedge clk)
begin
    mac_enable_tmp2 <= mac_enable_tmp;
end

always @ (posedge clk)
begin
    mac_enable <= mac_enable_tmp2;
end

// Counter to keep track of input data
always @ (posedge clk)
begin
    if(reset)
    begin
        data_count <= 6'b0;
        TC <= 1'b1;
    end
    else if(mac_enable)
    begin
        if(data_count == 6'b111111) data_count <= 6'b0;
        else data_count <= data_count + 1'b1;
    end
end
```

Fig 2.19 step2_module Data Path Control

3. Controller

The main functions of controller block are:

- Toggle enable signals among the 5 logic blocks
- Control slave reset for system restart
- Generate driver signal to control flow of data

The controller block is a 16-state Moore FSM logic. The states can be seen in Fig 2.21. Controller block ensures proper synchronization among the enables of the remaining 5 blocks namely quad0, quad1, quad2, quad3 and step2. At any time only one enable is active due to the limitation of read ports. In future, design can be updated for multiple read ports memories by just modifying the controller. Also, controller generates a driver signal based on it's current state which acts as a control for the main MUX in top. This MUX controls the data flow to and from MyDesign to external world since there is only way into the design. A code snippet is shown below to give an idea of the various signals the MUX is controlling.

```
if(driver == 3'b000)
begin
    dut__bvm__address <= quad0_bvm_addr;
    dut__dim__address <= quad0_dim_addr;
    quad0_bvm_data <= bvm__dut__data;
    quad0_dim_data <= dim__dut__data;
    dut__dim__data <= quad0_sp_data;
    dut__dom__data <= 16'b0;
    dut__dom__write <= 1'b0;
    if(quad0_sp_data_rdy) dut__dim__write <= 1'b1;
    else dut__dim__write <= 1'b0;
end
else if(driver == 3'b001)
begin
    dut__bvm__address <= quad1_bvm_addr;
    dut__dim__address <= quad1_dim_addr;
    quad1_bvm_data <= bvm__dut__data;
    quad1_dim_data <= dim__dut__data;
    dut__dim__data <= quad1_sp_data;
    dut__dom__data <= 16'b0;
    dut__dom__write <= 1'b0;
    if(quad1_sp_data_rdy) dut__dim__write <= 1'b1;
    else dut__dim__write <= 1'b0;
end
```

Fig 2.20 Main MUX signals

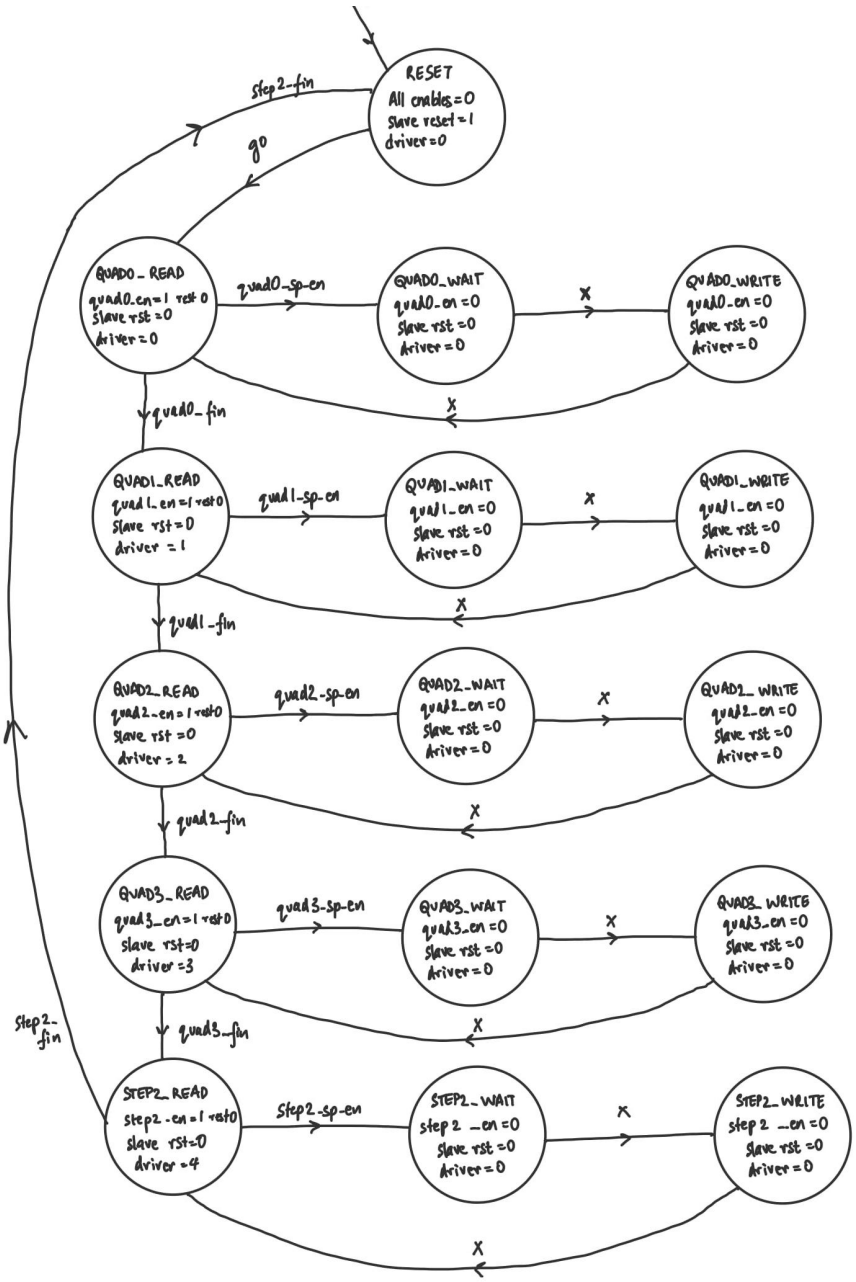

Fig 2.21 Controller FSM states

From the figure above we can see that for each logic block there are 3 states namely QUADx_READ, QUADx_WAIT, QUADx_WRITE. QUADx_READ state is used to read data from the memories while the intermediate output is being computed. QUADx_WAIT is added to synchronize the output data generated with the write address while QUADx_WRITE writes the data onto the bus towards scratchpad memory. Similar steps happens for all the logic blocks including step2_module. This process repeats until the the corresponding finish control signal is triggered by that particular logic block. Finally, when step2_module triggers finish signal the FSM goes into REST state where the slave reset is triggered. This resets the child modules and makes them ready for yet another computation.

3. Interface Specification

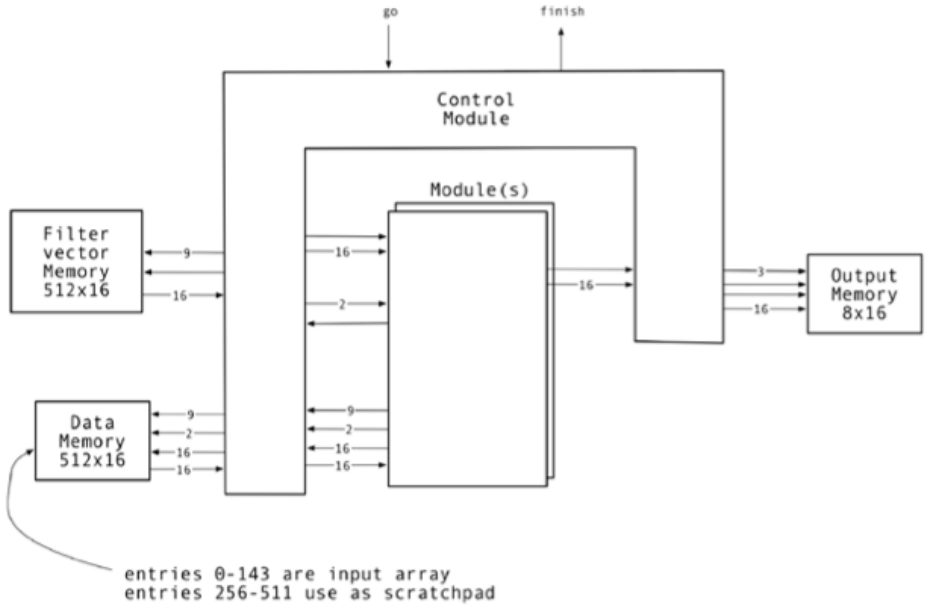

Fig 3.1: Interface

The above figure shows a high level sketch of the interfaces between DUT and SRAM memories and various control signals. The tables shown below summarizes the interfaces of each module.

Module	quad_module						
Inputs	**Width**	**Function**	**Modules Coming From**	**Outputs**	**Width**	**Function**	**Modules Conncted To**
clock	1	clocking	MyDesign	sp_data	16	Data Out	MyDesign
reset	1	reseting	controller	dim_addr	9	address for input data	MyDesign
dim_data	16	input data	MyDesign	bvm_addr	10	address for input data	MyDesign
bvm_data	16	input data	MyDesign	quad_finish	1	module finish	controller
quad_enable	1	module enable	controller	sp_enable	1	enable scratchpad memory	controller
quad_a_x_origin	4	hardwired to 0 or 6 based on instantiation	MyDesign	sp_data_rdy	1	scratchpad result is ready	MyDesign
quad_a_y_origin	4	hardwired to 0 or 6 based on instantiation	MyDesign				
sp_x_mask	1	hardwired to 0 or 1 based on instantiation	MyDesign				
sp_y_mask	1	hardwired to 0 or 1 based on instantiation	MyDesign				
quad_a_x_mask	4	hardwired to 5 always	MyDesign				
quad_a_y_mask	4	hardwired to 9 always	MyDesign				
Module	controller						
Inputs	**Width**	**Function**	**Modules Coming From**	**Outputs**	**Width**	**Function**	**Modules Conncted To**
clock	1	clocking	MyDesign	quad0_en	1	enable for Quad 0	quad_module
reset	1	reseting	MyDesign	quad0_en	1	enable for Quad 1	quad_module
quad0_fin	1	finish signal from Quad 0	quad_module	quad0_en	1	enable for Quad 2	quad_module
quad0_fin	1	finish signal from Quad 1	quad_module	quad0_en	1	enable for Quad 3	quad_module
quad0_fin	1	finish signal from Quad 2	quad_module	quad0_en	1	enable for step2	step2_module
quad0_fin	1	finish signal from Quad 3	quad_module	driver	3	MyDesign MUX control signal	MyDesign
step2_fin	1	finish signal from step2	step2module	reset_slave	1	slave reset for child modules	quad_module & step2_module
quad0_sp_en	1	scratchpad enable signal from Quad 0	quad_module				
quad0_sp_en	1	scratchpad enable signal from Quad 1	quad_module				
quad0_sp_en	1	scratchpad enable signal from Quad 2	quad_module				
quad0_sp_en	1	scratchpad enable signal from Quad 3	quad_module				
quad0_sp_en	1	scratchpad enable signal from step2	ste2_module				
go	1	controller enable	MyDesign				
Module	step2_module						
Inputs	**Width**	**Function**	**Modules Coming From**	**Outputs**	**Width**	**Function**	**Modules Conncted To**
clock	1	clocking	MyDesign	dom_data	16	Data Out	MyDesign
reset	1	reseting	controller	dim_addr	9	address for input data	MyDesign
dim_data	16	input data	MyDesign	bvm_addr	10	address for input data	MyDesign
bvm_data	16	input data	MyDesign	dom_addr	3	address for ouptut data	MyDesign
enable	1	module enable	controller	step2_finish	1	module finish	controller
filter_origin	10	hardwired to 0x40 always	MyDesign	dom_enable	1	enable scratchpad memory	controller
				dom_data_rdy	1	scratchpad result is ready	MyDesign

Fig 3.2 Interconnections among logic blocks

4. Technical Implementation

The design has been implemented in Verilog using Mentor Graphics ModelSim 10.3b and Synopsys DesignCompiler and DesignWare libraries. Fig 4.1 shows the simulated waveform in ModelSim for 3 datasets.

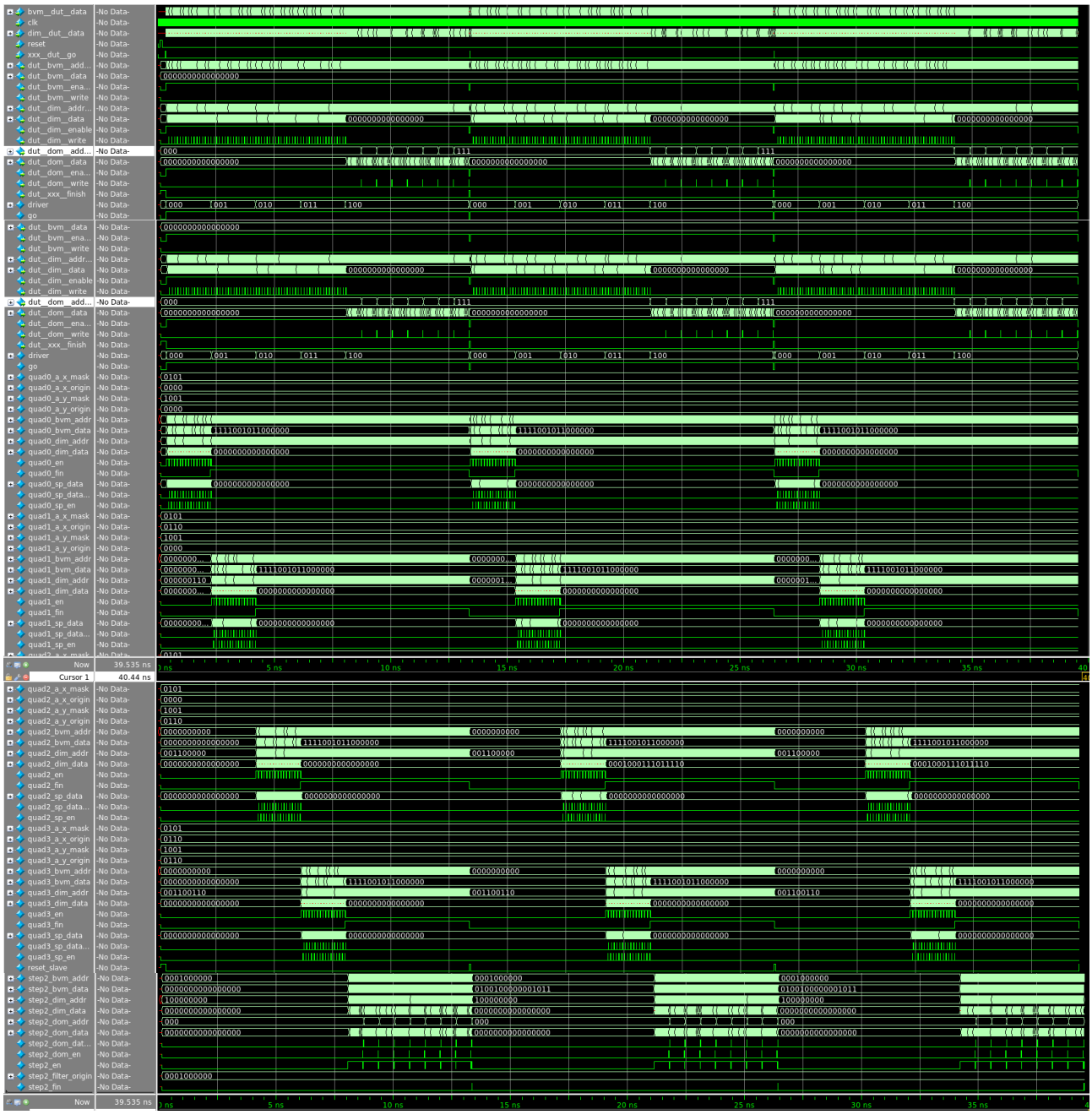

Fig 4.1 Waveform of MyDesign

Though the waveform is very dense and difficult to read, few points to notice about the design are:

- The design is run 3 times as xxx__dut__go is triggered thrice during the entire simulation. Each time ste2_fin signal triggers a reset_slave which resets the entire design and waits for xxx__dut__go signal.
- At any point only one of the 5 blocks quad0, quad1, quad2, quad3 or step2 are active.
- During this time, the quadx_sp_en is triggered several times indicating that intermediate results are being written to scratchpad memory.

- Also, during this time the driver value is constant and changes only after finish signal from each logic block to control the data flow.
- Finally, dut__dom__write is enabled 8 times to indicate that output data is being written 8 times.

To further confirm the correct implementation, the testbench indicates the data being written on to DOM which can been seen below.

```
# @          305: INFO: Start
# @          8745: INFO: Output Memory Write, addr=0
# @          8745: INFO: Output Value for element {0} = 0f90
# @          8745: PASS: Output Memory Write, writing 0f90, expecting 0f90, output status=00000001
# @          9405: INFO: Output Memory Write, addr=1
# @          9405: INFO: Output Value for element {1} = 0650
# @          9405: PASS: Output Memory Write, writing 0650, expecting 0650, output status=00000011
# @          10065: INFO: Output Memory Write, addr=2
# @          10065: INFO: Output Value for element {2} = 1483
# @          10065: PASS: Output Memory Write, writing 1483, expecting 1483, output status=00000111
# @          10725: INFO: Output Memory Write, addr=3
# @          10725: INFO: Output Value for element {3} = 1c7c
# @          10725: PASS: Output Memory Write, writing 1c7c, expecting 1c7c, output status=00001111
# @          11385: INFO: Output Memory Write, addr=4
# @          11385: INFO: Output Value for element {4} = 0000
# @          11385: PASS: Output Memory Write, writing 0000, expecting 0000, output status=00011111
# @          12045: INFO: Output Memory Write, addr=5
# @          12045: INFO: Output Value for element {5} = 0000
# @          12045: PASS: Output Memory Write, writing 0000, expecting 0000, output status=00111111
# @          12705: INFO: Output Memory Write, addr=6
# @          12705: INFO: Output Value for element {6} = 253f
# @          12705: PASS: Output Memory Write, writing 253f, expecting 253f, output status=01111111
# @          13365: INFO: Output Memory Write, addr=7
# @          13365: INFO: Output Value for element {7} = 0000
# @          13365: PASS: Output Memory Write, writing 0000, expecting 0000, output status=11111111
# @          13375: INFO: Done
# @          13375: PASS: Output array status: 11111111
# @          13385: INFO: Start
# @          21825: INFO: Output Memory Write, addr=0
# @          21825: INFO: Output Value for element {0} = 0f90
# @          21825: PASS: Output Memory Write, writing 0f90, expecting 0f90, output status=11111111
# @          22485: INFO: Output Memory Write, addr=1
# @          22485: INFO: Output Value for element {1} = 0650
# @          22485: PASS: Output Memory Write, writing 0650, expecting 0650, output status=11111111
# @          23145: INFO: Output Memory Write, addr=2
# @          23145: INFO: Output Value for element {2} = 1483
# @          23145: PASS: Output Memory Write, writing 1483, expecting 1483, output status=11111111
# @          23805: INFO: Output Memory Write, addr=3
# @          23805: INFO: Output Value for element {3} = 1c7c
# @          23805: PASS: Output Memory Write, writing 1c7c, expecting 1c7c, output status=11111111
# @          24465: INFO: Output Memory Write, addr=4
# @          24465: INFO: Output Value for element {4} = 0000
# @          24465: PASS: Output Memory Write, writing 0000, expecting 0000, output status=11111111
# @          25125: INFO: Output Memory Write, addr=5
# @          25125: INFO: Output Value for element {5} = 0000
# @          25125: PASS: Output Memory Write, writing 0000, expecting 0000, output status=11111111
# @          25785: INFO: Output Memory Write, addr=6
# @          25785: INFO: Output Value for element {6} = 253f
# @          25785: PASS: Output Memory Write, writing 253f, expecting 253f, output status=11111111
# @          26445: INFO: Output Memory Write, addr=7
# @          26445: INFO: Output Value for element {7} = 0000
# @          26445: PASS: Output Memory Write, writing 0000, expecting 0000, output status=11111111
# @          26455: INFO: Done
# @          26455: PASS: Output array status: 11111111
# @          26465: INFO: Start
# @          34905: INFO: Output Memory Write, addr=0
# @          34905: INFO: Output Value for element {0} = 0f90
# @          34905: PASS: Output Memory Write, writing 0f90, expecting 0f90, output status=11111111
# @          35565: INFO: Output Memory Write, addr=1
# @          35565: INFO: Output Value for element {1} = 0650
# @          35565: PASS: Output Memory Write, writing 0650, expecting 0650, output status=11111111
# @          36225: INFO: Output Memory Write, addr=2
# @          36225: INFO: Output Value for element {2} = 1483
# @          36225: PASS: Output Memory Write, writing 1483, expecting 1483, output status=11111111
# @          36885: INFO: Output Memory Write, addr=3
# @          36885: INFO: Output Value for element {3} = 1c7c
# @          36885: PASS: Output Memory Write, writing 1c7c, expecting 1c7c, output status=11111111
# @          37545: INFO: Output Memory Write, addr=4
# @          37545: INFO: Output Value for element {4} = 0000
# @          37545: PASS: Output Memory Write, writing 0000, expecting 0000, output status=11111111
# @          38205: INFO: Output Memory Write, addr=5
# @          38205: INFO: Output Value for element {5} = 0000
# @          38205: PASS: Output Memory Write, writing 0000, expecting 0000, output status=11111111
# @          38865: INFO: Output Memory Write, addr=6
# @          38865: INFO: Output Value for element {6} = 253f
# @          38865: PASS: Output Memory Write, writing 253f, expecting 253f, output status=11111111
# @          39525: INFO: Output Memory Write, addr=7
# @          39525: INFO: Output Value for element {7} = 0000
# @          39525: PASS: Output Memory Write, writing 0000, expecting 0000, output status=11111111
# @          39535: INFO: Done
# @          39535: PASS: Output array status: 11111111
```

We can see that once data starts arriving at cycle 305, it takes until cycle 8745 to write first output data and until cycle 13375 to complete computation on 1 dataset. Hence, this design took 13070 cycles to compute the output. This process repeats as the design has to compute 3 times.

Once simulation results agree with the design specifications, we synthesis the design to generate the netlist using Synopsys DesignCompiler. Fig 4.2 shows the synthesized schematic diagram of the design.

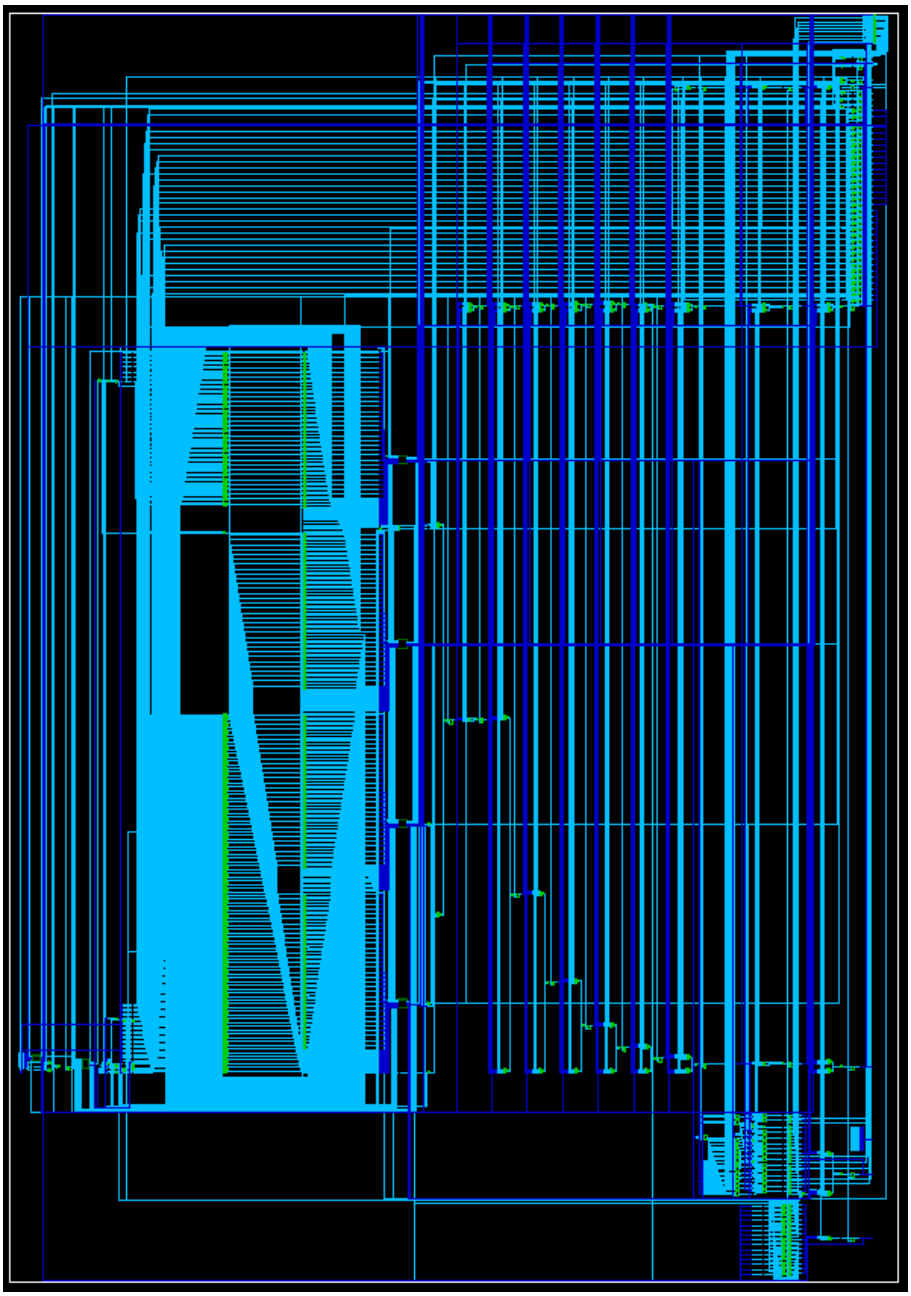

Fig 4.2 Synthesized design schematic

Now, we provide various timing and area constraints based on the standard cell library to generate a optimum design which meets both hold and setup timing at a particular clock. This design is synthesized at 10ns with 4.1ps of available slack.

Once these restrictions are met, we generate the netlist and again simulate it in ModelSim to ensure that synthesis was performed correctly. The generate netlist is provided in the project folder and it was successfully simulated.

5. Verification

A pre-designed test bench was provided to simulate and check the DUT. This test bench provides the required input and monitors the contents of the output memory. Based on the data written to output memory it displays PASS or ERROR. The test bench runs for 3 iterations checks is DUT is continuously reacting to the inputs.

Furthermore, various synthesis based errors and warning are also taken care of during generation of netlist.

6. Results Achieved

```
Operating Conditions: slow   Library: NangateOpenCellLibrary_PDKv1_2_v2008_10_slow_nldm
Wire Load Model Mode: top

  Startpoint: quad2/data_count_reg[2]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: quad2/c_in_reg[8]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Point                                                   Incr       Path
  ---------------------------------------------------------------------------
  clock clk (rise edge)                                 0.0000     0.0000
  clock network delay (ideal)                           0.0000     0.0000
  quad2/data_count_reg[2]/CK (DFF_X2)                   0.0000     0.0000 r
  quad2/data_count_reg[2]/QN (DFF_X2)                   0.4349     0.4349 r
  quad2/U18/ZN (AND2_X2)                                0.1596     0.5946 r
  quad2/U21/ZN (NAND3_X2)                               0.0639     0.6585 f
  quad2/U14/ZN (NAND2_X2)                               0.5463     1.2048 r
  quad2/U8/ZN (INV_X8)                                  0.1580     1.3628 f
  quad2/U134/ZN (AND2_X1)                               0.3061     1.6689 f
  quad2/mac0/A[8] (quad_module_2_DW02_mac_1)            0.0000     1.6689 f
  quad2/mac0/U776/ZN (XNOR2_X2)                         0.9014     2.5703 r
  quad2/mac0/U840/ZN (NAND2_X2)                         0.4286     2.9989 f
  quad2/mac0/U1062/ZN (OAI22_X2)                        0.2957     3.2946 r
  quad2/mac0/U342/S (FA_X1)                             0.8018     4.0964 f
  quad2/mac0/U339/CO (FA_X1)                            0.5985     4.6950 f
  quad2/mac0/U332/S (FA_X1)                             0.7993     5.4943 r
  quad2/mac0/U331/S (FA_X1)                             0.6887     6.1830 f
  quad2/mac0/U1031/ZN (OR2_X1)                          0.2901     6.4731 f
  quad2/mac0/U814/ZN (INV_X4)                           0.0690     6.5421 r
  quad2/mac0/U813/ZN (OAI21_X2)                         0.0720     6.6141 f
  quad2/mac0/U815/ZN (INV_X4)                           0.0803     6.6944 r
  quad2/mac0/U1003/ZN (OAI21_X2)                        0.0703     6.7646 f
  quad2/mac0/U792/ZN (AOI21_X4)                         0.1525     6.9171 r
  quad2/mac0/U1017/ZN (OAI21_X2)                        0.1024     7.0195 f
  quad2/mac0/U831/ZN (AOI21_X2)                         0.3524     7.3719 r
  quad2/mac0/U798/ZN (OAI21_X2)                         0.1315     7.5033 f
  quad2/mac0/U794/ZN (AOI21_X2)                         0.2196     7.7230 r
  quad2/mac0/U1049/ZN (OAI21_X2)                        0.1330     7.8559 f
  quad2/mac0/U53/CO (FA_X1)                             0.4855     8.3414 f
  quad2/mac0/U51/Z (XOR2_X2)                            0.4122     8.7536 f
  quad2/mac0/MAC[31] (quad_module_2_DW02_mac_1)         0.0000     8.7536 f
  quad2/U36/ZN (INV_X4)                                 0.1094     8.8630 r
  quad2/U16/ZN (NAND2_X2)                               0.0995     8.9625 f
  quad2/U26/ZN (OAI22_X2)                               0.3744     9.3369 r
  quad2/U236/ZN (INV_X4)                                0.0403     9.3771 f
  quad2/U237/ZN (OAI22_X2)                              0.3009     9.6780 r
  quad2/c_in_reg[8]/D (DFF_X2)                          0.0000     9.6780 r
  data arrival time                                                9.6780

  clock clk (rise edge)                                10.0000    10.0000
  clock network delay (ideal)                           0.0000    10.0000
  clock uncertainty                                    -0.0500     9.9500
  quad2/c_in_reg[8]/CK (DFF_X2)                         0.0000     9.9500 r
  library setup time                                   -0.2679     9.6821
  data required time                                               9.6821
  ---------------------------------------------------------------------------
  data required time                                               9.6821
  data arrival time                                               -9.6780
  ---------------------------------------------------------------------------
  slack (MET)                                                      0.0041
```

Fig 6.1 Timing Report

```
    NangateOpenCellLibrary_PDKv1_2_v2008_10_slow_nldm (File: /afs/eos.ncsu.edu/lockers/research/ece/wdavis/tech/nangate/NangateOpenCellLibrary_PDKv1_2_v2008_10/liberty/520/
NangateOpenCellLibrary_PDKv1_2_v2008_10_slow_nldm.db)

Number of ports:                         1133
Number of nets:                          8239
Number of cells:                         6100
Number of combinational cells:           5417
Number of sequential cells:               670
Number of macros/black boxes:               0
Number of buf/inv:                        828
Number of references:                      30

Combinational area:               8384.586076
Buf/Inv area:                      443.156004
Noncombinational area:            2642.976059
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                 11027.562135
Total area:                 undefined
1
```

Fig 6.2 Area Report

```
Library(s) Used:

    NangateOpenCellLibrary_PDKv1_2_v2008_10_slow_nldm (File: /afs/eos.ncsu.edu/lockers/research/ece/wdavis/tech/nangate/NangateOpenCellLibrary_PDKv1_2_v2008_10/liberty/520/
NangateOpenCellLibrary_PDKv1_2_v2008_10_slow_nldm.db)


Operating Conditions: slow   Library: NangateOpenCellLibrary_PDKv1_2_v2008_10_slow_nldm
Wire Load Model Mode: top


Global Operating Voltage = 0.95
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 327.4798 uW   (93%)
  Net Switching Power  =  23.2236 uW    (7%)
                         ---------
Total Dynamic Power    = 350.7034 uW  (100%)

Cell Leakage Power     =  61.9236 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register           0.2962        4.4156e-03          4.4237e+06            0.3051  (  73.94%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational   3.1243e-02        1.8808e-02          5.7500e+07            0.1076  (  26.06%)
--------------------------------------------------------------------------------------------------
Total              0.3275 mW     2.3224e-02 mW     6.1924e+07 pW          0.4126 mW
1
```

Fig 6.3 Power Report

- As expected the path through one of the MAC unit is the critical path.
- The fastest possible clock for the DUT is 10 ns.
- The device takes 13070 cycles to complete the execution and write the output vectors to output memory.
- The area generated after synthesis is: 11027.562135 um^2
- Total Power consumed by the DUT: 412.6 uW

7. Conclusion

A hardware implementation of a basic 3 layer CNN is designed and synthesized. Possible optimizations include using pipelined MAC unit to compute data simultaneously. This will drastically reduce the area and also improve timing.