Final Project Report First Page.
Multi-precision Artificial Neural Network

Name: Kashyap Ravichandran
Unityid: 200320511
StudentID:. Kravich2

Delay (ns to run provided example).
Clock period: 6.9ns
# cycles”: 9620

Logic Area: (um^2)

2334.41

Memory: N/A

1/(delay.area) (ns^{-1}.um^{-2})
1.549535 x 10^8

Delay (TA provided example. TA to complete)

1/(delay.area) (TA)

Abstract

Artificial Neural Networks are increasingly becoming common to solve complex problem. Artificial Neural Networks are made up of basic blocks that multiply the given input ‘I’ with a weight ‘W’ and added with a bias ‘B’ to form output ‘O’. We represent the weights using a n x n weight matrix and the input values using a n x 1 input matrix. We ignore the bias in this project to create an output matrix of size n x 1 which is formed by multiplying the weight matrix with the input matrix. This project aims to suggest a hardware function that does this. The function that is implemented uses a couple of 8-bit x 8-bit multiplier, and a couple of adders to have a through put of 2 Multiply and Accumulate per cycle. The clock period achieved was around 6.9ns and the area of the hardware function was around 2334 um^2.

Multi-precision Artificial Neural Network

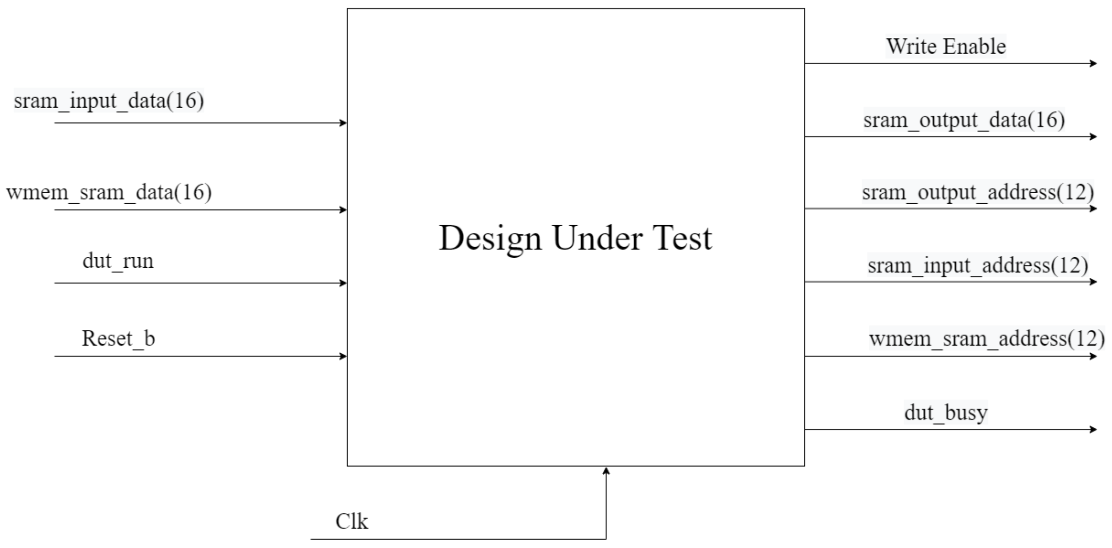
Kashyap Ravichandran

Abstract

Artificial Neural Networks are increasingly becoming common to solve complex problem. Artificial Neural Networks are made up of basic blocks that multiply the given input ‘I’ with a weight ‘W’ and added with a bias ‘B’ to form output ‘O’. We represent the weights using a n x n weight matrix and the input values using a n x 1 input matrix. We ignore the bias in this project to create an output matrix of size n x 1 which is formed by multiplying the weight matrix with the input matrix. This project aims to suggest a hardware function that does this. The function that is implemented uses a couple of 8-bit x 8-bit multiplier, and a couple of adders to have a through put of 2 Multiply and Accumulate per cycle. The clock period achieved was around 6.9ns and the area of the hardware function was around 2334 um^2.

Introduction

As mentioned earlier, an artificial neural network can be mathematically modelled as a matrix multiplier creating a n x 1 output matrix. In this project, we can vary the precision levels, by changing the number of bits that are required to represent the input or the weight. The output of the function implemented is always 16 bits. The weights and inputs are packed in a 16-bit SRAM line such that you can have 2 8-bit values or 4 4-bit value or 8 2-bit values in a line. This means that at least we need to multiply 4 bits in a cycle to make sure that we can have a MAC output every cycle. More on this will be explained in micro-architecture section. The major objective of this project was to optimize the delay*area product. An outcome of this was understanding the different tradeoffs between the clock period, number of cycles and the area of the function. An overall input/output of the design is as shown below:

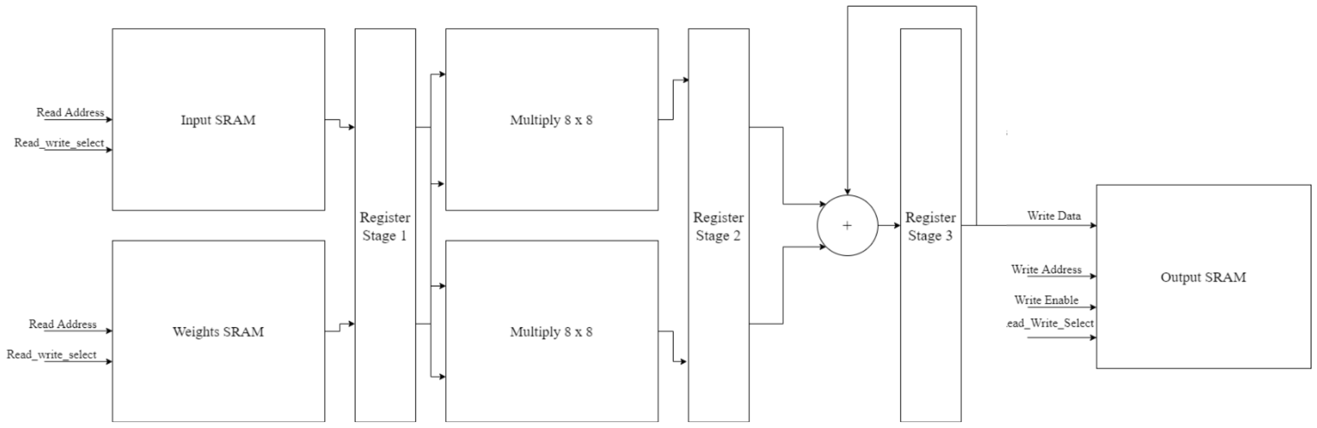


The report shows how the above drawn block does the function mentioned, key components of the system, ways to make sure that you get two MACs every cycle, and not lose any cycles with the housekeeping. The report also captures the synthesis reports, and the errors and report on why I think they arise and talk briefly on few key things that was done to reduce the area.

Microarchitecture and Technical Implementation

Datapath

The microarchitecture is pipelined to make sure that the systems produces two MAC outputs every cycle. A simplified data path of the microarchitecture is as shown below:

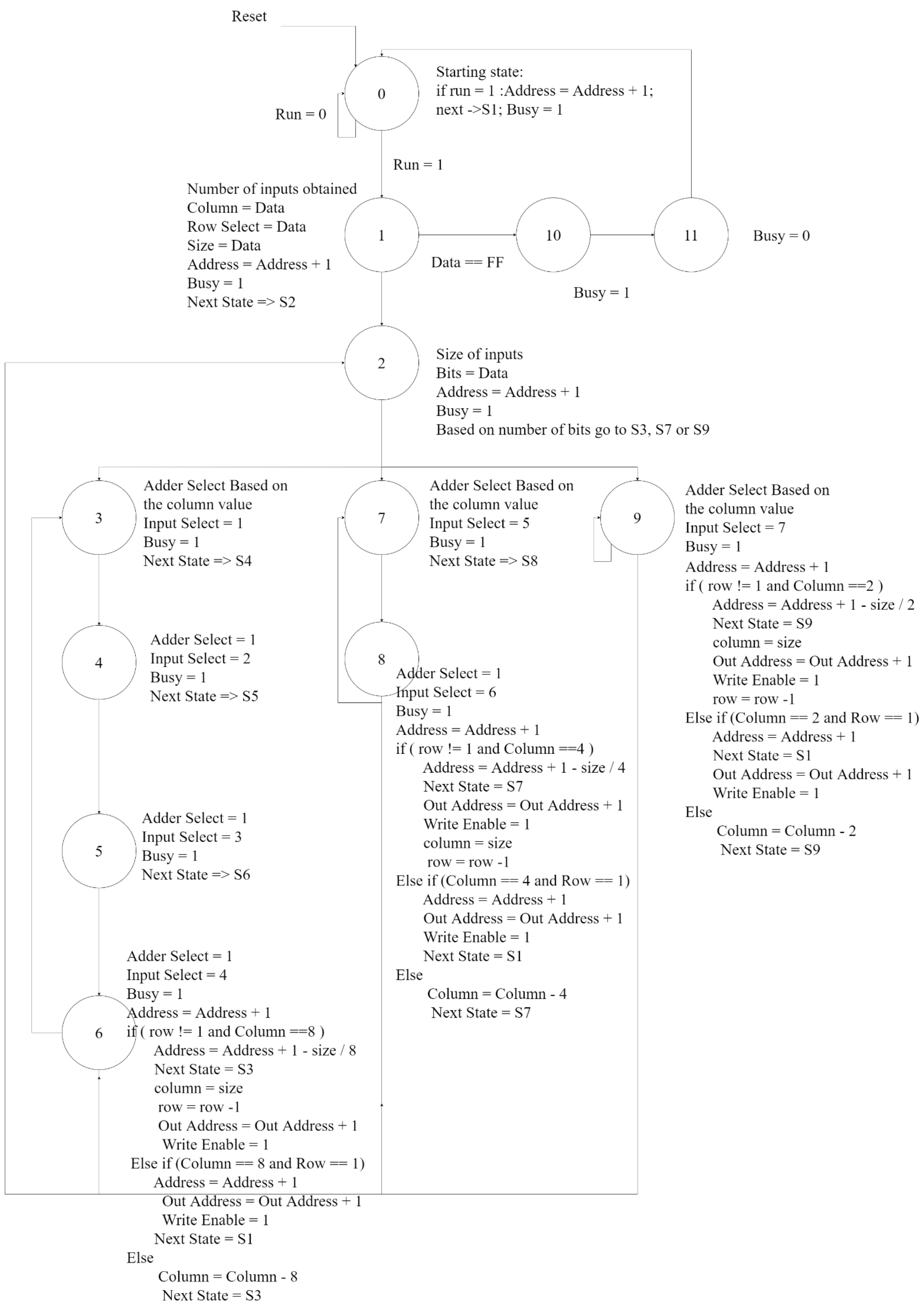


We can divide the data path into 4 stages. Let the SRAM cells be the first stage, the multiplying units be stage 2, the adder be stage 3 and writing the data back to SRAM be stage 4.

- We send the address for the input and the weight SRAM and we get the data the next cycle. With this data we always decide the inputs bits and the weight bits that are to be multiplied, the select signal to the adder mux, and the write enable signal for this data. With this control data, the inputs, and the weights are latched on to Register Stage 1 flip flops.
- Using the control signals we decide which bits from the input and the weight data are to be multiplied. The multiplication is done using 2 8-bit x 8-bit multipliers. The reason we have two multipliers is because we can perform at least two multiplication per cycle. The main constraints for this system are the data provided by the SRAMs. Having 2 multipliers would give us the opportunity to multiply the least number of inputs/weights that is given by the SRAMs. If the number of inputs/weights packed in a 16 bits line is 8/4, the data is kept at the same stage until all the data is multiplied. The output from the multiplier is 16 bits wide. At the next clock cycle two 16-bit outputs are latched on to the second stage register. The adder control signal and the write enable signals are latched from the previous register stage (Register Stage 1).
- The third stage in the microarchitecture has two adders, adding the two output from the multipliers that were latched onto register 2. This done using the first 17-bit adder. The second adder basically adds this value with the value stored in register stage 3 or with zero. This selection between register stage 3 and zero is made by adder control signal.
- At the fourth stage, we send the data we have to the output SRAM. The output is only written when the write enable line is asserted. The enable line is given by the control logic and is transferred from one pipeline stage to another.
- Once the enable line is asserted, the output is put in the SRAM in the next clock cycle.

Control Path

We have two FSM in the control path, one for the input and one for weights. The input FSM also takes care of output address register and the write enable line. Both FSM look very similar and the two FSMs are given in the following page. The FSMs given below are self-explanatory. The number of states can be reduced with another loop and register. But having an 11 state FSM makes a lot of control lines simpler and reduces the need to use a 4-bit register. The overall microarchitecture is shown after that. A point to note that all the select lines to the muxes are provided by the control path.



```
Starting state:
if run = 1 :Address = Address + 1;
next ->S1; Busy = 1
```

```
Number of inputs obtained
Column = Data
Row Select = Data
Size = Data
Address = Address + 1
Busy = 1
Next State => S2
```

```
Size of inputs
Bits = Data
Address = Address + 1
Busy = 1
Based on number of bits go to S3, S7 or S9
```

```
Adder Select Based on
the column value
Input Select = 1
Busy = 1
Next State => S4
```

```
Adder Select = 1
Input Select = 2
Busy = 1
Next State => S5
```

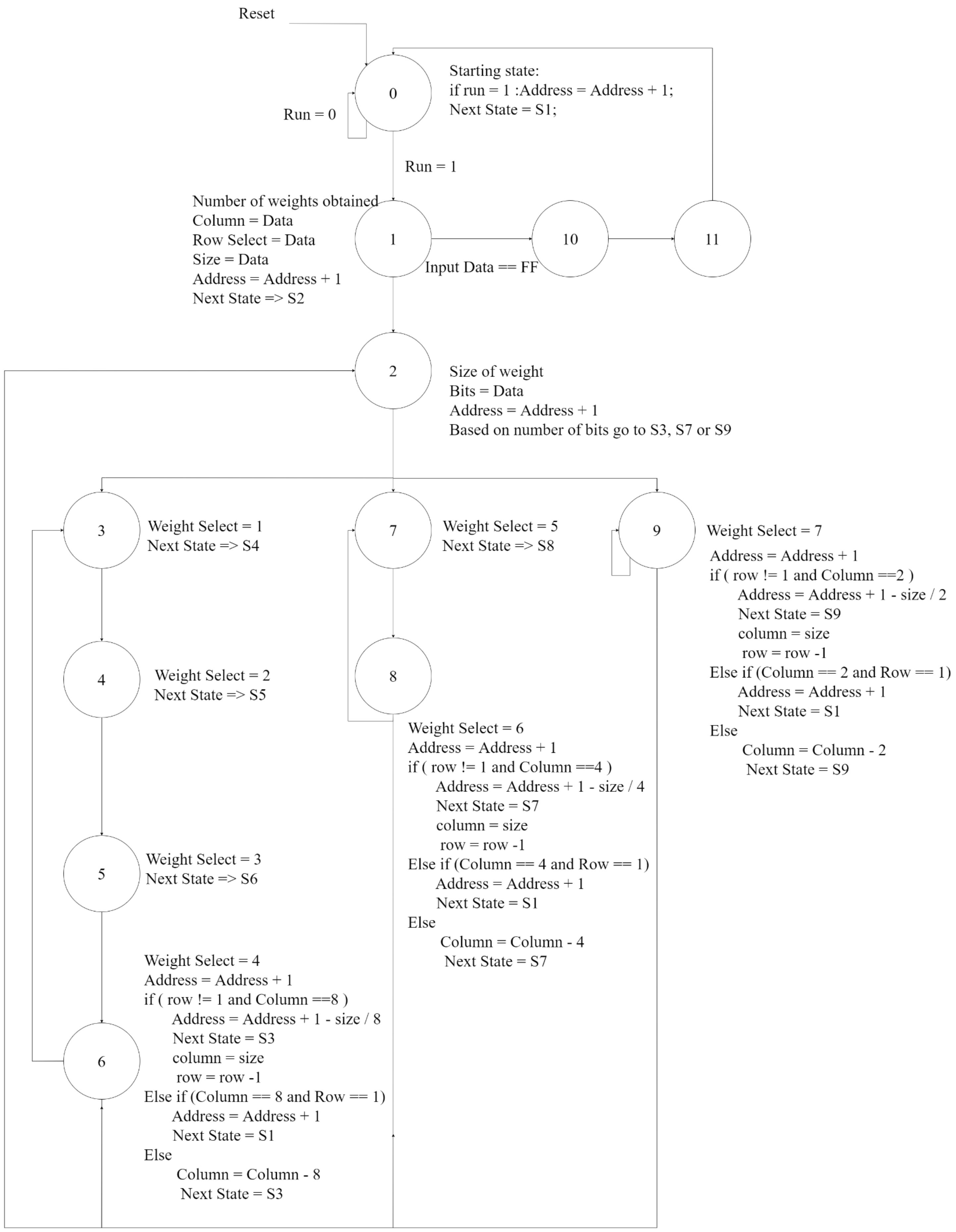
```
Adder Select = 1
Input Select = 3
Busy = 1
Next State => S6
```

```
Adder Select = 1
Input Select = 4
Busy = 1
Address = Address + 1
if ( row != 1 and Column ==8 )
      Address = Address + 1 - size / 8
      Next State = S3
      column = size
       row = row -1
       Out Address = Out Address + 1
        Write Enable = 1
 Else if (Column == 8 and Row == 1)
      Address = Address + 1
        Out Address = Out Address + 1
        Write Enable = 1
      Next State = S1
Else
       Column = Column - 8
        Next State = S3
```

```
Adder Select Based on
the column value
Input Select = 5
Busy = 1
Next State => S8
```

```
Adder Select = 1
Input Select = 6
Busy = 1
Address = Address + 1
if ( row != 1 and Column ==4 )
      Address = Address + 1 - size / 4
      Next State = S7
      Out Address = Out Address + 1
      Write Enable = 1
      column = size
       row = row -1
Else if (Column == 4 and Row == 1)
      Address = Address + 1
      Out Address = Out Address + 1
      Write Enable = 1
      Next State = S1
Else
       Column = Column - 4
        Next State = S7
```

```
Adder Select Based on
the column value
Input Select = 7
Busy = 1
Address = Address + 1
if ( row != 1 and Column ==2 )
      Address = Address + 1 - size / 2
      Next State = S9
      column = size
      Out Address = Out Address + 1
      Write Enable = 1
      row = row -1
Else if (Column == 2 and Row == 1)
      Address = Address + 1
      Next State = S1
      Out Address = Out Address + 1
      Write Enable = 1
Else
       Column = Column - 2
        Next State = S9
```



```
Starting state:
if run = 1 :Address = Address + 1;
Next State = S1;
```

```
Number of weights obtained
Column = Data
Row Select = Data
Size = Data
Address = Address + 1
Next State => S2
```

```
Size of weight
Bits = Data
Address = Address + 1
Based on number of bits go to S3, S7 or S9
```

```
Weight Select = 1
Next State => S4
```

```
Weight Select = 2
Next State => S5
```

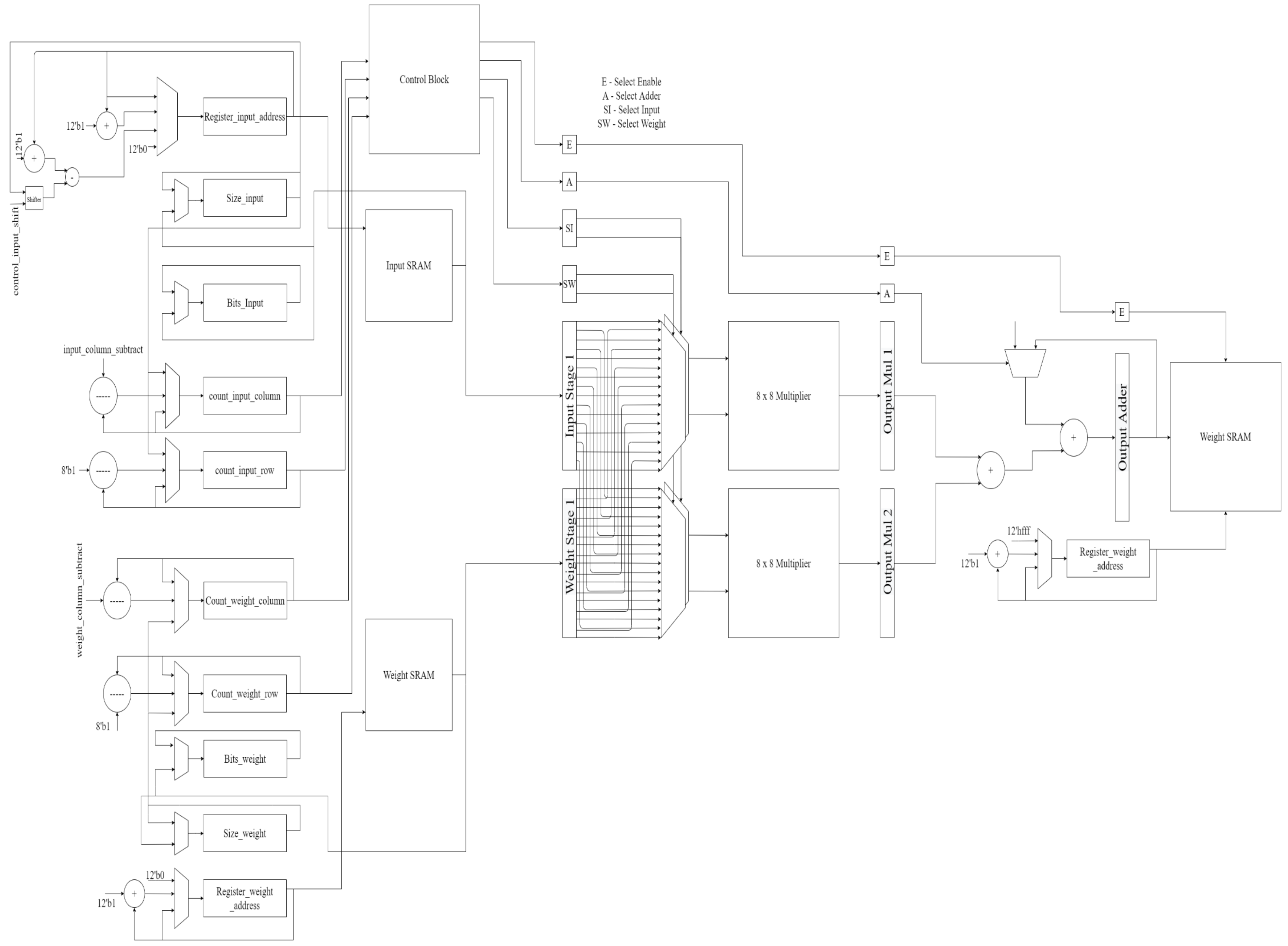
```
Weight Select = 3
Next State => S6
```

```
Weight Select = 4
Address = Address + 1
if ( row != 1 and Column ==8 )
      Address = Address + 1 - size / 8
      Next State = S3
      column = size
       row = row -1
Else if (Column == 8 and Row == 1)
      Address = Address + 1
      Next State = S1
Else
       Column = Column - 8
        Next State = S3
```

```
Weight Select = 5
Next State => S8
```

```
Weight Select = 6
Address = Address + 1
if ( row != 1 and Column ==4 )
      Address = Address + 1 - size / 4
      Next State = S7
      column = size
       row = row -1
Else if (Column == 4 and Row == 1)
      Address = Address + 1
      Next State = S1
Else
       Column = Column - 4
        Next State = S7
```

```
Weight Select = 7
Address = Address + 1
if ( row != 1 and Column ==2 )
      Address = Address + 1 - size / 2
      Next State = S9
      column = size
       row = row -1
Else if (Column == 2 and Row == 1)
      Address = Address + 1
      Next State = S1
Else
       Column = Column - 2
        Next State = S9
```


control_input_shift
Shifter
12'b1
12'b1
12'b0
Register_input_address
Size_input
Bits_Input
input_column_subtract
count_input_column
8'b1
count_input_row
weight_column_subtract
Count_weight_column
Count_weight_row
8'b1
Bits_weight
Size_weight
12'b0
12'b1
Register_weight_address
Control Block
Input SRAM
Weight SRAM
E - Select Enable
A - Select Adder
SI - Select Input
SW - Select Weight
E
A
SI
SW
Input Stage 1
Weight Stage 1
8 x 8 Multiplier
8 x 8 Multiplier
Output Mul 1
Output Mul 2
E
A
E
Output Adder
Weight SRAM
12'hfff
12'b1
Register_weight_address

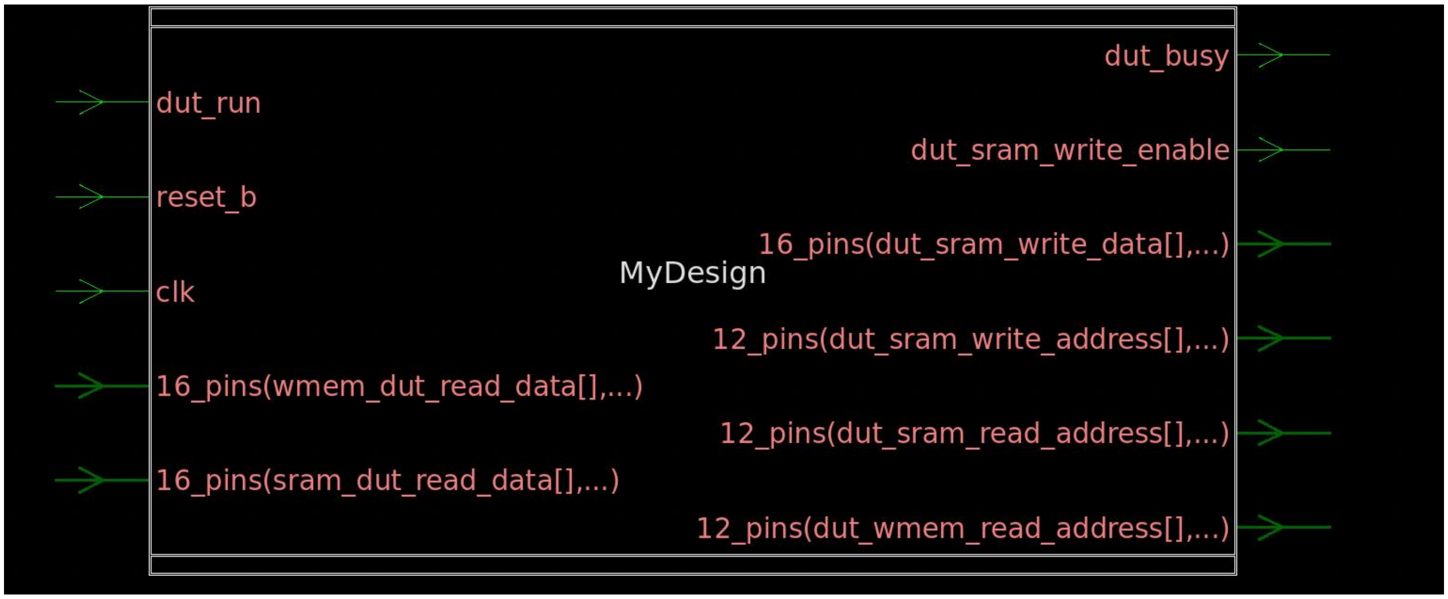
Input and Weight Selection

The most important reason behind choosing a 11 state FSM is to make the choosing input and the weight bits to multiply. Using an 8 to 1 mux at each multiplier for the input and weight data. The table drawn below gives us an understanding on which bits are selected to be given as input to the multipliers. When 2/4 bits are selected, the value is sign extended to form an 8-bit data that is fed to the multipliers.

Input Select	Mul1_Input	Mul2_Input	Weight Select	Mul1_Weight	Mul2_weight
0	8’b0	8’b0	0	8’b0	8’b0
1	[1:0]	[3:2]	1	[1:0]	[3:2]
2	[5:4]	[7:6]	2	[5:4]	[7:6]
3	[9:8]	[11:10]	3	[9:8]	[11:10]
4	[13:12]	[15:14]	4	[13:12]	[15:14]
5	[3:0]	[7:4]	5	[3:0]	[7:4]
6	[11:8]	[15:12]	6	[11:8]	[15:12]
7	[7:0]	[15:8]	7	[7:0]	[15:8]

As mentioned earlier, the select lines to all the muxes comes from the control block. The control lines and the muxes help us store the correct values at the correct registers. So instead of using state-based mux to set the address registers, we get a control line from the state machine which helps us reduce the size of the mux used. Another important point to note here is that the address output line from the mux is fed to the input and the weight SRAM instead of the output from the address registers, this helps us get the data at the very next clock cycle. This simplifies the state machine, and it keeps the flow of data properly. This makes sure that we have 2 MAC outputs every cycle.

Interface Specification


dut_busy
dut_run
dut_sram_write_enable
reset_b
16_pins(dut_sram_write_data[],...)
MyDesign
clk
12_pins(dut_sram_write_address[],...)
16_pins(wmem_dut_read_data[],...)
12_pins(dut_sram_read_address[],...)
16_pins(sram_dut_read_data[],...)
12_pins(dut_wmem_read_address[],...)

The system has 2 16-bit lines carrying the input and weight sram data, 1 reset line, 1 dut_run to notify the system to run a set of input and weights, a reset line to bring the system to a known state and a clock line. The outputs from the hardware are 3 12-bit address lines, one for input, one for output and one for weight. There is a 16-bit data line that carries the data to the output SRAM. There are two single bit line one to let the testbench know that the hardware is still calculating the outputs, and another line to denote that the output SRAM should write the data on the output data line to the address specified by the output address line. The interfaces are tabulated down below:

DUT Interface:

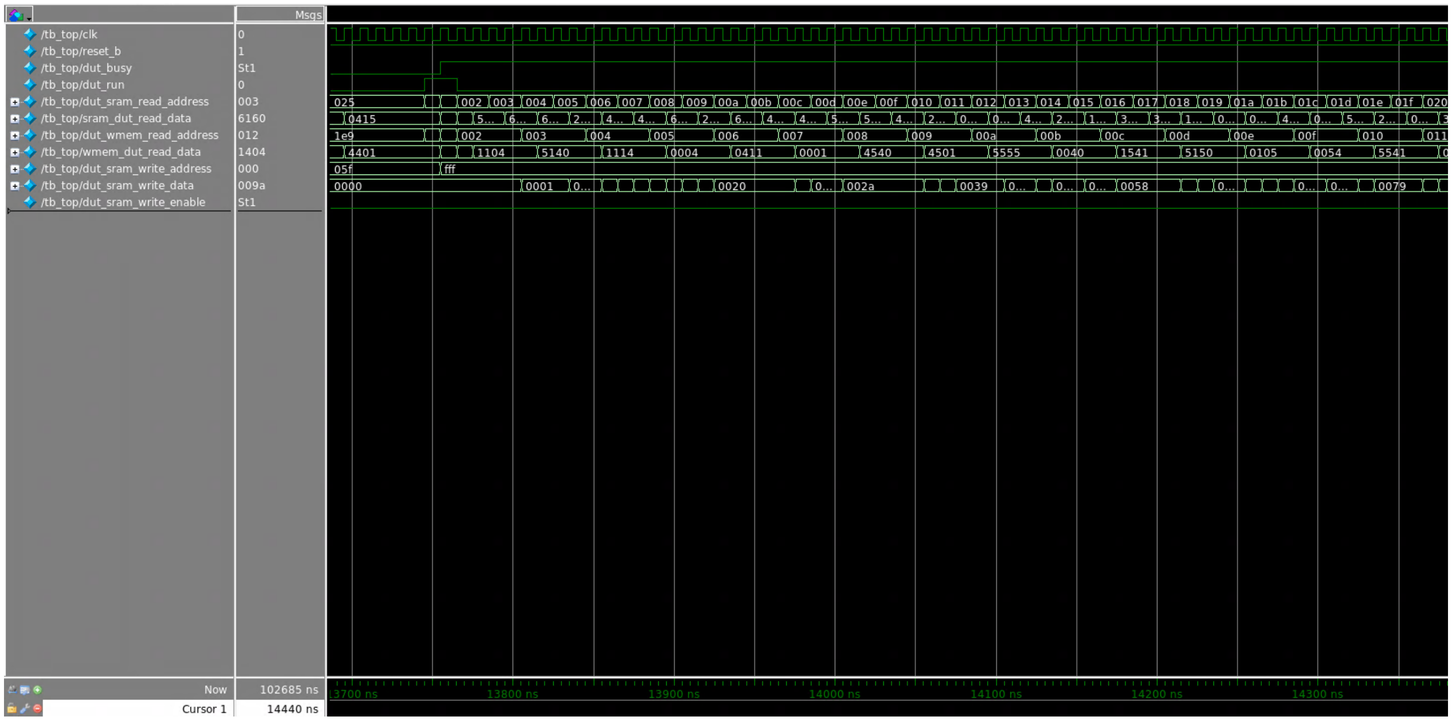
Port	**Type**	**Width**	**Function**
Dut_sram_read_data	input	16	To carry a line from the input SRAM to the design under test.
Wmem_sram_read_data	input	16	To carry a line from the weight SRAM to the design under test
Dut_run	Input	1	To notify the design under test to start computation
Reset_b	Input	1	To bring the two FSM to a known state.
Clk	Input	1	Provides the clock to the design under test
Dut_busy	Output	1	To notify the testbench that the DUT is computing for a given set of inputs and weights
Dut_sram_read_Address	Output	12	To provide the input SRAM with an address
Dut_sram_write_Address	Output	12	To provide the output SRAM with an address
Wmem_sram_read_address	Output	12	To provide the weight SRAM with an address
Dut_sram_write_data	Output	16	Carries the data that needs to write in the output SRAM
Write_Enable	Output	1	To notify the output SRAM to write the data carried by the output data line to the output address line.

Verification

Each weight and input present in problem 1 of the first input/weight file was manually multiplied and the intermediate values at each register stage and the outputs from the multiplier and the adders were verified with the manual calculation. The state machine was first manually simulated by hand and was verified by simulating the design. The simulation results and the synthesis results are reported in the following sections.

Simulation and Synthesis

The RTL was simulated using modelsim. The RTL is further synthesized using synopsys. Modifying our class script to take pipelines into account we obtain a clock of 5.5ns which does not violate the first setup violation check. However, with the second violation check, which we do after checking for hold violation shows us that the timing is not met. Increasing the clock to 6.9ns shows us that with the first setup violation met we also make sure that the hold violation and the second setup violation are also met. The timing reports and the area reports are present in the result folder. We have couple of unsigned to signed conversion warning. This arises because we have a couple of comparators that run on the raw data coming from the input SRAM. This is done mainly to move from S1 to S10 or to move from S2 to any one of the following states S3, S7 and S9. Apart from these we got OPT-106 and a couple of other warning. A point to note here is that all the warning that we obtained from synthesis were ignorable warnings. The waveform obtained from synthesis is presented in the next page.


```
/tb_top/clk                     0
/tb_top/reset_b                 1
/tb_top/dut_busy                St1
/tb_top/dut_run                 0
/tb_top/dut_sram_read_address   003
/tb_top/sram_dut_read_data      6160
/tb_top/dut_wmem_read_address   012
/tb_top/wmem_dut_read_data      1404
/tb_top/dut_sram_write_address  000
/tb_top/dut_sram_write_data     009a
/tb_top/dut_sram_write_enable   St1

Now        102685 ns
Cursor 1   14440 ns
```

Results Achieved and Key Inference

The clock period achieved with the compile_ultra -incremental iterative step was around 6.7ns. However, with the just compile_ultra we were not able to meet the timing requirements. I attached a ddc file which showed me that the clock with 6.7nS met the timing requirements.

The number of cycles that was required to complete the two-input files was found out to be 9620 cycles. A major reason why the cycle count is high can be attributed to the fact that we can get at most only 2 MACs per cycle. With 8-bit inputs/weight this does not matter but if the there are only 4-bits or 2-bits input/weight, this reduces the potential number of multiplication accumulator that we can get in a single cycle.

Few Key points were made in microarchitecture to reduce the number of muxes required for registers like register_input_address, register_otuput_address, register_weight_address, count_column_input, so and so forth. Instead of using a 16 to 1 mux to directly supply to value to be stored, we provide the select line thus reduce the size of the muxes that are required. With only 2 8-bit x 8-bit multipliers, instead of having multiple 2-bit x 2-bit multipliers, as mentioned in the bit fusion paper. Having 2-bit multiplier instead of 8-bit multipliers would also include huge steering networks and special networks to take care of sign. Having just 8-bit multipliers helps us reduce the area which offsets the high cycle count. We get an area of 2334.41 um^2.

- **Cycle Count:** 6.9nS
- **Clock Period:** 9620
- **Area:** 2334.41 um^2
- **Delay:** Cycle Count x Clock Period = 66.378 uS

Conclusion

The importance of ANN for modern computing shows us the important reason why we need to a hardware method to speed up calculation. The project shows a way to have 2 8-bit multipliers, and 2 serial adders to create a matrix multiplier that can effectively multiply an n x n weight matrix with a n x 1 input matrix to produce a n x 1 output matrix. The intermediate state was verified manually to make sure that the hardware function was operating the way it was intended. Few key decisions were made to reduce the area at the cost of number of cycles. The given method provides us a useful way to multiply the input and the weight matrices such that the delay-area product is as small as possible.