COL215 Assignment 2 Stage 2

Rajat Bhardwaj 2020CS50436
Geetansh Juneja 2020CS50649

October 9, 2022

1 Design Decisions

1.1 Shifter

Removed the clock from shifter to make it a combinational circuit because the delay caused due to it is very less compared to the clock cycle, so it doesn't require whole cycle to shift the input.

2 Control Logic

There are 2 concurrent Finite State Machine(FSM). FSM1 controls the reading of image inputs from global memory(ROM) to local memory(RAM). FSM2 controls data inputs from RAM and ROM and computes Hidden Layer 1 and Final layer.

2.1 FSM 1

FSM1 reads 784 image inputs from ROM to RAM. We only read image inputs from ROM to RAM because computation for hidden layer 1 require access of same image inputs multiple times, which improves hardware utilization and improve performance with more parallelism. .There are total 2 states in fsm1 $s0\_fsm1$ and $s1\_fsm1$. The Explanation of States is as follows:

2.1.1 `s0_fsm1`

In this state Image data is read from ROM port. This state also keeps track how many image inputs have been read from ROM and written in RAM which is stored in integer signal **current**. If current is less than 784 it transitions to `s1_fsm1` otherwise it remains in this state.

2.1.2 `s1_fsm1`

In this state read Image data is written in RAM by making RAM **we** signal 1. This state also increments **current** by 1 indicating that an image data input is written in RAM. Since Image data is 8 bits before writing in RAM we extend it 16 bits by concatinating "00000000" or "11111111" at front according to 8th bit of the Finally this state transitions to `s0_fsm1`.

2.2 FSM 2

We have declared 2 instantiations of RAM. RAM1 is used to store 784 image inputs and RAM2 stores hidden layer 1 and final layer outputs. This FSM works in parallel with FSM1. This FSM is responsible for reading image inputs from RAM1 and corresponding weights and biases from ROM for hidden layer 1 computation. Similarly, after computing hidden layer 1 in RAM1 it reads hidden layer 1 and corresponding weights and biases from ROM and outputs the final layer in RAM2 from address 0 to 9. After computing Final layer, this FSM also computes class number which has maximum value. States `s1_fsm2`, `s2_fsm2`, `s3_fsm2` are used for computation of hidden layer 1. States `s4_fsm2`, `s5_fsm2` and `s6_fsm2` is used for computation of final layer. State `s7_fsm2` is used to compute output class number.This FSM has total 7 states which are described as follows:

2.2.1 `s1_fsm2`

We keep track of weight address using signal `rom_addr` and image address using signal `ram_addr`. In this state we increment above signals by 1 so that we can move to next weight and image input. In this state we also keep track of how many hidden layer elements are computed by using signal `hidden_layer_addr`. If we have computed total 64 elements indicating that we have computed whole hidden layer 1 then we make transition to state `s4_fsm2`. If the above condition is not satisfied we check if current=783, if it is then we transition to state `s2_fsm2` else we remain in this state.

2.2.2 `s2_fsm2`

In this state control signal of mac is made 1 so that mac further does not accumulate more weight*image because to calculate an element of hidden layer 1 $\sum_{n=0}^{783} weight * image$ should be done only 784 times. This state makes signal current to 0. It also sets `ram_addr` to 0 so that image inputs can again be read from starting from RAM1. It also increments `rom_bias_addr` by 1. Then this State Transitions to `s3_fsm2`.

2.2.3 `s3_fsm2`

This state adds bias corresponding to the computed value by mac and stores it in the address `hidden_layer_addr` of RAM2 and finally incrementing `hidden_layer_addr` by 1. Then this State Transitions to `s1_fsm2`.

2.2.4 `s4_fsm2`

Now we have computed hidden layer 1. We keep track of weight address using signal `rom_addr`, hidden layer 1 input address using signal `hidden_layer_1_addr`. In this state we increment above signals by 1 so that we can move to next weight and hidden layer input. In this state we also keep track of how many final layer elements are computed by using signal `final_layer_addr`. If we have computed total 10 elements indicating that we have computed whole final layer then we make transition to state `s7_fsm2`. If the above condition is not satisfied we check if current=64, if it is then we transition to state `s5_fsm2` else we remain in this state.

2.2.5 `s5_fsm2`

In this state control signal of mac is made 1 so that mac further does not accumulate more weight*image because to calculate an element of final layer $\sum_{n=0}^{64} weight * \texttt{hidden layer}$ should be done only 64 times. This state makes signal current to 0. It also sets `ram_addr` to 0 so that hidden layer inputs can again be read from starting from RAM2. It also increments `rom_bias_addr` by 1.Then this State Transitions to `s6_fsm2`.

2.2.6 `s6_fsm2`

This state adds bias corresponding to the computed value by mac and stores it in the address `final_layer_addr` of RAM2 and finally incrementing `hidden_layer_addr` by 1. Then this State Transitions to `s1_fsm2`.

2.2.7 `s7_fsm2`

In this state we compute output class number. In each call of this state we check if value at answer index is less than current address value. If so we update the answer index.

3 Schematic

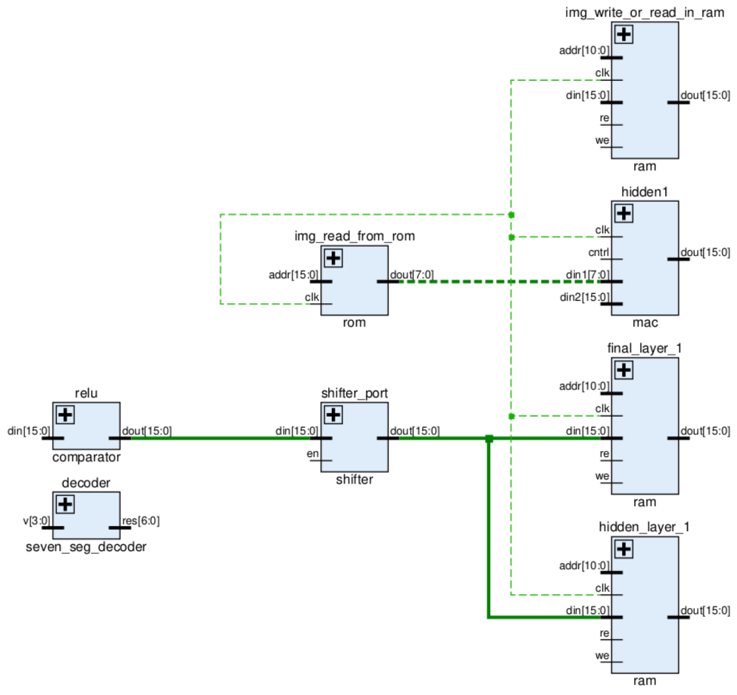
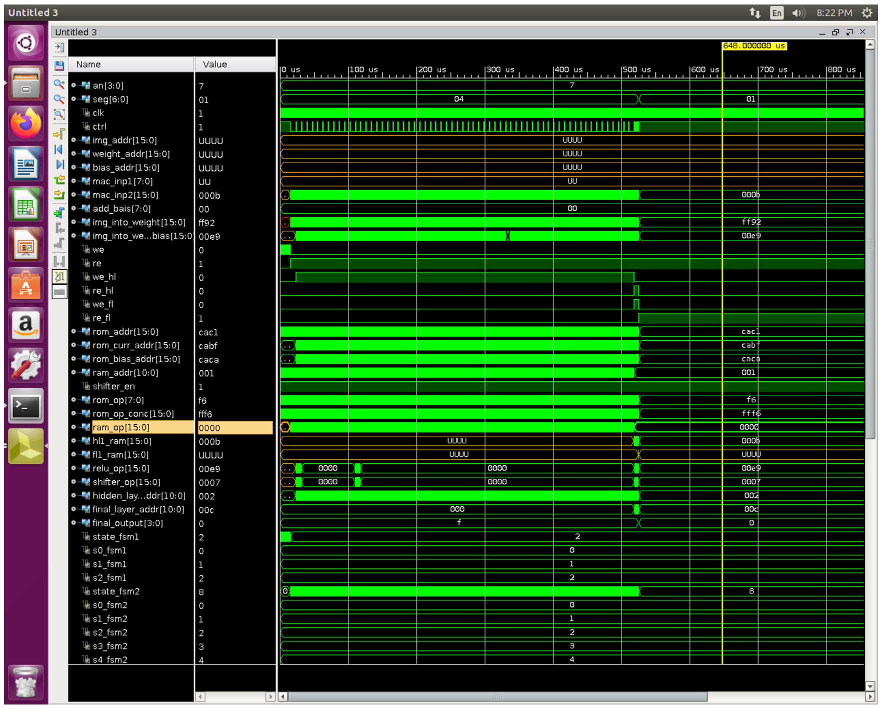


Figure 1: Data path

The explanation of the data path is as follows.
First the image is stored in ram(img_write_or_read_in_ram) from rom. Then ram is connected to mac, the mac takes weight and image from the ram and multiplies them and then the bias gets added manually and then the result is sent to comparator that performs the relu operation and then this is sent to shifter and then the result is stored in in ram (hidden_layer_1). Again, for the final layer same thing happens and the result is stored in ram (final_layer_1).

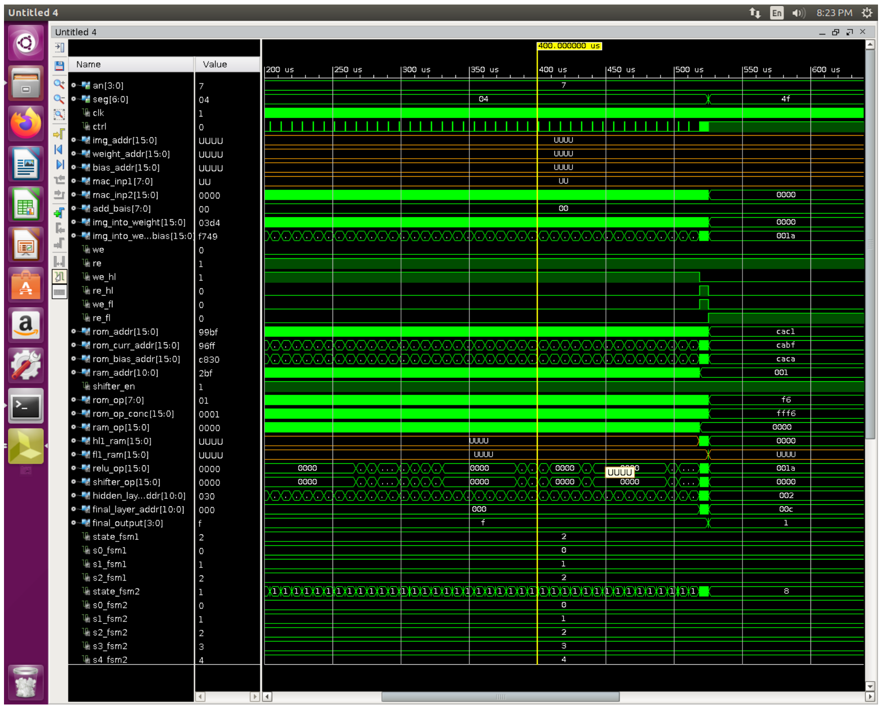
4 Simulation Screenshots

The output class number can be seen in the signal `final_output`.



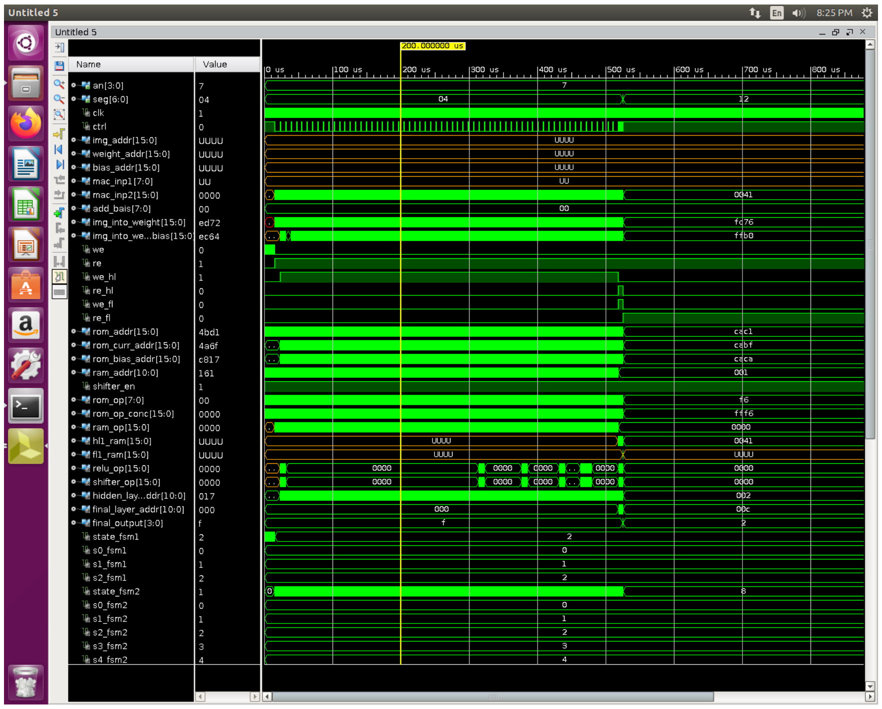
Name	Value
an[3:0]	7
seg[6:0]	01
clk	1
ctrl	1
img_addr[15:0]	UUUU
weight_addr[15:0]	UUUU
bias_addr[15:0]	UUUU
mac_inp1[7:0]	UU
mac_inp2[15:0]	000b
add_bais[7:0]	00
img_into_weight[15:0]	ff92
img_into_we...bias[15:0]	00e9
we	0
re	1
we_hl	0
re_hl	0
we_fl	0
re_fl	1
rom_addr[15:0]	cac1
rom_curr_addr[15:0]	cabf
rom_bias_addr[15:0]	caca
ram_addr[10:0]	001
shifter_en	1
rom_op[7:0]	f6
rom_op_conc[15:0]	fff6
ram_op[15:0]	0000
hl1_ram[15:0]	000b
fl1_ram[15:0]	UUUU
relu_op[15:0]	00e9
shifter_op[15:0]	0007
hidden_lay...ddr[10:0]	002
final_layer_addr[10:0]	00c
final_output[3:0]	0
state_fsm1	2
s0_fsm1	0
s1_fsm1	1
s2_fsm1	2
state_fsm2	8
s0_fsm2	0
s1_fsm2	1
s2_fsm2	2
s3_fsm2	3
s4_fsm2	4

Figure 2: Simulation for `imgdata_digit0.mif`



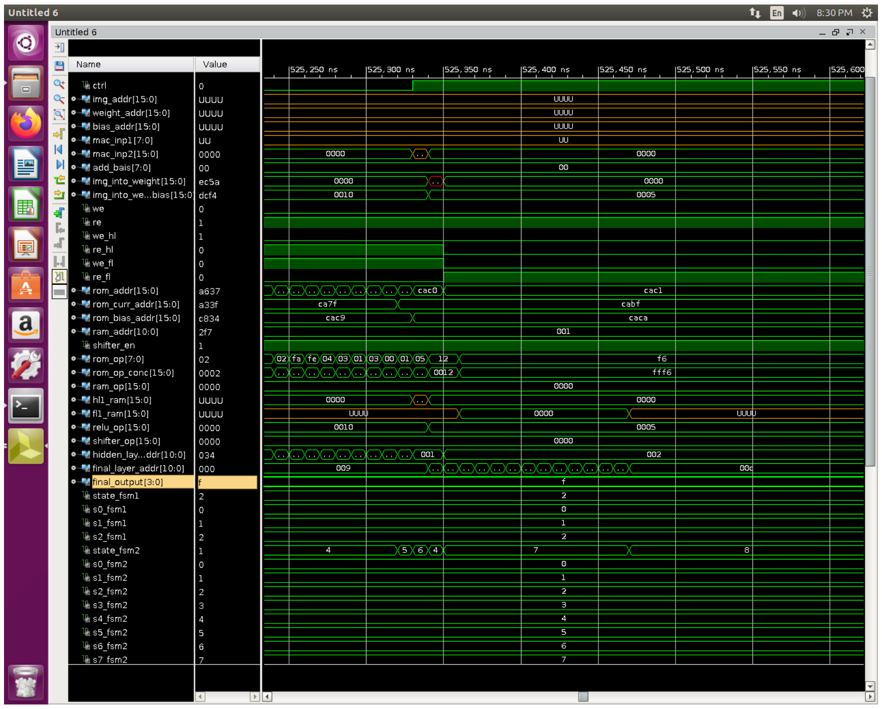
Name	Value
an[3:0]	7
seg[6:0]	04
clk	1
ctrl	0
img_addr[15:0]	UUUU
weight_addr[15:0]	UUUU
bias_addr[15:0]	UUUU
mac_inp1[7:0]	UU
mac_inp2[15:0]	0000
add_bais[7:0]	00
img_into_weight[15:0]	03d4
img_into_we...bias[15:0]	f749
we	0
re	1
we_hl	1
re_hl	0
we_fl	0
re_fl	0
rom_addr[15:0]	99bf
rom_curr_addr[15:0]	96ff
rom_bias_addr[15:0]	c830
ram_addr[10:0]	2bf
shifter_en	1
rom_op[7:0]	01
rom_op_conc[15:0]	0001
ram_op[15:0]	0000
hl1_ram[15:0]	UUUU
fl1_ram[15:0]	UUUU
relu_op[15:0]	0000
shifter_op[15:0]	0000
hidden_lay...ddr[10:0]	030
final_layer_addr[10:0]	000
final_output[3:0]	f
state_fsm1	2
s0_fsm1	0
s1_fsm1	1
s2_fsm1	2
state_fsm2	1
s0_fsm2	0
s1_fsm2	1
s2_fsm2	2
s3_fsm2	3
s4_fsm2	4

Figure 3: Simulation for `imgdata_digit1.mif`



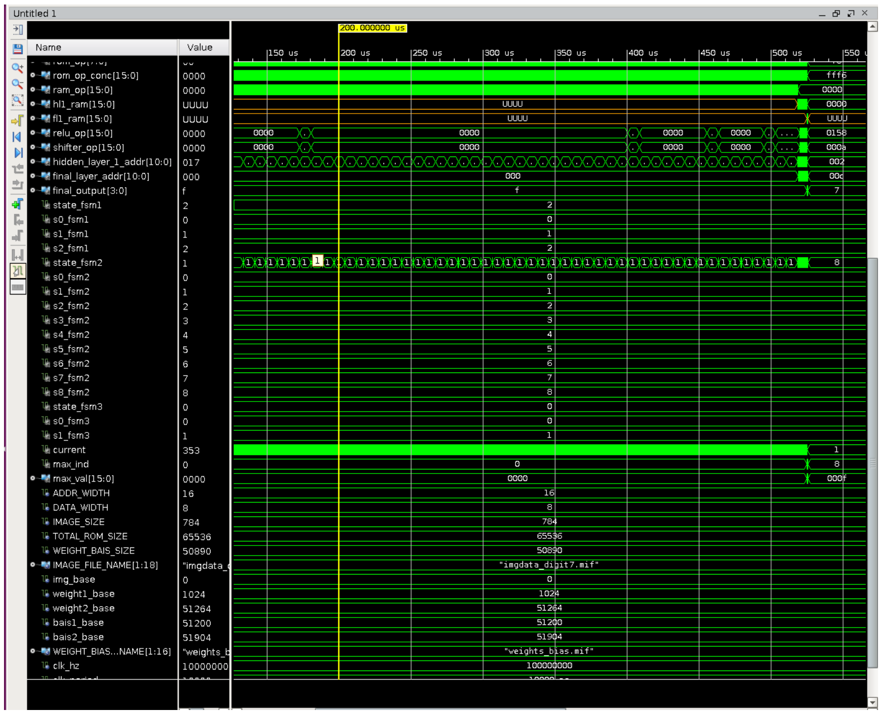
```
Untitled 5
Name                        Value
an[3:0]                     7
seg[6:0]                    04
clk                         1
ctrl                        0
img_addr[15:0]              UUUU
weight_addr[15:0]           UUUU
bias_addr[15:0]             UUUU
mac_inp1[7:0]               UU
mac_inp2[15:0]              0000
add_bais[7:0]               00
img_into_weight[15:0]       ed72
img_into_we...bias[15:0]    ec64
we                          0
re                          1
we_hl                       1
re_hl                       0
we_fl                       0
re_fl                       0
rom_addr[15:0]              4bd1
rom_curr_addr[15:0]         4a6f
rom_bias_addr[15:0]         c817
ram_addr[10:0]              161
shifter_en                  1
rom_op[7:0]                 00
rom_op_conc[15:0]           0000
ram_op[15:0]                0000
hl1_ram[15:0]               UUUU
fl1_ram[15:0]               UUUU
relu_op[15:0]               0000
shifter_op[15:0]            0000
hidden_lay...ddr[10:0]      017
final_layer_addr[10:0]      000
final_output[3:0]           f
state_fsm1                  2
s0_fsm1                     0
s1_fsm1                     1
s2_fsm1                     2
state_fsm2                  1
s0_fsm2                     0
s1_fsm2                     1
s2_fsm2                     2
s3_fsm2                     3
s4_fsm2                     4
```

Figure 4: Simulation for `imgdata_digit2.mif`



```
Untitled 6
Name                        Value
ctrl                        0
img_addr[15:0]              UUUU
weight_addr[15:0]           UUUU
bias_addr[15:0]             UUUU
mac_inp1[7:0]               UU
mac_inp2[15:0]              0000
add_bais[7:0]               00
img_into_weight[15:0]       ec5a
img_into_we...bias[15:0]    dcf4
we                          0
re                          1
we_hl                       1
re_hl                       0
we_fl                       0
re_fl                       0
rom_addr[15:0]              a637
rom_curr_addr[15:0]         a33f
rom_bias_addr[15:0]         c834
ram_addr[10:0]              2f7
shifter_en                  1
rom_op[7:0]                 02
rom_op_conc[15:0]           0002
ram_op[15:0]                0000
hl1_ram[15:0]               UUUU
fl1_ram[15:0]               UUUU
relu_op[15:0]               0000
shifter_op[15:0]            0000
hidden_lay...ddr[10:0]      034
final_layer_addr[10:0]      000
final_output[3:0]           f
state_fsm1                  2
s0_fsm1                     0
s1_fsm1                     1
s2_fsm1                     2
state_fsm2                  1
s0_fsm2                     0
s1_fsm2                     1
s2_fsm2                     2
s3_fsm2                     3
s4_fsm2                     4
s5_fsm2                     5
s6_fsm2                     6
s7_fsm2                     7
```

Figure 5: Simulation for `imgdata_digit3.mif`



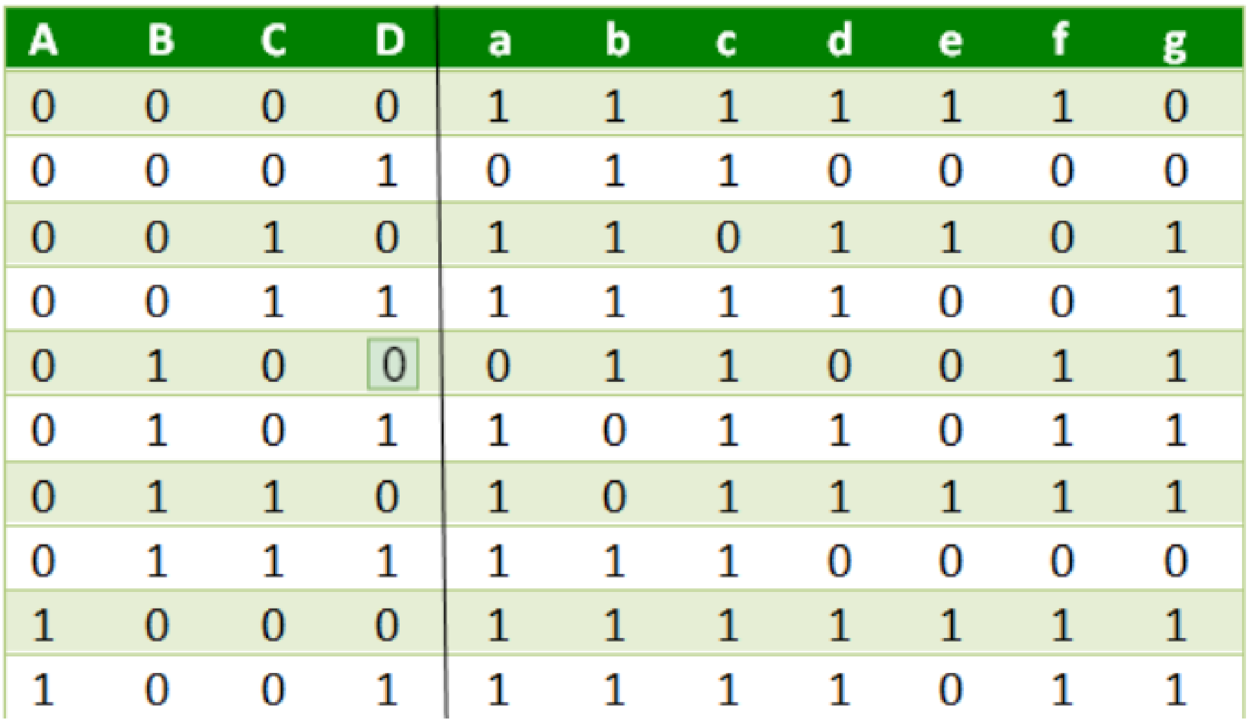
Name	Value
rom_op_conc[15:0]	0000
ram_op[15:0]	0000
hl1_ram[15:0]	UUUU
fl1_ram[15:0]	UUUU
relu_op[15:0]	0000
shifter_op[15:0]	0000
hidden_layer_1_addr[10:0]	017
final_layer_addr[10:0]	000
final_output[3:0]	f
state_fsm1	2
s0_fsm1	0
s1_fsm1	1
s2_fsm1	2
state_fsm2	1
s0_fsm2	0
s1_fsm2	1
s2_fsm2	2
s3_fsm2	3
s4_fsm2	4
s5_fsm2	5
s6_fsm2	6
s7_fsm2	7
s8_fsm2	8
state_fsm3	0
s0_fsm3	0
s1_fsm3	1
current	353
max_ind	0
max_val[15:0]	0000
ADDR_WIDTH	16
DATA_WIDTH	8
IMAGE_SIZE	784
TOTAL_ROM_SIZE	65536
WEIGHT_BAIS_SIZE	50890
IMAGE_FILE_NAME[1:18]	"imgdata_
img_base	0
weight1_base	1024
weight2_base	51264
bais1_base	51200
bais2_base	51904
WEIGHT_BIAS...NAME[1:16]	"weights_b
clk_hz	10000000

Figure 6: Simulation for `imgdata_digit7.mif`

5 Design Files Included

5.1 Seven Segment Decoder

It is a combinational circuit which takes a 4 bits input from 4 to 1 mux and outputs 7 bits which represent the cathode signals in the LED display. The logic equation of each the output is obtained using following truth table.

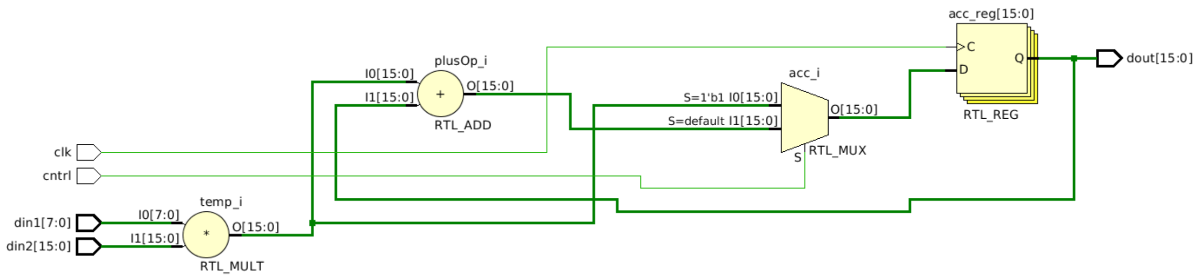


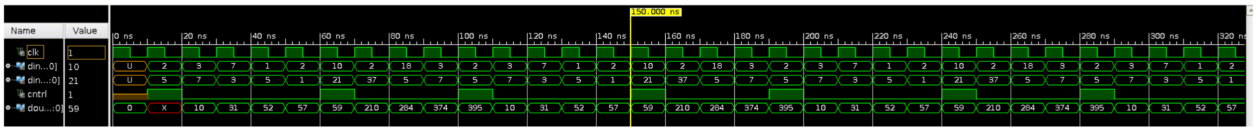
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Figure 7: Truth Table of Seven Segment Decoder

5.2 MAC

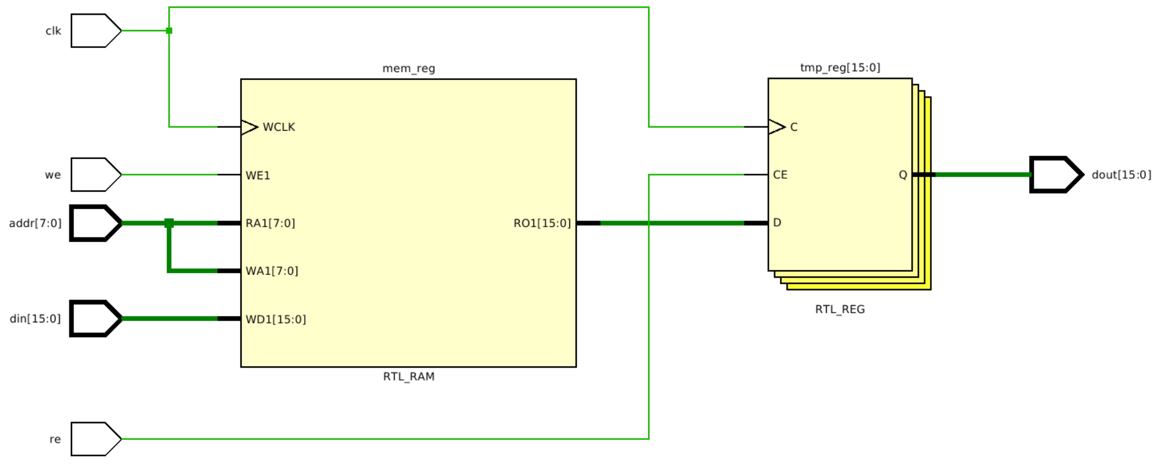
It takes 4 inputs namely - clk (std_logic) , din1 (8 bit vector) , din2(16 bit vector) , cntrl (std_logic). It gives the multiplication of din1 and din2 as output (dout which is a 16 bit vector) if cntrl is 1. If cntrl = 0 then it accumulates(adds) the output of din1*din2 and gives the output. It truncate the 23 downto 16th bits

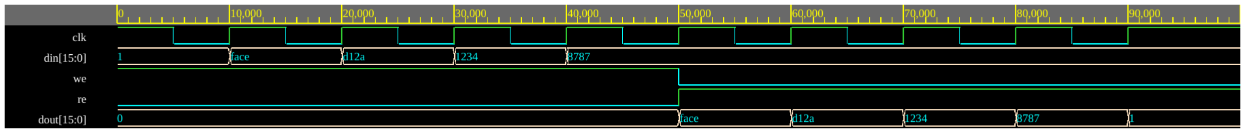



150.000 ns
Name
Value
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns
320 ns
clk 1
din...0] 10
U 2 3 7 1 2 10 2 18 3 2 3 7 1 2 10 2 18 3 2 3 7 1 2 10 2 18 3 2 3 7 1 2
din...:0] 21
U 5 7 3 5 1 21 37 5 7 5 7 3 5 1 21 37 5 7 5 7 3 5 1 21 37 5 7 5 7 3 5 1
cntrl 1
dou...:0] 59
0 X 10 31 52 57 59 210 284 374 395 10 31 52 57 59 210 284 374 395 10 31 52 57 59 210 284 374 395 10 31 52 57

MAC is clocked , it work in 2 clock cycles (design decision). At the first rising edge din1, din2 will be updates and at the 2nd rising edge the output is calculated.

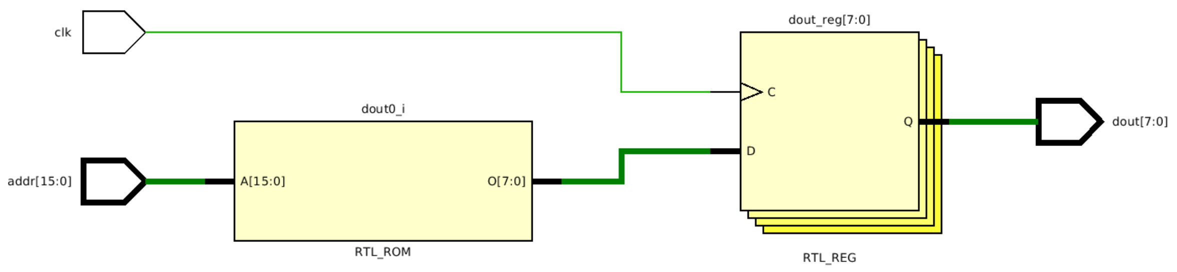
5.3 RAM

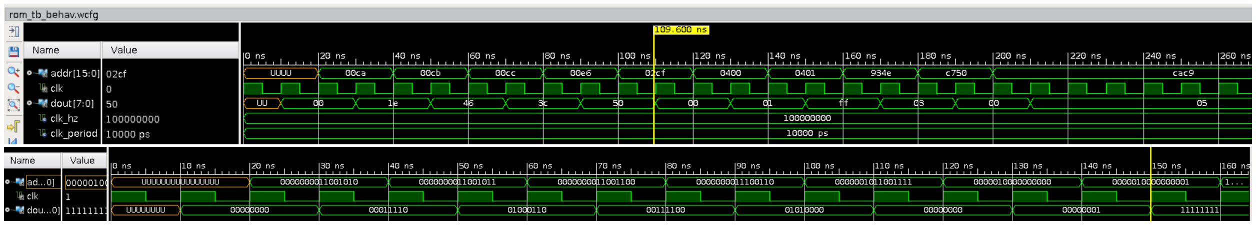




It is a 256 size array of 16 bits vector. It takes a clk , din (16 bit vector) , addr (8 bit vector) , re(read enable) , we(write enable) as input and dout as output . If we is 1 then the din is written on the array at address = addr. If re is 1 then the dout(16 bits) becomes equal to the element which is present at the address = addr.

5.4 ROM

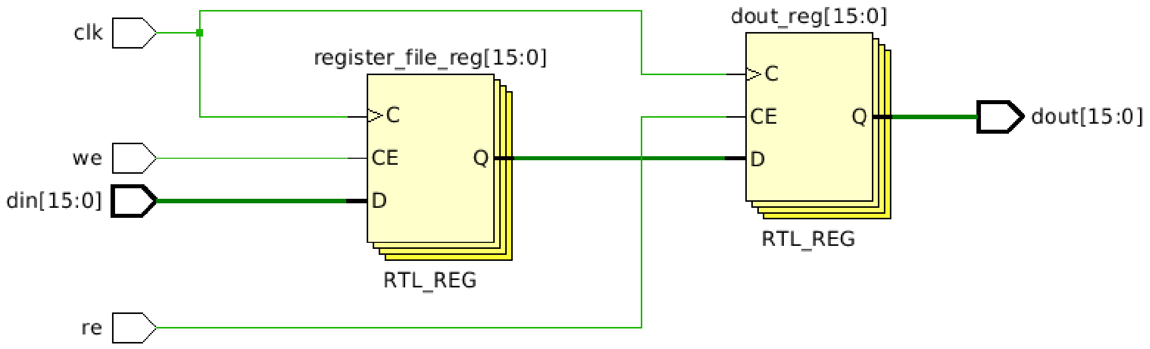


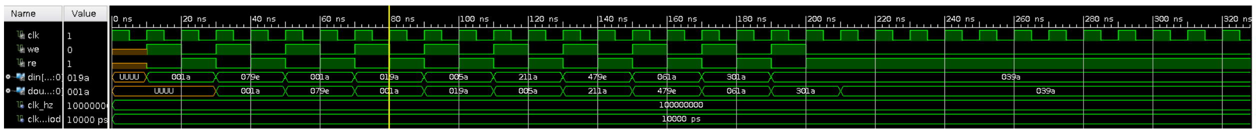


Simulation in singed binary radix

It is a 65536 size array of 8 bit vectors. The memory is initialised using init_mem function. The 784 sized image is stored from address 0 to 783 and the 50890 weights and biases are stored from address 1024. It takes clk , addr as input and outputs the 8 bit vector present at the address = addr. We have made it clocked because in the assignment it was given a clk as input.

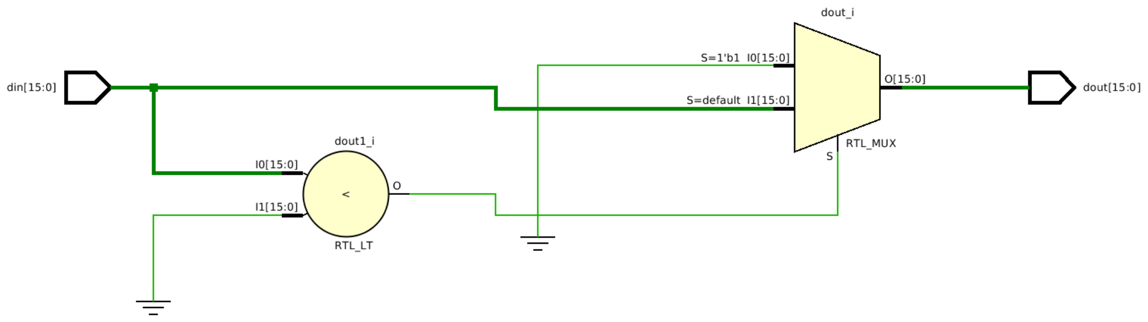
5.5 Register

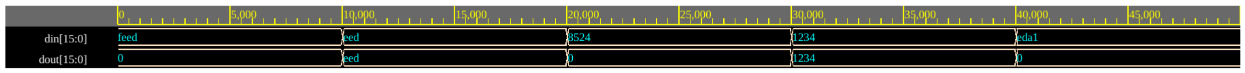




This is a vector of 16 bits. It takes clk, we , re , din as input and dout as output. The input and outputs (din and dout) are both 16 bits. The Register is written on if we is 1. If re is 1 then the value of register gets stored in dout and the same is given as output.

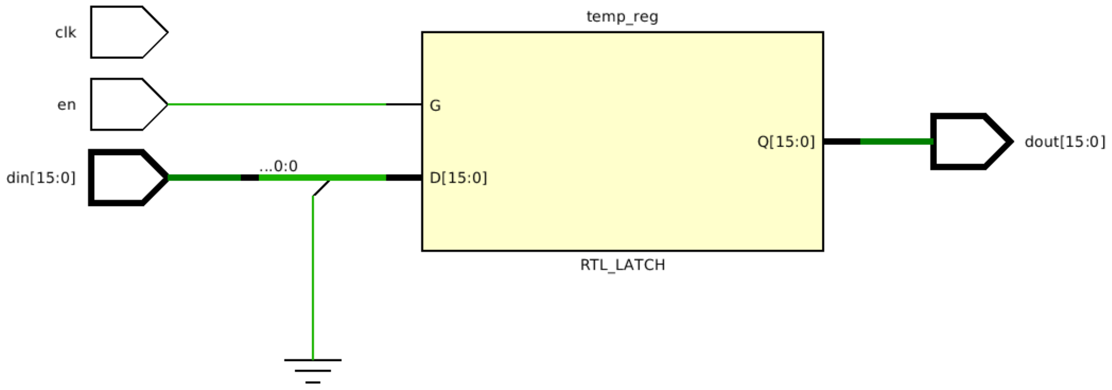
5.6 Comparator

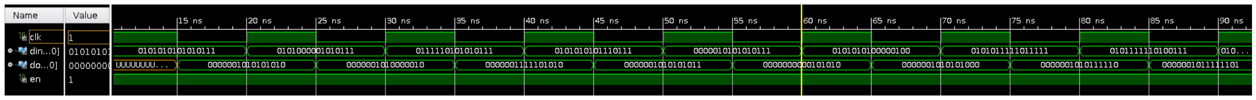




It takes din (16 bit vector) as input and converts it to signed. It compares the signed vector with 0 and return 0 if the signed vector is less than 0 or return the din as output if din is greater than 0.

5.7 Shifter





It takes clk , en (1 bit) , din (16 bits) as input. When en = 1 the din is shifter to right by 5 bits and is stored in dout and same is then output. dout is also 16 bits.