ESE 507 Project 3

Kshemal Gupte 114508816

Shruti Chandrakant Tajne 113398674

Question 1

Our control module is based on a Finite State Machine (FSM). There are 7 states– rst, idle, load_x, read, output_state, clear, and done.

1) "rst" state is enabled when we get the reset from the testbench. We set every signal except "clear counter signals" to 0. We clear our counters by setting them as 1.

2) "idle" state is there to ensure once we are done loading vector, memory does not load more data even if input_valid is 1 since Read and compute operation is taking place.

3) "load_x" is the load vector state. The counter "count_x" will increment based on (input valid = 1 & state = load_x & overflow_v=0). The addr_x is dependent on the counter output. The wr_en_x is also set high based on input_valid. The counter has an "overflow_v" that sets 1 when count_x reaches N.

4) "read" is the– read values from memory and compute– state. The counter "count" starts incrementing as soon as we enter this state. It has an "overflow_row" flag which sets 1 when the count reaches N. The purpose of it is to compute the first row. The addr_w and addr_x are set based on this counter. The en_acc is set to 1 in this state.

5) "output_state" is the state we enter as soon as the first row is computed and we wait there until we get output_ready = 1 from the testbench. The output_valid is set as 1 here.

6) "clear" is a state we enter to clear the accumulator value and be ready for the next row to be computed.

7) "done" state is when the whole calculation is done and we are ready for new inputs. We clear all the counters and move to the load_x state.

8) The counters used in the FSM keep track of the control logic.

All the counters are reaching their overflow values based on M and N. We have a total of 6 counters in our design.

1) counter #(N) countV – Counts the number of values in vector. Based on N.
2) counter #(N) countOP – Counts output after 1st row of MAC. Based on N.
3) counter #(M/P) countCLR – Counter for clearing output after 1st row MAC
4) counter #(N+1) countREAD – Counter to remain in read state for N+1 clock cycles.
5) counter #((M+1)/P) countACC – Counts the number of MAC operations i.e number of rows.
6) counter #(P) countMUX – Used when P>1

As the parameters M and N scale, the number of counters remain same but the memory and vector data increases, so the number of clock cycles to get the output vector increases significantly.

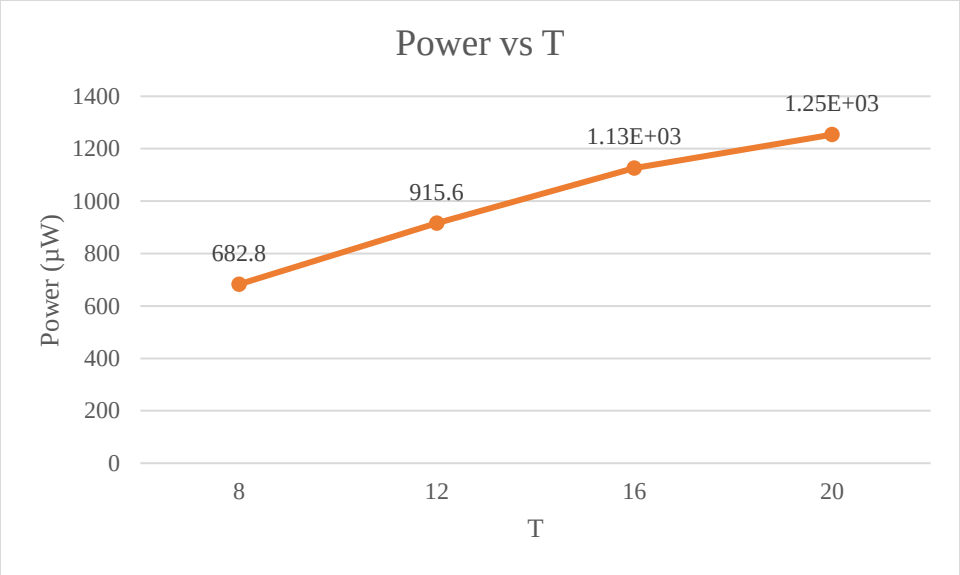
Question 2

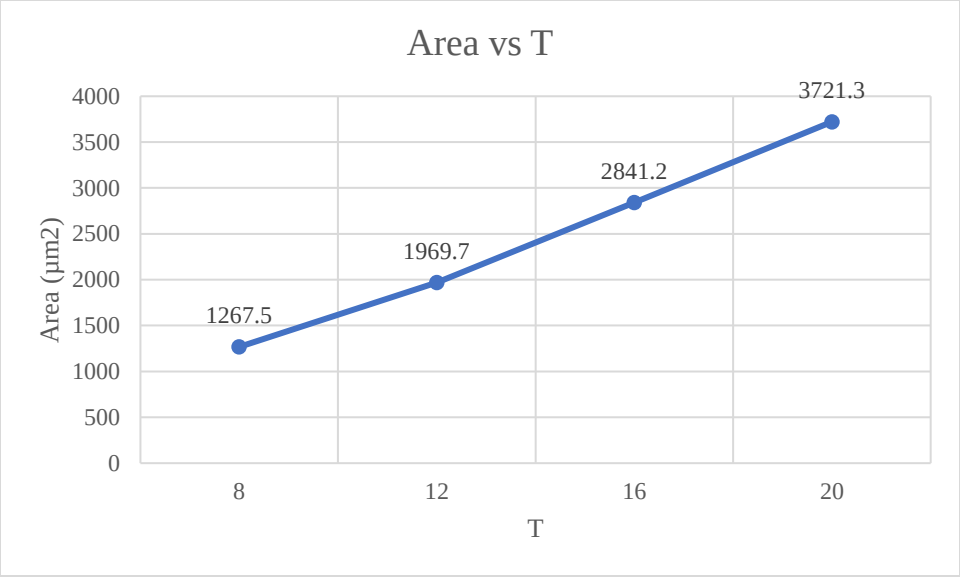
For implementing parallelism, we added a P:1 mux in the output stage to output P number of outputs to a single output stream. We programmed the hardware simulator so that it would instantiate the datapath 'P' number of times. Similarly, we reorganized the structure of ROM in order to break it down into 'P' number of ROMs having appropriate data. We also changed the parameters of the counters based on the value of P.

Question 3

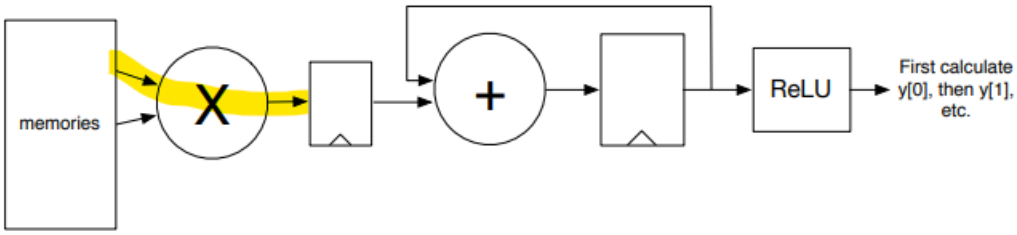
M=6, N=6, R=1, P=1

T	Power (µW)	Area (µm^2)
8	682.8	1267.5
12	915.6	1969.7
16	1.1262e+03	2841.2
20	1.2537e+03	3721.3





Critical Path:
T = 8:
The output of ROM W to the input of reg C3.
romW_1/z_reg[3] -> dp_1_fc_6_6_8_1_1/reg_C3/q_reg[0])



T = 12:
romW_1/z_reg[7] -> dp_1_fc_6_6_12_1_1/reg_C3/q_reg[2]

T = 16:
The output of the register of countCLR to input of ROM W.
control_fc_6_6_16_1_1/countCLR/regOut_reg[0] -> romW_1/z_reg[12]

T = 20:
romW_1/z_reg[7] -> dp_1_fc_6_6_20_1_1/reg_C3/q_reg[1]

Question 4

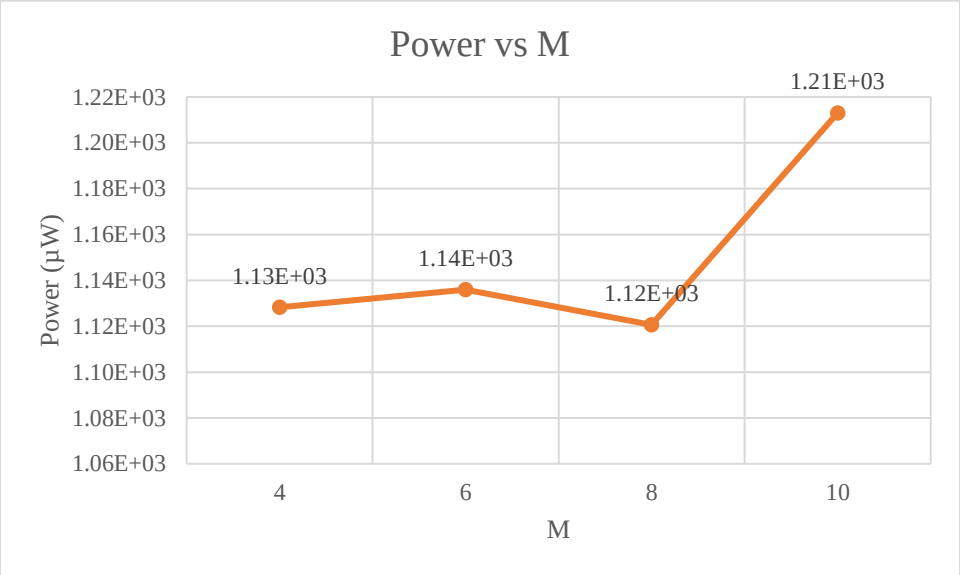
c = N + 1.0 + ((N+2)*(M/P)) + ((1+1)*(M/P)) + (1*((M/P)-1)) + (1*((M/P)-1)) + 2.0

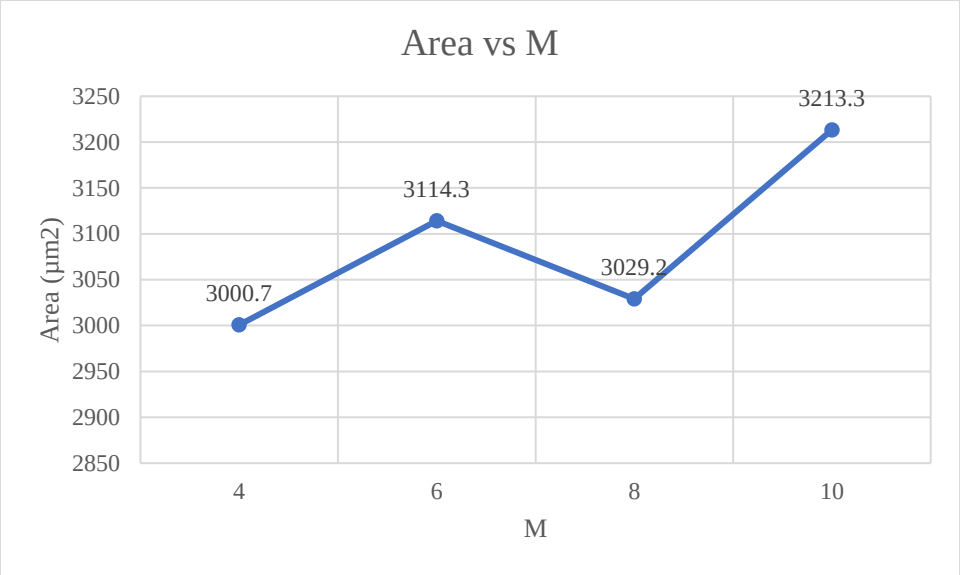
Load state: N clock cycles
Idle state: 1 clock cycle

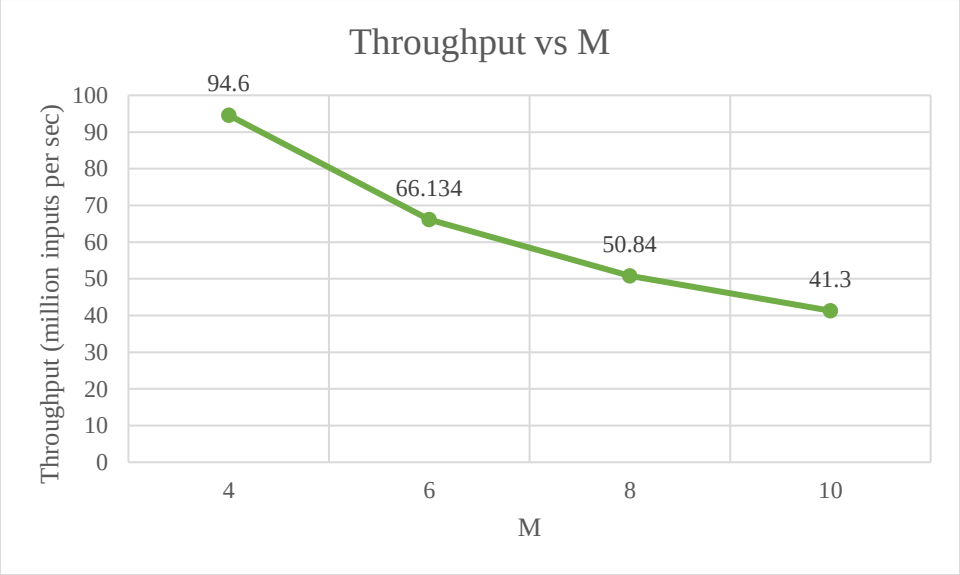
Read state: (N + 2) * (M/P) clock cycles
Output state: (P + 1) * (M/P) clock cycles
Clear state: 1 * (M/P - 1) clock cycles
Idle state: 1 * (M/P - 1) clock cycles
Done state: 2 clock cycles

N=8, T=16, R=1, P=1

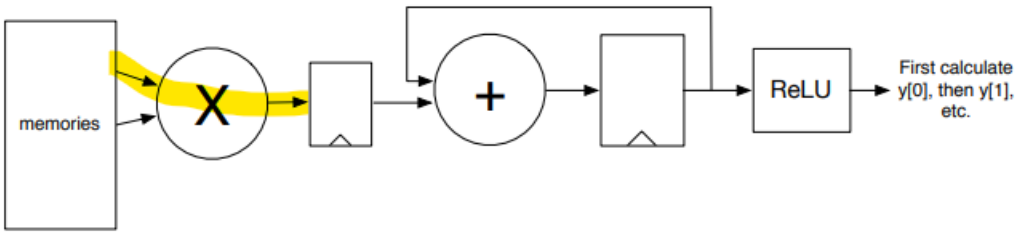
M	Power (µW)	Area (µm^2)	Throughput (million inputs per sec)	c (clock cycles)
4	1.1283e+03	3000.7	94.6	65
6	1.1359e+03	3114.3	66.134	93
8	1.1206e+03	3029.2	50.84	121
10	1.2130e+03	3213.3	41.3	149



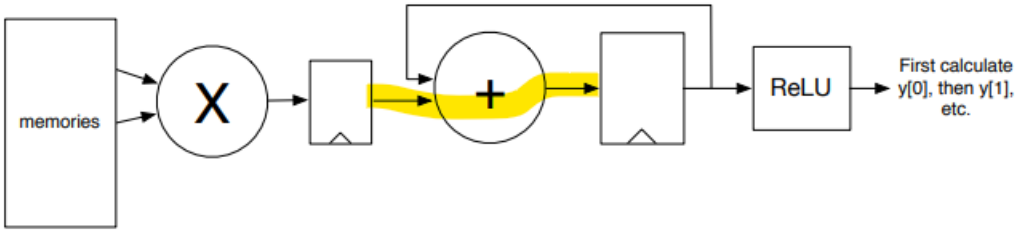




Critical Path:
M = 4:
The output of ROM W to the input of reg C3.
romW_1/z_reg[9] -> dp_1_fc_4_8_16_1_1/reg_C3/q_reg[0]



M = 6:
The output of reg C3 in the datapath to the input of accumulator register.
dp_1_fc_6_8_16_1_1/reg_C3/q_reg[0] -> dp_1_fc_6_8_16_1_1/reg_acc/q_reg[0]



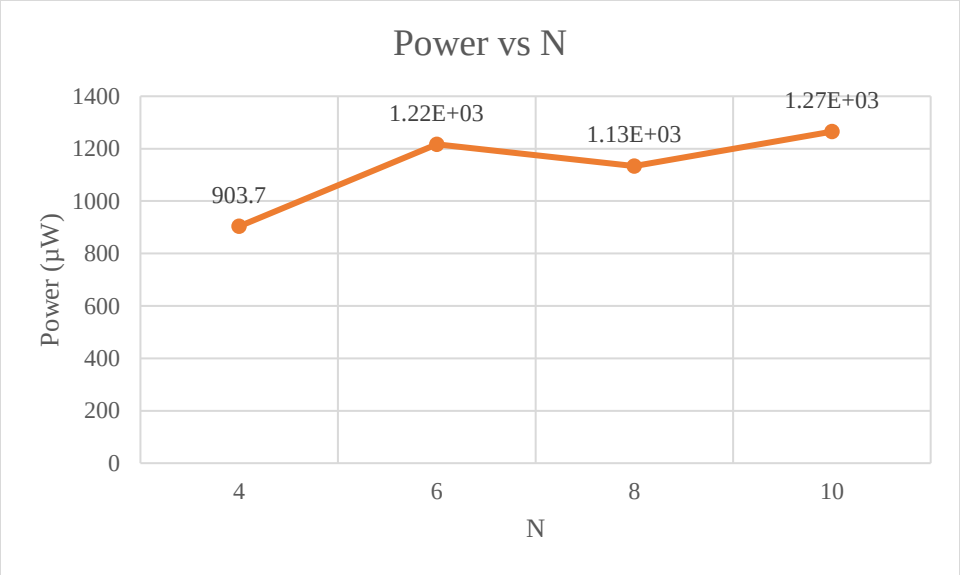
M = 8:
The output of ROM W to the input of reg C3.
romW_1/z_reg[11] -> dp_1_fc_8_8_16_1_1/reg_C3/q_reg[0]

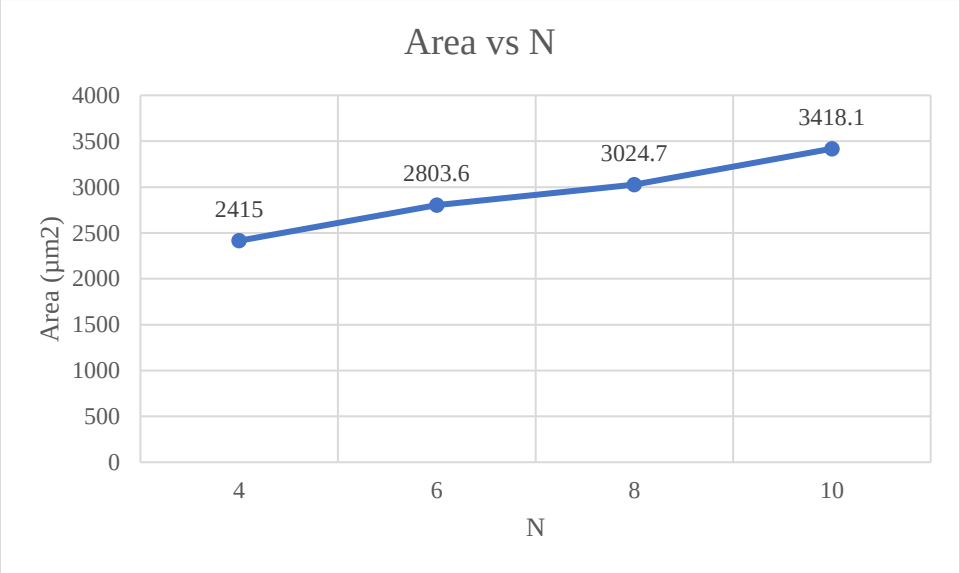
M = 10:
It seems that the critical path is inside the accumulator register itself.
dp_1_fc_10_8_16_1_1/reg_acc/q_reg[0] -> dp_1_fc_10_8_16_1_1/reg_acc/q_reg[0]

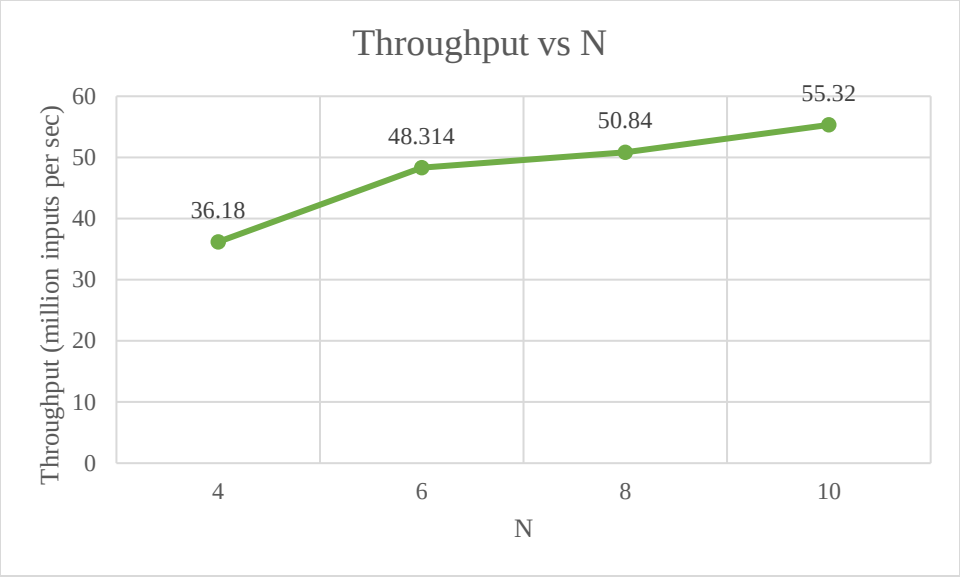
Question 5

M=8, T=16, R=1, P=1

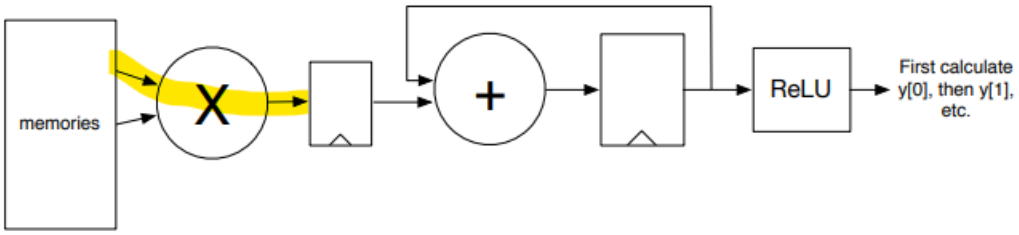
N	Power (µW)	Area (µm^2)	Throughput (million inputs per sec)	c (clock cycles)
4	903.7	2415.0	36.18	85
6	1.2167e+03	2803.6	48.314	103
8	1.1332e+03	3024.7	50.84	121
10	1.2651e+03	3418.1	55.32	139


Power vs N
Power (µW)
1400
1200
1000
800
600
400
200
0
903.7
1.22E+03
1.13E+03
1.27E+03
4
6
8
10
N


Area vs N
Area (µm2)
4000
3500
3000
2500
2000
1500
1000
500
0
2415
2803.6
3024.7
3418.1
4
6
8
10
N



Critical Path:

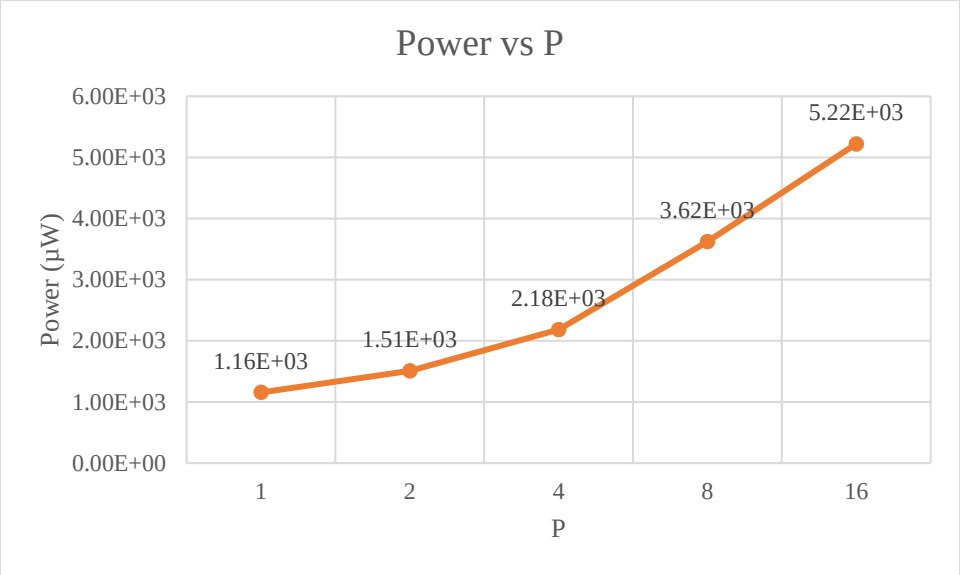


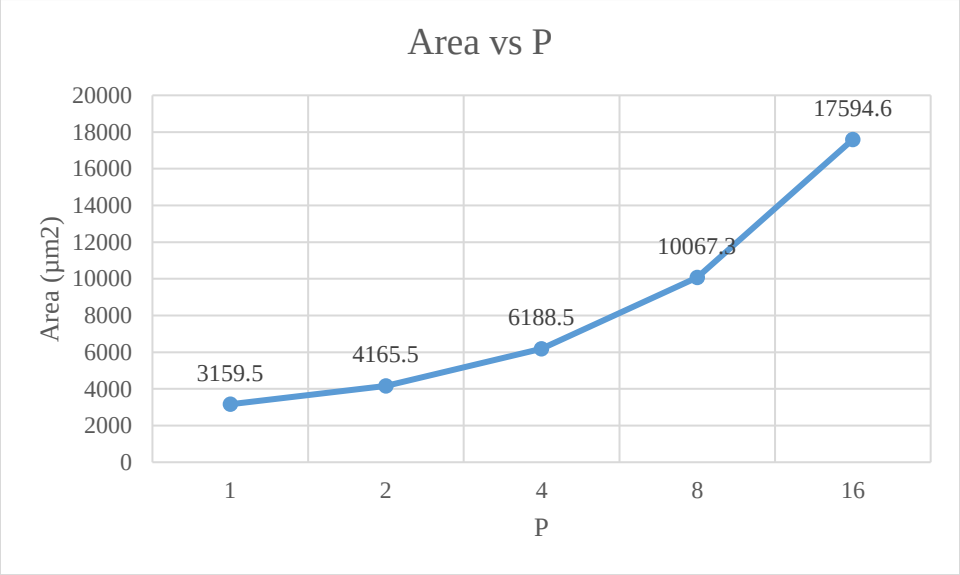
N = 4, 6, 8, 10
The output of ROM W to the input of reg C3.
romW_1/z_reg[7] -> dp_1_fc_8_4_16_1_1/reg_C3/q_reg[6]
romW_1/z_reg[0] -> dp_1_fc_8_6_16_1_1/reg_C3/q_reg[1]
romW_1/z_reg[9] -> dp_1_fc_8_8_16_1_1/reg_C3/q_reg[5]
romW_1/z_reg[11] -> dp_1_fc_8_10_16_1_1/reg_C3/q_reg[0]

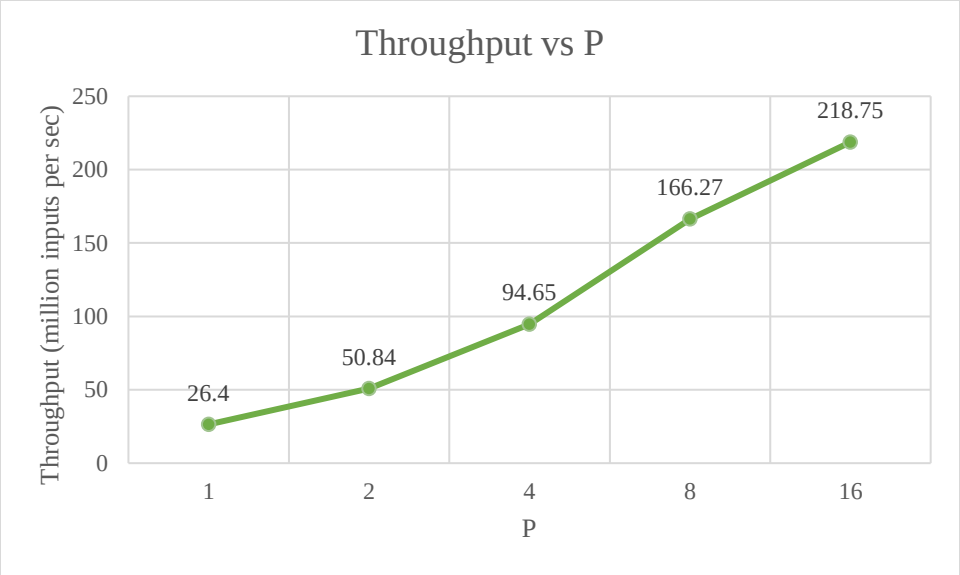
Question 6

M=16, N=8, T=16, R=1

P	Power (µW)	Area (µm^2)	Throughput (million inputs per sec)	c (clock cycles)
1	1.1568e+03	3159.5	26.4	233
2	1.5094e+03	4165.5	50.84	121
4	2.1827e+03	6188.5	94.65	65
8	3.6221e+03	10067.3	166.27	37
16	5.2218e+03	17594.6	218.75	23


Power vs P
6.00E+03
5.00E+03
4.00E+03
3.00E+03
2.00E+03
1.00E+03
0.00E+00
Power (µW)
1.16E+03
1.51E+03
2.18E+03
3.62E+03
5.22E+03
1
2
4
8
16
P


Area vs P
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Area (µm2)
3159.5
4165.5
6188.5
10067.3
17594.6
1
2
4
8
16
P



Critical Path:
P = 1:
The output of ROM W 1 to the input of reg C3.
romW_1/z_reg[13] -> dp_1_fc_16_8_16_1_1/reg_C3/q_reg[0]

P = 2:
The output of ROM W 2 to the input of reg C3.
romW_2/z_reg[5] -> dp_2_fc_16_8_16_1_2/reg_C3/q_reg[0]

P = 4:
From the output of memory V to the input of reg C3.
memV_fc_16_8_16_1_4/data_out_reg[0] -> dp_4_fc_16_8_16_1_4/reg_C3/q_reg[0]

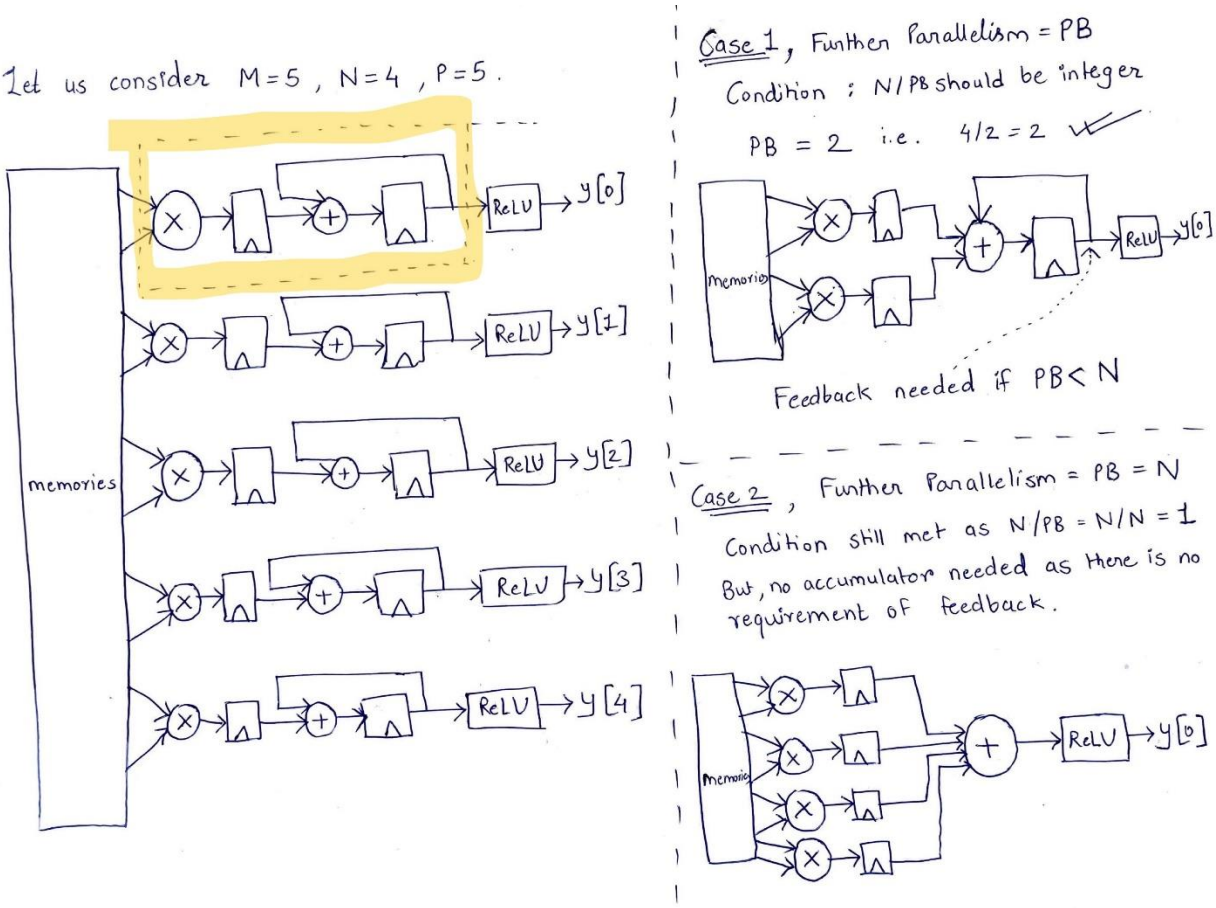
P = 8:
The output of ROM W 8 to the input of reg C3.
romW_8/z_reg[3] -> dp_8_fc_16_8_16_1_8/reg_C3/q_reg[0]

P = 16:
It seems that the critical path is inside the accumulator register itself.
dp_12_fc_16_8_16_1_16/reg_acc/q_reg[0] -> dp_12_fc_16_8_16_1_16/reg_acc/q_reg[6]

We think that the design with P = 8 is the most efficient than others based on the following trade-offs
1) The cost of area increases by approximately by 3x as compared to P=1.
2) The power consumption increases by approximately 3x as compared to P=1.
3) But the improvement in throughput is approximately 6x as compared to P=1.

Question 7

To further parallelize our design, we can parallelize the individual datapaths.



But case 2 seems inefficient because multipliers are costlier to design rather than adders.

Estimated throughput = N/c * clock frequency

c = N + 1.0 + (((N+2)/PB)*(M/P)) + ((1+1)*(M/P)) + (1*((M/P)-1)) + (1*((M/P)-1)) + 2.0

Load state: N clock cycles
Idle state: 1 clock cycle
Read state: ((N + 2)/PB) * (M/P) clock cycles
Output state: (P + 1) * (M/P) clock cycles
Clear state: 1 * (M/P - 1) clock cycles
Idle state: 1 * (M/P - 1) clock cycles
Done state: 2 clock cycles

Question 8

We have a budget as B.

If the value of B<3, then B=3. If value of B>M1+M2+M3, then B=M1+M2+M3.

The program first calculates the divisors of M1, M2 and M3. These are the valid values which can be taken by P1, P2 and P3.

Then calculates the throughput for P1=1, P2=1 and P3=1. It finds the layer in which throughput is minimum, i.e., we find where the bottleneck lies.

In that layer, it will increment to the next valid value of P using the divisors that were calculated earlier, if the budget allows. The value of throughput for that layer is recomputed and if the difference in throughputs is not delta (here delta=2), it does not increment P. If both the conditions are met, P is incremented.

P stops incrementing if the new value of P matches the old value.

Now for the new values of P1, P2 and P3, the throughput for each layer is computed again. And the above steps are repeated until it runs out of budget.

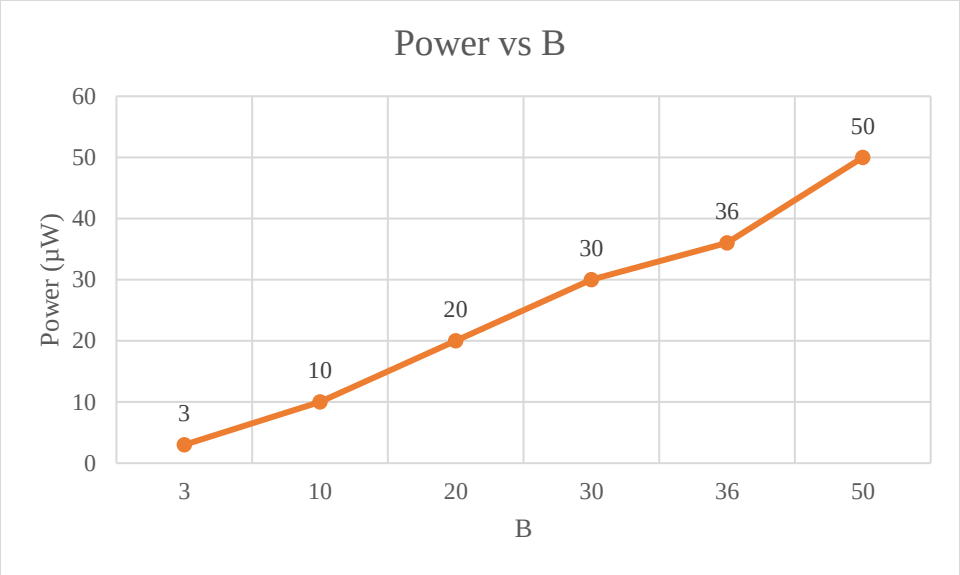
It can be further improved by adding weights based on some conditions, like dimensions of matrix and/or vector, and assigning P based on those weights.

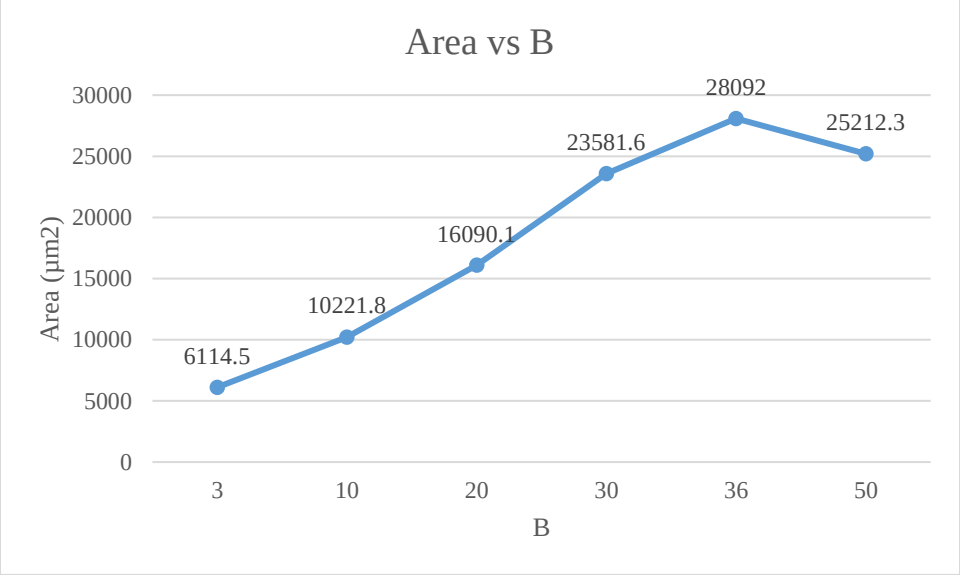
It can be observed from the design files that the layers that have bigger matrix and vector dimensions are allocated highest parallelism. Thus, the system seems to work well.

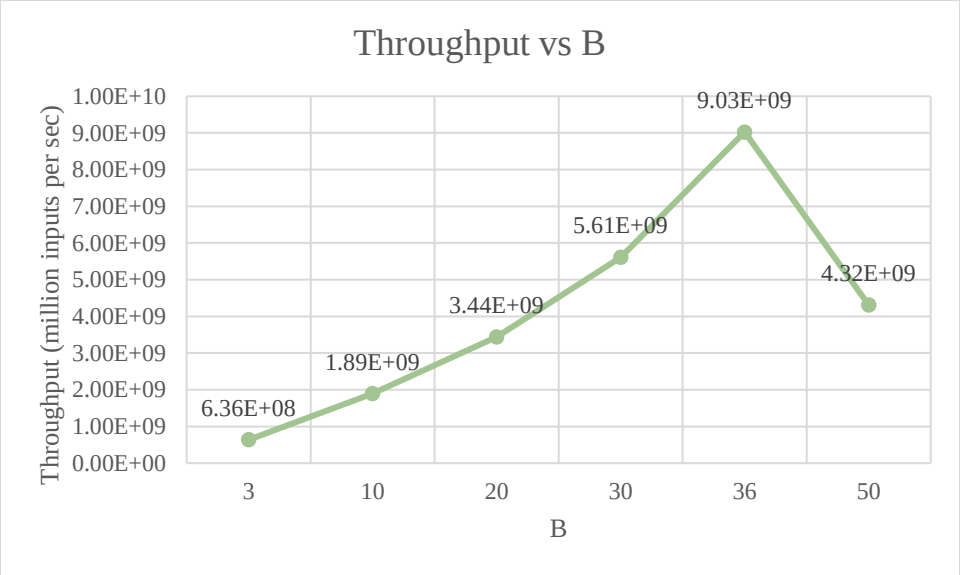
Question 9

N=4, M1=8, M2=12, M3=16, T=16, R=1

B	Power (μW)	Area (μm^2)	Throughput (million inputs per sec)
3	4.0528e+03	6114.5	6.36183e+08
10	5.5414e+03	10221.8	1.89349e+09
20	8.1239e+03	16090.1	3.44086e+09
30	1.1384e+04	23581.6	5.61403e+09
36	1.4570e+04	28092	9.02564e+09
50	6.8646e+03	25212.3	4.3159e+09


Power vs B
Power (µW)
60
50
40
30
20
10
0
3
10
20
30
36
50
B


Area vs B
Area (µm2)
30000
25000
20000
15000
10000
5000
0
6114.5
10221.8
16090.1
23581.6
28092
25212.3
3
10
20
30
36
50
B



Question 10

The hardware generator will call genFClayer function for the number of layers specified by the user. The top level module will also instantiate that many layers and connect them. The dimensions of the output of the initial layer should match the dimensions of the input of the next layer and so on. If the parallelism for every layer is independent of each other, then the hardware generator code should accommodate the added layers.

Question 11

We did everything together.