ESE 507

Project 3: Hardware Generation Tool

Ayman Azad - Mariano Bello

Due: 12/6/20

1. Describe how you designed your control logic. How did you structure the design so that it can be changed as the M and N parameters change? Explain how the structure changes as these parameters grow.

The initial design from project 2 already had a working structure for a module that used a matrix of size N by N and a vector which was also of size N. The module had an option to take in a new matrix or to simply take in N values for the vector. We were able to change this design in part 1 by removing the memory module instantiation used for the matrix and replaced it with a similar ROM module. This module would hold the values for the matrix and read synchronously. Since the matrix would never be written to, the values to be output would be determined in the C ++ program and assigned based on the values read from the constant values text file. The address for reading this module now had to be determined by the controller using the row and column indices. The new datapath for this module would now instantiate the ROM module instead of the memory module previously used for the matrix. The remaining changes were made in the controller.

The controller still operated in three states. Upon reset, the state would become READ, in this state the input values for the vector would be taken. Once all N values are received the state changes to state PROCESS. In this state, the indicated row would be multiplied by the vector using the MAC unit and then the result for a Y value is determined. After adding pipelining to the SatMac unit, each multiply and accumulate took an additional clock cycle, this meant that the state would require exactly 1+2N clock cycles to determine the result. After this, the state changes to DONE where it waits for the value to be output. If output_ready is asserted from the beginning, this state can occur at a minimum time of 1 clock cycle. After performing the output, it determines whether to go back to the process state to perform vector multiplication on the next row of the matrix or to go back to state READ if M values have been output.

Every single control signal, address size, memory size, or any other variable that depends on N, M, T, or P is determined by using those values. This makes the entire module parameterizable, thus, our C++ program had to only set the parameters and the ROM module when creating a new module.

2. Describe how you implemented the parallel (P>1) designs. Explain your structure. What extra logic and storage elements did you need to add? Did you find any clever optimizations to reduce cost?

To implement a dynamic number of modules based on a parameter, P, we utilized the generate function of SystemVerilog to create multiple modules as well as do assignments. By using this method, we were able to generalize our SystemVerilog files and not depend on the C++ program to implement the parallelism. In this way, the System Verilog code would be able to implement P modules based on the parameter alone. To accomplish this, the data path needed to be restructured so that the SacMac module can be used to declare P instances. However, since each SatMac module would perform the same functions as each other, our design allowed all of them to be controlled by the same control signals leading to little change in the controller logic. The SatMac units differ from one another in a certain aspect, they read different rows from the matrix at the same time. Our ROM module only was allowed a single read thus the SatMac modules could not all read from the same ROM module at once. In order to keep the control logic simple as well as optimizing the amount of logic used, we decided to have P smaller ROM modules that would contain the values that would be specifically read by a SatMac unit. To create these ROM modules, we declared a general parameterized ROM module that would have a parameter i which would determine the contents of its memory. This was done using a generate statement which assigned the values of an array of size M/P by N. The C++program would assign the constants accordingly to each SatMacs array. These smaller ROM modules were now placed inside each SatMac units instead of the datapath. The gen_SatMac module would use the generate function to create a total of P SatMacs while assigning the signals accordingly. In addition to creating P SatMac units, the parameter i was passed to the SatMac units denoting which of the P modules it was. This was done primarily for the SatMac module to pass the parameter i to its unique ROM module instantiated within it. As previously mentioned, this passed parameter determined the values in the smaller ROM modules.

By doing this method, we were able to generalize our design so that the top-level module would simply have to declare the parameter P to be passed on. We were also able to save much of the control logic required for handling the multiple modules by having them all share the same control signals, this included the address, address_w, for each of the small ROMs. This is because the process state would use these P SatMacs synchronously, by setting up the memory in each ROM so that each row in the sub ROMs would contain the rows that it would only need, meaning that for each vector multiplication, the same address control signal can be used for all the SatMac units. Our control logic now became more efficient with an increase in P, the process state would still require 1 + 2N clock cycles to perform vector multiplications, however, it now only needed to be used a total of M/P times instead of M. This is because at the end of each process state, there would be P values calculated and

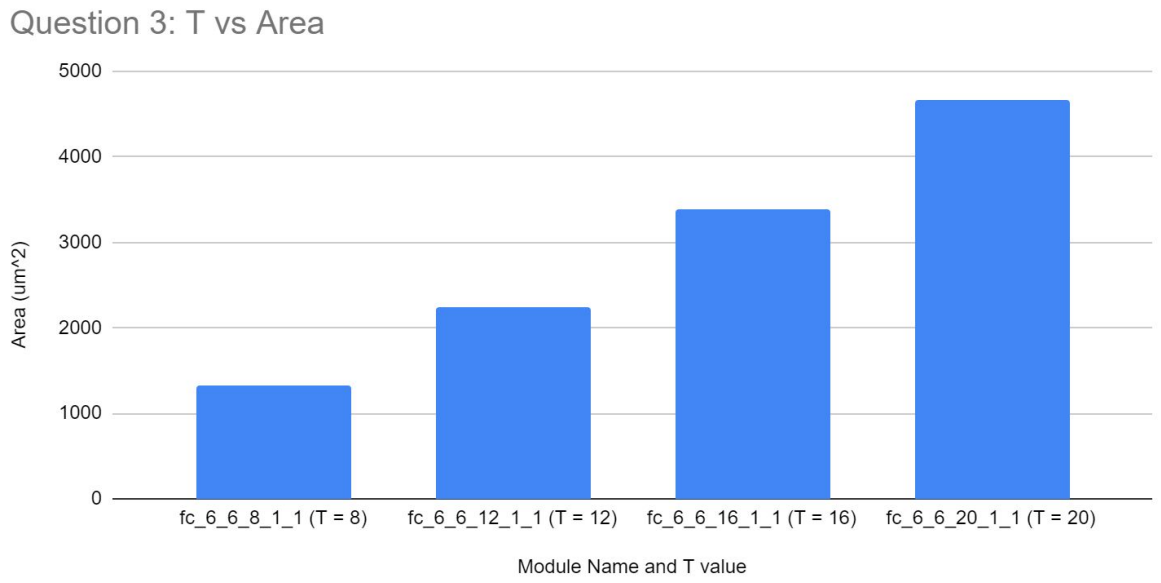
the DONE state would now output all P values then decide to either return to the process state for the next P results or return to the read state to perform new matrix multiplication. Thus, under perfect conditions, the module would be improved by P*N clock cycles per matrix multiplication.

3. Now, we will use synthesis to evaluate how the area and power of a layer change as the data precision T changes. Use your generator to produce four designs (Part 1) with T = 8, 12, 16, and 20, while you keep the other parameters constant: (M, N, P, R) = (6, 6, 1, 1). Then produce two graphs that illustrate: (1) power versus T and (2) area versus T for these designs. Describe where the critical path is located in each design.

Note: For all synthesis designs in this report, the slack is 0.

Table 3.1 shows the synthesis results for the required modules in question 3

Module Name	Period (ns)	Frequency (GHz)	Area (um^2)	Cell Internal Power (uW)	Net Switching Power (uW)	Total Dynamic Power (uW)	Cell Leakage Power (uW)
fc_6_6_8_1_1 (T = 8)	0.9	1.11	1328.137987	653.2737	180.9997	834.2734	28.0438
fc_6_6_12_1_1 (T = 12)	1.1	0.91	2247.965981	802.2253	252.7672	1054.9925	47.9306
fc_6_6_16_1_1 (T = 16)	1.25	0.80	3376.337971	1062.1	469.3283	1531.4283	71.8076
fc_6_6_20_1_1 (T = 20)	1.3	0.77	4658.723968	1412.7	714.0931	2126.7931	98.7342
fc_4_8_16_1_1 (M = 4)	1	1.00	2820.39797	1244.8	435.1061	1679.9061	59.9821



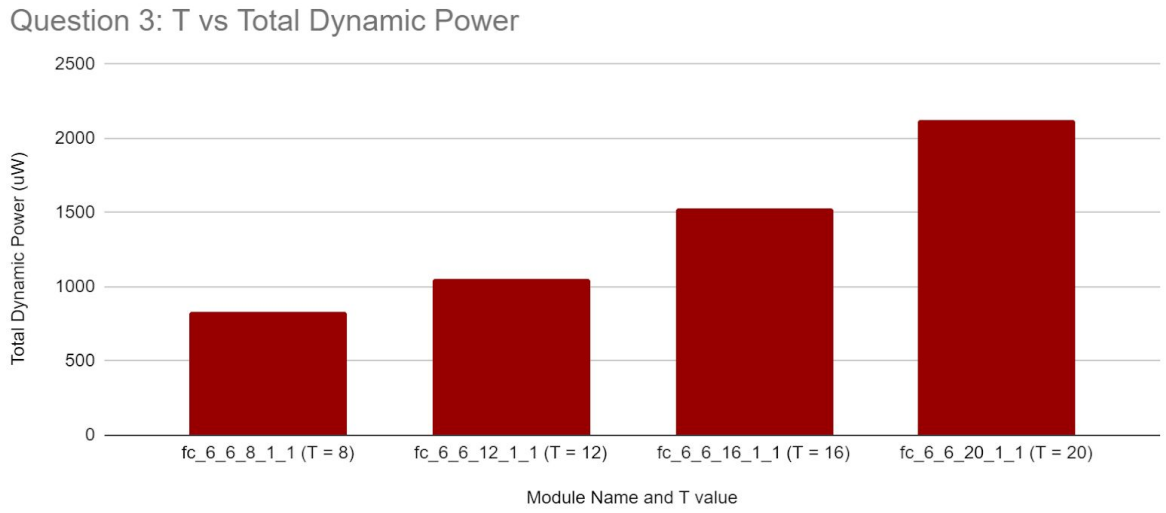


Table 3.3 shows the critical paths for the required modules in question 3

Module Name	Critical Path
fc_6_6_8_1_1 (T = 8)	Startpoint: datapath1/vectorMem/data_out_reg[3] (rising edge-triggered flip-flop clocked by clk) Endpoint: datapath1/satmac/my_SatMac[0].s/m_reg[0] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac. This is the pipeline register required by the specifications of the project

fc_6_6_12_1_1 (T = 12)	Startpoint:datapath1/satmac/my_SatMac[0].s/genblk1.rom1/z_reg[1] (rising edge-triggered flip-flop clocked by clk) Endpoint: datapath1/satmac/my_SatMac[0].s/m_reg[8] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the ROM memory output to the multiplication register inside the SatMac. Same as the previous result but it is coming from the ROM memory instead of the vector memory. These two paths should be theoretically of the same length.
fc_6_6_16_1_1 (T = 16)	Startpoint: datapath1/vectorMem/data_out_reg[1] (rising edge-triggered flip-flop clocked by clk) Endpoint: datapath1/satmac/my_SatMac[0].s/m_reg[2] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
fc_6_6_20_1_1 (T = 20)	Startpoint: datapath1/vectorMem/data_out_reg[15] (rising edge-triggered flip-flop clocked by clk) Endpoint: datapath1/satmac/my_SatMac[0].s/m_reg[19] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
Comment	In the previous project, we sat the critical path was going from either the vector memory to the accumulation register. The critical path changed in this project because a pipeline register was introduced after the multiplication step in the SatMac unit.

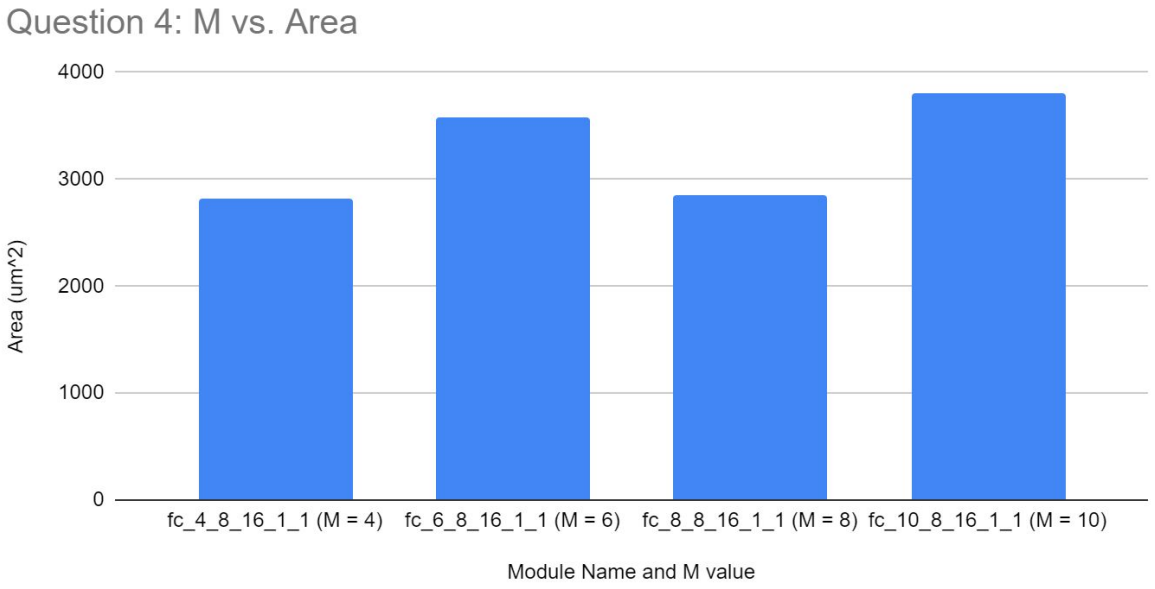
4. Next, we will evaluate how throughput, area, and power scale as M changes. Use your generator to produce four designs with M = 4, 6, 8, and 10, while you keep N=8, P=1 and T=16. Synthesize each design, and graph: (1) power versus M, (2) area versus M, and (3) throughput versus M. Does the location of the critical path change as M changes?

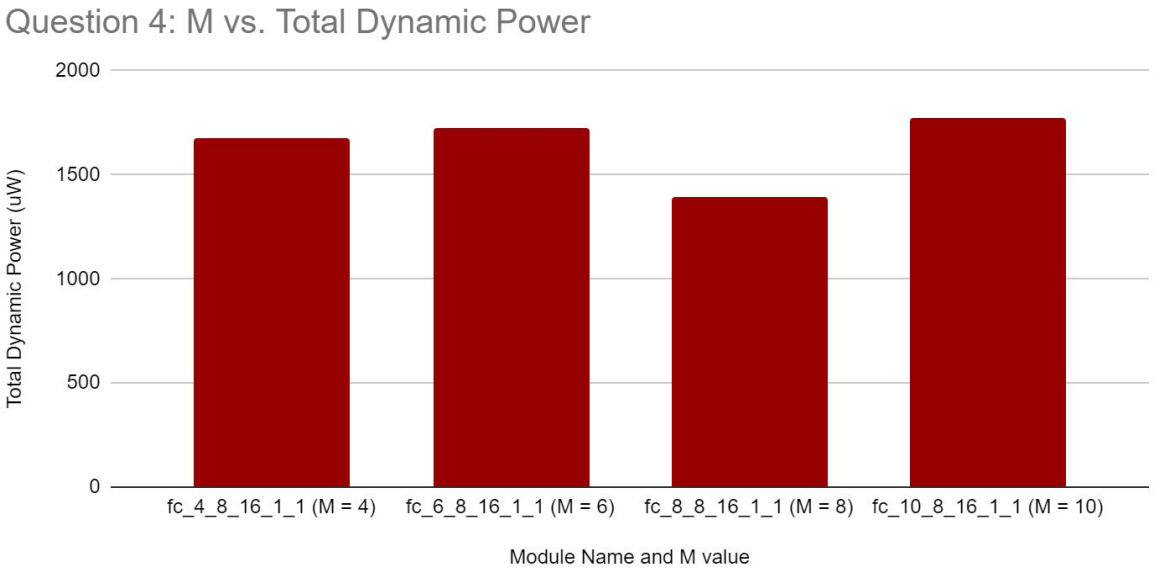
Table 4.1 shows the synthesis results for the required modules in question 4

Module Name	Period (ns)	Frequency (GHz)	Area (um^2)	Cell Internal Power (uW)	Net Switching Power (uW)	Total Dynamic Power (uW)	Cell Leakage Power (uW)
fc_4_8_16_1_1 (M = 4)	1	1.00	2820.39797	1244.8	435.1061	1679.9061	59.9821
fc_6_8_16_1_1 (M = 6)	1.18	0.85	3570.517973	1237.7	490.636	1728.336	75.1733
fc_8_8_16_1_1 (M = 8)	1.2	0.83	2857.371976	1050.2	343.5273	1393.7273	60.2184
fc_10_8_16_1_1 (M = 10)	1.2	0.83	3796.883966	1252.1	519.5376	1771.6376	81.3778

Table 4.2 shows the throughput calculations for each of the required modules in question 4

Module Name	M	N	P	C	Frequency (GHz)	Throughput (data inputs/s)
fc_4_8_16_1_1 (M = 4)	4	8	1	72	1.00	111111111
fc_6_8_16_1_1 (M = 6)	6	8	1	108	0.85	62774639
fc_8_8_16_1_1 (M = 8)	8	8	1	144	0.83	46296296
fc_10_8_16_1_1 (M = 10)	10	8	1	180	0.83	37037037





In the two graphs above, we see that the power trend is not obeyed by the third module. This might be because although every design was synthesized to have slack 0, it might have been the case that the other modules could have been synthesized to a slightest faster clock period and still be at slack 0. This small change might change the overall structure of the design.

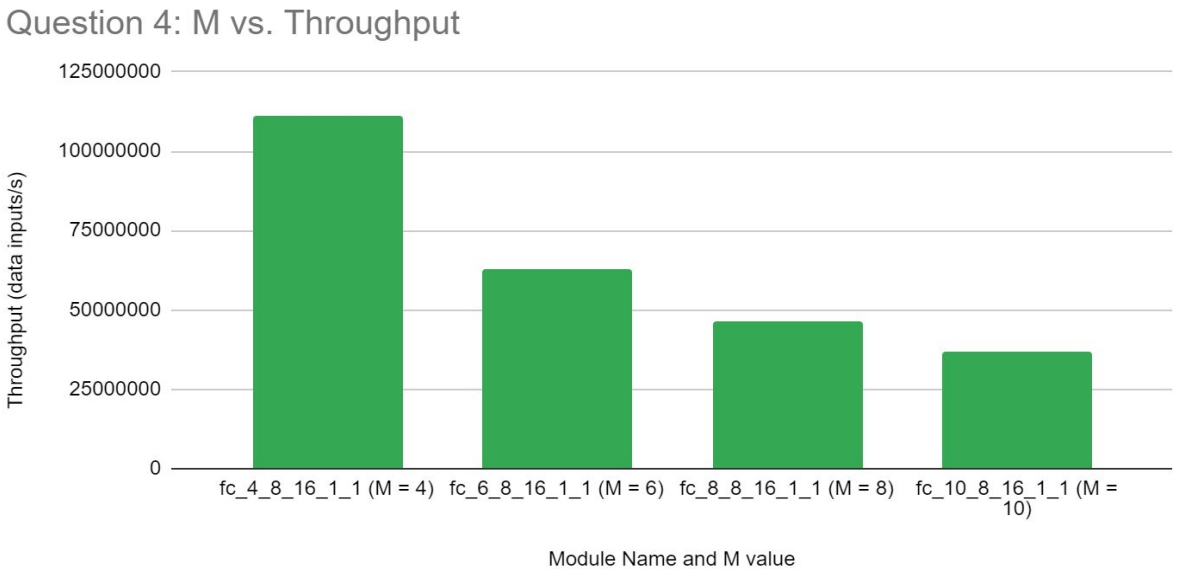


Table 4.3 Shows the critical paths for the required modules in question 4

Module Name	Critical Path
fc_4_8_16_1_1 (M = 4)	Startpoint: datapath1/vectorMem/data_out_reg[13] (rising edge-triggered flip-flop clocked by clk) Endpoint: datapath1/satmac/my_SatMac[0].s/m_reg[6] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
fc_6_8_16_1_1 (M = 6)	Startpoint: datapath1/vectorMem/data_out_reg[4] (rising edge-triggered flip-flop clocked by clk) Endpoint: datapath1/satmac/my_SatMac[0].s/m_reg[15] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
fc_8_8_16_1_1 (M = 8)	Startpoint: datapath1/vectorMem/data_out_reg[1] (rising edge-triggered flip-flop clocked by clk) datapath1/satmac/my_SatMac[0].s/m_reg[15] (rising edge-triggered flip-flop clocked by clk)

	Description: The critical path goes from the vector memory output to the multiplication register inside the SatMac.
fc_10_8_16_1_1 (M = 10)	Startpoint: datapath1/vectorMem/data_out_reg[9] (rising edge-triggered flip-flop clocked by clk) datapath1/satmac/my_SatMac[0].s/m_reg[15] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
Does the location of the critical path change as M change?	From the descriptions of the critical path of this table, we can see that as M changes, the critical path of the system does not change.

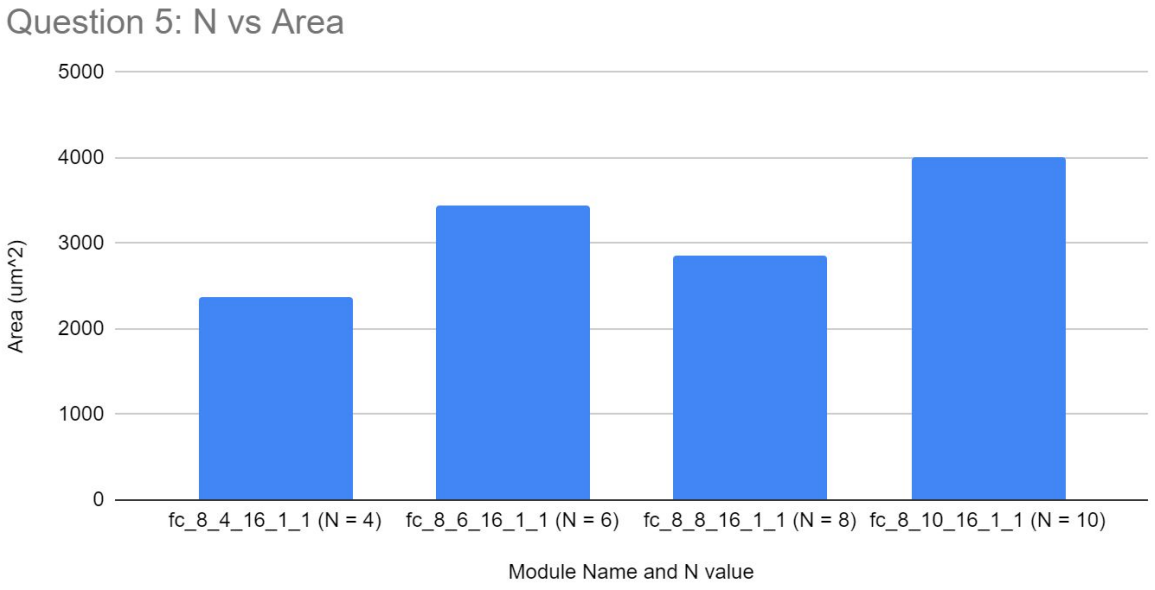
5. Now, we will repeat the previous question while N changes. Use your generator to produce four designs with N=4, 6, 8, and 10. Set M=8, P=1, T=16, and R=1. Synthesize each design, and graph: (1) power versus N, (2) area versus N, and (3) throughput versus N. Does the location of the critical path change as N change?

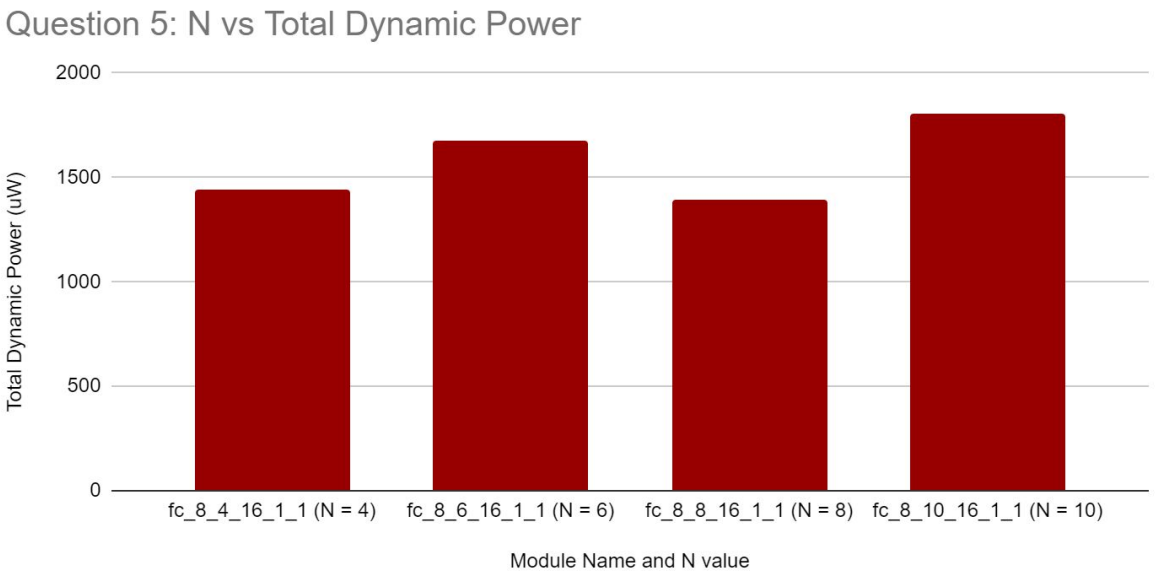
Table 5.1 shows the synthesis results for the required modules in question 5

Module Name	Period (ns)	Frequency (GHz)	Area (um^2)	Cell Internal Power (uW)	Net Switching Power (uW)	Total Dynamic Power (uW)	Cell Leakage Power (uW)
fc_8_4_16_1_1 (N = 4)	1	1.00	2380.167975	1002.9	438.7501	1441.6501	51.9208
fc_8_6_16_1_1 (N = 6)	1.18	0.85	3436.71997	1151.3	524.2079	1675.5079	73.8988
fc_8_8_16_1_1 (N = 8)	1.2	0.83	2857.371976	1050.2	343.5273	1393.7273	60.2184
fc_8_10_16_1_1 (N = 10)	1.2	0.83	4009.417965	1320	484.2173	1804.2173	84.9788

Table 5.2 shows the throughput calculations for each of the required modules in question 5

Module Name	M	N	P	C	Frequency (GHz)	Throughput (data inputs/s)
fc_8_4_16_1_1 (N = 4)	8	4	1	80	1.00	50000000
fc_8_6_16_1_1 (N = 6)	8	6	1	112	0.85	45399516
fc_8_8_16_1_1 (N = 8)	8	8	1	144	0.83	46296296
fc_8_10_16_1_1 (N = 10)	8	10	1	176	0.83	47348485





In the two graphs above, we see that the power trend is not obeyed by the third module. This might be because although every design was synthesized to have slack 0, it might have been the case that the other modules could have been synthesized to a slightest faster clock period and still be at slack 0. This small change might change the overall structure of the design.

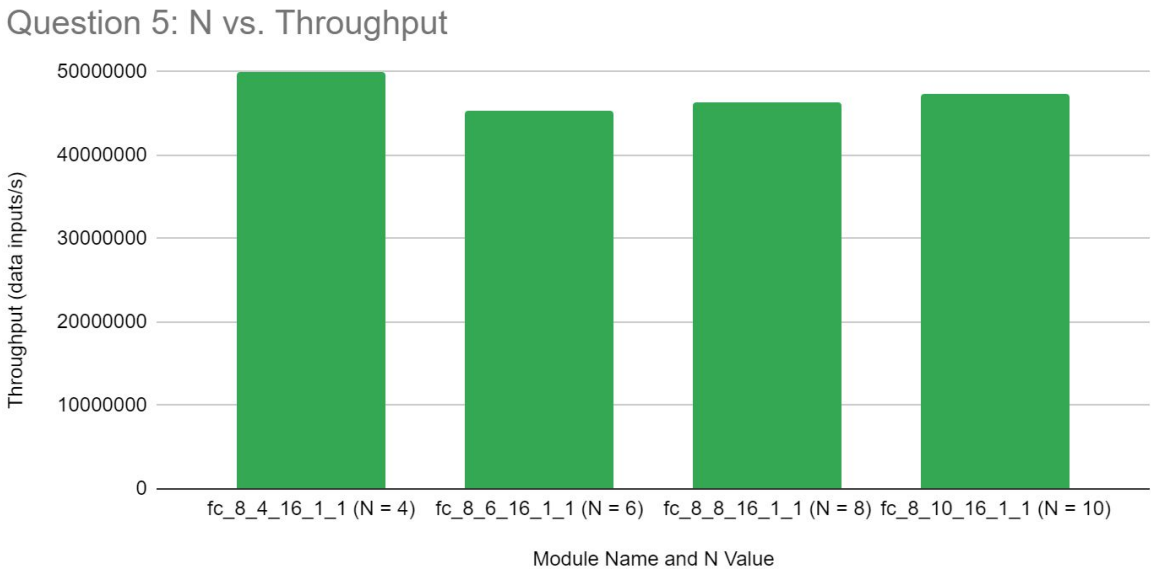


Table 5.3 shows the critical paths for the required modules in question 5

Module Name	Critical Path
fc_8_4_16_1_1 (N = 4)	Startpoint: datapath1/vectorMem/data_out_reg[13] (rising edge-triggered flip-flop clocked by clk) datapath1/satmac/my_SatMac[0].s/m_reg[4] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
fc_8_6_16_1_1 (N = 6)	Startpoint: datapath1/vectorMem/data_out_reg[11] (rising edge-triggered flip-flop clocked by clk) datapath1/satmac/my_SatMac[0].s/m_reg[11] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
fc_8_8_16_1_1 (N = 8)	Startpoint: datapath1/vectorMem/data_out_reg[1] (rising edge-triggered flip-flop clocked by clk) datapath1/satmac/my_SatMac[0].s/m_reg[15] (rising edge-triggered flip-flop clocked by clk)

	Description: The critical path goes from the vector memory output to the multiplication register inside the SatMac.
fc_8_10_16_1_1 (N = 10)	Startpoint: datapath1/vectorMem/data_out_reg[5] (rising edge-triggered flip-flop clocked by clk) Endpoint: datapath1/satmac/my_SatMac[0].s/m_reg[15] (rising edge-triggered flip-flop clocked by clk) **Description:** The critical path goes from the vector memory output to the multiplication register inside the SatMac.
Does the location of the critical path change as N change?	From the descriptions of the critical path of this table, we can see that as N changes, the critical path of the system does not change. This is the same result as in M.

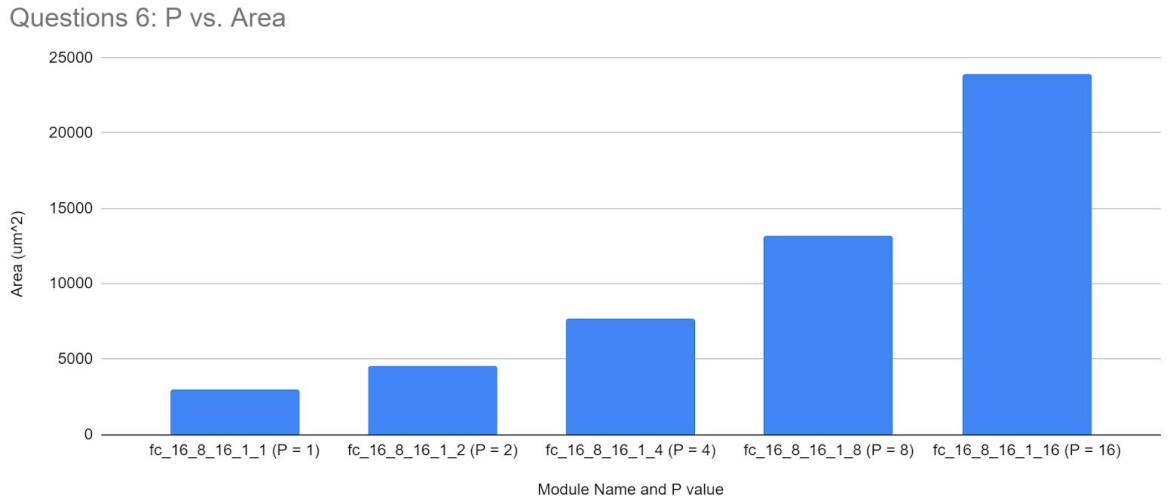
6. Evaluate how the designs change when you increase parallelism, by evaluating P=1, 2, 4, 8, 16, with M=16, N=8, R=1, and T=16. Graph: (1) power versus P, (2) area versus P, and (3) throughput versus P. As parallelism increases, the designs should get faster but also more expensive.

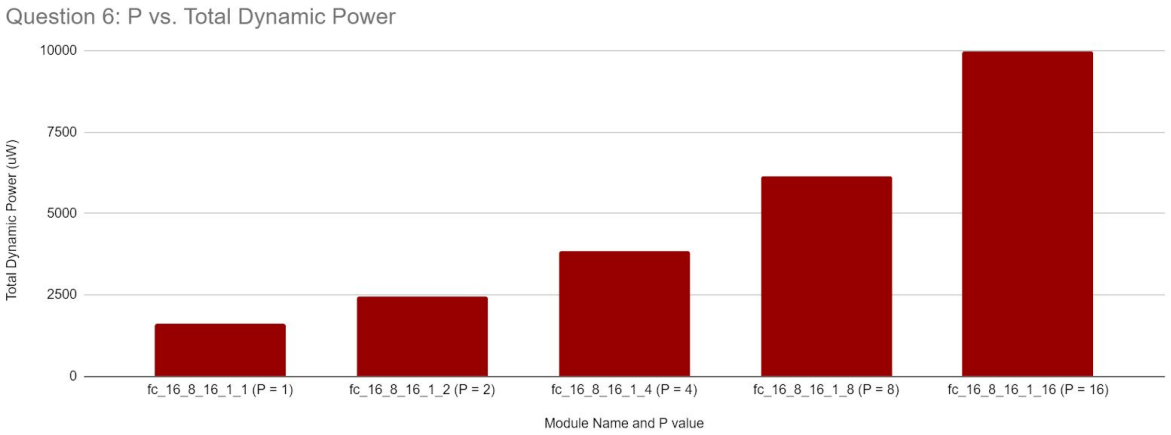
Table 6.1 shows the synthesis results for the required modules in question 6

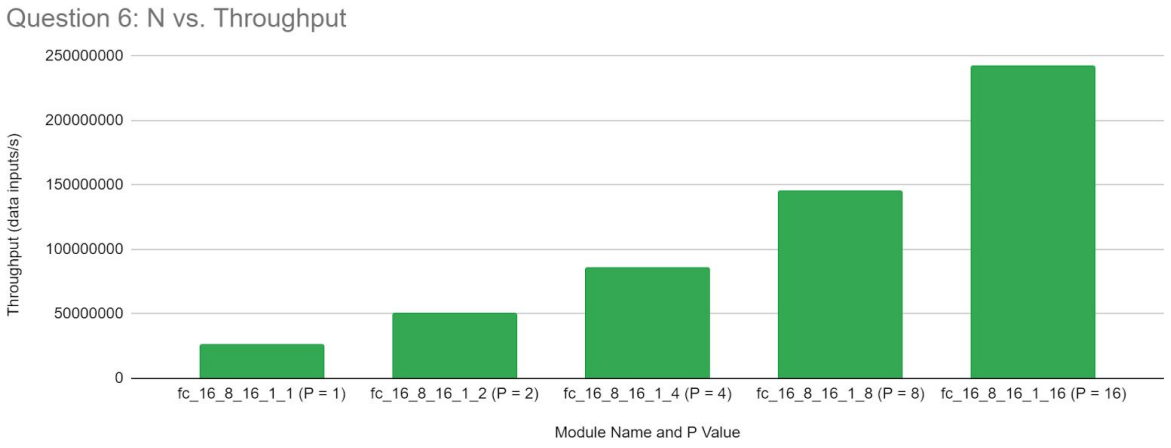
Module Name	Period (ns)	Frequency (GHz)	Area (um^2)	Cell Internal Power (uW)	Net Switching Power (uW)	Total Dynamic Power (uW)	Cell Leakage Power (uW)
fc_16_8_16_1_1 (P = 1)	1.05	0.95	3020.163976	1216.3	406.9967	1623.2967	64.6544
fc_16_8_16_1_2 (P = 2)	1.03	0.97	4553.121961	1721	721.7438	2442.7438	99.5152
fc_16_8_16_1_4 (P = 4)	1.1	0.91	7658.405937	2583.8	1272.2	3856	165.5274
fc_16_8_16_1_8 (P = 8)	1.1	0.91	13220.4659	4129.2	2003.9	6133.1	287.3401
fc_16_8_16_1_16 (P = 16)	1	1.00	23938.40384	6620.1	3370.8	9990.9	513.975

Table 6.2 shows the throughput calculations for each of the required modules in question 6

Module Name	M	N	P	C	Frequency (GHz)	Throughput (data inputs/s)
fc_16_8_16_1_1 (P = 1)	16	8	1	288	0.95	26455026
fc_16_8_16_1_2 (P = 2)	16	8	2	152	0.97	51098620
fc_16_8_16_1_4 (P = 4)	16	8	4	84	0.91	86580087
fc_16_8_16_1_8 (P = 8)	16	8	8	50	0.91	145454545
fc_16_8_16_1_16 (P = 16)	16	8	16	33	1.00	242424242







Which of these designs is the most efficient? Justify your answer quantitatively.

We can think about these questions in terms of area efficiency and power efficiency:

Table 6.3 Shows relations between throughput, area and power

Module Name	Throughput (data inputs/s)	Area (um^2)	Total Dynamic Power (uW)	Throughput/Area	Throughput/Power
fc_16_8_16_1_1 (P = 1)	26455026	3020.16	1623.30	8759	16297
fc_16_8_16_1_2 (P = 2)	51098620	4553.12	2442.74	11223	20919
fc_16_8_16_1_4 (P = 4)	86580087	7658.41	3856.00	11305	22453
fc_16_8_16_1_8 (P = 8)	145454545	13220.47	6133.10	11002	23716
fc_16_8_16_1_16 (P = 16)	242424242	23938.40	9990.90	10127	24265

In terms of area efficiency, that is: the most throughput per unit of area (um^2), the best design is the one with P = 4. In terms of power efficiency, which is the most throughput per unit of power (uW), the best design is the one with P =16.

7. In Part 2, we defined parallelism for this system as having P parallel multiply-accumulate units operating on P outputs concurrently. However, this limits us to using only M multipliers per layer. If you wanted to parallelize further, what could you do?

A pipelining process can be implemented so that multiple states can perform simultaneously. For example, our design includes three states to read the incoming data, perform computations for P rows then output the P values then process the next P values until all M outputs have been completed. This can be improved by pipelining the three stages in a way that more than one matrix-vector multiplication can be performed at a single time. Although in our specifications, the ROM modules inside each SatMac cannot be written to, however, if we were allowed to then after finishing P computations in the process state, the read state would be able to write to each ROM module to replace the values that were just used and no longer needed while the done state can output the values that were just calculated. This would improve performance by allowing the layer to load the next matrix while still using the current one. In a situation where we don't want to write to the matrix in each layer, this would still allow us to read values for the next computation while performing computations as well as outputting them.

Another way to parallelize the process even further is to alter the structure that the mac units are placed in. For a matrix with many more columns than rows, computing multiple vector multiplications wouldn't improve the performance as much. However, if we were to use the multipliers in a way such that more than than one multiplication can be done in a single clock cycle for each row, then matrices with large rows can benefit greatly. For example, in an extreme case where the size of each row, as well as the vector, is N, we can have N multipliers multiplying each element in the row with an element in the vector. The results can either be summed at the same time if the hardware is available or done in combinational layers. This means that each row takes a single clock cycle to process. To make this even faster, registers can be placed between the adders if the products are summed in layers to increase the overall frequency.

8. In Part 3, you were tasked with figuring out how to best optimize the parallelism parameters for your three-layer design, given a multiplier budget B. Explain how you did this, and why your approach is a good solution. Do you see any drawbacks to your approach? How did you evaluate whether or not your system worked well?

We noticed that the throughput of the entire connected network was determined by the layer with the lowest throughput. This meant that if we improved the throughput of the remaining layers then it still wouldn't improve the throughput of the entire system. Thus, the layer with the highest latency would always act as a bottleneck. Our solution must improve this layer before it can improve any other, so the overall program improves one layer at a time. If adding parallelism to a layer is out of the budget or it wouldn't improve the system as a whole then the improvement wouldn't be made. In order to determine at any point which later was the bottleneck of the system, we determined a variable for describing the latency. Based on our design for each individual module, the time it takes for a matrix-vector multiplication to be performed can be determined in the states in the controller. Since the system is connected so that the output of a layer is the input for the next, we determined that the time to take in the data would not be considered. Even if it were, it would not have any significant effects on estimating the latency for a layer.

Each layer performs the matrix-vector multiplication using the process state and the done state. For a layer with P SatMac units, the process state would require (1+2N) clock cycles to determine P result values to be output in the following done state. After outputting the values, the state returns to process. So in total, the process state is used M/P times, and the done state outputs M values in total. This meant that the latency can be measured by the following equation for each layer: $C = (1+2N)(M/P) + M$. So now that we have a method of determining the bottleneck of the system, we can create a process to optimize the values of P so that the maximum C value is minimized. Before the optimization process started, a stack of possible values of P was determined, this was simply the factors of the M variable of the layer. The optimization process was created by using multiple iterations until the optimal values of P were found. At the beginning of each iteration, the values of C would need to be calculated as well as which of the layers was the bottleneck of the system. After determining the exact layer which we have to improve with the pipeline, we check to see if the next value of P for that layer in its respective stack is within the budget. After this is done, we move onto the next iteration where we repeat the process. We know when to stop when the bottleneck layer cannot be improved any further. This means that even if the other layers can be improved because there is enough budget remaining, it wouldn't get implemented since this will not increase the throughput of the whole system but will still increase the cost.

To implement this process into a C++ program, we decided to be efficient and organized to create a class for each layer as well as a class for the network as a whole. This made the program files reusable as well as making it more efficient for testing different cases. The classes can even be used in a system with more than three layers if desired. The class definitions would take care of all the necessary tasks and a single function would implement the optimization of the P values. Since the process is a pattern that has a termination case and has the same functions per iteration, we decided it would be best to perform the optimization in a recurve function. The following is the function for optimizing the parallelism and is simple to follow:

```
void optimize_ps() {
    find_bottleneck();
    find_delta();
    cout << "New bottle neck L = " << bottleneck->layer_number << " with remaining B = " << B;
    if (!bottleneck->factors.empty()) {
        cout <<", Current P = "<< bottleneck->P <<" & next P = " << bottleneck->factors.top() << endl;
    }
    else {
        cout << endl << "This layer has reached it's max value of P, no further optimization will improve throughpout." << endl;
    }
    if (!bottleneck->factors.empty() && B >= (bottleneck->factors.top() - bottleneck->P)) {
        bottleneck->change_P();
        cout << "Optimized " << bottleneck->layer_number << endl;
        print_layers();
        optimize_ps();
    }

}
```

To create a network of layers and test different cases, the following code can be used. The following is an example of a case where we can visualize each iteration and the decision process:

```
N = 3;
M1 = 2;
M2 = 4;
M3 = 8;
B = 20;
layer layer1(1, N, M1);
layer layer2(2, M1, M2);
layer layer3(3, M2, M3);
print_Cs(&layer1, &layer2, &layer3);
network Network(&layer1, &layer2, &layer3);
Network.optimize_ps();
int p1 = Network.net.at(0)->P;
int p2 = Network.net.at(1)->P;
int p3 = Network.net.at(2)->P;
cout << "************************ final results ***************" << endl;
cout << "P1 = " << p1 << " P2 = " << p2 << " P3 = " << p3 << endl;
```

The following is the result of the output. Each iteration of the recursive process will improve the layer with the highest latency. In this specific case, the layer at the last iteration couldn't be further optimized since there is no greater value of P that can be used:

```
 Making layer 1
 Making layer 2
 Making layer 3
Layer 1: P = 1, (N,M1) = (3, 2).  C = 16
Layer 2: P = 1, (M1,M2) = (2, 4).  C = 24
Layer 2: P = 1, (M2,M3) = (4, 8).  C = 80
New bottle neck L = 3 with remaining B = 17, Current P = 1 & next P = 2
Optimized 3
Layer 1: P = 1, (N,M1) = (3, 2).  C = 16
Layer 2: P = 1, (M1,M2) = (2, 4).  C = 24
Layer 2: P = 2, (M2,M3) = (4, 8).  C = 44
 NEW B = 16
New bottle neck L = 3 with remaining B = 16, Current P = 2 & next P = 4
Optimized 3
Layer 1: P = 1, (N,M1) = (3, 2).  C = 16
Layer 2: P = 1, (M1,M2) = (2, 4).  C = 24
Layer 2: P = 4, (M2,M3) = (4, 8).  C = 26
 NEW B = 14
New bottle neck L = 3 with remaining B = 14, Current P = 4 & next P = 8
Optimized 3
Layer 1: P = 1, (N,M1) = (3, 2).  C = 16
Layer 2: P = 1, (M1,M2) = (2, 4).  C = 24
Layer 2: P = 8, (M2,M3) = (4, 8).  C = 17
 NEW B = 10
New bottle neck L = 2 with remaining B = 10, Current P = 1 & next P = 2
Optimized 2
Layer 1: P = 1, (N,M1) = (3, 2).  C = 16
Layer 2: P = 2, (M1,M2) = (2, 4).  C = 14
Layer 2: P = 8, (M2,M3) = (4, 8).  C = 17
 NEW B = 9
New bottle neck L = 3 with remaining B = 9
This layer has reached it's max value of P, no further optimization will improve throughpout.
********** final results ***************
P1 = 1 P2 = 2 P3 = 8
```

9. Lastly, we will evaluate the optimization method you developed for Part 3. For this experiment, set N=4, M1=8, M2=12, and M3=16. Set T=16 and R=1. Then, generate and synthesize five different designs, with B = 3, 10, 20, 30, 36, and 50. Graph (1) power versus B, (2) area versus B, and (3) throughput versus B.

Table 9.1 shows the synthesis results for the required modules in question 9

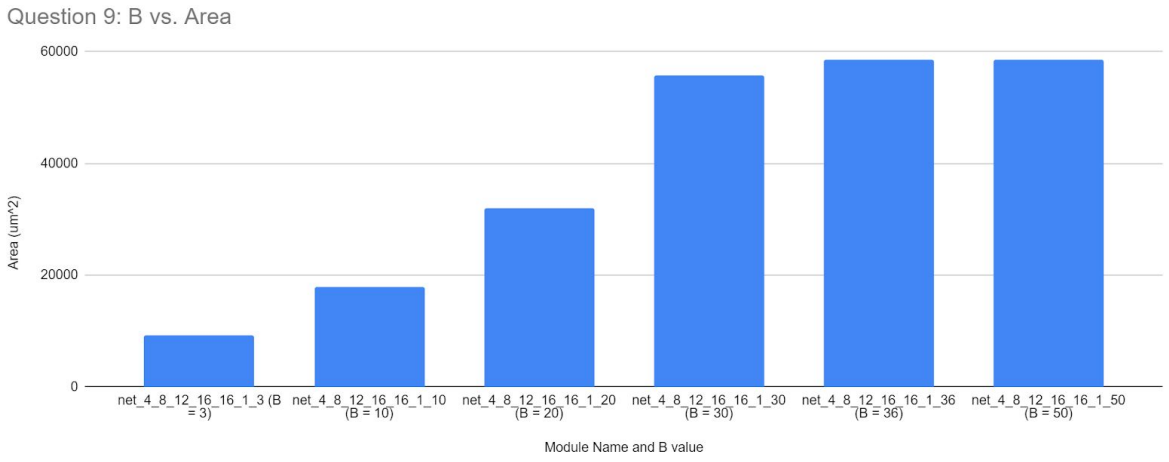
Module Name	Period (ns)	Frequency (GHz)	Area (um^2)	Cell Internal Power (uW)	Net Switching Power (uW)	Total Dynamic Power (uW)	Cell Leakage Power (uW)
net_4_8_12_16_16_1_3 (B = 3)	1.25	0.80	9149.601915	3093.1	242.81	3335.91	204.1075
net_4_8_12_16_16_1_10 (B = 10)	1.25	0.80	17893.02188	4223.4	284.4838	4507.8838	397.1182
net_4_8_12_16_16_1_20 (B = 20)	1.3	0.77	32022.40986	5825.2	537.3035	6362.5035	682.0074
net_4_8_12_16_16_1_30 (B = 30)	1.4	0.71	55698.53776	8676.7	644.8679	9321.5679	1146.3
net_4_8_12_16_16_1_36 (B = 36)	1.4	0.71	58608.84372	8934.6	919.0054	9853.6054	1219.7
net_4_8_12_16_16_1_50 (B = 50)	1.4	0.71	58565.21973	9039.2	1078.7	10117.9	1227.8

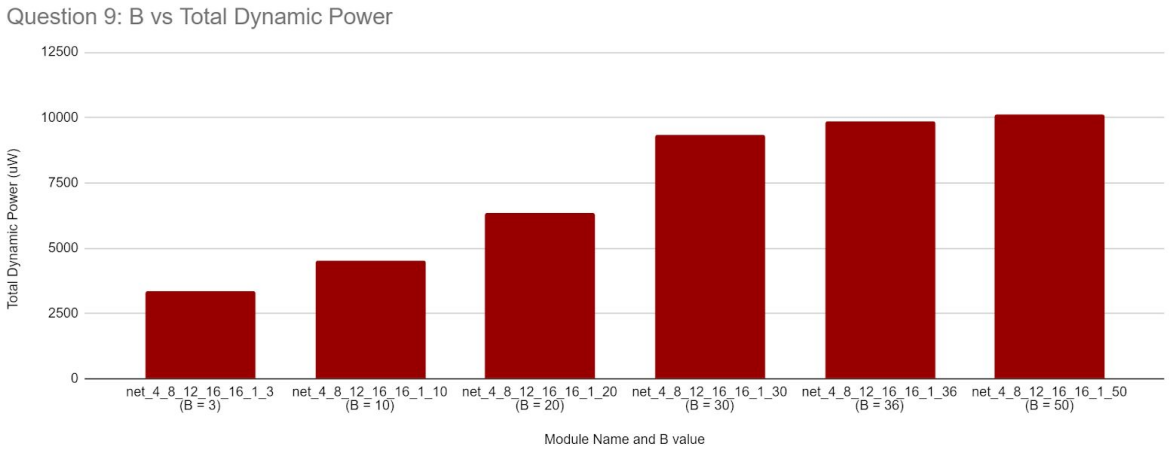
For the following table, we did the following calculation:

From our design, we get that C = (1+2N)(M/P) + M. With N, M1, M2, M3, and the optimized P values, we can calculate the throughput for each layer. Since the network is constraint by the layer with the lowest throughput, we choose that layer and multiply it by the frequency of the synthesized design.

Table 9.2 shows the throughput calculations for each of the required modules in question 9

Module Name	Throughput (data inputs/s)
net_4_8_12_16_16_1_3 (B = 3)	23076923.08
net_4_8_12_16_16_1_10 (B = 10)	40000000
net_4_8_12_16_16_1_20 (B = 20)	69930069.93
net_4_8_12_16_16_1_30 (B = 30)	64935064.94
net_4_8_12_16_16_1_36 (B = 36)	109890109.9
net_4_8_12_16_16_1_50 (B = 50)	109890109.9






Question 9: B vs Throughput
1.25E+8
1.00E+8
7.50E+7
5.00E+7
2.50E+7
0
Throughput (data inputs/s)
net_4_8_12_16_16_1_3 (B = 3)
net_4_8_12_16_16_1_10 (B = 10)
net_4_8_12_16_16_1_20 (B = 20)
net_4_8_12_16_16_1_30 (B = 30)
net_4_8_12_16_16_1_36 (B = 36)
net_4_8_12_16_16_1_50 (B = 50)
Module Name and B value

10. In Part 3, your system assumed it would produce a system with exactly three layers. How would you adapt your generator to create a system with an arbitrary (user specified) number of layers? What challenges would it create?

Since our design process was to first design SystemVerilog files that we would recreate using the compiler, our designs were highly parameterized. We developed each part with the intention to change a few variables or module names to test a system without the use of the C++code compiler. The generate function was the enabler for this aspect of our project, by using it to create an array of module instantiations, we were able to create P SatMac units within the SystemVerilog code without the C++compiler having to do any of the wirings. The following sample of code shows an example of how we created an array.

```
generate
    genvar i;
    for(i=0; i<P; i=i+1) begin: my_sat_mac
        sat_mac_p3_relu#(M,N,T,P,i,L) s(.clk(clk), .reset(reset), .a(a),
        .valid_in(valid_in), .f(f[i]), .valid_out(valid_out[i]), .en_acc(en_acc),
        .addr_w(addr_w));
    end
endgenerate
```

For purposes to fit the description for viewing purposes, the SatMac declaration inside the loop had been broken down into 3 lines, this is not how it is actually written in the code itself. As you can see, we took advantage of the variable i to allow each SatMac unit to be able to identify itself. This was used for deciding the values of the ROM module also generated inside of the SatMacs using the generate function. The parameter L is also passed to allow the SatMacs to know which layer it belongs to as well. From this sample, one can see that our code is heavily reliant on the use of parameters. The parameters simply have to be set in the top module, the rest of the code doesn't have to be touched. Therefore, we can create a general parameterizable module that can create a layer based on the parameters passed. As seen in the figure above, we can then generate multiple layers using the generate function and by passing the genvar parameter, the individual layers would be able to identify itself. Using parameters such as M, N, T, and P, it would not require any additional setup other than the assignments for the ROM values for the individual SatMacs. A top-level module can use this method to create as many layers as it wants by having a parameter itself that determines the number of layers to create. The other parameters required for each module can be passed in the top-level module or by using a generate statement in the declaration of the layer module, this would take the parameter passed during its creation to know which layer it is and assign values of M, N, T, P or any other value. Thus, the top module can simply be declared in the C++compiler by setting up parameters and the values to place in each individual ROM. Since we have already implemented a similar method, we encountered a few challenges across the way however they were not too difficult to overcome. One

may assume that a challenge would be wiring these layers, however as seen in our code, dynamic wiring can be easily done with an array or with the generate function. This method of dynamically instantiating multiple modules can be even taken further, for example, the network that can create multiple layers can be parameterized and used in another top module to create multiple networks. This wouldn’t even add to the complexity of the compiler either since only parameters need to be set up.

11. If you worked with a partner: please explain each partner's contribution to the project. Be specific about what each partner did.

	System Verilog	C ++	Report
Ayman	60 %	40 %	50 %
Mariano	40 %	60 %	50 %

Link to the spreadsheet of all the calculations: https://docs.google.com/spreadsheets/d/1Nic_DvnrWTatiSs_acBaa3TyMcGyLrpDmZT7hQbSKV8/edit?usp=sharing