Project 3: Hardware Generation Tool

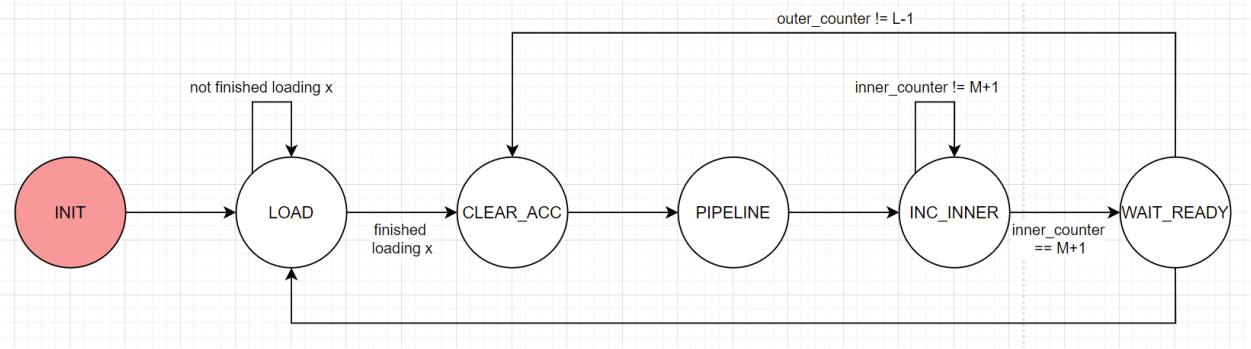
Jason Cheung
Evan Low

Professor Milder
ESE 507
Fall 2021

1.

The control logic is the same from project 2 with an extra stall stage for the pipeline register. It implements a double for loop with a finite state machine. Not all the signals are shown in the diagram for simplicity.

```
for n = 0 to L-1:
    y[n] = 0
    for k = 0 to M-1:
        y[n] += x[n+k]*f[k]
```



Upon reset, the system starts at INIT. Next, the system loads x into memory. If valid and ready signals are 1, the system will stay in this state for N cycles.

Along either transition to CLEAR_ACC, the system clears an address counter that reads the memories. In CLEAR_ACC, the accumulator is cleared. The address counter is cleared on the transition because an extra cycle is needed for the value in the first address to appear at the output due to synchronous read.

Next, the system stalls for a single cycle to load the pipeline register between the multiplier and the adder. The inner_counter is incremented to multiply the next pair of x and f simultaneously as the current product is accumulated.

After, the system completes the inner for loop. Since there is one cycle of delay for the synchronous read and the inner_counter is incremented an extra time in the pipeline stage, the system leaves the state when inner_counter == M+1, not inner_counter == M - 1.

Finally, the system moves to the WAIT_READY state and outputs y_valid = 1.

Since we use counters for M and N, the control module is flexible and their values are parameterizable. As M and N increase, the size of the memories and counters change. The number of cycles needed to compute the double for loop also increases. For this reason, we expect power and area to increase and throughput decreases.

2.

When P = 1, we have a single control module and datapath module. The memories are part of the datapath. For P > 1, we generate P control modules and P datapath modules. We can index the P control and datapath modules from i = 0 to P-1. The ith control module handles the set of y[P*j+i] outputs, where j is from 0 to L/P-1. This is implemented by initializing the inner counter of the ith control module to i upon reset and incrementing the inner counter by P.

Then, we add a top level control module that selects which of the P datapath modules we read from. Since we restrict P to be a factor of L, the final value always comes from the P-1 datapath. The P counter loops from 0 to P-1, and we are finished when we receive the done signal from the P-1 control module.

Since the memories in the datapaths are identical, we factor out the loading logic to the top level control module. This optimization saves us from duplicating the loading counters and logic.

There were other clever optimizations that we considered but did not implement. We can always add more parallelism and pipelining, as in project 2 part 3. We can also buffer the output so that the datapath modules can begin calculating the next iteration without waiting for the ready signal from the top level module. Or, we can buffer the input to overlap loading and processing. A final complicated optimization is that if P > M, each datapath[i] module does not actually read all the addresses, and we do not need to completely duplicate the memories. Each datapath module reads from the set of P*a+b+i addresses, where a is from 0 to L/P-1 and b is from 0 to M-1. If P > M, we can just duplicate the values that the datapath needs.

3.

We produce four designs with (N, M, P) = (16, 4, 1) and T = 8, 12, 16, and 20. We synthesize the designs at the maximum clock frequency and find the corresponding critical path, power, and area.

The critical path is the same in every design and is from one of the memories to the product register. This makes sense because the path contains the multiplier.

The plots of power vs T and area versus T show an increasing relationship. This makes sense because more gates are needed for larger bit widths.

T	Min Period (ns)	Max Freq (GHz)	Power (mW)	Area (um^2)
8	0.72	1.39	1.3113	1625.791979
12	0.916	1.09	1.5241	2363.675968
16	1.01	0.99	1.9636	3261.957964
20	1.08	0.926	2.3707	4203.065953

For T = 8:

```
Startpoint: d/myMemInst_f/z_reg[3]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[7]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

For T = 12:

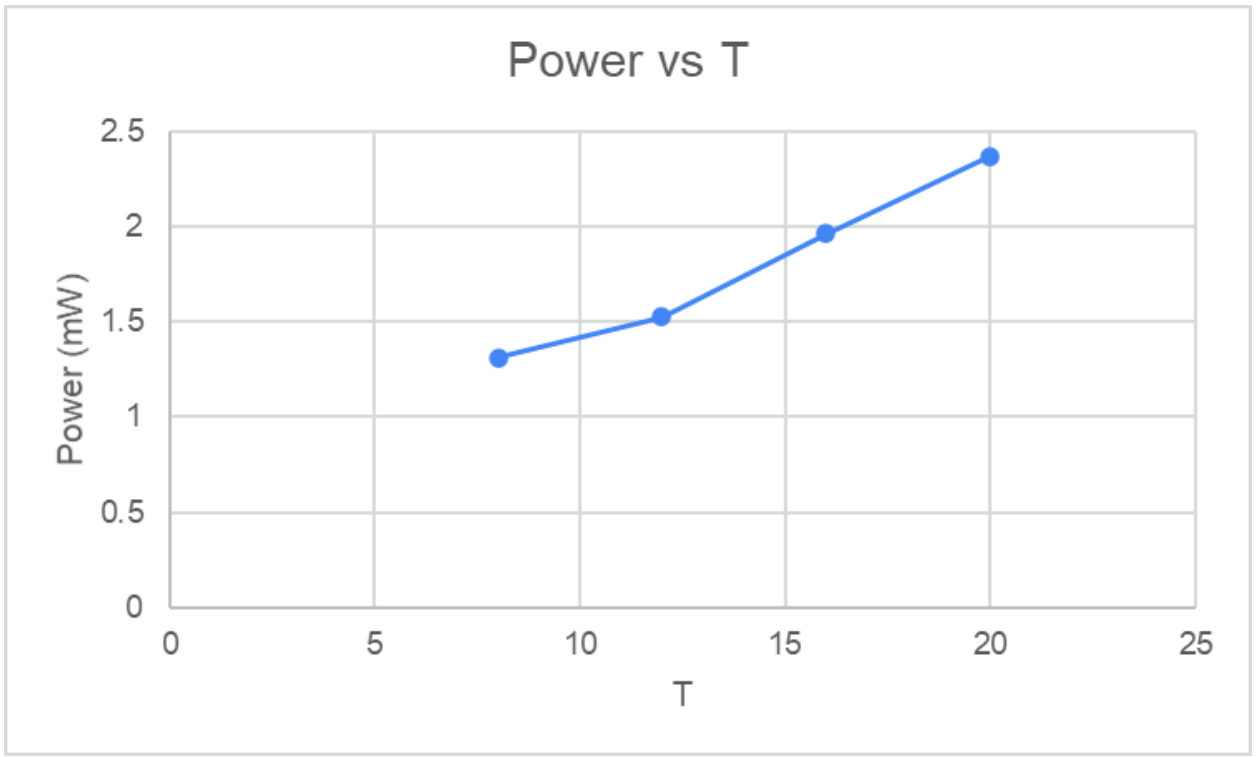
```
Startpoint: d/myMemInst_f/z_reg[3]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[11]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

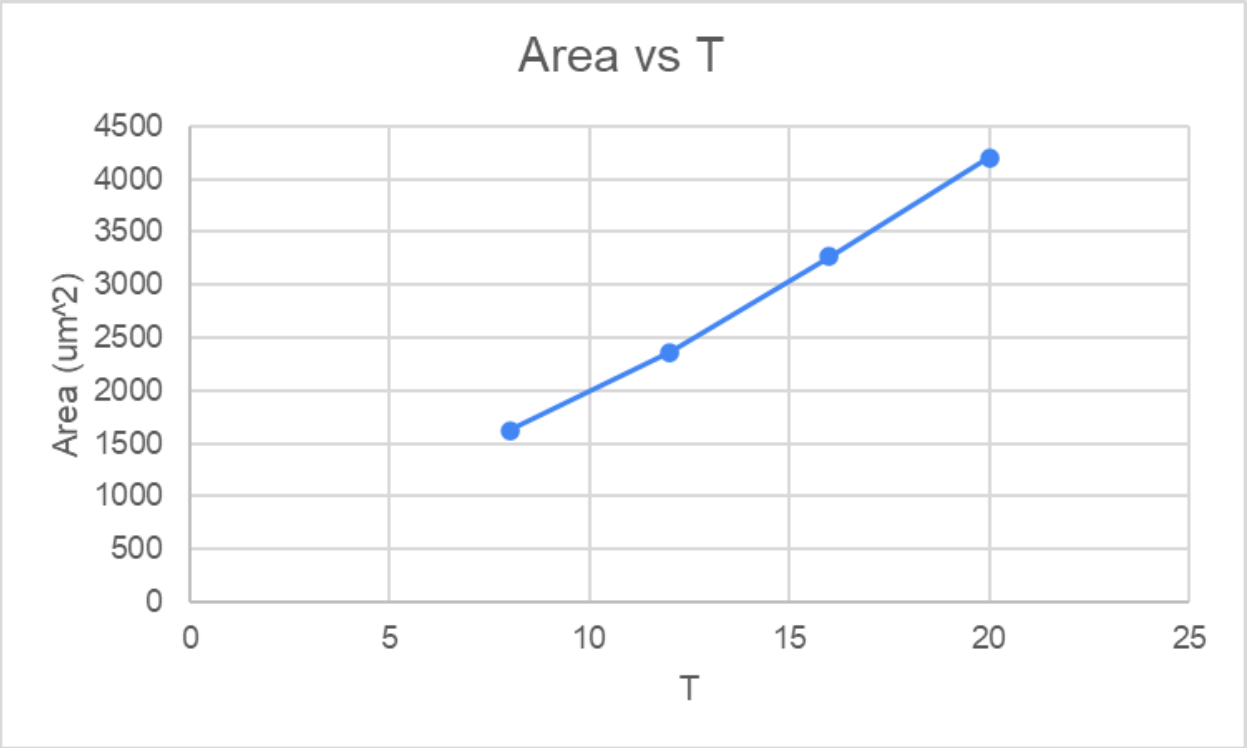
For T = 16:

```
Startpoint: d/myMemInst_x/data_out_reg[11]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[14]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

For T = 20:

```
Startpoint: d/myMemInst_x/data_out_reg[8]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[19]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```





4.

We produce four designs with (N, T, P) = (32, 16, 1) and M = 4, 6, 8, and 10. We synthesize the designs at the maximum clock frequency and find the corresponding critical path, power, area, and throughput.

As before, the critical path is the same in every design and is from one of the memories to the product register. This makes sense because the path contains the multiplier.

For M=4:

```
Startpoint: d/myMemInst_x/data_out_reg[13]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[2]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

For M=6:

```
Startpoint: d/myMemInst_x/data_out_reg[10]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[3]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

For M=8:

```
Startpoint: d/myMemInst_x/data_out_reg[7]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[2]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

For M=10:

```
Startpoint: d/myMemInst_f/z_reg[4]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[2]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

The number of cycles between inputs or between outputs can be calculated by the following formula:

$$c = N + L * (3 + M)$$

If all valid and ready signals are 1, it takes N cycles to load. It takes 3 + M cycles for each of the L outputs because 1 cycle to clear the output, 1 cycle to clear the pipeline, M cycles for the actual inner loop, and 1 cycle in WAIT_READY.

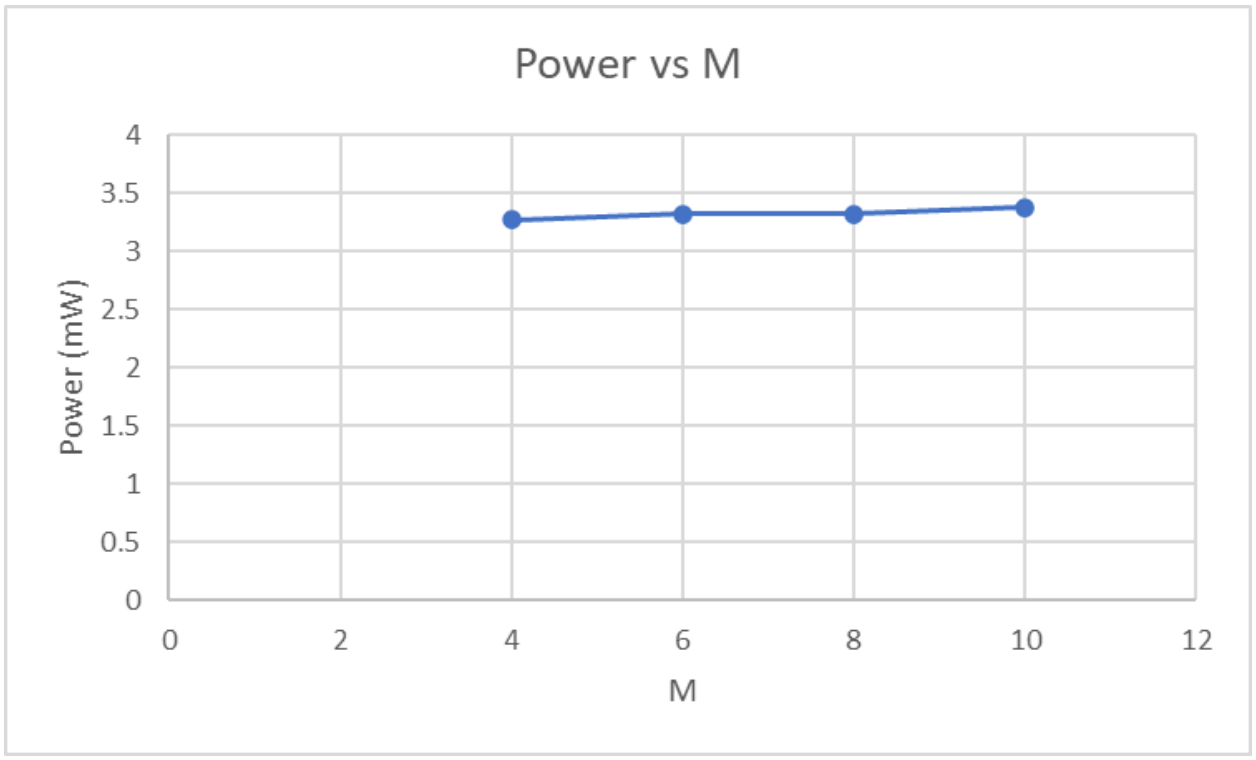
The correctness of the formula is checked by measuring c from the waveforms.

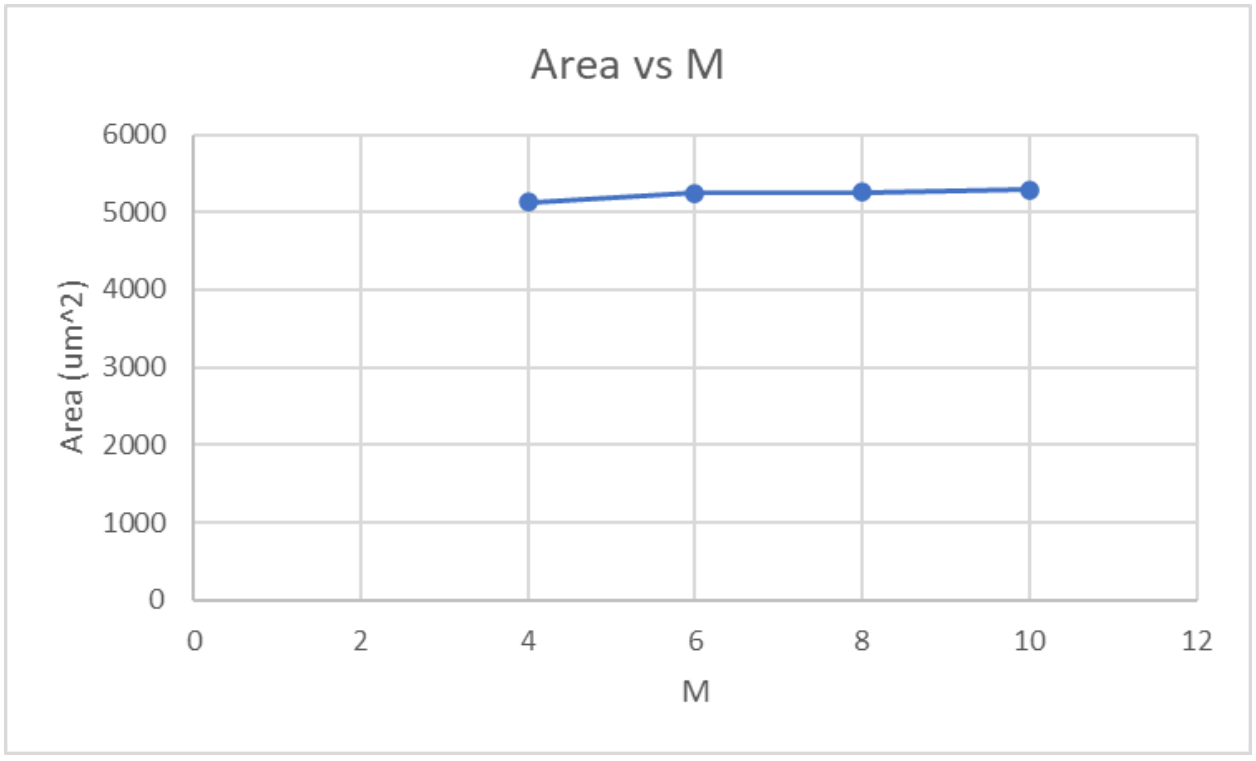
The throughput is calculated by

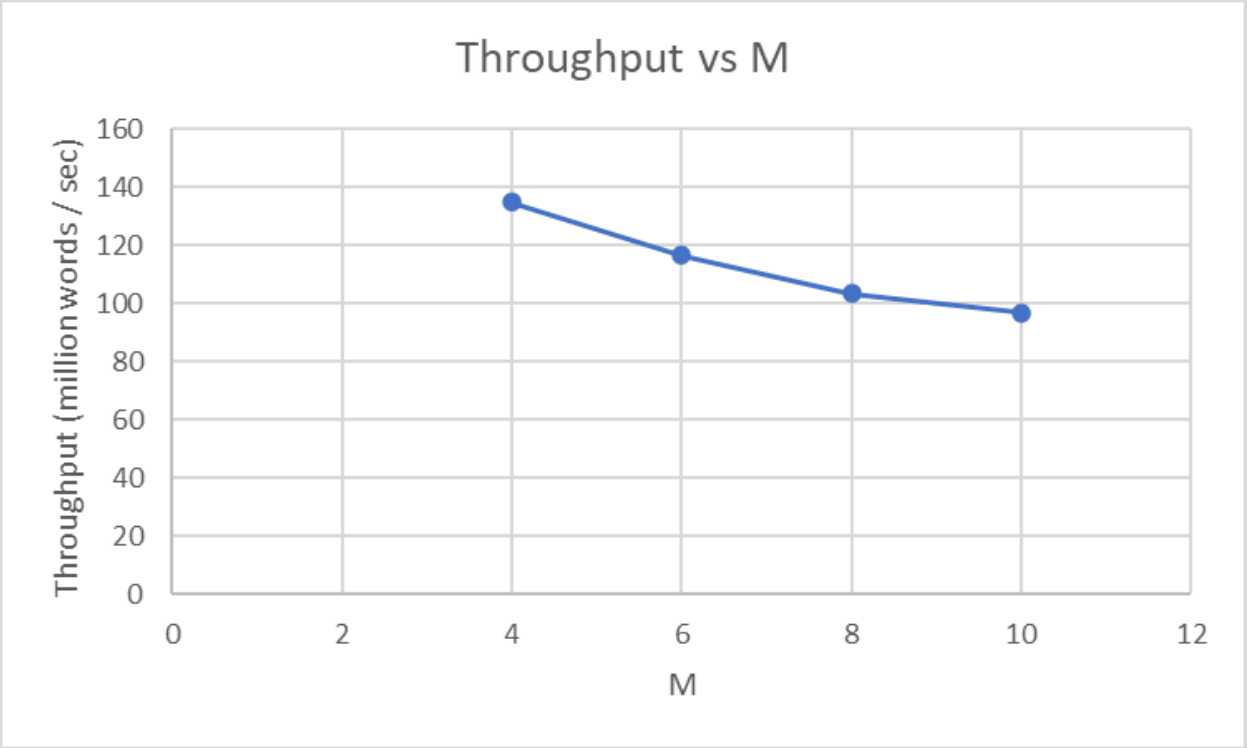
$$\text{Throughput} = N/c * \text{fmax}$$

The maximum frequency, power, and area are roughly the same across all designs because the critical path remains the same, and M increases by only a small amount. N is also a constant, so the throughput depends solely on c. As M increases, c increases, so the throughput decreases.

M	Period (ns)	Freq (GHz)	Area (um^2)	Power (mW)	c (cycles)	Tput
4	1.01	0.99	5124.223940	3.270777	235	135 million words/sec
6	1	1	5247.647939	3.318832	275	116 million words/sec
8	1.01	0.99	5256.957939	3.321283	307	103 million words/sec
10	1	1	5293.665940	3.376105	331	96.7 million words/sec


Power vs M
Power (mW)
4
3.5
3
2.5
2
1.5
1
0.5
0
0
2
4
6
8
10
12
M


Area vs M
Area (um^2)
6000
5000
4000
3000
2000
1000
0
0
2
4
6
8
10
12
M



5.

We produce four designs with (M, T, P) = (8, 16, 1) and N = 16, 32, 64, and 128. We synthesize the designs at the maximum clock frequency and find the corresponding critical path, power, area, and throughput.

For N != 128, the critical path is from one of the memories to the product register. This makes sense because the path contains the multiplier. When N = 128, the critical path is in the address calculation logic. The memories are not real SRAM but flip flops and multiplexers. The multiplexer path may get longer as N increases. There is also an adder in the path to read x[n+k].

N = 16

```
Startpoint: d/myMemInst_x/data_out_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[14]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

N = 3

```
Startpoint: d/myMemInst_x/data_out_reg[11]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[2]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

N = 64

```
Startpoint: d/myMemInst_x/data_out_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/product_reg_reg[1]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

N = 128

```
Startpoint: c/outer_counter_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: d/myMemInst_x/mem_reg[29][0]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
```

The number of cycles between inputs or between outputs can be calculated as before:

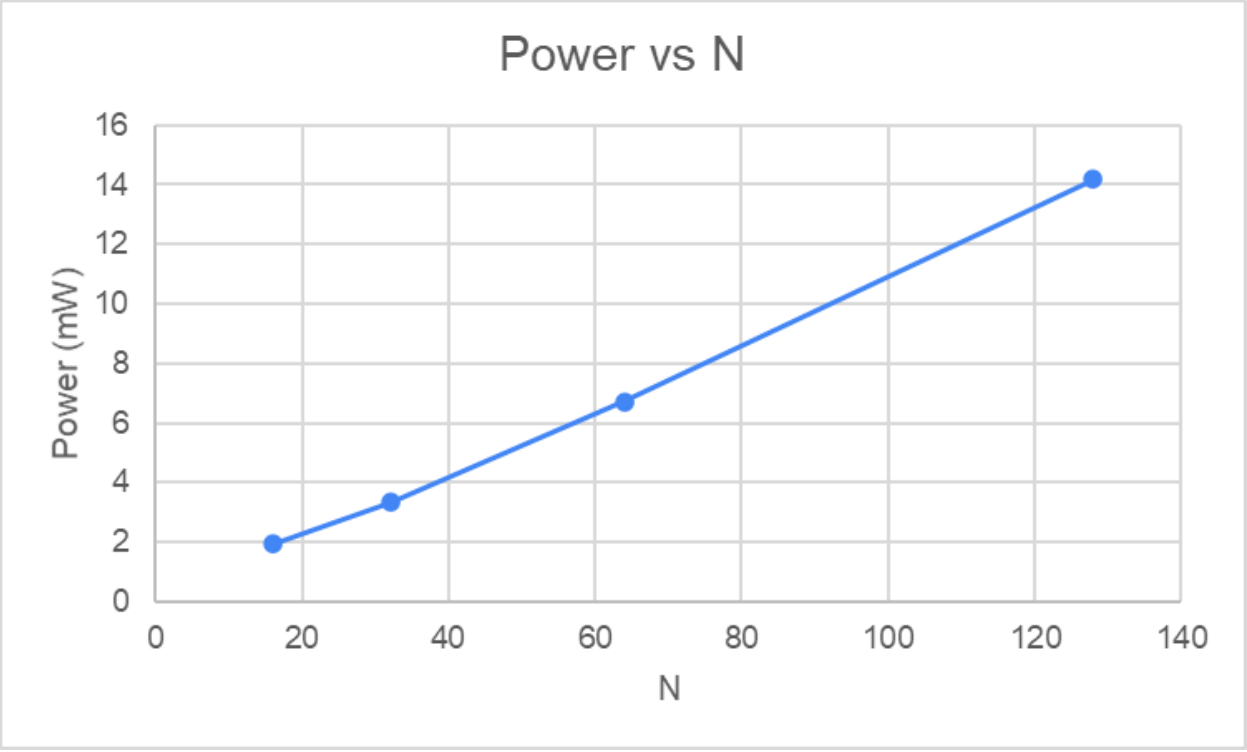
$$c = N + L * (3 + M)$$

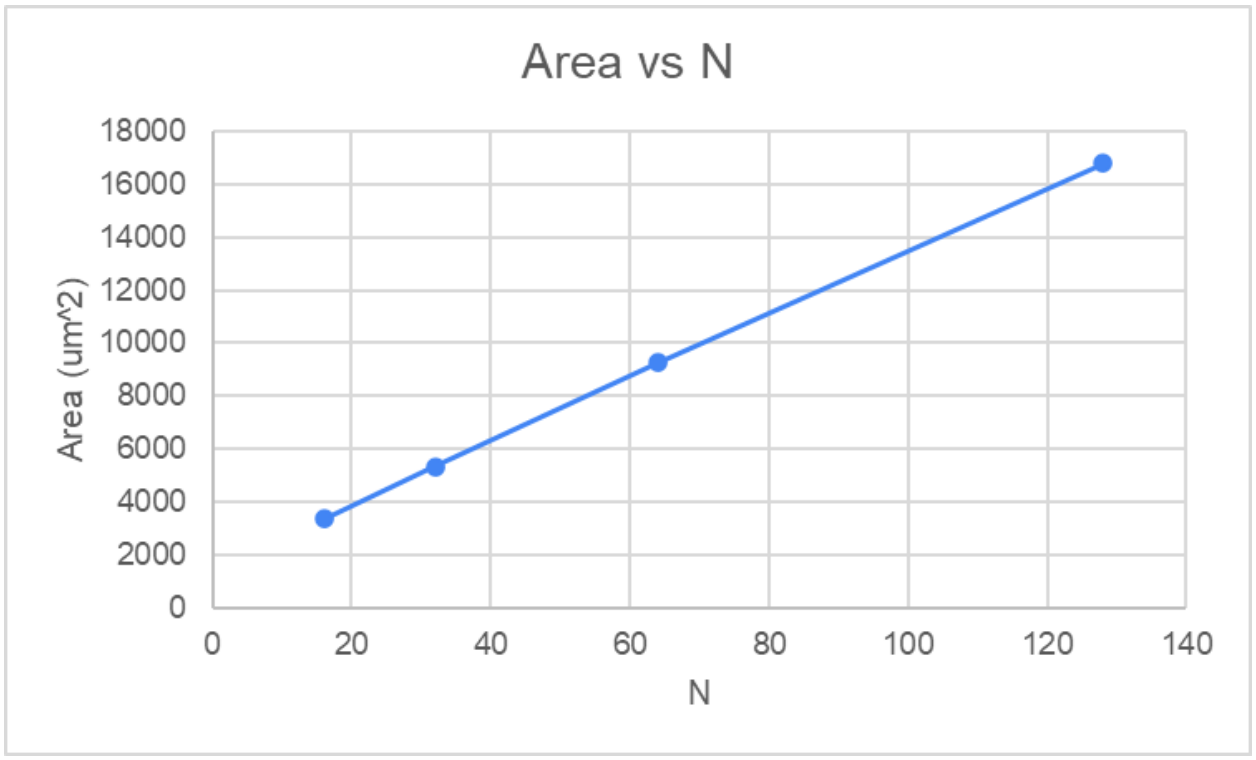
The throughput is calculated by

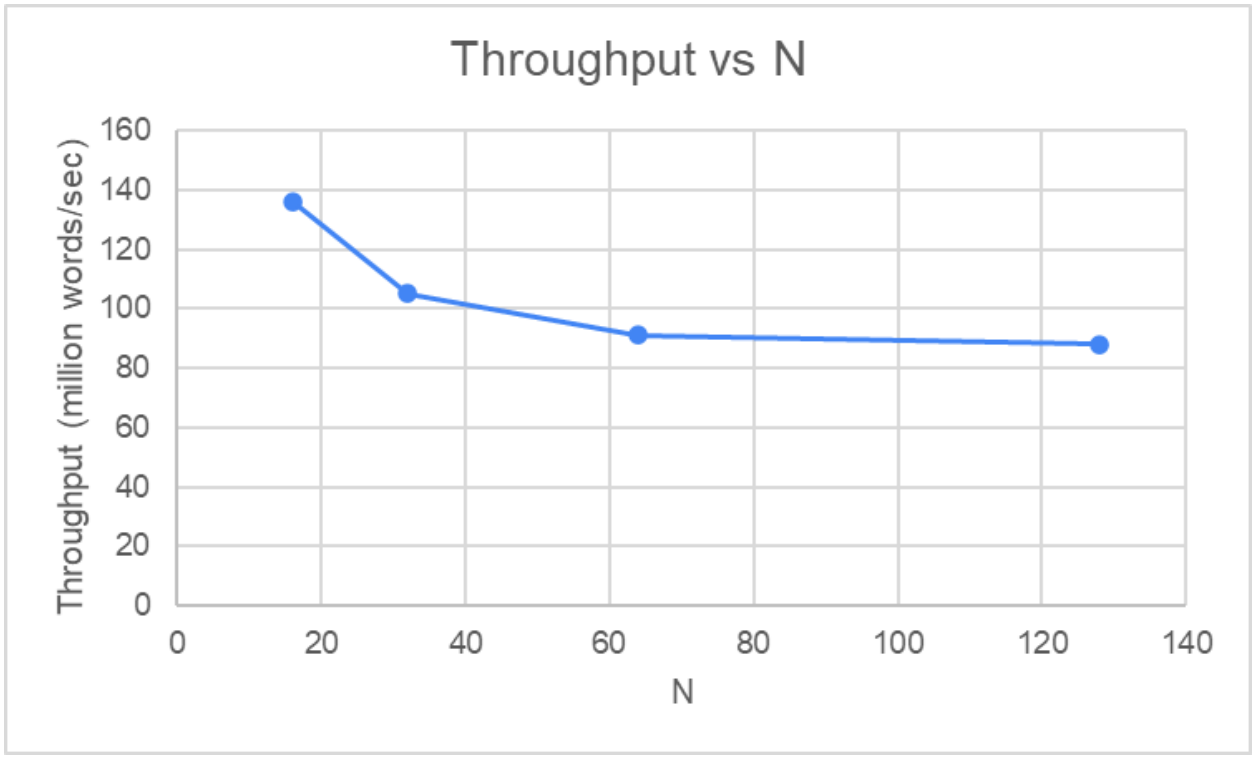
$$\text{Throughput} = N/c * f_{max} = N/(12N-77) * f_{max}$$

The maximum frequency stays roughly the same across all designs because the critical path is the same. The power and area increase because N increases by a considerable amount, increasing the size of the memories. N/(12N-77) is a decreasing function as N increases, so the throughput decreases.

N	Period (ns)	Freq (GHz)	Power (mW)	Area (um^2)	c (cycles)	Tput
16	1.02	0.98	1.9427	3357.717962	115	136 million words/sec
32	0.99	1.01	3.3530	5329.043939	307	105 million words/sec
64	1.02	0.98	6.7229	9252.011884	691	91 million words/sec
128	1	1	14.1925	16808.007787	1459	88 million words/sec




Area vs N
Area (um^2)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
20
40
60
80
100
120
140
N


Throughput vs N
Throughput (million words/sec)
160
140
120
100
80
60
40
20
0
0
20
40
60
80
100
120
140
N

6.

We produce five designs with (N, M, T) = (96, 65, 16) and P = 1, 2, 4, 8, and 16. We synthesize the designs at the maximum clock frequency and find the corresponding critical path, power, area, and throughput. As before, we noted the critical path to be from one of the memories to the product register.

The number of cycles between inputs or between outputs can be calculated as:

$$c = N + (L/P)*(3+M)+P-1$$

With parallelism, the outer loop runs L/P times. There is an offset of P-1 because the last output comes from the P-1 datapath. For the first iteration of the inner loop, the P datapaths finish computing the output at the same time. However, there are P-1 cycles of delay before the P-1 datapath begins the second iteration.

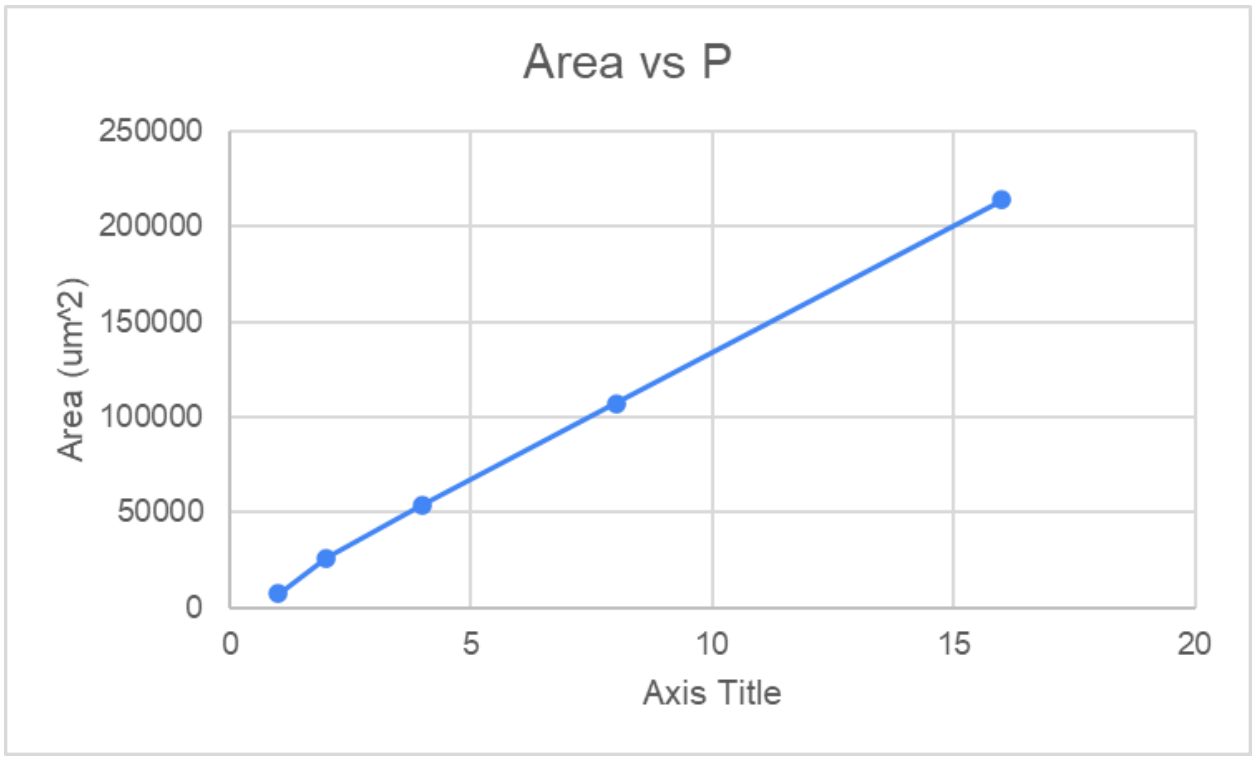
Generating P datapaths will increase the power and area. The benefit is that increasing P decreases c and therefore increases throughput.

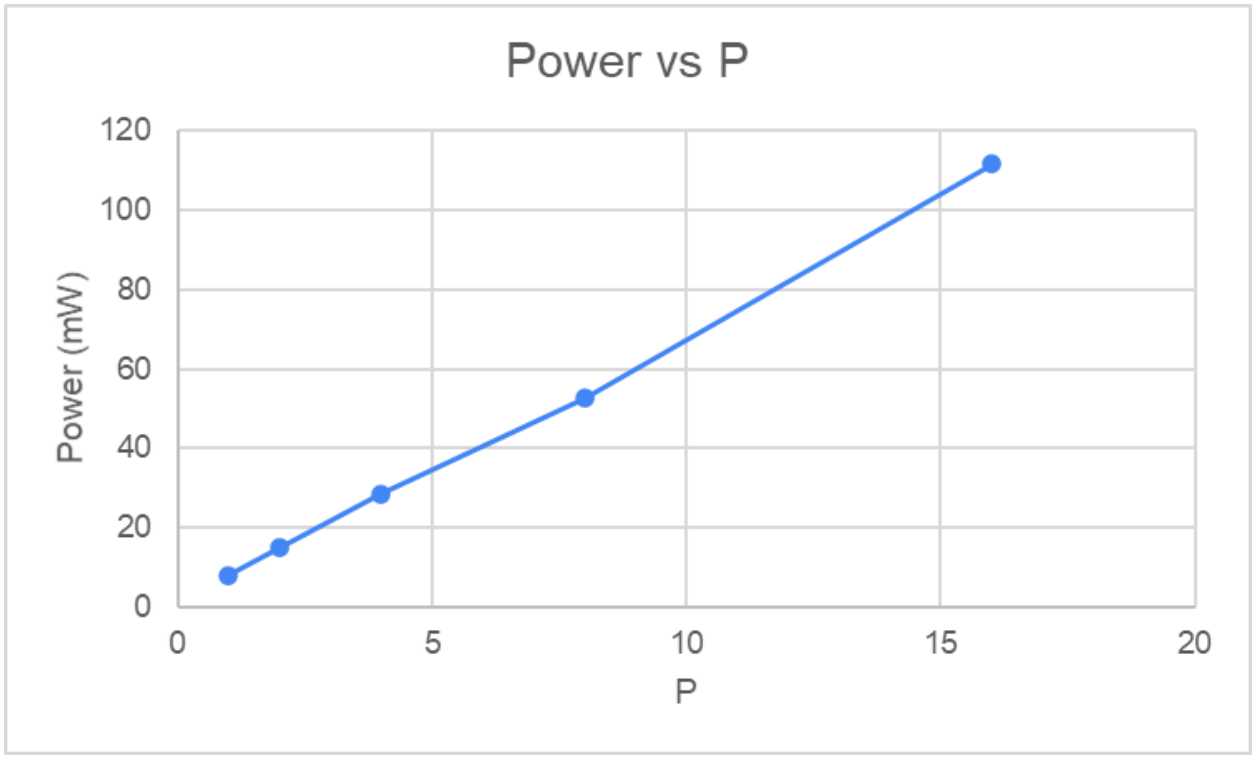
We can calculate efficiency of the designs by calculating the amount of energy per convolution using the following equation:

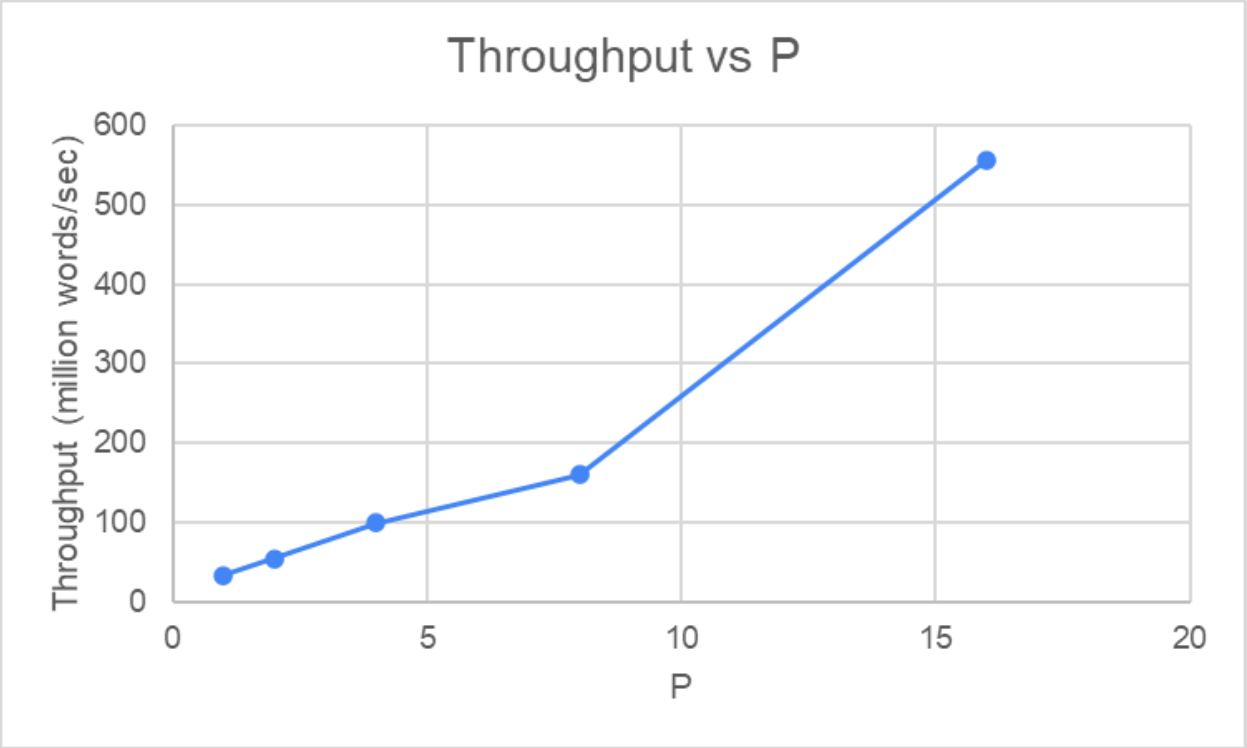
Energy/convolution = Clock period * total power * c

Through our calculations, the design with P = 16 has the least energy consumed per convolution at 19 nJ.

P	Period (ns)	Freq (MHz)	Power (mW)	Area (um^2)	c (cycles)	Tput	Energy/conv (nJ)
1	1.3	769	8.0232	7317.127735	2272	32.5 million words/sec	24
2	1.4	714	14.9655	25878.873689	1185	54 million words/sec	25
4	1.5	667	28.67003	53896.121413	643	99.5 million words/sec	28
8	1.6	625	52.6977	107065.264856	375	160 million words/sec	32
16	1.5	667	111.6925	214149.149724	115	556.5 million words/sec	19


Area vs P
250000
200000
150000
100000
50000
0
Area (um^2)
0
5
10
15
20
Axis Title

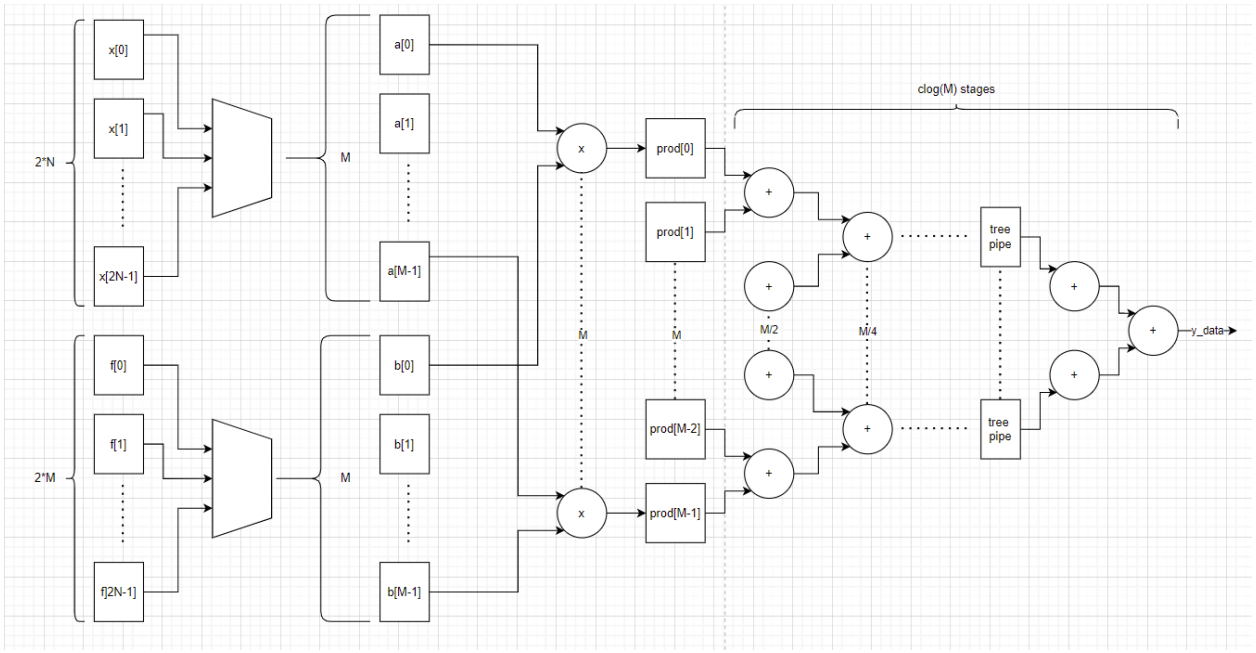

Power vs P
120
100
80
60
40
20
0
Power (mW)
0
5
10
15
20
P



7.

The parallelism in part 2 unrolls the outer loop. The outer for loop runs for L iterations, so L = N-M+1 multipliers per layer is the limit. If we wanted to parallelize further, we could unroll the inner for loop. This is what we did in project 2 part 3. If we unroll the inner loop, we multiply multiple x[i] and f[i] in parallel and sum their values using an adder tree.

A schematic from our project 2 that completely unrolls the inner loop is shown. Combined with unrolling the outer loop, we would generate this datapath P times. However, there is no benefit to completely unroll both loops and compute all L values in a single cycle because we can only output 1 value per cycle. It would also be very expensive. Realistically, we would unroll the loop by a smaller amount. We notice these limitations when we allocate multipliers in the next question.



8.

Our first approach to this problem was with a greedy algorithm. First, we approximate c for each layer by using equation 6. This approximation underestimates c because it treats each layer as independent. Then, the greedy algorithm allocates P to minimize the total c. This can be done through a heuristic or with exhaustive search. Since we restrict P to be a factor of L, the search space is small.

For N = 64, M1 = 33, M2 = 9, M3 = 10, T = 16, the search space is:

L1 = 32

L2 = 24

L3 = 15

P1 = [1, 2, 4, 8, 16, 32]

P2 = [1, 2, 3, 4, 6, 8, 12, 24]

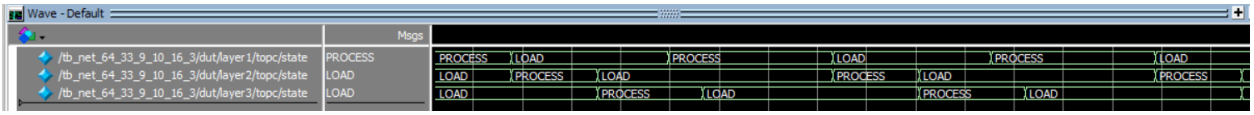
P3 = [1, 3, 5, 15]

The heuristic can allocate multipliers in steps. Start with P1 = 1, P2 = 1, and P3 = 1. At the next step, evaluate the cycles per multiplier saved if we allocate multipliers so P1 = 2, or P2 =2, or P3 = 3, and choose the best one. Advance the index for the layer that was just allocated some multipliers. Repeat until you run out of multipliers. For our

implementation, we use exhaustive search since the search space is small and we can find the guaranteed best solution.

This is a good solution because it minimizes the total number of cycles the system is in the processing state across the 3 layers. However, the drawback is that this algorithm always allocates as many multipliers as it can, even if there is no benefit in throughput. The gains in throughput eventually become bottlenecked by waiting in the load state. This is shown in the waveforms below:

For the same parameters and B = 50, the solution was P1 = 32, P2 = 12, and P3 = 5.



Notice that the layers overlap often in the LOAD state, meaning they are waiting for each other.

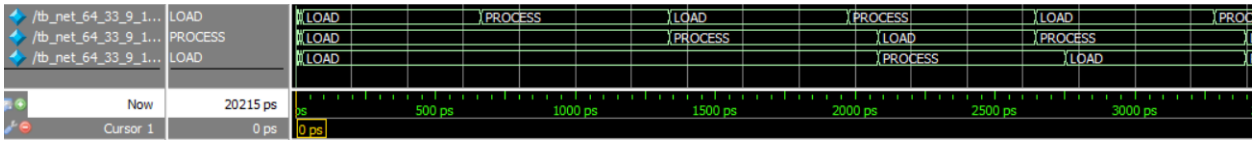
We can improve the algorithm by setting a maximum limit on how many cycles are actually saved. It is desirable for process_i+1 < load_i so that process_i does not need to wait for layer_i+1 to be ready. However, we learned that there is no benefit to being much faster than load_i. We can set this limit by again assuming the layers are independent and approximating load_i = Li.

The drawback to this approach is that the assumptions are only an approximation. Of course there are interlayer dependencies as the layers must wait for each other to be ready. To find the guaranteed optimal solution, we can directly calculate c and exhaustively search for the design that minimizes c. Decreasing c increases the throughput. We can afford to do this search because the search space is small.

We calculate c by simulating at least the first 3 cycles in software. We need at least 3 cycles so that the entire pipeline is filled.

Assuming all valid and ready signals are 1, we know that load for layer 1 always takes N cycles, and process for layer 3 takes L/P*(3+M)+P-1 cycles. We also know that load_i + process_i must overlap with process_i+1 and load_i+1 because they wait for each other.

For the first set of inputs, all y_ready signals are 1 since all layers are in load state. This means that the number of cycles for the first process state in each layer is L/P*(3+M)+P-1.



int load1 = N // always

int process1 = L1/P1*(3+M1)+P1-1

int process2 = L2/P2*(3+M2)+P1-1

int process3 = L3/P3*(3+M3)+P3-1 // always

for at least 3 cycles simulate:

load2 = L1/P1*(3+M1)+P1-1 // cycles if layer1 was independent

// load state can be shorter or longer depending on overlap/waiting cycles

if (process2 > load1)

load2 -= min(process2 - load1, cycles until first layer1 output is valid)

else

load2 += load1 - (process2)

int load3 = L2/P2*(3+M2)+P2-1 // cycles if layer2 was independent

// load state can be shorter or longer depending on overlap/waiting cycles

if (process3 > load2)

load3 -= min(process3, cycles until first layer2 output is valid)

else

load3 += load2 - process3

```
// we calculate new process lengths by knowing that load_i + process_i overlaps with process_i+1 and load_i+1

int process2 = process3+load3-load2;

int process1 = process2+load2-load1;


// c is load1+process1 (time between inputs) or load3+process3 (time between outputs) or any layers in between

int cycles = load1+process1;
```

We verify the correctness of this software simulation with modelsim waveforms. Since we do an exhaustive search that minimizes c, the algorithm is guaranteed to be optimal. Because exhaustive search does not scale well when we increase the number of layers or parameters, we also described heuristics we can use for good solutions.

To evaluate how well the algorithms work, we compare the unoptimized greedy approach that uses as many multipliers as possible to the exhaustive search. Notice that the optimal designs achieve the same throughput with fewer multipliers.

Greedy

B	P1	P2	P3	c (cycles)
3	1	1	1	1405
8	4	3	1	355
14	8	3	3	215
30	16	8	5	151
50	32	12	5	131

Exhaustive

B	P1	P2	P3	c (cycles)

3	1	1	1	1405
8	4	3	1	355
14	8	3	3	215
30	16	4	3	151
50	32	4	3	131

9.

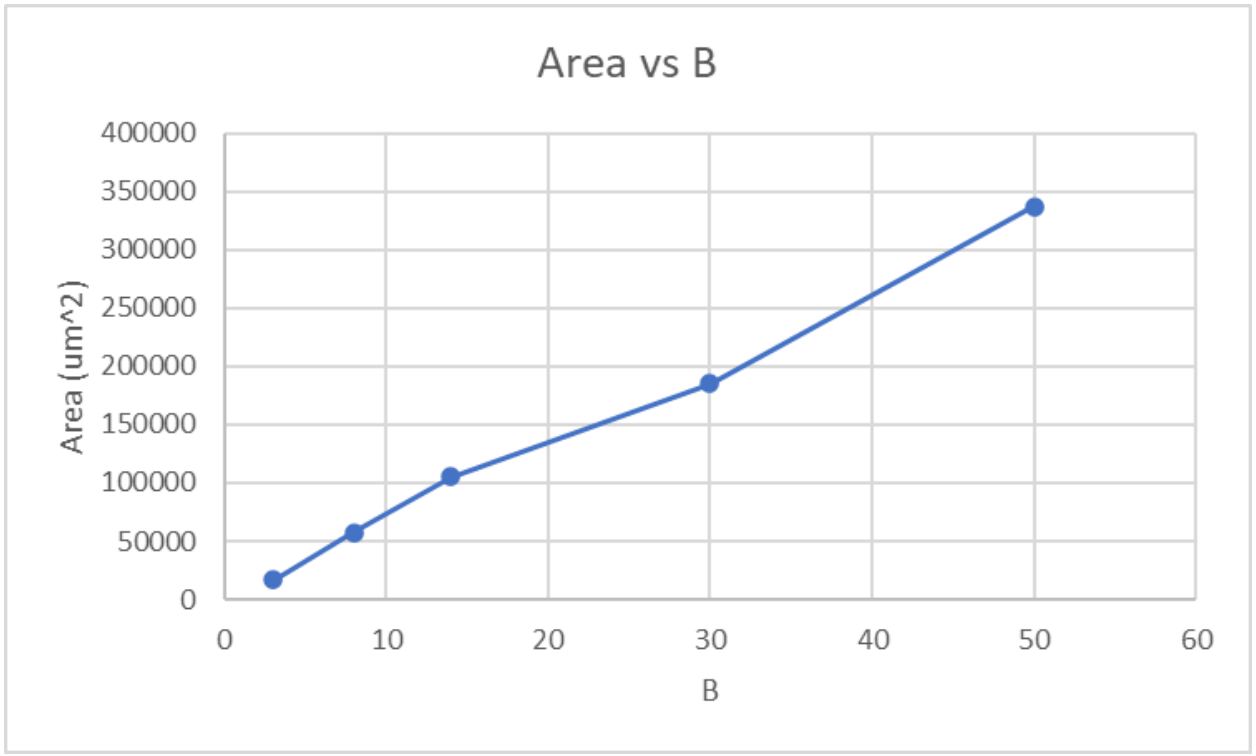
We produce five designs with (N, M1, M2, M3, T) = (64, 33, 9, 10, 16) and B = 3, 8, 14, 30, and 50. We synthesize the designs at the maximum clock frequency and find the corresponding critical path, power, area, and throughput. As before, we noted the critical path to be from one of the memories to the product register.

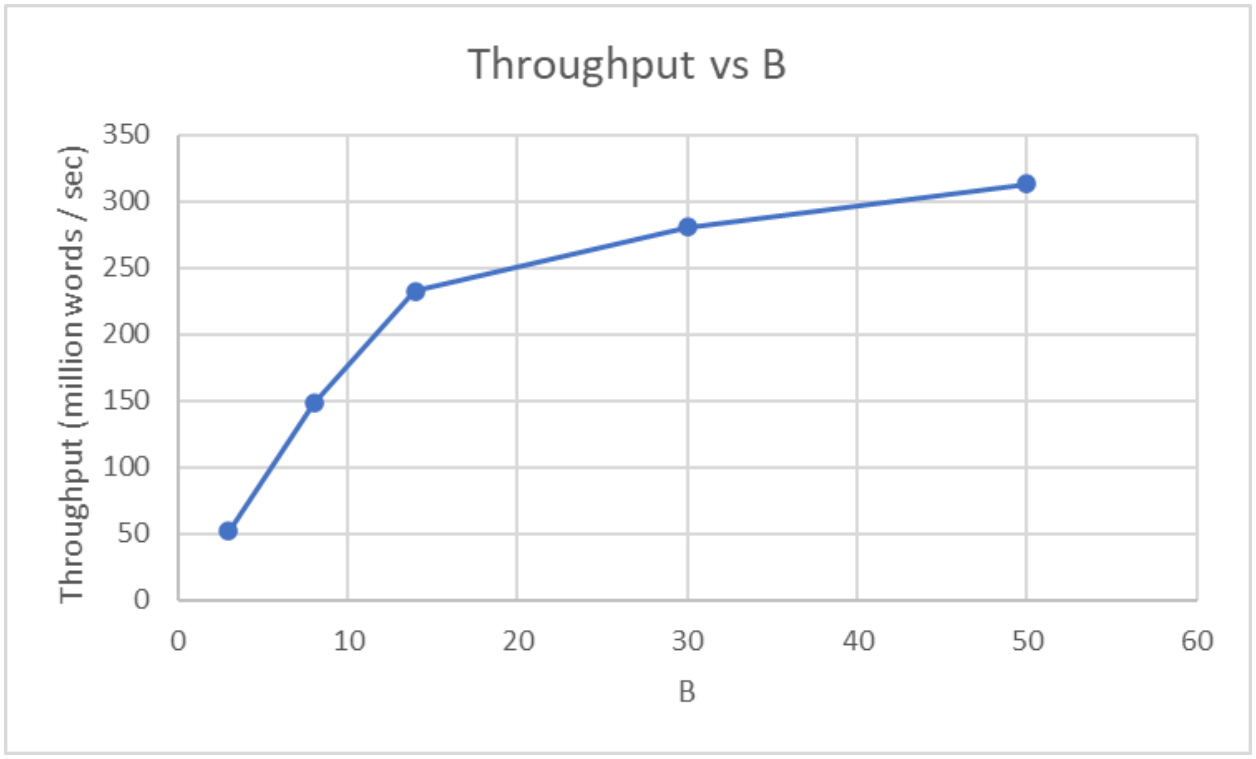
We calculated c through simulation and verified the values from Modelsim waveforms. The values calculated for P1, P2, and P3 are shown in the previous table.

Allocating more multipliers will increase the power and area. The benefit is that increasing P decreases c and increases throughput. Notice that the throughput plot plateaus, meaning that the rate of improvement in the throughput decreases. This makes sense because we are eventually bottlenecked by loads. In terms of performance and energy efficiency, B = 14 (P1 = 8, P2 = 3, and P3 = 3) appears to be the best choice because it considerably increases the throughput and uses less energy than other designs.

B	Period (ns)	Freq (MHz)	Area (um^2)	Power (mW)	c (cycles)	Tput	energy/N (nJ)
3	0.88	1136	17111.779762	15.43014	1405	51.8 million words/sec	19
8	1.22	820	57359.175347	34.4642	355	148 million words/sec	15
14	1.28	781	105615.830839	58.7253	215	233 million words/sec	16
30	1.51	662	185144.244016	89.2487	151	281 million words/sec	20

50	1.56	641	336961.082469	154.5304	131	313 million words/sec	31


Area vs B
Area (um^2)
400000
350000
300000
250000
200000
150000
100000
50000
0
0
10
20
30
40
50
60
B


Throughput vs B
Throughput (million words / sec)
350
300
250
200
150
100
50
0
0
10
20
30
40
50
60
B

10.

We can create a system with an arbitrary number of layers by modifying the top level module. Currently, our top level module (nettemplate.txt) instantiates three layers and wires them together. Instead, this can be handled by a generate loop. The code should connect ports and set the parameters correctly, particularly N = L of the previous layer. Since the number of inputs decreases with each layer, a user should be aware of setting parameters that make sense (N > M).

Adding more layers and parameters dramatically increases the runtime of our exhaustive search optimization strategy. Our optimization strategy simulates the model in software to determine c. If the search space is too large, we may want to resort to heuristics that we discussed.

11. If you worked with a partner: please explain each partner's contribution to the project. Be specific about what each partner did. (If you worked alone, skip this.)

We pair programmed each part of the project together but gathered synthesis results and made plots separately. Evan made plots and wrote explanations for 3, 5, and 6 and Jason made plots and wrote explanations for 4 and 8. We finalized everything else together.