Stony Brook University
College of Engineering & Applied Sciences

ESE 507.01: Advanced Digital System Design
and Generation – Fall 2025

Final Project Report

Date: 9/27/25 - 12/12/25

Professor Peter Milder

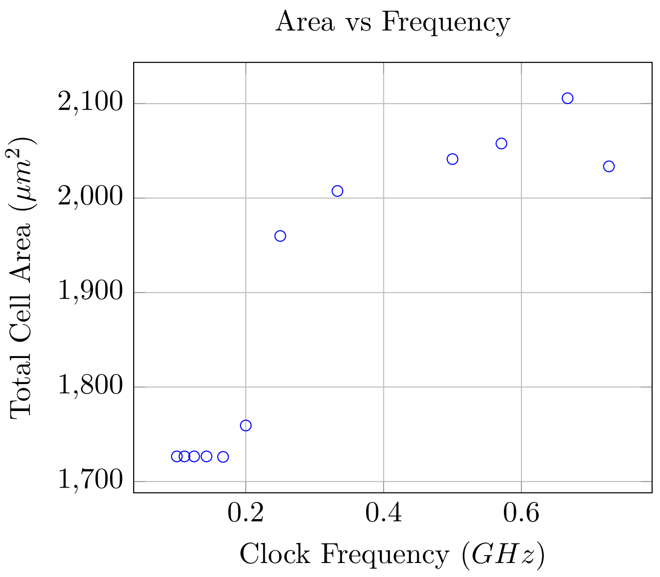
Huabin Wu (115067644)
huabin.wu@stonybrook.edu

&

Ryan Lin (114737153)
ryan.lin@stonybrook.edu

Project Part 1: MAC Unit

1. Use Synopsys DesignCompiler to synthesize your unpipelined design with INW=16 and OUTW=64 for a range of different clock frequencies from slow to fast. Adapt the scripts you used in HW2. For each frequency you try, record the area, power, the critical path location, and whether the timing constraint was met or violated. In your report, make a table that shows this data for each attempted frequency. Make sure you include units on all values you report (here and everywhere else in the report).
Make graphs that show the relationships you found between clock frequency and both area and power. Explain the trends that you observed and explain why they occur. (Make two graphs. On both, show clock frequency on the x-axis; then show area as the y-axis on one graph and power as the y axis on the other.) Make sure use graphs that plot both axes proportionally (like a scatter graph, not a line graph). Only include the design points where the timing constraint is MET.
For each frequency, give a description in your report of where the critical path is. Don't just copy-/paste the endpoints from the synthesis report, but explain logically where the critical path lies in the module. (For example, "the critical path starts at the output of register [register name], passes through logic that computes [description of the logic], and ends at the input to register [register name].")



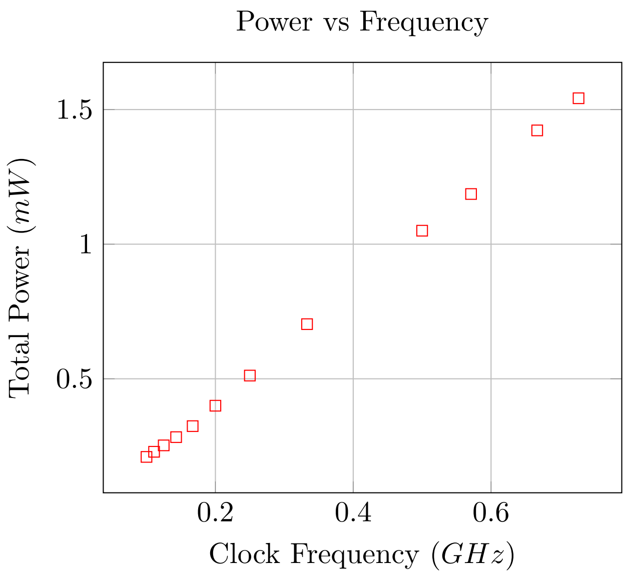


Figure 1: Scatter plots of frequency vs area (left) and frequency vs power (right), only showing slack-met designs.

- **Area vs Frequency:** Area increases slightly with higher clock frequencies due to the need for larger/faster cells to meet timing.
- **Power vs Frequency:** Power rises with frequency since dynamic power scales with f while leakage stays nearly constant.

Frequency (GHz)	Area (μm^2)	Total Power (mW)	Critical Path (Data Arrival, ns)	Slack Status	Slack (ns)
0.800	2092.89	1.7235	input0[4] → out_reg[42] (1.29)	Violated	None
0.727	2033.57	1.5420	input0[7] → out_reg[52] (1.34)	Met	0.0
0.667	2105.66	1.4226	input0[13] → out_reg[41] (1.46)	Met	0.0
0.571	2057.78	1.1863	input0[5] → out_reg[62] (1.71)	Met	0.01
0.500	2041.28	1.0501	input0[0] → out_reg[63] (1.96)	Met	0.01
0.333	2007.50	0.7030	input0[3] → out_reg[63] (2.95)	Met	0.02
0.250	1959.89	0.5120	input0[4] → out_reg[63] (3.97)	Met	0.0
0.200	1759.32	0.4000	input0[1] → out_reg[63] (4.95)	Met	0.0
0.167	1726.07	0.3242	input0[3] → out_reg[63] (5.46)	Met	0.5
0.143	1726.61	0.2834	input0[3] → out_reg[63] (5.46)	Met	1.5
0.125	1726.61	0.2527	input0[3] → out_reg[63] (5.46)	Met	2.5
0.100	1726.61	0.2099	input0[3] → out_reg[63] (5.46)	Met	4.5

Table 1: MAC unpipelined design (INW=16, OUTW=64). Power is sum of dynamic and leakage. All design points shown, including slack violations.

2. Now, repeat the tasks from question 1 for your pipelined design with the same values of INW and OUTW. Additionally, answer the following: Did pipelining help make this module faster? Explain why or why not and show how this is reflected in the synthesis data and critical path.

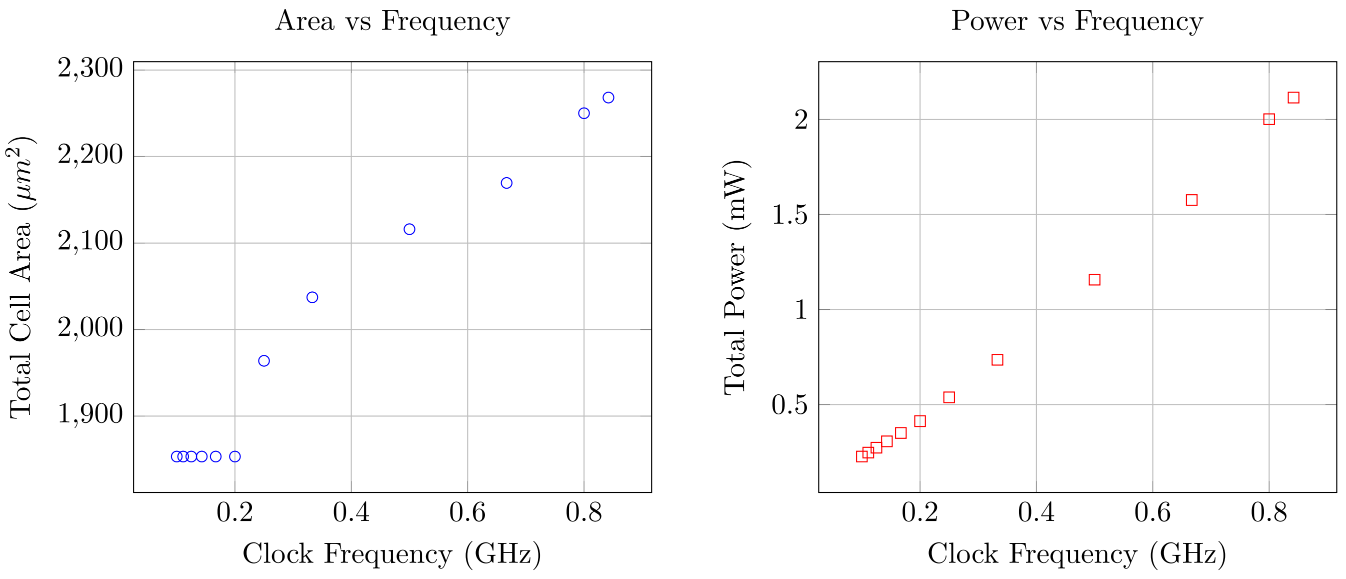


Figure 2: Scatter plots of frequency vs area (left) and frequency vs power (right), only showing slack-met designs.

- **Area vs Frequency:** Area increases slightly with higher clock frequencies due to the need for larger/faster cells to meet timing.
- **Power vs Frequency:** Power rises with frequency since dynamic power scales with f while leakage stays nearly constant.

Frequency (GHz)	Area (μm^2)	Total Power (mW)	Critical Path (Data Arrival, ns)	Slack Status	Slack (ns)
1.000	2298.77	2.5754	input0[4] → piped_product_reg[21] (1.13)	Violated	None
0.889	2328.03	2.3171	input0[13] → piped_product_reg[28] (1.14)	Violated	None
0.842	2268.18	2.1156	input0[8] → piped_product_reg[20] (1.16)	Met	0.0
0.800	2250.09	2.0018	input0[1] → piped_product_reg[24] (1.22)	Met	0.0
0.667	2169.50	2.0053	input0[7] → piped_product_reg[30] (1.46)	Met	1.46
0.500	2116.03	1.1573	piped_product_reg[31] → out_reg[63] (1.96)	Met	0.0
0.333	2037.29	0.7357	input0[2] → piped_product_reg[30] (2.96)	Met	0.0
0.250	1963.88	0.5378	out_reg[0] → out_reg[63] (3.95)	Met	0.01
0.200	1853.22	0.4125	out_reg[0] → out_reg[63] (4.87)	Met	0.1
0.167	1853.22	0.3504	out_reg[0] → out_reg[63] (4.87)	Met	1.1
0.143	1853.22	0.3062	out_reg[0] → out_reg[63] (4.87)	Met	2.1
0.125	1853.22	0.2729	out_reg[0] → out_reg[63] (4.87)	Met	3.1
0.111	1853.22	0.2470	out_reg[0] → out_reg[63] (4.87)	Met	4.1
0.100	1853.22	0.2262	out_reg[0] → out_reg[63] (4.87)	Met	5.1

Table 2: MAC pipelined design (INW=16, OUTW=64). Power is sum of dynamic and leakage. All design points shown, including slack violations.

3. For the pipelined design with the maximum clock frequency you found, how much energy would your system consume if it were to process a sequence of 50 sets of input values? Assume you have to wait until the final output comes out of the system, and don't forget that your pipelined design takes more than 50 cycles to compute 50 sets of inputs.
Remember: energy is measured in joules. Power = energy per time. 1 Watt = 1 Joule / 1 second. Use the power obtained from synthesis and your understanding of the time it would take for your system to fully compute 50 sets of input values.

If we have 50 sets of inputs, then it would take 51 clock cycles to completely process all 50 sets of inputs. This is because our pipelined design is now separated into two stages where two different sets of inputs can be processed at the same time (one per stage) per clock cycle. The maximum clock frequency our pipelined design can handle without violating slack is 0.842 GHz. The total power, which includes the total dynamic power and cell leakage power, used is 2.1156mW. Thus, the total energy consumed for 50 sets of inputs would be

$$(107.8956\,\mathrm{mW}) \times 51 \times (1.1875\,\mathrm{ns}) = 6.5344\,\mathrm{nJ}.$$

4. Would the energy you computed in question 3 change if you resynthesized the design targeting different clock frequencies? Explain and justify your answer. Think carefully about what changes when you change the target frequency and how those changes affect the power the system consumes.

If the target clock frequency changes in synthesis, then the power consumption would alter alongside it. There is a direct relationship between clock frequency and total power consumed for our synthesized design, as seen in Figure 2.

5. Make a table that compares the power, area, latency, and throughput of your pipelined and unpipelined MAC designs (at the maximum clock frequency you previously found) with INW=16 and OUTW=64. In your report show how you calculated the latency and throughput. Quantify latency in seconds (or ns), and quantify throughput in terms of MACs per second. (If needed, review these concepts in the Topic 6 slides.) Based on the trade-offs seen in your table, explain when it would make sense for a designer to choose the pipelined design and when it would make sense to use the unpipelined design.

Table 3: Comparison of Unpipelined and Pipelined MAC Designs (INW=16, OUTW=64)

Design	**Max Freq** (GHz)	**Power** (mW)	**Area** (μm^2)	**Latency** (ns)	**Throughput** ($MACs/sec$)
Unpipelined	0.727	1.5420	2033.57	1.375	7.27×10^8
Pipelined	0.842	2.1156	2268.18	2.375	1.684×10^9

Note:

- Latency is the minimum clock period required for one stage — in this case, it's the time delay from the multiply register to the accumulate register.
- Latency (ns) = 1.375 × 1 stage (for unpipelined design)
- Latency (ns) = 1.1875 × 2 stages = 2.375 (for pipelined design)
- Throughput (MACs/s) = 1 MAC per cycle × 0.727 GHz = 727 million MACs/s
- Throughput (MACs/s) = 1 proper MAC per cycle *across two stages assuming steady state* × 0.842 GHz = 842 million MACs/s
- We considered MACs to be operations performed per stage; so either a multiply operation or an accumulate operation can count as a MAC.

Based on the results of the table, a designer would choose an unpipelined design if he/she wish for lower latency, and choose a pipelined design if he/she wish for higher throughput. The tradeoff is between the latency (to get a multiply-added output from a set of inputs) and the throughput (in operations and inputs per second).

6. Your design is pipelined as much as possible if you assume that you cannot pipeline the arithmetic units themselves. However, as we discussed in Topic 6, we could also pipeline the multiplier itself. For example, you can replace the multiplier with one that is pipelined into more stages. Based on your results to questions 1 and 2, would you expect that deeper pipelining in the multiplier might help you reach higher clock frequencies? Justify why or why not.
If you were to pipeline the multiplier in this way, what other changes would you have to make in your module?
Would pipelining the adder be possible and a good idea? Why or why not?
(Answer this question based on your understanding of the design and your answers to prior questions. You do not need to modify your design/code to answer this question.)

The multiplier of the design can be pipelined by segmenting our set of inputs (input0 and input1) into smaller streams that could be stored in registers and multiplied concurrently. For example, if INW = 16, then the two inputs (input0 and input1 [15:0]) can be separated into eight groups of two-bit [1:0] values that are multiplied simultaneously. Doing so may result in a possible higher target clock frequency for our system because smaller bit-length value multiplication operations can be performed per piped stage, thus less combinational logic may be required for data to travel from one register to another. However, we are still limited by the critical path from the accumulator register's output back to its adder.

Pipelining the adder is also possible and beneficial in this design. Since adders typically involve carry propagation across multiple bits in the combinational path, in our case, across the OUTW bit width, it is a good idea to partition the accumulator adder's inputs into smaller bit segments. By inserting pipeline registers between these segments, the addition can be performed over multiple stages. Similar to pipelining the multiplier, it allows simultaneous processing of smaller sum segments and effectively reduces the critical path delay, improving the overall clock frequency.

7. In questions 1 and 2, you always synthesized using the same parameter values of INW=16 and OUTW=64. Here, explore how changing INW and OUTW affects the critical path location and

maximum clock frequency of your pipelined MAC module. Don't forget: to change these parameters for synthesis, you should edit the default parameter values in your source code (mac_pipe.sv).
First, do a set of experiments where you set INW=12 and synthesize three designs with OUTW=24, 48, and 64.
Next, do a new set of experiments where you set OUTW=48 and synthesize three designs with INW=8, 16, 24.
Do the clock frequency and critical path location change as OUTW changes? Do they change when INW changes? Explain what you see and what you learn from it. You should expect to see differences in the scaling behavior of INW and OUTW. Explain why they behave differently.
Include the six synthesis reports for this question along with your Final Report.

Table 4: Synthesis results for different (INW, OUTW) configurations (pipelined MAC)

Parameter Config.	Minimum Clock Period (ns)	Crit. Path (Startpoint → Endpoint)
INW=12, OUTW=24	1.0555	`input0[3]` → `piped_product_reg[19]`
INW=12, OUTW=48	1.0670	`input0[11]` → `piped_product_reg[23]`
INW=12, OUTW=64	1.0535	`input0[5]` → `piped_product_reg[19]`
INW=8, OUTW=48	0.8430	`input0[3]` → `piped_product_reg[10]`
INW=16, OUTW=48	1.2800	`input0[6]` → `piped_product_reg[30]`
INW=24, OUTW=48	1.4850	`input0[21]` → `piped_product_reg[34]`

Observations and explanation:

- The maximum frequency is related to the bit width of the input parameter `INW`, but not so much to the bit width of the output parameter `OUTW`. We can observe that the minimum clock period per synthesized design increases as the bit width of the input `INW` increases. There were minimal changes in the minimum clock period when `INW` stays constant while `OUTW` increases.
- At a higher level, the location of the critical path does not change significantly. It starts from either set of the inputs and ends at the piped product register, `piped_product_reg`. However, when we zoom in, the critical path location changes slightly. The specific order of the bit element of each bit array changes for both the input and `piped_product_reg`. This is because, on the lower-level abstraction, each bit in the input and output arrays maps to different gate-level networks, which may result in slight variations in propagation delay.

8. The MAC's accumulator holds OUTW bits. As you know, if the value stored in the accumulator grows large enough, it will overflow. That is, OUTW bits may not be enough to store the resulting number.

Assume INW = 5 and OUTW = 16. What is the maximum number of MACs your system could perform while guaranteeing that the accumulator cannot overflow? (Hint: what is the largest magnitude number you could produce on the multiplier's output? Then, how many accumulations would it take for that number to produce an overflow in the accumulator?) Don't forget that our values are all signed integers, and don't forget that the accumulator can be initialized to a signed INW-bit number. Show your reasoning and justify your answer.

Given:

$$\text{INW} = 5 \quad \Rightarrow \quad \text{each input} \in [-2^4,\, 2^4 - 1] = [-16,\, 15],$$

So the largest magnitude possible from a single multiply (two signed INW-bit values) is

$$\max|\text{product}| = 16 \times 16 = 256.$$

The accumulator is OUTW = 16 bits, so its signed range is

$$\text{accumulator} \in [-2^{15},\, 2^{15}-1] = [-32768,\, 32767].$$

To guarantee no overflow when repeatedly accumulating the worst-case magnitude product (all products have the same sign and magnitude 256), we must consider the worst initialization of the accumulator (the initialization that gives the smallest headroom in the direction of accumulation). For positive-direction accumulation the worst initial accumulator value is +15 (the largest INW signed initialization). Thus require

$$-16 + 256 \cdot N \leq -32768,$$

where N is the number of MAC accumulations. Solving:

$$256 \cdot N \leq -32768 + 16 = -32752,$$

$$N \leq \left\lfloor \frac{32752}{256} \right\rfloor = \lfloor 127.96875 \rfloor = 127.$$

(Checking the negative direction gives the same integer bound: starting at -16 and repeatedly adding -256 would underflow at the same count $N = 127$.)

Answer: The maximum number of MAC accumulations that can be performed while *guaranteeing* the 16-bit accumulator will not overflow (for INW = 5, OUTW = 16) is

$$\boxed{N_{\max} = 127.}$$

9. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

Huabin: Wrote the unpiped and piped MAC modules, simulated and synthesized both designs with different input and output bit width parameters. Answered conceptual questions and performed required calculations for the lab report.

Ryan: Wrote a python script program to parse the data of interest (area, total power consumption, critical path, slack time) from the synthesized file to speed up the process of collecting data points for tables and graphs. Assisted in formatting the lab report.

Project Part 2: Output FIFO

1. The basic form of the FIFO was discussed in class, but here you needed to adapt that to interface its output with AXI-Stream. Explain how you did that and how your logic works.

When it comes to designing a FIFO AXI-stream system, we need to think and view it from the perspective of being in between the input interface of one module and the output interface of another module. In this part of the project, we made several adjustments to accommodate for this. Firstly, we have control signals such as TVALID and TREADY from both interfaces telling us when a module is ready to transmit and or receive data. When the FIFO is not empty, that means it can have valid data coming out of it (OUT_TVALID). When the FIFO is not full, that means it is ready to receive data (IN_TREADY). Notice that both these control signals are actually two different modules interfacing with each other, hence creating an extra layer of security, namely the AXI-stream handshake, that respects the FIFO flow.

2. Previously you synthesized this design with OUTW=24 and DEPTH=19. In your report, give the maximum clock frequency and the area, power, and critical path location for this frequency. Note that the critical path location may be somewhat confusing. Make sure you carefully trace it so you can thoroughly explain what logic the critical path includes. (Don't just list the critical path's start and end points; explain what the logic is doing between them.)

Table 5: Synthesis results of different (OUTW, DEPTH) configurations for Output FIFO

Parameter Config.	Area (μm^2)	Power (mW)	Max. Freq. (GHz)	Crit. Path (Startpoint → Endpoint)
OUTW=12, DEPTH=19	1907.49	1.3436	1.136	`read_ptr_reg[1]` → `fifo_mem/data_out_reg[1]`
OUTW=24, DEPTH=19	3711.76	2.3152	1.053	`capacity_reg[1]` → `fifo_mem/data_out_reg[3]`
OUTW=24, DEPTH=38	6798.96	5.1237	1.075	`read_ptr_reg[1]` → `fifo_mem/data_out_reg[9]`

3. Repeat question 2 with both:

- **OUTW=12 and DEPTH=19**
- **OUTW=24 and DEPTH=38**

Report all statistics for both designs. Explain the trends you see in area, power, and frequency. In other words, explain how these three sets of parameters affect the area, power, and frequency, and explain why.

We see a clear trend in higher area and power consumptions as the absolute values of the OUTW and DEPTH parameters increase. More importantly, the OUTW parameter has a significant impact. This is intuitive because wider memory words will result in more congested routing and larger multiplexing logic to select from wider bit-length word data. This obviously requires more power and spacing to be processed.

However, we notice an odd trend observing the maximum frequency. Higher DEPTH actually resulted in a slightly short minimum clock period yield given that the OUTW parameter remains unchanged.

Critical Path Analysis:

- The critical path from read_ptr register to data_out register likely goes into address decoders and multiplexers selecting from memory array blocks and through data_out registers. This makes sense as we can see from the SystemVerilog code statement “data_out $\leftarrow$ mem[read_addr];”.

- The critical path from capacity register to data_out register likely stems from capacity comparison logic, and it likely goes into OUT_AXIS_TVALID control signal that routes to read enable signal, and out the data_out register.
- The changes in critical path and the slight decrease in maximum frequency is likely due to the way our synthesis tool handles the memory architecture used to hold word data. From (12,19) $\rightarrow$ (24,19), the critical path changes because our OUT_AXIS_TVALID signal after capacity comparison contains higher fanouts. From (24,19) $\rightarrow$ (24,38), pre-existing optimized memory tools might handle the delays better than the design created from our code.

4. If you worked with a partner, please carefully describe each partner’s contribution to this part of the project. (If you did not work with a partner, skip this.)

Huabin: Fixed counter overflow and resetting logic.

Ryan: Got the look-ahead logic for memory reading logic to work.

Project Part 3: Input Memory Module

1. This part of the project required you to design a significant amount of control logic that interacts with the AXI-Stream interface, the memories, the K and B registers, and the inputs_loaded and compute_finished signals. Carefully and thoroughly document this module including your control logic. Your documentation should allow the reader to fully understand how your input_mems module works (and any submodules) without looking at the code.

The input_mems module is a memory block that holds key important components necessary for computing a convolution operation. The module include registers that contain the following information:

- A weight 'W' matrix for which its dimensions are defined by KxK, where the value of K is read and stored in a MAXK-bit register during the first cycle of every weight matrix read.
- An optional bias 'B' value that is stored in an INW-bit register on the cycle following the completion of the weight matrix read.
- An input 'X' matrix for which the kernel matrix is performing the convolution operation on.

In our system, the control flow of our input_mems module is implemented via a finite state machine with five states: idle_state, input_W_matrix_state, input_B_value_state, input_X_matrix_state, and inputs_loaded_state.

We begin in the idle_state where we wait for any valid inputs to be picked up via AXI-stream input interface. At this point, we look for the control bit "new_W" sent during the same cycle to be "set". If it is, then we know to transition from the idle_state to the input_W_matrix_state and that there is a new input weight matrix to consider. However, if "new_W" was not "set", then we transition to the input_X_matrix_state, and keep the weight matrix and bias value from the most recent loaded inputs to compute the next convolution against. However, we must have a weight matrix inside our memory register, and a bias value inside our 'B' register before any input 'X' matrix can be read afterwards.

There are two different complete flows:

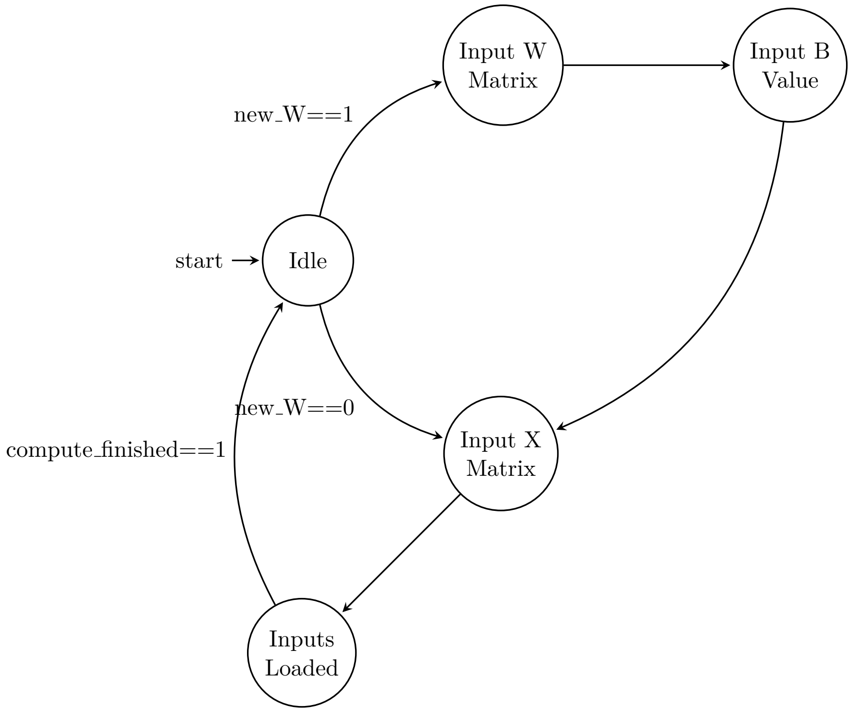


Figure 3: Finite State Machine flow for Inputs Memory Module

It is important to know that the input_mems module can be both written to and read from depending on what state logic we are at. At the inputs_loaded_state, the writing freezes by "setting" the "inputs_loaded" control signal and we let the AXI-stream's output interface handle all of the necessary reading and computation before "setting" the "compute_finished" control signal, telling us to go back to idle_state and watch for the "new_W" control bit.

2. The number of cycles required by this module is largely determined by:

- **the parameters (R, C)**
- **the value of K for this input**
- **and how the testbench asserts AXIS TVALID**

However, there are places where you as the designer could make choices that affect the number of cycles required by your module. For example, if your system unnecessarily sets AXIS TREADY to 0, or it adds extra cycles of delay between steps, the system will be less efficient.

One way to quantify this is to measure how long your system takes to complete a task. Run a simulation where you set INW=10, R=15, C=13, MAXK=7 and INPUT VALID PROB=1. When new_W == 1, the testbench will begin by feeding in the $K * K$ values of w, then the value of B, then the $R * C = 15 * 13 = 195$ values of x. In other tests where new_W == 0, the system will simply feed in the $R * C = 15 * 13 = 195$ values of x.

Simulate this design in Questa Sim's waveform view and count the number of cycles between when your design sets AXIS TREADY to 1, and when it sets inputs_loaded to 1. Do this for a few sets of inputs where new_W == 1 and a few sets of inputs where new_W == 0, and record the number of cycles and the value of K.

Hint: you can view the amount of simulated time that passes in the waveform and then divide it by 10ns to get the number of simulated clock cycles.

In your report, for each test give this cycle count and the value of K (for both new w of 0 and 1). In the report, quantify how efficient your system is with respect to the number of clock cycles by computing the following ratios:

- **When new_W == 1, use efficiency = $(K * K + 1 + R * C)$/cycles**
- **When new_W == 0, use efficiency = $R * C$/cycles**

In these metrics, 1.0 is perfectly efficient a good implementation will be close to this. Report both metrics and the data you collected. If your efficiency number is not close to 1, where could your logic be improved?

Table 6: Simulation cycle counts and efficiency for input_mems module

new_W	K	# of cycles	Efficiency
1	3	205	1
1	7	245	1
1	4	212	1
0	X	195	1
0	X	195	1
0	X	195	1

3. In the previous question, you measured the cycle count. Now, use your understanding of your system's behavior to write equations for the cycle count with respect to R, C, and K. You should have one equation for new W == 1, and one equation for new W == 0.

Since the cycle efficiency of our system is 1, the cycle count equations are defined as:

$$\# \text{ cycles} = \begin{cases} K^2 + R \cdot C + 1 & \text{if } \texttt{new\_W} = 1 \\ R \cdot C & \text{if } \texttt{new\_W} = 0 \end{cases}$$

4. For the Part 3 submission (above), you synthesized the design with parameters:

- **INW = 24, R = 9, C = 8, MAXK = 4**
- **INW = 10, R = 15, C = 13, MAXK = 7**

For each set of parameters, report the clock frequency, area, power, and critical path location. Carefully explain each design's critical path. Don't just list its start and end points; explain what logic is included in the path and what it means. Does the critical path location change between these two designs? Explain why or why not.

Table 7: Synthesis results for input_mems module

Parameter Config.	Area (μm^2)	Power (mW)	Max Freq. (GHz)	Crit. Path (Startpoint → Endpoint)
INW=24, R=9, C=8, MAXK=4	16581.38	16.57	1.342	`current_state_reg[2]` → `X_memory_inst/data_out_reg[13]`
INW=10, R=15, C=13, MAXK=7	18666.82	19.37	1.198	`current_state_reg[2]` → `X_memory_inst/data_out_reg[4]`

Critical Path Analysis:

- The critical path `current_state_reg[2]` → `X_memory_inst/data_out_reg[13]` suggests that the path likely runs from trying to retrieve the current state of the system, likely `inputs_loaded_state`, and going through combinational logic to get the address to access the indexed value of the flattened instantiated X memory holding the X matrix values.
- The critical path `current_state_reg[2]` → `X_memory_inst/data_out_reg[4]` suggests that the path likely runs from very similarly. Since the values of the each index in the X memory is now larger, the synthesis tools determines which indexed value took the longest.

5. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

Huabin: Contributed in debugging the waveforms, provided the framework for the overall FSM flow control logic for the module, and the necessary control logic for state transitions.

Ryan: Contributed in debugging the waveforms and analyzing the testbench for a better understanding of how the control logic should update.

Project Part 4: Top-Level 2D Convolution System

1. This part of the project required you to design a significant amount of control logic that interacts with the existing modules. Carefully and thoroughly document how your top-level system works, especially your control logic. Your documentation should allow the reader to fully understand how it works without looking at the code.

The top level convolution module that connects all of the other modules, that is, `mac_pipe` module, `fifo_out` module, and `input_mems` module, utilizes a significant amount of control signal logic together with a finite state machine to handle the control flow of the overall system. The finite state machine has six states:

- `IDLE_STATE`
- `INIT_PIXEL_STATE`
- `COMPUTING_ACC_STATE`
- `WRITE_PIXEL_STATE`
- `WRITE_FIRST_ADDR`
- `DONE_STATE`

The most important control signals are:

- `COMPUTE_FINISHED_DATA_IN`
- `DATA_AVAILABLE`
- `MAC_OUTPUT_VALID`
- `INITIALIZE_ACCUMULATOR`

The complete flows are as follows:

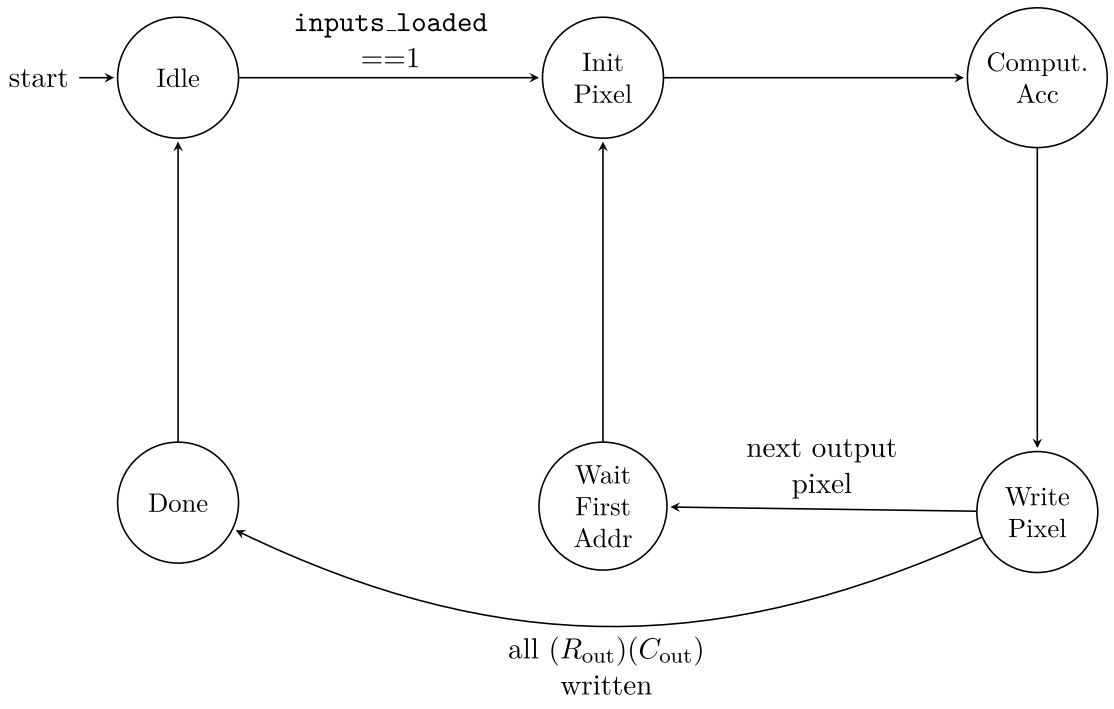


Figure 4: Finite State Machine flow for Top-Level 2D Convolution System

In the IDLE_STATE, the system will wait for valid data to flow in via the AXI-stream input interface. All control signals are "off" in this state. The system transitions out of the IDLE_STATE into the INIT_PIXEL_STATE when the kernel weight matrix and the input 'X' matrix along with the kernel size and the bias value are all properly loaded into the input_mems registers.

During the INIT_PIXEL_STATE, we initialize a singular pixel of the resulting 'Y' matrix (R_out, C_out) with the bias value for one cycle. This bias value is then added to the result of the matrix multiplication, specific to the pixel we are computing (R, C). In this state, we also keep track of the position of the input matrix that we are performing the matrix multiplication on, and reset the pointers of the kernel matrix position back to the beginning $(i = 0, j = 0)$.

Following the INIT_PIXEL_STATE is the COMPUTING_ACC_STATE, which is where the matrix multiplication actually happens for ideally, K^2 number of cycles handled by the mac_pipe module.

Once the COMPUTING_ACC_STATE finishes, it will transition to the WRITE_PIXEL_STATE where the control signal "MAC_OUTPUT_VALID" is "set" and the corresponding output 'Y' pixel gets written to the buffer in the fifo_out module. In this state, we increment our cursor to the next starting pixel in the output 'X' matrix and continue the matrix multiplication for the next output 'Y' pixel.

When we finish writing the entire output 'Y' matrix to the fifo_out module, the system transitions to the DONE_STATE where the control signal "COMPUTE_FINISHED_DATA_IN" is set and the system gets ready to load in the next input 'X' matrix or it can load in a new set of matrices (including both 'X' and weight matrix) along with a new bias value while "idling" in the IDLE_STATE.

2. In Part 3, you wrote equations that described the number of cycles that the input_mems module requires (given R, C, and K) when new_W == 0 and new_W == 1.

Now, you should use your understanding of your system to write equations to describe the number of cycles for the entire convolution operation in the same scenarios. Your equations should reflect the best-case number of cycles between when your system starts receiving inputs and when it is done with that computation and ready to receive a new set of inputs (where "best case" implies that INPUT_TVALID and OUTPUT_TREADY are always 1). Don't forget that you can do simulations to verify your equations.

Based on your equations, is your system's performance limited by any one phase of execution? For example, if your input loading time is much higher than the compute time, this shows that the input loading time limits your speed much more than computation. On the other hand, if the system spends most of its time doing MAC operations on data, then the computational unit is the limiting factor. If the times are close to balanced, then this shows your system's performance is highly dependent on both of them. Importantly, keep in mind that this answer can change based on the values of R, C, and K. In other words, you may find you are limited by one factor when they are small, and another when they are large. Think carefully and explain fully.

Justify and explain your answers.

For parameters $R = 9$, $C = 8$, $K = 4$:

Case 1: `new_W` = 1

$$\begin{aligned}
\text{Cycles} = {} & \underbrace{(RC + 1 + K^2 + 1)}_{\text{Initial Load}} \\
& + \underbrace{[((R-K+1)(C-K+1)) - 1]}_{\text{Pixels } 1\ldots(N-1)} \times \underbrace{(1 + (K^2+1) + 1 + 1)}_{\text{Latency per Pixel}} \\
& + \quad \underbrace{(1 + (K^2+1) + 1 + 1)}_{\text{Last Pixel (set compute finished)}} \\
= {} & \mathbf{679} \text{ cycles}
\end{aligned}$$

Case 2: `new_W` = 0

$$\begin{aligned}
\text{Cycles} = {} & \underbrace{(RC + 1)}_{\text{Initial Load}} \\
& + \underbrace{[((R-K+1)(C-K+1)) - 1]}_{\text{Pixels } 1\ldots(N-1)} \times \underbrace{(1 + (K^2+1) + 1 + 1)}_{\text{Latency per Pixel}} \\
& + \underbrace{(1 + (K^2+1) + 1 + 1)}_{\text{Last Pixel}} \\
= {} & \mathbf{662} \text{ cycles}
\end{aligned}$$

The system spends the majority of the time loading inputs in the input_mems module during the `IDLE_STATE` and the multiply-accumulate operations state. This is dictated by how large the parameters R, C, and K are. If $R * C \gg K * K$ then the system spends most of its time computing the MAC operation. On the other hand, if $R * C \approx K * K$, then the system spends a more balanced amount of time between doing the input loading and the MAC operation. But the combined time to compute the entire output matrix still takes up most of the time. Of course, this also depends on if new_W is "set" or not. If new_W is not "set", then loading the inputs during the IDLE_STATE will take "$K * K + 1$" less number of cycles as compared to if it was set because we aren't loading in any new kernel weight matrix and bias values, but just solely loading a new input 'X' matrix instead.

3. For the Part 4 submission, you synthesized the design with three sets of parameters:

- **INW = 12, R = 9, C = 8, MAXK = 5**
- **INW = 18, R = 9, C = 8, MAXK = 5**
- **INW = 24, R = 16, C = 17, MAXK = 9**

For each set of parameters, report the maximum clock frequency, minimum clock period, area, and power. For each, describe where the critical path is in the design. (Make sure you explain the critical path fully; don't just list the start and end points.) If the different designs have meaningfully different critical path locations, explain or speculate as to why the location changes. You only need to report data for the smallest clock period you were able to find for each design.

Table 8: Synthesis results for 2D Convolution Top-Level System

Parameter Config.	Area (μm^2)	Power (mW)	Max Freq. (GHz)	Crit. Path (Startpoint → Endpoint)
INW=12, R=9, C=8, MAXK=5	21703.74	17.78	0.90497	`input_mems_instantiation/ X_memory_inst/data_out_reg[5]` → `mac_pipe_instantiation/ piped_product_reg[22]`
INW=18, R=9, C=8, MAXK=5	31465.41	20.58	0.72463	`input_mems_instantiation/ X_memory_inst/data_out_reg[5]` → `mac_pipe_instantiation/ piped_product_reg[34]`
INW=24, R=16, C=17, MAXK=9	164809.08	97.02	0.63291	`input_mems_instantiation/ X_memory_inst/data_out_reg[20]` → `mac_pipe_instantiation/ piped_product_reg[41]`

Critical Path Analysis:

- The critical path `input_mems_instantiation/X_memory_inst/data_out_reg[5]` → `mac_pipe_instantiation /piped_product_reg[22]` suggest that the data read from the input_mems module holding the values of the input 'X' matrix is being fed as one of the inputs for combinational multiplication logic in the mac_pipe module and getting stored in the `piped_product_reg`.
- The critical path `input_mems_instantiation/X_memory_inst/data_out_reg[5]` → `mac_pipe_instantiation /piped_product_reg[34]` suggest that the situation is similar to the first critical path described above. The difference in index value of the registers depends on what the synthesis tool algorithms calculate.
- The critical path `input_mems_instantiation/X_memory_inst/data_out_reg[20]` → `mac_pipe_instantiation /piped_product_reg[41]` suggest that similar to the critical paths of the previous two design parameters described above has occured. It is up to the synthesis tool to determine which index of the register actually has the critical path.

4. Now, find the throughput of the second and third of the three designs you considered in question 3:

- **INW = 18, R = 9, C = 8, MAXK = 5**
- **INW = 24, R = 16, C = 17, MAXK = 9**

(All of the following questions will refer to these two designs.)

Find the throughput of each of the two designs under three different assumptions about testbench parameters INPUT_TVALID_PROB and OUTPUT_TREADY_PROB: 0.1, 0.5, and 1. (That is, do three simulations for each design, one where both _PROB parameters are 0.1, one where they are both 0.5, and one where they are both 1.)

For each, record the number of clock cycles needed for 10,000 convolutions, as reported by the testbench with the parameters set appropriately. Then use those cycle counts along with the clock periods you found from synthesis to find the throughput in number of convolutions per second for each design under the three assumptions of the_PROB parameters.

In your report, make a table that shows the cycle counts and computed throughputs for all 6

scenarios (two designs with three sets of assumptions each). Make sure your tables include units (here and in all questions).

Table 9: Throughput results for 2D Convolution Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)		**Design 2** (INW=24, R=16, C=17, MAXK=9)	
TVALID/TREADY prob.	**Cycle Count**	**Throughput** (ops/sec)	**Cycle Count**	**Throughput** (ops/sec)
0.1	13400388	540758	74805492	84607
0.5	7017694	1032586	51462470	122985
1.0	6225557	1163972	48560650	130334

5. The average delay of a system is the average amount of time that elapses between when the system starts a computation and when it finishes it. For the 6 scenarios evaluated in question 4, determine the delay in seconds (or ms, us, ns, etc., as appropriate). Use the cycle counts you determined in the previous question and the clock period you determined in question 3. (Don't forget that the cycle counts reported in question 4 are for 10,000 convolutions-you need to find the average delay for a single convolution). Report these delays in a table.

Table 10: System delay results for 2D Convolution Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)	**Design 2 (INW=24, R=16, C=17, MAXK=9)**
TVALID/TREADY prob.	**System Delay** (μs)	**System Delay** (μs)
0.1	1.3400388	7.4805492
0.5	0.7017694	5.146247
1.0	0.6225557	4.856065

6. The synthesis tool gives you an estimate of the power of your system. Use the power obtained from synthesis and the delays you computed in question 5 to determine the average energy your system consumes per convolution for each of the six scenarios you have evaluated in the previous questions. Report these values in a table.

Remember: energy is measured in joules. Power = energy per time. 1 Watt = 1 Joule / 1 second.

Table 11: Average energy consumption per convolution results for 2D Convolution Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)	**Design 2 (INW=24, R=16, C=17, MAXK=9)**
TVALID/TREADY prob.	**Avg. Energy per Convolution** (J)	**Avg. Energy per Convolution** (J)
0.1	$(19.92mW + 0.66\mu W) * 1.34\mu s = 0.0267$	$(94.20mW + 2.81mW) * 7.48\mu s = 0.7257$
0.5	$(19.92mW + 0.66\mu W) * 0.70\mu s = 0.0140$	$(94.20mW + 2.81mW) * 5.15\mu s = 0.4993$
1.0	$(19.92mW + 0.66\mu W) * 0.62\mu s = 0.0124$	$(94.20mW + 2.81mW) * 4.86\mu s = 0.4711$

7. A joint metric that combines the effects of area and speed in a single value is the area-delay product. The area-delay product is found by multiplying the area of the system times its delay. (Since these are both metrics that we want to minimize, lower area-delay products are better than

higher ones.) Calculate the area-delay product of your system under these six scenarios and report the results in a table. Don't forget to include units in your answer.

Table 12: Area-delay product results for 2D Convolution Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)	**Design 2 (INW=24, R=16, C=17, MAXK=9)**
TVALID/TREADY prob.	**Area-Delay Product** ($\mu m^2 \cdot \mu s$)	**Area-Delay Product** ($\mu m^2 \cdot \mu s$)
0.1	$31465.41\mu m^2 * 1.34\mu s = 42164.86$	$164809.08\mu m^2 * 7.48\mu s = 1232862.40$
0.5	$31465.41\mu m^2 * 0.70\mu s = 22081.46$	$164809.08\mu m^2 * 5.15\mu s = 848148.21$
1.0	$31465.41\mu m^2 * 0.62\mu s = 19588.97$	$164809.08\mu m^2 * 4.86\mu s = 800323.59$

8. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

Huabin: Constructed the working structure for the FSM control flow logic of the top-level module and fixed bugs in previous modules to work compatibly with the top-level module.

Ryan: Assisted in debugging + did the mathematical calculations for statistics of different designs.

Project Part 5: Top-Level 2D Convolution System Performance Optimization

1. What techniques did you perform to improve the performance of your system? Explain and document your approach in detail and carefully describe how your optimized system works. Explain why you chose these techniques. Do you believe they were effective? (This question is important, so make sure you answer it fully.)

The system was optimized significantly through two different methods:

a. By using DesignWare's n-staged pipelined multipliers, the synthesis tool was allowed to optimize the `mac_pipe` module through pre-existing libraries and chose the best designs for throughput. This change was also accompanied by registering the `mac_pipe` module's `input0` and `input1` in our top-level instead of directly connecting the `INPUT_MEMS_W_DATA_OUT` and `INPUT_MEMS_X_DATA_OUT` to it. We found that a higher magnitude of pipeline stages did not necessarily reduce the minimum clock period, but rather worsened it. So a two-stage pipeline was the best for our design. All of this was effective in reducing the minimum clock period of each design by a fair amount, thus higher frequency. More details in the tables.

b. By adding parallelism to the top-level module. Instead of reading one address at a time for multiply-accumulate operations, modifications were added to have the system read two addresses at a time. This can be thought of as having two kernels sliding across the input X matrix at once. So to do that, another `input_mems` module and `mac_pipe` module were instantiated as `input_mems_instantiation2` and `mac_pipe_instantiation2` respectively. This was effective in reducing the total number of cycles required to complete a full convolution operation, and ideally, by a factor of two.

2. In Part 3, you wrote equations that described the number of cycles that the input_mems module requires (given R, C, and K) when new_W == 0 and new_W == 1.

Now, you should use your understanding of your system to write equations to describe the number of cycles for the entire convolution operation in the same scenarios. Your equations should reflect the best-case number of cycles between when your system starts receiving inputs and when it is done with that computation and ready to receive a new set of inputs (where "best case" implies that INPUT_TVALID and OUTPUT_TREADY are always 1). Don't forget that you can do simulations to verify your equations.

Based on your equations, is your system's performance limited by any one phase of execution? For example, if your input loading time is much higher than the compute time, this shows that the input loading time limits your speed much more than computation. On the other hand, if the system spends most of its time doing MAC operations on data, then the computational unit is the limiting factor. If the times are close to balanced, then this shows your system's performance is highly dependent on both of them. Importantly, keep in mind that this answer can change based on the values of R, C, and K. In other words, you may find you are limited by one factor when they are small, and another when they are large. Think carefully and explain fully.

Justify and explain your answers.

For parameters $R = 9$, $C = 8$, $K = 4$, *$C\_out = 5$*:

Case 1: `new_W` = 1 and if number of strides across the columns, or `C_out` is odd

$$\begin{aligned}\text{Cycles} = &\underbrace{(RC + 1 + K^2 + 1)}_{\text{Initial Load}} \\ &+ \underbrace{[((R\_out)(C\_out + 1)/2) - 1]}_{\text{Pixels } 1\ldots(N-1)} \times \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Latency per Pixel}} \\ &+ \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Last Pixel (set compute finished)}} \\ = &\ \mathbf{486} \text{ cycles}\end{aligned}$$

Case 2: `new_W` = 0 and if number of strides across the columns, or `C_out` is odd

$$\begin{aligned}\text{Cycles} = &\underbrace{(RC + 1)}_{\text{Initial Load}} \\ &+ \underbrace{[((R\_out)(C\_out + 1)/2) - 1]}_{\text{Pixels } 1\ldots(N-1)} \times \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Latency per Pixel}} \\ &+ \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Last Pixel (set compute finished)}} \\ = &\ \mathbf{469} \text{ cycles}\end{aligned}$$

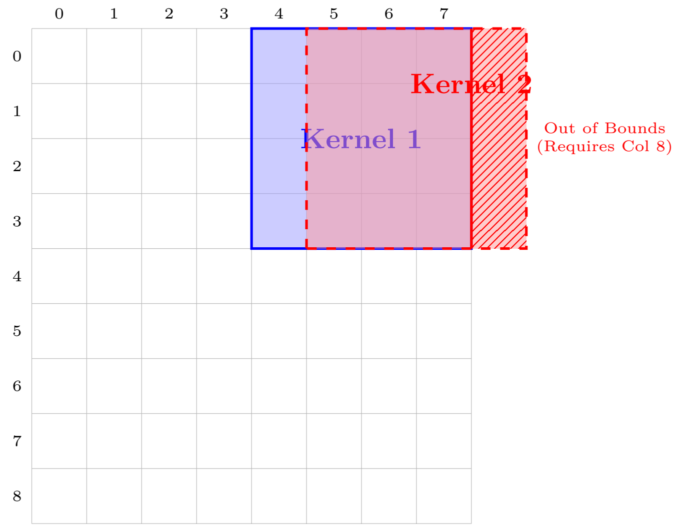
For parameters $R = 9$, $C = 8$, $K = 5$, $*C\_out = 4*$:

Case 3: `new_W` = 1 and if number of strides across the columns, or `C_out` is even

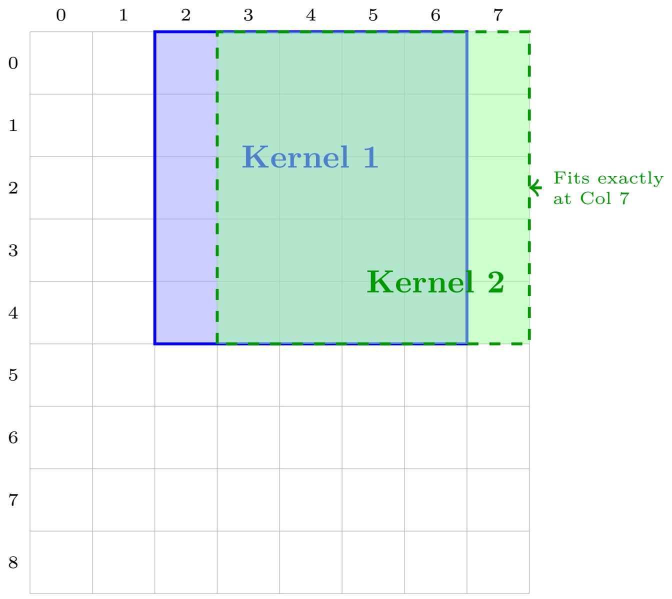
$$\begin{aligned}\text{Cycles} = &\underbrace{(RC + 1 + K^2 + 1)}_{\text{Initial Load}} \\ &+ \underbrace{[((R\_out)(C\_out)/2) - 1]}_{\text{Pixels } 1\ldots(N-1)} \times \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Latency per Pixel}} \\ &+ \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Last Pixel (set compute finished)}} \\ = &\ \mathbf{409} \text{ cycles}\end{aligned}$$

Case 4: `new_W` = 0 and if number of strides across the columns, or `C_out` is even

$$\begin{aligned}\text{Cycles} = &\underbrace{(RC + 1)}_{\text{Initial Load}} \\ &+ \underbrace{[((R\_out)(C\_out)/2) - 1]}_{\text{Pixels } 1\ldots(N-1)} \times \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Latency per Pixel}} \\ &+ \underbrace{(1 + 1 + (K^2 + 1) + 2 + 1)}_{\text{Last Pixel (set compute finished)}} \\ = &\ \mathbf{383} \text{ cycles}\end{aligned}$$



(a) Visualizing Parallel Stride 3 ($K = 4$): Kernel 1 processes valid cols 4–7. Kernel 2 attempts cols 5–8 (OOB). The extra MAC is computed but discarded, and the valid calculation repeats on the next stride by Kernel 1.



(b) Visualizing Parallel Stride 2 ($K = 5$): Kernel 1 processes cols 2–6. Kernel 2 processes cols 3–7. Both kernels fit perfectly within the 8-column matrix.

Figure 5: Comparison of boundary conditions between different Kernel sizes and Strides.

Similar to the design in Part 4, the system spends the majority of the time loading inputs in the input_mems module during the `IDLE_STATE` and computing the output matrix during the multiply-accumulate operations state. This is dictated by how large the parameters R, C, and K are. Since we introduced parallelism with two multiply-accumulate operations per cycle, the system spends less time computing the output matrix compared to the design in Part 4. However, computing the entire output matrix still takes the longest. If $R * C \gg K * K$ then the system spends most of its time computing the MAC operation. On the other hand, if $R * C \approx K * K$, then the system spends a more balanced amount of time between doing the input loading and the MAC operation. From the equations, we can conclude that the parallelism implementation works more effectively if the number of strides across the columns of the input 'X' matrix is an even number (if `C_OUT` is even). Of course, this also depends on if `new_W` is "set" or not. If `new_W` is not "set", then loading the inputs during the IDLE_STATE will take "$K * K + 1$" less number of cycles as compared to if it was set because we aren't loading in any new kernel weight matrix and bias values, but just solely loading a new input 'X' matrix instead.

Table 13: Synthesis results for Optimized Top-Level System

Parameter Config.	Area (μm^2)	Power (mW)	Max Freq. (GHz)	Crit. Path (Startpoint → Endpoint)
INW=12, R=9, C=8, MAXK=5	32537.38	29.47	1.064	`kernel_counter_for_acc_reg[2]` → `internal_x_read_address2_reg[6]`
INW=18, R=9, C=8, MAXK=5	48455.35	36.46	0.917	`input_mems_instantiation/current_state_reg[1]` → `input_mems_instantiation/X_memory_inst/mem_reg[46][0]`
INW=24, R=16, C=17, MAXK=9	226974.33	182.61	0.855	`fifo_out_instantiation/read_ptr_reg[0]` → `fifo_out_instantiation/fifo_mem/data_out_reg[26]`

Critical Path Analysis:

- The critical path `kernel_counter_for_acc_reg[2]` $\rightarrow$ `internal_x_read_address2_reg[6]` suggest that the register responsible for holding the counter value for K^2 number of multiply-accumulate operations on the input 'X' matrix, goes through logic that selects the read address for the input of the second of the parallel mac_pipe unit is the longest.
- The critical path `input_mems_instantiation/current_state_reg[1]` $\rightarrow$ `input_mems_instantiation/X_memory_inst/mem_reg[46][0]` suggests that the register holding our current state, specifically looking for `input_W_matrix_state` goes through selecting logic to write the weight value onto the 2D memory register for our kernel matrix.
- The critical path `fifo_out_instantiation/read_ptr_reg[0]` $\rightarrow$ `fifo_out_instantiation/fifo_mem/data_out_reg[26]` suggests that the critical path occurs in the same FIFO dual-port memory instantiations where the register holding the read pointer goes through read memory access logic, likely a decoder to select from a larger array of values.

Table 14: Throughput results for Optimized Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)		**Design 2** (INW=24, R=16, C=17, MAXK=9)	
TVALID/TREADY prob.	**Cycle Count**	**Throughput** (ops/sec)	**Cycle Count**	**Throughput** (ops/sec)
0.1	11328937	809812	54641647	156419
0.5	4944783	1855351	31309490	272984
1.0	4153925	2208588	28408253	300863

Table 15: System delay results for Optimized Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)	**Design 2 (INW=24, R=16, C=17, MAXK=9)**
TVALID/TREADY prob.	**System Delay** (μs)	**System Delay** (μs)
0.1	1.23485	6.39308
0.5	0.53898	3.66322
1.0	0.45278	3.32377

Table 16: Average energy consumption per convolution results for Optimized Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)	**Design 2 (INW=24, R=16, C=17, MAXK=9)**
TVALID/TREADY prob.	**Avg. Energy per Convolution** (J)	**Avg. Energy per Convolution** (J)
0.1	$(36.46mW) * 1.235\mu s = 4.50 \times 10^{-8}$	$(182.61mW) * 6.393\mu s = 1.17 \times 10^{-6}$
0.5	$(36.46mW) * 0.539\mu s = 1.97 \times 10^{-8}$	$(182.61mW) * 3.663\mu s = 6.69 \times 10^{-7}$
1.0	$(36.46mW) * 0.453\mu s = 1.65 \times 10^{-8}$	$(182.61mW) * 3.324\mu s = 6.07 \times 10^{-7}$

Table 17: Area-delay product results for Optimized Top-Level System

	Design 1 (INW=18, R=9, C=8, MAXK=5)	**Design 2 (INW=24, R=16, C=17, MAXK=9)**
TVALID/TREADY prob.	**Area-Delay Product** ($\mu m^2 \cdot \mu s$)	**Area-Delay Product** ($\mu m^2 \cdot \mu s$)
0.1	$48455.35\mu m^2 * 1.235\mu s = 59835.08$	$226974.33\mu m^2 * 6.393\mu s = 1451064.6$
0.5	$48455.35\mu m^2 * 0.539\mu s = 26116.46$	$226974.33\mu m^2 * 3.663\mu s = 831454.9$
1.0	$48455.35\mu m^2 * 0.453\mu s = 21939.61$	$226974.33\mu m^2 * 3.324\mu s = 754409.0$

8. Your new design performs the same computation as your design in Part 4, but it should be faster, larger, and consume higher power. In questions 6 and 7, you compared your new design's energy consumption and area-delay with your Part 4 design. Based on these metrics, would you say your speed-optimized design is more efficient or less efficient than your previous design? Why?

To answer this, we calculate the percentage change between Part 4 (Baseline) and Part 5 (Optimized) using the formula $\frac{\text{Part 5}-\text{Part 4}}{\text{Part 4}} \times 100\%$.

Table 18: Comparison of Baseline (Part 4) vs. Optimized (Part 5) Designs

		Design 1 (INW=18, R=9, C=8, MAXK=5)			**Design 2** (INW=24, R=16, C=17, MAXK=9)		
Metric	**Prob.**	**Part 4**	**Part 5**	**% Change**	**Part 4**	**Part 5**	**% Change**
Cycle Count	0.1	13,400,388	11,328,937	**-15.46%**	74,805,492	54,641,647	**-26.96%**
	0.5	7,017,694	4,944,783	**-29.54%**	51,462,470	31,309,490	**-39.16%**
	1.0	6,225,557	4,153,925	**-33.28%**	48,560,650	28,408,253	**-41.50%**
Throughput (*ops/sec*)	0.1	540,758	809,812	**+49.75%**	84,607	156,419	**+84.88%**
	0.5	1,032,586	1,855,351	**+79.68%**	122,985	272,984	**+121.97%**
	1.0	1,163,972	2,208,588	**+89.75%**	130,334	300,863	**+130.84%**
System Delay (μs)	0.1	1.3400	1.2349	**-7.85%**	7.4805	6.3931	**-14.54%**
	0.5	0.7018	0.5390	**-23.20%**	5.1462	3.6632	**-28.82%**
	1.0	0.6226	0.4528	**-27.27%**	4.8561	3.3238	**-31.55%**
Avg. Energy (J)	0.1	2.67×10^{-8}	4.50×10^{-8}	**+68.54%**	7.26×10^{-7}	1.17×10^{-6}	**+61.16%**
	0.5	1.40×10^{-8}	1.97×10^{-8}	**+40.71%**	4.99×10^{-7}	6.69×10^{-7}	**+34.07%**
	1.0	1.24×10^{-8}	1.65×10^{-8}	**+33.06%**	4.71×10^{-7}	6.07×10^{-7}	**+28.87%**
Area-Delay ($\mu m^2 \cdot \mu s$)	0.1	42,164.86	59,835.08	**+41.91%**	1,232,862.4	1,451,064.6	**+17.70%**
	0.5	22,081.46	26,116.46	**+18.27%**	848,148.2	831,454.9	**-1.97%**
	1.0	19,588.97	21,939.61	**+12.00%**	800,323.6	754,409.0	**-5.74%**

Analysis:

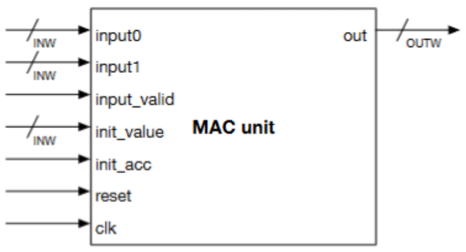
- **Throughput:** The optimizations were highly effective, increasing throughput by up to **89.75%** for Design 1 and **130.84%** for Design 2.
- **Energy:** The optimized design consumes significantly more energy per convolution (approx. **30% to 68%** increase). This is expected because the power consumption increased proportionally more than the delay decreased.
- **Efficiency (Area-Delay):** For Design 1, efficiency worsened (Area-Delay increased by 12-42%). For Design 2, efficiency improved slightly (Area-Delay decreased by 2-6%) in high-probability scenarios, suggesting that the parallelism is justified for larger workloads.

9. If you worked with a partner, please carefully describe each partner's contribution to this part of the project. (If you did not work with a partner, skip this.)

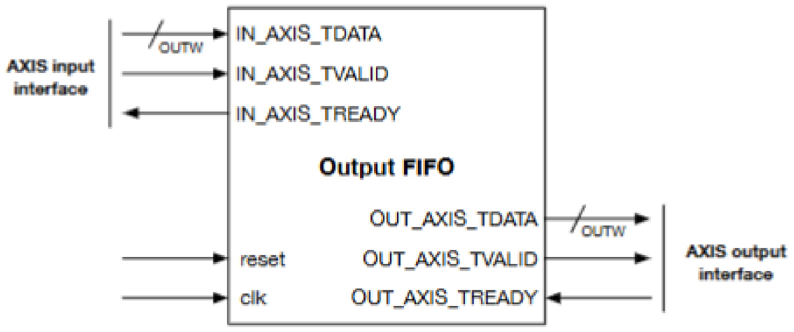
Huabin: Added buffering stages in the FSM to accommodate for the DesignWare staged multiplier instantiation and re-structured the top-level logic to have parallelism with two kernels doing MAC operations.

Ryan: Helped create visual figures in latex for better illustration of how the new optimized system works. Also fixed errors in formatting issues in stat tables.

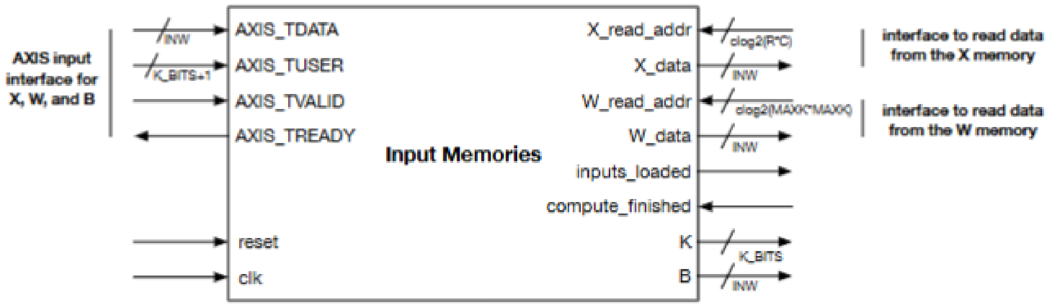
Appendix: System Block Diagrams



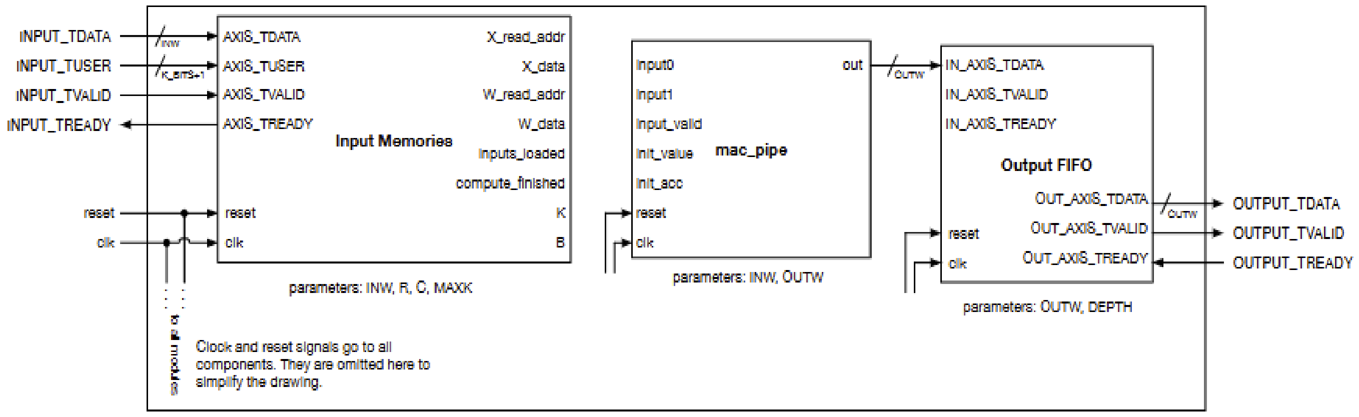
(a) Design 1 (Part 1: MAC Unit)



(b) Design 2 (Part 2: Output FIFO)



(c) Design 3 (Part 3: Input Memory)



(d) Design 4 (Part 4: Top-Level System)

Figure 6: Complete system architecture evolution (Designs 1–4).