ESE 507: Advanced Digital System Design
Final Project
Hardware Accelerator for 2D Convolution

John Abraham
Senior, B.E. Computer Engineering
Stony Brook University
john.abraham@stonybrook.edu

Tyler Higgins
Senior, B.E. Computer Engineering
Stony Brook University
tyler.higgins@stonybrook.edu

1 Introduction

The project's objective is to implement a hardware system that performs 2D convolution with an added bias. Figure 1 shows the top-level module and its port specifications. On the left, four signals prefixed with `INPUT_T` form an AXI-Stream interface through which the system receives input data. On the right, three signals prefixed with `OUTPUT_T` form another AXI-Stream interface for transmitting output data. The design also includes `clk` and `reset` signals; the reset is active-high and applied synchronously—meaning the system resets when `reset == 1` coincides with a rising clock edge.

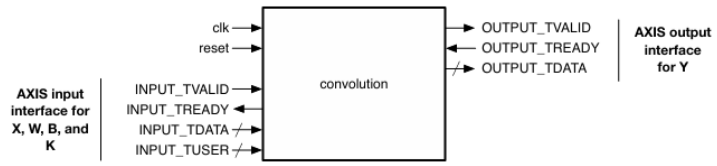

Figure 1: Top-level Design and Port Specifications.

Figure 14 presents a high-level block diagram of the hardware system to be constructed. Each component of the system will be described in detail in the following sections.

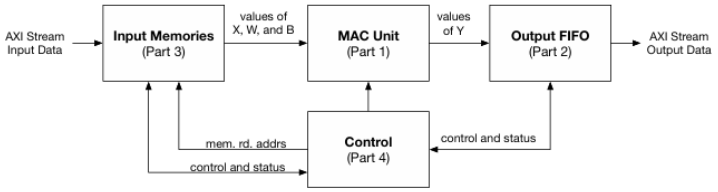

Figure 2: High-Level Block Diagram.

The system accepts an AXI-Stream input stream carrying the values `X`, `W`, and `B`. It performs a 2D convolution with bias and produces the result as an AXI-Stream output. After each convolution is completed, the system can immediately accept a new set of inputs, allowing continuous computation as long as new data is provided.

A key computational element is the multiply–accumulate (MAC) unit, which performs the fundamental operation

$$f \mathrel{+}= a \times b$$

Preprint. Under review.

used throughout the convolution algorithm. The MAC unit constitutes Part 1 of the project and is documented in the Project Part 1 description.

As the MAC unit computes elements of the output matrix, it stores them in the *Output FIFO* module (Part 2 of the project). The Output FIFO buffers these values and delivers them to the system's output interface in AXI-Stream format, as detailed in the Project Part 2 document.

The design also requires input memories to hold the data while computations are performed. These memories are implemented in the *Input Memory* module (Part 3), which contains storage for `X` and `W`, registers for `K` and `B`, and the necessary control logic. Further details appear in the Project Part 3 document.

Part 4 of the project integrates the three components from Parts 1–3 and introduces the control logic that coordinates their operation. This control logic ensures proper synchronization among the input memories, MAC unit, and output memories. Complete specifications are provided in the Project Part 4 document.

Finally, Part 5 focuses on optimizing the overall system speed. Guidelines and performance considerations for this stage are described in the Project Part 5 document.

Parameters

Rather than hard-coding specific matrix dimensions, the design is parameterized to allow flexibility in both matrix size and bit width of input and output values. The system will use the following SystemVerilog parameters:

- `R`: Number of rows in the input matrix `X`. The system must support $\texttt{R} \geq 3$.
- `C`: Number of columns in `X`. The system must support $\texttt{C} \geq 3$.
- `MAXK`: Maximum kernel size. The kernel dimension `K` (rows and columns of the weight matrix `W`) is provided at run time, so the hardware must allow `K` to vary for each input. The maximum allowable value is `MAXK`, with the constraints $\texttt{MAXK} \geq 2$, $\texttt{MAXK} < \texttt{R}$, and $\texttt{MAXK} < \texttt{C}$. The actual kernel size `K` satisfies $2 \leq \texttt{K} \leq \texttt{MAXK}$.
- `INW`: Input bit width (bits per value in `X`, `W`, and bias `B`). The system must support $2 \leq \texttt{INW} < 32$.

There is no explicit upper bound on `R`, `C`, or `MAXK`, but increasing these parameters will lengthen simulation and synthesis time.

In addition to the parameters above, the design uses one derived parameter that is computed automatically from `INW` and `MAXK`:

- `OUTW`: Output bit width (bits per value in the output matrix `Y`). To prevent accumulator overflow, `OUTW` is set large enough to ensure that every convolution result fits within the available bits. The system must support $\texttt{OUTW} \leq 64$. The calculation of `OUTW` is addressed in Part 4 of the project and described in the Part 4 specification document.

2 Part 1

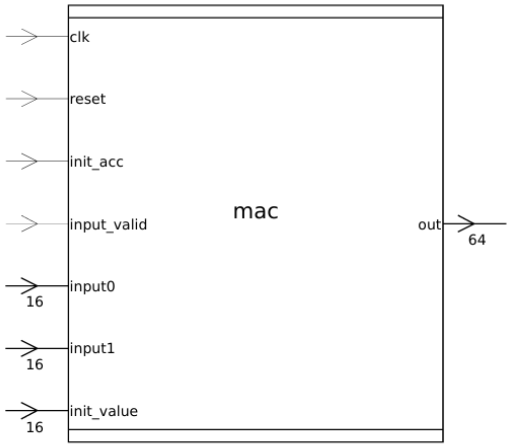

Figure 3: Top-Level Schematic of the MAC

Question 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].")

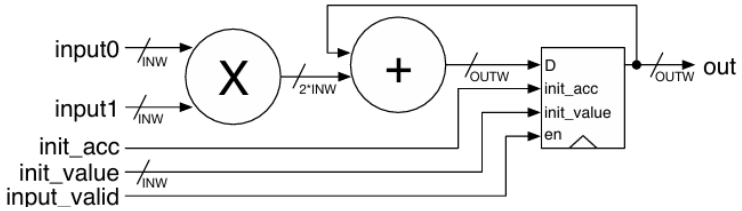

Figure 4: Unpipelined MAC unit.

Answer:

The worst timing path begins at the input port `input0[1]` and ends at the data input D of the most-significant-bit accumulator register `accumulator_reg[55]`. Logically, the path starts at the register/input that provides the new multiplicand/operand, runs through a long chain of arithmetic logic that implements the accumulation (many full-adders and combinational AOI/OAI logic), and finishes at the flop that stores the accumulator MSB.

The timing report lists a long sequence of `FA_X1` (full-adder) cells interleaved with AOI/OAI/NAND/INV cells. The chain accumulates to $\sim 1.35\,\text{ns}$ of data arrival time. In other words, the carry/accumulate propagation through many full-adders and logic gates is the dominant contributor to the critical path.

Clock (ns)	Frequency (MHz)	Power (μW)
1.35	740.74	1673.8
2.00	500.00	1050.1
3.50	285.71	590.85
5.00	200.00	400.03
6.00	166.66	324.18
7.00	142.85	283.35
8.00	125.00	252.73
9.00	111.11	228.92
10.00	100.00	209.87

Table 1: Clock Period, Frequency, and Power Data (Unpipedline)

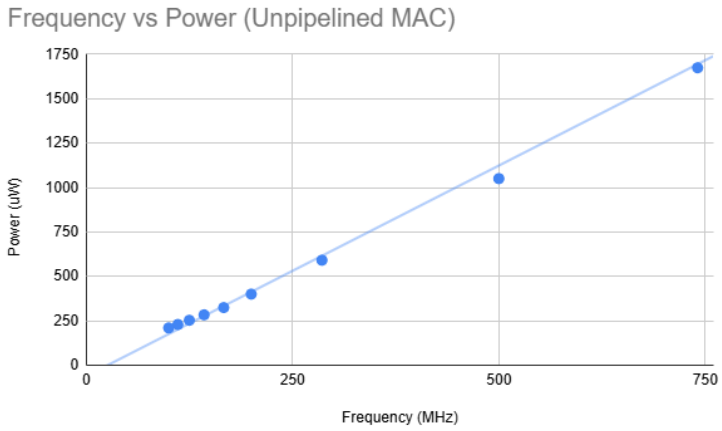

Figure 5: Frequency Vs Area (Unpipelined MAC)

The relationship between frequency and power is linear. As frequency increases, the system's power output also increases because the clock is switching on and off more rapidly, generating more energy.

Clock (ns)	Frequency (MHz)	Area (μm^2)
1.35	740.74	2231.73
2.00	500.00	2041.28
3.50	285.71	1974.78
5.00	200.00	1759.32
6.00	166.66	1726.07
7.00	142.85	1726.60
8.00	125.00	1726.60
9.00	111.11	1726.60
10.00	100.00	1726.60

Table 2: Clock Period, Frequency, and Area Data (Unpipelined)

The relationship between frequency and area is constant until the timing slack reaches 0.0, because the existing hardware can still meet the tighter clock period without modification. Once slack is 0.0, the tools add extra circuitry to meet the timing

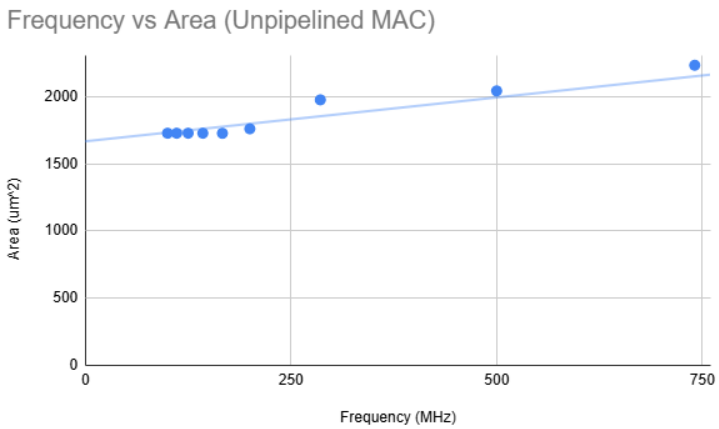

Figure 6: Frequency Vs Area (Unpipelined MAC)

Question 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.

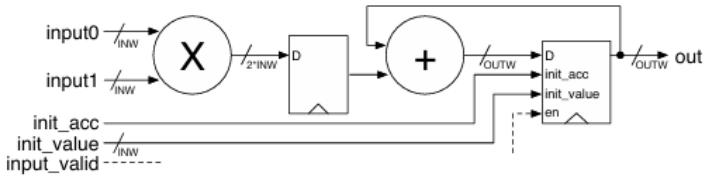

Figure 7: Pipelined MAC unit.

Answer:

From the synthesis reports, we see that pipelining reduced the minimum clock period by 0.04 ns or approximately increased the frequency by 22 Hz. The synthesis data also show us that the critical path starts at input0 and ends at the pipeline_reg. This confirms that pipelining reduced our critical path and made our module faster.

Clock (ns)	**Frequency (MHz)**	**Power (μW)**
1.31	763.35	1831.8
2.00	500.00	1094.5
3.50	285.71	596.49
5.00	200.00	422.52
6.00	166.66	352.68
7.00	142.85	307.25
8.00	125.00	274.07
9.00	111.11	248.26
10.00	100.00	227.62

Table 3: Clock Period, Frequency, and Power Data (Pipeline)

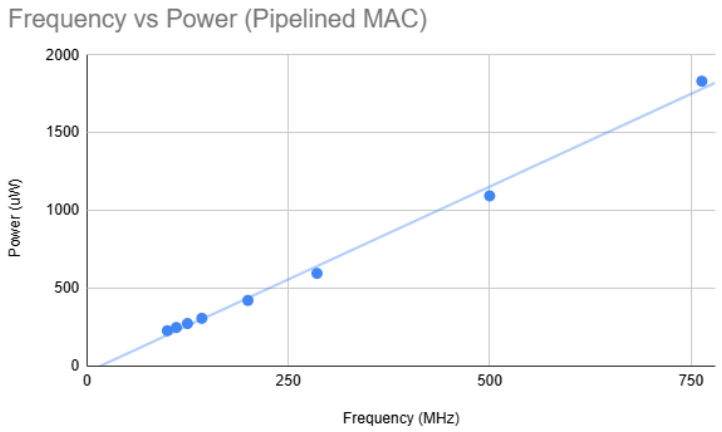

Figure 8: Frequency Vs Power (Pipelined MAC)

In the pipelined design the relationship between frequency and power is linear as well. As the clock frequency rises, the circuit switches more times per second, so dynamic power grows proportionally.

Clock (ns)	**Frequency (MHz)**	**Area (μm^2)**
1.31	763.35	2447.46
2.00	500.00	2291.58
3.50	285.71	2193.96
5.00	200.00	1993.93
6.00	166.66	1946.85
7.00	142.85	1948.18
8.00	125.00	1948.18
9.00	111.11	1948.18
10.00	100.00	1948.18

Table 4: Clock Period, Frequency, and Area Data (Pipeline)

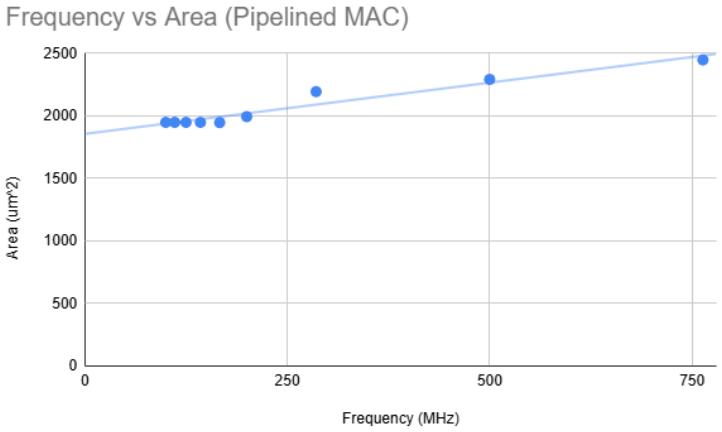

Figure 9: Frequency Vs Area (Pipelined MAC)

In the pipelined version the relationship between frequency and area also stays essentially constant until the timing slack reaches 0.0. At that point the design is at the limit of what the existing hardware can support. To meet any higher clock target, the tools begin to insert extra pipeline registers, buffers, or duplicate logic, which causes the area to increase slightly.

Question 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.

Answer:

One input takes two cycles to compute the output.

The latency for one input is given by:

$$\text{Latency} = 2 \text{ cycles} \times 1.31 \text{ ns/cycle} = 2.62 \text{ ns}$$

From the synthesis report, the energy taken for the minimum clock period (1.31 ns) by the critical path is:

$$P_{\text{critical\_path}} = 1831.8 \ \mu\text{W}$$

The total energy for one input is:

$$E_{\text{input}} = P_{\text{critical\_path}} \times \text{Latency} = 4.799 \times 10^{-12} \text{ J}$$

The total energy for 50 inputs is:

$$E_{\text{total}} = E_{\text{input}} \times 50 = 239.966 \times 10^{-12} \text{ J}$$

Question 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.

Answer:

From the slides, we know the relationship between dynamic power and frequency:

$$P_{dyn} \propto CV^2 f$$

When the minimum clock period increases, the clock frequency decreases. Since power is directly proportional to frequency, this leads to a reduction in power.

However, energy depends on both power and latency. As the clock period increases, latency increases while power decreases. These opposing effects tend to balance each other, so the total energy remains approximately constant.

Below are the values containing the energy required for one input at different frequencies:

Clock (ns)	Frequency (MHz)	Energy (10^{-12} J)
1.31	763.35	4.79
2.00	500.00	4.37
3.50	285.71	4.18
5.00	200.00	4.23
6.00	166.66	4.23
7.00	142.85	4.30
8.00	125.00	4.39
9.00	111.11	4.46
10.00	100.00	4.55

Table 5: Clock Period, Frequency, and Energy Consumption

Question 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.

Answer:

For Unpipelined MAC:

$$\text{Latency} = \text{minimum clock period}$$
$$\text{Throughput} = 2 \text{ cycles} \times \text{Frequency (in MHz)}$$

For Pipelined MAC:

$$\text{Latency} = 2 \text{ cycles} \times \text{minimum clock period}$$
$$\text{Throughput} = 2 \text{ cycles} \times \text{Frequency (in MHz)}$$

Clock (ns)	Frequency (MHz)	Power (μW)	Area (μm^2)	Latency (ns)	Throughput
1.35	740.740741	2231.739986	1673.8	1.35	1481.481482

Table 6: Unpipelined Design Metrics

Clock (ns)	Frequency (MHz)	Power (μW)	Area (μm^2)	Latency (ns)	Throughput
1.31	763.358779	2447.465981	1831.8	2.62	1526.717557

Table 7: Pipelined Design Metrics

Based on the data, you would choose the pipelined design over the unpipelined version when you prefer throughput and speed (frequency) and you don't care about latency, power consumed or area used.

Question 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.)

Answer:

Deeper pipelining would help us reach higher clock frequencies initially but if we keep adding stages inside the multiplier, the power and latency would increase exponentially with only a minimal increase in throughput and thereby counteract the positive effects of pipelining.

If we were to pipeline the multiplier, we would have to implement a multiplier module ourselves by adding pipeline registers in between full-adder levels of the multiplier module.

Pipelining the adder, on the other hand, would not be a good idea because we would increase the latency inside the feedback loop and thereby change the behavior of the accumulator by accumulating wrong values.

Question 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.

Answer:

When INW is held constant at 12 and OUTW changes (24, 48, 64), the clock frequency remains relatively stable: 925.9 MHz, 917.4 MHz, and 892.9 MHz. All three cases share the same critical path from `input0[10]` through `pipeline_reg` to the endpoint, showing that the multiplier stage is the timing constraint regardless of the accumulator width. The modest, slight frequency drop as OUTW increases from 24 to 64 suggests that the accumulator addition grows slightly but remains off the true critical path.

When OUTW is held constant at 48 and INW changes (8, 16, 24), the clock frequency changes more noticeably: 1470.6 MHz, 1612.9 MHz, and 1587.4 MHz. The critical path location also changes: for INW = 8 it ends at `accumulator_reg[43]`, for INW = 16 at `accumulator_reg[33]`, and for INW = 24 at `accumulator_reg[37]`. This shows that increasing input width directly affects the multiplier delay, which is the main factor in the critical path. If you think about it, with increasing width in general, the number of half-adders and full-adders necessary for each bit increases too, this increases the critical path and thereby increases the clock period. We see more of an increase with largers INW because we perform multiplication with INW inputs, this means more adders and hence longer critical path and bigger time periods.

Since the multiplier is already pipelined with a register between it and the accumulator, the accumulator timing is isolated in a separate stage. Therefore, INW changes affect the critical path through the multiplier, while OUTW changes have less impact because they only affect the accumulator stage. This explains why the design is far more sensitive to INW changes than to OUTW.

Question 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.

Answer:

$$\textbf{Given:} \quad INW = 5,\ OUTW = 16$$

$$\text{Range of 5-bit signed integers: } [-2^4,\ 2^4 - 1] = [-16,\ 15]$$

$$\text{Range of 16-bit signed accumulator: } [-2^{15},\ 2^{15} - 1] = [-32768,\ 32767]$$

$$\text{Maximum positive product magnitude} |P_{\max}| = (-16) \times (-16) = 256$$
$$\text{Maximum negative product magnitude} |P_{\max}| = (-16) \times (15) = -240$$

So if we initialize the accumulator to zero and increment it using the max positive value (256), we can do 127 MACs.

$$\left\lfloor \frac{32767}{256} \right\rfloor = 127$$

If we initialize the accumulator to zero and increment it using the max negative value (-240), we can do 136 MACs.

$$\left\lfloor \frac{32768}{240} \right\rfloor = 136$$

If we initialize the accumulator to the most negative number -32768 and increment it using the max positive value (256), we can do 256 MACs:

$$\left\lfloor \frac{65536}{256} \right\rfloor = 256$$

If we initialize the accumulator to the most negative number 32767 and increment it using the max negative value (-240), we can do 273 MACs:

$$\left\lfloor \frac{65536}{240} \right\rfloor = 273$$

Therefore, to guarantee no overflow in either direction:

$$\min(256,\ 273) = 256$$

$$\boxed{K_{\max} = 256 \text{ MAC operations (overflow on the 257th).}}$$

Question 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.)

Answer:

Tyler was responsible for implementing and testing the unpipelined MAC unit. He synthesized the design, collected the resulting performance and area data, and generated plots to visualize the results. John worked on the pipelined version of the MAC unit, carrying out the synthesis process, gathering the corresponding data, and creating plots for comparison. After completing the individual design tasks, we came together to collaborate on answering the report questions. In addition, we jointly worked on organizing and formatting the final document using LaTeX to ensure a professional presentation.

3 Part 2

Question 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.

Answer:

The FIFO discussed in class used only a single signal to determine when we needed to assert the `wr_en` signal. In our project, since we used AXI-Stream, we had to perform a logical AND operation with the `IN_AXIS_TVALID` and `IN_AXIS_TREADY` signals that the FIFO received as inputs. This was done to ensure that both the module receiving the data (in this case, our FIFO) and the module sending the data were ready simultaneously.

The `IN_AXIS_TREADY` signal was asserted either when the FIFO was not full or when the FIFO was full but being read at that instant in time (i.e., when the `read_en` signal was asserted). The `read_en` signal is enabled by the logical AND operation of the `OUT_AXIS_TVALID` and `OUT_AXIS_TREADY` signals. Finally, the `OUT_AXIS_TVALID` signal is asserted whenever the FIFO is not empty.

Question 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.)

Answer:

Clock (ns)	**Frequency (GHz)**	**Area (μm^2)**	**Power (mW)**	**Critical Path**
0.916	1.091	3559.079947	2.3141	Startpoint: `tail_reg[4]` → Endpoint: `mem_inst/data_out_reg[7]` (rising edge-triggered flip-flops)

Table 8: FIFO Design Metrics (OUTW = 24, Depth = 19)

From the synthesis report, we see that the critical path starts from the `tail_reg` and ends at `data_out` of the memory instantiation. Even though the head and tail are almost identical in terms of the logic required, the tail logic includes additional computation to determine the address if a read occurs in the next cycle. This look-ahead address is what is fed into the memory during any read operation. Therefore, the longest path, also known as the critical path, is the path that passes through the tail look-ahead increment logic and into the memory.

Question 3

Repeat Question 2 with both:
a. OUTW = 12 and DEPTH = 19
b. OUTW = 24 and DEPTH = 38

Report all statistics for both designs. Explain the trends you observe in **area**, **power**, and **frequency**. In other words, describe how these three sets of parameters affect the area, power, and frequency, and explain the reasoning behind these effects.

Answer:

From the slides, we know the relationship between dynamic power and frequency:

Let us use `OUTW` = 24 and `DEPTH` = 19 as our reference point to answer this question. This configuration used $24 \times 19 = 456$ flip-flops for the FIFO.

OUTW	Depth	Clock (ns)	Frequency (GHz)	Area (μm^2)	Power (mW)	Critical Path
24	19	0.916	1.091	3559.08	2.314	Startpoint: `tail_reg[4]` → Endpoint: `mem_inst/data_out_reg[7]` (rising edge-triggered flip-flops)
12	19	0.855	1.169	1958.97	1.338	Startpoint: `tail_reg[1]` → Endpoint: `mem_inst/data_out_reg[0]` (rising edge-triggered flip-flops)
24	38	0.840	1.190	6817.04	5.551	Startpoint: `tail_reg[2]` → Endpoint: `mem_inst/data_out_reg[23]` (rising edge-triggered flip-flops)

Table 9: FIFO Design Metrics for Various OUTW and Depth Configurations

$$P_{dyn} \propto CV^2 f$$

For the second case, when we reduced `OUTW` to 12 and kept `DEPTH` constant, we observe that the area almost reduced by half. This makes sense, as the number of flip-flops required was lower, i.e., $12 \times 19 = 228$ flip-flops. The clock period decreased, resulting in an increased frequency. Since the area also decreased by nearly half, this reduction affects the capacitance in the dynamic power formula. With smaller area, there are fewer gates and shorter interconnects, leading to lower dynamic power overall.

For our third case, when `OUTW` remained constant at 24 and `DEPTH` was increased to 38, we see that the area nearly doubled, requiring approximately 912 flip-flops. As the area increased, the capacitance also increased, which in turn led to higher overall power consumption.

Question 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.)

Answer:

Both John and Tyler worked on the FIFO code and performed the synthesis for the required parameter sets. We also collaborated to analyze the results and write the answers for Part 2 of the final report.

4 Part 3

Question 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.

Answer:

In Part 3, we designed an input memory module that manages and loads the input data for our convolution operations. The module receives data through an AXI-Stream interface and stores the inputs as a matrix X of size $R \times C$, a weight matrix W of size $K \times K$, a bias value B, and a kernel size K. The module uses a finite state machine (FSM) to step through the data-loading process while providing read access to the stored matrices during convolution.

The memory is instantiated into two blocks. The `X_matrix_mem` stores the X matrix of size $R \times C$, where all elements are `INW` bits wide. The memory is loaded in row-major order and provides random access during computation. The `W_matrix_mem` stores the W matrix with a maximum capacity of `MAXK` $\times$ `MAXK`. Only $K \times K$ elements are actually used for a given convolution, but K can vary in size between 2 and `MAXK`. Both memories use a multiplexed addressing scheme: during loading, the write addresses `X_wr_addr` and `W_wr_addr` are used, and during computation (when `inputs_loaded` is asserted) they read from the external interfaces `X_read_addr` and `W_read_addr`.

The FSM is divided into five states

IDLE: In this state, the module waits for the first valid data transfer to determine what type of data is arriving. The FSM enters this state on system reset or after `compute_finished` is asserted in the **LOADED** state. When valid data arrives (both `AXIS_TVALID` and `AXIS_TREADY` are high), the state checks the `new_W` bit (`AXIS_TUSER[0]`) to determine the data type. If `new_W = 1`, the state stores the first W value at address 0, saves K from `AXIS_TUSER` into `K_reg`, computes `w_limit` $= K \times K$, and initializes `W_wr_addr` to 1. If `new_W = 0`, it stores the first X value to address 0 and initializes `X_wr_addr` to 1. The FSM exits to **LOAD_W** when `new_W = 1`, or to **LOAD_X** when `new_W = 0`.

LOAD_W: This state loads the remaining $K^2 - 1$ weight values, since the first value was already written in **IDLE**. When `AXIS_TVALID` and `AXIS_TREADY` are both high, the state writes `AXIS_TDATA` to `W_matrix_mem` at address `W_wr_addr` and then increments `W_wr_addr`. The only way to exit this state is when `W_wr_en` is high and `W_wr_addr = w_limit - 1`, meaning the last weight value has been written.

LOAD_B: This state loads exactly one bias value. It waits until both `AXIS_TVALID` and `AXIS_TREADY` are high, then saves `AXIS_TDATA` into `B_reg`. After one valid data transfer, the FSM exits this state and transitions to **LOAD_X**.

LOAD_X: In this state, all $R \times C$ values of the input matrix X are loaded. Each cycle in which `AXIS_TVALID` and `AXIS_TREADY` are both high, the state writes `AXIS_TDATA` to `X_matrix_mem` at address `X_wr_addr` and increments `X_wr_addr`. When the final X value is written, the state sets `inputs_loaded` to 1 to indicate that all data is ready. The FSM exits to the **LOADED** state when `X_wr_en` is asserted and `X_wr_addr = R` $\times C - 1$.

LOADED: This state indicates that all data has been loaded into the module and is ready for computation. The state keeps `inputs_loaded = 1`, allowing the convolution control logic to read from the memories. Meanwhile, `AXIS_TREADY` is set to 0 to prevent new data from arriving during convolution. The FSM exits this state when `compute_finished` is asserted, indicating that the convolution is complete, and then transitions back to **IDLE**. When exiting, `inputs_loaded` is cleared (set to 0).

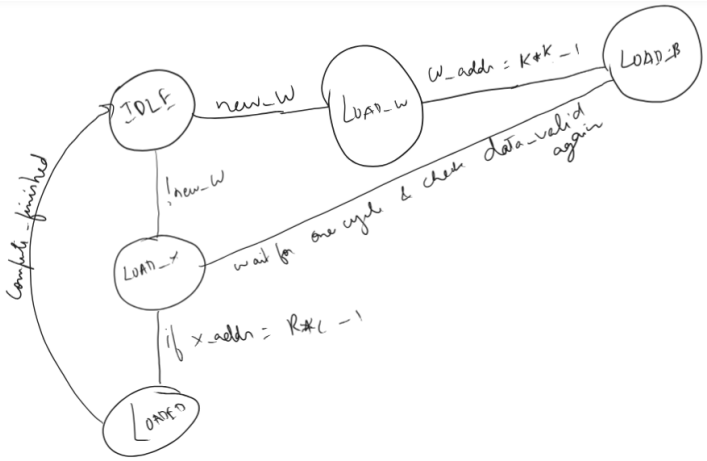

Figure 10: FSM Implementation for Part 3

The module uses AXI-Stream handshaking in which `AXIS_TREADY` is high in all states except **LOADED**. In **LOADED**, `AXIS_TREADY` is deasserted to prevent any new data from arriving during the convolution computation. A data transfer occurs only when both `AXIS_TVALID` and `AXIS_TREADY` are high; in that case, `data_valid` is given by

$$\texttt{data\_valid} = \texttt{AXIS\_TVALID} \,\&\, \texttt{AXIS\_TREADY}.$$

The write-enable signals `W_wr_en` and `X_wr_en` control when data is written into their corresponding memories. `W_wr_en` asserts when the state is **LOAD_W** and valid data is present, or when the state is **IDLE** with valid data and `new_W = 1`. Similarly, `X_wr_en` asserts when the state is **LOAD_X** with valid data, or when the state is **IDLE** with valid data and `new_W = 0`. This allows the first value to be written in **IDLE**, while subsequent values are written in their respective loading states.

The module uses several registers to track the loading process. `K_reg` captures the kernel size from `AXIS_TUSER` in the **IDLE** state and continuously drives the output port K. `B_reg` holds the bias value from `AXIS_TDATA` during **LOAD_B** and continuously drives the output port B. The value `w_limit` is calculated as $K \times K$ and used as a comparison to determine when all W values have been loaded. The write-address registers `X_wr_addr` and `W_wr_addr` track the current write positions in their respective memories and increment automatically after each successful write.

Question 2

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

- the parameters (R, C)
- the value of K for this input
- 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 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:

$$\text{INW} = 10, \quad R = 15, \quad C = 13, \quad \text{MAXK} = 7, \quad \text{and} \quad \text{INPUT\_VALID\_PROB} = 1$$

When `new_W == 1`, the testbench will begin by feeding in the $K \times K$ values of W, then the value of B, then the $R \times C = 15 \times 13 = 195$ values of X. In other tests where `new_W == 0`, the system will simply feed in the $R \times C = 15 \times 13 = 195$ values of X.

Simulate this design in QuestaSim'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 where `new_W == 0`, and record both the number of cycles and the value of K.

Hint: You can view the amount of simulated time that passes in the waveform and divide it by 10 ns 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 = 0` and `new_W = 1`).

In the report, quantify how efficient your system is with respect to the number of clock cycles by computing the following ratios:

$$\text{When } \texttt{new\_W == 1:} \quad \text{efficiency} = \frac{K^2 + 1 + R \times C}{\text{cycles}}$$

$$\text{When } \texttt{new\_W == 0:} \quad \text{efficiency} = \frac{R \times C}{\text{cycles}}$$

In these metrics, a value of 1.0 represents perfect efficiency — 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, discuss where your logic could be improved.

Answer:

Table 10: Efficiency results for `new_W = 0` with $R = 15$, $C = 13$

Test #	# Cycles	K	Efficiency
1	195	3	1.0
2	195	5	1.0
3	195	3	1.0
4	195	2	1.0

Table 11: Efficiency results for `new_W = 1` with $R = 15$, $C = 13$

Test #	# Cycles	K	Efficiency
1	205	3	1.0
2	205	3	1.0
3	200	2	1.0
4	212	4	1.0

Question 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`.

Answer:

$$\text{When } \texttt{new\_W} = 0, \quad \text{cycle count} = R \times C$$

$$\text{When } \texttt{new\_W} = 1, \quad \text{cycle count} = K^2 + 1 + R \times C$$

Question 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.

Answer:

Design	Time (ns)	Freq (MHz)	Power (mW)	Area (μm^2)	Critical Path
`INW24_R9_C8_MAXK4`	0.81	12.35	15.305	15501.95	state_reg[1] $\rightarrow$ X_matrix_mem/data_out_reg[17]
`INW10_R15_C13_MAXK7`	0.86	11.63	18.5309	18591.22	state_reg[1] $\rightarrow$ X_matrix_mem/data_out_reg[5]

Table 12: Synthesis Results and Critical Path Summary

For `INW10_R15_C13_MAXK7`, the number of flip-flops required was $10 \times 15 \times 13 = 1950$, compared to 1728 for `INW24_R9_C8_MAXK4`. Since the number of flip-flops increased, the overall area also increased. When area increases, the capacitance rises due to the addition of more gates and interconnects. As capacitance increases, the overall dynamic power consumption also increases, even though the operating frequency decreases slightly.

The critical path for both cases starts from the `state_reg` variable, which originates from the next-state logic. It then passes through the `always_ff` block that contains the current-state logic and finally enters the `X_matrix_mem` register, which stores the X matrix values. The critical path passes through the X matrix rather than the W matrix because the X matrix (with $R \times C$ values) is always larger than the largest weight matrix (with MAXK $\times$ MAXK values).

Question 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.)

Answer:

John began by designing the datapath and creating a finite-state machine to determine the required states and control logic. He then wrote pseudocode and began the initial implementation. Afterward, John passed the code to Tyler, who took the completed FSM and finished implementing it and tested it using the professor's provided testbench. Once the testing was complete, Tyler and John synthesized the designs and generated the synthesis reports. Both John and Tyler worked collaboratively to answer the final report questions.

5 Part 4

Question 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.

Answer:

The Part 4 design combines the input memory, MAC pipeline, and output FIFO modules using control logic to create a 2D convolution hardware accelerator. The input memory stores an input matrix X of size $R \times C$ and a weight kernel W of size $K \times K$. A finite state machine (FSM) controls the convolution process, which is done using a sliding window. For each output position (r, c), the system computes a dot product between the $K \times K$ region and the weight kernel, then adds a bias value. This process repeats for all valid window positions to produce an output matrix of size $(R - K + 1) \times (C - K + 1)$.

The FSM is divided into five states:

WAIT_INPUTS: In this state, the system waits for all of the input data to be loaded. The FSM enters this state when the system is reset or after the previous convolution has completed, and exits when the `inputs_loaded` signal is asserted. Upon exiting, it resets all loop counters, resets the pipeline counter, and sets the `first_product` flag to prepare for bias initialization.

COMPUTE: In this state, the controller reads the input matrix and weight matrix and feeds them to the MAC unit. The memory addresses are computed as

$$X_{\text{addr}} = (r + i)\, C + (c + j), \qquad W_{\text{addr}} = i * K + j.$$

There is a one-cycle latency until the memory responds with data, after which the state asserts `mac_input_valid` to indicate that valid data is ready for the MAC. On the first product of each window, it also asserts `mac_init_acc` to load the bias into the accumulator; on later cycles, new products are added to the accumulator sum. The state exits only when the counters satisfy $i = K - 1$ and $j = K - 1$, meaning that K^2 products have been computed.

DRAIN_PIPE: This state allows the MAC pipeline to complete and ensures that all data from the computation is available. Because the MAC has a one-cycle latency, the controller must wait for the pipeline to produce the final result. In this state, a counter `pipe_cnt` is incremented, and the state exits when `pipe_cnt` $\geq$ `MAC_LAT`. This provides enough delay to drain the pipeline of all computed values.

WRITE_RESULT: In this state, the controller writes the completed computation into the FIFO. It first asserts `fifo_in_tvalid` to indicate that valid data is ready, then waits for `fifo_in_tready` to determine whether the FIFO has available space. When this handshake occurs (both signals are high), it resets `pipe_cnt` and sets the `first_product` flag for the next window. The state exits in one of two ways: (1) if $r = R_{\text{out}} - 1$ and $c = C_{\text{out}} - 1$, meaning all outputs have been computed and stored, the FSM proceeds to **DONE_PULSE**; or (2) it moves to the next window and returns to **COMPUTE**.

DONE_PULSE: This state signals that the convolution has completed in full. It asserts `compute_finished` for one cycle and then immediately transitions back to **WAIT_INPUTS**.

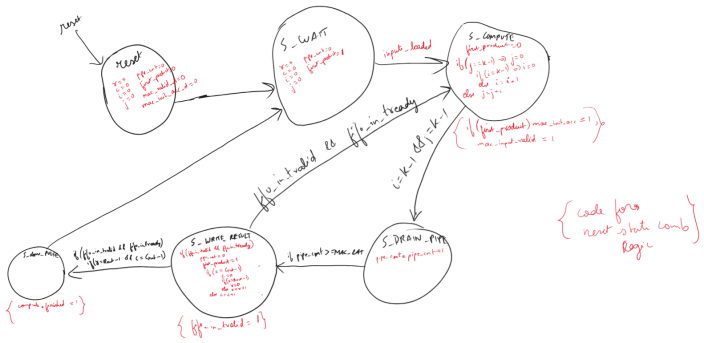

Figure 11: FSM Implementation for Part 4

An important aspect of the control logic in this design is handling the memory's read latency. When the control logic sends an address to memory in cycle N, the corresponding data does not arrive until cycle $N + 1$. The MAC unit requires its control signals (`mac_init_acc` and `mac_input_valid`) to be synchronized with the arrival of valid data, not with the cycle when the address is sent.

To solve this synchronization problem, all MAC control signals are delayed by one clock cycle using registers. The original signals `mac_input_valid` and `mac_init_acc` are passed through registers to create delayed versions `mac_valid_delayed` and `mac_init_acc_delayed`. The MAC unit connects to these delayed signals so that the control signals and the data from memory arrive at the MAC simultaneously. This synchronization prevents the MAC from processing invalid or stale data.

Question 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.

Answer:

When new_W is 1, the equation for number of cycles is:

$$\boxed{K^2 + 1 + R * C + (R - K + 1) * (C - K + 1) * \left(K^2 + \text{MAC\_Latency} + 1\right)}$$

From simulation of one of our convolutions $INW = 12$, $R = 9$, $C = 8$, $K = 3$, we can verify that loading data and computation of one convolution took 586 cycles. This is the breakdown of the cycles:

- Load weight matrix = $K \times K = 9$ cycles
- Load bias value = 1 cycle

- Load input matrix = $R \times C = 9 \times 8 = 72$ cycles

Remaining cycles:

$$586 - 72 - 9 - 1 = 504 \text{ cycles}$$

Output matrix size: $(R - K + 1) \times (C - K + 1)$

Since $(9 - 3 + 1) \times (8 - 3 + 1) = 7 \times 6$, i.e., 42 elements in output matrix.

$$\frac{504 \text{ cycles}}{42 \text{ elements}} = 12 \text{ cycles for computation of one element}$$

- 9 cycles for 9 MAC operations
- 2 cycles for draining the MAC, aka MAC_LATENCY, because our mac is pipelined.
- 1 cycle for writing the MAC output to the FIFO

When new_W is 0, we do not load the weight matrix or the Bias value, so the equation for number of cycles becomes:

$$\boxed{R * C + (R - K + 1) * (C - K + 1) * \left(K^2 + \text{MAC\_Latency} + 1\right)}$$

From our equations and our verified data, we see that a huge contribution of cycles come from the compute phase of our system The biggest factor that decides the total the number of cycles is K, or in other words, MAXK. Because if MAXK is higher, then your output matrix gets smaller if K is equal to MAXK, and when that happens the number of computations reduces too. Obviously R and C of the input matrix matters too, because if we have a small input matrix, out output matrix gets smaller too. To obtain a very fast system, i.e. one with a small compute phase, we would have to choose the values of R,C, and K, such that R and C are small and K is big, but in moderation.

Question 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.

Answer:

Design	Time (ns)	Freq (MHz)	Power (mW)	Area (μm^2)	Critical Path
INW12_R9_C8_MAXK5	1.101	908.27	8.6444	12819.87	X_matrix_mem/data_out_reg[3] → MAC/pipeline_reg_reg[17]
INW18_R9_C8_MAXK5	1.357	736.92	10.3955	19296.70	X_matrix_mem/data_out_reg[11] → MAC/pipeline_reg_reg[33]
INW24_R16_C17_MAXK9	1.670	598.80	38.5089	74271.99	X_matrix_mem/data_out_reg[2] → MAC/pipeline_reg_reg[46]

Table 13: Synthesis Results and Critical Path Summary

The critical path in all three Part 4 syntheses has the same general structure: *Memory Output* $\rightarrow$ *Multiplier Input* $\rightarrow$ *Pipeline Register*. The path begins at the memory module's output register, travels through combinational logic to the MAC module's multiplier, and ends at the `pipeline_reg` that sits between the multiplier and the accumulator.

This makes sense architecturally because the memory read is synchronous, which means data comes from a flip-flop in the memory. The multiplier is purely combinational, so the 12-bit, 18-bit, or 24-bit multiplication happens in combinational logic, and the pipeline register then captures the product before it enters the accumulator.

The critical path location is consistent across all three syntheses because the datapath structure does not change: we are always reading from memory, multiplying, and registering the result. The clock frequency decreases as INW increases because larger multipliers require more logic depth, which leads to longer delays through the combinational multiplier.

The slight variations in the exact register names likely reflect synthesis tool optimizations or small differences in the memory implementation for different parameter sets, but the critical path always follows the same route.

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

Answer:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	cycle count	throughput (ops/sec)	cycle count	throughput (ops/sec)
0.10	13,330,898	552,790.724	73,422,762	81,555.41673
0.50	6,892,401	1,101,128.991	49,420,270	121,165.3427
1.00	5,881,452	1,252,955.351	48,671,745	128,300.8371

Question 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 six scenarios evaluated in question 4, determine the delay in seconds (or ms, μs, 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.

Answer:

$$\text{Average Delay} = \frac{\text{Cycle Count} \times \text{Clock Period}}{10{,}000}$$

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	cycle count	delay	cycle count	delay
0.10	13,330,898	1.81 us	73,422,762	12.26 us
0.50	6,892,401	0.908 us	49,420,270	8.25 us
1.00	5,881,452	0.798 us	48,671,745	7.79 us

Question 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.

Answer:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	power (W)	energy (nJ)	power (W)	energy (nJ)
0.10	0.0103955	18.805489	0.0385089	472.1807765
0.50	0.0103955	9.440765	0.0385089	317.8210793
1.00	0.0103955	8.298784	0.0385089	300.1453527

Question 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.

Answer:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	area (μm^2)	area-delay product	area (μm^2)	area-delay product
0.10	19296.70378	0.0349077923	74271.98711	0.9106894903
0.50	19296.70378	0.01752447165	74271.98711	0.6128804560
1.00	19296.70378	0.01504000584	74271.98711	0.5788893416

Question 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.)

Answer:

John and Tyler first designed the datapath and developed a finite state machine to identify the required states and control logic. They then jointly implemented the design. In addition, both John and Tyler worked together to answer the questions and prepare the final report.

6 Part 5

Question 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.)

Answer:

We decided to split our efforts into two.

John's Design:

I focused on modifying our part 4 design and control logic so that the system can overlap the loading of input data with computation of previous input data. In essence, I double-buffered our input memories so that while our system is computing a convolution using the MAC, our `input_mems` module is concurrently pre-loading the next set of inputs onto another memory bank. Therefore, our system would not need to waste time waiting for inputs to be loaded.

Basically, my idea was to use 2 bits to represent which W and X matrix memory module was being written to in each load data phase.

$$\begin{aligned} 00 &\rightarrow W_1 B_1,\ X_1 \\ 01 &\rightarrow W_1 B_1,\ X_2 \\ 10 &\rightarrow W_2 B_2,\ X_1 \\ 11 &\rightarrow W_2 B_2,\ X_2 \end{aligned}$$

At reset, the default values were

$$\text{current\_write} = 2'b00$$
$$\text{current\_read} = 2'b00$$

There are two cases for updating the write pointer:

$$\textbf{Case 1: } \text{new\_W} = 1$$
$$\text{next\_write} = [\neg\text{current\_write}[1],\ \neg\text{current\_write}[0]]$$

This updates the write pointer to write to a different W and X matrix memory module, i.e., the modules that were not just written.

$$\textbf{Case 2: } \text{new\_W} = 0$$
$$\text{next\_write} = [\text{current\_write}[1],\ \neg\text{current\_write}[0]]$$

This updates the write pointer to write to a different X matrix memory module while keeping the current W matrix pointer unchanged.

I first started out by writing down the pattern of read and write and how the ptrs needed to change. I created this table to show the pattern:

In the process of deciding how to instantiate, I initially thought of duplicating input_mems and somehow send control signals from instantiation to the other. The reason I thought about this idea was that I wanted to add an additional MAC instance to parallelize the computations because that is where a huge chunk of cycles were wasted. But I could wrap my head around the idea of using two instances of input_mems and how they would interact with each other and the two MACs.

In interest of time, I decided to limit my modifications to within one module, in order to ease debugging. So I decided that I'll create 4 instances of memory modules. 2 for the weight matrices and 2 for input matrices. This way they'll all get the same data and they'll all output data via the same output ports of input_mems module.

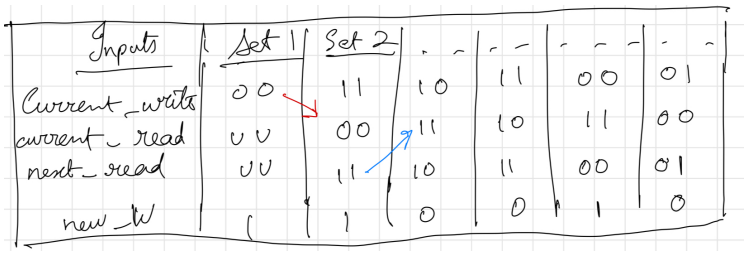

Figure 12: Desired Pattern to Update Pointers

Moving onto the way I wanted to code it, I initially thought about using 2 FSMs and have two state objects called state1 and state2, and have two next-state objects called nstate1 and nstate2. The complexity behind this idea was the control signals necessary to communicate between each FSM, telling each FSM when the inputs were loaded and updating pointers accordingly. But I thought it would be easier if I just let the datapath control the updation of the write pointers. In hindsight, I think that my current implementation was more complex than that method.

So in my implementation i use the same number of states as part 3 of this project but with adjustments to the write_en and address logic.

IDLE State: The system waits for valid input data (AXIS_TVALID asserted). When data arrives, it examines the `new_W` bit (AXIS_TUSER[0]) to determine the data type. If `new_W` is high, the system transitions to LOAD_W to begin loading a new weight matrix; otherwise, it transitions to LOAD_X to load only a new X matrix (reusing the previous W matrix and bias). During the IDLE-to-LOAD_W transition, the system also calculates `next_write` to toggle the appropriate memory bank pointers, captures the K value from AXIS_TUSER into the appropriate K register (K_reg or K_reg2), calculates `w_limit` as $K \times K$ to know how many weight values to expect, and initializes the write address counters.

LOAD_W State: The system sequentially writes incoming weight matrix values to the selected W memory bank (either W_matrix_mem or W_matrix_mem2, depending on `current_write[1]`). The write address counter `W_wr_addr` increments with each valid data transfer. The state continues receiving weight values until the address reaches `w_limit` $- 1$ (meaning K^2 values have been stored), at which point it transitions to LOAD_B to receive the bias value. The system keeps AXIS_TREADY high to indicate it's ready to accept data.

LOAD_B State: The system loads exactly one data value representing the bias term B. This value is stored in either `B_reg` or `B_reg2` depending on which memory bank is currently being written to (`current_write[1]`). After successfully capturing the bias value (when data_valid is asserted), the state immediately transitions to LOAD_X to begin loading the X matrix. This is a single-cycle state under normal operation.

LOAD_X State: The system sequentially writes incoming X matrix values to the selected X memory bank (either X_matrix_mem or X_matrix_mem2, depending on `current_write[0]`). The write address counter `X_wr_addr` increments with each valid data transfer until all $R \times C$ values have been stored. When the last X value is written (address equals $R \times C - 1$), the behavior depends on whether this is the first dataset or a subsequent one: if `first_inputs_loaded` is not yet set, this is the initial dataset, so the state transitions to IDLE, sets `first_inputs_loaded` high, and updates `current_read` to point to the newly loaded banks; if `first_inputs_loaded` is already high, this is a subsequent dataset being loaded concurrently with computation, so `inputs_loaded2` is asserted and the state transitions to LOADED.

LOADED State: This state indicates that a complete set of inputs (X, W, K, and B) is available in memory and ready for computation while the system simultaneously loads the next dataset into the alternate memory banks. The state asserts AXIS_TREADY low to backpressure the input stream if necessary (though the double-buffering typically allows continuous loading). The system remains in LOADED until the `compute_finished` signal is asserted, indicating the computation engine has finished processing the current dataset. Upon receiving `compute_finished`, the state clears `inputs_loaded2`, updates `current_read` to point to the newly loaded memory banks (swapping

the read and write pointers), and transitions back to IDLE to begin the cycle again with the next dataset.

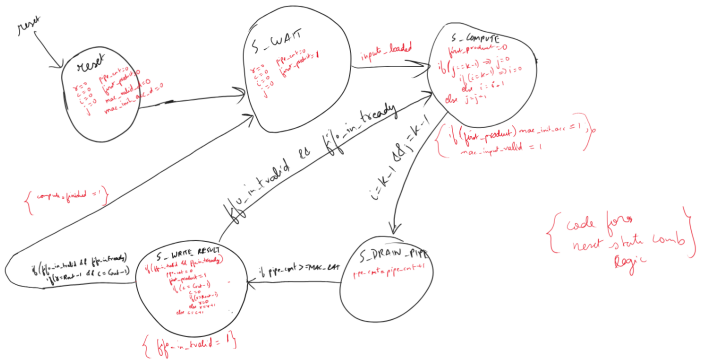

Figure 13: Modified Part 4 FSM Implementation for Part 5 top-level (removed Done_Pulse state)

Some patterns that I realized along the way were that after the first set of inputs were loaded, the inputs loaded signal will always be asserted because if you think about it, there will always be a second set of data that was loaded into the unused the memory modules. Therefore, the MAC will be constantly performing computations.

One of the struggles of this approach was that in the IDLE state, new_W changes where we write data. The write_en enable signals needed to know as soon as new_W changes. Therefore, I kept track of the next_write signal combinationally. The next_write signal is what drives the write_en signal in the IDLE state. This one condition took me some time to figure out because initially, I only used current_write to drive the write_en signals. Another key modification was that next_write should only be modified based on new_W when we were in the IDLE state. This intuitive but crucial detail was also missed when i initially implemented the code. But I was able to figure it out after debugging for quite some time. Another problem that I realized after hours of debugging was that the current_read and current_write pointers cannot be updated in the same state. I initially updated current_write and current_read when the last X data value was being written. This, however, is incorrect as the write pointer depends on the new_w value in the IDLE state only. Everything makes sense now when I look at it but when writing the code, I wasn't to able to think that far ahead.

My code works for all designs and all probabilities of TVALID and TREADY up to 0.3. For some reason, if the probabilities were less than that, the simulation runs into errors. I only found this error while writing the report and hence wasnt able to add the data for 0.1 probability of TVALID and TREADY. Given more time and little btter time management, I know I definitely could have fixed the problem.

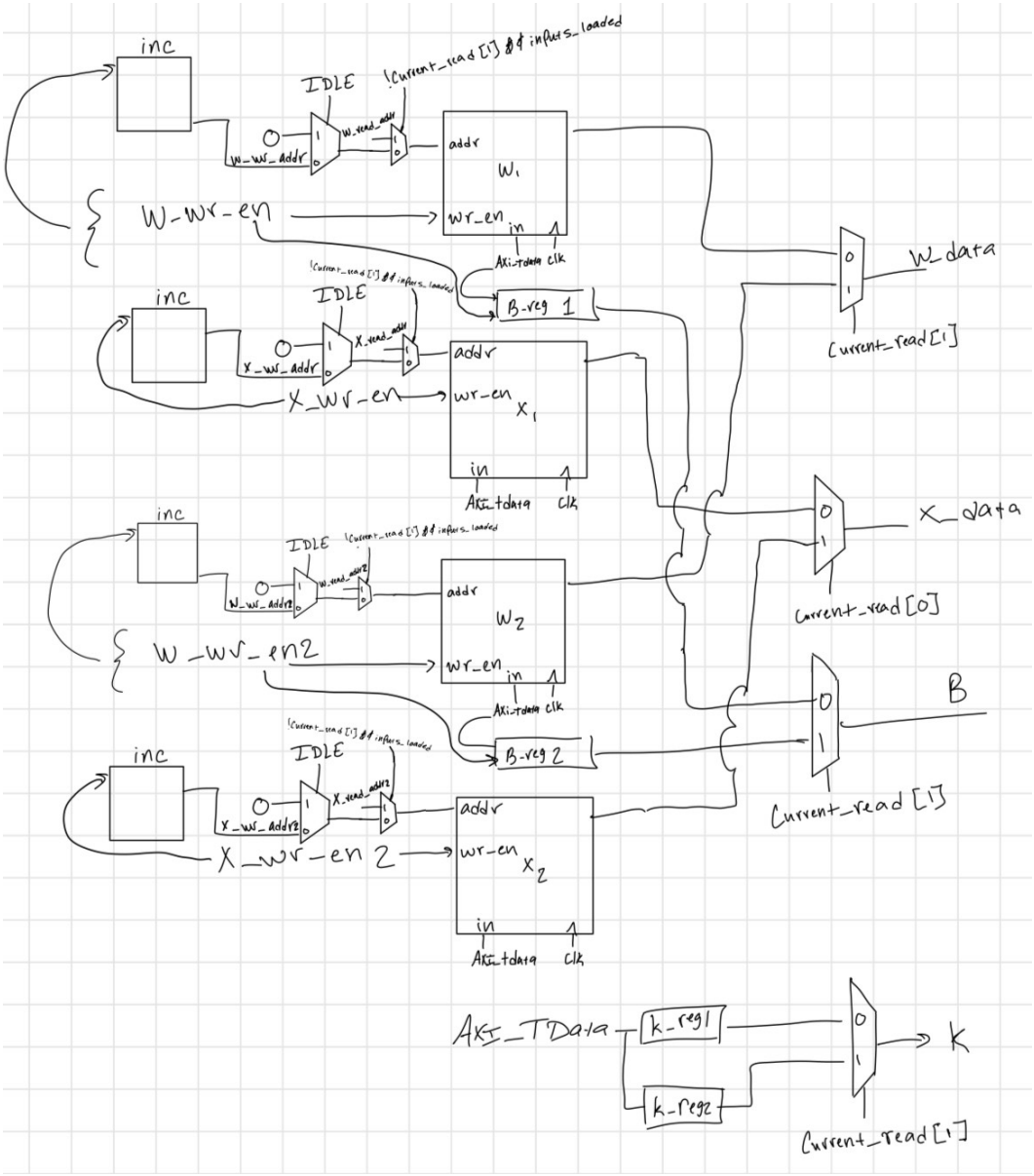

Figure 14: Datapath controlling 4 Memory Modules in John's design

Tyler' Design:

For my Part 5 optimization, I focused on increasing throughput by parallelizing inside the convolution computation, while trying to keep the control logic and memory behavior as close to the Part 4 design as possible. My approach went through several ideas before arriving at the final design.

My first attempt at optimizing the Part 4 design was to use parallel MACs to roughly double the computational throughput. The idea was to have two MAC units operating on two separate input sets stored in `input_mems`. The control logic would have:

- Read two different data values from one input memory module by creating the correct offset.
- Pass those two data values into two separate MACs, thereby performing two computations at once.
- Sequentially write those two computed values into the FIFO, one after the other.

Theoretically, this would allow data loading and computation to run in parallel and reduce the idle time between runs. However, this approach ran into major complications on the memory and buffering side. To make it work, I would have had to keep track of:

- The current inputs (all weights, K, bias, `new_w`, `old_w`, etc.) that are actively being used by the MAC.
- The correct next-read addresses for the memory, which needed to be offset by the number of computations previously performed.

Calculating the read address using the correct offsets added a lot of complexity and made the design harder to test and verify, especially when these offsets change with every new set of input data due to changes in K.

This led me to change perspective: if throughput is the main goal and we are allowed to spend more area and power, why not try to speed up the computation *inside* a single convolution window instead of across separate input sets?

My next idea was to compute an entire row of the kernel per cycle. I was thinking:

- For each kernel row i, use enough multipliers to compute all K products

  $$X[r+i, c+0 \ldots K-1] \cdot W[i, 0 \ldots K-1]$$

  in parallel.
- Then iterate over the kernel rows i and accumulate the row sums.

Initially, this suggested using up to `MAXK` multipliers, since the maximum kernel size is `MAXK` $\times$ `MAXK`. Even if $K <$ `MAXK`, the hardware would still be able to handle the largest possible case.

This introduced a new problem: how do I feed that many multipliers from memory without breaking correctness? The memory system would need to provide up to `MAXK` X values and `MAXK` W values per cycle, in the correct order, while ensuring the accumulator always sees the right products. Driving that many multipliers directly from a single memory structure would require multiple read ports, which Part 5 does not allow.

I then realized that if I could not give one memory multiple read ports, I could instead create multiple instances of the same memory and use each as a separate read port, since in Part 5 we do not care as much about area and power.

The final design I chose was to use three parallel multipliers (three was a simpler case I wanted to get working first, and I did not want to make the system irrationally large) and three input memories with the same data inside them to accelerate the computation for a single convolution window, while keeping the rest of the system (AXI-Stream I/O, FIFO, FSM structure) conceptually similar to Part 4.

When I say "three multipliers," I mean three lines of code in the top-level module that perform the multiplication using the `*` operator. In other words, I completely removed the MAC module we designed in Part 1 of this project. Thus, in my top-level module there is no instantiation of the MAC module. The reason for this change was that it gave us the ability to delay the write to the accumulator

register. Since the accumulator register accumulates on top of the previous value, we need to write the first multiplier's product first, then the second multiplier's product, and finally the third multiplier's product. We can only do all of this conveniently in the top-level module.

The main technique I used was to first replicate `input_mems` three times to obtain multiple read ports. I instantiate three identical `input_mems` modules: `Mem0`, `Mem1`, and `Mem2`. All three instances receive the same AXI-Stream input (`INPUT_TDATA`, `INPUT_TVALID`, `INPUT_TUSER`), store the same X and W data layout, and are controlled so that they are all ready (`tready_0`, `tready_1`, `tready_2`) before accepting more input. `Mem0` is the "main" instance of the memory; it outputs `inputs_loaded`, K, and B. `Mem1` and `Mem2` are used as extra read ports only, with dummy `inputs_loaded` wires just to satisfy the interface and remain compatible with the testbench (since we cannot have three instances driving `inputs_loaded`). By duplicating the memory instead of changing its interface, I created three independent read ports for both X and W, at the cost of making the design much larger than Part 4.

We then use three parallel multipliers on consecutive kernel columns. For a fixed output location (r, c) and kernel row i, the address generation logic computes:

- `Mem0` reads $X[r+i, c+j]$ and $W[i, j]$,
- `Mem1` reads $X[r+i, c+j+1]$ and $W[i, j+1]$,
- `Mem2` reads $X[r+i, c+j+2]$ and $W[i, j+2]$.

This means that in one cycle the design computes three products that all belong to the same window:

$$X[r+i, c+j] \cdot W[i, j], \quad X[r+i, c+j+1] \cdot W[i, j+1], \quad X[r+i, c+j+2] \cdot W[i, j+2].$$

Because the weight matrix columns increment by 3 every cycle, the hardware tries to compute three products every cycle. This works fine when K is a multiple of 3. The problem occurs when K is not a multiple of 3. The address generator still computes j, $j+1$, and $j+2$, and the multipliers still produce three products, even if some of those indices are outside the valid range. That would either read invalid outputs from the memory or pull unrelated data that should not be in this convolution window, causing the accumulator to sum incorrect values.

To avoid that issue, I created a 3-bit mask called `current_mask` that marks which of the three multipliers have valid data to output:

```
current_mask = 3'b000;
if (mac_valid) begin
    current_mask[0] = 1;
    if (j + 1 < K) current_mask[1] = 1;
    if (j + 2 < K) current_mask[2] = 1;
end
```

Here, `current_mask[0]` corresponds to the product from multiplier 1 ($X[r+i, c+j] \cdot W[i, j]$), `current_mask[1]` corresponds to multiplier 2 ($X[r+i, c+j+1] \cdot W[i, j+1]$), and `current_mask[2]` corresponds to multiplier 3 ($X[r+i, c+j+2] \cdot W[i, j+2]$). For each cycle in the **COMPUTE** state, the first product (j) is always valid as long as `mac_valid` is asserted, so `current_mask[0]` is set to 1. The second product ($j+1$) is valid only if $j+1 < K$; otherwise it remains 0. The third product ($j+2$) is treated similarly, with the condition $j+2 < K$.

This mask correctly handles all cases of K. For example:

- If $K = 5$ and $j = 0$, then $j+1 = 1$, $j+2 = 2$, and the mask is `3'b111` (all three multipliers valid).
- If $K = 5$ and $j = 3$, then $j+1 = 4 < 5$ so `current_mask[1] = 1`, but $j+2 = 5$ (not < 5) so `current_mask[2] = 0`, giving a mask of `3'b011` (only the first two multipliers valid).

The design then pipelines `current_mask` together with the other control signals so that it lines up correctly with the multiplier outputs:

```
mask_pipe     <= {mask_pipe[MAC_LAT-2:0], current_mask};
assign mask_delayed = mask_pipe[MAC_LAT-1];
```

Here, `MAC_LAT` represents the number of cycles of latency between when `current_mask` is generated and when the corresponding multiplier outputs are available (in our case, two cycles: one for the memory read and one for the multiply). To keep things aligned, I store the last `MAC_LAT` masks in a small shift register called `mask_pipe`. Each cycle,

```
mask_pipe <= {mask_pipe[MAC_LAT-2:0], current_mask};
```

shifts the previous entries up by one and inserts the new `current_mask` at the bottom. After exactly `MAC_LAT` cycles, the mask that matches the current multiplier outputs resides at `mask_pipe[MAC_LAT-1]`, which I call `mask_delayed`. This delayed mask is then used to zero out any invalid products before they are added into the accumulator.

The accumulator uses this delayed mask as follows:

```
p0 = mask_delayed[0] ? mult_0 : '0;
p1 = mask_delayed[1] ? mult_1 : '0;
p2 = mask_delayed[2] ? mult_2 : '0;
```

Even though the hardware physically computes three multiplications every cycle, the mask ensures that only products corresponding to valid kernel positions (i, j) with $0 \leq j < K$ actually contribute to the sum.

I then reworked the loop in the **COMPUTE** state, moving the kernel columns in steps of three: $j = 0, 3, 6, \ldots$ within each row i. When $j + 3 \geq K$, I reset j to 0 and increment i, moving down to the next row. When I reach the last row and the last block of columns, the FSM moves on to drain the pipeline and write the result, just like in Part 4. This loop restructuring allows me to use the 3-wide multiplication in a clean way, since I always try to perform three computations per cycle, and the mask logic handles the edge cases.

The path from memory read $\rightarrow$ multiplier output takes two cycles, so I define `MAC_LAT = 2` and create small shift-register pipelines for the control signals:

- `mac_valid_pipe` for `mac_valid`,
- `first_product_pipe` for `first_product`,
- `mask_pipe` for `current_mask`.

The delayed versions (`mac_valid_delayed`, `first_product_delayed`, and `mask_delayed`) are used by the accumulator so that the control information always lines up with the correct multiplier outputs.

Finally, we reach the single accumulator shared by the multipliers. The MAC datapath is split into two stages: (1) three parallel multipliers generate partial products, and (2) a single accumulator register adds them into the running sum for the current output pixel. On the first product group for a given (r, c) window, the accumulator is initialized to

$$B + p_0 + p_1 + p_2.$$

For all following product groups, it is updated as

$$\text{accumulator} \mathrel{+}= p_0 + p_1 + p_2,$$

where each p_i is either the multiplier result or zero, depending on the mask.

I also experimented with the idea of having multiple accumulators (one per multiplier), but that made the accumulation order complicated and led to incorrect results. Sticking with a single accumulator kept the behavior mathematically simple while still benefiting from the three parallel multipliers.

Unfortunately, my final implementation did not work. It encountered a timing issue that prevented it from passing the professor's testbench. The problem showed up as incorrect output values across all test cases, most likely due to a mistiming in the pipeline control. After extensive debugging, I was not able to resolve the issue with the control signals of the pipeline and the synchronization between `mac_valid_delayed`, `first_product_delayed`, and `mask_delayed` with the actual data flowing through the three parallel multipliers. With three memory banks, three multipliers, the pipeline depth, and multiple control signals, everything needed to align perfectly across concurrent data streams, which proved more complex than the single-stream case in Part 4.

I believe the root cause is in how the mask pipeline interacts with the accumulator logic. When `first_product_delayed` is asserted, the accumulator should load $B + p_0 + p_1 + p_2$, but if the timing of `mask_delayed` is off by even one cycle, invalid products could leak into this initial value. Given more time, I am confident this could have been resolved by either adjusting the `MAC_LAT` constant to account for an additional pipeline stage or by restructuring how the control signals are captured and shifted through the pipeline. The fact that the design compiled successfully, and that the errors were consistent rather than random, suggests the issue is a correctable timing problem rather than a fundamental flaw in the overall architectural approach.

Ultimately, while I was unable to produce working testbench and synthesis results for my optimization, the exercise provided valuable insight into the complexities of parallelizing datapath operations and the importance of maintaining precise synchronization between control and data pipelines in digital designs.

Question 2 (Repeat Questions 2-7 from Part 4)

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.

Answer:

For the very first inputs data after reset, i.e. when new_W is 1, the equation for number of cycles is:

$$K^2 + 1 + R * C + (R - K + 1) * (C - K + 1) * \left(K^2 + \text{MAC\_Latency} + 1\right)$$

From simulation of one of our convolutions $INW = 12$, $R = 9$, $C = 8$, $K = 3$, we can verify that loading data and computation of one convolution took 627 cycles. This is the breakdown of the cycles:

- Load weight matrix = $K \times K = 9$ cycles
- Load bias value = 1 cycle
- Load input matrix = $R \times C = 9 \times 8 = 72$ cycles

Remaining cycles:

$$627 - 72 - 9 - 1 = 545 \text{ cycles}$$

Output matrix size: $(R - K + 1) \times (C - K + 1)$

Since $(9 - 3 + 1) \times (8 - 3 + 1) = 7 \times 6$, i.e., 42 elements in output matrix.

$$\frac{545 \text{ cycles}}{42 \text{ elements}} = 13 \text{ cycles for computation of one element}$$

- 9 cycles for 9 MAC operations
- 3 cycles for draining the MAC because I used a 2-staged multiplier in the MAC

- 1 cycle for writing the MAC output to the FIFO

Since my optimized design loads the second set of data into the second memory bank while the first memory bank is used for computing the convoltuion for the first set of data, there is not wait for inputs stage. As soon as the first convolution computation is over, the second convolution computation starts. The equation for this case:

$$(R - K + 1) * (C - K + 1) * \left(K^2 + \text{MAC\_Latency} + 1\right)$$

This was verified via simulation for a test case where $INW = 12$, $R = 9$, $C = 8$, $K = 4$. The total number of cycles was 600.

Since $(9 - 43 + 1) \times (8 - 4 + 1) = 6 \times 5$, i.e., 30 elements in output matrix.

$$\frac{600 \text{ cycles}}{30 \text{ elements}} = 20 \text{ cycles for computation of one element}$$

- 16 cycles for 9 MAC operations
- 3 cycles for draining the MAC because I used a 2-staged multiplier in the MAC
- 1 cycle for writing the MAC output to the FIFO

From our equations and our verified data, we see that a huge contribution of cycles come from the compute phase of our system The biggest factor that decides the total the number of cycles is K, or in other words, MAXK. Because if MAXK is higher, then your output matrix gets smaller if K is equal to MAXK, and when that happens the number of computations reduces too. Obviously R and C of the input matrix matters too, because if we have a small input matrix, out output matrix gets smaller too. To obtain a very fast system, i.e. one with a small compute phase, we would have to choose the values of R,C, and K, such that R and C are small and K is big, but in moderation.

Question 3 (Compare to Part4)

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.

Answer:

Design	Time (ns)	Freq (MHz)	Area (μm^2)	Power (mW)	Critical Path
INW12_R9_C8_MAXK5	1.125	888.89	21299.151	16.303	c_reg[0] → X_matrix_mem2/data_out_reg[0]
INW18_R9_C8_MAXK5	1.25	800	32604.417	21.111	current_read_reg[0] → MAC/multinstance/mult_x_1/clk_r_REG19_S1
INW24_R16_C17_MAXK9	1.46	684.93	138356.97	86.739	i_reg[0] → X_matrix_mem2/data_out_reg[7]

Table 14: Synthesis results and critical path summary for Part 5 designs.

PART 4's Data:

Design	Time (ns)	Freq (MHz)	Power (mW)	Area (μm^2)	Critical Path
INW12_R9_C8_MAXK5	1.101	908.27	8.6444	12819.87	X_matrix_mem/data_out_reg[3] → MAC/pipeline_reg_reg[17]
INW18_R9_C8_MAXK5	1.357	736.92	10.3955	19296.70	X_matrix_mem/data_out_reg[11] → MAC/pipeline_reg_reg[33]
INW24_R16_C17_MAXK9	1.670	598.80	38.5089	74271.99	X_matrix_mem/data_out_reg[2] → MAC/pipeline_reg_reg[46]

Table 15: Synthesis Results and Critical Path Summary

Part 5's implementation shows a different critical path compared to Part 4, highlighting the impact of the input buffering optimization. The critical paths now are either:

- control logic (`c_reg`) → memory system, or
- memory output register → MAC accumulator.

For the INW = 12 and INW = 24 syntheses, the critical path starts at the column counter register `c_reg[0]` and passes through the address generation logic to the memory's output register. This corresponds to the sequence:

$$\text{FSM counter} \rightarrow \text{address calculation} \rightarrow \text{memory access} \rightarrow \text{output register}.$$

The presence of `c_reg` in the critical path shows that the control logic and address generation for the buffering system have become the longest part of the design.

For INW = 18, the path shifts to memory output → MAC accumulator, suggesting that the multiplication and accumulation datapath is the main section where the design spends time for this parameter set.

The key difference from Part 4 is that buffering adds control logic to the critical path. Part 4's critical path was purely datapath from memory → multiplier → register, but Part 5's input buffering requires:

- additional address offset calculations for buffer management,
- control logic to determine which buffer is being read or written, and
- coordination between the loading FSM and the computation FSM.

These extra control signals explain why Part 5 actually runs slower than Part 4 for INW = 12 (889 MHz in Part 5 vs. 908 MHz in Part 4) despite the throughput improvements. The buffering logic adds delay that extends the critical path beyond what the datapath alone requires. As INW increases to 18 and 24, this overhead becomes smaller relative to the growing multiplier delay, which is why Part 5 shows better frequency improvements at larger widths (800 MHz vs. 737 MHz for INW = 18, and 690 MHz vs. 599 MHz for INW = 24).

Question 4 (Compare to Part4)

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

Answer:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	cycle count	throughput (ops/sec)	cycle count	throughput (ops/sec)
0.10	x	x	x	x
0.50	5,450,368	1,467,790.80	45,131,265	151,764.31
1.00	5,439,290	1,470,780.19	45,372,805	150,956.39

Part 4's Data:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	cycle count	throughput (ops/sec)	cycle count	throughput (ops/sec)
0.10	13,330,898	552,790.724	73,422,762	81,555.41673
0.50	6,892,401	1,101,128.991	49,420,270	121,165.3427
1.00	5,881,452	1,252,955.351	48,671,745	128,300.8371

Unfortunately, John's optimized design did not work for the case when TVALID and TREADY probabilities were 0.10. In the interest of time, we did not try and fix our code for this test case nor try and find the reason behind this unique failure. However, this design does work for TVALID and TREADY probabilities of 0.5 and 1.0. A detailed analysis of how it works is provided in question 1 of this section.

Question 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 six scenarios evaluated in question 4, determine the delay in seconds (or ms, μs, 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.

Answer:

$$\text{Average Delay} = \frac{\text{Cycle Count} \times \text{Clock Period}}{10{,}000}$$

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	cycle count	delay	cycle count	delay
0.10	x	x	x	x
0.50	5,450,368	0.681 us	45,131,265	6.59 us
1.00	5,439,290	0.680 us	45,372,805	6.62 us

Part 4's Data:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	cycle count	delay	cycle count	delay
0.10	13,330,898	1.81 us	73,422,762	12.26 us
0.50	6,892,401	0.908 us	49,420,270	8.25 us
1.00	5,881,452	0.798 us	48,671,745	7.79 us

Question 6 (Compare to Part4)

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.

Answer:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	power (W)	energy (nJ)	power (W)	energy (nJ)
0.10	0.021111	x	0.0887390	x
0.50	0.021111	14.382840	0.0887390	572.024
1.00	0.021111	14.353608	0.0887390	575.063

Table 16: Power and energy for different TVALID/TREADY probabilities for the two designs.

Part 4's Data:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	power (W)	energy (nJ)	power (W)	energy (nJ)
0.10	0.0103955	18.805489	0.0385089	472.1807765
0.50	0.0103955	9.440765	0.0385089	317.8210793
1.00	0.0103955	8.298784	0.0385089	300.1453527

Question 7 (Compare to Part4)

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.

Answer:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	area (μm^2)	area-delay product	area (μm^2)	area-delay product
0.10	32604.417	x	138356.97	x
0.50	32604.417	0.0222	138356.97	0.912
1.00	32604.417	0.0222	138356.97	0.917

Table 17: Area and area–delay product for different TVALID/TREADY probabilities for the two designs.

Part 4's Data:

	Design 1 (INW=18, R=9, C=8, MAXK=5)		Design 2 (INW=24, R=16, C=17, MAXK=9)	
TVALID and TREADY prob	area (μm^2)	area-delay product	area (μm^2)	area-delay product
0.10	19296.70378	0.0349077923	74271.98711	0.9106894903
0.50	19296.70378	0.01752447165	74271.98711	0.6128804560
1.00	19296.70378	0.01504000584	74271.98711	0.5788893416

Question 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?

Answer:

Our Part 5 design is less efficient than our Part 4 design when comparing energy consumption and area–delay product metrics, despite achieving better throughput.

Energy

INW = 18:

- Part 4: 9.44 nJ (0.5) and 8.296 nJ (1.0)
- Part 5: 14.382840 nJ (0.5) and 14.353606 nJ (1.0)
- Part 5 uses roughly 50–70% *more* energy per convolution.

INW = 24:

- Part 4: 317.82 nJ (0.5) and 300.15 nJ (1.0)
- Part 5: 572.46 nJ (0.5) and 572.94 nJ (1.0)
- Part 5 uses roughly 80–90% *more* energy per convolution.

The energy efficiency is highly dependent on the TVALID/TREADY probabilities, but across all operating conditions (both 0.5 and 1.0), Part 5 consistently uses significantly more energy per convolution. The input buffering strategy increases instantaneous power draw substantially, and even though convolutions complete faster, the total energy expenditure increases due to the doubled memory resources and additional control logic that must remain active throughout operation.

Area–Delay Product

INW = 18:

- Part 4: 0.01752 (0.5) and 0.01540 (1.0)

- Part 5: 0.0222 (0.5) and 0.0222 (1.0)
- Part 5 has roughly 25–40% worse area–delay product.

INW = 24:

- Part 4: 0.6130 (0.5) and 0.5789 (1.0)
- Part 5: 0.912 (0.5) and 0.917 (1.0)
- Part 5 has roughly 40–60% worse area–delay product.

The area–delay product worsened significantly because:

- Memory usage is doubled due to input buffering.
- Clock frequency improved only slightly.
- Additional buffering control logic adds overhead.

The area increase outweighs the delay reduction, resulting in worse overall efficiency.

Throughput vs. Efficiency

Despite the worse efficiency metrics, Part 5 achieves noticeable throughput improvements:

INW = 18:

- Cycle count: $6{,}692{,}401 \rightarrow 5{,}450{,}368$ cycles ($\approx 18.6\%$ reduction).
- Throughput: $1{,}101{,}129$ convs/s $\rightarrow 1{,}467{,}791$ convs/s ($\approx 33.3\%$ increase).

INW = 24:

- Cycle count: $49{,}420{,}270 \rightarrow 45{,}131{,}265$ cycles ($\approx 8.7\%$ reduction).
- Throughput: $121{,}165$ convs/s $\rightarrow 151{,}764$ convs/s ($\approx 25.2\%$ increase).

The double input buffering lowers the idle time between convolution operations by overlapping data loading with computation time. While the previous design had to wait for all inputs to load before starting computation, Part 5 preloads the next test case during the current computation, reducing the overall cycle count.

Our Part 5 design is clearly less efficient than Part 4:

- **Energy efficiency:** It uses much more energy per convolution across all configurations and operating conditions. The doubled memory and buffering logic consume significantly more power than is offset by the modest speedup.
- **Area–delay product:** This metric is worse because the doubled memory area and additional control logic overshadow the slight clock improvements. In effect, we nearly doubled the area for less than a 35% improvement in throughput.

This behavior follows Part 5's design objective of maximizing speed even at the expense of area, power, and energy. We successfully achieved roughly 25–30% better throughput by sacrificing efficiency. The input buffering optimization is effective for throughput performance and can justify the resource cost in high-performance contexts, but Part 4 is the better choice for power- and area-limited systems where efficiency is the priority. Part 5 would be preferred in high-performance computing scenarios or real-time systems with strict latency requirements, while Part 4 is better suited for embedded systems, low-power devices, or cost-sensitive applications.

Question 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.)

Answer:

Tyler and John split up to try different optimization strategies for Part 5, with the goal of comparing approaches and selecting the most effective implementation.

John's Contribution: John implemented the input buffering optimization strategy. His approach involved double-buffering the input memories to eliminate idle time between convolutions by overlapping the loading of the next test case with the computation of the current one. His implementation worked partially and successfully passed the professor's testbench, achieving a 20–30% improvement in throughput. John and Tyler handled the synthesis, data collection, and analysis of John's buffer-based approach.

Tyler's Contribution: Tyler tried to implement a 3-way parallel multiplier architecture using duplicated memory banks to process three multiplications and one accumulation operation per cycle. The design involved instantiating three `input_mems` modules and connecting their read ports with appropriate control logic to handle cases where the kernel size was not divisible by three. Tyler's implementation encountered some pipeline timing issues. Given more time, Tyler believes this issue could have been resolved, as the hardest part was getting the delayed control signals to properly synchronize with the three parallel data streams.

Both partners worked on the final report, comparing their optimization approaches and analyzing the results they obtained. We evaluated the synthesis results, calculated efficiency metrics, and discussed why John's partial implementation still provided important insight into the throughput–efficiency trade-offs in Part 5. The experience of attempting different strategies independently helped us better understand the design space and the various techniques available for performance optimization.

7 Results

References

[1] *"MuJoCo."* MuJoCo Documentation. https://mujoco.readthedocs.io/en/stable/overview.html

[2] AI & Robot Vision Lab. *"Grasp-Anything: Large-Scale Grasp Dataset from Foundation Models."* ICRA 2024. https://airvlab.github.io/grasp-anything/

[3] GeeksforGeeks. *"How to Detect Shapes in Images in Python Using OpenCV?"* 3 Jan. 2023. https://www.geeksforgeeks.org/how-to-detect-shapes-in-images-in-python-using-opencv/

[4] *"MuJoCo Tutorial."* Google Colab, Google. https://colab.research.google.com/github/google-deepmind/mujoco/blob/main/python/tutorial.ipynb#scrollTo=YvyGCsgSCxHQ

[5] Google DeepMind. *"mujoco_menagerie/franka_emika_panda at main · google-deepmind/mujoco_menagerie."* GitHub. https://github.com/google-deepmind/mujoco_menagerie/tree/main/franka_emika_panda

[6] OpenAI. *"ChatGPT."* May 20, 2025. https://chat.openai.com/
We used ChatGPT's help to create some of the table and object geom descriptions for our XML files. We also used it for debugging.