Stony Brook University

Department of Electrical and Computer Engineering

Final Report

ESE 507: Advanced Digital Systems & Design

Author:
Michael Bove

Instructor:
Peter Milder

January 3, 2026

Part 1

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].")

Answer: Please refer to Table 1, Figure 1, and Figure 2 below for unpipelined MAC results.

Table 1: Unpipelined MAC Synthesis Results for INW=16 and OUTW=64

Period (ns)	**Freq** (MHz)	**Area** (μm^2)	**Power** (μW)	**Timing Constraint**	**Critical Path Location**
10.0	100.00	1726.61	209.87	MET	Input0 → Multiplier → Adder → out_reg (register for output)
7.5	133.33	1726.61	267.02	MET	
5.0	200.00	1759.32	400.03	MET	
4.5	222.22	1817.84	447.92	MET	
4.0	250.00	1959.89	511.99	MET	
3.5	285.71	1974.78	590.86	MET	
3.0	333.33	2007.50	702.98	MET	
2.5	400.00	2059.90	880.06	MET	
2.0	500.00	2041.28	1050.10	MET	
1.5	666.67	2105.66	1422.60	MET	
1.4	714.29	2167.63	1597.50	MET	
1.3477	742.00	2215.25	1655.00	MET	
1.3476	742.06	2195.30	1674.30	VIOLATED	

Figure 1: The figure below shows the relationship between Area (μm^2) and the clock frequency (MHz) for the unpipelined design where the timing constraint is MET.

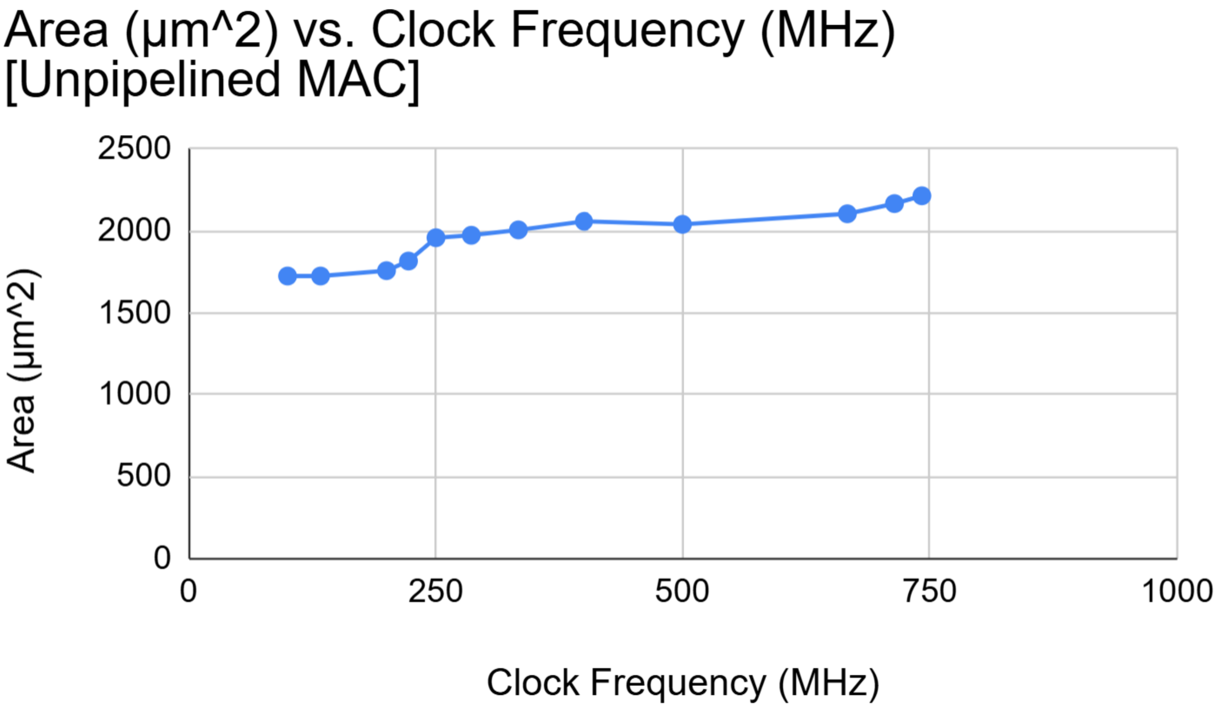
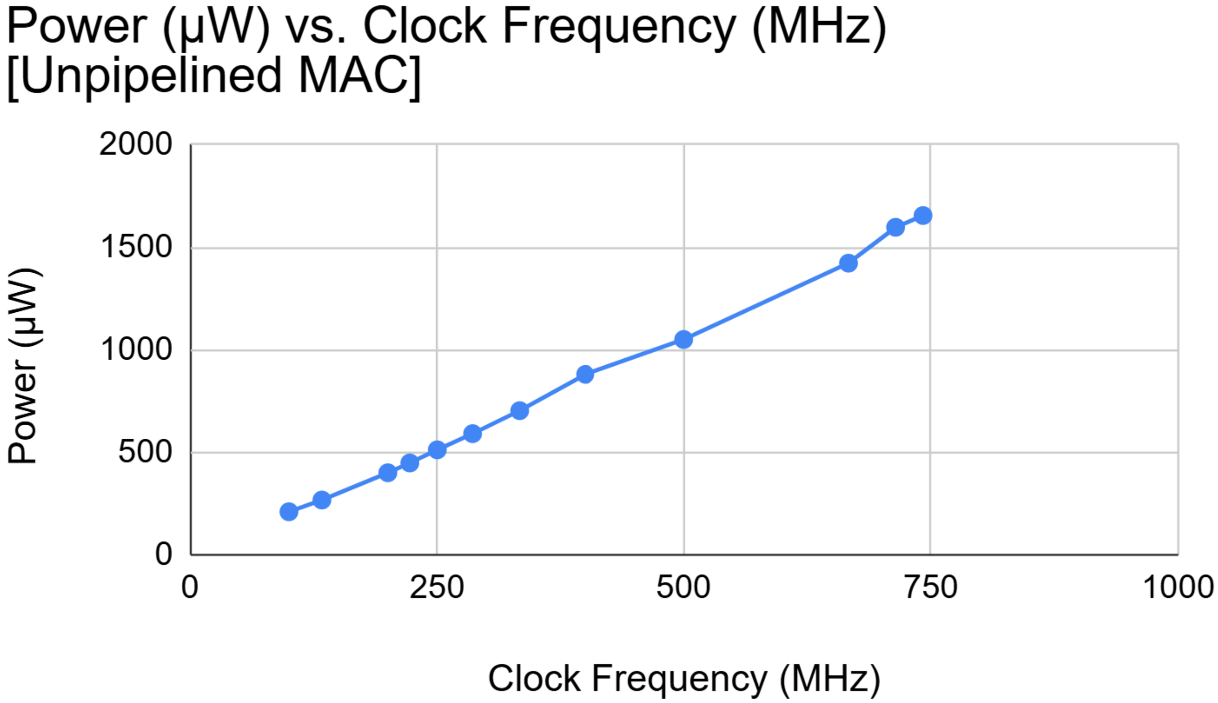


Figure 2: The figure below shows the relationship between Power (μW) and the clock frequency (MHz) for the unpipelined design where the timing constraint is MET.



Answer continued: As noted in **Table 1**, for **every** frequency in the synthesized unpipelined MAC, the **critical path** remained the **same**. In this case, it went from the **input** (input0 in our case, but it should be from either input0 or input1), through the **multiplier** that multiplies input0 and input1, through the **adder** that sums that result with the previous output of the output register (out_reg), into the **data (D) inpu**t of out_reg.

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.

Answer: Please refer to Table 2, Figure 3, and Figure 4 for pipelined MAC results.

Table 2: Pipelined MAC Synthesis Results for INW=16 and OUTW=64

Period (ns)	**Freq** (MHz)	**Area** (μm^2)	**Power** (μW)	**Timing Stat**	**Critical Path Location**
10	100.0	1887.80	230.72	MET	Output of the output register (out_reg), through the feedback path that sums with the output of the multiplier reg (mult_reg) back to the input of out_reg
7.5	133.3	1887.80	294.15	MET	
5	200.0	1930.63	435.89	MET	
4.5	222.2	1953.24	489.10	MET	
4	250.0	1991.81	551.01	MET	
3.5	285.7	2109.91	639.71	MET	
3	333.3	2135.98	759.65	MET	
2.5	400.0	2202.48	968.77	MET	Input0 through the multiplier of input0 and input1 into the input of the pipeline register (that separates the multiplier logic and the adder logic)
2	500.0	2220.83	1200.00	MET	
1.5	666.7	2254.62	1630.00	MET	
1.4	714.3	2267.65	1770.00	MET	
1.3	769.2	2275.63	1890.00	MET	
1.25	800.0	2294.78	2001.60	MET	
1.221	819.0	2321.38	2067.20	MET	
1.220	819.7	2323.24	2083.10	VIOLATED	

Note: The divider between 2.5 ns and 3 ns indicates a change in the critical path

Figure 3: The figure below shows the relationship between Area (μm^2) and the clock frequency (MHz) for the pipelined design where the timing constraint is MET.

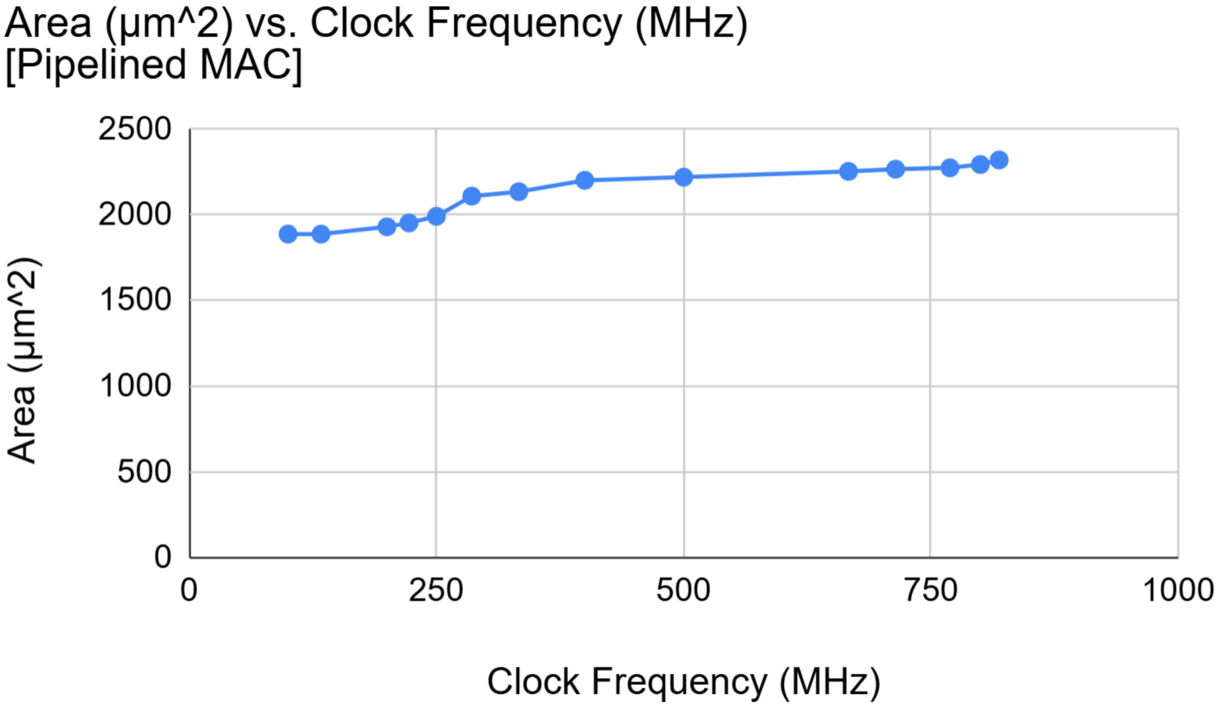
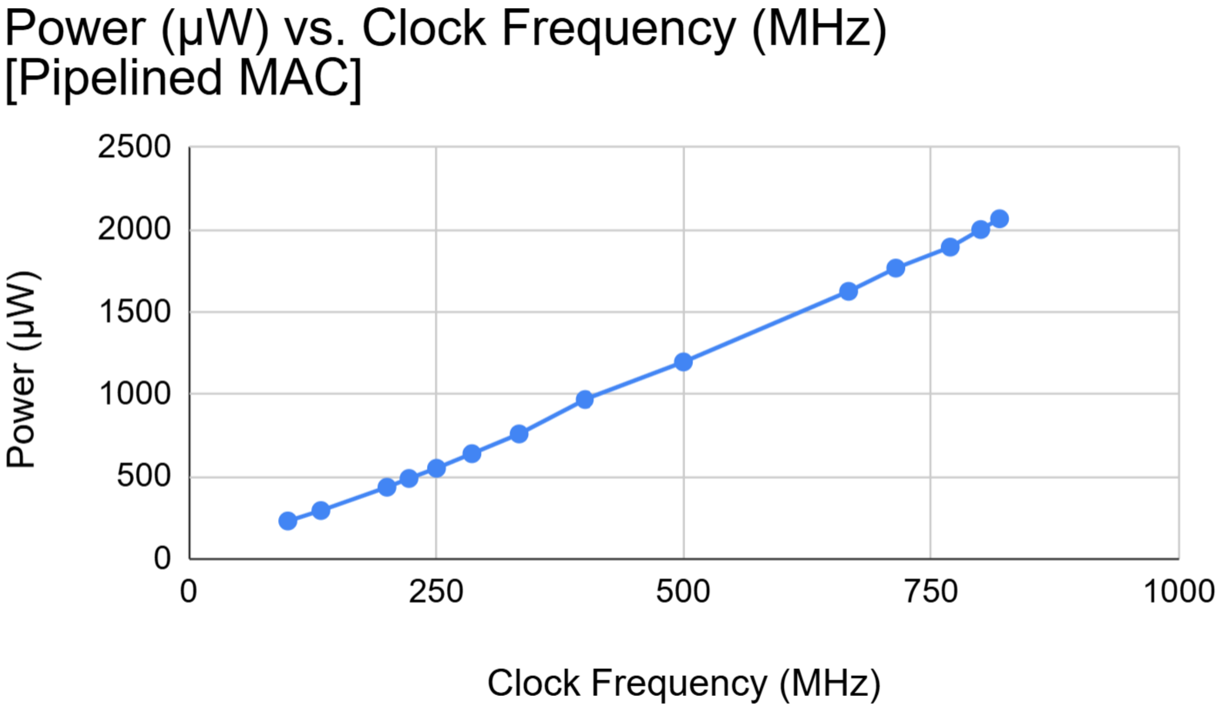


Figure 4: The figure below shows the relationship between Power (μW) and the clock frequency (MHz) for the pipelined design where the timing constraint is MET.



Answer continued: As shown in Table 2, for synthesized designs with a frequency **less than (and including) 333.3 MHz**, the **critical path** was from the feedback path from the **adder**, going into the **input** of the **adder** to sum its previously accumulated value with the newly computed multiplication.

For synthesizable designs with a frequency **greater than and including 400 MHz**, the **critical path** was from the **input** of the MAC module (specifically input0 in the synthesis report, but it could be from input1 as well) going through the **multiplier** into the input of the **pipeline multiplier register** (called mult_reg).

Pipelining the MAC **did** make the module faster. This is reflected in the maximum synthesizable clock frequency. Instead of capping off at 742 MHz (period of 1.3477 ns), in the pipelined design, we were able to achieve a maximum clock frequency of 819.7 MHz (period of 1.220 ns). Moreover, this is due to the **decreased critical path**. Instead of needing time to go from the input of the MAC module through the multiplier **AND** adder to the input of out_reg (the output register), it only has to go from the input of the MAC module to the input of mult_reg (the pipeline register between the multiplier and adder). The next cycle, it goes through the adder into the input of out_reg, which separates the design into two smaller combinational sections instead of one large section.

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:
Note: The number of clock cycles is calculated based on 49 previous clock edges, assuming the first input data arrives before the first positive clock edge, and the 50th input arrives right before the 50th clock cycle.
Power: 2067.20 μW
Number of cycles: 49 (49 previous inputs) + 2 (pipeline register + output register) = 51 clock cycles
Period: $\frac{1.221\,ns}{1\,cycle}$
Time: 51 cycles x $\frac{1.221\,ns}{1\,cycle}$ = 62.271 ns
Energy:
Energy = Power * Time
Energy = 2067.20 μW * 62.271 ns
Energy Consumed = 128.73 pJ

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: Yes, the energy would change if I resynthesized the design targeting different clock frequencies.

Energy is the product of power and time. Power is proportional to frequency (i.e. higher frequency = more power). This is partially due to the increased dynamic power, due to more switching. Not only that, but the capacitance also increases as area increases, thus contributing to a higher power (power is also proportional to capacitance and voltage).

However, as frequency increases, the period decreases, so it takes less time to perform the same calculation.

Overall, time and frequency are reciprocals of each other, so they cancel out, which leaves the capacitance. Thus, it is generally true that energy consumption increases as frequency increases, due to the power increasing at a higher rate compared to the rate at which the time decreases. This is because the **capacitance increases** as area increases

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:
Note:
Latency = Time from data arriving on input to data arriving on output.
Throughput = Number of MACs per second.

Calcuations:
Latency (unpipelined) = Clock period = **1.3477 ns**
Latency (pipelined) = Clock period x 2 = 1.221 ns x 2 = **2.442 ns**

Throughput (unpipelined) = $\frac{1}{Period} = \frac{1}{1.3477\,ns}$ = **742,004,897 MACs/sec**
Throughput (pipelined) = $\frac{1}{Period} = \frac{1}{1.221\,ns}$ = **819,000,819 MACs/sec**

Table 3: Comparison of Unpipelined vs. Pipelined Design Results

Metric	Unpipelined	Pipelined
Period (ns)	1.3477	1.221
Clock Frequency (MHz)	742	819
Power (μW)	1655.0	2067.2
Area (μm^2)	2215.25	2321.38
Latency (ns)	1.3477	2.442
Throughput (MACs/sec)	742,004,897	819,000,819

The calculated latency is the time it takes from receiving the input to the MAC to the time it is ready on the output. In this case, it is the time it is ready on the input to the output register. It is also the same time it will be ready on the output of the output register, assuming a propagation delay of 0 ns (which is the value for an ideal register). Since the unpipelined version only has one register (the output register), the latency is just the time it takes to go through the MAC in one clock cycle. In the pipelined design, it has to also go through the pipeline register, which adds 1 register delay (1 clock cycle) to the system. Thus, this is why we calculated latency the way we did.

The calculated throughput is essentially how many MACs we can perform per second (assuming a full pipeline). This is basically just the clock frequency (or the reciprocal of the period).

The reason someone would want to use the unpipelined version is if they care about how long it takes to process the data from receiving it on the input to the MAC (i.e. if you care from going from the input to the output in less time); this is essentially just having a lower latency. However, if someone is concerned with processing as much data in as little time as possible (by overlapping computations), higher throughput would be more suitable, where the pipelined version would provide more use.

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: I would expect deeper pipelining in the multiplier **would help** reach **higher clock frequencies**. In the recently pipelined design, the **critical path** is from the **multiplier** into the input of mult_reg. If we pipelined the multiplier, we'd be able to increase the clock frequency because we'd be **shortening the critical path**.

Pipelining the adder would **technically** be **possible**, but it would change the intended functionality of our system. This is because it would **break the feedback path** going back into out_reg. We would have to change our datapath/control logic to account for this change. Because of this, pipelining the adder would **not** be a good idea.

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: The clock frequency and critical path location do **NOT** change as OUTW changes (the period increased by 0.01 using OUTW = 48, but I assume that is just because of Design Compiler). Howeve, clock frequency and critical path location **DO** change as INW changes. This is because the multiplier has a way larger propagation delay compared to the adder. Even for large OUTW values when INW is 12, the critical path is **STILL** from the input, through the multiplier, into mult_reg. because INW is constant in the first part, and the critical path's input and output is defined by INW, the maximum clock frequency stays the same.

Consequently, as INW changes when OUTW stays the same (48), the **critical path** is still defined by the **multiplier**. Because of this, as **INW increases**, the **maximum clock frequency decreases**.

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: The maximum value we can achieve from a multiplication is 256 (that is, -16 * -16 = 256, since INW = 5). The minimum value we can achieve is -240.

Moreover, the maximum value the MAC can safely add to is 32,767. The minimum value is -32,768.

The maximum value we can initialize the MAC to is 15, and the minimum is -16.

Using these numbers, the MAC is limited by the most positive number, as it can be achieved in less cycles compared to the negative one; 15 + 256 x (number of MACs) approaches 32,767 faster than -16 - 240 x (number of MACs) approaches -32,768. By figuring this number out, it is **guaranteed** the accumulator cannot overflow. To figure this number out, let's compute it.

Calculation:
$15 + 256c \leq 32,767$
$256c \leq 32,752$
$c \leq 127.9375$

In the above calculation, **c** represents the **number of MACs**, which can be safely. Because the number of MACs is less than or equal to 127.9375, the maximum number of MACs we can safely perform is **127**; 128 MACs could lead to overflow.

Part 2

Note: Since the original submission, the conditions under which IN_AXIS_TREADY gets asserted were slightly modified. Instead of basing it purely on the capacity, it also uses the rd_en to calculate if the ready signal should be high or not (if capacity is 0 and rd_en is 0, that is the only case which we are not ready, otherwise read data).

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 first part of the FIFO is the input logic. IN_AXIS_TREADY is the signal that is generated based on the availability of the FIFO. Essentially, if the capacity is 0 or if the rd_en (read enable) signal is 1, then we are ready to receive new data. This is because if the capacity signal is 0 and rd_en signal is 0, we can't store any new data. If capacity is 0 and rd_en is 1, then we can store new data, because the old one is being written out. In that case, IN_AXIS_TREADY is a 1.

To account for the head logic, we reset it to 0 when the reset is asserted. Otherwise, if wr_en is asserted (write enable), we store at the current wr_addr, and increment the wr_addr counter. The wr_en is asserted if we have room in the FIFO and if we have valid input (using IN_AXIS_TVALID Axi-Stream input and IN_AXIS_TREADY computed logic in the FIFO). It will also roll over to 0 if the previous value was equal to DEPTH - 1, which rolls over to 0 for all values of the DEPTH of the FIFO. This check is for DEPTH - 1, because the values we're storing at are from 0 to DEPTH - 1.

The tail logic was a little more complicated. First, rd_en checks if OUT_AXIS_TREADY (the AXI-Stream signal that's an input to our FIFO) and OUT_AXIS_TVALID (the output we generated based on if the FIFO isn't empty) are both asserted (both 1). If these are true, we can read out data (denoted by rd_en being 1). Similarly to the head logic, if the tail equals DEPTH - 1, we set tail to 0, otherwise it just increments. In this case, however, it is slightly more complicated. If rd_en is 0, tail is correct pointing to the right address (rd_addr = tail). Otherwise, if it is asserted, we need to set the rd_addr to the address after the tail is pointing to due to the timing of our system. For example, if rd_en is 1, the read address points to tail + 1 (or 0 if tail == DEPTH - 1), because we need to prepare the data for the next positive clock edge (compared to the tail, which points to the old data, since it also needs to increment on the positive clock edge).

Lastly, we adjust the capacity as necessary. If rd_en is asserted, and wr_en is not, then increment the capacity (both use the AXI-Stream interface, as mentioned before to assert these signals). If rd_en is not asserted and wr_en is, then decrease the capacity . If they are both asserted or both not asserted, then the capacity doesn't change.

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.)

Table 4: Output FIFO Synthesis Results for OUTW = 24, DEPTH = 19

OUTW (bits)	**DEPTH** (bits)	**Period** (ns)	**Freq** (MHz)	**Power** (μW)	**Area** (μm^2)	**Critical Path**
24	19	0.9	1111.11	2440.2	3697.40	capacity_reg → OUT_AXIS_TVALID → rd_en → tail → rd_addr → input to data_out_reg

Answer: The critical path starts at the output of the **capacity** register. Based on the value of capacity, **OUT_AXIS_TVALID** checks if capacity equals the FIFO's depth. That signal is used to determine if **rd_en** should be enabled for the memory. If this signal and OUT_AXIS_TREADY are set, rd_en is enabled. Then, we check what the **tail** is. If it equals DEPTH - 1 of the FIFO, the read address, **rd_addr**, is 0, otherwise it is tail + 1 (calculated elsewhere). Lastly, this value is used as the i**nput to the data_out register** (the input to the memory's register), which is responsible for reading out the data based on the address and enable signal).

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 see in area, power, and frequency. In other words, explain how these three sets of parameters affect the area, power, and frequency, and explain why.

Answer:

Table 5: Output FIFO Synthesis Results for OUTW =12, 24 and DEPTH = 19, 38

OUTW (bits)	**Depth** (bits)	**Period** (ns)	**Freq** (MHz)	**Power** (μW)	**Area** (μm^2)
12	19	0.87	1149.43	1385.9	1975.58
24	38	0.94	1063.83	5132.2	6831.15

We can compare data in Table 4 to Table 5 data. When we keep DEPTH constant and **decrease OUTW**, we achieve a [slightly] **higher frequency**. **The power is significantly reduced, and the area is greatly reduced as well**. This makes sense. The reason this happens is because the **amount of space** we have in our FIFO **decreases** to about 50% of its original storage, since OUTW is 50% smaller. By doing so, we need **less area** (almost 50% less). Moreover, because of the way memory works, we can r**ead out data quicker** (because it has less distance to look up the address and can come out quicker); it also uses **less power** because there is **less area**, which also implies a **lower capacitance (again less power)**. Similarly, when we **double the DEPTH** (keeping OUTW the same), we essentially double the size of the memory (keeping OUTW the same as the original value of 24). In doing so, the period increases (frequency decreases) because it takes **more time to read out the data (longer distances)**. Moreover, the area increases (not quite double the original amount), and the power also increases, as the capacitance and area increase, due to the increased storage.

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: The **datapath** for this design is fairly simple. Firstly, there are two memories: one for holding the X matrix and one for holding the W matrix. Both memories use synchronous reads and writes, in addition to single address ports (for both reading and writing); both memories have a data in and a data out port. There is also a basic counter to reset and increment a counter for the address to write X to, in addition to a multiplexer controlling the address port (depending on if we have loaded all the inputs or not). Likewise, the same logic exists for the W memory; there is a counter for the write addresses, and there is the same logic for controlling the addresses of the W matrix memory. There are also registers with load enables to store the B value and K value. Lastly, to know when we are done loading the W matrix, there is combinational logic to compute the last index to put for W based on K (K*K-1). This is only important information when in the LOAD_W_MATRIX state below, and it is calculated after loading the K value into a register. Moreover, this is done combinationally, because the critical path does not include the multiplier; if it did, this path would be fed into a register to break up the combinational logic.

Both memories' inputs come from AXIS_TDATA. The counter registers use AXIS_TDATA as their input, the K register uses TUSER_K (where TUSER_K = AXIS_TUSER[$clog2(MAXK+1):1]), new_W uses AXIS_TUSER[0], the B register uses AXIS_TDATA as its input, and the register calculating the last index for W comes directly from the K register.

The **FSM** for the input_mems module has five states:
1. WAIT_FOR_READY
2. LOAD_W_MATRIX
3. LOAD_B_VAL
4. LOAD_X_MATRIX
5. LOAD_DONE

Note: all control signals are 0, except for AXIS_TREADY (default is 1), unless otherwise noted.

1. At reset, the system goes into the WAIT_FOR_READY state. In this state, we wait for AXIS_TVALID to go high to begin loading data. If it is not high, we stay in this state. Once it is high, we can change state to LOAD_W_MATRIX or LOAD_X_MATRIX. To determine which state to transition to (and which control signals to assert), we check the least significant bit (LSB) of AXIS_TUSER to see the new_W signal.

If the LSB is 1, we assert the ld_k signal, which is a control signal for a register to keep track of K, which gets loaded on the first clock cycle. Additionally, W_wr_en and incr_w also get asserted. This is because we want to write W[0][0] into the W register, and prepare the counter to increment to the next address on the next clock cycle (the counter increments at the next clock edge, which is when we change states). Moreover, the next state is LOAD_W_MATRIX in this case, because we need to load the entirety of the W matrix (and the B value).

If the LSB is 0, we assert the X write enable and prepare to increment X on the clock edge (just like for W, and for the same reason too). In this case, since we are not loading a new W, our next state will be LOAD_X_MATRIX.

2. Assuming we need to load the W matrix (new_W is true), we transition to LOAD_W_MATRIX. As long as AXIS_TVALID is asserted (1), we enable the same control signals as the previous state (W_wr_en and incr_w). If it is not, we don't do anything and stay in the state. Once the W write address equals the last index and AXIS_TVALID is asserted (meaning a write will occur), we change states to LOAD_B_VAL.

3. Similarly to LOAD_W_MATRIX, the LOAD_B_VAL state only occurs when new_W was previously asserted (1). In this state, if AXIS_TVALID is 1, we enable the ld_B signal and at the next positive clock edge (when the B register loads), we change state to LOAD_X_MATRIX.

4. Once we transition from either the WAIT_FOR_READY state or the LOAD_B_VAL state, we transition to the LOAD_X_MATRIX state. Similarly to the LOAD_W_MATRIX state, we have the same control inputs; in this case, we have X_wr_en and incr_x enabled (both logic 1) assuming AXIS_TVALID is 1 (otherwise, do nothing and stay in this state). We keep these signals asserted until the X counter reaches the maximum value for the X address AXIS_TVALID is asserted (logic 1); this was computed using R * C - 1, where R and C are given as parameters at compile time, where the computation occurs at compile time. Once it reaches the maximum value and AXIS_TVALID is 1, we transition to the done state.

5. In the LOAD_DONE state, we have finished loading the inputs. Thus, we set AXIS_TREADY to 0, inputs_loaded to 1, and we set the reset to clear the x address and w address counter registers (because they are now being externally driven from our system). Using the AXIS_TREADY signal as the select signal to the address multiplexers, we will use that to read the external addresses.

When we are in the LOAD_DONE state and we receive the compute_finished signal, we transition back to the WAIT_FOR_READY, otherwise we stay in the state until the computation is finished.

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 it adds extra cycles of delay between steps, the system will be less efficient.

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

   Simulate this design in 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 of inputs where new_W==0, and record the number of cycles and the value of K.

   Hint: you can view the amount of simulated time that passes in the waveform and then divide it by 10ns to get the number of simulated clock cycles. In your report, for each test give this cycle count and the value of K (for both new_W of 0 and 1).

   In the report, quantify how efficient your system is with respect to the number of clock cycles by computing the following ratios:
   o When new_W==1, use efficiency = (K*K+1+R*C)/cycles
   o When new_W==0, use efficiency = R*C/cycles

   In these metrics, 1.0 is perfectly efficient—a good implementation will be close to this. Report both metrics and the data you collected.

   If your efficiency number is not close to 1, where could your logic be improved?

Answer:
1. new_W = 1:
Efficiency = (K*K+1+R*C)/cycles
R = 15, C = 13

Example 1: K = 2

Time elapsed: 2005 ns - 5 ns
Cycles elapsed: (2000 ns) / (10 ns / cycle) = 200 cycles
Efficiency = (2*2+1+15*13)/200 = 200/200 = **1.0**

Example 2: K = 4
Time elapsed: 14025 ns - 11905 ns = 2120 ns
Cycles elapsed: (2120 ns) / (10 ns / cycle) = 212 cycles
Efficiency = (4*4+1+15*13)/212 = 212/212 = **1.0**

Example 3: K = 5
Time elapsed: 491905 ns - 489695 ns = 2210 ns
Cycles elapsed: (2210 ns) / (10 ns / cycle) = 221 cycles
Efficiency = (5*5+1+15*13)/221 = 221/21 = **1.0**

2. new_W = 0
Efficiency = R*C/cycles
R = 15, C = 13

Example 1:
Time elapsed: 496065 ns - 494115 ns = 1950 ns
Cycles elapsed: (1950 ns) / (10 ns / cycle) = 195 cycles
Efficiency = (15*13)/195 = 195/195 = **1.0**

Example 2:
Time elapsed: 500225 ns - 498275 ns = 1950 ns
Cycles elapsed: (1950 ns) / (10 ns / cycle) = 195 cycles
Efficiency = (15*13)/195 = 195/195 = **1.0**

Example 3:
Time elapsed: 504385 ns - 502435 ns = 1950 ns
Cycles elapsed: (1950 ns) / (10 ns / cycle) = 195 cycles
Efficiency = (15*13)/195 = 195/195 = **1.0**

As can be seen by the results, I have a 1.0 efficiency for both new_W == 1 and new_W == 0. Thus, my logic could not be improved, as this is perfectly efficient.

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: Because my system has perfect efficiency, the equation for the cycle count is identical to the ideal cycle count (the numerator of the previous efficiency equations). Thus, for **new_W == 1**, the equation for my system's cycle count is **K*K+1+R*C**. This is because the cycle count is equal to the size of the K matrix, the X matrix, plus one cycle for the B value; this matches the ideal cycle count. Moreover, for **new_W == 0**, my system's cycle count is **R*C** (the size of the X matrix we must load), again matching the ideal amount.

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:

Table 6: Synthesis results for the maximum frequency of input_mems design using different parameters

INW	R	C	MAXK	Period (ns)	Clock Freq. (MHz)	Power (μW)	Area (μm^2)
24	9	8	4	0.713	1402.5	17470	15933.7
10	15	13	7	0.8	1250.0	19962	18571.9

Critical Path Location and Explanation:
The critical path for both designs was the same (it does not change): from the state register to the input of the data_out register for the X memory. The traced critical path started from the FSM state register. Once the state register transitions either into the LOAD_DONE or WAIT_FOR_READY the control input into the multiplexer feeding into the memory changes. It either selects the write address (the address generated from the counter) or the read address (external address taken as input. Once this multiplexer select signal changes, the data (either the read address or write address) propagates through the multiplexer into the address port of the memory. The reason this is the critical path in **BOTH** is because it needs time to propagate from the control logic through the multiplexer, and into the port of the memory, which in this case, takes the most time for both instances. The address bits (for the input to the address) is ceil[log_2(R*C)] and the control logic is fairly large. Since it is the bottleneck in the first case where R = 9 and C = 8, it likely would still be the bottleneck for larger R and C values (which is proven in the second part of the data), despite different INW and MAXK values.

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 **datapath** of the convolution system consists of the previous parts connected to a control unit (and some extra logic that makes the convolution unit successful). The input_mems module (Part 3 of the project) has an AXI-Stream interface, which connects externally. Additionally the output FIFO has an AXI-Stream interface to determine if it can send data out, as well as a separate AXI-Stream interface connecting to the control logic to determine if data is ready to be loaded. The control unit uses a signal to note if the data on the input to the FIFO (which comes from the pipelined MAC) is valid, and the FIFO exerts a ready signal to show if it is ready to read data or not. The output of the W matrix (read data) is connected to the first input of the MAC and the output of the X matrix (read data) is connected to the second input. The bias register (B) output is sent directly to the output of the input_mems module and connected directly to the init_val port of the pipelined MAC.

The registers and additional **datapath** components for the control logic to function correctly is slightly involved. First for the X matrix read address, there are multiple registers contributing to its value: the X base address register, the j register output (indicating which column we're on), and the i address register output (indicating which row we're on.) The X base address register increments when we're done performing a multiplication between every row and column in the current convolution window and W matrix. The j register increments on every clock cycle (until it reaches its maximum value of K - 1, which is when it resets) and the i address register increments by C every time the j register reaches its maximum value, which then resets when a separate additional register denoted as the i register equals K - 1 (because this one increments by 1 every time C == K - 1). The X address computer equals the X base address register plus the j register (column offset) plus the i address register (the address offset indicating the row we're on). The X base address register either increments by 1 or K depending on if another register called the c register (which keeps track of which column we're on in the X matrix) equals C - K.

As mentioned previously the c register keeps track of which column we're on in the X matrix, so it increments after we multiply an entire W matrix with the current window in the X matrix. This value resets when it equals C -K, and increments the r register. The r register keeps track of which row we're on, and when it equals R - K, it is at its maximum value.

Additionally, there is a counter which indicates how many clock cycles have occurred during and after the last input has been loaded into the MAC (i.e. the MAC counter equals 1 when the last data is loaded into the MAC).

This concludes the required understanding of the extra datapath logic required for our functional system.

The **control logic** for this system is fairly complex. Note: the default values for all control signals are 0 unless asserted. Initially, the system resets to a state called WAIT_FOR_LOAD, which does exactly what it sounds like it would: it waits until the inputs_mem module is fully loaded. In this state, assuming the inputs haven't been loaded yet (denoted by inputs_loaded == 0), the registers are held in a reset state, where the clears for the i, j, w, c, r, base address and MAC counters are all asserted. Otherwise (if inputs_loaded == 1), we set the init_acc signal to initialize the MAC to the bias value and set the increment signals for the w and j registers. We cannot start loading inputs on this clock cycle (or increment the memory address the clock cycle before), as once the inputs_loaded signal is asserted, only then does the memory use the read address we give it (so it is delayed by one clock cycle, so we cannot both initialize the MAC and load input data during the first cycle).

Once we are in the LOAD_MAC state, we have data from memory address 0 and the next address should be memory address 1 from both the W and X matrix, read on the next clock cycle. Thus, we set the input_valid signal to 1 in this state. Additionally, we always increment the W counter, assuming we still have valid data, denoted by if the i and j counters have reached their maximum value (K-1). Once they have, we increment the MAC and we de-assert the increment signals. Once the MAC counter == 2, we de-assert the input_valid signal (because the last valid input is read on the clock cycle the MAC counter goes to 2). Additionally, when the MAC counter is 1, we set the clear signals on our register (which will take effect again on the next clock edge, when the MAC increments to 2, and data is no longer valid). Once the MAC counter reaches PIPELINE_DEPTH, which is the depth of our MAC's pipeline in clock cycles Once it reaches this value, we stop incrementing the MAC. If the IN_AXIS_TREADY signal is asserted (FIFO is ready to receive) data, we set the increment signals on the j and w registers to prepare the memory address of the X and W registers 2 clock cycles from now. Additionally, since the FIFO can receieve data, we switch states (assuming the MAC counter == PIPELINE_DEPTH and IN_AXIS_TREADY is asserted) to the WRITE_DATA state.

In the WRITE_DATA, we already know the FIFO is free and we have our data loaded, so all we need to do is set IN_AXIS_TVALID to 1, implying we are writing data. Now, since the MAC is known to be free (we just wrote the old data out), we can initialize it because we previously cleared the memory address to 0 in the previous state, and the data is ready by now; additionally, we can load the inputs into the MAC, as the data is also prepared for X and W. We previously set it to increment to 1, but the data coming from the memory address for W and X won't come out until the next clock cycle (meaning we can intiailize the MAC and read inputs at the same time without stalling). Moreover, because the j register can go up to only 1 (when K == 2), we have a check to see if j == K - 1 (or 1 in this case because K - 1 == 1). If it is, we can make sure to reset j and increment the i register (again, so we don't have to stall by waiting for this check in another state). We also set the increment signal for c or r here (i.e. we clear c and set the r counter if C == C - K, otherwise increment c by 1). Not only this, but we clear the MAC counter to prepare it for the next time we need to increment it. Lastly, if we are on the last row (R - K) and column (C - K) of the X matrix, we clear all our counters and de-assert all our control signals. We also transition back to the initial state to load data back into the input_mems module. Additionally, the compute_finished signal is asserted for one clock cycle. Note: by clearing the counters, despite a possible increment signal being high, the clear signal has precedence, so values will reset regardless. Otherwise, if we are not, we just increment the r or c counter, and prepare to load more data into the MAC.

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: For the best case scenario (INPUT_TVALID and OUTPUT_TREADY are always 1), here are the results (verified through testing in simulation):

new_W == 0

When new_W == 0, the time spent loading the input_mems module is R*C (which only accounts for the X matrix). Then, there is 1 clock cycle overhead to get the new data (because you can only read once inputs_loaded is asserted. On the next clock cycle, data is ready. The addresses are generated K*K times (because that is the size of the window). While the last address is being generated, the MAC counter is incrementing. It will increment 1 more time (to the depth of the pipeline: 2 in our case) before checking if IN _AXIS_TREADY == 1. If it does, we go to the WRITE_DATA state, where we write the data, initialize the MAC, and load the next set of inputs, assuming we aren't done with the entire convolution (indicated by the row register being R - K and the column register being C - K). On the last iteration, we also add two additional clock cycles, since we need to account for the transition back to the waiting for accepting inputs state. Thus, we go through (K*K + 1) cycles (C–K+1) * (R–K+1) times. So, the overall amount of cycles we go through is R * C + 1 + (K * K + 1) * (C - K + 1) * (R - K + 1) + 1 cycles when new_W == 0.

new_W == 1
Similarly, everything is the same for when new_W == 1, except we also need to account for the time taken to load the W matrix and B value. So, the formula for the amount of clock cycles for new_W == 1 is (K * K + 1) + (R * C + 1 + (K * K + 1) * (C - K + 1) * (R - K + 1) + 1). We can slightly simplify both formulas, but for clarity (understanding what each part does), I will leave it in the expanded form.

Limiting Factor assuming new_W == 1 (more computation in loading data):

Case 1: K is small (2), R and C are relatively small (3 and 3)
Cycle count for input_mems = (K * K + 1) + (R * C + 1) = (2 * 2 + 1) + (3 * 3 + 1) = **15 cycles**
Cycle count for computation = (K * K + 1) * (C - K + 1) * (R - K + 1) + 1 = (2 * 2 + 1) * (3 - 2 + 1) * (3 - 2 + 1) + 1 = **21 cycles**
Bottleneck: Computation

Case 2: K is small (2), R and C are relatively large (10 and 10)
Cycle count for input_mems = (K * K + 1) + (R * C + 1) = (2 * 2 + 1) + (10 * 10 + 1) = **106 cycles**
Cycle count for computation = (K * K + 1) * (C - K + 1) * (R - K + 1) + 1 = (2 * 2 + 1) * (10 - 2 + 1) * (10 - 2 + 1) + 1 = **406 cycles**
Bottleneck: Computation

Case 3: K is larger (10), R and C are similar size to K (11 and 11)
Cycle count for input_mems = (K * K + 1) + (R * C + 1) = (10 * 10 + 1) + (11 * 1 + 1) = **113 cycles**
Cycle count for computation = (K * K + 1) * (C - K + 1) * (R - K + 1) + 1 = (10 * 10 + 1) * (11 - 10 + 1) * (11 - 10 + 1) + 1 = **405 cycles**
Bottleneck: Computation

Case 4: K is larger (10), R and C are relatively larger compared to K (20 and 20)
Cycle count for input_mems = (K * K + 1) + (R * C + 1) = (10 * 10 + 1) + (20 * 20 + 1) = **502 cycles**
Cycle count for computation = (K * K + 1) * (C - K + 1) * (R - K + 1) + 1 = (10 * 10 + 1) * (20 - 10 + 1) * (20 - 10 + 1) + 1 = **12222 cycles**
Bottleneck: Computation

Regardless of the case, the **bottleneck** is **always** the **computation** part of the convolution. This is because when we perform the convolution, we need to read out the same memory address multiple times (from X). This causes extra memory reads compared to the amount of cycles it takes to load both the W and X matrices, as we read the same memory address multiple times, adding up to way more cycles because of this redundancy and because it happens sequentially. This idea can be realized when sliding the W matrix's kernal accross the X matrix (way more data reads compared to cycles spent loading the matrices one time). In addition, due to the fact that the convolution always requires reading more data than there are entries in the X matrix and because of the sequential nature of the hardware in this case, it makes sense this is **always** the **bottleneck**. Moreover, to verify this is the case, one can see all the test cases above, which proves this is true (the bottleneck is always the computation).

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:

Table 7: Convolution System Synthesis Results for Varying Parameters

INW	**R**	**C**	**MAXK**	**Period** (ns)	**Freq** (MHz)	**Area** (μm^2)	**Power** (μW)
12	9	8	5	1.05	952.38	12948.61	9310
18	9	8	5	1.35	740.74	19057.04	10728
24	16	17	9	1.52	657.89	75607.57	43686

Critical Path Location: The **critical path** for the **first two** designs of Table 7 are the same: output of the X memory's data out register into the MAC's input through the multiplier, into the multiplier's pipeline register (between the multiplier and adder). The critical path for the third entry in Table 7 is slightly different (but not really): it goes from the output of the W memory's data out register into the MAC's input through the multiplier, into the multiplier's pipeline register (between the multiplier and adder).

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

Answer:
Throughput $(\frac{ops}{sec}) = \frac{\#\,Convolutions\,(ops)}{Cycle\,count\,(cycles) * Period\,(ns)} * 10^9\,\frac{ns}{s}$

Design 1: INW=18, R=9, C =8, MAXK=5, minimum period = 1.35 ns
1. TVALID = TREADY = 0.10
Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{12{,}885{,}693\,cycles * 1.35\,ns} * 10^9\,\frac{ns}{s} =$ **574,855.18** $\frac{ops}{sec}$

2. TVALID = TREADY = 0.50
Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{5{,}937{,}036\,cycles * 1.35\,ns} * 10^9\,\frac{ns}{s} =$ **1,247,660.85** $\frac{ops}{sec}$

3. TVALID = TREADY = 1.00
Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{5{,}112{,}832\,cycles * 1.35\,ns} * 10^9\,\frac{ns}{s} =$ **1,448,787.56** $\frac{ops}{sec}$

Design 2: INW=24, R=16, C =17, MAXK=9, minimum period = 1.52 ns

1. TVALID = TREADY = 0.10

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{71{,}498{,}452\,cycles * 1.52\,ns} * 10^9\,\frac{ns}{s} = \mathbf{92{,}015.24}\,\frac{ops}{sec}$

2. TVALID = TREADY = 0.50

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{46{,}379{,}921\,cycles * 1.52\,ns} * 10^9\,\frac{ns}{s} = \mathbf{141{,}849.04}\,\frac{ops}{sec}$

3. TVALID = TREADY = 1.00

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{43{,}714{,}068\,cycles * 1.52\,ns} * 10^9\,\frac{ns}{s} = \mathbf{150{,}499.55}\,\frac{ops}{sec}$

Table 8: Convolution System Design Comparison (Cycle Count and Throughput)

	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	12,885,693	574,855.18	71,498,452	92,015.24
0.50	5,937,036	1,247,660.85	46,379,921	141,849.04
1.00	5,112,832	1,448,787.56	43,714,068	150,499.54

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 6 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:

Average delay (s) $= \frac{Cycle\,count\,(cycles) * Period\,(\frac{ns}{cycle})}{\#\,Convolutions\,(ops)}$

Design 1: INW=18, R=9, C=8, MAXK=5, minimum period = 1.35 ns

1. TVALID = TREADY = 0.10

Average delay (s) $= \frac{12{,}885{,}693\,cycles * 1.35\,\frac{ns}{cycle}}{10{,}000\,ops} * 10^{-9}\frac{s}{ns} = 1.739 \times 10^{-6}$ **s**

2. TVALID = TREADY = 0.50

Average delay (s) $= \frac{5{,}937{,}036\,cycles * 1.35\,\frac{ns}{cycle}}{10{,}000\,ops} * 10^{-9}\frac{s}{ns} = 801.50 \times 10^{-9}$ **s**

3. TVALID = TREADY = 1.00

Average delay (s) $= \frac{5{,}112{,}832\,cycles * 1.35\,\frac{ns}{cycle}}{10{,}000\,ops} * 10^{-9}\frac{s}{ns} = 690.23 \times 10^{-9}$ **s**

Design 2: INW=24, R=16, C =17, MAXK=9, minimum period = 1.52 ns

1. TVALID = TREADY = 0.10

Average delay (s) $= \frac{71{,}498{,}452\, cycles * 1.52\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 10.867 \times 10^{-6}$ **s**

2. TVALID = TREADY = 0.50

Average delay (s) $= \frac{46{,}379{,}921\, cycles * 1.52\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 7.050 \times 10^{-6}$ **s**

3. TVALID = TREADY = 1.00

Average delay (s) $= \frac{43{,}714{,}068\, cycles * 1.52\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 6.645 \times 10^{-6}$ **s**

Table 9: Convolution System Design Comparison (Average Delay)

	Design 1 (INW=18, R=9, C=8, MAXK=5)	**Design 2 (INW=24, R=16, C=17, MAXK=9)**
TVALID and TREADY prob	**Average Delay** (s)	**Average Delay** (s)
0.10	1.739×10^{-6}	10.867×10^{-6}
0.50	801.50×10^{-9}	7.050×10^{-6}
1.00	690.23×10^{-9}	6.645×10^{-6}

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:

Energy (J) = Power (W) * Time (s)

Design 1: INW=18, R=9, C=8, MAXK=5, Power = $10\,728 \times 10^{-6}$ W

1. TVALID = TREADY = 0.10

Energy (J) $= (10\,728 \times 10^{-6})$ W * 1.739×10^{-6} s $= 18.656 \times 10^{-9}$ **J**

2. TVALID = TREADY = 0.50

Energy (J) $= (10\,728 \times 10^{-6})$ W * 801.50×10^{-9} s $= 8.598 \times 10^{-9}$ **J**

3. TVALID = TREADY = 1.00

Energy (J) $= (10\,728 \times 10^{-6})$ W * 690.23×10^{-9} s $= 7.405 \times 10^{-9}$ **J**

Design 2: INW=24, R=16, C=17, MAXK=9, Power = $43\,686 \times 10^{-6}$ W
1. TVALID = TREADY = 0.10
Energy (J) = $(43\,686 \times 10^{-6})$ W * 10.867×10^{-6} s = 474.736×10^{-9} **J**

2. TVALID = TREADY = 0.50
Energy (J) = $(43\,686 \times 10^{-6})$ W * 7.050×10^{-6} s = 307.986×10^{-9} **J**

3. TVALID = TREADY = 1.00
Energy (J) = $(43\,686 \times 10^{-6})$ W * 6.645×10^{-6} s = 290.293×10^{-9} **J**

Table 10: Convolution System Design Comparison (Average Energy)

	Design 1 **(INW=18, R=9, C=8, MAXK=5)**	**Design 2** **(INW=24, R=16, C=17, MAXK=9)**
TVALID and TREADY prob	**Average Energy** (J)	**Average Energy** (J)
0.10	18.656×10^{-9}	474.736×10^{-9}
0.50	8.598×10^{-9}	307.986×10^{-9}
1.00	7.405×10^{-9}	290.293×10^{-9}

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:
Area-Delay Product (μm^2 * s) = Area (μm^2) * Time (s)

Design 1: INW=18, R=9, C=8, MAXK=5, Area = 19057.04 μm^2
1. TVALID = TREADY = 0.10
Area-Delay Product (μm^2*s) = 19057.04 μm^2 * 1.739×10^{-6} s = 33.14×10^{-3} **μm^2*s**

2. TVALID = TREADY = 0.50
Area-Delay Product (μm^2*s) = 19057.04 μm^2 * 801.50×10^{-9} s = 15.27×10^{-3} **μm^2*s**

3. TVALID = TREADY = 1.00
Area-Delay Product (μm^2*s) = 19057.04 μm^2 * 690.23×10^{-9} s = 13.15×10^{-3} **μm^2*s**

Design 2: INW=24, R=16, C=17, MAXK=9, Area = 75607.57 μm^2
1. TVALID = TREADY = 0.10
Area-Delay Product (μm^2*s) = 75607.57 μm^2 * 10.867×10^{-6} s = 821.63×10^{-3} **μm^2*s**

2. TVALID = TREADY = 0.50
Area-Delay Product (μm^2*s) = 75607.57 μm^2 * 7.050×10^{-6} s = 533.03×10^{-3} **μm^2*s**

3. TVALID = TREADY = 1.00
Area-Delay Product (μm^2*s) = 75607.57 μm^2 * 6.645×10^{-6} s = 502.41×10^{-3} **μm^2*s**

Table 11: Convolution System Design Comparison (Area-Delay Product)

	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-Delay Product** (μm$^2 \cdot s$)	**Area-Delay Product** (μm$^2 \cdot s$)
0.10	33.14×10^{-3}	821.63×10^{-3}
0.50	15.27×10^{-3}	533.03×10^{-3}
1.00	13.15×10^{-3}	502.41×10^{-3}

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: The way my project works is trying to parallelize the system as much as possible, handling redundant data only once. Because we're performing convolutions, this will lead to a huge improvement, as most of the data we use is used somewhere else.

The way this works is the following: during the initial load of data, we have two buffers: one, which holds the entirety of the W matrix (because we need to apply this weight filter across the X matrix) and one which partially holds the X matrix. The W matrix is easy, as we just load all the values once and put them on a register, which is an input to another register. The X matrix is harder to deal with, since we don't want to just pass the entirety of the data through the input_mems module, which we do for the W matrix. Instead, we use a line buffer. A line buffer is essentially a small FIFO with a shift register. The idea is to have the required data (i.e. if we need K items, we have those K items, and then the one that will no longer used gets shifted out, and one item we need to use later gets shifted in). This allows us to load the inputs to the multiply-add unit MUCH quicker (all of them are loaded in parallel at once for K^2 items. Not only do we parallel load these, but additionally, to load this X buffer, we can do this WHILE the X is loading the inputs (essentially pre-fetching the data before we do the convolution, because we do not need to wait for it).

Getting this parallel data is already a huge benefit from the previous design, as instead of wasting K^2 cycles, we can reading out the data in parallel to achieve higher throughput. Additionally, because we parallelized it, it adds more complexity in our datapath logic for arithmetic. Instead of having 1 MAC, we have up to $MAXK^2$ multipliers, followed by an adder tree, in which the amount decreases by a factor of 2 per stage. Additionally, padding was used to account for the extra unncessary units when K < MAXK. Synthesis can handle this (see it is unnecessary), and it saves design complexity.

Additionally, because the MAC was a bottleneck in the previous part, I decided to slightly pipeline it (making it 3 stages instead of 1) to slightly increase the clock frequency at the cost of one more clock cycle latency (or rather 2, because as mentioned in the Part 5 description, it is a good idea to synchronize the incoming data on a positive clock edge).

Moreover, to avoid complexity from the control logic, I decided it would be best to send the data once the FIFO had enough room to store the entire output matrix (space for up to (R-1)*(C-1) items). The FIFO is initialize to 2 times this value, which gives some extra overhead. I'm not sure if this part if helpful or not, but I decided to divide by 2, as that is indicated by a right shift (instead of division, which is expensive in hardware and time).

Additionally, the FIFO has a capacity signal, which I can check, giving me insight on how many computations we can perform.

Lastly, we need to discuss the control logic, although it is very minimal, as most computations are handled by the datapath. There are three states: one for loading, one for doing most of the calculations, and the last one, which just waits for the last instruction to start its pipeline. After PIPELINE_DEPTH - 1 iterations (where this is defined by 1 clock cycle for the original register, 4 cycles for our pipelined multiplier, and 1 more clock cycle for the pipeline register between the last adder that adds the bias to the number), we jump back to the beginning. As noted previously, we have 4 clock cycles for the multiplier, as using a 5-stage pipelined multiplier brought the clock frequency to 1 GHz. I originally did not think waiting this long was necessary, however, I received errors when changing it, so I had to add it back. It also makes sense because technically the FIFO could go over the limit during that computation, and by the time we check it in the next iteration (done when changing states into loading the data), we could send data that will not be able to get written. In going back to the start, we reset our system (just like we do when we start it), as outlined in Part 4, but with slightly different control signals here.

I believe my optimizations were very effective!

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:
Let's start with new_W = 1. The only difference between this and when it is 0 is we need to load the W matrix, bias, and W matrix. The formula is the following:
(K*K+1) + (R*C). Now, once that is loaded, we need the delay of my system assuming full throughput. The extra amount of cycles is (C-K+1)*(R-K+1) + PIPELINE_DEPTH -1 cycles (next power of 2 for MAXK^2 + 4 + 1).

So, the total delay for new_W = 1 is ((K*K+1) + (R*C)) + ((C-K+1)*(R-K+1) + closest power of 2 for MAXK^2 + 4 + 1). For new_W = 1, it is ((R*C)) + ((C-K+1)*(R-K+1) + closest power of 2 for MAXK^2 + 4 + 1), which again indicates similar results as new_W = 1.

It can be seen that when R and C are large, and it is very close to the K value, loading the memory is the bottleneck, as less parallel loads need to occur. Moreover, when doing the computation and R and C are large, and K is small, they are roughly even in terms of efficiency.

More could theoretically be done to have even more memory to offload the data into another memory before doing the computation, but it is unclear how much more efficient that would be (if at all), as I basically already have that: a line buffer. Thus, I met the goal of creating a system that is very efficient in my opinion.

Question:

3. For the Part 5 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:

Table 12: Optimized Convolution System Synthesis Results for Varying Parameters

INW	**R**	**C**	**MAXK**	**Period** (ns)	**Freq** (MHz)	**Area** (μm^2)	**Power** (μW)
12	9	8	5	1.00	1000.00	78124.73	672190
18	9	8	5	1.02	980.39	129884.87	678550
24	16	17	9	3.00*	333.33	728539.68	1037500

*Note: A valid result was not obtained for the third design case, as it took too long to synthesize (about an hour each time). 3.00 ns is used as an estimated placeholder for calculation purposes.

Critical Path Location: The critical path is thee capacity data coming out of the FIFO, getting right shifted by 1 (divide by 2) and then being checked by a comparator in the convolution system. This is true for the first two designs, for the last design, it is the W matrix output, which has K^2 elements. This makes sense it is the critical path, as it is basically a smaller memory.

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

Answer:

Throughput $(\frac{ops}{sec}) = \frac{\#\, Convolutions\,(ops)}{Cycle\, count\,(cycles) * Period\,(ns)} * 10^9\, \frac{ns}{s}$

Design 1: INW=18, R=9, C =8, MAXK=5, minimum period = 1.02 ns

1. TVALID = TREADY = 0.10

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{8{,}609{,}188\,cycles * 1.02\,ns} * 10^9\,\frac{ns}{s} = \mathbf{1{,}138{,}774.25}\,\frac{ops}{sec}$

2. TVALID = TREADY = 0.50

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{2{,}224{,}576\,cycles * 1.02\,ns} * 10^9\,\frac{ns}{s} = \mathbf{4{,}407{,}096.71}\,\frac{ops}{sec}$

3. TVALID = TREADY = 1.00

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{1{,}435{,}994\,cycles * 1.02\,ns} * 10^9\,\frac{ns}{s} = \mathbf{6{,}827{,}271.96}\,\frac{ops}{sec}$

Design 2: INW=24, R=16, C =17, MAXK=9, minimum period = 3.00 ns

1. TVALID = TREADY = 0.10

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{31{,}264{,}621\,cycles * 3.00\,ns} * 10^9\,\frac{ns}{s} = \mathbf{106{,}616.78}\,\frac{ops}{sec}$

2. TVALID = TREADY = 0.50

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{7{,}887{,}821\,cycles * 3.00\,ns} * 10^9\,\frac{ns}{s} = \mathbf{422{,}592.42}\,\frac{ops}{sec}$

3. TVALID = TREADY = 1.00

Throughput $(\frac{ops}{sec}) = \frac{10{,}000\,ops}{5{,}003{,}221\,cycles * 3.00\,ns} * 10^9\,\frac{ns}{s} = \mathbf{666{,}237.48}\,\frac{ops}{sec}$

Table 13: Convolution System Design Comparison (Cycle Count and Throughput)

	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	8,609,188	1,138,774.25	31,264,621	106,616.78
0.50	2,224,576	4,407,096.71	7,887,821	422,592.42
1.00	1,435,994	6,827,271.96	5,003,221	666,237.48

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 6 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:

Average delay (s) $= \frac{Cycle\,count\,(cycles) * Period\,(\frac{ns}{cycle})}{\#\,Convolutions\,(ops)}$

Design 1: INW=18, R=9, C=8, MAXK=5, minimum period = 1.02 ns

1. TVALID = TREADY = 0.10

$$\textbf{Average delay (s)} = \frac{8{,}609{,}188\, cycles * 1.02\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 878.14 \times 10^{-9}\ \textbf{s}$$

2. TVALID = TREADY = 0.50

$$\textbf{Average delay (s)} = \frac{2{,}224{,}576\, cycles * 1.02\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 226.91 \times 10^{-9}\ \textbf{s}$$

3. TVALID = TREADY = 1.00

$$\textbf{Average delay (s)} = \frac{1{,}435{,}994\, cycles * 1.02\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 146.47 \times 10^{-9}\ \textbf{s}$$

Design 2: INW=24, R=16, C =17, MAXK=9, minimum period = 3.00 ns

1. TVALID = TREADY = 0.10

$$\textbf{Average delay (s)} = \frac{31{,}264{,}621\, cycles * 3.00\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 9.3794 \times 10^{-6}\ \textbf{s}$$

2. TVALID = TREADY = 0.50

$$\textbf{Average delay (s)} = \frac{7{,}887{,}821\, cycles * 3.00\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 2.3663 \times 10^{-6}\ \textbf{s}$$

3. TVALID = TREADY = 1.00

$$\textbf{Average delay (s)} = \frac{5{,}003{,}221\, cycles * 3.00\, \frac{ns}{cycle}}{10{,}000\, ops} * 10^{-9} \frac{s}{ns} = 1.5010 \times 10^{-6}\ \textbf{s}$$

Table 14: Convolution System Design Comparison (Average Delay)

	Design 1 **(INW=18, R=9, C=8, MAXK=5)**	**Design 2** **(INW=24, R=16, C=17, MAXK=9)**
TVALID and TREADY prob	**Average Delay** (s)	**Average Delay** (s)
0.10	878.14×10^{-9}	9.3794×10^{-6}
0.50	226.91×10^{-9}	2.3663×10^{-6}
1.00	146.47×10^{-9}	1.5010×10^{-6}

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:

Energy (J) = Power (W) * Time (s)

Design 1: INW=18, R=9, C=8, MAXK=5, Power = $678\,550 \times 10^{-6}$ W
1. TVALID = TREADY = 0.10
Energy (J) = ($678\,550 \times 10^{-6}$) W * 878.14×10^{-9} s = 595.86×10^{-9} **J**

2. TVALID = TREADY = 0.50
Energy (J) = ($678\,550 \times 10^{-6}$) W * 226.91×10^{-9} s = 153.97×10^{-9} **J**

3. TVALID = TREADY = 1.00
Energy (J) = ($678\,550 \times 10^{-6}$) W * 146.47×10^{-9} s = 99.39×10^{-9} **J**

Design 2: INW=24, R=16, C=17, MAXK=9, Power = 1.0375 W
1. TVALID = TREADY = 0.10
Energy (J) = 1.0375 W * 9.3794×10^{-6} s = 9.731×10^{-6} **J**

2. TVALID = TREADY = 0.50
Energy (J) = 1.0375 W * 2.3663×10^{-6} s = 2.455×10^{-6} **J**

3. TVALID = TREADY = 1.00
Energy (J) = 1.0375 W * 1.5010×10^{-6} s = 1.557×10^{-6} **J**

Table 15: Convolution System Design Comparison (Average Energy)

	Design 1 **(INW=18, R=9, C=8, MAXK=5)**	**Design 2** **(INW=24, R=16, C=17, MAXK=9)**
TVALID and TREADY prob	**Average Energy** (J)	**Average Energy** (J)
0.10	595.86×10^{-9}	9.731×10^{-6}
0.50	153.97×10^{-9}	2.455×10^{-6}
1.00	99.39×10^{-9}	1.557×10^{-6}

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:
Area-Delay Product (μm^2 * s) = Area (μm^2) * Time (s)

Design 1: INW=18, R=9, C=8, MAXK=5, Area = 129,884.87 μm^2
1. TVALID = TREADY = 0.10
Area-Delay Product (μm^2*s) = 129,884.87 μm^2 * 878.14×10^{-9} s = 114.06×10^{-3} **μm^2*s**

2. TVALID = TREADY = 0.50

Area-Delay Product (μm^2*s) = 129,884.87 μm^2 * 226.91×10^{-9} s = 29.47×10^{-3} **μm^2*s**

3. TVALID = TREADY = 1.00
Area-Delay Product (μm^2*s) = 129,884.87 μm^2 * 146.47×10^{-9} s = 19.02×10^{-3} **μm^2*s**

Design 2: INW=24, R=16, C=17, MAXK=9, Area = 728,539.68 μm^2
1. TVALID = TREADY = 0.10
Area-Delay Product (μm^2*s) = 728,539.68 μm^2 * 9.3794×10^{-6} s = 6.833 **μm^2*s**

2. TVALID = TREADY = 0.50
Area-Delay Product (μm^2*s) = 728,539.68 μm^2 * 2.3663×10^{-6} s = 1.724 **μm^2*s**

3. TVALID = TREADY = 1.00
Area-Delay Product (μm^2*s) = 728,539.68 μm^2 * 1.5010×10^{-6} s = 1.094 **μm^2*s**

Table 16: Convolution System Design Comparison (Area-Delay Product)

	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-Delay Product** (μm$^2 \cdot s$)	**Area-Delay Product** (μm$^2 \cdot s$)
0.10	114.06×10^{-3}	6.833
0.50	29.47×10^{-3}	1.724
1.00	19.02×10^{-3}	1.094

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: Based on these metrics, I would say my design is less efficient than Part 4's system. This is due to the increase in area-delay product and energy. Due to the amount of power it would consume for smaller designs, it is feasible, and very efficient. For larger designs, it is not possible because it draws so much power.