1.0 Introduction

The following document serves as the lab report for assignment 1 of ECE 1756 (Fall 2021). This report is written by Sheng Zhao (1003273913).

Following section 1.0's introduction, section 2.0 describes the implementations of the pipelined and shared hardware circuits, as well as the corresponding simulations that were conducted. Section 3.0 then summarizes the desired data as specified by the assignment document in the table that was provided, as well as answer the questions provided. Brief calculations for the different entries will be included in the appendix for tidiness. The appendix goes into section 4.0.

2.0 Circuit Implementation

Three circuits were used to complete the assignment. Each subsection seeks to provide the necessary information to describe each circuit in full.

2.1 Baseline Circuit

For brevity, since the baseline circuit used was provided in full with the assignment document, this section will only provide a brief overview of the circuit.

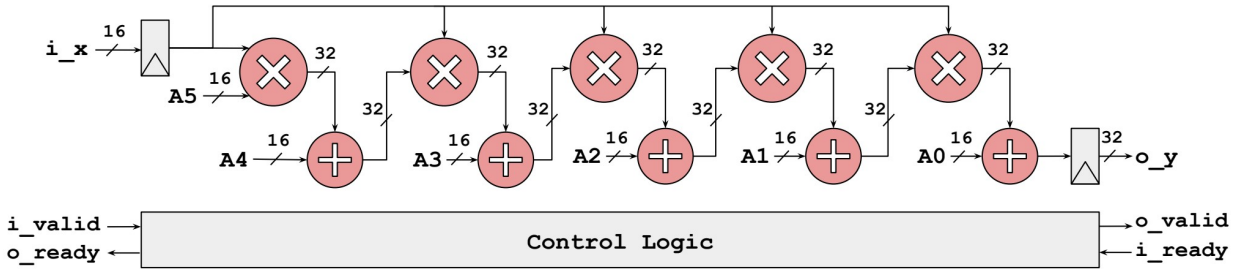

Figure 1: Baseline circuit block diagram

With reference to figure 1 (copied from the assignment document), the baseline circuit implements the desired calculations, a Taylor expansion series computed up to the 5^{th} power, with a streaming dataflow model. For every given valid input i_x, the value is stored in the first register in the first positive edge of the clock. Once stored, the computation begins and completes with enough setup time before the next positive edge for the output register. The output o_y is then ready after the second positive edge of the clock.

2.2 Pipelined Circuit

The pipelined circuit seeks to improve the maximum operating frequency by minimizing the amount of computation done within each clock cycle.

2.2.1 Datapath

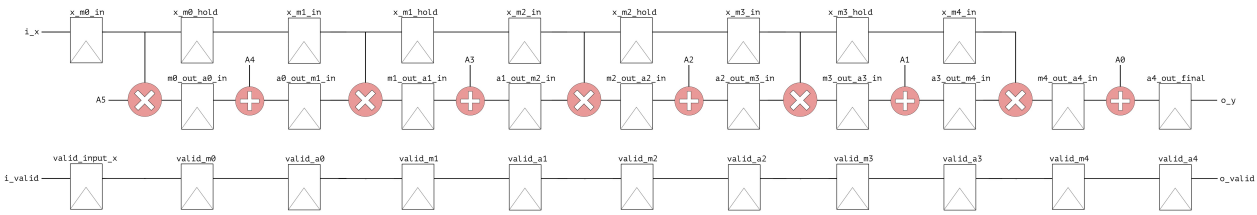

Figure 2: Pipelined circuit datapath

With respect to the baseline circuit, and Figure 2, more registers are added between the multipliers and adders. Since less work now needs to be down within each clock cycle, the period can be decreased further compared to before, leading to a higher operating frequency. In order to preserve the correctness of the circuit, registers are also added for the the i_valid propagation path. Since signals will only proceed when the downstream circuit is ready and consuming data, both the valid signals and actual computation result will propagate in sync.

2.2.2 Operation

Similar to the baseline circuit, the circuit performs calculation in a streaming dataflow way. As valid inputs are inserted into the pipeline, it proceeds through the various steps of the computation every clock cycle. Since there are registers after each multiplier/adder, new inputs can be safely inserted and begin their own computations without interferring with earlier ones. When invalid i_x values are inserted, the calculation is still performed on them. However, since the valid signal is also propagated down the pipeline in sync, the o_valid signal will turn false as invalid computations are completed, thereby automatically prevent downstream circuits from receiving invalid outputs.

2.2.3 Correctness

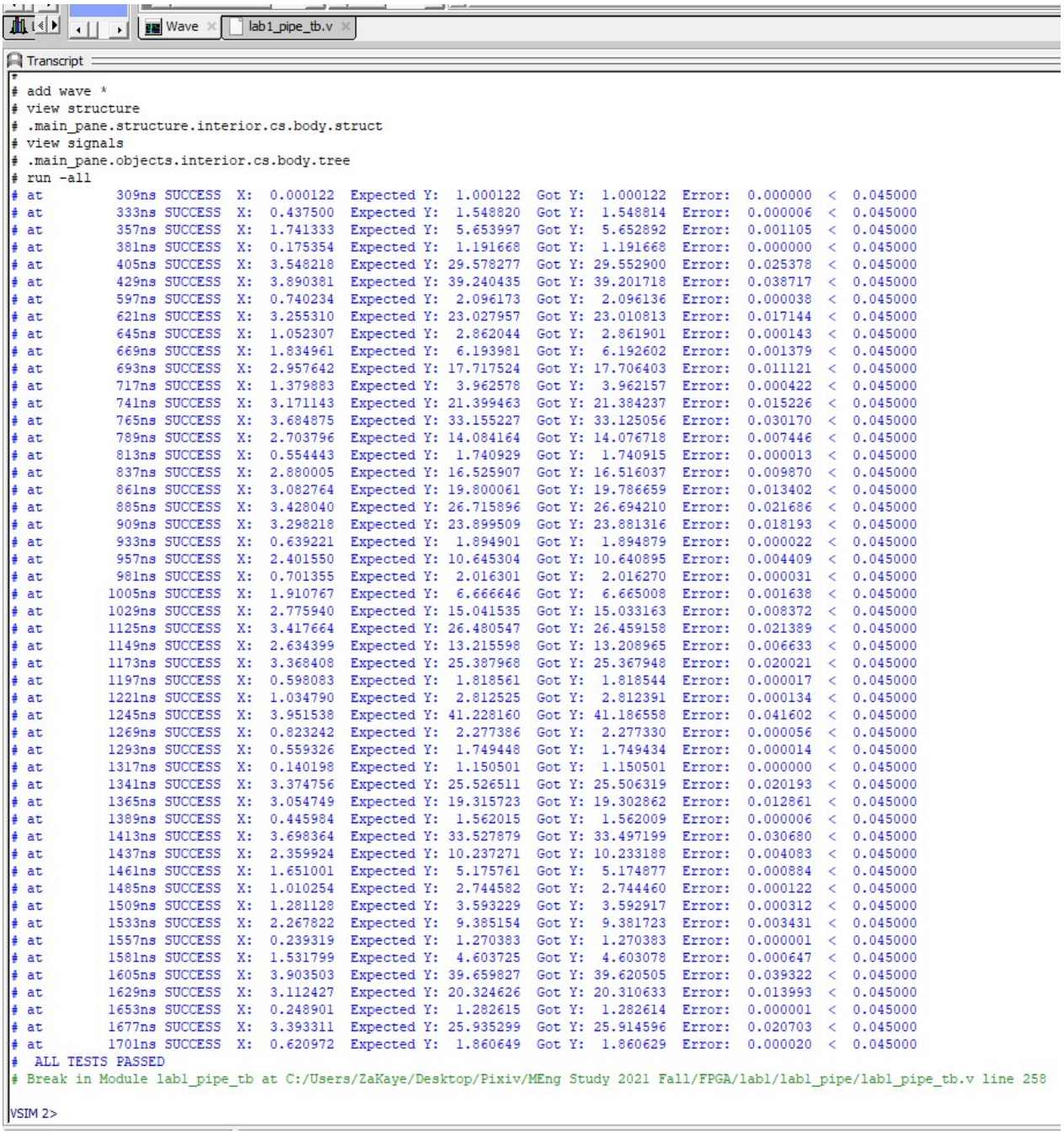

Figure 3: Pipelined circuit correctness

The pipelined circuit successfully passes all test cases in ModelSim.

2.3 Shared Hardware Circuit

The shared hardware circuit limits the use of multiplier and adder blocks to just one each. A Finite State Machine (FSM) is then added to control the datapath between the input i_x to the output o_y to facilitate the proper computation of the desired Taylor series expansion while reusing the same computation units.

2.3.1 Datapath

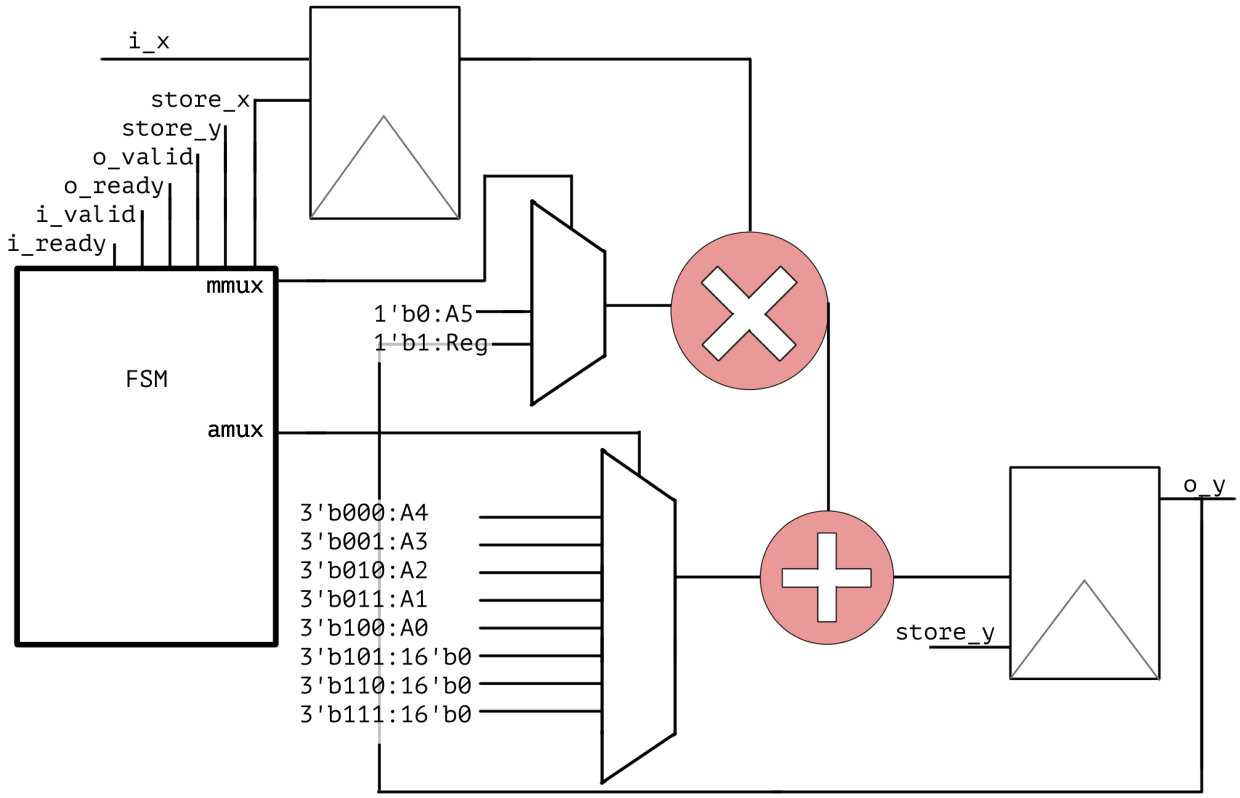

Figure 4: Shared hardware circuit datapath

To reuse the multiplier and adder, we first need to construct a circuit that allows us to control the inputs going into these units. With respect to Figure 4, a 2:1 multiplexer is added to the multiplier while a 8:1 multiplexer is added to the adder. The corresponding control signals are named mmux and amux (for multiplier mux and adder mux respectively). In addition, enable signals (named store_x and store_y) are added for the input and output registers to prevent these from setting invalid values.

2.3.2 State Transition

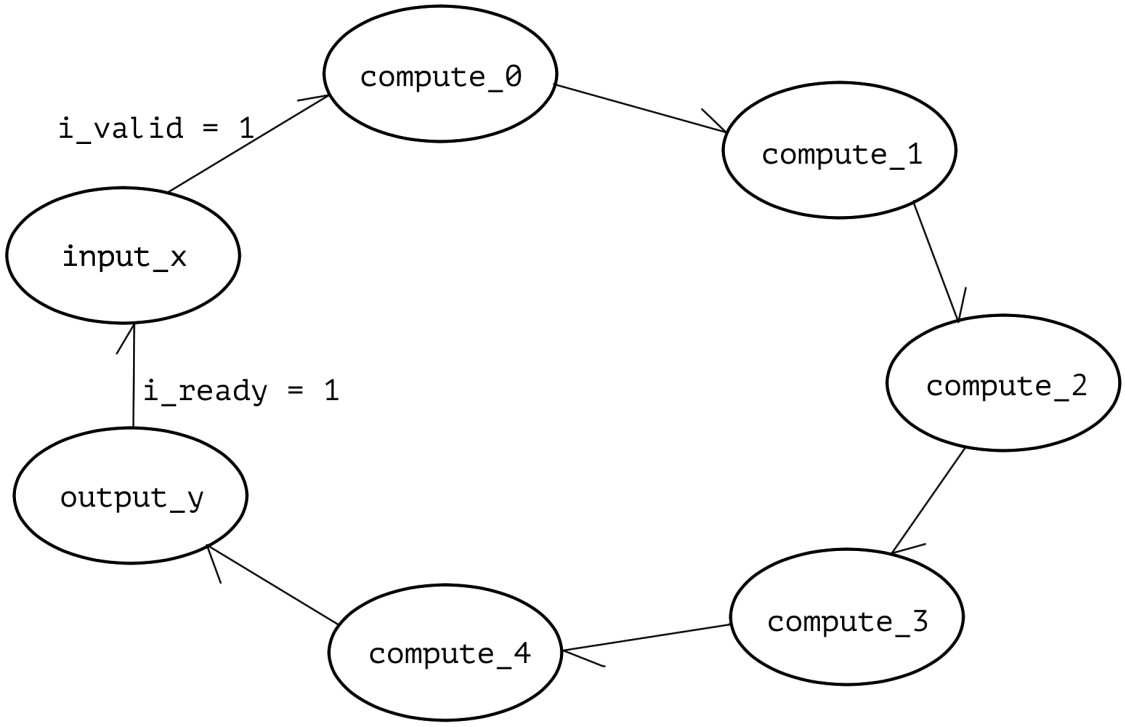

Figure 5: Shared hardware circuit state transitions

The state transitions for the FSM block are show in Figure 5. There are a total of 7 states, one for input phase, five for the computation phase, and one for the output phase. With the exception of states input_x and output_y, all other states transit to the next on each subsequent clock cycle since they do not affect circuits downstream or upstream. In the input_x state, we wait until i_valid is ready since we need a valid i_x to begin our calculations. Similarly in the output_y state, we wait until the downstream circuit is ready and received our computed value before proceeding with the next calculation.

The control signals in each state is summarized in Table 1. Note that during the input and output phases where no computation is done, signals mmux and amux are irrelevant (although they are both set to zero).

	input_x	compute_0	compute_1	compute_2	compute_3	compute_4	output_y
o_valid	1'b0						1'b1
o_ready	1'b1	1'b0					
store_x	1'b1	1'b0					
store_y	1'b0	1'b1					1'b0
mmux	DC	1'b0	1'b1				DC
amux	DC	3'b000	3'b001	3'b010	3'b011	3'b100	DC

Table 1: Control signal values for each state

2.3.3 Operation

Computation for each valid value of i_x begins with the input_x state. In this state, instead of making sure the incoming i_x is valid before storing, any i_x that is inputted gets stored into the input register. Instead, the state will only transit to the next when i_valid is toggled true.

Once the state transits to compute_0, the computation begins. First, the mmux control signal is set to 1'b0 to select for A5 as the input to the multiplier. Note that since A5 is only 16 bits wide, we have to pad 5 zeros on the left and 11 zeros on the right (to convert from Q2.14 to Q7.25) before passing to the 32x16 multiplier. The amux control signal is set to 3'b000 for A4 and summed with the result from the multiplier. This sum is then stored in the output register. However, since it is only the intermediate result, o_valid is set to 1'b0 to prevent the downstream circuit from retrieving it. Once done, we simply transit to the next state, compute_1, to carry on the computation. This time with mmux set to 1'b1 to use the result from the previous state, and amux set to 3'b001 for A3. The process remains the same for all compute_x states with only the amux control signal changing.

Once we finish computation in compute_4 and have the final result ready in the output register, we transit to output_y, which sets the o_valid signal to 1'b1. We then wait for the downstream circuit to set i_ready to 1'b1, which signals to us that it is ready and will retrieve the signal. Assuming the result is retrieved in one clock cycle, we can then transit back to input_x and wait for the next i_x. Note that the output_y and input_x states are not merged since it might introduce an error in the event that the downstream circuit mistakes the output of the previous input to be the one for the new input that was just sent into our circuit.

2.3.4 Correctness

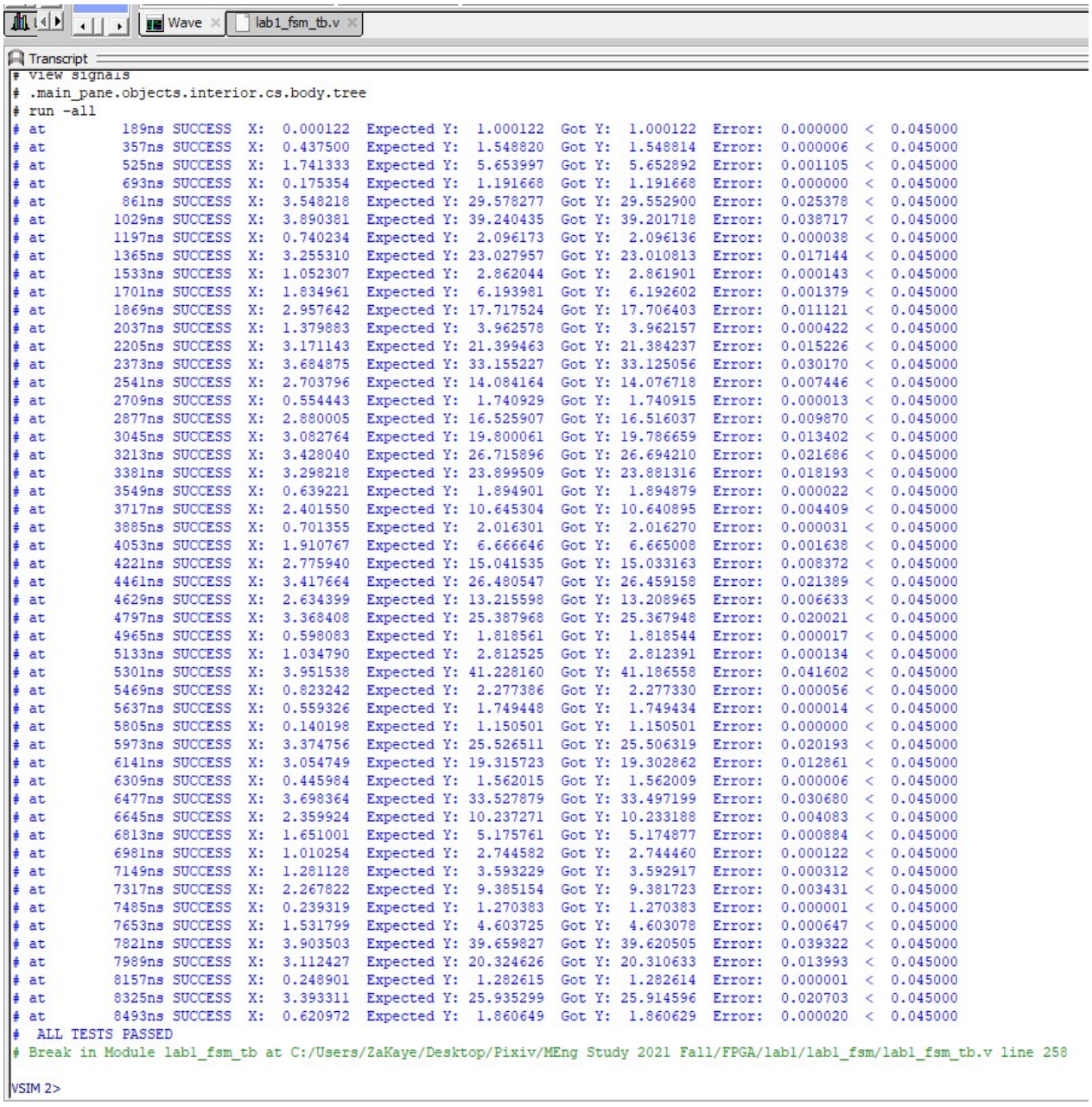

Figure 6: Shared hardware circuit correctness

The shared hardware circuit successfully passes all test cases in ModelSim.

3.0 Results

The following table is a copy of table 1 from the assignment pdf.

	Baseline Circuit (Appendix A)	Pipelined Circuit (Appendix B)	Shared HW Circuit (Appendix C)
Resources for one circuit	21 ALMs + 9 DSPs	98 ALMs + 9 DSPs	28 ALMs + 2 DSPs
Operating frequency	46.1 MHz	249.1 MHz	161.3 MHz
Critical path	DSP mult-add + 2x DSP mult + LE-based adder + 6x DSP mult + LE-based adder	LE-based mult + LE-based adder + 1x DSP mult	LE-based Mux + 2x DSP mult + LE-based adder
Cycles per valid output	1	1 (once saturated)	7
Max. # of copies/device	168	168	759
Max. throughput for a full device (computations/s)	7.744×10^9 (@ 46.1 MHz) 7.056×10^9 (@ 42 MHz)	41.848×10^9 (@ 249.1 MHz) 7.056×10^9 (@ 42 MHz)	23.014×10^6 (@ 161.3 MHz) 7.056×10^9 (@ 42 MHz)
Dynamic power of one circuit @ 42 MHz	1.91 mW	2.01 mW	0.60 mW
Max. throughput/Watt for a full device @ 42 MHz	3.482×10^9	3.454×10^9	2.109×10^9

Table 2: Implementation results for the 3 different implementations of the studied circuit.

3.1 Questions

3.1.1 What are the different sources of error (i.e. difference between exp(x) and Hardware Output in the graph you plotted for the testbench output)? What changes could you make to the circuit to reduce this error?

A significant source of error is due to the formula used to compute the exponential itself. Since we are limited to the amount of hardware resources available, we are not able to expand the Taylor expansion up to higher powers of x indefinitely. As a result, as values of x get larger, the higher power terms, despite having a relatively smaller coefficient compared to the lower power terms, starts to be large enough to cause a difference between the computed value and the true value. This is also supported by the error between the true value and the computed value getting larger as the values of x get larger. Changing the circuits to factor in more terms would increase the accuracy of the computations.

Another source of error would come from the resolution of the calculation, specifically the number of digits used to contain the intermediate and final calculation results. Depending on the values of x and the coefficients of the series, some portion of the results might be lost at each step due to the number of bits not being able to support the resolution required. More bits could be used to increase the accuracy of the computations.

3.1.2 Which of the 3 hardware circuits (baseline, pipelined, and shared) achieves the highest throughput/device? Explain the reasons for the efficiency differences between them.

Should each of the three hardware circuits operate at their maximum operating frequency, the pipelined circuit would have the highest throughput by far (41.848×10^9 computations/s), followed by the baseline circuit (7.744×10^9 computations/s) and the shared hardware circuit in last (23.014×10^6 computations/s).

Due to being pipelined, the computations required to be completed within each clock cycle is significantly reduced compared to the baseline circuit. As a result, the overall design can operate at a much higher frequency of 249.1 MHz compared to 46.1 MHz. While each input will now take significantly more cycles to be computed (10 times more), once the pipeline is saturated, it can still produce a valid output every cycle (assuming a valid input is available every cycle). Combined with the much higher frequency, the pipelined circuit achieves a much higher throughput.

The shared hardware circuit on the other hand, due to it reusing the functional units, has siginificantly less functional units compared to the other two. Each valid input thus takes more cycles to finish computing (7 times more) than the baseline circuit. In addition, due to the functional units being reused, new inputs have to wait until the current input is processed before they can get started. Thus, unlike the pipelined circuit, every input will always take seven cycles to complete. While the operating frequency has increase by around four times (161.3 MHz compared to 46.1 MHz), the seven times increase in cycle count caused the shared hardware circuit to have significantly less throughput.

3.1.3 Look at the average toggle rates of the blocks for the 3 circuits (this information is in the PowerAnalyzer report). Explain why some circuit styles lead to higher toggle rates for the DSP blocks and combinational/registered logic cells than others. Comment on the relative efficiency of the 3 circuits in terms of computations/J, and explain why each style is more or less efficient in computations/J than the others.

With respect to Table 3, we can see that the baseline circuit has the highest toggle rate, followed by the shared hardware circuit, and the pipelined circuit in last.

In terms of power efficiency, calculations were done at 42 MHz (for both power and throughput) for each design since power analysis was only available at 42 MHz. Based on Table 2,

	Baseline circuit	Pipelined circuit	Shared HW circuit
Toggle rate (millions of transition/s)	12.819	11.527	12.010

Table 3: Toggle rates for the 3 different circuit implementations (Appendix D)

the baseline circuit was the best at throughput/Watt, followed by the pipelined circuit, and the shared hardware circuit in last.

With respect to the baseline circuit numbers, the pipeline followed closely after due to the fact that the advantage it gained in terms of higher operating frequency was negated due to the calculation being performed at 42 MHz. As a result, the increase in number of registers present in the pipelined circuit caused the dynamic power to be slightly higher than the baseline circuit, and with the throughput at 42 MHz being the same for the two, the pipelined circuit lost to the baseline circuit in terms of power efficiency by a small margin.

As for the shared hardware circuit, since there are significantly less functional units, the dynamic power was significantly less. However, the throughput fell even more, causing the energy efficiency numbers to not reflect the energy savings.

4.0 Appendix

Appendix A: Calculations for baseline circuit

i. Resources for one circuit

Assignment Editor | Compilation Report - lab1

Fitter Resource Usage Summary

a

	Resource	Usage	%
1	Logic utilization (ALMs n...d / total ALMs on device)	48 / 427,200	< 1 %
2	ALMs needed [=A-B+C]	48	
1	[A] ALMs used in final placement [=a+b+c+d]	28 / 427,200	< 1 %
2	[B] Estimate of ALMs re...erable by dense packing	7 / 427,200	< 1 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	27 / 427,200	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	27	
3	Difficulty packing design	Low	
4	Total LABs: partially or completely used	5 / 42,720	< 1 %
5	Combinational ALUT usage for logic	43	
6	Dedicated logic registers	34	
7	Virtual pins	53	
8	I/O pins	1 / 992	< 1 %
9	Total MLAB memory bits	0	
10	Total block memory bits	0 / 55,562,240	0 %
11	Total block memory implementation bits	0 / 55,562,240	0 %
12	Total DSP Blocks	9 / 1,518	< 1 %
13	Global signals	1	
14	JTAGs	0 / 1	0 %

Figure 7: Baseline circuit resource usage

$$UsedALMS = 48 - 27 = 21ALMs$$

ii. Operating frequency

```
        Found TIMING_ANALYZER_REPORT_SCRIPT_INCLUDE_DEFAULT_ANALYSIS =
        Analyzing Slow 900mV 100C Model
332148 Timing requirements not met
332146 Worst-case setup slack is -19.600
332146 Worst-case hold slack is 0.103
332140 No Recovery paths to report
332140 No Removal paths to report
332146 Worst-case minimum pulse width slack is -0.820
        Analyzing Slow 900mV 0C Model
332146 Worst-case setup slack is -20.678
332146 Worst-case hold slack is 0.070
332140 No Recovery paths to report
332140 No Removal paths to report
```

Figure 8: Baseline circuit compilation report on slacks

$$f_{max} = 1/(1 + 20.678) \approx 46.1MHz$$

iii. Critical path

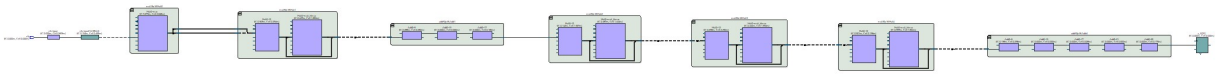

Figure 9: Baseline circuit critical path

Critical path:

- one DSP block for Mult0 and Addr0
- two DSP blocks for Mult1
- three logic blocks for Addr1
- two DSP blocks for Mult2
- two DSP blocks for Mult3
- two DSP blocks for Mult4
- five logic blocks for Addr4

iv. Cycles per valid output

Each input takes one clock cycle to be processed.

v. Max. # of copies/device

$$Max.\ copies\ (based\ on\ ALM\ count) = 427200/48 = 8900$$

$$Max.\ copies\ (based\ on\ DSP\ count) = 1518/9 \approx 168$$

Since DSP count is the limiting factor, approximate maximum number of copies per device is 168.

vi. Max. throughput for a full device (computations/s)

Operating at worst case of 46.1 MHz with each input taking one cycle to compute its corresponding output, we can process up to 46,100,000 inputs per second. Factoring the approximate number of copies per device:

$$Max.\ throughput = 46.1 \times 10^6 * 168 \approx 7.744 \times 10^9\ computations/sec$$

To facilitate computation of subsection viii., we shall repeat the computation with a operating frequency of 42 MHz:

$$Max.\ throughput = 42.1 \times 10^6 * 168 = 7.056 \times 10^9\ computations/sec$$

vii. Dynamic power of one circuit at 42 MHz

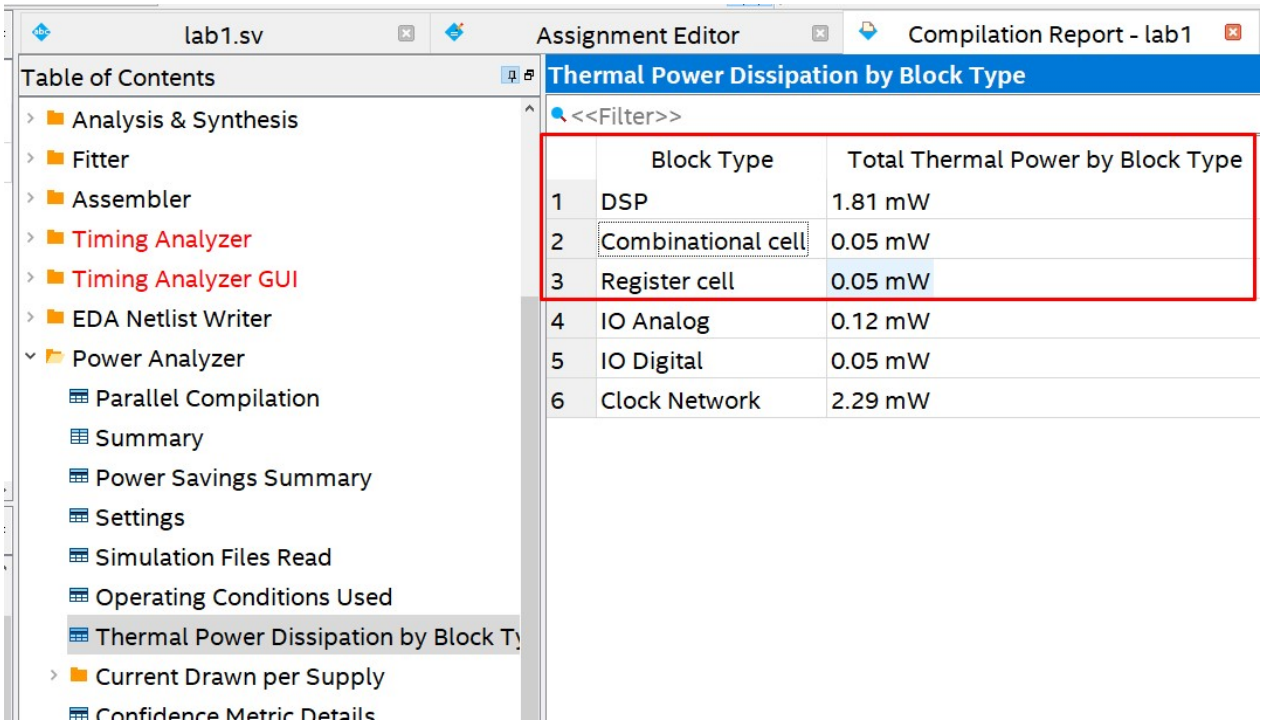

Figure 10: Baseline circuit power consumption at 42 MHz

$$Power\ of\ one\ circuit = 1.81 + 0.05 + 0.05 = 1.91mW$$

viii. Max. throughput/Watt for a full device

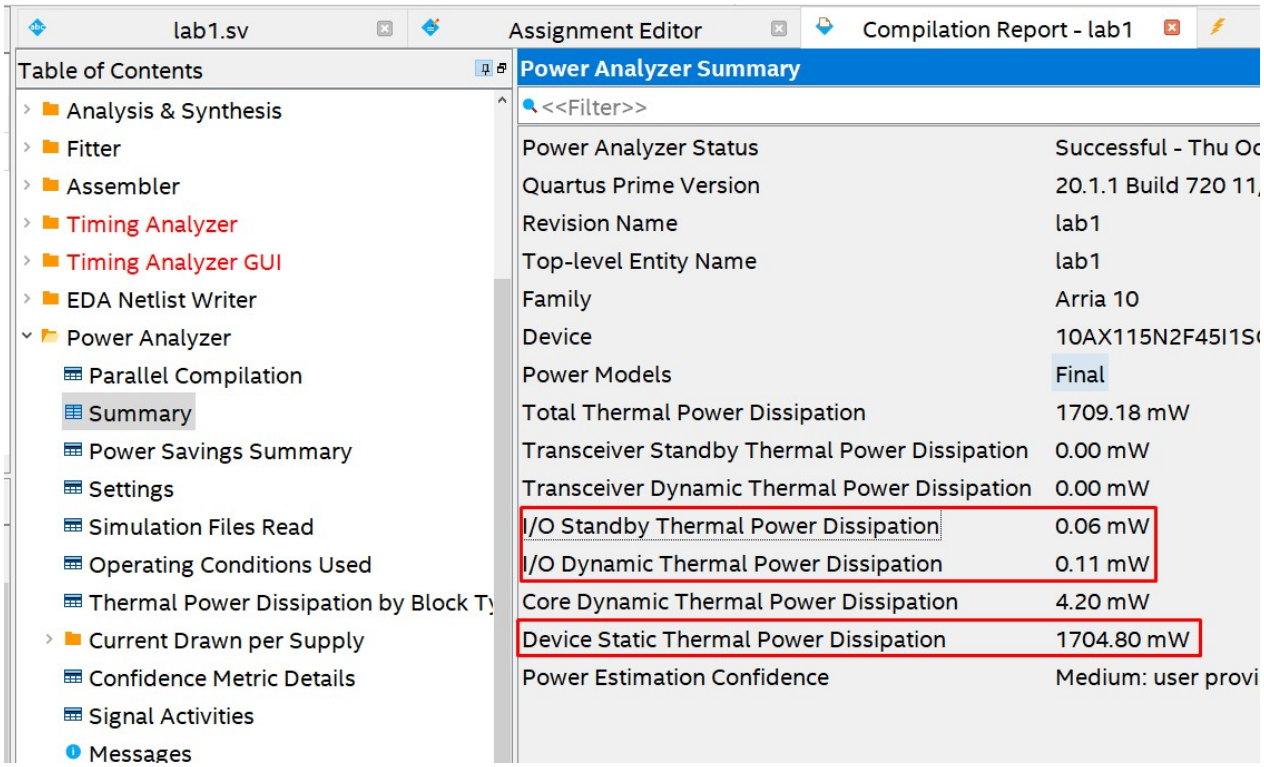

Figure 11: Baseline circuit power consumption summary at 42 MHz

To compute the throughput/Watt, we need to compute the total approximate power consumption assuming we have the maximum number of copies of the circuit on the device. Assuming I/O and static power does not change, with 168 copies of the circuit:

$$Total\ power\ consumption = (1.91 * 168) + 1704.8 + 0.06 + 0.11 = 2025.85mW$$

$$Throughput/Watt\ @\ 42\ MHz = 7.056 \times 10^9 / 2025.85 \times 10^{-3} \approx 3.482 \times 10^9\ computations/Watt$$

Appendix B: Calculations for pipelined circuit

i. Resources for one circuit

lab1_pipe.sv | Assignment Editor

Fitter Resource Usage Summary

a

	Resource	Usage	%
1	Logic utilization (ALMs n...d / total ALMs on device)	125 / 427,200	< 1 %
2	ALMs needed [=A-B+C]	125	
1	[A] ALMs used in final placement [=a+b+c+d]	168 / 427,200	< 1 %
2	[B] Estimate of ALMs re...erable by dense packing	70 / 427,200	< 1 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	27 / 427,200	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	27	
3	Difficulty packing design	Low	
4	Total LABs: partially or completely used	25 / 42,720	< 1 %
5	Combinational ALUT usage for logic	173	
6	Dedicated logic registers	299	
7	Virtual pins	53	
8	I/O pins	1 / 992	< 1 %
9	Total MLAB memory bits	0	
10	Total block memory bits	0 / 55,562,240	0 %
11	Total block memory implementation bits	0 / 55,562,240	0 %
12	Total DSP Blocks	9 / 1,518	< 1 %
13	Global signals	2	

Figure 12: Pipelined circuit resource usage

$$UsedALMS = 125 - 27 = 98ALMs$$

ii. Operating frequency

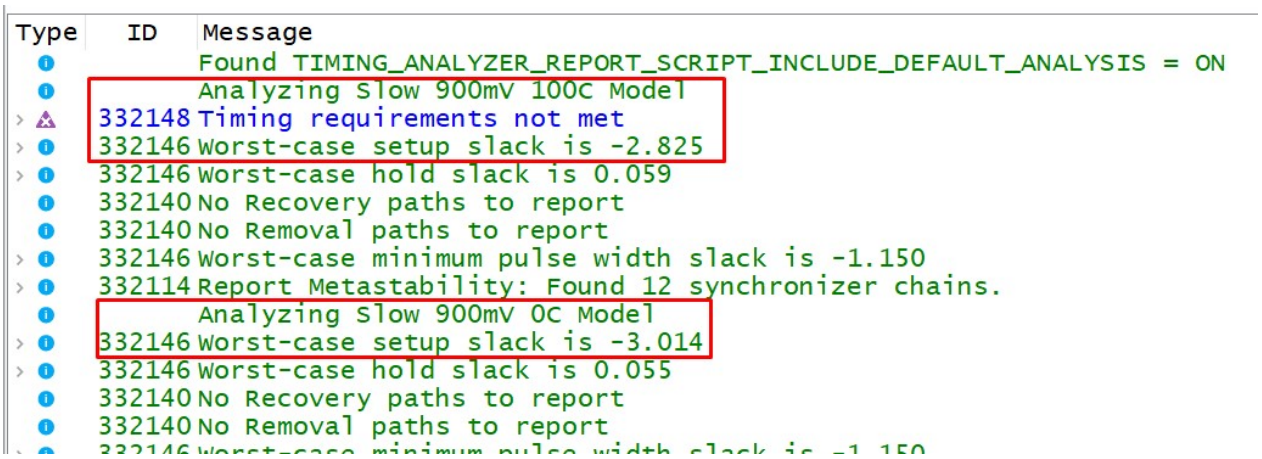

Figure 13: Pipelined circuit compilation report on slacks

$$f_{max} = 1/(1 + 3.014) \approx 249.1MHz$$

iii. Critical path

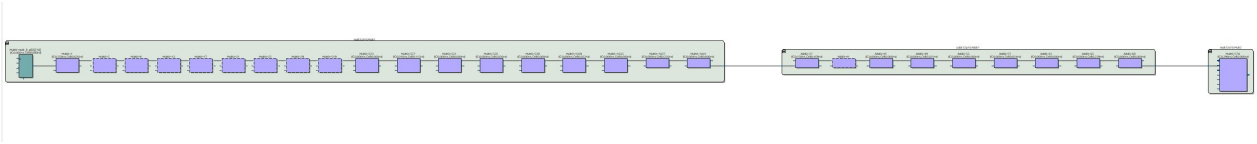

Figure 14: Pipelined circuit critical path

Critical path:

- 18 logic blocks for Mult1
- 9 logic blocks for Addr1
- one DSP block for Mult2

iv. Cycles per valid output

Although each input takes 10 clock cycles to compute, once the pipeline is saturated, output is ready every clock cycle. Assuming many inputs are to be processed, the average number of cycles per valid output will be one.

v. Max. # of copies/device

$$Max.\ copies\ (based\ on\ ALM\ count) = 427200/125 = 3417.6$$

$$Max.\ copies\ (based\ on\ DSP\ count) = 1518/9 \approx 168$$

Since DSP count is the limiting factor, approximate maximum number of copies per device is 168.

vi. Max. throughput for a full device (computations/s)

Operating at worst case of 249.1 MHz with each input taking one cycle to compute its corresponding output, we can process up to 249,100,000 inputs per second. Factoring the approximate number of copies per device:

$$Max.\ throughput = 249.1 \times 10^6 * 168 \approx 41.848 \times 10^9\ computations/sec$$

To facilitate computation of subsection viii., we shall repeat the computation with a operating frequency of 42 MHz:

$$Max.\ throughput = 42.1 \times 10^6 * 168 = 7.056 \times 10^9\ computations/sec$$

vii. Dynamic power of one circuit at 42 MHz

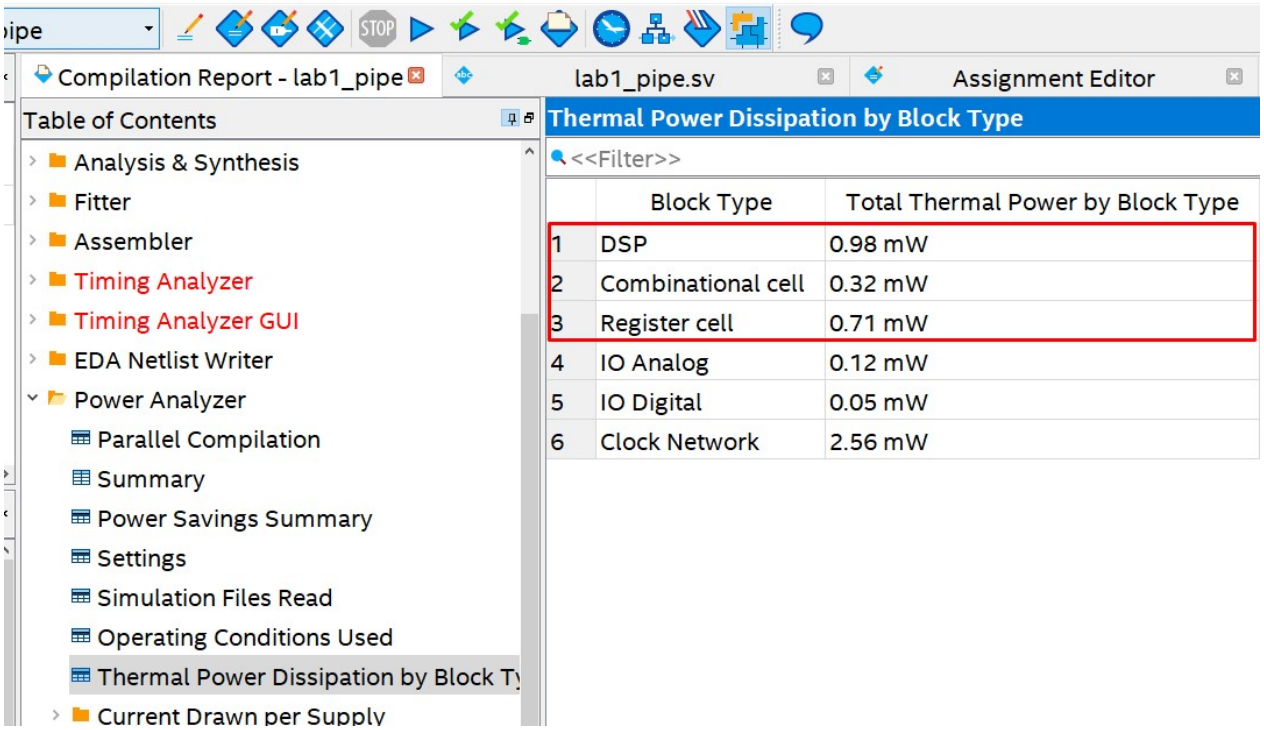

Figure 15: Pipelined circuit power consumption at 42 MHz

$$Power\ of\ one\ circuit = 0.98 + 0.32 + 0.71 = 2.01mW$$

viii. Max. throughput/Watt for a full device

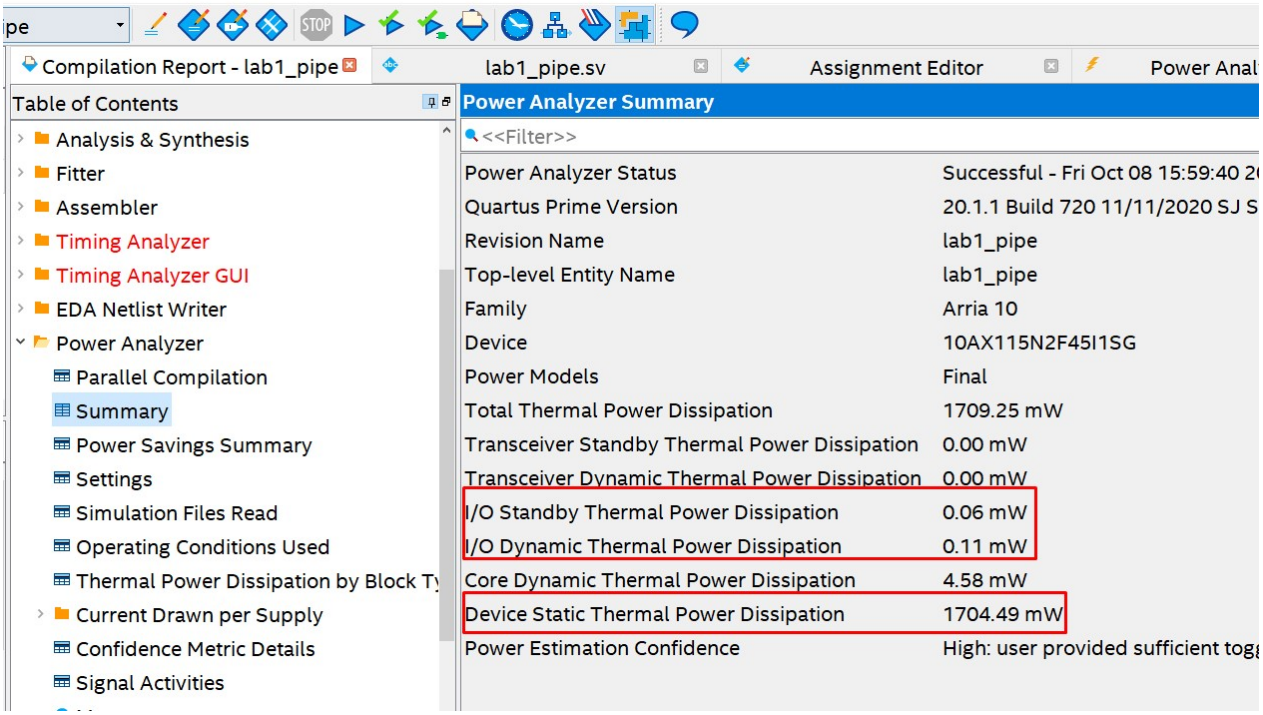

Figure 16: Pipelined circuit power consumption summary at 42 MHz

To compute the throughput/Watt, we need to compute the total approximate power consumption assuming we have the maximum number of copies of the circuit on the device. Assuming I/O and static power does not change, with 168 copies of the circuit:

$$Total\ power\ consumption = (2.01 * 168) + 1704.49 + 0.06 + 0.11 = 2042.34mW$$

$$Throughput/Watt\ @\ 42\ MHz = 7.056 \times 10^9 / 2042.34 \times 10^{-3} \approx 3.454 \times 10^9\ computations/Watt$$

Appendix C: Calculations for shared hardware circuit

i. Resources for one circuit

Assignment Editor | Compilation Report - lab1_fsm

Fitter Resource Usage Summary

a

	Resource	Usage	%
1	Logic utilization (ALMs n...d / total ALMs on device)	55 / 427,200	< 1 %
2	ALMs needed [=A-B+C]	55	
1	[A] ALMs used in final placement [=a+b+c+d]	32 / 427,200	< 1 %
2	[B] Estimate of ALMs re...erable by dense packing	4 / 427,200	< 1 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	27 / 427,200	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	27	
3	Difficulty packing design	Low	
4	Total LABs: partially or completely used	4 / 42,720	< 1 %
5	Combinational ALUT usage for logic	56	
6	Dedicated logic registers	43	
7	Virtual pins	53	
8	I/O pins	1 / 992	< 1 %
9	Total MLAB memory bits	0	
10	Total block memory bits	0 / 55,562,240	0 %
11	Total block memory implementation bits	0 / 55,562,240	0 %
12	Total DSP Blocks	2 / 1,518	< 1 %
13	Global signals	1	

Figure 17: Shared hardware circuit resource usage

$$UsedALMS = 55 - 27 = 28ALMs$$

ii. Operating frequency

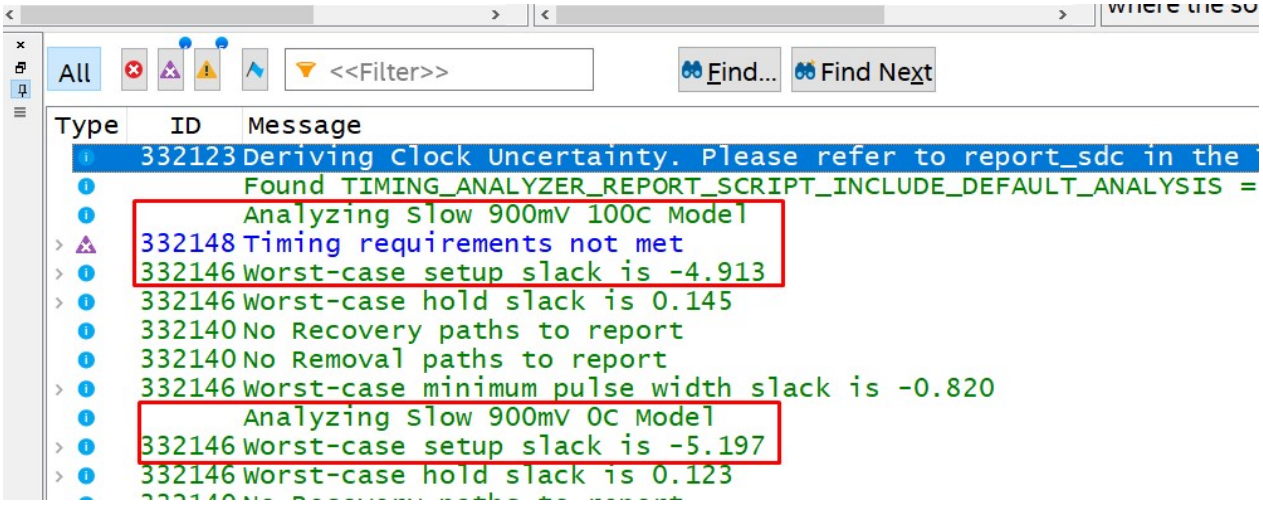

Figure 18: Shared hardware circuit compilation report on slacks

$$f_{max} = 1/(1 + 5.197) \approx 161.3MHz$$

iii. Critical path

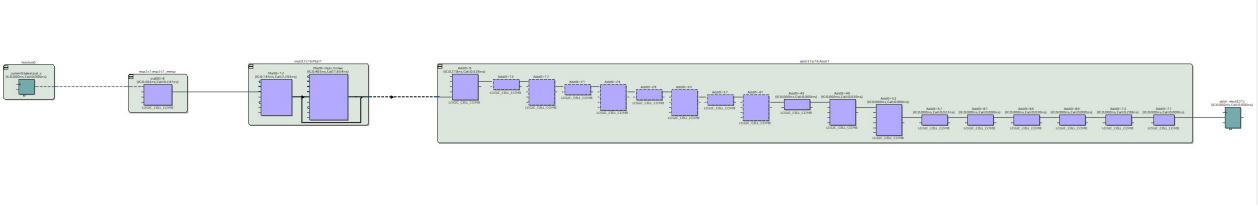

Figure 19: Shared hardware circuit critical path

Critical path:

- one logic block for 2:1 Mux
- two DSP blocks for Mult1
- 18 logic blocks for Addr1

iv. Cycles per valid output

Suppose the circuit is currently processing an input, then for the next input to be processed, it needs to wait one cycle to be stored, five cycles to be computed, and is ready on the seventh. Since the subsequent input cannot be processed until the next set of seven cycles begin (i.e. the eighth cycle in this current computation), each input takes a total of seven cycles despite only requiring five for the computation.

v. Max. # of copies/device

$$Max.\ copies\ (based\ on\ ALM\ count) = 427200/55 = 7767$$

$$Max.\ copies\ (based\ on\ DSP\ count) = 1518/2 = 759$$

Since DSP count is the limiting factor, approximate maximum number of copies per device is 759.

vi. Max. throughput for a full device (computations/s)

Operating at worst case of 161.1 MHz with each input taking seven cycle to compute its corresponding output, we can process up to:

$$Throughput\ per\ second = 161.1 \times 10^6/7 \approx 23.014 \times 10^6\ computations/sec$$

23,014,285 inputs per second. Factoring the approximate number of copies per device:

$$Max.\ throughput = 23014285 * 759 \approx 17.467 \times 10^9\ computations/sec$$

To facilitate computation of subsection viii., we shall repeat the computation with a operating frequency of 42 MHz:

$$Throughput\ per\ second = 42 \times 10^6/7 = 6 \times 10^6\ computations/sec$$

$$Max.\ throughput = 6 \times 10^6 * 759 = 4.554 \times 10^9\ computations/sec$$

vii. Dynamic power of one circuit at 42 MHz

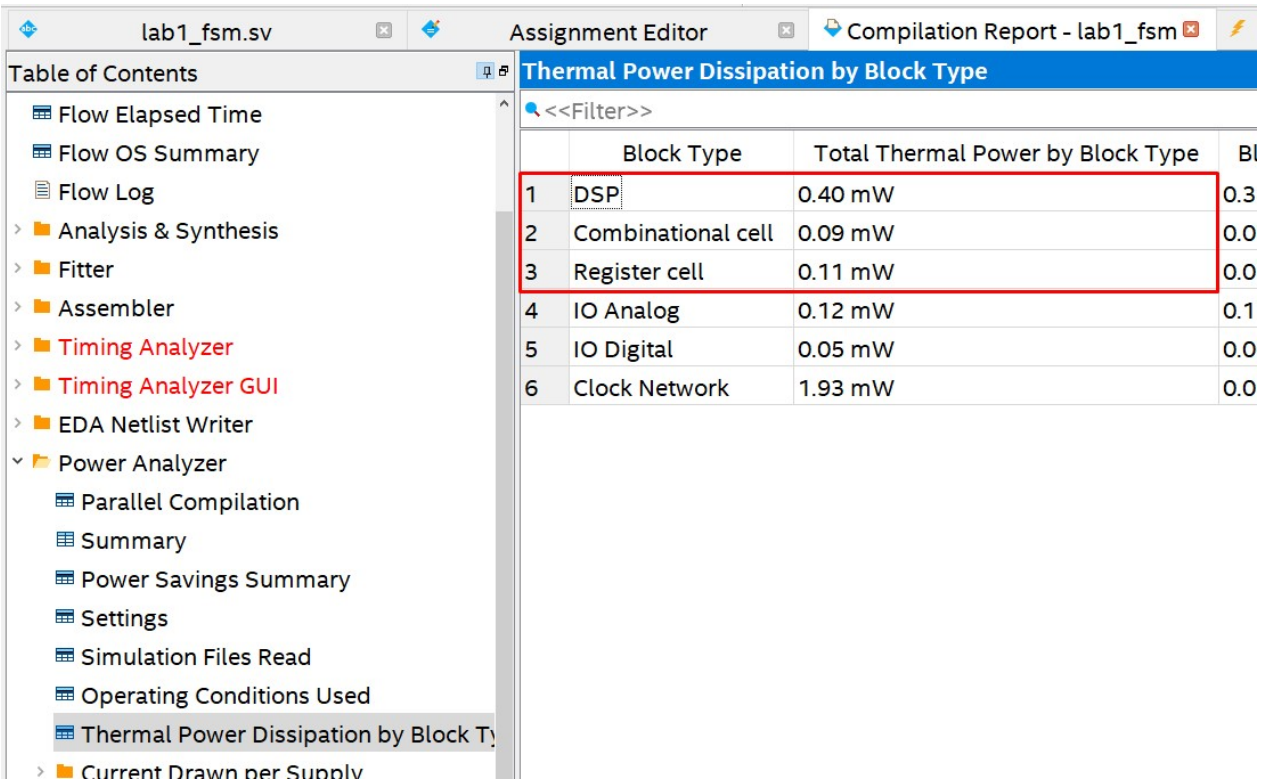

Figure 20: Shared hardware circuit power consumption at 42 MHz

$$Power\ of\ one\ circuit = 0.40 + 0.09 + 0.11 = 0.60mW$$

viii. Max. throughput/Watt for a full device

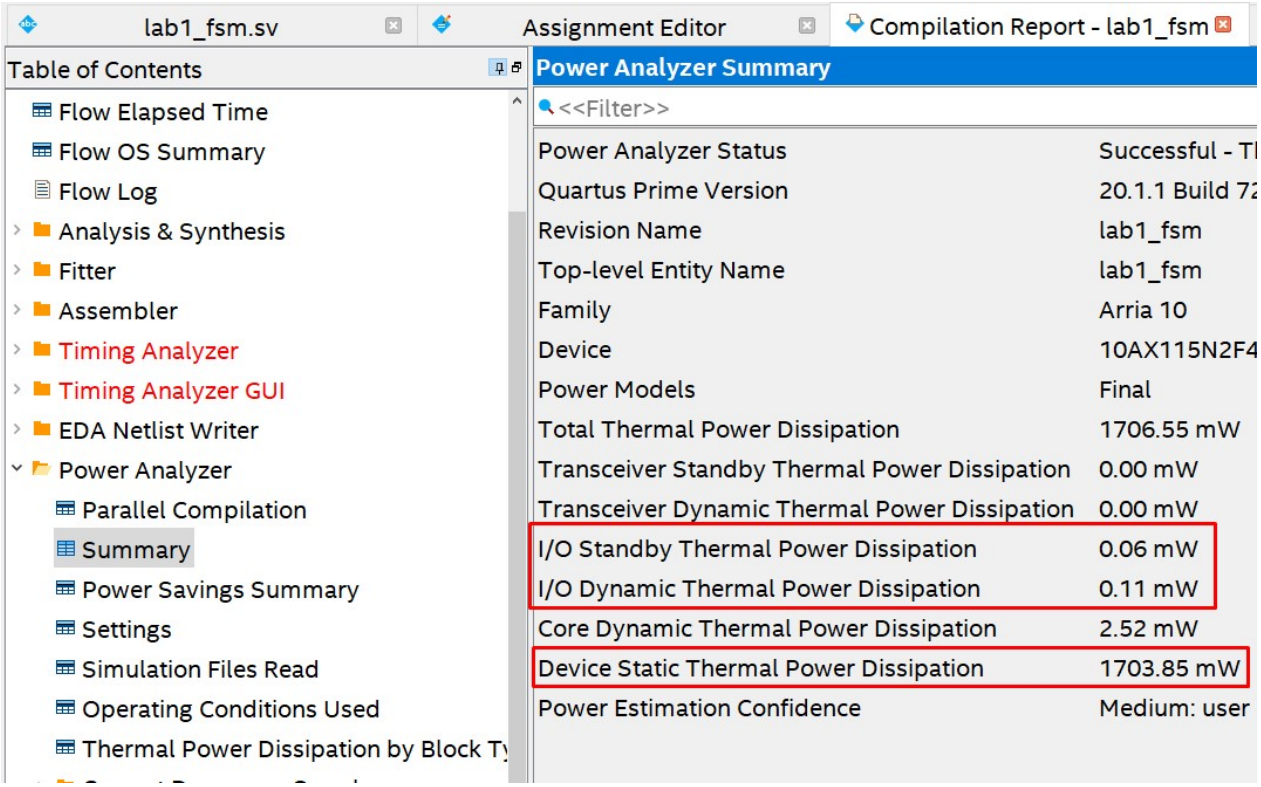

Figure 21: Shared hardware circuit power consumption summary at 42 MHz

To compute the throughput/Watt, we need to compute the total approximate power consumption assuming we have the maximum number of copies of the circuit on the device. Assuming I/O and static power does not change, with 168 copies of the circuit:

$$Total\ power\ consumption = (0.60 * 759) + 1703.85 + 0.06 + 0.11 = 2159.42mW$$

$$Throughput/Watt\ @\ 42\ MHz = 4.554 \times 10^9 / 2159.42 \times 10^{-3} \approx 2.109 \times 10^9\ computations/Watt$$

Appendix D: Circuit toggle rates

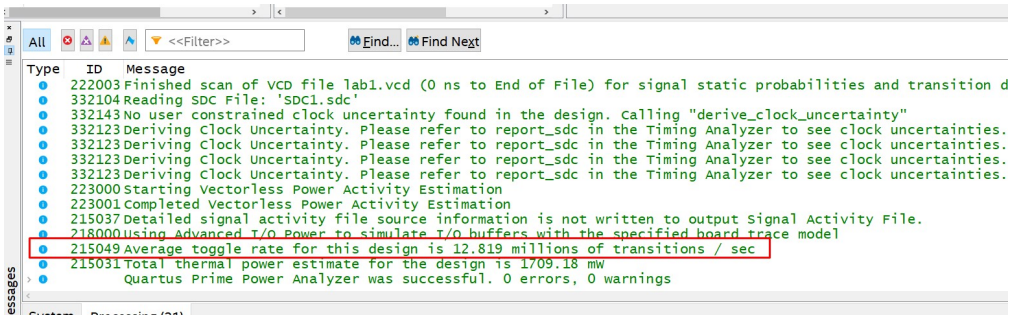

Figure 22: Baseline circuit toggle rate

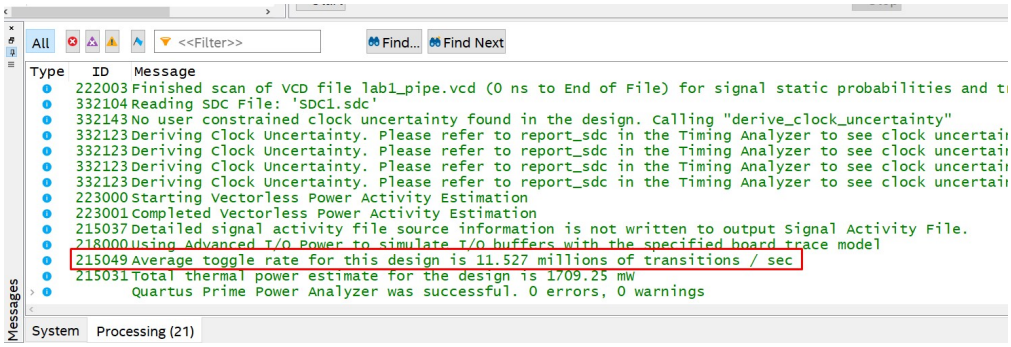

Figure 23: Pipelined circuit toggle rate

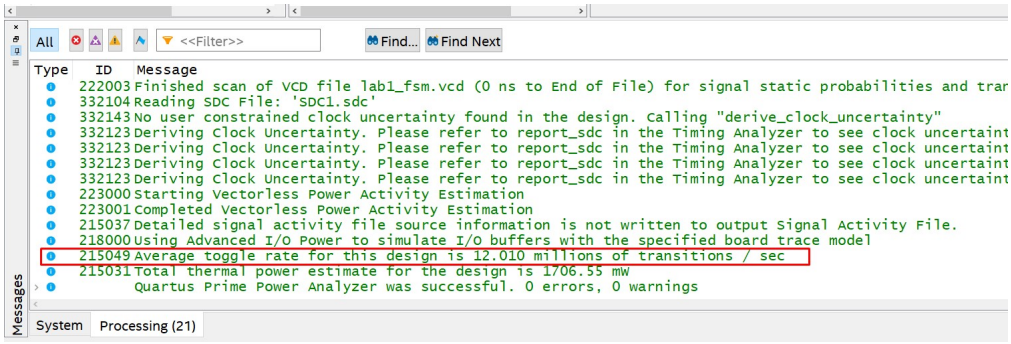

Figure 24: Shared hardware circuit toggle rate

Appendix E: .sv file for pipelined circuit

```
module lab1_pipe #
(
        parameter WIDTHIN = 16,                 // Input format is Q2.14 (2
        ↪  integer bits + 14 fractional bits = 16 bits)
        parameter WIDTHOUT = 32,        // Intermediate/Output format is
        ↪  Q7.25 (7 integer bits + 25 fractional bits = 32 bits)
        // Taylor coefficients for the first five terms in Q2.14 format
        parameter [WIDTHIN-1:0] A0 = 16'b01_00000000000000, // a0 = 1
        parameter [WIDTHIN-1:0] A1 = 16'b01_00000000000000, // a1 = 1
        parameter [WIDTHIN-1:0] A2 = 16'b00_10000000000000, // a2 = 1/2
        parameter [WIDTHIN-1:0] A3 = 16'b00_00101010101010, // a3 = 1/6
        parameter [WIDTHIN-1:0] A4 = 16'b00_00001010101010, // a4 = 1/24
        parameter [WIDTHIN-1:0] A5 = 16'b00_00000010001000  // a5 = 1/120
)
(
        input clk,
        input reset,

        input i_valid,
        input i_ready,
        output o_valid,
        output o_ready,

        input [WIDTHIN-1:0] i_x,
        output [WIDTHOUT-1:0] o_y
);
//Output value could overflow (32-bit output, and 16-bit inputs multiplied
//together repeatedly).  Don't worry about that -- assume that only the
↪  bottom
//32 bits are of interest, and keep them.
logic [WIDTHIN-1:0] x_m0_in;        // Register to hold input X, as input
↪  to m0
logic [WIDTHIN-1:0] x_m0_hold;// To extend propagation chain
logic [WIDTHIN-1:0] x_m1_in;        // Pipeline register for x, as input to
↪  m1
logic [WIDTHIN-1:0] x_m1_hold;
logic [WIDTHIN-1:0] x_m2_in;        // Pipeline register for x, as input to
↪  m2
logic [WIDTHIN-1:0] x_m2_hold;
```

```
logic [WIDTHIN-1:0] x_m3_in;        // Pipeline register for x, as input to
↪  m3
logic [WIDTHIN-1:0] x_m3_hold;
logic [WIDTHIN-1:0] x_m4_in;        // Pipeline register for x, as input to
↪  m4

logic [WIDTHOUT-1:0] m0_out_a0_in_pipe;        // Pipeline register after
↪  m0
logic [WIDTHOUT-1:0] a0_out_m1_in_pipe;        // Pipeline register after
↪  a0
logic [WIDTHOUT-1:0] m1_out_a1_in_pipe;        // Pipeline register after
↪  m1
logic [WIDTHOUT-1:0] a1_out_m2_in_pipe;        // Pipeline register after
↪  a1
logic [WIDTHOUT-1:0] m2_out_a2_in_pipe;        // Pipeline register after
↪  m2
logic [WIDTHOUT-1:0] a2_out_m3_in_pipe;        // Pipeline register after
↪  a2
logic [WIDTHOUT-1:0] m3_out_a3_in_pipe;        // Pipeline register after
↪  m3
logic [WIDTHOUT-1:0] a3_out_m4_in_pipe;        // Pipeline register after
↪  a3
logic [WIDTHOUT-1:0] m4_out_a4_in_pipe;        // Pipeline register after
↪  m4
logic [WIDTHOUT-1:0] a4_out_final;                        // Register to
↪  hold output Y, a4_out goes directly into y_Q

logic valid_input_x;                        // Output of register x is
↪  valid
logic valid_m0;                             // Output of m0 is valid
logic valid_a0;                             // Output of a0 is valid
logic valid_m1;                             // Output of m1 is valid
logic valid_a1;                             // Output of a1 is valid
logic valid_m2;                             // Output of m2 is valid
logic valid_a2;                             // Output of a2 is valid
logic valid_m3;                             // Output of m3 is valid
logic valid_a3;                             // Output of a3 is valid
logic valid_m4;                             // Output of m4 is valid
logic valid_a4_final;                // Output of a4 (final result) is
↪  valid

// signal for enabling sequential circuit elements
```

```
logic enable;

// Signals for computing the y output
logic [WIDTHOUT-1:0] m0_out; // A5 * x
logic [WIDTHOUT-1:0] a0_out; // A5 * x + A4
logic [WIDTHOUT-1:0] m1_out; // (A5 * x + A4) * x
logic [WIDTHOUT-1:0] a1_out; // (A5 * x + A4) * x + A3
logic [WIDTHOUT-1:0] m2_out; // ((A5 * x + A4) * x + A3) * x
logic [WIDTHOUT-1:0] a2_out; // ((A5 * x + A4) * x + A3) * x + A2
logic [WIDTHOUT-1:0] m3_out; // (((A5 * x + A4) * x + A3) * x + A2) * x
logic [WIDTHOUT-1:0] a3_out; // (((A5 * x + A4) * x + A3) * x + A2) * x +
↪  A1
logic [WIDTHOUT-1:0] m4_out; // ((((A5 * x + A4) * x + A3) * x + A2) * x +
↪  A1) * x
logic [WIDTHOUT-1:0] a4_out; // ((((A5 * x + A4) * x + A3) * x + A2) * x +
↪  A1) * x + A0

// compute y value
mult16x16 Mult0 (.i_dataa(A5),                .i_datab(x_m0_in),
↪          .o_res(m0_out));
addr32p16 Addr0 (.i_dataa(m0_out_a0_in_pipe),        .i_datab(A4),
↪          .o_res(a0_out));

mult32x16 Mult1 (.i_dataa(a0_out_m1_in_pipe),        .i_datab(x_m1_in),
↪          .o_res(m1_out));
addr32p16 Addr1 (.i_dataa(m1_out_a1_in_pipe),        .i_datab(A3),
↪          .o_res(a1_out));

mult32x16 Mult2 (.i_dataa(a1_out_m2_in_pipe),        .i_datab(x_m2_in),
↪          .o_res(m2_out));
addr32p16 Addr2 (.i_dataa(m2_out_a2_in_pipe),        .i_datab(A2),
↪          .o_res(a2_out));

mult32x16 Mult3 (.i_dataa(a2_out_m3_in_pipe),        .i_datab(x_m3_in),
↪          .o_res(m3_out));
addr32p16 Addr3 (.i_dataa(m3_out_a3_in_pipe),        .i_datab(A1),
↪          .o_res(a3_out));

mult32x16 Mult4 (.i_dataa(a3_out_m4_in_pipe),        .i_datab(x_m4_in),
↪          .o_res(m4_out));
addr32p16 Addr4 (.i_dataa(m4_out_a4_in_pipe),        .i_datab(A0),
↪          .o_res(a4_out));
```

```
// Combinational logic
always_comb begin
        // signal for enable
        enable = i_ready;
end

// Infer the registers
always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
                x_m0_in <= 0;
                x_m0_hold <= 0;
                x_m1_in <= 0;
                x_m1_hold <= 0;
                x_m2_in <= 0;
                x_m2_hold <= 0;
                x_m3_in <= 0;
                x_m3_hold <= 0;
                x_m4_in <= 0;

                valid_input_x <= 1'b0;
                valid_m0 <= 1'b0;
                valid_a0 <= 1'b0;
                valid_m1 <= 1'b0;
                valid_a1 <= 1'b0;
                valid_m2 <= 1'b0;
                valid_a2 <= 1'b0;
                valid_m3 <= 1'b0;
                valid_a3 <= 1'b0;
                valid_m4 <= 1'b0;
                valid_a4_final <= 1'b0;

                m0_out_a0_in_pipe <= 1'b0;
                a0_out_m1_in_pipe <= 1'b0;
                m1_out_a1_in_pipe <= 1'b0;
                a1_out_m2_in_pipe <= 1'b0;
                m2_out_a2_in_pipe <= 1'b0;
                a2_out_m3_in_pipe <= 1'b0;
                m3_out_a3_in_pipe <= 1'b0;
                a3_out_m4_in_pipe <= 1'b0;
                m4_out_a4_in_pipe <= 1'b0;
```

```
                a4_out_final <= 1'b0;

        end else if (enable) begin
                // As long as circuit is operating
                // read in new x value and propagate
                x_m0_in <= i_x;
                x_m0_hold <= x_m0_in;
                x_m1_in <= x_m0_hold;
                x_m1_hold <= x_m1_in;
                x_m2_in <= x_m1_hold;
                x_m2_hold <= x_m2_in;
                x_m3_in <= x_m2_hold;
                x_m3_hold <= x_m3_in;
                x_m4_in <= x_m3_hold;

                // propagate the valid value
                valid_input_x <= i_valid;
                valid_m0 <= valid_input_x;
                valid_a0 <= valid_m0;
                valid_m1 <= valid_a0;
                valid_a1 <= valid_m1;
                valid_m2 <= valid_a1;
                valid_a2 <= valid_m2;
                valid_m3 <= valid_a2;
                valid_a3 <= valid_m3;
                valid_m4 <= valid_a3;
                valid_a4_final <= valid_m4;

                // update outs
                m0_out_a0_in_pipe <= m0_out;
                a0_out_m1_in_pipe <= a0_out;
                m1_out_a1_in_pipe <= m1_out;
                a1_out_m2_in_pipe <= a1_out;
                m2_out_a2_in_pipe <= m2_out;
                a2_out_m3_in_pipe <= a2_out;
                m3_out_a3_in_pipe <= m3_out;
                a3_out_m4_in_pipe <= a3_out;
                m4_out_a4_in_pipe <= m4_out;
                a4_out_final <= a4_out;

        end
end
```

```
// assign outputs
assign o_y = a4_out_final;
// ready for inputs as long as receiver is ready for outputs */
assign o_ready = i_ready;
// the output is valid as long as the corresponding input was valid and
//        the receiver is ready. If the receiver isn't ready, the computed
↪  output
//        will still remain on the register outputs and the circuit will
↪  resume
//  normal operation with the receiver is ready again (i_ready is high)*/
assign o_valid = valid_a4_final & i_ready;

endmodule

/*****************************************************************************

// Multiplier module for the first 16x16 multiplication
module mult16x16 (
        input  [15:0] i_dataa,
        input  [15:0] i_datab,
        output [31:0] o_res
);

logic [31:0] result;

always_comb begin
        result = i_dataa * i_datab;
end

// The result of Q2.14 x Q2.14 is in the Q4.28 format. Therefore we need
↪  to change it
// to the Q7.25 format specified in the assignment by shifting right and
↪  padding with zeros.
assign o_res = {3'b000, result[31:3]};

endmodule

/*****************************************************************************

// Multiplier module for all the remaining 32x16 multiplications
module mult32x16 (
```

```
        input  [31:0] i_dataa,
        input  [15:0] i_datab,
        output [31:0] o_res
);

logic [47:0] result;

always_comb begin
        result = i_dataa * i_datab;
end

// The result of Q7.25 x Q2.14 is in the Q9.39 format. Therefore we need
↪  to change it
// to the Q7.25 format specified in the assignment by selecting the
↪  appropriate bits
// (i.e. dropping the most-significant 2 bits and least-significant 14
↪  bits).
assign o_res = result[45:14];

endmodule

/*****************************************************************************

// Adder module for all the 32b+16b addition operations
module addr32p16 (
        input [31:0] i_dataa,
        input [15:0] i_datab,
        output [31:0] o_res
);

// The 16-bit Q2.14 input needs to be aligned with the 32-bit Q7.25 input
↪  by zero padding
assign o_res = i_dataa + {5'b00000, i_datab, 11'b00000000000};

endmodule

/*****************************************************************************
```

Appendix F: .sv file for shared hardware circuit

```
module lab1_fsm #
(
        parameter WIDTHIN = 16,                 // Input format is Q2.14 (2
        ↪  integer bits + 14 fractional bits = 16 bits)
        parameter WIDTHOUT = 32,        // Intermediate/Output format is
        ↪  Q7.25 (7 integer bits + 25 fractional bits = 32 bits)
        // Taylor coefficients for the first five terms in Q2.14 format
        parameter [WIDTHIN-1:0] A0 = 16'b01_00000000000000, // a0 = 1
        parameter [WIDTHIN-1:0] A1 = 16'b01_00000000000000, // a1 = 1
        parameter [WIDTHIN-1:0] A2 = 16'b00_10000000000000, // a2 = 1/2
        parameter [WIDTHIN-1:0] A3 = 16'b00_00101010101010, // a3 = 1/6
        parameter [WIDTHIN-1:0] A4 = 16'b00_00001010101010, // a4 = 1/24
        parameter [WIDTHIN-1:0] A5 = 16'b00_00000010001000  // a5 = 1/120
)
(
        input clk,
        input reset,

        input i_valid,
        input i_ready,
        output o_valid,
        output o_ready,

        input [WIDTHIN-1:0] i_x,
        output [WIDTHOUT-1:0] o_y
);

// FSM control signals
logic store_x;
logic store_y;

logic mmux;
logic [2:0] amux;

// Register declarations
logic [WIDTHIN-1:0] x_reg;
logic [WIDTHOUT-1:0] y_reg; // Doubles as register for intermediate steps
↪  of the calculation

// Wires
```

```
logic [WIDTHOUT-1:0] m_in;
logic [WIDTHOUT-1:0] m_out;
logic [WIDTHIN-1:0] a_in;
logic [WIDTHOUT-1:0] a_out;


fsm fsm0(.clk(clk), .reset(reset), .i_valid(i_valid), .i_ready(i_ready),
                                .o_valid(o_valid), .o_ready(o_ready),
                                ↪  .store_x(store_x), .store_y(store_y),
                                ↪  .mmux(mmux), .amux(amux));

// Multiplexers for inputs to multiplier and adder
mux2x1 mux2x1_mmux(.sel(mmux), .a({5'b0, A5, 11'b0}), .b(y_reg),
↪  .out(m_in));
mux8x1 mux8x1_amux(.sel(amux), .a(A4), .b(A3), .c(A2), .d(A1), .e(A0),
↪  .f(16'b0), .g(16'b0), .h(16'b0), .out(a_in));

// Only one set of mulitplier and adder
mult32x16 Mult1 (.i_dataa(m_in),         .i_datab(x_reg),
↪                  .o_res(m_out));
addr32p16 Addr1 (.i_dataa(m_out),         .i_datab(a_in),
↪          .o_res(a_out));

// Registers
always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
                x_reg <= 0;
                y_reg <= 0;
        end else if (store_x) begin
                x_reg <= i_x;
        end else if (store_y) begin
                y_reg <= a_out;
        end
end

assign o_y = y_reg;

endmodule

/*****************************************************************************

module fsm(
```

```
        input clk,
        input reset,

        input i_valid,
        input i_ready,
        output logic o_valid,
        output logic o_ready,

        output logic store_x,
        output logic store_y,
        output logic mmux,                  //        Mux control signal for
        ↪  multiplier
        output logic [2:0] amux // Mux control signal for adder
);

typedef enum logic [2:0] { input_x, compute_0, compute_1, compute_2,
↪  compute_3, compute_4, output_y } State;
State currentState, nextState;

logic input_valid;
logic enable;

// State register updates
always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
                currentState <= input_x;
        end else begin
                currentState <= nextState;        // Logic to check for
                ↪  enable/i_ready in comb
        end
end

// Some signal assignments
always_comb begin
        input_valid = i_valid;
        enable = i_ready;
end

// FSM next state computation
always_comb begin
        case(currentState)
```

```
                input_x: if(input_valid) begin        // input_x state
                ↪  waits for valid input
                        nextState = compute_0;
                end else begin
                        nextState = input_x;
                end
                compute_0: nextState = compute_1;         // compute_<x>
                ↪  states computes the corresponding o_y regardless if
                compute_1: nextState = compute_2;        // ready to output
                ↪  to downstream or not
                compute_2: nextState = compute_3;
                compute_3: nextState = compute_4;
                compute_4: nextState = output_y;
                output_y: if(enable) begin                          // o_y is
                ↪  computed, o_valid is toggled, waits for i_ready before
                ↪  moving on
                        nextState = input_x;
                end else begin
                        nextState = output_y;
                end
        endcase
end

// FSM control signals
always_comb begin
        // Base case
        o_valid = 1'b0;
        o_ready = 1'b0;

        store_x = 1'b0;                         // Controls x_reg
        store_y = 1'b0;                         // Controls y_reg

        mmux = 1'b0;                                // Controls multiplier
        ↪  mux
        amux = 3'b000;                               // Controls adder mux

        case(currentState)
                input_x: begin
                        o_valid = 1'b0;         // Repeating for clarity,
                        ↪  output not ready
                        o_ready = 1'b1;         // Ready for new input
                        store_x = 1'b1;         // Store x into register
```

```
end
compute_0: begin
        mmux = 1'b0;                    // Repeating for
        ↪ clarity, selecting a5 for multiplier, a5 padded
        ↪ to Q7.25
        amux = 3'b000;                  // Selecting a4 for
        ↪ adder
        store_y = 1'b1;         // Store intermediate result
        ↪ into register
end
compute_1: begin
        mmux = 1'b1;                    // Selecting previously
        ↪ computed value for multiplier
        amux = 3'b001;                  // Selecting a3 for
        ↪ adder
        store_y = 1'b1;         // Store intermediate result
        ↪ into register
end
compute_2: begin
        mmux = 1'b1;                    // Selecting previously
        ↪ computed value for multiplier
        amux = 3'b010;                  // Selecting a2 for
        ↪ adder
        store_y = 1'b1;         // Store intermediate result
        ↪ into register
end
compute_3: begin
        mmux = 1'b1;                    // Selecting previously
        ↪ computed value for multiplier
        amux = 3'b011;                  // Selecting a1 for
        ↪ adder
        store_y = 1'b1;         // Store intermediate result
        ↪ into register
end
compute_4: begin
        mmux = 1'b1;                    // Selecting previously
        ↪ computed value for multiplier
        amux = 3'b100;                  // Selecting a0 for
        ↪ adder
        store_y = 1'b1;         // Store final result into
        ↪ register
end
```

```
                output_y: begin
                        o_valid = 1'b1;         // Output ready
                        o_ready = 1'b0;         // Repeating for clarity.
                         ↪  not ready for new input
                end
        endcase
end

endmodule

/*****************************************************************************

module mux2x1 (
        input sel,
        input [31:0] a,
        input [31:0] b,

        output logic [31:0] out
);

always_comb begin
        case (sel)
                1'b0: out = a;
                1'b1: out = b;
        endcase
end

endmodule

/*****************************************************************************

module mux8x1 (
        input [2:0] sel,
        input [15:0] a,
        input [15:0] b,
        input [15:0] c,
        input [15:0] d,
        input [15:0] e,
        input [15:0] f,
        input [15:0] g,
        input [15:0] h,
```

```
        output logic [15:0] out
);

always_comb begin
        case (sel[2:0])
                3'b000: out = a;
                3'b001: out = b;
                3'b010: out = c;
                3'b011: out = d;
                3'b100: out = e;
                3'b101: out = f;
                3'b110: out = g;
                3'b111: out = h;
        endcase
end

endmodule

/*******************************************************************************

//// Multiplier module for the first 16x16 multiplication
//module mult16x16 (
//        input  [15:0] i_dataa,
//        input  [15:0] i_datab,
//        output [31:0] o_res
//);
//
//logic [31:0] result;
//
//always_comb begin
//        result = i_dataa * i_datab;
//end
//
//// The result of Q2.14 x Q2.14 is in the Q4.28 format. Therefore we need
 ↪  to change it
//// to the Q7.25 format specified in the assignment by shifting right and
 ↪  padding with zeros.
//assign o_res = {3'b000, result[31:3]};
//
//endmodule

/*******************************************************************************
```

```
// Multiplier module for all the remaining 32x16 multiplications
module mult32x16 (
        input  [31:0] i_dataa,
        input  [15:0] i_datab,
        output [31:0] o_res
);

logic [47:0] result;

always_comb begin
        result = i_dataa * i_datab;
end

// The result of Q7.25 x Q2.14 is in the Q9.39 format. Therefore we need
↪  to change it
// to the Q7.25 format specified in the assignment by selecting the
↪  appropriate bits
// (i.e. dropping the most-significant 2 bits and least-significant 14
↪  bits).
assign o_res = result[45:14];

endmodule

/*****************************************************************************

// Adder module for all the 32b+16b addition operations
module addr32p16 (
        input [31:0] i_dataa,
        input [15:0] i_datab,
        output [31:0] o_res
);

// The 16-bit Q2.14 input needs to be aligned with the 32-bit Q7.25 input
↪  by zero padding
assign o_res = i_dataa + {5'b00000, i_datab, 11'b00000000000};

endmodule

/*****************************************************************************
```