

1 Lab1 Implementation

In this assignment we are asked to implement the Taylor expansion which is a relatively simple and approximate version of e^x . The Taylor expansion approximates a function with n degree polynomial and the higher values of 'n' are directly proportional to precision or correctness of the calculation. We are to consider expansion term only until the 5th Taylor polynomial, the exponential function is given by

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} \dots$$

e^x is a very common function that is used very much in Machine learning concepts, we shall create a circuit based on the Taylor polynomial that can calculate the value for us.

2 Part 1 - Pipelining

In part 1 we try to make our model reduce its slack time. Currently the circuit has a slack time of 20.415ns and our fastest cycle time is $1 + 20.415\text{ns} = 21.415\text{ns}$ at 100°C, sometimes the slowest speeds occur at 0°C and in the baseline implementation it gives us a slack time of 21.311ns and our fastest cycle time is 22.311ns.

In part 1 we shall try to reduce this with the help of pipelining achieved using D-Flip Flops. We add the flip flops before every consecutive 32x16 multiplication operation and one after every add operation. We add a flip flop at the initial stage right after X register. As explained we use a total of 9 flip flops to achieve higher throughput and lesser slack time.

For our implementation we take the 6 basic input and output signals and also store our values as shown below:

1. i_x : Provides the value of x we use to compute
2. o_y : Provides the output of x after computation
3. i_valid : When its 1 it signals that its a valid value
4. i_ready : When its 1 it indicates the downstream circuit is ready to accept output
5. o_valid : Its 1 when the circuit outputs a valid o_y
6. o_ready : Its 1 to signal that we are ready for a new x value
7. A0=1 ; A1=1 ; A2=1/2 ; A3=1/6 ; A4=1/24 ; A5=1/120

2.1 Appending Flip Flops

In order for us to append flip flops we need to first create a module in our system verilog file (.sv). Then we call the module at connection points described initially. The code for the D-Flip Flop is shown in the below image. Fig 1

```

/*****
//D-flipflop
module Dflipflop #(parameter width = 32)
(
    input [width-1:0] d,
    input clk,
    input rst,
    input en,
    output logic [width-1:0] q
);

    always_ff @(posedge clk)
    begin
        if (rst)
            q <= 0;
        else if (en)
            q <= d;
    end
endmodule

```

Figure 1: D-FlipFlop module declaration

After we create the module we call the module at where we need the value to be stored. We can see in the next diagram where we call the module and how to circuit looks in Fig 2

```

lab1.sv
67 // compute y value
68 Dflipflop #(width(WIDTHIN)) flip1 (.clk(clk), .rst(reset), .en(enable), .d(x), .q(d0_out));
69 mult16x16 Mult0 (.i_dataa(A5), .i_datab(d0_out), .o_res(m0_out));
70 addr32p16 Addr0 (.i_dataa(m0_out), .i_datab(A4), .o_res(a0_out));
71 Dflipflop #(width(WIDTHOUT)) flip2 (.clk(clk), .rst(reset), .en(enable), .d(a0_out), .q(d1_out));
72
73 Dflipflop #(width(WIDTHIN)) flip3 (.clk(clk), .rst(reset), .en(enable), .d(d0_out), .q(d2_out));
74 mult32x16 Mult1 (.i_dataa(d1_out), .i_datab(d2_out), .o_res(m1_out));
75 addr32p16 Addr1 (.i_dataa(m1_out), .i_datab(A3), .o_res(a1_out));
76 Dflipflop #(width(WIDTHOUT)) flip4 (.clk(clk), .rst(reset), .en(enable), .d(a1_out), .q(d3_out));
77
78 Dflipflop #(width(WIDTHIN)) flip5 (.clk(clk), .rst(reset), .en(enable), .d(d2_out), .q(d4_out));
79 mult32x16 Mult2 (.i_dataa(d3_out), .i_datab(d4_out), .o_res(m2_out));
80 addr32p16 Addr2 (.i_dataa(m2_out), .i_datab(A2), .o_res(a2_out));
81 Dflipflop #(width(WIDTHOUT)) flip6 (.clk(clk), .rst(reset), .en(enable), .d(a2_out), .q(d5_out));
82
83 Dflipflop #(width(WIDTHIN)) flip7 (.clk(clk), .rst(reset), .en(enable), .d(d4_out), .q(d6_out));
84 mult32x16 Mult3 (.i_dataa(d5_out), .i_datab(d6_out), .o_res(m3_out));
85 addr32p16 Addr3 (.i_dataa(m3_out), .i_datab(A1), .o_res(a3_out));
86 Dflipflop #(width(WIDTHOUT)) flip8 (.clk(clk), .rst(reset), .en(enable), .d(a3_out), .q(d7_out));
87
88 Dflipflop #(width(WIDTHIN)) flip9 (.clk(clk), .rst(reset), .en(enable), .d(d6_out), .q(d8_out));
89 mult32x16 Mult4 (.i_dataa(d7_out), .i_datab(d8_out), .o_res(m4_out));
90 addr32p16 Addr4 (.i_dataa(m4_out), .i_datab(A0), .o_res(a4_out));
91
92 assign y_D = a4_out;
93

```

Figure 2: D-FlipFlop module calling

Fig 3 shows how the flip flops are placed in the circuit. The green rectangles are the flip flops and the green squares are the adder and multiplication block.

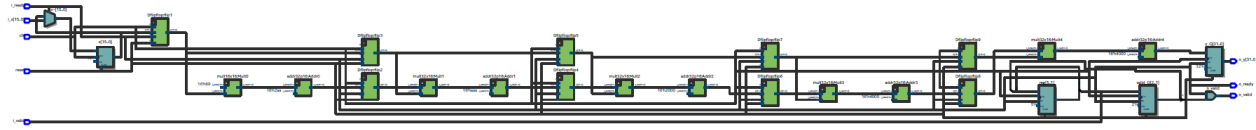


Figure 3: RTL View

2.2 Simulation and Verification

To check if our model works without any issues, we simulate it in Modelsim and see if all the values provided by the test bench receive the correct output and this can be visualised using graph simulations or also as transcript output. In our simulated waveform as per Fig 4 the out o_y takes 192ns to provide the first valid output. According to simulation we can see that o_y provides output, to check if they are correct output we can verify that against our expected values as shown in Fig 5. The transcript prints the simulated output and verifies it with respect to the expected and also calculates the error deviation rate

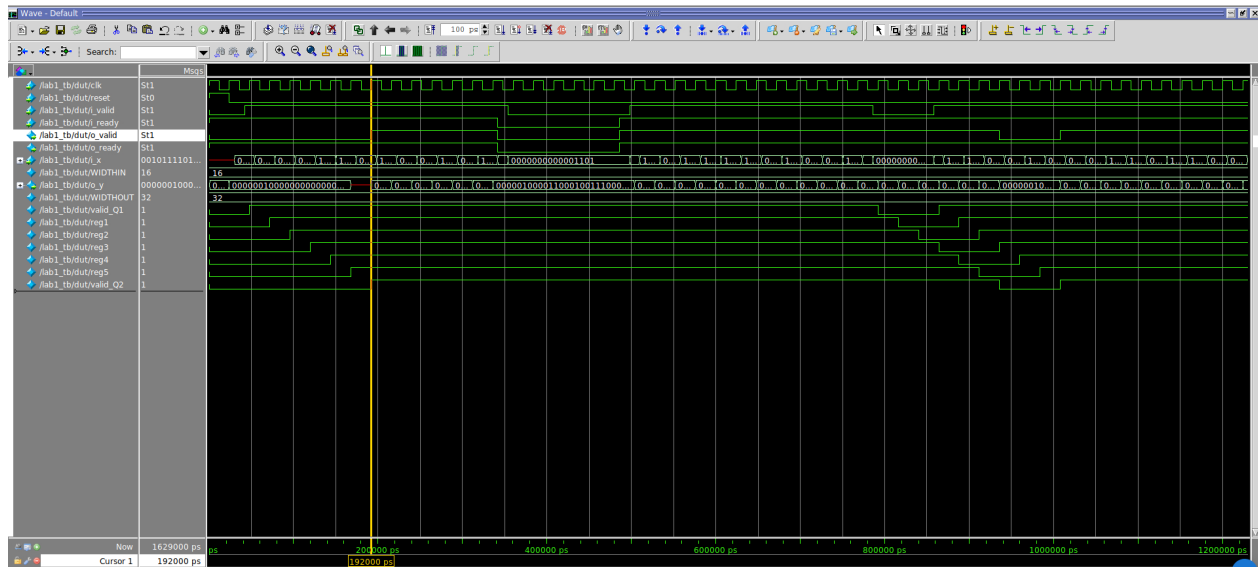


Figure 4: Simulated waveform

We then compare our model performance vs the actual calculated e^x and this is shown in Fig 6, where our output (marked as x) is very close to the output produced by the exponent term. To make even more precise we can add more number of states, in this pipeline model we use only 5, but adding another add and multiply block would make our plot even more close to the exponential plot.

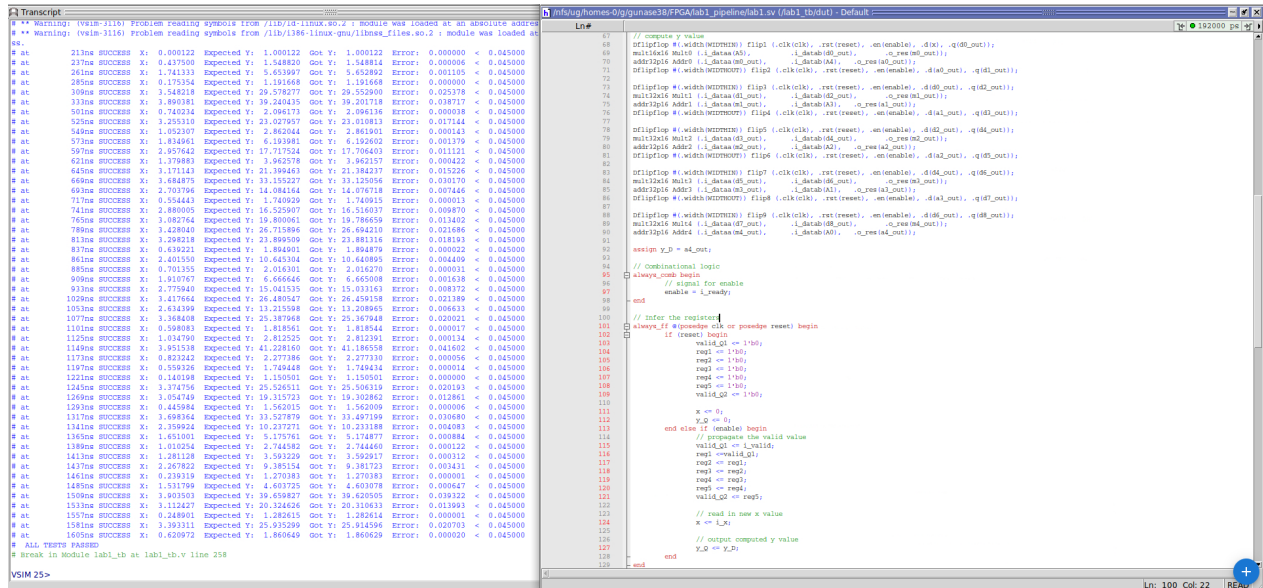


Figure 5: Test bench output

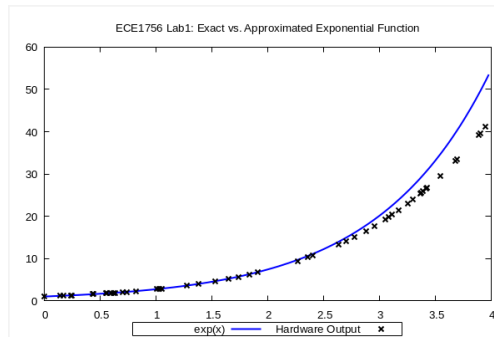


Figure 6: e^x vs Hardware

2.3 Resource Utilization

We can observe the amount of resources we have used , we can observe this in Quartus using Fitter resource utilization data sheet, in Fig 7 we can use how many resource have been utilized for the pipe lining version.We have used 119 registers, 252 LUT , 9 DSP blocks, when we compare our model to baseline we see that we have significantly used more LUT and registers and the same number of DSP blocks. Fig 8 gives us the summary of the usage We have used a total of 153 logic modules and of those 27 are used for virtual pins 126 are used for logic based operations.

The number of times we can replicate this model in the given FPGA is calculated by $427,200/126 = 3390$ times

and the number of DSP that can run on the same FPGA is given by $1518/9=168$.

Fitter Resource Utilization by Entity													
<<Filter>>													
	Compilation Hierarchy Node	ALMs needed [=A+B+C]	Combinational ALUTs	Dedicated Logic Registers	I/O Registers	Block Memory Bits	M20Ks	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name	Entity Name	Library Name
1	lab1	152.5 (32.7)	252 (1)	119 (55)	0 (0)	0	0	9	1	53	lab1		
1	[Dflipflop:flip1]	8.5 (8.5)	17 (17)	16 (16)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip1]	Dflipflop	work
2	[Dflipflop:flip2]	15.5 (15.5)	32 (32)	0 (0)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip2]	Dflipflop	work
3	[Dflipflop:flip3]	8.0 (8.0)	16 (16)	16 (16)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip3]	Dflipflop	work
4	[Dflipflop:flip4]	15.8 (15.8)	32 (32)	0 (0)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip4]	Dflipflop	work
5	[Dflipflop:flip5]	8.0 (8.0)	16 (16)	16 (16)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip5]	Dflipflop	work
6	[Dflipflop:flip6]	16.0 (16.0)	32 (32)	0 (0)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip6]	Dflipflop	work
7	[Dflipflop:flip7]	8.0 (8.0)	16 (16)	16 (16)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip7]	Dflipflop	work
8	[Dflipflop:flip8]	13.0 (13.0)	32 (32)	0 (0)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip8]	Dflipflop	work
9	[Dflipflop:flip9]	4.0 (4.0)	16 (16)	0 (0)	0 (0)	0	0	0	0	0	lab1[Dflipflop:flip9]	Dflipflop	work
10	[addr32p16:Addr1]	9.0 (9.0)	20 (20)	0 (0)	0 (0)	0	0	0	0	0	lab1[addr32p16:Addr1]	addr32p16	work
11	[addr32p16:Addr2]	4.0 (4.0)	8 (8)	0 (0)	0 (0)	0	0	0	0	0	lab1[addr32p16:Addr2]	addr32p16	work
12	[addr32p16:Addr3]	2.0 (2.0)	7 (7)	0 (0)	0 (0)	0	0	0	0	0	lab1[addr32p16:Addr3]	addr32p16	work
13	[addr32p16:Addr4]	1.8 (1.8)	7 (7)	0 (0)	0 (0)	0	0	0	0	0	lab1[addr32p16:Addr4]	addr32p16	work
14	[mult16x16:Mult0]	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	1	0	0	lab1[mult16x16:Mult0]	mult16x16	work
15	[mult32x16:Mult1]	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2	0	0	lab1[mult32x16:Mult1]	mult32x16	work
16	[mult32x16:Mult2]	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2	0	0	lab1[mult32x16:Mult2]	mult32x16	work
17	[mult32x16:Mult3]	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2	0	0	lab1[mult32x16:Mult3]	mult32x16	work
18	[mult32x16:Mult4]	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2	0	0	lab1[mult32x16:Mult4]	mult32x16	work

Figure 7: Resource utilization by entity

2.4 Critical Path

We know from above that at 0 is when there is more delay observed we shall also monitor it for the same criteria In Fig 9 we calculate the clock delay between the source and destination register. Which is given by $4.487 - 4.116 = 0.371\text{ns}$

The data path delay is given as 6.235ns which is lesser than baseline implementation

Based on the critical path of the circuit in Fig 10 we see that

- Mult1 has 2 DSP blocks
- Addr1 created by logic elements
- D_flipflop created by 1 logic element
- one more DSP for Mult2

2.5 Power Dissipation

To estimate power we need to analyse the toggles rate of each signal To get an accurate value we perform gate level or timing simulation using Modelsim. We get a .vcd file from Modelsim when we run the EDA netlist writer in Quartus and then run gate level simulation in Modelsim.

Fitter Resource Usage Summary			
<<Filter>>			
	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	153 / 427,200	< 1 %
2	▾ ALMs needed [=A-B+C]	153	
1	▾ [A] ALMs used in final placement [=a+b+c+d]	163 / 427,200	< 1 %
1	[a] ALMs used for LUT logic and registers	23	
2	[b] ALMs used for LUT logic	104	
3	[c] ALMs used for registers	36	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	37 / 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			
4	Difficulty packing design	Low	
5			
6	▾ Total LABs: partially or completely used	37 / 42,720	< 1 %
1	-- Logic LABs	37	
2	-- Memory LABs (up to half of total LABs)	0	
7			
8	▾ Combinational ALUT usage for logic	252	
1	-- 7 input functions	0	
2	-- 6 input functions	0	
3	-- 5 input functions	0	
4	-- 4 input functions	0	
5	-- <=3 input functions	252	
9			
10	▾ Dedicated logic registers	119	
1	▾ -- By type:		
1	-- Primary logic registers	116 / 854,400	< 1 %
2	-- Secondary logic registers	3 / 854,400	< 1 %
2	▾ -- By function:		
1	-- Design implementation registers	119	
2	-- Routing optimization registers	0	
11			
12	Virtual pins	53	
13	▾ I/O pins	1 / 992	< 1 %
1	-- Clock pins	1 / 48	2 %
2	-- Dedicated input pins	0 / 107	0 %
14			

83%

Figure 8: Resource usage

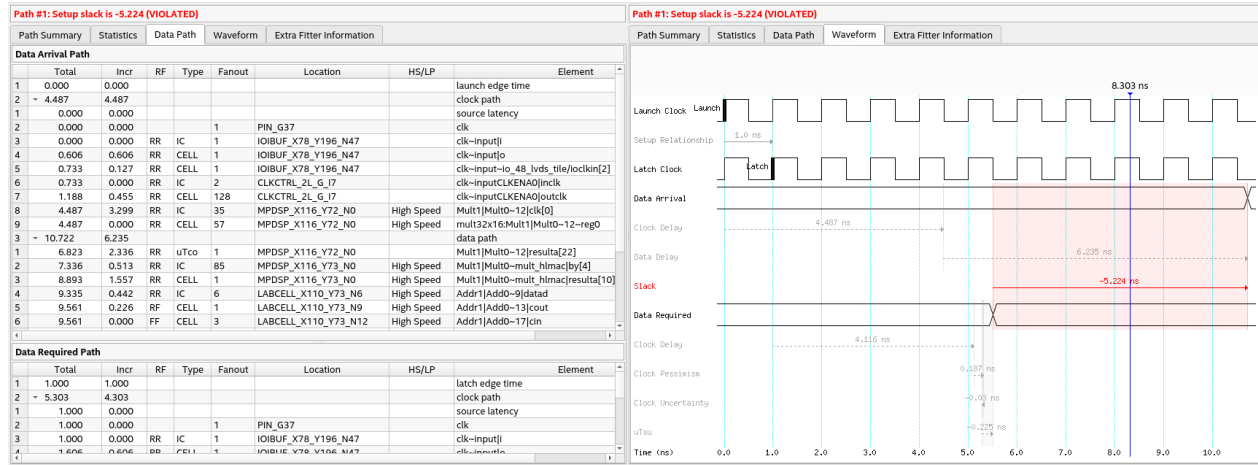


Figure 9: Critical Path Delay of circuit

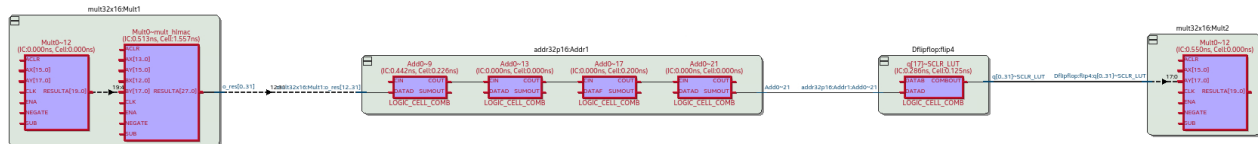
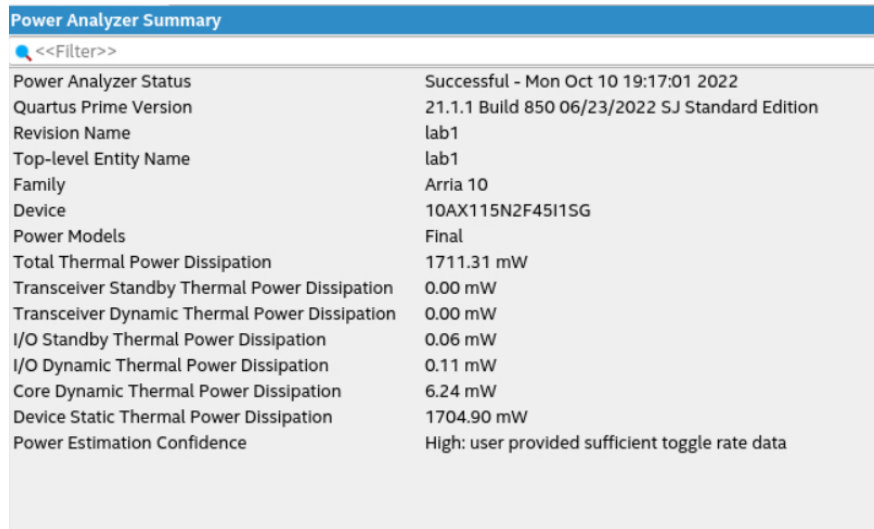


Figure 10: Critical Path of circuit

Thermal Power Dissipation by Block Type				
<<Filter>>				
Block Type	Total Thermal Power by Block Type	Block Thermal Dynamic Power	Block Thermal Standby Power	Routing Thermal Dynamic Power
1 DSP	2.65 mW	2.25 mW	0.00 mW	0.40 mW
2 Combinational cell	0.77 mW	0.29 mW	0.00 mW	0.48 mW
3 Register cell	0.19 mW	0.08 mW	0.00 mW	0.12 mW
4 IO Analog	0.12 mW	0.11 mW	0.01 mW	0.00 mW
5 IO Digital	0.05 mW	0.00 mW	0.05 mW	0.00 mW
6 Clock Network	2.62 mW	0.17 mW	0.00 mW	2.45 mW

Figure 11: Thermal power dissipated by block type



Power Analyzer Summary	
<<Filter>>	
Power Analyzer Status	Successful - Mon Oct 10 19:17:01 2022
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Standard Edition
Revision Name	lab1
Top-level Entity Name	lab1
Family	Arria 10
Device	10AX115N2F45I1SG
Power Models	Final
Total Thermal Power Dissipation	1711.31 mW
Transceiver Standby Thermal Power Dissipation	0.00 mW
Transceiver Dynamic Thermal Power Dissipation	0.00 mW
I/O Standby Thermal Power Dissipation	0.06 mW
I/O Dynamic Thermal Power Dissipation	0.11 mW
Core Dynamic Thermal Power Dissipation	6.24 mW
Device Static Thermal Power Dissipation	1704.90 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Figure 12: Power Analyzer summary

We can see in Fig 11 that clock network power usage as 2.32mW and the combined power usage of DSP, combinational cell and register cell is 3.61mW

From Fig 12 power analyzer summary, we can find the devices static power to be 1704.90mW and I/O power 0.17mW.

3 Part 2 - Shared

In part 2 we try to recreate the whole model to achieve the same output but with lesser resources and using them more than once.

We need to achieve the same output as baseline implementation but with a more efficient design that make use of one multiplication block and one adder block to achieve the output.

For our implementation we take the 6 basic input and output signals and we make use of 2 types of control signals and also store our values as shown below:

1. `i_x` : Provides the value of `x` we use to compute
2. `o_y` : Provides the output of `x` after computation
3. `i_valid` : When its 1 it signals that its a valid value
4. `i_ready` : When its 1 it indicates the downstream circuit is ready to accept output
5. `o_valid` : Its 1 when the circuit outputs a valid `o_y`
6. `o_ready` : Its 1 to signal that we are ready for a new `x` value
7. `A0=1 ; A1=1 ; A2=1/2 ; A3=1/6 ; A4=1/24 ; A5=1/120`
8. Stage : We use this to decide what the next step is
9. Control: We use this to decide what input needs to go to adder

3.1 Flow of model

Control for the model is done through 2 registers, Stage and Control. Stage is used to monitor what happens at each point in the circuit. Its the master control for the whole circuit where as control is only deciding which value should be the input for the adder. The stage also decides when to stop the operation to provide the output `o_y` and also when to ask for a new input `i_x`. It responds to the input only if `i_valid` is high.

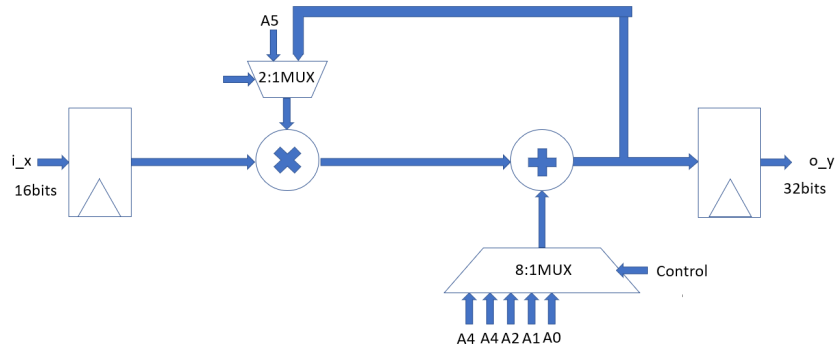


Figure 13: basic circuit diagram

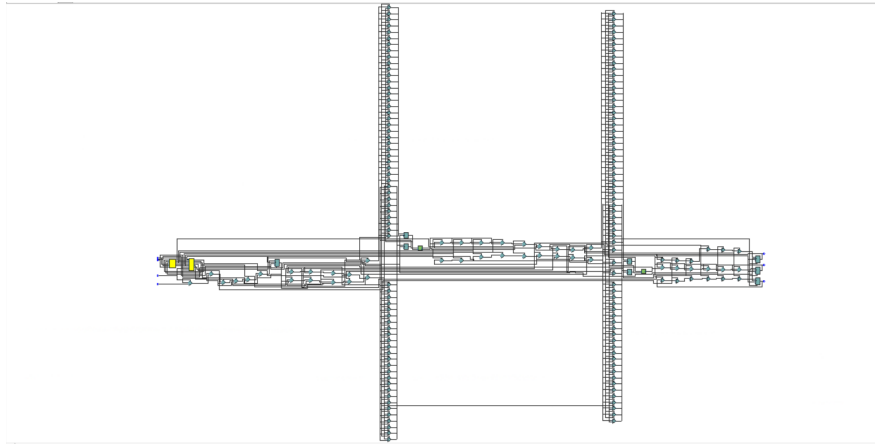


Figure 14: TRL view

3.2 Simulation and Verification

To check if our model works without any issues, we simulate it in Modelsim and see if all the values provided by the test bench receive the correct output and this can be visualised using graph simulations or also as transcript output. In our simulated waveform as per Fig 14 the out o_y takes 312ns to provide the first valid output. According to simulation we can see that o_y provides output, to check if they are correct output we can verify that against our expected values as shown in Fig 15. The transcript prints the simulated output and verifies it with respect to the expected and also calculates the error deviation rate

We then compare our model performance vs the actual calculated e^x and this is shown in Fig 16, where our output (marked as x) is very close to the output produced by the exponent term. To make even more precise we can add more number of states, in this pipeline model we use only 5, but adding another add and multiply block would make our plot even more

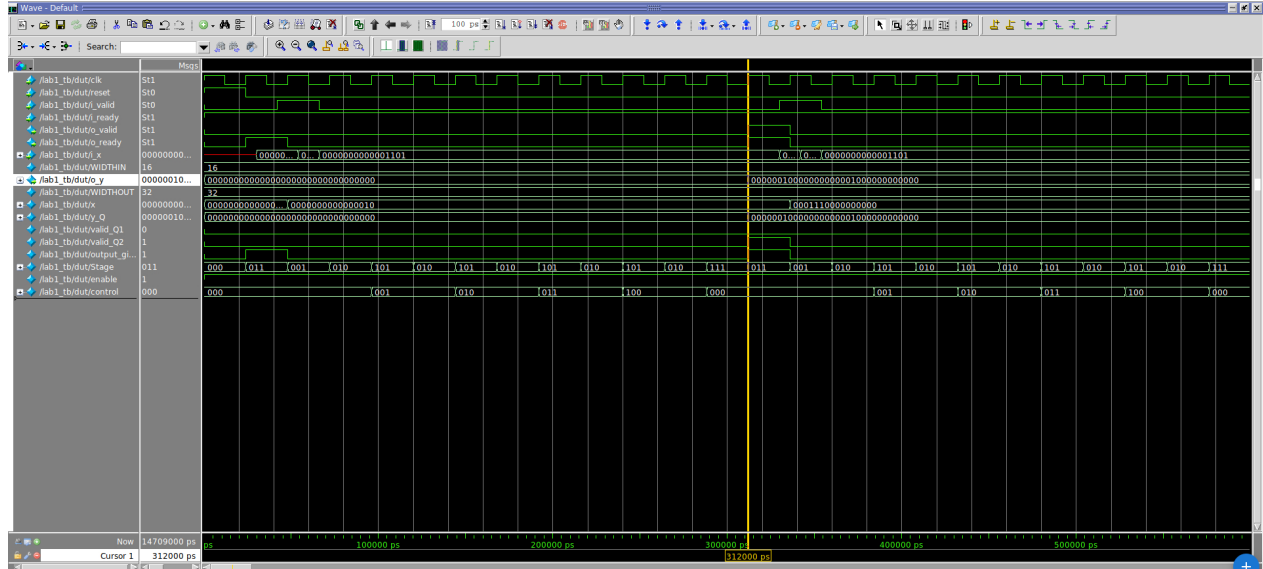


Figure 15: Simulated waveform

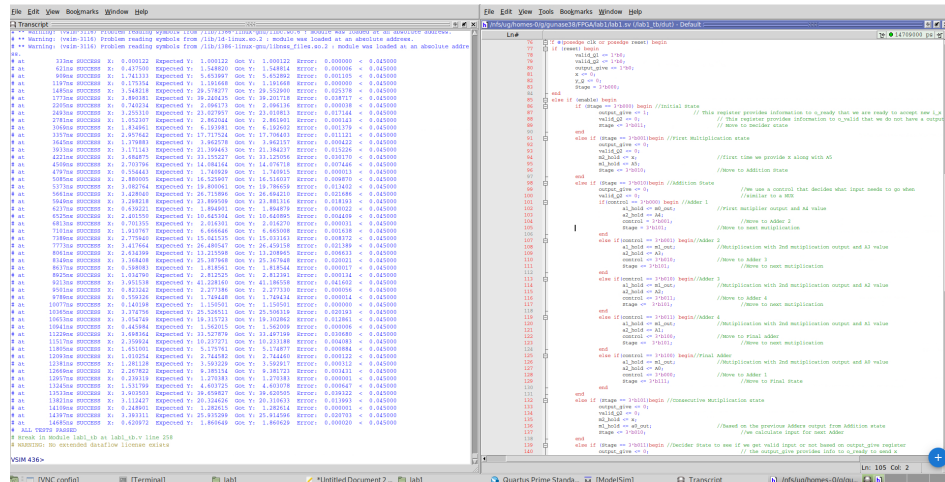


Figure 16: Testbench output

close to the exponential plot.

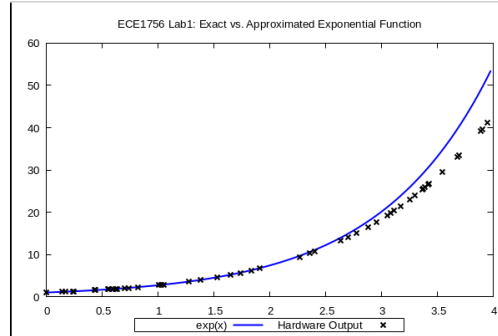


Figure 17: e^x vs Hardware

3.3 Resource Utilization

We can observe the amount of resources we have used, we can observe this in Quartus using Fitter resource utilization data sheet, in Fig 17 we can use how many resource have been utilized for the shared hardware version. We have used 97 registers, 104 LUT, 2 DSP blocks, when we compare our model to baseline we see that we have significantly used more LUT and registers and the same number of DSP blocks. Fig 18 gives us the summary of the usage

Fitter Resource Utilization by Entity								
<<Filter>>								
Compilation Hierarchy Node	ALMs needed [F=A-B+C]	[A] ALMs used in final placement	[B] Estimate of ALMs recoverable by dense packing	[C] Estimate of ALMs unavailable	ALMs used for memory	Combinational ALUTs	Dedicated	
1 - jlab1	80.0 (75.0)	65.5 (60.5)	12.0 (12.0)	26.5 (26.5)	0.0 (0.0)	104 (84)	97 (97)	
1 addr32p16:Addr0	5.0 (5.0)	5.0 (5.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	20 (20)	0 (0)	
2 mult32x16:Mult0	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0 (0)	0 (0)	

Figure 18: Resource utilization by entity

We have used a total of 80 logic modules and of those 27 are used for virtual pins 53 are used for logic based operations.

The number of times we can replicate this model in the given FPGA is calculated by $427,200/53 = 8060$ times

and the number of DSP that can run on the same FPGA is given by $1518/2=759$.

3.4 Critical Path

We know from above that at 0 is when there is more delay observed we shall also monitor it for the same criteria In Fig 19 we calculate the clock delay between the source and destination

Fitter Resource Usage Summary			
<<Filter>>			
	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	80 / 427,200	< 1 %
2	▾ ALMs needed [=A-B+C]	80	
1	▾ [A] ALMs used in final placement [=a+b+c+d]	67 / 427,200	< 1 %
1	[a] ALMs used for LUT logic and registers	35	
2	[b] ALMs used for LUT logic	21	
3	[c] ALMs used for registers	11	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	14 / 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			
4	Difficulty packing design	Low	
5			
6	▸ Total LABs: partially or completely used	7 / 42,720	< 1 %
7			
8	▸ Combinational ALUT usage for logic	104	
9			
10	▸ Dedicated logic registers	97	
11			
12	Virtual pins	53	
13	▸ I/O pins	1 / 992	< 1 %
14			
15	M20K blocks	0 / 2,713	0 %
16	Total MLAB memory bits	0	
17	Total block memory bits	0 / 55,562,240	0 %
18	Total block memory implementation bits	0 / 55,562,240	0 %
19			
100%			

Figure 19: Resource usage

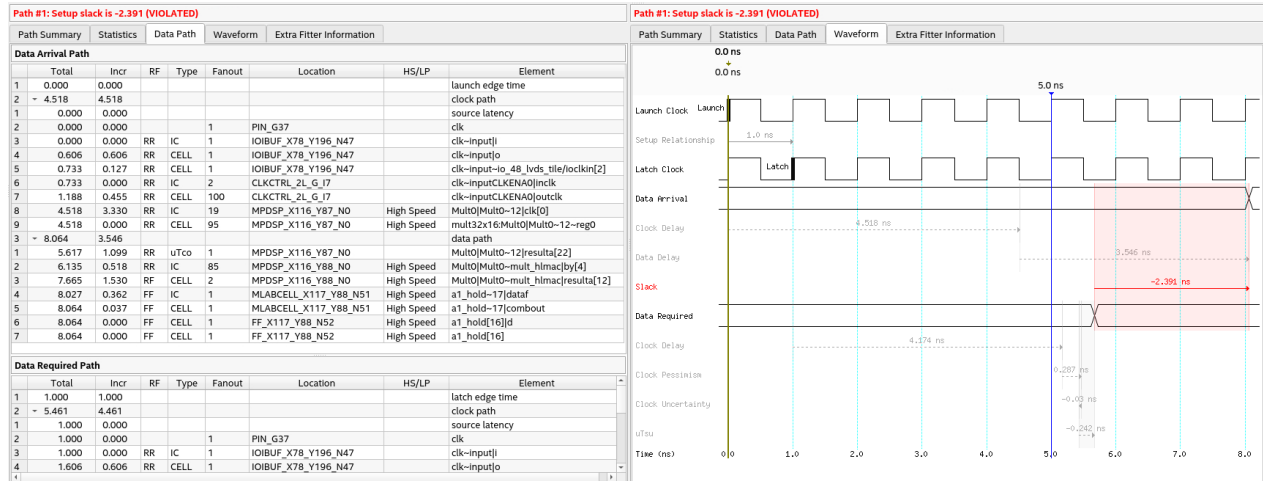


Figure 20: Critical Path Delay of circuit

register. Which is given by $4.518 - 4.174 = 0.344\text{ns}$

The data path delay is given as 3.546ns which is lesser than baseline implementation and our pipeline model

Based on the critical path of the circuit in Fig 20 we see that

- clk input
- clk controller
- output hold register cell
- and y_Q register

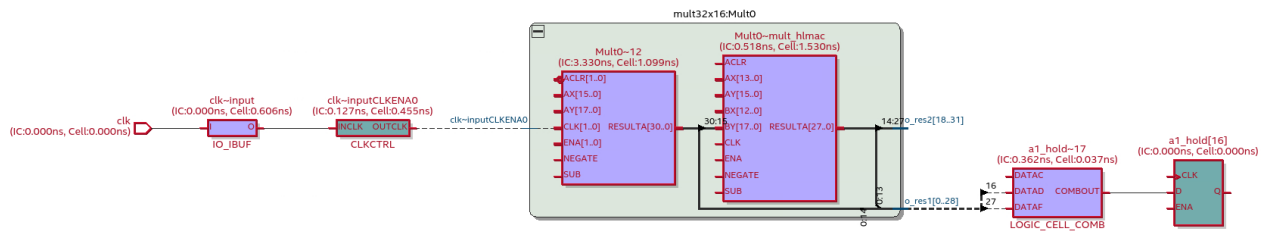


Figure 21: Critical Path of circuit

Thermal Power Dissipation by Block Type				
<<Filter>>				
	Block Type	Total Thermal Power by Block Type	Block Thermal Dynamic Power	Block Thermal Standby Power
1	DSP	0.20 mW	0.15 mW	0.00 mW
2	Combinational cell	0.17 mW	0.10 mW	0.00 mW
3	Register cell	0.10 mW	0.04 mW	0.00 mW
4	IO Analog	0.12 mW	0.11 mW	0.01 mW
5	IO Digital	0.05 mW	0.00 mW	0.05 mW
6	Clock Network	1.95 mW	0.06 mW	0.00 mW

Figure 22: Thermal power dissipated by block type

3.5 Power Dissipation

To estimate power we need to analyse the toggles rate of each signal To get an accurate value we perform gate level or timing simulation using Modelsim. We get a .vcd file from Modelsim when we run the EDA netlist writer in Quartus and then run gate level simulation in Modelsim.

We can see in Fig 11 that clock network power usage as 1.95mW and the combined power usage of DSP, combinational cell and register cell is 0.47mW

Power Analyzer Summary	
<<Filter>>	
Power Analyzer Status	Successful - Mon Oct 10 21:16:36 2022
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Standard Edition
Revision Name	lab1
Top-level Entity Name	lab1
Family	Arria 10
Device	10AX115N2F45I1SG
Power Models	Final
Total Thermal Power Dissipation	1706.45 mW
Transceiver Standby Thermal Power Dissipation	0.00 mW
Transceiver Dynamic Thermal Power Dissipation	0.00 mW
I/O Standby Thermal Power Dissipation	0.06 mW
I/O Dynamic Thermal Power Dissipation	0.11 mW
Core Dynamic Thermal Power Dissipation	2.43 mW
Device Static Thermal Power Dissipation	1703.84 mW
Power Estimation Confidence	Medium: user provided moderately complete toggle rate data

Figure 23: Power Analyzer summary

From Fig 12 power analyzer summary, we can find the devices static power to be 1703.84mW and I/O power 0.17mW.

4 Final Observations

We can analyze that when compared to pipeline model our shared model consumes lesser power dissipation and also uses lesser number of resources.

The critical path for is smaller and the delays are small when we compare them to the baseline and pipeline version. I natural to assume that this would be the case as we are using lesser compute elements and lesser resources as shown by the utilization chart.

5 Comparison Table

We can observe the slack for 0°C and 100°C for pipelining in the below image

Slow 900mV 0C Model Setup Summary			Slow 900mV 100C Model Setup Summary		
<<Filter>>			<<Filter>>		
Clock	Slack	End Point TNS	Clock	Slack	End Point TNS
1 clk	-5.338	-586.892	1 clk	-5.224	-568.729

Figure 24: slack values

We can observe the slack for 0°C and 100°C for shared Hardware in the below image

Slow 900mV 0C Model Setup Summary			Slow 900mV 100C Model Setup Summary		
<<Filter>>			<<Filter>>		
Clock	Slack	End Point TNS	Clock	Slack	End Point TNS
1 clk	-2.404	-114.696	1 clk	-2.272	-112.974

Figure 25: slack values

In Fig 28, we can see the comparison table that gives us how each model is performing

	Baseline Circuit	Pipelined Circuit	Shared HW Circuit
Resources for one circuit	21 ALM's + 9DSP's	126 ALM's + 9DSP's	53 ALM's + 2DSP's
Operating frequency	44.8MHz	157.77MHz	293MHz
Critical path	DSP mult-add+2x DSP mult +LE-based adder + 6x DSP mult +LE-based adder	2xDSP+LE based-adder +LE-based FlipFlop+DSP	I/O buffer+Clk Cntl+2xDSP +LE based-reg
Cycles per valid output	1	1	13
Max. # of copies/device	168	168	759
Max. Throughput for a one device (computations/s)	44.8M	157.77M	22.53M
Max. Throughput for a full device (computations/s)	168*44.8M=7.526G	168*157.77=26.505G	17.106G
Dynamic power of one circuit @ 42 MHz	1.96mW	3.61mW	0.47mW
Dynamic power of full circuit @ 42 MHz	1.96mW*168=329.28mW	3.61mW*168=606.48mW	356.73mW
Max. throughput/Watt for a full device	44.8M/1.96mW= 22.855GFLOPs/W	157.77M/3.61mW= 43.703GFLOPs/W	22.53M/0.47mW= 47.93GPLO's/W

Figure 26: Comparison Table

6 Discussions

(a) 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?

The sources of error are due factors such as bit sequence mix up, delay can cause error in outputs and wrong routing. These are all general sources of errors and all can be corrected in due time. But to make our circuit output more accurate we can use higher order Taylor polynomials.

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

The pipeline version has better throughput, it has higher throughput when compared to both than baseline by a factor of 3.5 times and by a factor of 7 when compared to the shared HW version. It could be due to pipelining increase throughput and also in the shared hardware version it takes more than a single cycle to provide a valid output

(c) 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.?

The pipelined circuit, which makes use of all DSPs and logic cells for each cycle, has the maximum toggle rate. The pipelined version constantly uses every block. The toggle rate is therefore higher. Similar to this, the baseline circuit has more toggle because there are 1 cycle per valid output. However, there are 13 cycles per valid output in the case of shared hardware.

Also efficiency of pipeline is better than other 2 circuits as it does it faster and with lesser power. Even though we shared hardware is faster it takes more cycles to provide output.

```

/*Jashwant Raj G Pipelined*/

module lab1 #

(
    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; // Register to hold input X
logic [WIDTHOUT-1:0] y_Q; // Register to hold output Y
logic valid_Q1; // Output of register x is valid
logic valid_Q2; // Output of register y is valid

// Registers used for i_valid signals propagation
logic reg1;
logic reg2;
logic reg3;
logic reg4;
logic reg5;

// 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; //  $(((((A5 * x + A4) * x + A3) * x + A2) * x + A1) * x + A0$ 
logic [WIDTHOUT-1:0] y_D;

```

```

// Registers to hold flipflop output values

logic [WIDTHIN-1:0] d0_out;

logic [WIDTHOUT-1:0] d1_out;

logic [WIDTHIN-1:0] d2_out;

logic [WIDTHOUT-1:0] d3_out;

logic [WIDTHIN-1:0] d4_out;

logic [WIDTHOUT-1:0] d5_out;

logic [WIDTHIN-1:0] d6_out;

logic [WIDTHOUT-1:0] d7_out;

logic [WIDTHIN-1:0] d8_out;


// compute y value

//D flipflop at initial stage

Dflipflop #(.width(WIDTHIN)) flip1 (.clk(clk), .rst(reset), .en(enable), .d(x), .q(d0_out));

mult16x16 Mult0 (.i_dataa(A5),      .i_datab(d0_out),      .o_res(m0_out));

addr32p16 Addr0 (.i_dataa(m0_out),  .i_datab(A4),      .o_res(a0_out)); // flipflop after each adder

Dflipflop #(.width(WIDTHOUT)) flip2 (.clk(clk), .rst(reset), .en(enable), .d(a0_out), .q(d1_out));

//flipflop before every x input to the consequent multipliers

Dflipflop #(.width(WIDTHIN)) flip3 (.clk(clk), .rst(reset), .en(enable), .d(d0_out), .q(d2_out));

mult32x16 Mult1 (.i_dataa(d1_out),  .i_datab(d2_out),      .o_res(m1_out));

addr32p16 Addr1 (.i_dataa(m1_out),  .i_datab(A3),      .o_res(a1_out)); // flipflop after each adder

Dflipflop #(.width(WIDTHOUT)) flip4 (.clk(clk), .rst(reset), .en(enable), .d(a1_out), .q(d3_out));

//flipflop before every x input to the consequent multipliers

Dflipflop #(.width(WIDTHIN)) flip5 (.clk(clk), .rst(reset), .en(enable), .d(d2_out), .q(d4_out));

mult32x16 Mult2 (.i_dataa(d3_out),  .i_datab(d4_out),      .o_res(m2_out));

addr32p16 Addr2 (.i_dataa(m2_out),  .i_datab(A2),      .o_res(a2_out)); // flipflop after each adder

Dflipflop #(.width(WIDTHOUT)) flip6 (.clk(clk), .rst(reset), .en(enable), .d(a2_out), .q(d5_out));

//flipflop before every x input to the consequent multipliers

Dflipflop #(.width(WIDTHIN)) flip7 (.clk(clk), .rst(reset), .en(enable), .d(d4_out), .q(d6_out));

```

```

mult32x16 Mult3 (.i_dataa(d5_out),    .i_datab(d6_out),    .o_res(m3_out));
addr32p16 Addr3 (.i_dataa(m3_out),    .i_datab(A1),    .o_res(a3_out)); // flipflop after each adder
Dflipflop #(.width(WIDTHOUT)) flip8 (.clk(clk), .rst(reset), .en(enable), .d(a3_out), .q(d7_out));
//flipflop before every x input to the consequent multipliers
Dflipflop #(.width(WIDTHIN)) flip9 (.clk(clk), .rst(reset), .en(enable), .d(d6_out), .q(d8_out));
mult32x16 Mult4 (.i_dataa(d7_out),    .i_datab(d8_out),    .o_res(m4_out));
addr32p16 Addr4 (.i_dataa(m4_out),    .i_datab(A0),    .o_res(a4_out));

//Final output
assign y_D = 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
        valid_Q1 <= 1'b0;
        reg1 <= 1'b0;
        reg2 <= 1'b0;
        reg3 <= 1'b0;
        reg4 <= 1'b0;
        reg5 <= 1'b0;
        valid_Q2 <= 1'b0;

        x <= 0;
        y_Q <= 0;
    end
end

```

```

    end else if (enable) begin
        // propagate the valid value
        valid_Q1 <= i_valid;
        reg1 <= valid_Q1;
        reg2 <= reg1;
        reg3 <= reg2;
        reg4 <= reg3;
        reg5 <= reg4;
        valid_Q2 <= reg5;

        // read in new x value
        x <= i_x;

        // output computed y value
        y_Q <= y_D;
    end
end

// assign outputs
assign o_y = y_Q;
// 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 when the receiver is ready again (i_ready is high)
assign o_valid = valid_Q2 & 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_dataaa * 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_dataaa,  
    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_dataaa + {5'b00000, i_datab, 11'b000000000000};
```

```
endmodule
```

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

```
//D-flipflop
```

```
module Dflipflop #(parameter width = 32)
```

```
(
```

```
    input [width-1:0] d,
```

```
    input clk,
```

```
    input rst,
```

```
    input en,
```

```
    output logic [width-1:0] q
```

```
);
```

```
always_ff @(posedge clk)
```

```
begin
```

```
    if (rst)
```

```
        q <= 0;
```

```
    else if (en)
```

```
        q <= d;
```

```
end
```

```
endmodule
```

```

/*Jashwant Raj G shared hardware circuit*/

module lab1 #

(
    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; // Register to hold input X
logic [WIDTHOUT-1:0] y_Q; // Register to hold output Y
logic valid_Q1; // Output of register x is valid
logic valid_Q2; // Output of register y is valid
logic output_give; //Used to determine if we need a new i_x

//Stages defined
logic [2:0] Stage = 3'b000;

// signal for enabling sequential circuit elements
logic enable;

//register to control all the input at each point
logic [2:0] control=3'b000;

// Signals for computing the y output
logic [WIDTHOUT-1:0] m0_out;
logic [WIDTHOUT-1:0] a0_out;
logic [WIDTHOUT-1:0] m1_out;
logic [WIDTHOUT-1:0] y_D;

//Registers to select what input goes in
logic [WIDTHOUT-1:0] m0_in;
logic [WIDTHOUT-1:0] m1_in;
logic [WIDTHOUT-1:0] a0_in;
logic [WIDTHOUT-1:0] a1_in;

//Registes to hold the values that go into m0,m1,a0,a1

```

```

logic [WIDTHOUT-1:0] m1_hold;
logic [WIDTHOUT-1:0] m2_hold;
logic [WIDTHOUT-1:0] a1_hold;
logic [WIDTHOUT-1:0] a2_hold;

// compute y value
//We decide which output we need based on the input size
mult32x16 Mult0 (.i_dataa(m0_in), .i_datab(m1_in), .o_res1(m0_out), .o_res2(m1_out));
addr32p16 Addr0 (.i_dataa(a0_in), .i_datab(a1_in), .o_res(a0_out));

// Combinational logic
always_comb begin
    // signal for enable
    enable = i_ready;
    m0_in = m1_hold;           //These are registers that take the values of hold registers
    m1_in = m2_hold;           //irrespective of clock
    a0_in = a1_hold;
    a1_in = a2_hold;
end

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        valid_Q1 <= 1'b0;
        valid_Q2 <= 1'b0;
        output_give <= 1'b0;
        x <= 0;
        y_Q <= 0;
        Stage = 3'b000;
    end
end

```

```

else if (enable) begin
    if (Stage == 3'b000) begin //Initial State
        output_give <= 1;          // This register provides information to o_ready
that we are ready to accept new i_x
        valid_Q2 <= 0;             // This register provides information to
o_valid that we do not have a output ready
        Stage <= 3'b011;          // Move to Decider state
    end

    else if (Stage == 3'b001) begin //First Multiplication state
        output_give <= 0;
        valid_Q2 <= 0;
        m2_hold <= x;              //first time we provide X along with A5
        m1_hold <= A5;
        Stage <= 3'b010;          //Move to Addition State
    end

    else if (Stage == 3'b010) begin //Addition State
        output_give <= 0;          //We use a control that decides
what input needs to go when
        valid_Q2 <= 0;            //similar to a MUX
        if(control == 3'b000) begin //Adder 1
            a1_hold <= m0_out;      //First mutiplier output and A4 value
            a2_hold <= A4;
            control = 3'b001;       //Move to Adder 2
            Stage = 3'b101;         //Move to next mutiplication
        end

        else if(control == 3'b001) begin//Adder 2
            a1_hold <= m1_out;      //Mutiplication with 2nd mutiplication
output and A3 value
            a2_hold <= A3;
            control <= 3'b010;     //Move to Adder 3
        end
    end
end

```

```

        Stage <= 3'b101;                //Move to next mutiplication
    end
    else if(control == 3'b010) begin//Adder 3
        a1_hold <= m1_out;              //Mutiplication with 2nd mutiplication
        output and A2 value

        a2_hold <= A2;
        control <= 3'b011;              //Move to Adder 4
        Stage <= 3'b101;                //Move to next mutiplication
    end
    else if(control == 3'b011) begin//Adder 4
        a1_hold <= m1_out;              //Mutiplication with 2nd mutiplication
        output and A1 value

        a2_hold <= A1;
        control <= 3'b100;              //Move to Final adder
        Stage <= 3'b101;                //Move to next mutiplication
    end
    else if(control == 3'b100) begin//Final Adder
        a1_hold <= m1_out;              //Mutiplication with 2nd mutiplication
        output and A0 value

        a2_hold <= A0;
        control <= 3'b000;              //Move to Adder 1
        Stage <= 3'b111;                //Move to Final State
    end
end

else if (Stage == 3'b101)begin //Consecutive Mutiplication state
    output_give <= 0;
    valid_Q2 <= 0;
    m2_hold <= x;
    m1_hold <= a0_out;                  //Based on the previous Adders output
    from Addition state

```

```

        Stage <= 3'b010;                                //we calculate input for next
Adder
    end
    else if (Stage == 3'b011)begin //Decider State to see if we get valid input or not based on
output_give register
        output_give <= 0;                                // the output_give provides info
to o_ready to send x
        valid_Q2 <= 0;
        if (i_valid) begin // Based on i_valid we decide if we compute output
            x <= i_x;                                    // Store X values for future use
            Stage <= 3'b001;    //Moves to first mutiplication state.
        end
        else begin
            Stage <= 3'b000; //Moves to initial state
        end
    end
    else if (Stage == 3'b111) begin // Final state where we store our final output
        y_Q <= a0_out;                                    //We used y_Q to store it and
later on push it to o_y
        output_give <= 1;                                // This register provides
information to o_ready that we are ready to accept new i_x
        valid_Q2 <= 1;                                    // This register provides
information to o_valid that we have a output ready
        Stage <= 3'b011;                                //Moves to Decider State
    end
end
end
// assign outputs
assign o_y = y_Q;
// ready for inputs as long as receiver is ready for outputs */
assign o_ready = output_give;

```



```

// 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 when the receiver is ready again (i_ready is high)
assign o_valid = valid_Q2;

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_res1,
    output [31:0] o_res2
);

    logic [47:0] 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_res1 = {3'b000, result[31:3]}; // For 16*16 bit multiplier

    // 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_res2 = result[45:14]; // For 32*16 bit multiplier

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'b000000000000};
```

```
endmodule
```

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