

1.Introduction

For this assignment, I have implemented different digital circuits that compute the exponential function e^x for 16-bit fixed-point input values. The exponential function is a simple but very common mathematical operation in many applications. And the results are tabulated correspondingly.

2.Results Table

	Baseline Circuit	Pipelined Circuit	Shared HW Circuit
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	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 1: Implementation results for the 3 different implementations of the studied circuit. The process of arriving to the above tabulated result will be shown later in the report.

2.1 Discussion Section

- (a) The most important error is due to the equation we used to fit the fact that we are limited by the amount of hardware resources, so it's impossible to extend the Taylor series infinitely, resulting in significant difference between computed value and true value. Another source of error is also mathematics, it's from the precision of the input resulted by the number of digits we have in our equations; we could use more bits to increase the computation accuracy.
- (b) Pipelined circuit achieved the highest throughput, followed by baselined circuit, then shared HW circuit. The nature of pipelined circuit structure reduced clock cycle computation time significantly, the overall design achieved much higher frequency compared to baselined circuit design.
- However, the shared HW circuit reuses its functional units due to having significantly less functional units compared to baselined circuit and pipelined circuit, resulting in more cycles

(extra time) taken for each input to finish computing. Thus making shared HW circuit the lowest throughput circuit.

- (c) Baselined circuit has the highest toggle rate of 12.819 million per second, followed by Shared HW circuit which has 12.010 millions per second, pipelined circuit has the least rate of 11.527 million per second.

Talking about power efficiency, all the circuit calculation are finished under 42 MHz which is the only available option; baselined circuit is the best at throughput/Watt and followed by the pipelined circuit and shared HW circuit finished at last because the advantage it had on operating frequency was negated due to the 42MHz environment. As result, larger numbers of registers in pipelined circuit have made its dynamic power higher than baselined circuit; as of shared HW circuit, since it has much less functional units, the dynamic power consumption is significantly less.

3. Pipelined circuit analysis

3.1 Block diagram (RTL viewer)

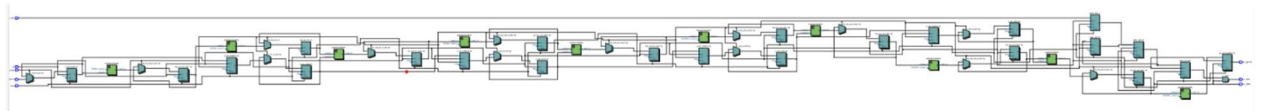


Figure 1: Pipelined circuit block diagram

Pipelined circuits achieve maximum operating frequency by reducing the amount of computation done for each clock cycle.

3.2 Readable simulation waveform and testbench output

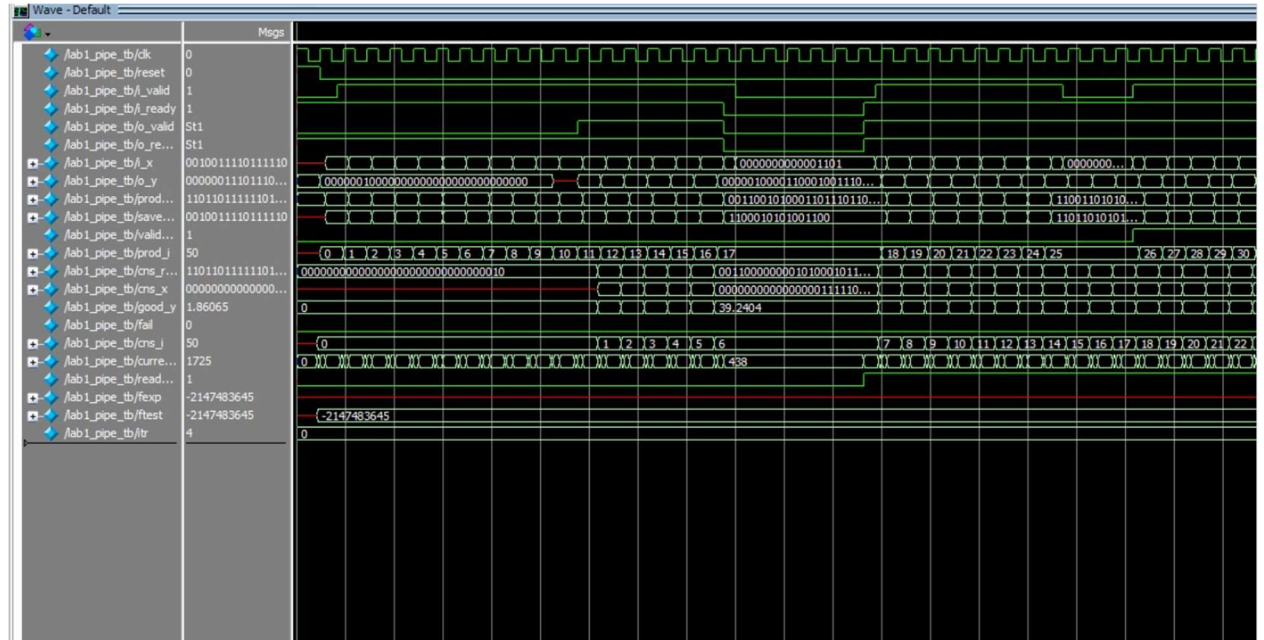


Figure 2: Pipelined circuit simulation waveform

Figure 3: Pipelined circuit testbench output

4.1 Block Diagram(RTL viewer)



The shared HW circuit has limited number of multiplier and adder blocks, thus a finite state machine is necessary to control the datapath between input `i_x` and output `o_y`. Construction of a controllable circuit is necessary. The corresponding control signals are named `m_mult` and `a_mult`, along with enable signals `s_x` and `s_y` to prevent i/o registers from getting invalid values.

4.2 Readable simulation waveform and testbench output

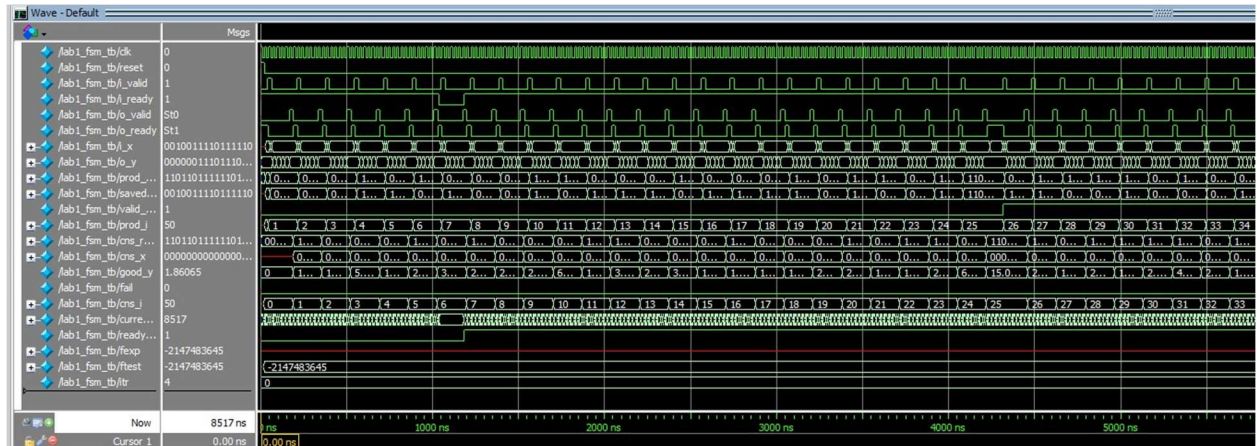


Figure 5: Shared Hardware circuit waveform


```
# at 525ns SUCCESS X: 1.741333 Expected Y: 5.653997 Got Y: 5.652892 Error: 0.001105 < 0.045000
# at 693ns SUCCESS X: 0.175354 Expected Y: 1.191668 Got Y: 1.191668 Error: 0.000000 < 0.045000
# at 861ns SUCCESS X: 3.548218 Expected Y: 29.578277 Got Y: 29.552900 Error: 0.025378 < 0.045000
# at 1029ns SUCCESS X: 3.890381 Expected Y: 39.240435 Got Y: 39.201718 Error: 0.038717 < 0.045000
# at 1197ns SUCCESS X: 0.740234 Expected Y: 2.096173 Got Y: 2.096136 Error: 0.000038 < 0.045000
# at 1365ns SUCCESS X: 3.255310 Expected Y: 23.027957 Got Y: 23.010813 Error: 0.017144 < 0.045000
# at 1533ns SUCCESS X: 1.052307 Expected Y: 2.862044 Got Y: 2.861901 Error: 0.000143 < 0.045000
# at 1701ns SUCCESS X: 1.834961 Expected Y: 6.193981 Got Y: 6.192602 Error: 0.001379 < 0.045000
# at 1869ns SUCCESS X: 2.957642 Expected Y: 17.717524 Got Y: 17.706403 Error: 0.011121 < 0.045000
# at 2037ns SUCCESS X: 1.379883 Expected Y: 3.962578 Got Y: 3.962157 Error: 0.000422 < 0.045000
# at 2205ns SUCCESS X: 3.171143 Expected Y: 21.399463 Got Y: 21.384237 Error: 0.015226 < 0.045000
# at 2373ns SUCCESS X: 3.684875 Expected Y: 33.155227 Got Y: 33.125056 Error: 0.030170 < 0.045000
# at 2541ns SUCCESS X: 2.703796 Expected Y: 14.084164 Got Y: 14.076718 Error: 0.007446 < 0.045000
# at 2709ns SUCCESS X: 0.554443 Expected Y: 1.740929 Got Y: 1.740915 Error: 0.000013 < 0.045000
# at 2877ns SUCCESS X: 2.880005 Expected Y: 16.525907 Got Y: 16.516037 Error: 0.009870 < 0.045000
# at 3045ns SUCCESS X: 3.082764 Expected Y: 19.800061 Got Y: 19.786659 Error: 0.013402 < 0.045000
# at 3213ns SUCCESS X: 3.428040 Expected Y: 26.715896 Got Y: 26.694210 Error: 0.021686 < 0.045000
# at 3381ns SUCCESS X: 3.298218 Expected Y: 23.899509 Got Y: 23.881316 Error: 0.018193 < 0.045000
# at 3549ns SUCCESS X: 0.639221 Expected Y: 1.894901 Got Y: 1.894879 Error: 0.000022 < 0.045000
# at 3717ns SUCCESS X: 2.401550 Expected Y: 10.645304 Got Y: 10.640895 Error: 0.004409 < 0.045000
# at 3885ns SUCCESS X: 0.701355 Expected Y: 2.016301 Got Y: 2.016270 Error: 0.000031 < 0.045000
# at 4053ns SUCCESS X: 1.910767 Expected Y: 6.666646 Got Y: 6.665008 Error: 0.001638 < 0.045000
# at 4221ns SUCCESS X: 2.775940 Expected Y: 15.041535 Got Y: 15.033163 Error: 0.008372 < 0.045000
# at 4461ns SUCCESS X: 3.417664 Expected Y: 26.480547 Got Y: 26.459158 Error: 0.021389 < 0.045000
# at 4629ns SUCCESS X: 2.634399 Expected Y: 13.215598 Got Y: 13.208965 Error: 0.006633 < 0.045000
# at 4797ns SUCCESS X: 3.368408 Expected Y: 25.387968 Got Y: 25.367948 Error: 0.020021 < 0.045000
# at 4965ns SUCCESS X: 0.598083 Expected Y: 1.818561 Got Y: 1.818544 Error: 0.000017 < 0.045000
# at 5133ns SUCCESS X: 1.034790 Expected Y: 2.812525 Got Y: 2.812391 Error: 0.000134 < 0.045000
# at 5301ns SUCCESS X: 3.951538 Expected Y: 41.228160 Got Y: 41.186558 Error: 0.041602 < 0.045000
# at 5469ns SUCCESS X: 0.823242 Expected Y: 2.277386 Got Y: 2.277330 Error: 0.000056 < 0.045000
# at 5637ns SUCCESS X: 0.559326 Expected Y: 1.749448 Got Y: 1.749434 Error: 0.000014 < 0.045000
# at 5805ns SUCCESS X: 0.140198 Expected Y: 1.150501 Got Y: 1.150501 Error: 0.000000 < 0.045000
# at 5973ns SUCCESS X: 3.374756 Expected Y: 25.526511 Got Y: 25.506319 Error: 0.020193 < 0.045000
# at 6141ns SUCCESS X: 3.054749 Expected Y: 19.315723 Got Y: 19.302862 Error: 0.012861 < 0.045000
# at 6309ns SUCCESS X: 0.445984 Expected Y: 1.562015 Got Y: 1.562009 Error: 0.000006 < 0.045000
# at 6477ns SUCCESS X: 3.698364 Expected Y: 33.527879 Got Y: 33.497199 Error: 0.030680 < 0.045000
# at 6645ns SUCCESS X: 2.359924 Expected Y: 10.237271 Got Y: 10.233188 Error: 0.004083 < 0.045000
# at 6813ns SUCCESS X: 1.651001 Expected Y: 5.175761 Got Y: 5.174877 Error: 0.000884 < 0.045000
# at 6981ns SUCCESS X: 1.010254 Expected Y: 2.744582 Got Y: 2.744460 Error: 0.000122 < 0.045000
# at 7149ns SUCCESS X: 1.281128 Expected Y: 3.593229 Got Y: 3.592917 Error: 0.000312 < 0.045000
# at 7317ns SUCCESS X: 2.267822 Expected Y: 9.385154 Got Y: 9.381723 Error: 0.003431 < 0.045000
# at 7485ns SUCCESS X: 0.239319 Expected Y: 1.270383 Got Y: 1.270383 Error: 0.000001 < 0.045000
# at 7653ns SUCCESS X: 1.531799 Expected Y: 4.603725 Got Y: 4.603078 Error: 0.000647 < 0.045000
# at 7821ns SUCCESS X: 3.903503 Expected Y: 39.659827 Got Y: 39.620505 Error: 0.039322 < 0.045000
# at 7989ns SUCCESS X: 3.112427 Expected Y: 20.324626 Got Y: 20.310633 Error: 0.013993 < 0.045000
# at 8157ns SUCCESS X: 0.248901 Expected Y: 1.282615 Got Y: 1.282614 Error: 0.000001 < 0.045000
# at 8325ns SUCCESS X: 3.393311 Expected Y: 25.935299 Got Y: 25.914596 Error: 0.020703 < 0.045000
# at 8493ns SUCCESS X: 0.620972 Expected Y: 1.860649 Got Y: 1.860629 Error: 0.000020 < 0.045000
# ALL TESTS PASSED
# Break in Module lab1_fsm_tb at C:/Users/ZaKaye/Desktop/Pixiv/MEng Study 2021 Fall/FPGA/lab1/lab1_fsm/lab1_fsm_tb.v line 258
VSIM 2>
```

Figure 6: Shared Hardware circuit testbench output

5. Result table tabulation

The data I tabulated in the table are from simulation of the lab for which I will show below.

Resource for one circuit:

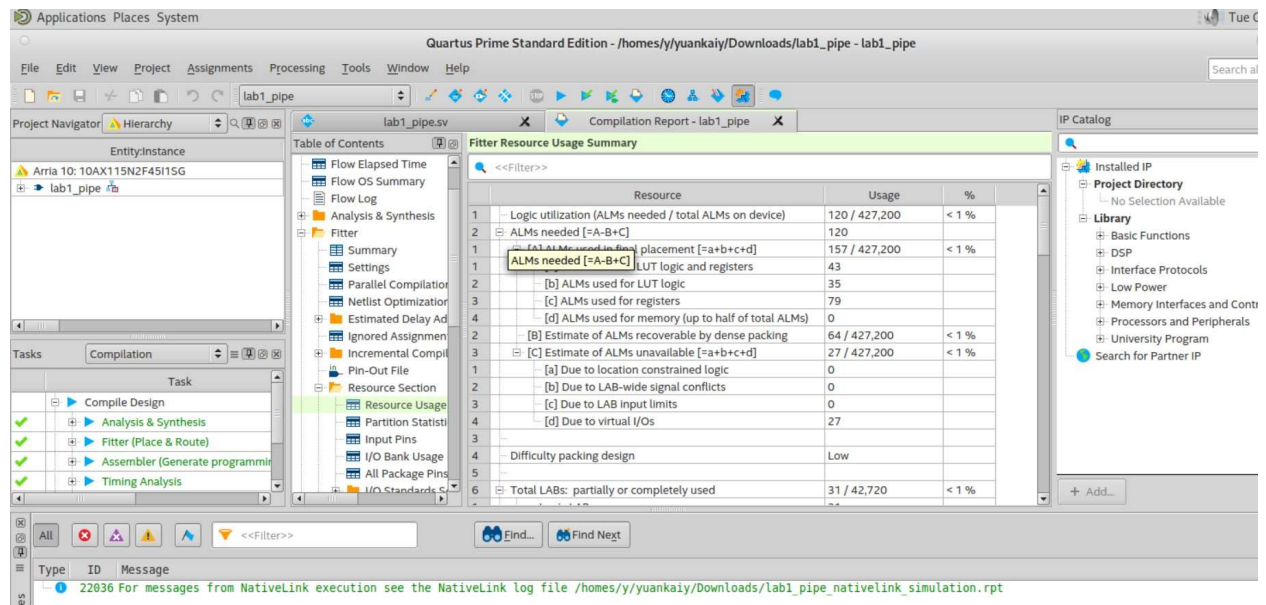


Figure 7: Resource for pipelined circuit

Operating Frequency:

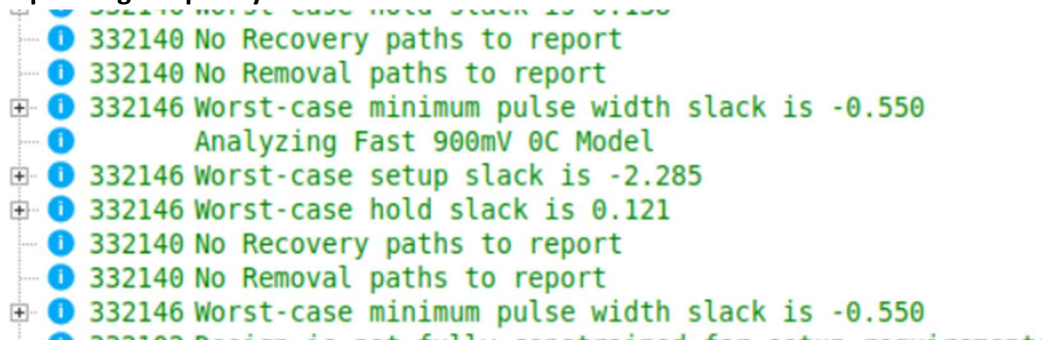


Figure 8: SharedHW circuit compilation report

Critical Path:

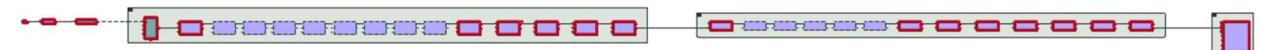


Figure 9: SharedHW circuit critical path

Cycle for valid unit:

Each input only takes 1 cycle to process, SharedHW circuit has 7 input.

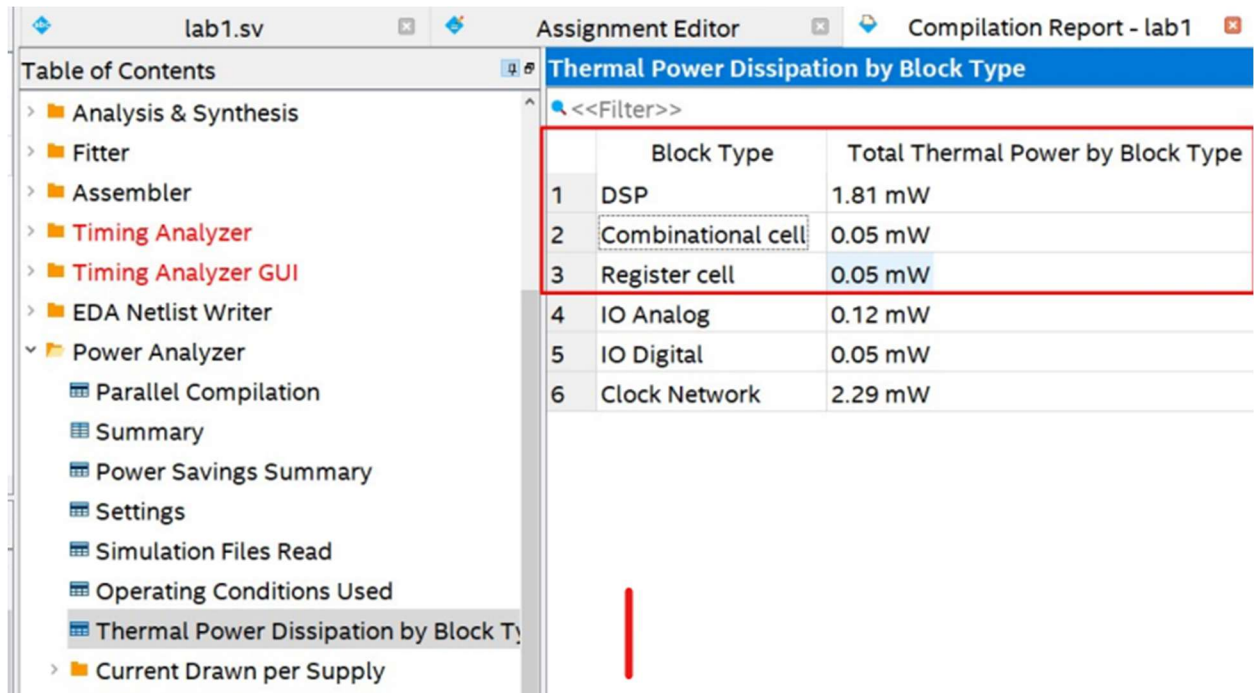
Max. # of copies/device:

Referring to line 12 Total DSP Blocks of Fitter Resource of Usage Summary, $1518/9=168$

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

To get the value of the Max. throughput, simply take operating frequency and multiply by Max. # of copies/device.

Dynamic power of one circuit at 42 MH:



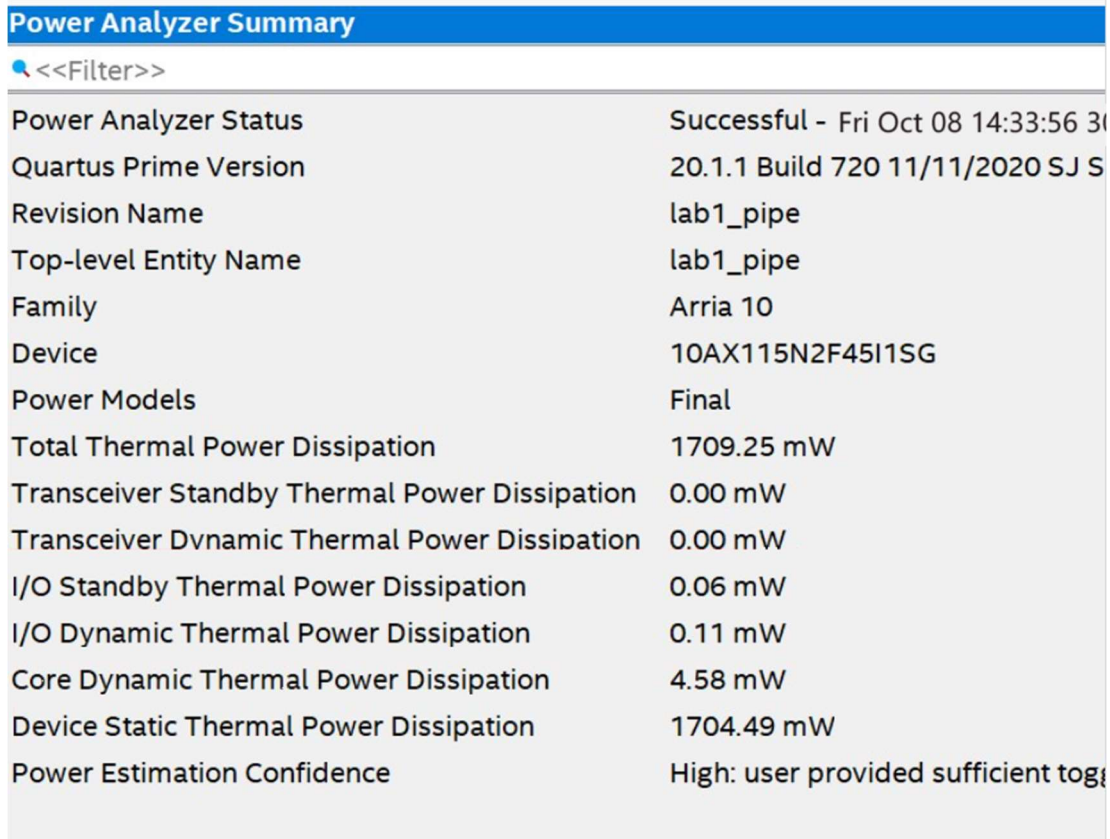
The screenshot shows the Quartus Prime Assignment Editor with the 'lab1.sv' project open. The 'Table of Contents' on the left lists various analysis tools, with 'Thermal Power Dissipation by Block Type' selected. The main window displays a table titled 'Thermal Power Dissipation by Block Type' with a search filter '<<Filter>>'. The table lists six block types and their corresponding total thermal power dissipation.

	Block Type	Total Thermal Power by Block Type
1	DSP	1.81 mW
2	Combinational cell	0.05 mW
3	Register cell	0.05 mW
4	IO Analog	0.12 mW
5	IO Digital	0.05 mW
6	Clock Network	2.29 mW

Figure 10: Thermal Power Dissipation for baselined circuit

$1.81\text{mW} + 0.05\text{mW} + 0.05\text{mW} = 1.86\text{mW}$

Max. throughput/Watt for a full device:



The screenshot shows the 'Power Analyzer Summary' window. It provides a detailed overview of the power analysis results, including the status, version, revision name, top-level entity name, family, device, power models, and various power dissipation metrics.

Power Analyzer Summary	
<<Filter>>	
Power Analyzer Status	Successful - Fri Oct 08 14:33:56 30
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ S
Revision Name	lab1_pipe
Top-level Entity Name	lab1_pipe
Family	Arria 10
Device	10AX115N2F45I1SG
Power Models	Final
Total Thermal Power Dissipation	1709.25 mW
Transceiver Standby Thermal Power Dissipation	0.00 mW
Transceiver Dvnamic 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	4.58 mW
Device Static Thermal Power Dissipation	1704.49 mW
Power Estimation Confidence	High: user provided sufficient tog

Figure 11: pipelined circuit power consumption summary

First, we need to compute the total approximate power consumption.

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

Throughput/W att @ 42 MHz = $7.056 \times 10^9 / 2025.85 \times 10^{-3} \approx 3.454 \times 10^9$ computations/W

Above are the brief explanations of how I came to the data shown in the table at the beginning of this report.

Appendix

Pipelined circuit code:

```
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] In_x_m0, Hold_x_m0; //set input x register as input to m0, propagation delay
logic [WIDTHIN-1:0] In_x_m1, Hold_x_m1, In_x_m2, Hold_x_m2, In_x_m3, Hold_x_m3, In_x_m4;
//pipeline x register as input to m1,m2,m3,m4 correspondingly

// Pipeline registers for output values, for example m0_out_a0_in is the register output after m0 as input
into at

logic [WIDTHOUT-1:0] m0_out_a0_in, a0_out_m1_in_pipe,

        m1_out_a1_in, a1_out_m2_in,

        m2_out_a2_in, a2_out_m3_in,

        m3_out_a3_in, a3_out_m4_in,

        m4_out_a4_in, a4_out_final;

logic V_In_x;                // Output of register x is valid

logic V_m0, V_a0;            // Output of m0 and a0 is valid

logic V_m1, V_a1;            // Output of m1 and a1 is valid

logic V_m2, V_a2;            // Output of m2 and a2 is valid

logic V_m3, V_a3;            // Output of m3 and a3 is valid

logic V_m4, V_a4_Final;      // Output of m4 and final result a4 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(ln_x_m0),      .o_res(m0_out));
addr32p16 Addr0 (.i_dataa(m0_out_a0_in),      .i_datab(A4),      .o_res(a0_out));

mult32x16 Mult1 (.i_dataa(a0_out_m1_in),      .i_datab(ln_x_m1),      .o_res(m1_out));
addr32p16 Addr1 (.i_dataa(m1_out_a1_in),      .i_datab(A3),      .o_res(a1_out));

mult32x16 Mult2 (.i_dataa(a1_out_m2_in),      .i_datab(ln_x_m2),      .o_res(m2_out));
addr32p16 Addr2 (.i_dataa(m2_out_a2_in),      .i_datab(A2),      .o_res(a2_out));

mult32x16 Mult3 (.i_dataa(a2_out_m3_in),      .i_datab(ln_x_m3),      .o_res(m3_out));
addr32p16 Addr3 (.i_dataa(m3_out_a3_in),      .i_datab(A1),      .o_res(a3_out));

mult32x16 Mult4 (.i_dataa(a3_out_m4_in),      .i_datab(ln_x_m4),      .o_res(m4_out));
addr32p16 Addr4 (.i_dataa(m4_out_a4_in),      .i_datab(A0),      .o_res(a4_out));

// Combinational logic
```

Kaiyi Yuan
1003084716
always_comb begin

Assignment1

ECE1756
OCT 10 2023

```
    // signal for enable  
    enable = i_ready;  
end
```

```
// Infer the registers
```

```
always_ff @(posedge clk or posedge reset) begin
```

```
    if (reset) begin
```

```
        ln_x_m0 <= 0;
```

```
        Hold_x_m0 <= 0;
```

```
        ln_x_m1 <= 0;
```

```
        Hold_x_m1 <= 0;
```

```
        ln_x_m2 <= 0;
```

```
        Hold_x_m2 <= 0;
```

```
        ln_x_m3 <= 0;
```

```
        Hold_x_m3 <= 0;
```

```
        ln_x_m4 <= 0;
```

```
        V_ln_x <= 1'b0;
```

```
        V_m0 <= 1'b0;
```

```
        V_a0 <= 1'b0;
```

```
        V_m1 <= 1'b0;
```

```
        V_a1 <= 1'b0;
```

```
        V_m2 <= 1'b0;
```

```
        V_a2 <= 1'b0;
```

```
        V_m3 <= 1'b0;
```

```
        V_a3 <= 1'b0;
```

```
        V_m4 <= 1'b0;
```

```
        V_a4_Final <= 1'b0;
```

Assignment1

```
m0_out_a0_in <= 1'b0;
a0_out_m1_in <= 1'b0;
m1_out_a1_in <= 1'b0;
a1_out_m2_in <= 1'b0;
m2_out_a2_in <= 1'b0;
a2_out_m3_in <= 1'b0;
m3_out_a3_in <= 1'b0;
a3_out_m4_in <= 1'b0;
m4_out_a4_in <= 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
    ln_x_m0 <= i_x;
    Hold_x_m0 <= ln_x_m0;
    ln_x_m1 <= Hold_x_m0;
    Hold_x_m1 <= ln_x_m1;
    ln_x_m2 <= Hold_x_m1;
    Hold_x_m2 <= ln_x_m2;
    ln_x_m3 <= Hold_x_m2;
    Hold_x_m3 <= ln_x_m3;
    ln_x_m4 <= Hold_x_m3;

    // propagate the valid value
    V_ln_x <= i_valid;
    V_m0 <= V_ln_x;
    V_a0 <= V_m0;
```


Assignment1

```
V_m1 <= V_a0;  
V_a1 <= V_m1;  
V_m2 <= V_a1;  
V_a2 <= V_m2;  
V_m3 <= V_a2;  
V_a3 <= V_m3;  
V_m4 <= V_a3;  
V_a4_Final <= V_m4;
```

```
// update outs  
m0_out_a0_in <= m0_out;  
a0_out_m1_in <= a0_out;  
m1_out_a1_in <= m1_out;  
a1_out_m2_in <= a1_out;  
m2_out_a2_in <= m2_out;  
a2_out_m3_in <= a2_out;  
m3_out_a3_in <= m3_out;  
a3_out_m4_in <= a3_out;  
m4_out_a4_in <= 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 = V_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_dataaa + {5'b00000, i_datab, 11'b000000000000};
```

```
endmodule
```

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

Shared HW circuit code:

```
module lab1_shared #  
(  
    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  
);
```


Assignment1

```
// control signals
logic s_x;//enable signals
logic s_y;

logic m_mult;//for multiplier
logic [2:0] a_mult;//for adder

// Register declarations
logic [WIDTHIN-1:0] x_register;
logic [WIDTHOUT-1:0] y_register;

// Wires
logic [WIDTHOUT-1:0] inputm;
logic [WIDTHOUT-1:0] outputm;
logic [WIDTHIN-1:0] inputa;
logic [WIDTHOUT-1:0] outputa;

// one set of adder and multiplexer
mult32x16 Mult1 (.i_dataa(inputm),      .i_datab(x_register),
                .o_res(outputm));
addr32p16 Addr1 (.i_dataa(outputm),      .i_datab(inputa),      .o_res(outputa));

// input multiplexers to multiplier and adder
mux2x1 mux2x1_m_mult(.sel(m_mult), .a({5'b0, A5, 11'b0}), .b(y_register), .out(inputm));
mux8x1
mux8x1_a_mult(.sel(a_mult), .a(A4), .b(A3), .c(A2), .d(A1), .e(A0), .f(16'b0), .g(16'b0), .h(16'b
0), .out(inputa));

// Registers
always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        x_register <= 0;
        y_register <= 0;
    end else if (s_x) begin
        x_register <= i_x;
    end else if (s_y) begin
        y_register <= outputa;
    end
end
end
```

```
    assign o_y = y_register;

endmodule

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

module shared(
    input clk,
    input reset,
    input i_valid,
    input i_ready,
    output logic o_valid,
    output logic o_ready,
    output logic s_x,
    output logic s_y,
    output logic m_mult,          //      multiplexer signal to control multiplier
    output logic [2:0] a_mult // multiplexer signal to control adder
);

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

logic input_valid;
logic enable;

// update status register
always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        current <= input_x;
    end else begin
        current <= next; // chekc enable i/o in comb
    end
end

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

// next state computation
always_comb begin
```

Assignment1

```
case(current)
    input_x: if(input_valid) begin    // wait for input
        next = compute_0;
    end else begin
        next = input_x;
    end
    compute_0: next = compute_1; // cascading computation of x state and y state
regardless
    compute_1: next = compute_2; // whether it's ready to output or not
    compute_2: next = compute_3;
    compute_3: next = compute_4;
    compute_4: next = output_y;
    output_y: if(enable) begin                // wait for i_ready before
continue
        next = input_x;
    end else begin
        next = output_y;
    end
endcase
end

// shared control signals
always_comb begin
    o_valid = 1'b0;
    o_ready = 1'b0;
    s_x = 1'b0;           // x_register control
    s_y = 1'b0;           // y_register control
    m_mult = 1'b0;        // Controls multiplier mux
    a_mult = 3'b000;      // Controls adder mux
    case(current)
        input_x: begin
            o_valid = 1'b0; // repeat for output not ready
            o_ready = 1'b1; // new input ready
            s_x = 1'b1;     // x into register
        end
        compute_0: begin
            m_mult = 1'b0;   // select a5 for multiplier
            a_mult = 3'b000; // Select a4 for adder
            s_y = 1'b1;     // Store result into register
        end
        compute_1: begin
            m_mult = 1'b1;   // Select previous value for multiplier
            a_mult = 3'b001; // Selecting a3 for adder
            s_y = 1'b1;     // Store result into register
        end
    endcase
end
```

Assignment1

```
end
compute_2: begin
    m_mult = 1'b1;          // Select previous value for multiplier
    a_mult = 3'b010;        // Selecting a2 for adder
    s_y = 1'b1;             // Store result into register
end
compute_3: begin
    m_mult = 1'b1;          // Select previous value for multiplier
    a_mult = 3'b011;        // Selecting a1 for adder
    s_y = 1'b1;             // Store result into register
end
compute_4: begin
    m_mult = 1'b1;          // Select previous value for multiplier
    a_mult = 3'b100;        // Selecting a0 for adder
    s_y = 1'b1;             // Store the final result into register
end
output_y: begin
    o_valid = 1'b1; // Output ready
    o_ready = 1'b0;
end
endcase
end

endmodule
```

```
// Multiplier 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

    assign o_res = result[45:14];

endmodule
```


Kaiyi Yuan
1003084716

Assignment1

ECE1756
OCT 10 2023

```

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

// 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
);

assign o_res = i_dataa + {5'b00000, i_datab, 11'b000000000000};

endmodule

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