UNIVERSITY OF NEVADA LAS VEGAS. DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING LABORATORIES.

Class:	CPE300L	Semester:	Fall 2021

Points		Document author:	Jerrod Batu
		Author's email:	batuj1@unlv.nevada.edu
		Document topic:	Postlab 8

Instructor's comments:

1. Introduction / Theory of Operation

In this lab, I will continue my learning about the general datapath's implementation for a given algorithm. I will also learn more about timing simulations by setting up timing constraints and timing information as well as modeling memories.

a. In regards to time quest analysis **slack** is the range of where the timing requirement is met or not. When the timing requirement is met, this is called positive slack; hence, the negative slack is when the timing requirement is not met. Realistic slack values are met when all of the clocks in the design are constrained with real values.
b. It is important to consider **timing analysis** in a digital circuit design because incorrect format and outputs may result in different timing cycles. In addition, the timing analysis allows the user to easily make a circuit design more feasible and efficient on how fast the outputs can be processed through the inputs. Timing analysis, specifically, will determine the amount of time it takes between the inputs and outputs to be initialized. By having the timing requirement not met and negative slack, the circuit will not be maximized and efficient as there may be a slight delay between outputs after the clock is on the positive edge.

2. Prelab

https://docs.google.com/document/d/1PKcaACI4mB_VmDI2OBT4IPvLjs1yjqKz/edit?usp=sharing&ouid=102808507017671072128&rtpof=true&sd=true

This is the link to my prelab8.

3. Results of Experiments

EXPERIMENT 1

```
`timescale 1ns / 100ps

module GCD (startGDP, rstGDP, clkGDP, n1GDP, n2GDP, complete, GCDoutDP);
	input startGDP, clkGDP, rstGDP;
	input [7:0] n1GDP, n2GDP;
	output complete;
	output [7:0] GCDoutDP;

	wire WE, RAE, RBE, OE, IE, neq, gt;
	wire [1:0] WA, RAA, RBA, SH;
	wire [2:0] ALU;

	CU control (~startGDP, clkGDP, ~rstGDP, neq, gt, IE, WE, WA, RAE, RAA, RBE,
RBA, ALU, SH, OE);
	DP datapath (n1GDP, n2GDP, clkGDP, IE, WE, WA, RAE, RAA, RBE, RBA, ALU,
SH, OE, neq, gt, GCDoutDP);

	assign complete = OE;
endmodule

module CU (start, clk, rst, neq, gt, IE, WE, WA, RAE, RAA, RBE, RBA, ALU, SH, OE);
	input start, clk, rst;
	output IE, WE, RAE, RBE, OE;
	output [1:0] WA, RAA, RBA, SH;
	output [2:0] ALU;

	input wire neq, gt;
	reg [2:0] state;
	reg [2:0] nextstate;

	parameter S0 = 3'b000;
	parameter S1 = 3'b001;
	parameter S2 = 3'b010;
	parameter S3 = 3'b011;
	parameter S4 = 3'b100;
	parameter S5 = 3'b101;
	parameter S6 = 3'b110;

	initial
		state = S0;
```

```
// State register
always @ (posedge clk)
begin
        state <= nextstate;
end

// Next State Logic
always @ (*)
        case (state)
S0: if (start)                          // output a = 0
                        nextstate = S1;
                else
                        nextstate = S0;
        S1: nextstate = S2;                     // input a
S2: nextstate = S3;                     // input b
S3: if (neq && gt)                      // a > b
                        nextstate = S4;
        else if (neq && !gt)    // b < a
                        nextstate = S5;
                else
                        nextstate = S6;         // neither
        S4: nextstate = S3;
        S5: nextstate = S3;
        S6: if (rst)
                        nextstate = S0;
                else
                        nextstate = S6;
        default: nextstate  = S0;
        endcase

// Output Logic
assign IE = (state == S1) || (state == S2);

assign WE = (state == S1) || (state == S2) ||  (state == S4) || (state == S5);
assign WA[1] = (state == S2) || (state == S5);
assign WA[0] = (state == S1) || (state == S4);
assign RAE = (state == S3) || (state == S4) || (state == S5) || (state == S6);
assign RAA[1] = state == S5;
assign RAA[0] = (state == S3) || (state == S4) || (state == S6);

assign RBE = (state == S3) || (state == S4) || (state == S5);
assign RBA[1] = (state == S3) || (state == S4);
assign RBA[0] = state == S5;
```

```
        assign ALU[2] = (state == S3) || (state == S4) || (state == S5);
        assign ALU[1] = 0;
        assign ALU[0] = (state == S3) || (state == S4) || (state == S5);

        assign SH[1] = 0;
        assign SH[0] = 0;
        assign OE = state == S6;
endmodule

//Data Path
module DP (n1In, n2In, clk, IE, WE, WA, RAE, RAA, RBE, RBA, ALU, SH, OE, neq, gt,
out);
        input clk, IE, WE, RAE, RBE, OE;
        input [1:0] WA, RAA, RBA, SH;
        input [2:0] ALU;
        input [7:0] n1In, n2In;
        output neq, gt;
        output wire [7:0] out;

        reg [7:0] rfIn1, rfIn2;
        wire [7:0] rfA, rfB, aluOut, shOut, n1, n2;

        initial begin
                rfIn1 = 0;
        rfIn2 = 0;
   end

        always @ (*)
    begin
                rfIn1 = n1;
      rfIn2 = n2;
    end

        gdpMux1 mux1 (shOut, n1In, IE, n1);
        gdpMux2 mux2 (shOut, n2In, IE, n2);
        gdpRF RF (clk, WE, RAE, RBE, RAA, RBA, WA, rfIn1, rfIn2, rfA, rfB);
        gdpALU theALU (rfA, rfB, ALU, aluOut);
        gdpShift SHIFT (aluOut, SH, shOut);
        gdpBuffer buff (shOut, OE, out);

        assign neq = (|aluOut [7:0]);
        assign gt = (~aluOut[7]) && (|aluOut[6:0]);
endmodule
```

```
// 2-to-1 mux
module gdpMux1 (a, b, sel, out);
	input [7:0] a, b;
	input sel;
	output reg [7:0] out;

	always @(*) begin
		if(sel == 0)
			out = a;
	   else
			out = b;
	end
endmodule

// 2-to-1 mux
module gdpMux2 (a, b, sel, out);
	input [7:0] a, b;
	input sel;
	output reg [7:0] out;

	always @(*) begin
      if(sel == 0)
			out = a;
		else
			out = b;
	end
endmodule

// register file
module gdpRF(clk, WE, RAE, RBE, RAA, RBA, WA, inD1, inD2, ReadA, ReadB);
	input clk, WE, RAE, RBE;
	input [1:0] RAA, RBA, WA;
   input [7:0] inD1, inD2;
	output [7:0] ReadA, ReadB;
   reg [7:0] regF1 [0:3];
   reg [7:0] regF2 [0:3];

	// Write when WE asserted
	always @(posedge clk)
     begin
		if (WE == 1)
			begin
				regF1[WA] <= inD1;
```

```
                                    regF2[WA] <= inD2;
                            end
    end

        //reading to Port A and B, combinational
  assign ReadA = (RAE)? regF1 [RAA]:0;
  assign ReadB = (RBE)? regF2 [RBA]:0;
endmodule

// arithmetic logic unit
module gdpALU (a,b,sel, out);
   input [7:0] a,b;
   input [2:0] sel;
   output reg [7:0] out;

   always @ (*)
   begin
      case (sel)
         3'b000: out = a;
         3'b001: out = a & b;
         3'b010: out = a | b;
         3'b011: out = !a;
         3'b100: out = a + b;
         3'b101: out = a - b;
         3'b110: out = a + 1;
         3'b111: out = a - 1;
      endcase
   end
endmodule

// Shifter
module gdpShift (a,sh,out);
   input [7:0] a;
   input [1:0] sh;
   output reg [7:0] out;

   always @ (*)
   begin
      case(sh)
         3'b00: out = a;
         3'b01: out = a << 1;
         3'b10: out = a >> 1;
         3'b11: out= { a[6],a[5],a[4],a[3],a[2],a[1],a[0], a[7] } ;
      endcase
```

```
   end
endmodule

// Buffer
module gdpBuffer (a, buff, out);
	input [7:0] a;
	input buff;
	output reg [7:0] out;

	always @(*)
		if(buff == 1)
			out = a;
		else
			out = 8'bzzzz_zzzz;
endmodule
```

This is the general datapath for the GCD algorithm.

```
`timescale 1ns / 100ps

module GDPTB;
	reg startTB, rstTB, clkTB;
	reg [7:0] n1TB, n2TB;
	wire displayGCD;
	wire [7:0] GCDTest;

	// Test Vectors for desired output
	reg [7:0] inTest1 [1:5];
	reg [7:0] inTest2 [1:5];
	reg [7:0] outExpect [1:5];

	integer i;

	initial
		begin
			// initialize inputs to test and their expected outputs
	inTest1[1]   = 8'b00000000;
	inTest2[1]   = 8'b00000000;
		outExpect[1] = 8'b00000000;

	inTest1[2]   = 8'b00010110;
	inTest2[2]   = 8'b00010110;
```

```
        outExpect[2] = 8'b00010110;

        inTest1[3]    = 8'b00000010;
        inTest2[3]    = 8'b00000100;
        outExpect[3] = 8'b00000010;

        inTest1[4]    = 8'b10101010;
        inTest2[4]    = 8'b01010101;
        outExpect[4] = 8'b01010101;

        inTest1[5]    = 8'b11111111;
        inTest2[5]    = 8'b00001010;
        outExpect[5] = 8'b00000101;

                        startTB = 1;
                        rstTB = 1;
                        clkTB = 0;

                        // generate clock
                        forever
                                #2 clkTB = ~clkTB;
                end

        always
                begin
                        for(i = 1; i <= 5; i = i + 1) // test for the 5 inputs
                                begin
                                        n1TB <= inTest1[i];
                n2TB <= inTest2[i];
                                        startTB = 0;
                                        #40 rstTB = 1;

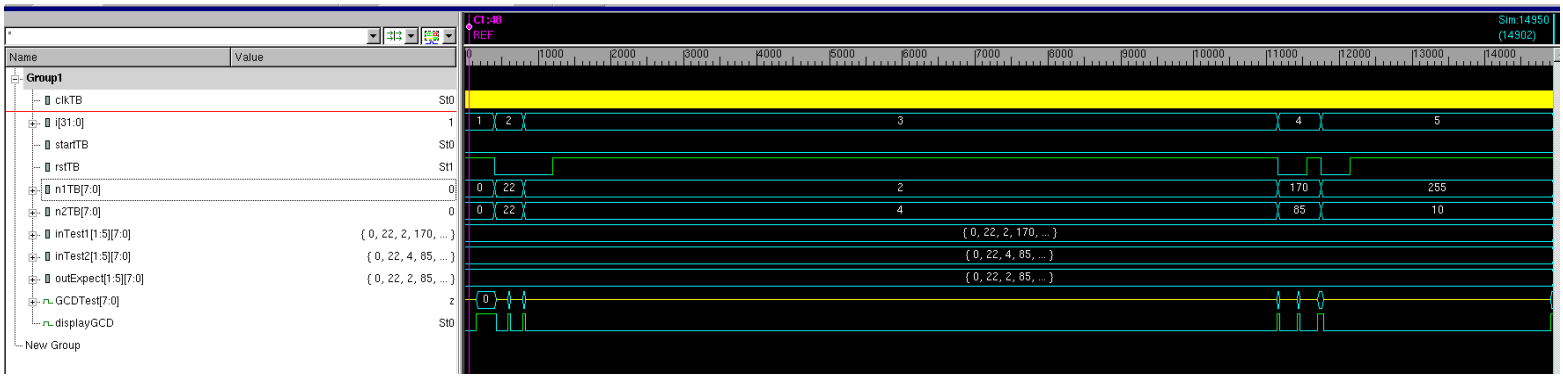
                                        // Waiting for code to finish then displayResults when
done
                                        while (displayGCD != 1)
                                                #5 begin end

            $write ("InputA: %b  InputB: %b  Expected: n =%d  Calculated: n =%d: ",
                        n1TB, n2TB, outExpect[i], GCDTest);
                                        if (GCDTest == outExpect[i])
                                                $display("Correct!");
                                        else
                                                $display("Incorrect!");
                                        rstTB = 0;
```

```
                                    startTB = 1;
                                end
                        $stop;
                end

        //Instantiate GDP
  GCD mainGDP (startTB, rstTB, clkTB, n1TB, n2TB, displayGCD, GCDTest);
endmodule
```

This is the testbench for the general datapath for the GCD algorithm.



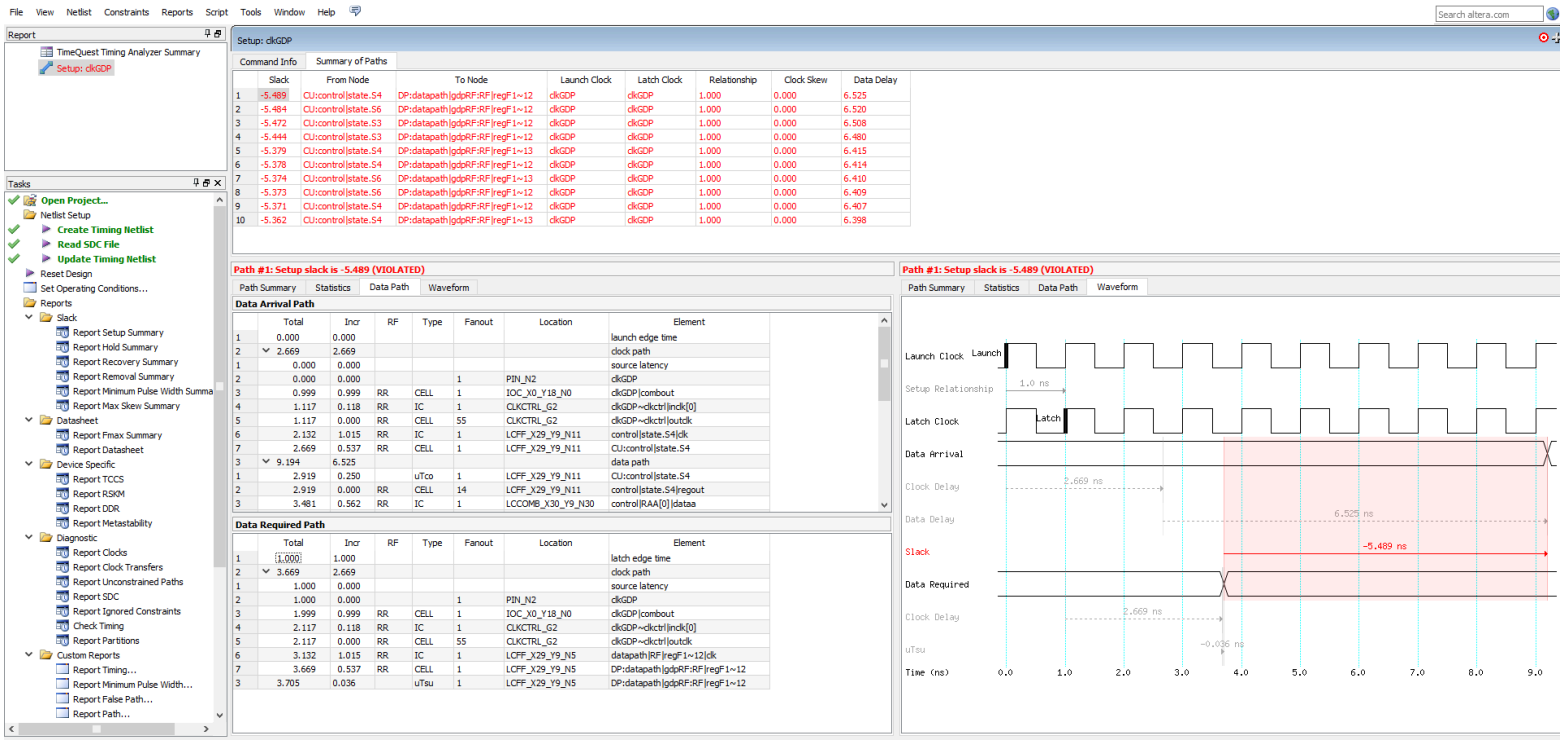
These are my VCS waveforms for the general datapath for the GCD algorithm. The clock is a yellow bar because there are several instantiations of clock within the full waveform view. The yellow line for the GCDTest represents "Z" as "Z" would only be outputted if there is no GCD.

```
Type:  Severity:  Code:  All
Chronologic VCS simulator copyright 1991-2017
Contains Synopsys proprietary information.
Compiler version N-2017.12-SP2-14; Runtime version N-2017.12-SP2-14;  Oct 20 22:36 2021
VCD+ Writer N-2017.12-SP2-14 Copyright (c) 1991-2017 by Synopsys Inc.
The file '/home/batuj1/Fall2021/lab8/inter.vpd' was opened successfully.
InputA: 00000000  InputB: 00000000  Expected: n =  0  Calculated: n =  0:
Correct!
InputA: 00010110  InputB: 00010110  Expected: n = 22  Calculated: n = 22:
Correct!
InputA: 00000010  InputB: 00000100  Expected: n =  2  Calculated: n =  2:
Correct!
InputA: 10101010  InputB: 01010101  Expected: n = 85  Calculated: n = 85:
Correct!
InputA: 11111111  InputB: 00001010  Expected: n =  5  Calculated: n =  5:
Correct!
```

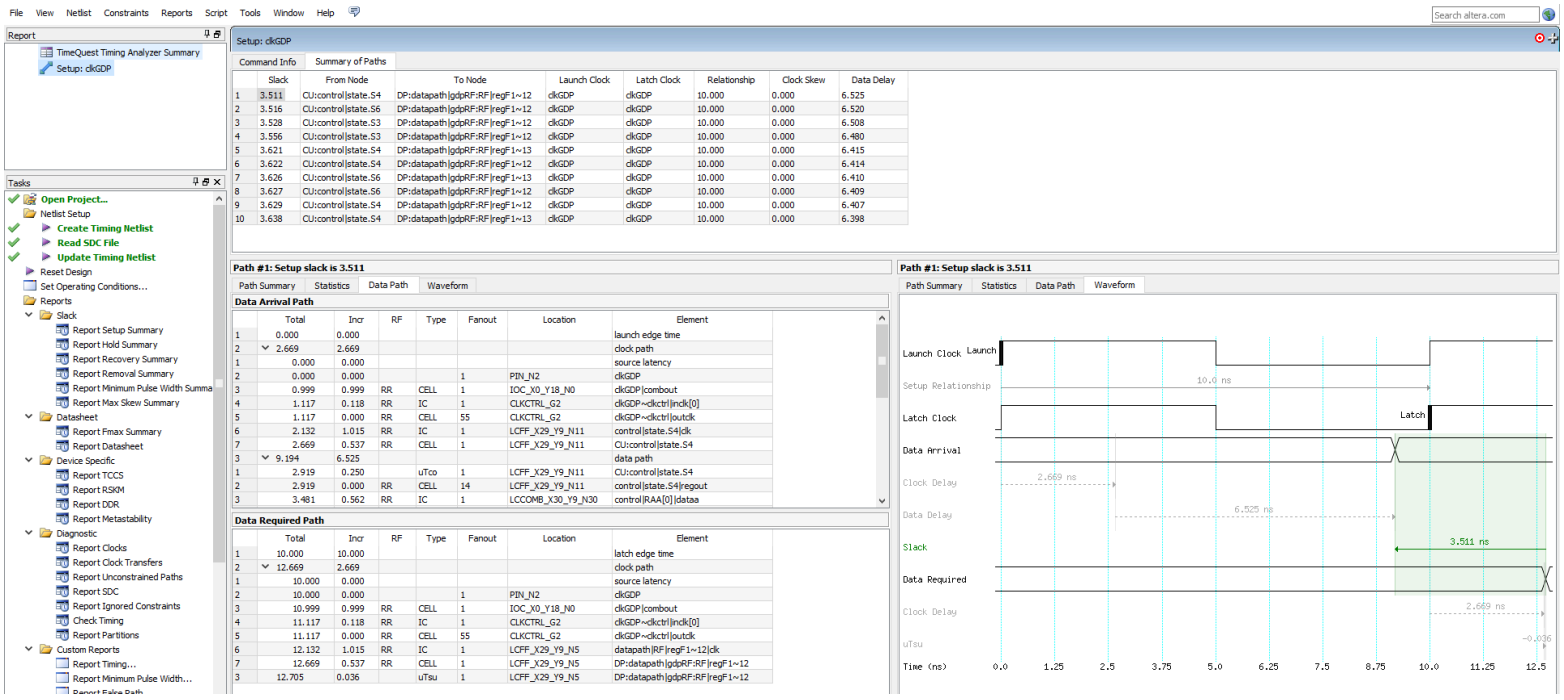
This is my VCS console for the general datapath for the GCD algorithm.

https://drive.google.com/file/d/1CBf2H-Cdq6xv_JaxTidzjIVeSvzgOlaM/view?usp=sharing
This is the link to my video delivery of the GCD general datapath implemented onto the DE2 board.

EXPERIMENT 2



This is the timing analysis of Experiment 1 before setting up the constraints.



This is the timing analysis of Experiment 1 after setting up the constraints.

EXPERIMENT 3

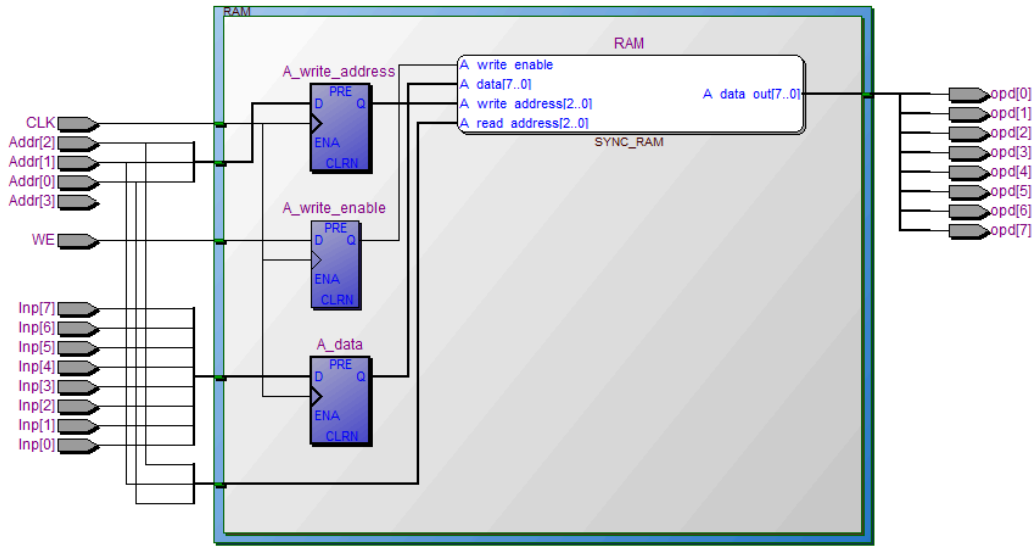
```
module sngPrtRAM (WE, CLK, Inp, Addr, opd);
	input WE, CLK;
	input [3:0] Addr;
	input [7:0] Inp;
	output wire [7:0] opd;

	//16x8 ram
	reg [15:0] RAM [0:7];

	always@(posedge CLK)
		begin
			if(WE == 1'b1)
				RAM[Addr] <= Inp;
		end

	assign opd = RAM[Addr];
endmodule
```

This is my Verilog code for the single port RAM.



This is my RTL View for the single port RAM.

https://drive.google.com/file/d/1BAPfnyEVVPpajUcIp9QMVUPvnrNUuBL6/view?usp=sharing
This is the link for my video delivery of the single port RAM implementation onto the DE2 board.

4. Answers to questions

Question 1:

```
input RE, CLK
input Addr
output read

reg RAM

always @ (posedge CLK)
    if (RE == 1'b1)
        read <= RAM[Addr]
    else
        read <= 0
```

Question 2:

Memory initialization in Quartus II is actually created through a Memory Initialization File (.mif).

Purposes:

- Specifies initial content of a memory block
    - Specifies the initial values for each address
    - Used during project compilation or simulation
- Serves as input file for memory initialization in compiler and simulator
- Can be used in Hexadecimal file to provide memory initialization data
- Contains initial values for each address in memory
- Specifies memory depth and width values
    - Specifies data radixes in different representations (binary, hexadecimal, decimal, etc.)
- Overall purpose of initializing memory is to ensure proper range of memory is to be used within a program to prevent any overflow or underflow when coding
    - Overflow or underflow of memory will result into a program to crash as not enough memory is allocated for any functions to process

5. Conclusions & Summary

Experiment 1 and the last prelab question were the hardest parts in regards to the lab. The main issue that I kept having was using two inputs for the general datapath because we have been using only one input for almost all of the labs. In addition, I had a hard time implementing the ALUout to determine if a is equal to b. To solve these issues, I kept rewatching the lab video over and over to try to fully understand what was meant to be achieved from my problems. Eventually, I was able to figure out my Verilog code and register file implementations with the two different inputs. I am very familiar with the general data path after this lab, and I am getting a much better understanding of how timing and delay can affect the overall performance of Verilog code. In addition, this lab definitely assisted my understanding of the importance of RAM, and it's implementation in Verilog code.