**EECE 2323 Digital Logic Design Lab Report**

Lab 8 Adding Branch Logic to the Datapath to Complete Your

Computer

| Student Name:  Ousmane Toure & Kaitlyn O’Flaherty | Section #: 1 |
| --- | --- |
| Instructor Name: Dr. Aboelela  Lab TA Name: Keshav Bharadwaj Vaidyanathan, Srinidi Somasundaram Subbulakshmi, Yuchao Su | Date:08/19/2022 |

1. **Background & Purpose**:

In this experiment, we implemented branch logic into the datapath we created in the previous labs to complete our computer. The objective was to rewrite the program counter to support branching, as well as set the seven segment display (SSD) to the datapath to display the output of the ALU instructions (already set by default) . From our previous labs, our instruction memory was instantiated and the pc counter is able to increment normally when branch is not being executed, and the ALU performs all necessary operations to generate new values and needs a register file to remember and store these values in memory. The data memory is the storage place for saved data and the instruction decoder divides the machine code into its necessary pieces to perform these operations. The results were then displayed utilizing the VIO Dashboard on Vivado and the LEDs on the add on board of the PYNQ. The branching instructions are a vital part of the CPU because it allows the programmer to make more complicated programs using labels and loops. The goal of this lab was to create a program counter that supports branching, then utilize verilog code in vivado to create, simulate, and physically implement the new datapath. Creating these modules and instantiations in our CPU enhances confidence and proficiency in vivado and computational architecture.

This lab is important to the scientific community because each processor needs to be able to have conditional statements in the programs to address certain operations and more complicated functions.

1. **Pre-Lab Response:**

**PC Module:**

module pc\_log(

input clk, rst, take\_branch,

input [7:0] immediate,

output reg [7:0] pc);

always@( posedge clk, posedge rst)

begin

//pc <= rst? 8'b0 : (pc + ( take\_branch ? 8'b1: immediate));

if (rst) pc <= 8'b0;

else if (take\_branch) pc <= pc + immediate;

else pc <= pc+1;

end

endmodule

**Assembly: 5 \* -2**

clr $0

clr $1

clr $2

clr $3

addi $0, $0, 0x05 #5

addi $1, $1, 0xFE #-2

andi $2, $1, 0x80

beq $2, $3, loop

inv $0, $0

addi $0, $0, 0x01 #2s complement num 1

inv $1, $1

addi $1,$1, 0x01 #2s complement num 2

sw $1, 0x08($0)

sw $0, 0x06($0)

loop:

clr $2

#counter

while:

beq $2, $1, done

add $3, $3, $0

addi $2, $2, 0x01

beq $3, $3, while

done:

clr $0

sw $3, 0x10($0)

**Assembly: 5\*5**

clr $0

clr $1

clr $2

clr $3

addi $0, $0, 0x05 #5

addi $1, $1, 0x05 #5

andi $2, $1, 0x80

beq $2, $3, loop

inv $0, $0

addi $0, $0, 0x01 #2s complement num 1

inv $1, $1

addi $1,$1, 0x01 #2s complement num 2

sw $1, 0x08($0)

sw $0, 0x06($0)

loop:

clr $2

#counter

while:

beq $2, $1, done

add $3, $3, $0

addi $2, $2, 0x01

beq $3, $3, while

done:

clr $0

sw $3, 0x10($0)

**Assembly: -2 \* 8**

clr $0

clr $1

clr $2

clr $3

addi $0, $0, 0xFE #-2

addi $1, $1, 0x08 #8

andi $2, $1, 0x80

beq $2, $3, loop

inv $0, $0

addi $0, $0, 0x01 #2s complement num 1

inv $1, $1

addi $1,$1, 0x01 #2s complement num 2

sw $1, 0x08($0)

sw $0, 0x06($0)

loop:

clr $2

#counter

while:

beq $2, $1, done

add $3, $3, $0

addi $2, $2, 0x01

beq $3, $3, while

done:

clr $0

sw $3, 0x10($0)

**Assembly: -5 \* -4**

clr $0

clr $1

clr $2

clr $3

addi $0, $0, 0xFE #-5

addi $1, $1, 0xFC #-4

andi $2, $1, 0x80

beq $2, $3, loop

inv $0, $0

addi $0, $0, 0x01 #2s complement num 1

inv $1, $1

addi $1,$1, 0x01 #2s complement num 2

sw $1, 0x08($0)

sw $0, 0x06($0)

loop:

clr $2

#counter

while:

beq $2, $1, done

add $3, $3, $0

addi $2, $2, 0x01

beq $3, $3, while

done:

clr $0

sw $3, 0x10($0)

1. **Summary of Design Implementation**
   1. **Results and Analysis:**

When conducting this experiment, the single-cycle datapath from Lab 7 was utilized and updated to include branch logic. The implementation was run by creating machine code from the prelab seen above, which was then uploaded to the instruction memory to be utilized and tested. The machine code resulted in the correct execution of commands being run and displayed onto the add-on board of the PYNQ. The instruction included following operations : loading values to registers, checking the values to see if they were negative, making a value the argument to a loop, calling a loop to add the other value to itself that number of times, to result in the multiplication of two signed numbers. All video files have been uploaded separately to show the code working correctly under all conditions. After programming our board and connecting the add on board, we opened the virtual input output (VIO) dashboard which allowed us to test our values as our PYNQ board did not have enough physical inputs. The result of the test instructions being run is the instruction being inputted and the resultant operation being completed, which is shown on the VIO in the files attached. The branching instructions were tested using the VIO dashboard and the resulting screenshots were attached. All tests resulted in values that were consistent with our pre lab truth table highlighting that our circuit was in fact correct. You could face many errors when conducting this lab. For example, when creating your test code, if you incorrectly have the wrong hex or binary value for your machine code, you could cause various inconsistencies in your data. Another error could have been using a MIPS register instead of the 4 registers ($0-$3) that we have access to in our partial CPU. An error we made was using the value 10000000 instead of 0x80 to check for negativity.

**Conclusion & Recommendations:**

Based on our results, we can conclude that Lab 8 consists of creating and implementing the necessary program counter to support the branching instruction into a single-cycle datapath. Synthesizing its implementation virtually by generating a bitstream confirmed that our verilog code had no errors which gave us the green light to program the PYNQ Board and test our code using Virtual Input Output (VIO) ports. The lab resulted in successfully being able to execute branching instructions stored in our instruction memory, and increment the instruction properly/ find the correct target address using the program counter. The IM was able to give instructions to our central processing unit through the instruction decoder, allowing better functionality to complete different arithmetic and logical functions like addition and inversion, as well as loading and storing data and addresses. These tools are crucial for a Central Processing Unit, and gives it the ability to perform varying tasks of different degrees. Completing this lab highlighted the importance of the IM and PC with branching.

Recommendations going forward would be to give a set of machine code that is standard across the class, this allows more collaboration between groups and students. Also giving set instructions for our multiplier that we would translate into assembly language is recommended.

**\*VIDEOS OF DEMONSTRATION ATTACHED SEPARATELY THRU CANVAS.**

**Appendix A: Top File**

Top File:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer: Majid Sabbagh (sabbagh.m@husky.neu.edu)

//

// Create Date: 08/17/2014 02:18:36 PM

// Design Name:

// Module Name: eightbit\_alu\_top

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module pdatapath\_top(

input wire clk, // General clock input

input wire top\_pb\_clk, // PBN1 clock input

input wire rst\_general, // PBN0 clock reset for memory blocks

output [7:0] led, // add-on board led[5:0], + LD0, LD1

output wire ovf\_ctrl, // LD3 for overflow

output [3:0] disp\_en, // 7-Segment display enable

output [6:0] seg7\_output // 7-segment display output

);

// ALU interface

wire [7:0] alu\_input1, alu\_input2;

wire [7:0] alu\_output;

wire [2:0] ALUOp;

wire alu\_ovf;

wire take\_branch ;

wire [15:0] instruction;

//instruction fields

wire [3:0] opcode;

wire [1:0] rs\_addr;

wire [1:0] rt\_addr;

wire [1:0] rd\_addr;

wire [7:0] immediate;

//control signals

wire RegDst;

wire RegWrite;

wire ALUSrc1;

wire ALUSrc2;

wire MemWrite;

wire MemToReg;

wire [1:0] regfile\_WriteAddress;//destination register address

wire [8:0] regfile\_WriteData;//result data

wire [8:0] regfile\_ReadData1;//source register1 data

wire [8:0] regfile\_ReadData2;//source register2 data

wire [8:0] alu\_result;

wire [8:0] Data\_Mem\_Out;

wire [7:0] zero\_register;

// PC and debounce clock

wire [7:0] pc;

wire pb\_clk\_debounced;

assign zero\_register = 8'b0; //ZERO constant

assign alu\_result = {alu\_ovf, alu\_output};

// Assign LEDs

assign led = alu\_output;

assign ovf\_ctrl = alu\_ovf;

// Debounce circuit

debounce debounce\_clk(

.clk\_in(clk),

.rst\_in(rst\_general),

.sig\_in(top\_pb\_clk),

.sig\_debounced\_out(pb\_clk\_debounced)

);

// 7-Segment display module

Adaptor\_display display(

.clk(clk), // system clock

.input\_value(alu\_output), // 8-bit input [7:0] value to display

.disp\_en(disp\_en), // output [3:0] 7 segment display enable

.seg7\_output(seg7\_output) // output [6:0] 7 segment signals

);

//Instantiate Your PC Register here

pc\_log pc\_log(.clk(pb\_clk\_debounced), .rst(rst\_general), .take\_branch(take\_branch), .immediate(immediate), .pc(pc));

//Instantiate Your instruction Memory here

instr\_mem im(

.a(pc), // input wire [7 : 0] a

.spo(instruction) ); // output wire [15 : 0] spo

//Instantiate Your instruction decoder here

inst\_decoder id(.instruction(instruction), .opcode(opcode), .rs\_addr(rs\_addr), .rt\_addr(rt\_addr), .rd\_addr(rd\_addr), .immediate(immediate), .RegDst(RegDst), .RegWrite(RegWrite), .ALUSrc1(ALUSrc1),

.ALUSrc2(ALUSrc2), .ALUOp(ALUOp), .MemWrite(MemWrite), .MemToReg(MemToReg));

//Instantiate Your alu-regfile here

regfile rf(.rd0\_data(regfile\_ReadData1),.rd1\_data(regfile\_ReadData2),.wr\_data(regfile\_WriteData),.rd0\_addr(rs\_addr),.rd1\_addr(rt\_addr),.wr\_addr(regfile\_WriteAddress),.wr\_en(RegWrite),.clk(pb\_clk\_debounced),.rst(rst\_general));

Mux m1(.in1(regfile\_ReadData1),.sel(ALUSrc1),.in2(zero\_register),.out(alu\_input1)); //instantiate template

Mux m2(.in1(regfile\_ReadData2),.sel(ALUSrc2),.in2(immediate),.out(alu\_input2)); //instantiate template

eightbit\_alu a1(.a(alu\_input1),.b(alu\_input2),.s(ALUOp),.f(alu\_output),.ovf(alu\_ovf),.take\_branch(take\_branch));

//Instantiate Your data memory here

data\_memory data (

.a(alu\_output), // input wire [7 : 0] a

.d(regfile\_ReadData2), // input wire [8 : 0] d

.clk(pb\_clk\_debounced), // input wire clk

.we(MemWrite), // input wire we

.spo(Data\_Mem\_Out)); // output wire [8 : 0] spo

//Mux for regfile\_writedata

Mux m3(.in1(alu\_result),.sel(MemtoReg),.in2(Data\_Mem\_Out),.out(regfile\_WriteData));

//Mux for RegDST

Mux m4(.in1(rt\_addr),.sel(RegDst),.in2(rd\_addr),.out(regfile\_WriteAddress)); //instantiate template

//Instantiate Your VIO core here

vio\_0 your\_instance\_name (

.clk(clk), // input wire clk

.probe\_in0(alu\_output), // input wire [7 : 0] probe\_in0

.probe\_in1(alu\_ovf), // input wire [0 : 0] probe\_in1

.probe\_in2(take\_branch), // input wire [0 : 0] probe\_in2

.probe\_in3(regfile\_ReadData1), // input wire [7 : 0] probe\_in3

.probe\_in4(regfile\_ReadData2), // input wire [7 : 0] probe\_in4

.probe\_in5(alu\_input1), // input wire [7 : 0] probe\_in5

.probe\_in6(alu\_input2), // input wire [7 : 0] probe\_in6

.probe\_in7(regfile\_WriteData), // input wire [8 : 0] probe\_in7

.probe\_in8(Data\_Mem\_Out), // input wire [8 : 0] probe\_in8

.probe\_in9(opcode), // input wire [3 : 0] probe\_in9

.probe\_in10(rs\_addr), // input wire [1 : 0] probe\_in10

.probe\_in11(rt\_addr), // input wire [1 : 0] probe\_in11

.probe\_in12(rd\_addr), // input wire [1 : 0] probe\_in12

.probe\_in13(immediate), // input wire [7 : 0] probe\_in13

.probe\_in14(RegDst), // input wire [0 : 0] probe\_in14

.probe\_in15(RegWrite), // input wire [0 : 0] probe\_in15

.probe\_in16(ALUSrc1), // input wire [0 : 0] probe\_in16

.probe\_in17(ALUSrc2), // input wire [0 : 0] probe\_in17

.probe\_in18(ALUOp), // input wire [2 : 0] probe\_in18

.probe\_in19(MemWrite), // input wire [0 : 0] probe\_in19

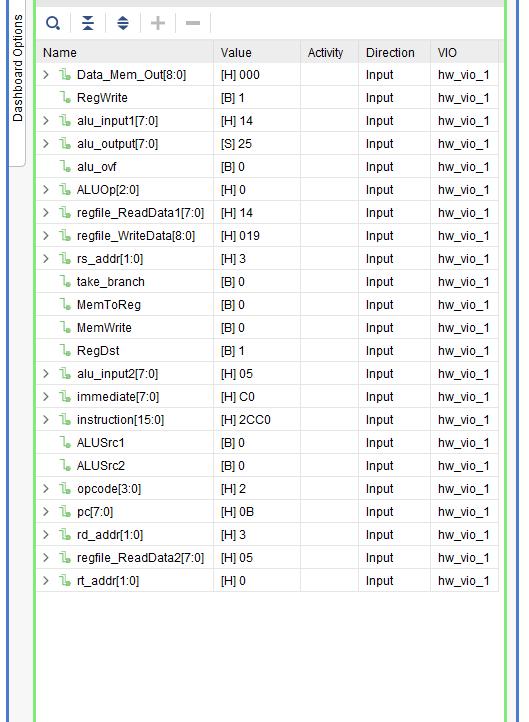
.probe\_in20(MemToReg), // input wire [0 : 0] probe\_in20

.probe\_in21(pc), // input wire [7 : 0] probe\_in21

.probe\_in22(instruction) // input wire [15 : 0] probe\_in22

);

endmodule



5 \* 5 =25