**EECE 2323 Digital Logic Design Lab Report**

Lab 7 Adding Instruction Memory and Program Counter

| 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:08/09/2022 |

1. **Background & Purpose**:

In this experiment, we implemented an instruction memory into the datapath we created in the previous Labs. The objective was to have stored instructions which could be executed by the previous components of our processor created as well as incremented by the program counter. From our previous labs, our 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 instruction memory and program counter are vital parts of the CPU because it gives the processor instructions to execute and the PC keeps track of which instructions have been and need to be executed. The goal of this lab was to create an instruction set for our instruction memory, create a PC to update the instruction addresses, 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 instructions stored to be able to execute commands, as well as knowing which instruction needs to be executed next. In turn, the CPU can receive the correct instructions, process the instructions correctly and then output the desired result. This allows the processor to quickly access information, and without memory a computer would not be able to function properly.

1. **Pre-Lab Response:**

| **Instruction (MIPS)** | **Task** |
| --- | --- |
| **clr $0 #clearing all addresses to start from shifted address**  **clr $1**  **clr $2**  **clr $3**  **# initialize address 0x4 with 0x10 and 0x5 wih 0x0F**  **addi $1, $0, 0x10**  **addi $2, $0, 0x0F**  **sw $1, 0x4($0)**  **sw $2, 0x5($0)** | Initialize addresses 0x4 and 0x5 of the Data Memory with values 0x10 and 0x0F respectively. |
| **clr $0 #clearing all addresses to start from shifted address**  **clr $1**  **clr $2**  **clr $3**  **lw $1, 0x5($0) #add 0x4 and 0x5 sum into 0x11**  **lw $2, 0x4($0)**  **add $3, $1, $2**  **sw** $3**, 0x11($0)** | Add the numbers you stored in addresses 0x4 and 0x5 and store the sum in address 0x11. |
| **clr $0 #clearing all addresses to start from shifted address**  **clr $1**  **clr $2**  **clr $3**  **lw $1, 0x5($0) #0x12 store two's complement 0x5**  **inv $2, $1**  **addi $3, $2, 0x01**  **sw $3, 0x12($0)** | Store in address 0x12 the two’s complement of the number stored in address 0x5. |
| **clr $0 #clearing all addresses to start from shifted address**  **clr $1**  **clr $2**  **clr $3**  **lw $1, 0x4($0) # subtract 0x12 from 0x4 store in 0x13**  **lw $2, 0x12($0)**  **inv $1, $1**  **addi $1, $1, 0x1**  **add $3, $1, $2**  **sw $0, 0x13($0)** | Subtract the number stored in 0x12 from the number in 0x4. Store the result in address 0x13. |

**Output:**

**memory\_initialization\_radix=16;**

**memory\_initialization\_vector=d000,d140,d280,d3c0,3110,320f,1104,1205,d000,d140,d280,d3c0,0105,0204,26c0,1311,d000,d140,d280,d3c0,0105,4180,3b01,1312,d000,d140,d280,d3c0,0104,0212,4140,3501,26c0,1013;**

**\*Output file also submitted separately from Report (name: machine\_code.coe)**

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

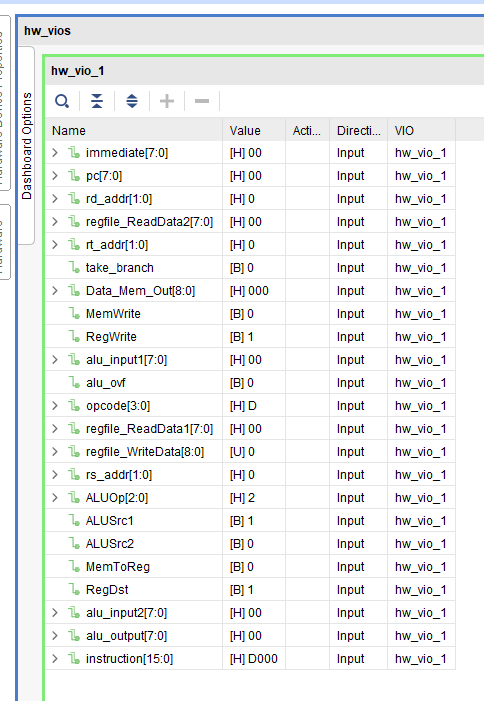
When conducting this experiment, the single-cycle datapath from Lab 6 was utilized and updated to include an instruction memory and program counter. The simulation was run by creating machine code from the prelab table seen above, which was then uploaded to the instruction memory to be utilized and tested. 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 both addresses and data to a register, storing to a register and data memory, arithmetic add of two register elements, adding an constant value to a registers value, inverting a reg value, and clearing the registers. All files can be found in Appendix B. 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 Appendix A. Storing values in the register and all of the necessary ALU outputs were tested using the VIO dashboard and the resulting screenshots were placed in Appendix B. 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 was adding addresses in registers rather than the data in those addresses.

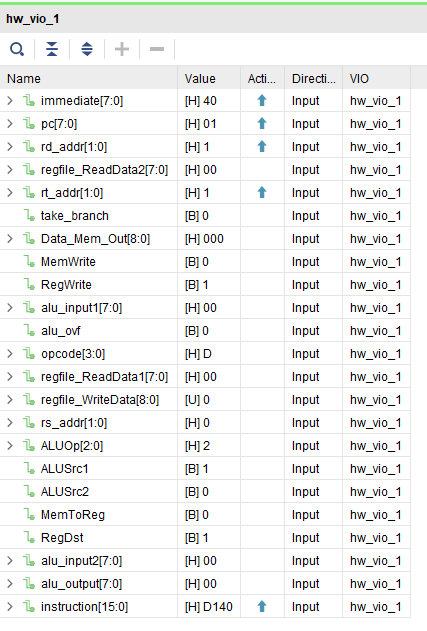
* 1. **Conclusion & Recommendations:**

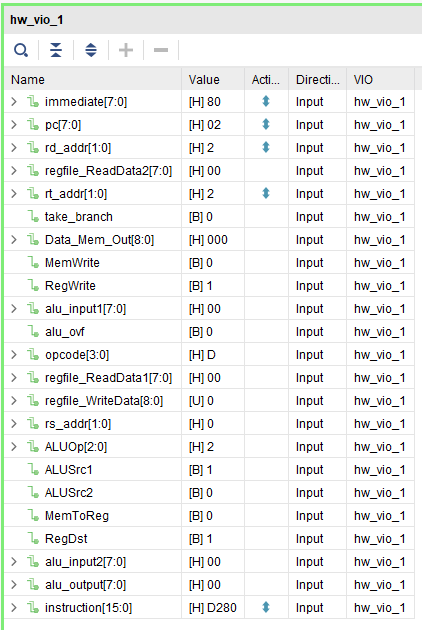
Based on our results, we can conclude that Lab 7 consists of creating and implementing an instruction memory and program counter into a single-cycle datapath. Synthesizing its implementation virtually by generating a bitstream confirmed that our verilog code had no errors which gave us the green light to program the PYNQ Board and test our code using Virtual Input Output (VIO) ports. The lab resulted in successfully being able to retrieve instructions stored in our instruction memory, and increment the instruction properly 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.

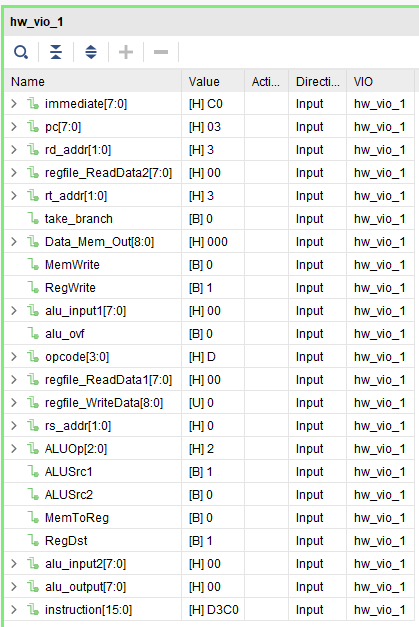
Recommendations going forward would be to give a set of machine code that is standard across the class, this allows more collaboration between groups and students. Furthermore.

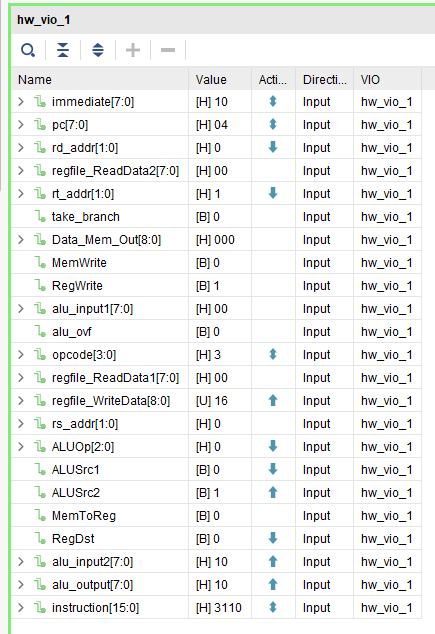
**Appendix A: Screenshot of VIO Outputs**

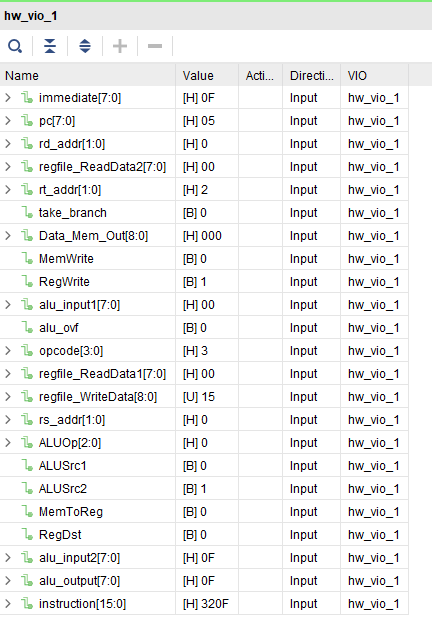




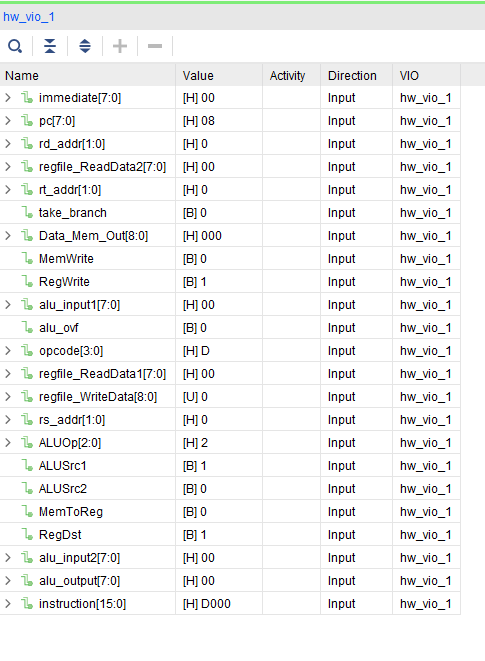


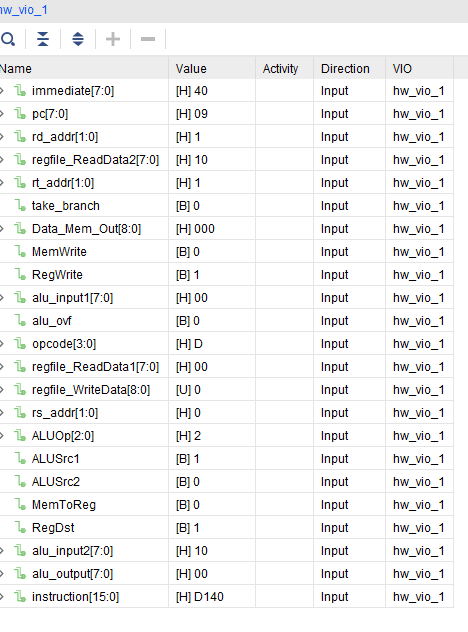




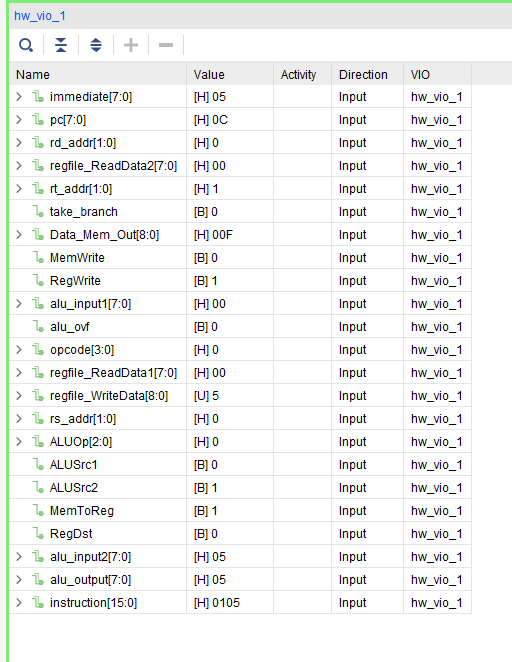












**Appendix B: Design Program Files (Verilog modules, etc)**

`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 inteface

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;

//insturction 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 debouce 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\_counter count(.clk(pb\_clk\_debounced), .rst(rst\_general), .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

alu a1(.a(alu\_input1),.b(alu\_input2),.sel(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 vio (

.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