Lab #6 & 7: Adding Instruction Decoding to the Datapath

Adding Instruction Memory and Program Counter to Your  
Computer

EECE 2323 – Prof. Xiaolin Xu

Date: 03/25/2022

Name: Fengyi Quan && Yichen Sun

Northeastern University

1. **Background & Purpose**

The objective of this lab is to implement instruction decoder and instruction memory to the datapath. The instruction decoder is a combinational logic, and a clock edge is only needed for writing to the register file and writing to memory in the datapath. It will be implemented a minimal instruction set architecture with a MIPS-encoding.

Then, a program counter is implemented to address the instruction memory. The instruction memory is 16 bits wide by 256 entries. An instruction for the lab computer is 16 bits wide, and 16 bits is defined as a word in the IM.

1. **Prelab6**

**2.1 Instruction Decoder: Control Signal**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | Opcode | RegDst | RegWrite | ALUSrc1 | ALUSrc2 | ALUOp[2:0] | MemWrite | MemToReg |
| lw | 0000 | 1’b0 | 1’b1 | 1’b0 | 1’b1 | 3’b000 | 1’b0 | 1’b1 |
| sw | 0001 | 1’b0 | 1’b0 | 1’b0 | 1’b1 | 3’b000 | 1’b1 | 1’b0 |
| add | 0010 | 1’b1 | 1’b1 | 1’b0 | 1’b0 | 3’b000 | 1’b0 | 1’b0 |
| addi | 0011 | 1’b0 | 1’b1 | 1’b0 | 1’b1 | 3’b000 | 1’b0 | 1’b0 |
| inv | 0100 | 1’b1 | 1’b1 | 1’b1 | 1’b0 | 3’b001 | 1’b0 | 1’b0 |
| and | 0101 | 1’b1 | 1’b1 | 1’b0 | 1’b0 | 3’b010 | 1’b0 | 1’b0 |
| andi | 0110 | 1’b0 | 1’b1 | 1’b0 | 1’b1 | 3’b010 | 1’b0 | 1’b0 |
| or | 0111 | 1’b1 | 1’b1 | 1’b0 | 1’b0 | 3’b011 | 1’b0 | 1’b0 |
| ori | 1000 | 1’b0 | 1’b1 | 1’b0 | 1’b1 | 3’b011 | 1’b0 | 1’b0 |
| sra | 1001 | 1’b0 | 1’b1 | 1’b0 | 1’b1 | 3’b100 | 1’b0 | 1’b0 |
| sll | 1010 | 1’b0 | 1’b1 | 1’b0 | 1’b1 | 3’b101 | 1’b0 | 1’b0 |
| beq | 1011 | 1’b0 | 1’b0 | 1’b0 | 1’b0 | 3’b110 | 1’b0 | 1’b0 |
| bne | 1100 | 1’b0 | 1’b0 | 1’b0 | 1’b0 | 3’b111 | 1’b0 | 1’b0 |
| clr | 1101 | 1’b0 | 1’b1 | 1’b1 | 1’b0 | 3’b010 | 1’b0 | 1’b0 |

**2.2 Instruction Decoder: Verilog Module**

Please see below Appendix A: Instruction Decoder Module for more details.

**2.3 Translate the Steps to Instructions**

Generate and store a value to data memory  
---------------

* Instruction 1:   
  1- Press the right button to reset everything  
  2- Change the regfile\_read\_address2 to 1  
  3- Set ALUSrc2 to 0  
  4- Set the ALUOp to 1 (for 'inv' operation)  
  5- Set MemToReg to 0  
  6- Change the regfile\_write\_address2 to 1  
  7- Set RegWrite to 1  
  8- Reset RegWrite back to 0

inv $1, $1

* Instruction 2:  
  1- Change the regfile\_read\_address1 to 1  
  2- Change the regfile\_write\_address to 1  
  3- Change the instr\_i to 8'h03  
  4- Change the ALUOp to 0x5 (for 'sll' operation)  
  5- Set RegWrite to 1  
  6- reset RegWrite back to 0

sll $1, $1, 0x 3

* Instruction 3:   
  1- Set the ALUSrc2 to 1  
  2- Set the value of instr\_i to 0xFF  
  3- Set the ALUOp to 0 (for 'add' operation)  
  4- Set the regfile\_read\_address1 to 3  
  5- Change the regfile\_read\_address2 to 1  
  6- Set MemWrite to 1  
  7- Reset MemWrite back to 0

sw $1, 0xFF($3)

Load the stored value from data memory  
---------------

* Instruction 4:   
  1- Set the value of instr\_i to 0xFF  
  2- Set the ALUSrc2 to 1  
  3- Change the regfile\_read\_address1 to 3  
  4- Set the ALUOp to 0 for 'add'  
  5- Set MemToReg to 1  
  6- Set regfile\_write\_address to 2  
  7- Set RegWrite to 1  
  8- Set RegWrite to 0

lw $2, 0xFF($3)

To check if the correct value is actually loaded to register file at   
address 0x2 or not:  
---------------

* Instruction 5:   
  1- Change the regfile read address1 to 2  
  2- Change the ALUOp to 3'b011  
  3- Change the instr\_i to 8'hF0  
  4- Change the ALUSrc2 to 1  
  5- Change the regfile write address to 2  
  6- Change the RegWrite to 1  
  7- Change the RegWrite to 0

ori $2, $2, 0xF0

**2.4 Generate the Machine Codes**

|  |  |  |
| --- | --- | --- |
| Type | Instruction | Binary Machine Code |
| R | inv $1, $1 | 0100\_00\_01\_01\_000000 |
| I | sll $1, $1, 3 | 1010\_01\_01\_00000011 |
| I | sw $1, 0xFF($3) | 0001\_01\_11\_11111111 |
| I | lw $2, 0xFF($3) | 0000\_10\_11\_11111111 |
| I | ori $2, $2, 0xF0 | 1000\_10\_10\_11110000 |

1. **Prelab7**

**3.1 Straight Line Assembly Program**

Text

Description automatically generated

**3.2 Generate the Machine Codes using the Assembler**

Graphical user interface

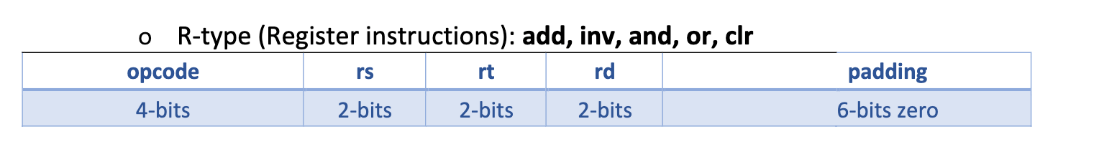
Description automatically generated with low confidence

1. **Design Implementation**

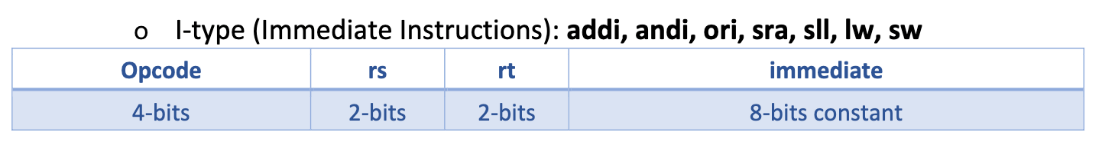
**4.1 Instruction Set**

For our 8-bit datapath, it has an instruction set architecture with the fixed-length of 16bits instructions. It follows the convention of MIPS architecture with three types of instructions, R-type, I-type, and J-type. Since the implementation only has 4 registers, which can be encoded in 2-bit address. The address for source and destination both are used only two bits. And because of limited instruction set, we can encode all instruction opcode in four bits.

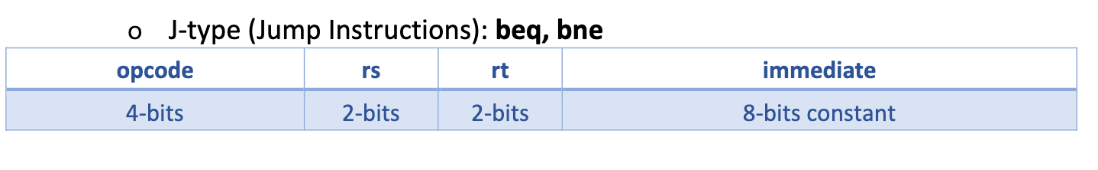
The R-type instruction includes add, inv, and, or, clr. The following table indicates the setup and format of a R-type instruction.



The I-type instruction includes addi, andi, ori, sra, sll, lw, sw. The following table indicates the setup and format of a I-type instruction.



The J-type instruction includes beq, bne. The following table indicates the setup and format of a J-type instruction.



The I-type and J-type have the same format of machine code. Immediate field in I-type indicates a numeric literal used by the ALU to perform the computation. And immediate field in J-type indicates the address to the jump destination.

The implementation of the instruction decoder uses a case statement to check which instruction it is using the following table.

|  |  |
| --- | --- |
| Instruction | Opcode |
| lw | 0000 |
| sw | 0001 |
| add | 0010 |
| addi | 0011 |
| inv | 0100 |
| and | 0101 |
| andi | 0110 |
| or | 0111 |
| ori | 1000 |
| sra | 1001 |
| sll | 1010 |
| beq | 1011 |
| bne | 1100 |
| clr | 1101 |

**4.2 Program Counter**

For program counter, we use a asynchronous reset counter. The implementation is pretty simple. The count field increment every positive clock edge and will reset when it receives a reset signal. The count field keep track of the current instruction address to be executed. The program counter does not support branching right now and will implemented in the next lab. For implementation detail, please see Appendix G: Program Counter Module.

Text

Description automatically generated

1. **Results and Analysis**

The instruction decoder is a combinational circuit that take in instruction and configure the signals with the instruction set. This opcode table provides necessary reference on 8-bit datapath instructions as mentioned on Deign Implementation section and their corresponding control signals. The implementation of instruction decoder in this lab signal outputs connects to the combinational circuit in whole. And these signals select the corresponding register write address from the instruction is rd field or the rt field. First of all, a testbench tests the instruction decode circuitry that covers all possible types of instruction by varying opcode in Prelab opcode of Table. Then, the synthesis runs correctly, and continue by implementing the FPGA board. On the board, the BTNO connects to rst\_general to reset the memory and register file; the BTN1 adds the single step to store data in either memory or the register file.

Adding Program Counter (PC) and Instruction Memory to the design is much easier compared to the implementation in lab 6. A program counter adds a module in a combinational circuit that allows the lab computer to address the instruction memory. The design of a program counter contains two components, a PC logic takes in two inputs, a pb\_clk\_debounced and right\_pb\_rst\_general, and outputs an 8-bit instruction. Then the instruction memory works as a black box that receives pc[7:0] and converts to instruction[15:0] with machine\_codes.coe generated by the straight-line assembly program. Next, the inst\_decoder decodes the output of instruction memory and distributes bits to corresponding outputs that signal other modules in the program. Finally, the VIO module validates the correctness of the design of instruction memory. Refer VIO output results in Appendix I.

1. **Conclusion & Recommendations**

In the first half of this lab, the prelab ‘s table for opcodes and their corresponding control signals establishes the necessary workflow in an instruction decoder. The combinational circuit takes in an instruction[15:0] and correctly generate the signal outputs that write address from different type instructions.

In the second half of this lab, the instruction memory adds to the digital logic successfully and a Program Counter (PC) addresses the Instruction Memory (IM). Also, the prelab assembly program fulfills the rules of own instruction set and converts to machine codes with assembly compiler.

In lab, the reg\_file is a crucial module in lab combinational circuit. Putting assigned registers inside the clock and outside the clock makes significant difference which causes instruction execution postpone. This behavior should be noticed in the future. Moreover, the development of digital logic design is a long process, there exists many features to be implemented and errors to be fixed in the following labs.

1. **Appendices**

Appendix A: Instruction Decoder Module

Table

Description automatically generated

Graphical user interface, application

Description automatically generated

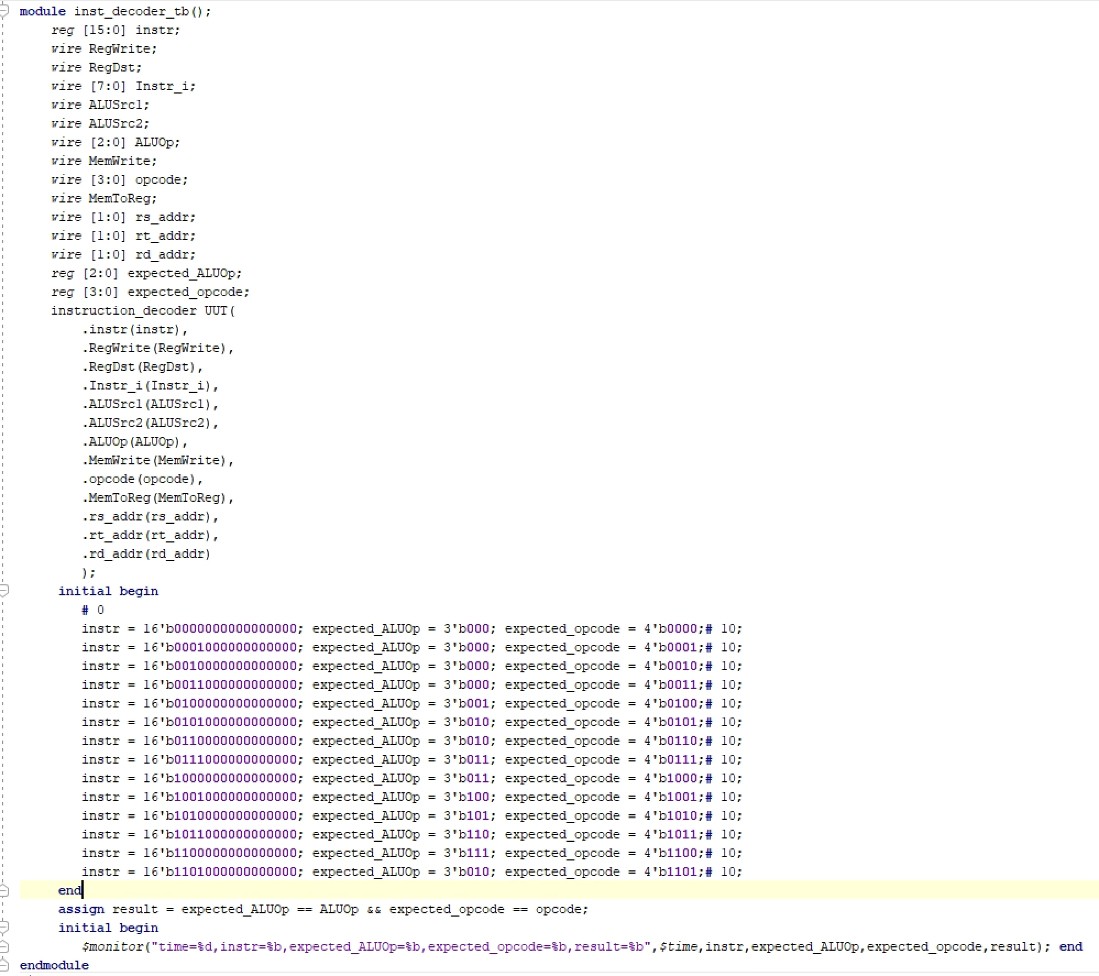
Table

Description automatically generated

Table

Description automatically generated

Appendix B: Instruction Decoder testbench

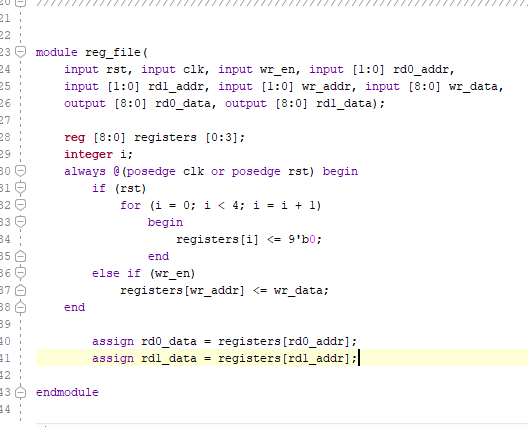


Appendix C: Instruction Decoder testbench simulation waveform

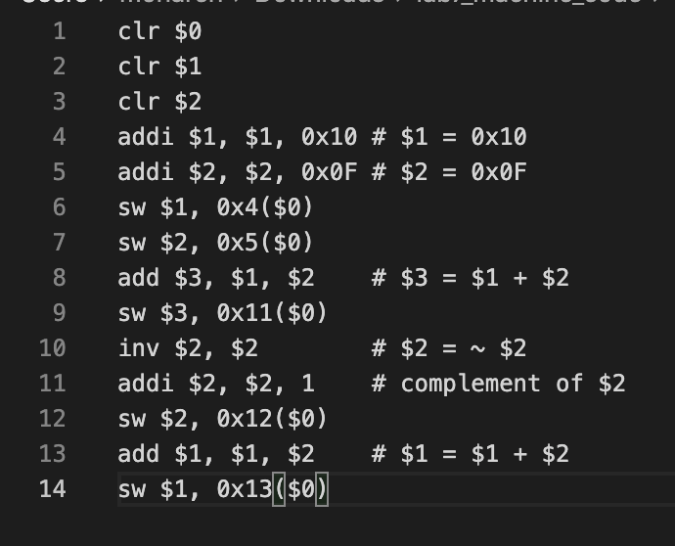
Timeline

Description automatically generated

Appendix D: Regfile Module



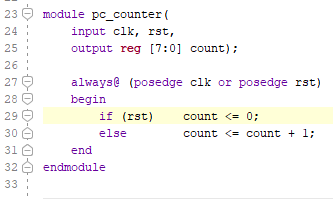
Appendix E: Assembly Program



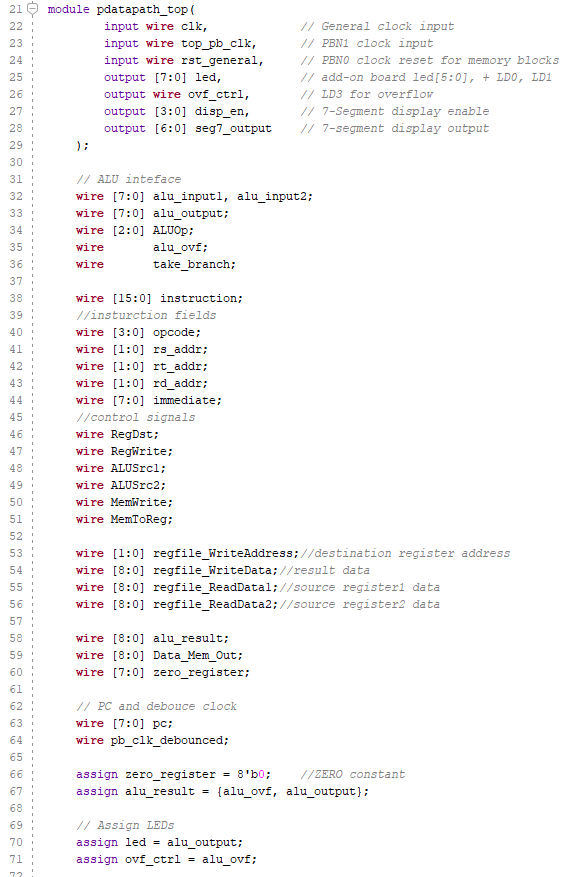
Appendix F: Machine Code

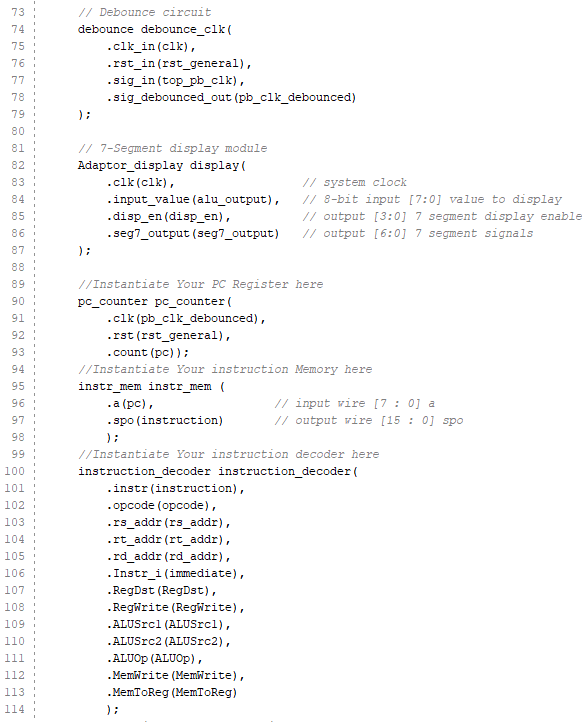
Graphical user interface

Description automatically generated with low confidence

Appendix G: Program Counter Module

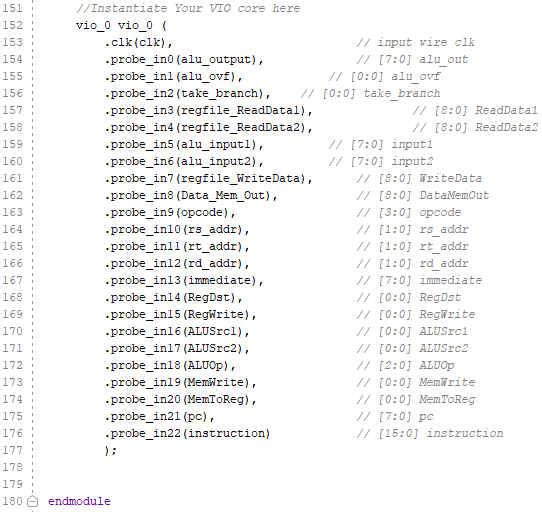
Appendix H: Pdatapath Top Module





Text

Description automatically generated with low confidence



Appendix I: Output screenshots

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, table

Description automatically generated with medium confidence

Table

Description automatically generated

Graphical user interface, table

Description automatically generated

Graphical user interface, table

Description automatically generated with medium confidence

Table

Description automatically generated

Table

Description automatically generated

Graphical user interface, table

Description automatically generated with medium confidence

Table

Description automatically generated

Graphical user interface, table

Description automatically generated

Table

Description automatically generated with medium confidence

Table

Description automatically generated

Table

Description automatically generated with medium confidence

Table

Description automatically generated