CSE 141L Final Submission

Matthew Mizumoto, A16907397. Jack Barkes, A16893784

Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

- Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
- Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

Matthew Mizumoto
Jack Barkes

0. Team

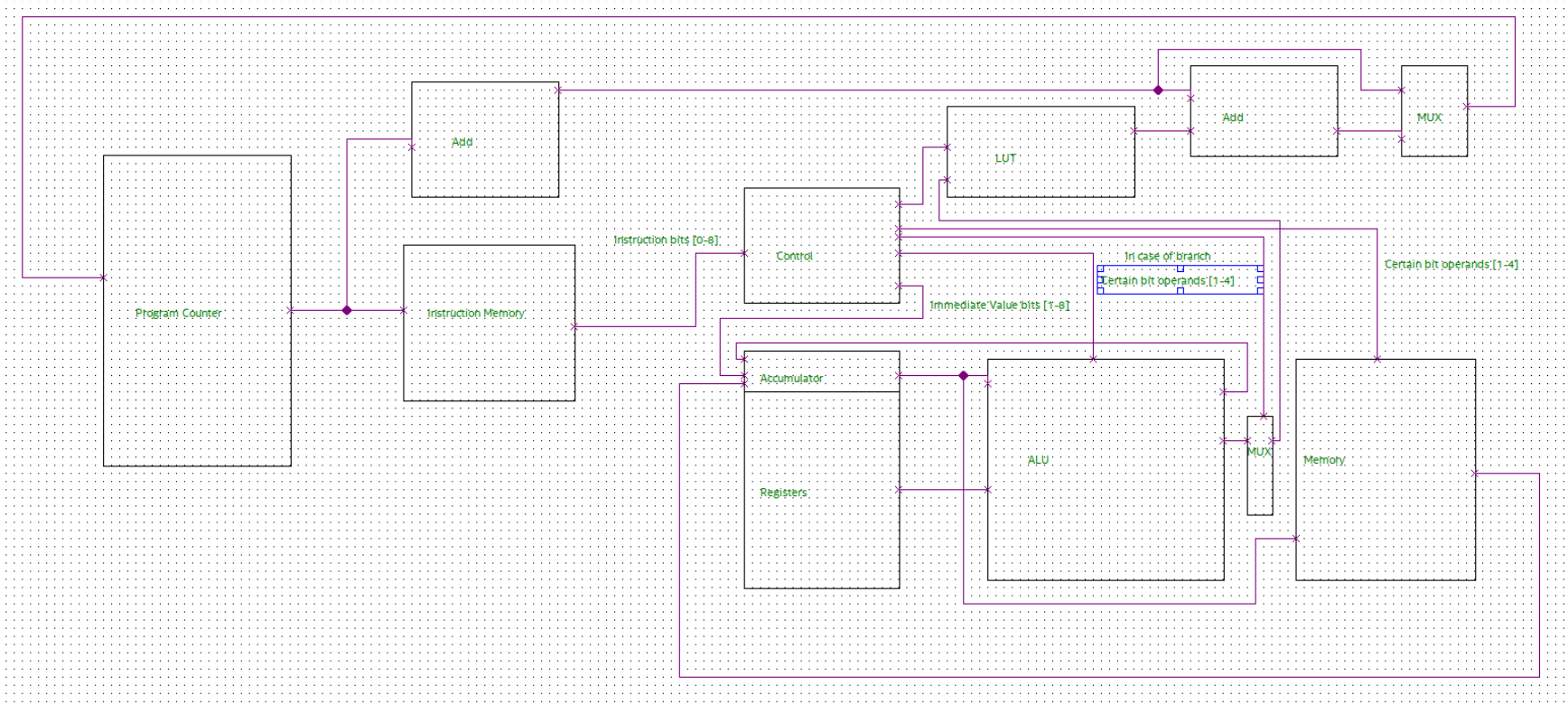
Matthew Mizumoto, Jack Barkes

1. Introduction

TODO. Name your architecture. What is your overall philosophy? What specific goals did you strive to achieve? Can you classify your machine in any of the standard ways (e.g., stack machine, accumulator, register-register/load-store, register-memory)? If so, which? If not, devise a name for your class of machine. Word limit: 200 words.
My architecture will be an accumulator type. My goal will be to rely on my accumulator register as much as possible to do any of my work. My accumulator register will get information from the memory and if I want to use other registers I will have to put data from memory into the accumulator then from the accumulator put it into another register. Although this method may be repetitive it allows for me to have more registers available to use so I believe it will help me in the long run. The name of my architecture is ZeroHero because my zero index register will be doing all the work.

2. Architectural Overview

TODO. This must be in picture form. What are the major building blocks you expect your processor to be made up of? You must have data memory in your architecture. (Example of MIPS: https://www.researchgate.net/figure/The-MIPS-architecture_fig1_251924531)


Add
Add
MUX
LUT
Program Counter
Instruction Memory
Instruction bits [0-8]
Control
In case of branch
Certain bit operands [1-4]
Certain bit operands [1-4]
Immediate Value bits [1-8]
Accumulator
Registers
ALU
MUX
Memory

3. Machine Specification

Instruction formats

Two example rows have been filled for you. When you submit, do not include the example types. Add rows as necessary. In your submission, please delete this paragraph.

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	1 bit type, 4 bits opcode, 4 bits register	and, add, sub, xor, shr, shl, sw, lw, mov, beq, bne, bgt, bge
I	1 bit type, 8 bit immediate value	imm

Operations

An example row has been filled for you. When you submit, do not include the example type. In the name column, be sure to also add the definition of what the example actually does. For example, "lsl = logical shift left" would be an appropriate value to put in the name column. In the bit breakdown column, add in parenthesis what specific values the bits should be in order. X indicates that it will be specified by the programmer's instruction itself (i.e. specifying registers). In the example column, give an example of an “assembly language” instruction in your machine, then translate it into machine code. Add rows as necessary. In your submission, please delete this paragraph.

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
and = logical and	R	1 bit type (0). 4 bits opcode (0000). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 and R1 = 0b0000_0010 # after and instruction, R0 now holds 0b0000_0010	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
add = addition	R	1 bit type (0). 4 bits opcode (0001). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 add R1 = 0b1110_1100 # after and instruction, R0 now holds 0b1110_1100	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
sub = subtraction	R	1 bit type (0). 4 bits opcode (0010). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 sub R1 = 0b1001_1000	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator

			# after and instruction, R0 now holds 0b1001_1000	
xor = logical xor	R	1 bit type (0). 4 bits opcode (0011). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 xor R1 = 0b1110_1000 # after xor instruction, R0 now holds 0b1110_1000	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
shr = logical right shift	R	1 bit type (0). 4 bits opcode (0100). 4 bits operand register (0000)	# Assume R0 (accumulator) has 0b1100_0010 R0 >> 1 = 0b0110_0001 # after shr instruction, R0 now holds 0b0110_0001	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
shl = logical left shift	R	1 bit type (0). 4 bits opcode (0101). 4 bits operand register (0000)	# Assume R0 (accumulator) has 0b1100_0010 R0 << 1 = 0b1000_0100 # after shr instruction, R0 now holds 0b1000_0100	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
sw = store word	R	1 bit type (0). 4 bits opcode (0110). 4 bits operand register (0001)	# Assume R0 has 0b0000_0000 # Assume R1 has 0b0010_1010 # after sw instruction, the value in R0 has been stored to a memory address that R15 is currently holding	This example assumes we already know the address we want to store and put that address into R0
lw = load	R	1 bit type (0). 4 bits opcode	# Assume R0 has 0b0000_0000	This example assumes we

word		(0111). 4 bits operand register (0001)	# Assume R1 has 0b0010_1010 # after lw instruction, the value in R0 will change the value found at the memory address that R15 is currently holding	already know the address we want to load and put that address into R0
Mov = move	R	1 bit type (0). 4 bits opcode (1000). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 mov R1 = 0b1100_0010 # after mov instruction, R1 now holds 0b1100_0010	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
beq = branch equal to	R	1 bit type (0). 4 bits opcode (1001). 4 bits operand register (0001)	# Assume R2 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 beq R1 = 0b0000_0000 # after beq instruction, we do not jump since the result is false	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
bne = branch not equal	R	1 bit type (0). 4 bits opcode (1010). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 beq R1 = 0b0000_0000 # after beq instruction, we jump to the memory address stored in a specific location in memory since the equation is true	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
bgt =	R	1 bit type (0). 4 bits opcode	# Assume R0 (accumulator) has	This example assumes we have

branch greater than		(1011). 4 bits operand register (0001)	0b1100_0010 # Assume R1 has 0b0010_1010 R0 beq R1 = 0b0000_0000 # after bgt instruction, we jump to the memory address stored in a specific location in memory since the equation is true	already loaded in a value from memory or done some other operations already to get some value into the accumulator
bge = branch greater equal	R	1 bit type (0). 4 bits opcode (1100). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 beq R1 = 0b0000_0000 # after bge instruction, we jump to the memory address stored in a specific location in memory since the equation is true	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
blt = branch less than	R	1 bit type (0). 4 bits opcode (1101). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 # Assume R1 has 0b0010_1010 R0 blt R1 = 0b0000_0000 # after blt instruction, we do not jump since R0 is not less than R1.	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
sbr = shift barrel right	R	1 bit type (0). 4 bits opcode (1110). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 #After it becomes 0b01100_001	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator

sbl = shift barrel left	R	1 bit type (0). 4 bits opcode (1111). 4 bits operand register (0001)	# Assume R0 (accumulator) has 0b1100_0010 #After it becomes 0b100_0101	This example assumes we have already loaded in a value from memory or done some other operations already to get some value into the accumulator
imm = immediate value	I	1 bit type (1), 8 bits immediate value (1010_1010)	# after imm instruction, R0 now stores the value 0b1010_1010	

Internal Operands

16 registers are supported. 1 register is the accumulator and that will be register 0. 1 register (reg 15) will hold only addresses, which will be used for jumping. 1 register (reg 14) will hold a value that determines how much we go back in the program counter. The rest of them are for general purpose.

Control Flow (branches)

The branches that are supported will be given in the lookup table of the machine. The target addresses are going to be stored in a memory location that the lookup table will give. To reach the target address we will find the difference between the current and target address then subtract that amount from the current address to reach the target address. The maximum branch distance supported will be 256 since the memory addresses can only store a byte.

Addressing Modes

Only indirect addressing is supported. To address a memory we need to first just to a memory address. Then we can either load the value in the memory address into a register or we can store the contents of a register into the memory address. We cannot directly apply operations to values in memory.

4. Programmer's Model [Lite]

My machine operates on an accumulator type and does not have direct access to memory so a programmer should prioritize loading in the necessary values from memory into as many registers as possible before performing calculations. The machine has a large amount of registers specifically so that more values can be stored in the registers from memory in order to perform calculations. Although if there are not enough registers it is still possible to store the accumulator value in another register and load more values from memory, it is a roundabout process so it is unrecommended.

No, we cannot copy instructions/operations from MIPS or ARM because we are using an accumulator ISA. Due to this, when you would usually input two input registers for an operation in MIPS or ARM, you would instead input 1 register since the other register is always going to be an accumulator for our machine.

4.3: Yes I will be using the ALU for non-arithmetic instructions such as comparison instructions which also involve a jump to a memory location. This complication will involve adding an extra mux into my design.

5. Individual Component Section

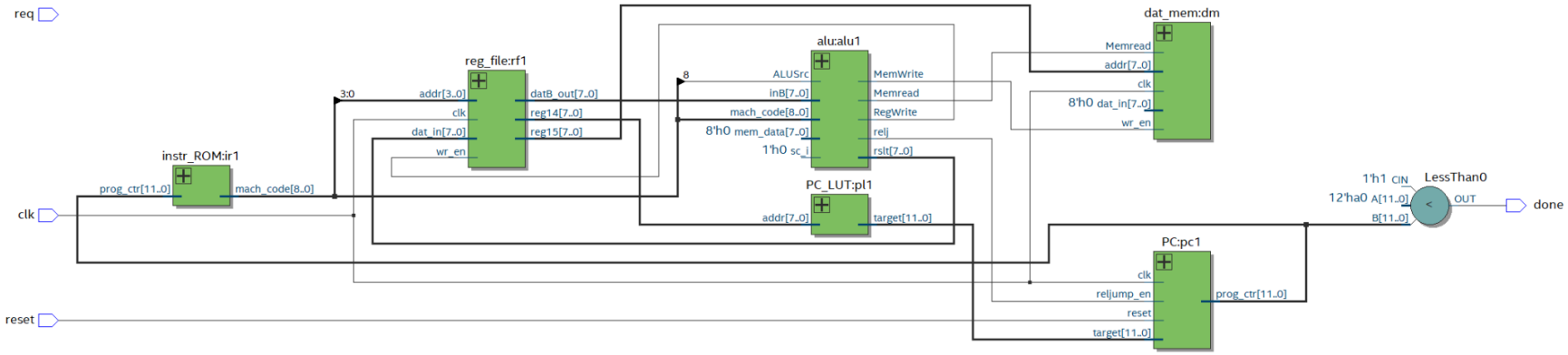
Top Level

Module file name: top_level.sv

Functionality Description

The top level module puts together all the other modules to make them work together.

Schematic



Resource Usage Summary

	Resource	Usage
1	Estimated ALUTs Used	167
1	-- Combinational ALUTs	167
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	140
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	167
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	71
3	-- 5 input functions	32
4	-- 4 input functions	45
5	-- <=3 input functions	19
8		
9	Combinational ALUTs by mode	
1	-- normal mode	147
2	-- extended LUT mode	0
3	-- arithmetic mode	20
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	255
12		
13	Total registers	140
1	-- Dedicated logic registers	140
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	4
17		
18	DSP block 18-bit elements	0
19		
20	Maximum fan-out node	clk~input
21	Maximum fan-out	140
22	Total fan-out	1251
23	Average fan-out	3.97

Program Counter

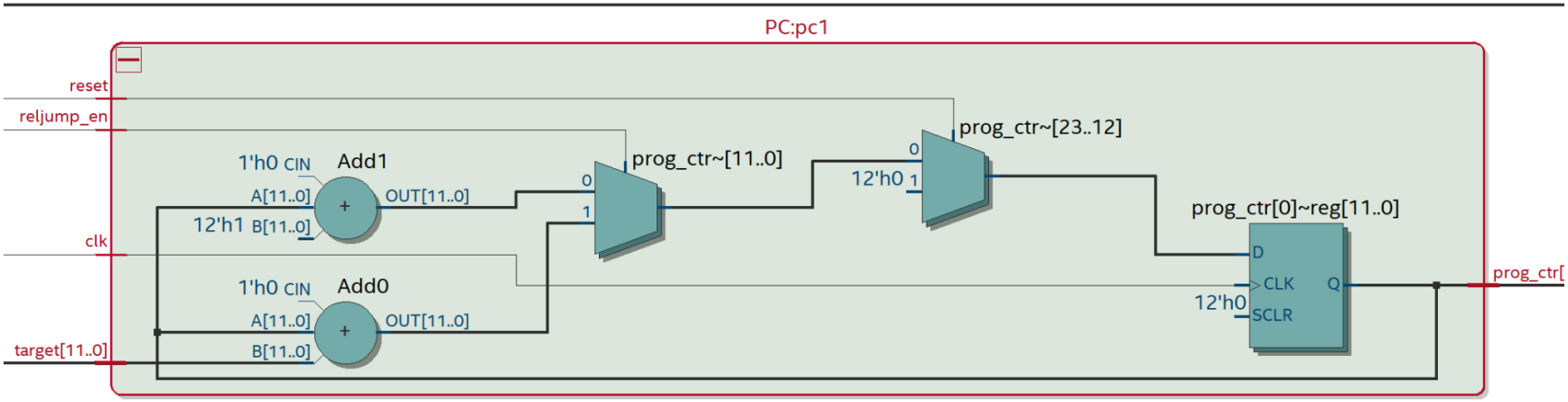
Module file name: PC.sv
Module testbench file name: TODO

Functionality Description

Keeps track of the program counter which is needed in the other modules. Also keeps track of jumps.

Schematic

TODO. Show us your schematic for the fetch unit.



(Optional) Timing Diagram

TODO. Show us a screenshot of the timing diagram that demonstrates all relevant functions of the fetch unit.

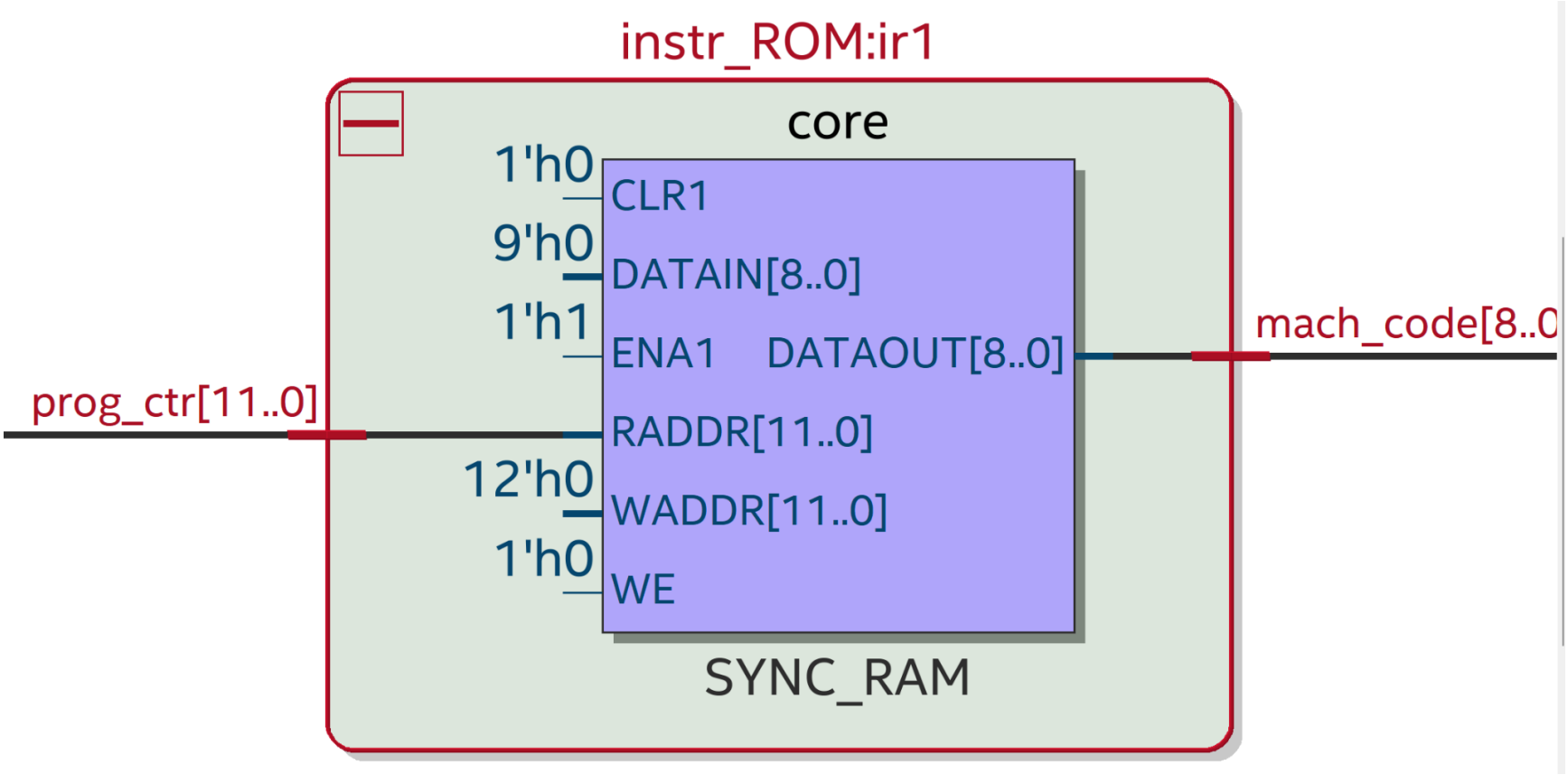
Instruction Memory

Module file name: Instr_ROM.sv

Functionality Description

Keeps track of all the bits for instructions so that each instruction does something different.

Schematic



This schematic is from the demo code, and will be updated next time when I have finished more of my own code.

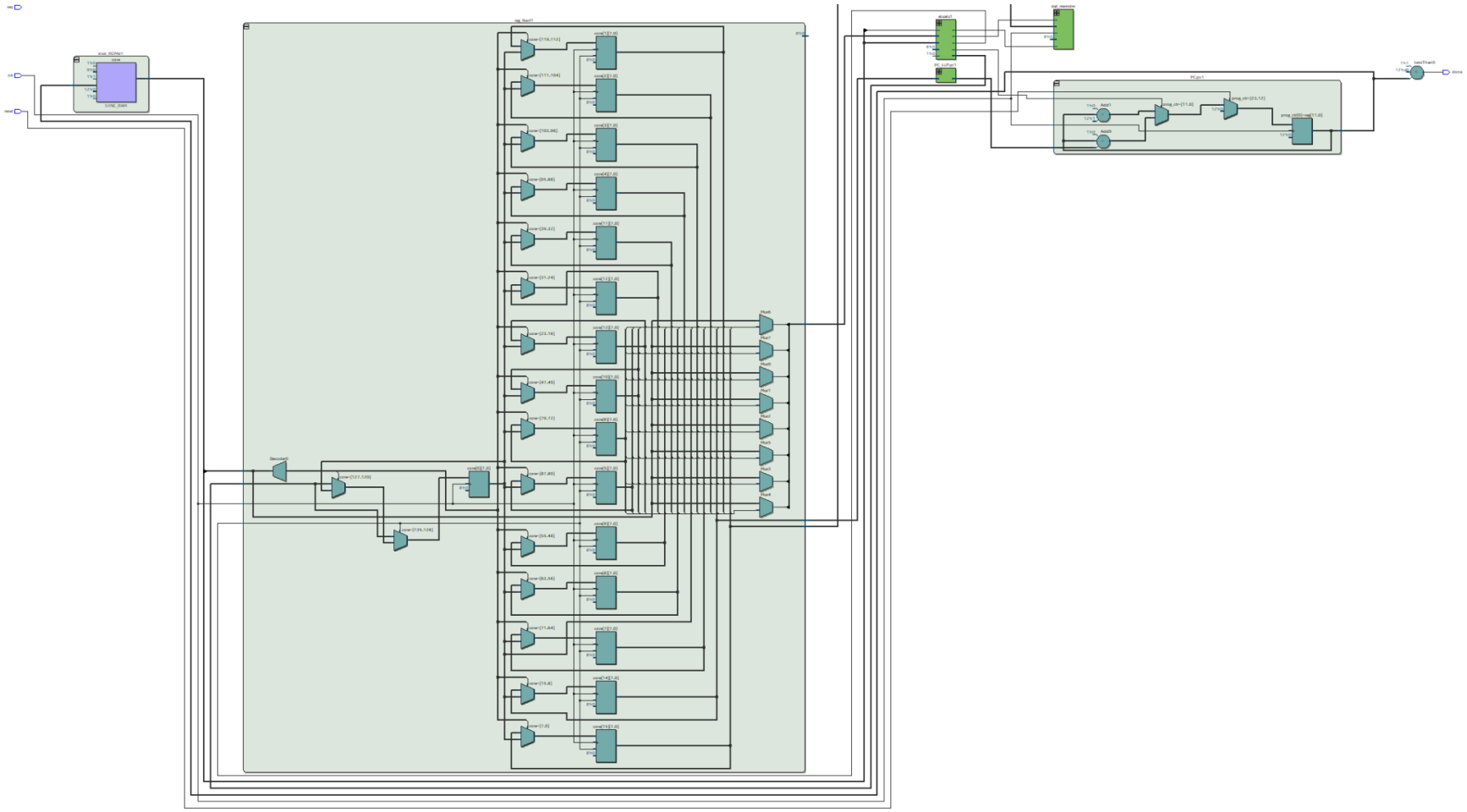
Register File

Module file name: reg_file.sv

Functionality Description

Keeps track of all the registers and their contents.

Schematic



ALU (Arithmetic Logic Unit)

Module file name: alu.sv
Module testbench file name: TODO

Functionality Description

This module will handle all of the arithmetic logic that is needed for my microprocessor. It will handle all the R type instructions that involve some sort of math.

(Optional) Testbench Description

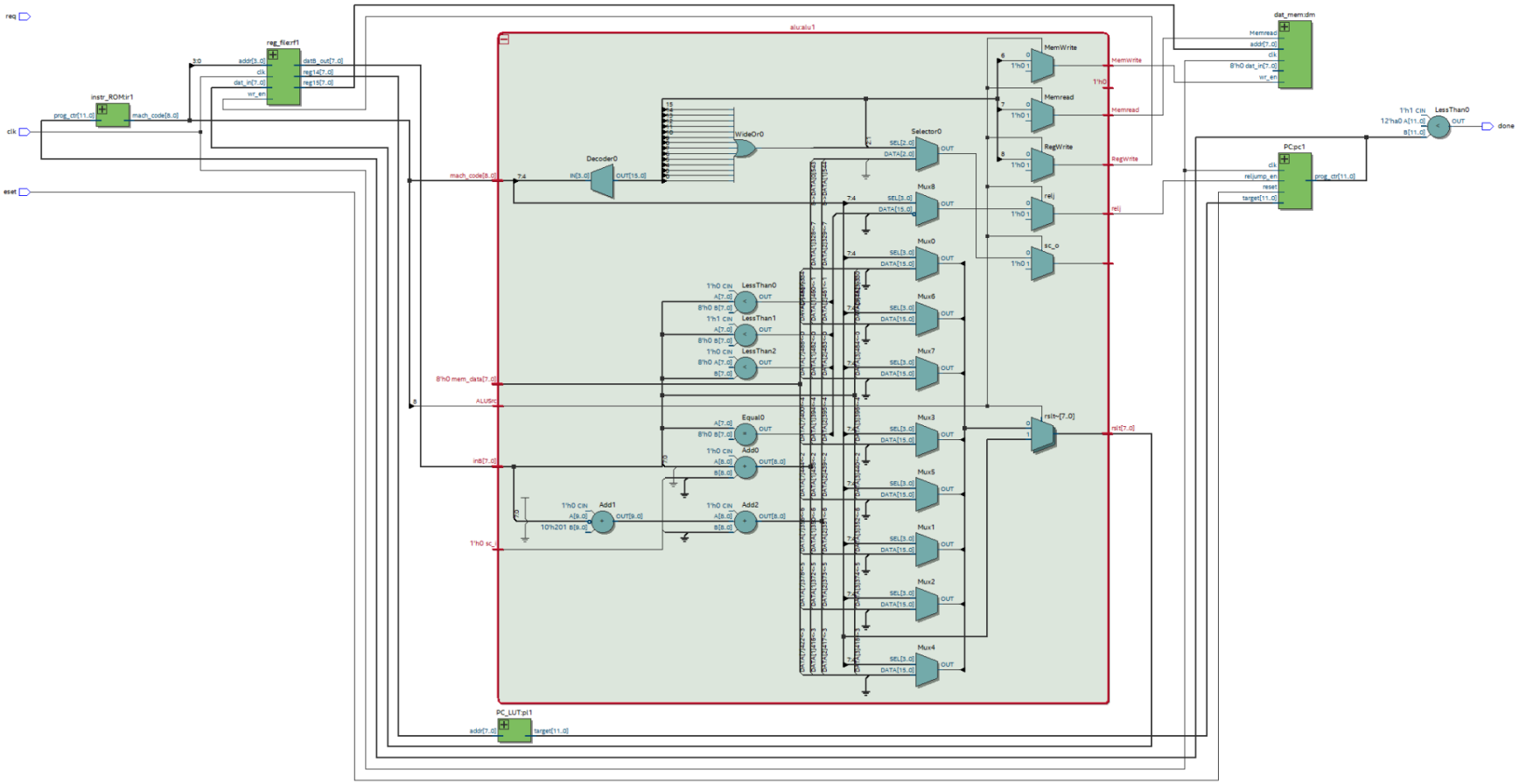
TODO. Describe your testbench. How does it work? What test cases does it test?

ALU Operations

TODO. What ALU operations will you be demonstrating? What instructions are they relevant to?
The ALU will demonstrate a comparison for greater than, greater than or equal to, equal to and not equal to. It will also have all the basic gates for the and, xor. It will support bit shifting for the shifting instructions. It will support adding and subtracting for the ADD and SUB instructions.

Schematic

TODO. Show us your schematic for the register file.



(Optional) Timing Diagram

TODO. Show us a screenshot of the timing diagram that demonstrates all relevant operations you mentioned in the ALU Operations section.

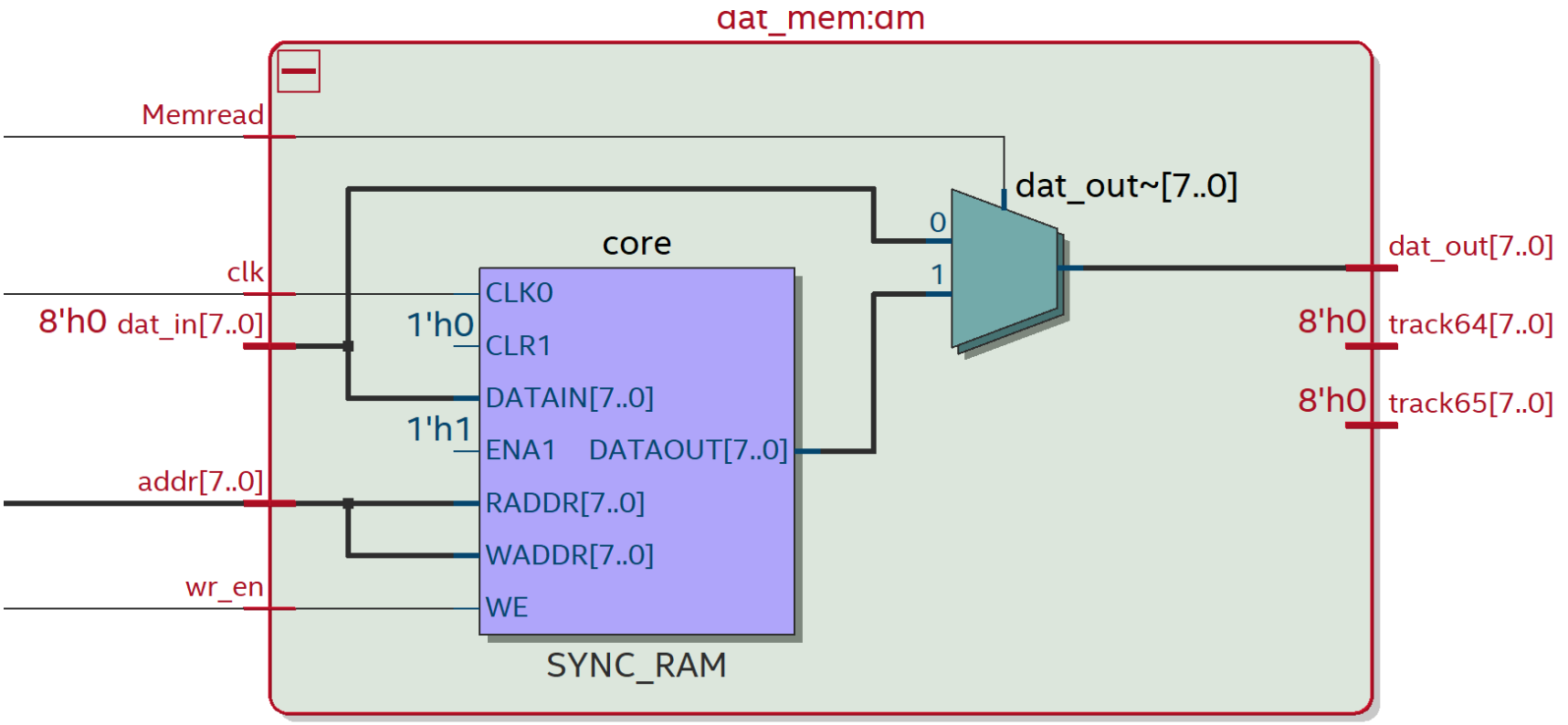
Data Memory

Module file name: dat_mem.sv

Functionality Description

This module will contain the memory slots for holding data. It will also begin by holding some data in some specific memory locations for later use.

Schematic



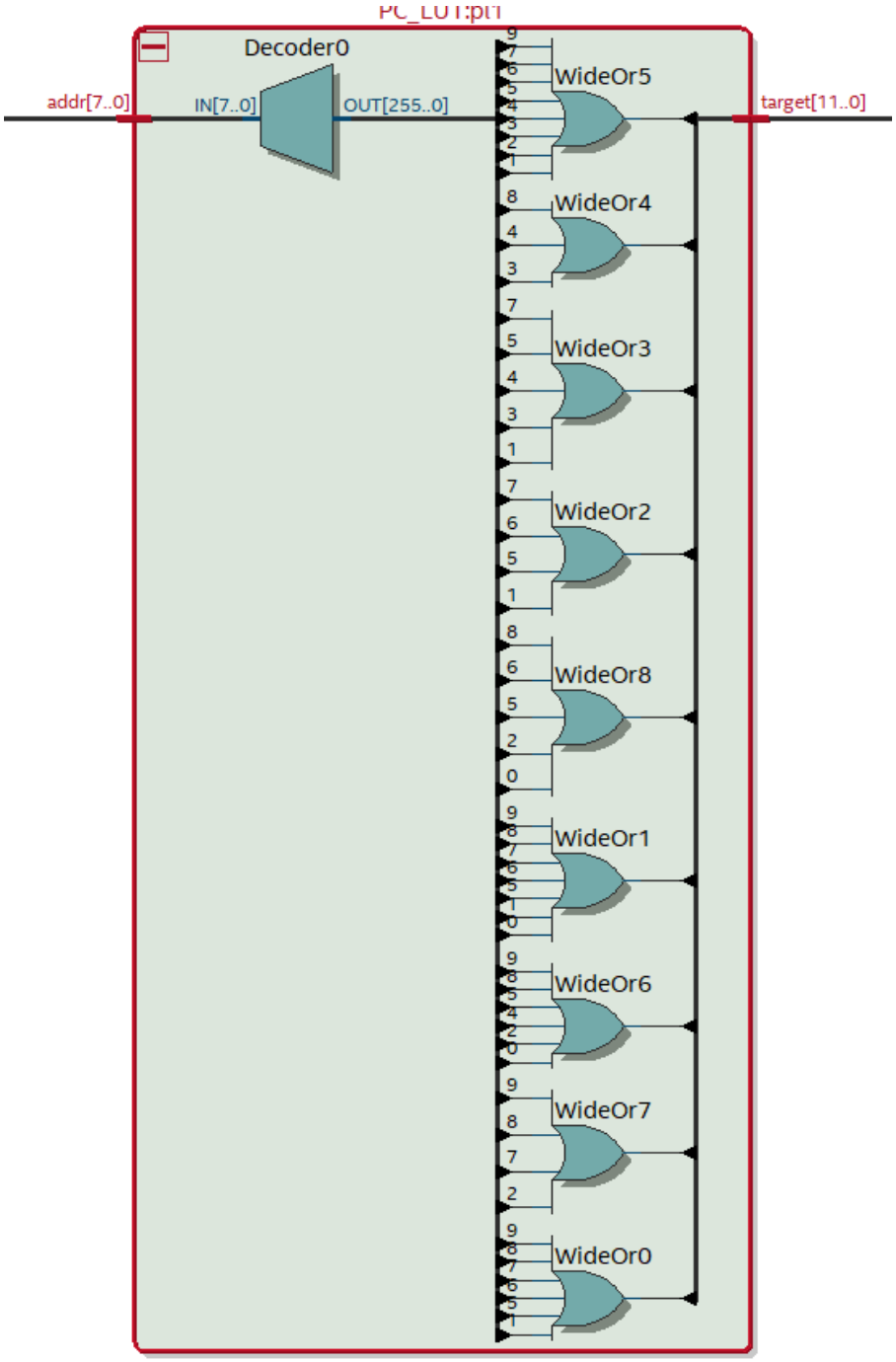
Lookup Tables

Module file name: PC_LUT.sv

Functionality Description

This module will contain some tables that are indexed by reg14 that have specific values for how much we want to jump when branching.

Schematic


PC_LUT:pt1
Decoder0
addr[7..0]
IN[7..0]
OUT[255..0]
target[11..0]
WideOr5
WideOr4
WideOr3
WideOr2
WideOr8
WideOr1
WideOr6
WideOr7
WideOr0

Other Modules (if necessary)

Module file name: TODO

Functionality Description

TODO. Write a brief description of the functionality of this module.

Schematic

TODO. Show us your schematic for your module.

6. Program Implementation

An example Pseudocode and Assembly Code has been filled out for you. When you submit, please delete the example along with this paragraph.

Example Pseudocode

```
# function that performs division
mul_inverse(operand):
  divisor = operand
  dividend = 1
  result = 0
  counter = 0
  while counter != 16:
    if dividend > divisor:
      dividend -= divisor
      result = (result << 1) || 1
    else:
      result = (result << 1)
    dividend <<= 1
    counter += 1
  return result
```

Example Assembly Code

```
# Do not try to understand this code. It is bogus code, but a good example of what to submit.

# loading divisor
load R0, %0010        # 0010 = location of the divisor in memory
load R1, %0100        # 0100 = location of the dividend in memory

add R0, R1, R2        # R0 + R1 => R2 adding the divisor and the dividend together
...
# more assembly code
...

# note that this may be several pages long. The teaching staff will not be verifying
correctness of your assembly code for Milestone 1.
```

Program 1 Pseudocode

Use designated register for closest hamming distance, and designated register for maximum hamming distance
Create a branch that will stop looping once all pairs have been checked:

- The first half-word will be stored in a designated and the second half-word will be stored in another designated register
- XOR the two half-words together and store it
- If the XOR value is less than the minimum, change the minimum
- If the XOR value is greater than the maximum, change the maximum

Write the minimum distance to location 64 and maximum distance to 65

Program 1 Assembly Code

Look at file assembly_program1

Program 2 Pseudocode

Use designated register for smallest arithmetic pair, and designated register for maximum arithmetic pair
Create a branch that will keep looping until all pairs are checked:

- Store the first integer the second integer
- Subtract 00 from 01
- If subtraction value is less than smallest, change the smallest
- If subtraction value is greater than maximum, change the maximum

Write the smallest arithmetic pair to 66 and maximum to 67

Program 2 Assembly Code

Look at file assembly_program2

Program 3 Pseudocode

Create a branch that will keep looping for all operands

- Load operands for first number into designated register
- Load operands for second number into designated register
- Create a branch that will keep looping for all bits
    - If current least significant bit of the second number is 1
        - Shift bits left by 1
        - Add operand A and B
        - Store into corresponding place in memory

Program 3 Assembly Code

Look at file assembly_program3

7. Changelog

- Milestone 3/Final Product
    - Team
        - Added team member, Jack Barkes, when originally both were working alone
    - Machine Specification
        - Many instructions were changed in order to fit the format of an accumulator
        - Also adjusted how to get and load from memory
        - Branch instructions were updated to always take from R15
        - Barrel shifting was added
        - In order to index lookup table, register R14 became the designated register for this
    - Individual Component Section
        - All the Schematics in this section have been updated to show our working schematics
        - We have removed the Decode control component and instead made the ALU file handle all of the opcode. (Even code that does not require computation)
    - Program Implementation
        - Added pseudo code that worked as a basis for the assembly
        - Changed the assembly code to state which file the assembly code is in.
- Milestone 2
    - Team
        - Specified that I am working alone
    - Introduction
        - I gave my accumulator a name
    - Architectural Overview
        - I changed my overview from a google block design to a Quartus block design diagram
    - Machine Specification
        - I removed some branch instructions (less than and less than or equal to since this can be done with greater than and greater than or equal to and swapping the order)
        - I also removed jump since
    - Individual Component Section
        - All new material
- Milestone 1

- Initial version