Lab 2

Melissa Ly, Sawandeep Kaur, Alan Chen

A11908353, A11601962, A12045305

### 

### 

[Dropbox link of schematic](https://www.dropbox.com/s/9f58t8moqnvow9f/cse141lschematic.png?dl=0)

## Introduction

In this lab, we simulated the individual computer parts that make up our hardware design. At the moment, they work only independently, but will later be connected via a series of signals as shown in the schematic above. We will begin by reviewing the operations our hardware must support. Then, we will describe each relevant component in isolation: the register file, the fetch unit, and the ALU.

### Operations Supported

**General Operations**

* set memory ptr [imm]: 0000
  + The immediate hashes to another number, so only key memory locations can be accessed.
  + This operation sets the memory pointer (r0) to the hashed number, then pushes the contents of that memory location into the stack.

|  |  |  |  |
| --- | --- | --- | --- |
| Immediate | Hashed location | Immediate | Hashed location |
| 00000 | 1 | 00111 | 19 |
| 00001 | 2 | 01000 | 20 |
| 00010 | 3 | 01001 | 32 |
| 00011 | 4 | 01010 | 64 |
| 00100 | 5 | 01011 | 127 |
| 00101 | 6 | 01100 | 128 |
| 00110 | 7 | 01101 | 255 |

* store [reg]: 0001
  + stores the value in the register to where the memory pointer (r0) is currently pointing
* push [reg]: 0010
  + pushes the register contents into the stack
* add [reg]: 0011
  + adds the two top items in the stack
  + stores the result in the register
  + pops the addends
  + sets the overflow flag
* set [reg]: 0100
  + stores the topmost value on the stack inside the register
* push immediate [imm]: 0101
  + pushes the immediate onto the stack
* pop [imm]: 0110
  + pop the immediate number of items from the stack
* blt [label]: 0111
  + checks if the top value on the stack is less than the penultimate value on the stack
    - If true, branch to label
    - Else, continue
* inc [reg]: 1000
  + increments the value in the register by 1
  + If the register is 00000, also push the contents of the register to the stack

**Product Operations**

* &and shift [reg]: 1001
  + The value in the register is an index, i
  + Performs a bitwise 'and' with the i'th bit (starting from the LSB) from the penultimate stack item and the top stack item
  + Logical shifts the result 8-i bits to the right and pushes the result to the stack
  + Shifts the result of the bitwise 'and' left by i and pushes that to the stack
* add overflow [reg]: 1010
  + Adds the bit from the overflow to the register
  + Adds the top value in the stack to the register
  + Pops the top value

**String Match Operations**

* contains [imm]: 1011
  + Check if the top value on the stack contains the penultimate value on the stack
    - If so, branch to target address

**Closest Pair Operations**

* sub [reg]: 1100
  + Subtracts the two top items on the stack
  + Stores the result in the register
* abs[reg]: 1101
  + Gets the absolute value of the value in the register
  + Stores the result in the register
  + Pushes the value to the stack

**Other Operations**

halt: 1110

to be determined: 1111

### Instruction Fetch:



**Timing diagram:**

Program Counter:













Instruction ROM:





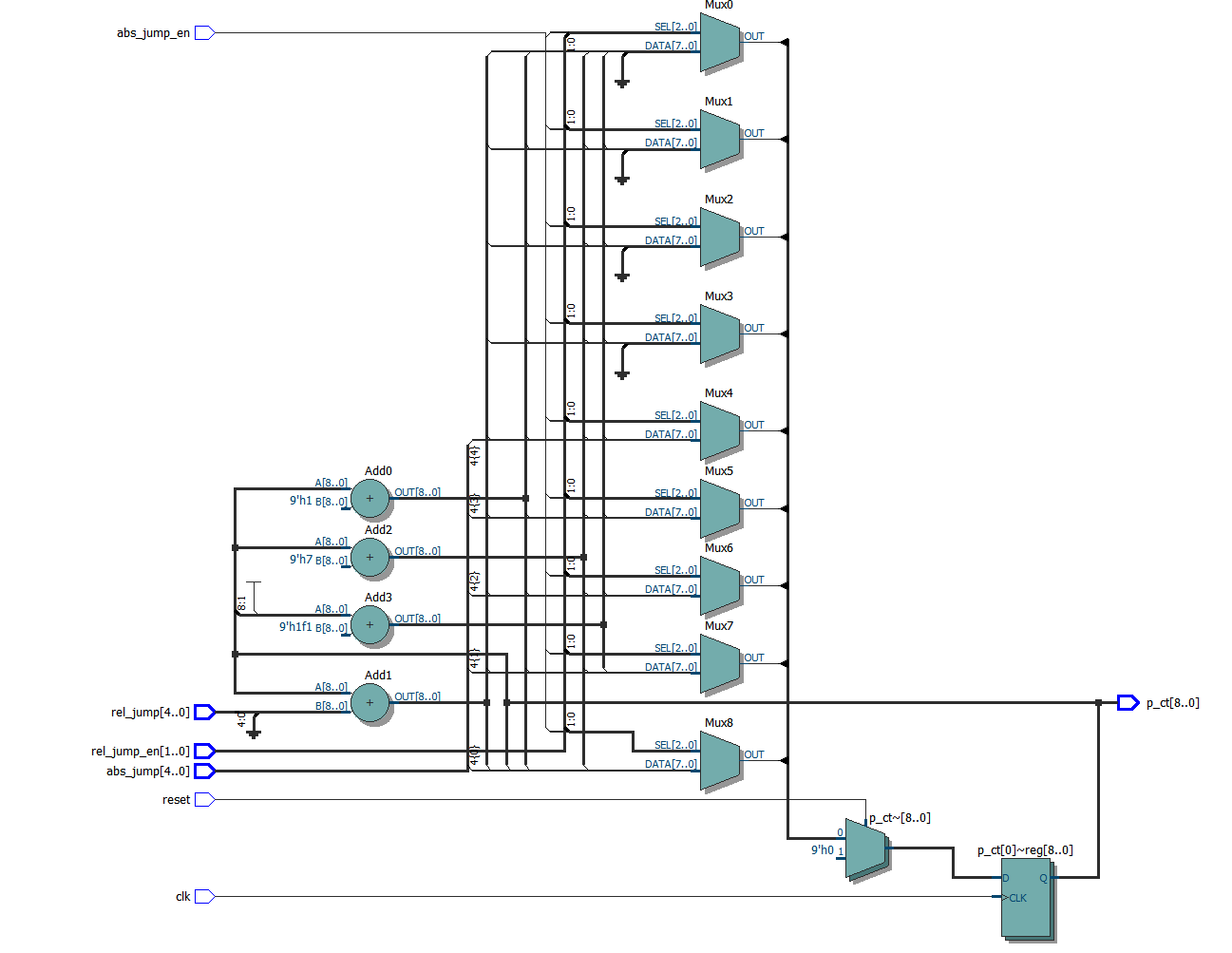
Instruction File:

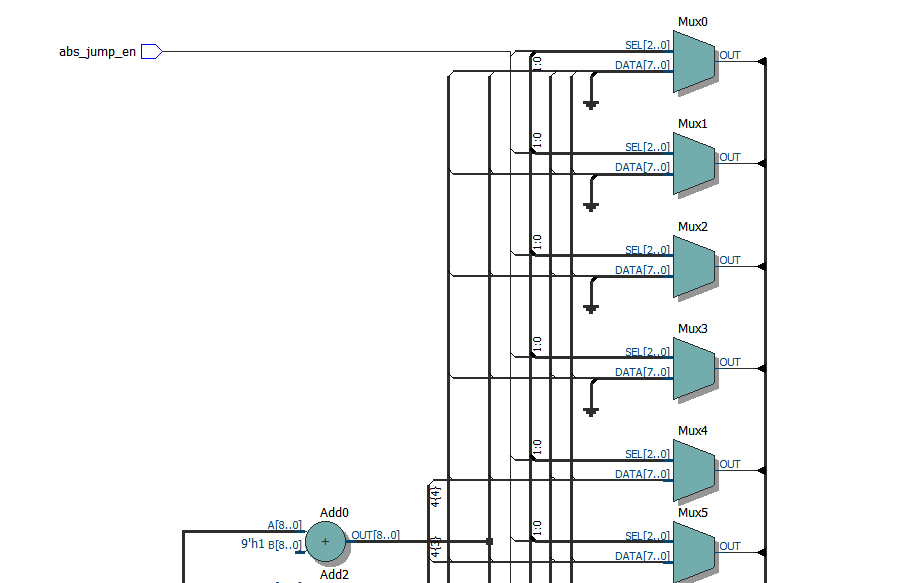




**Schematic:**

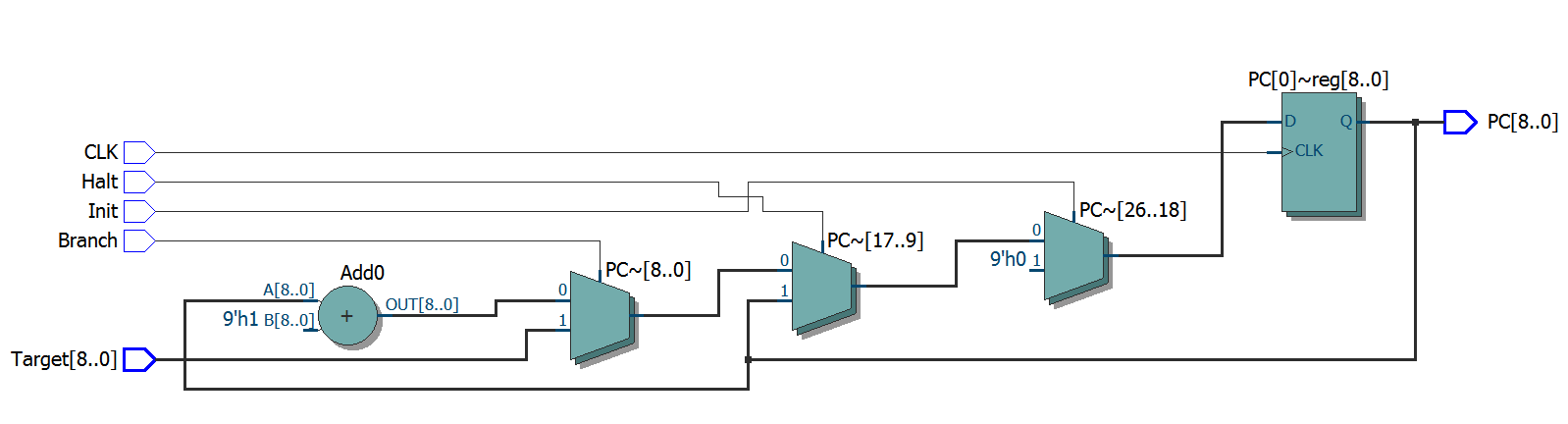
Program counter:



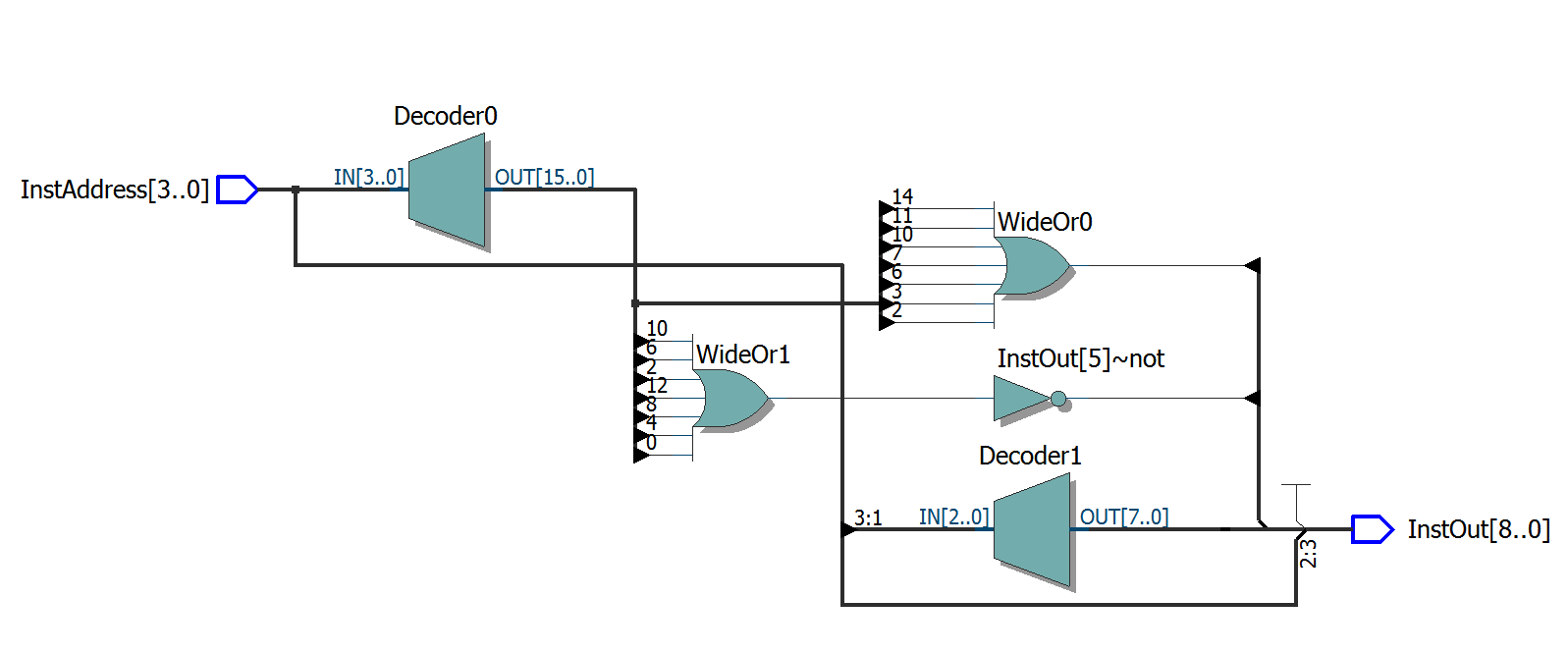




Instruction File:



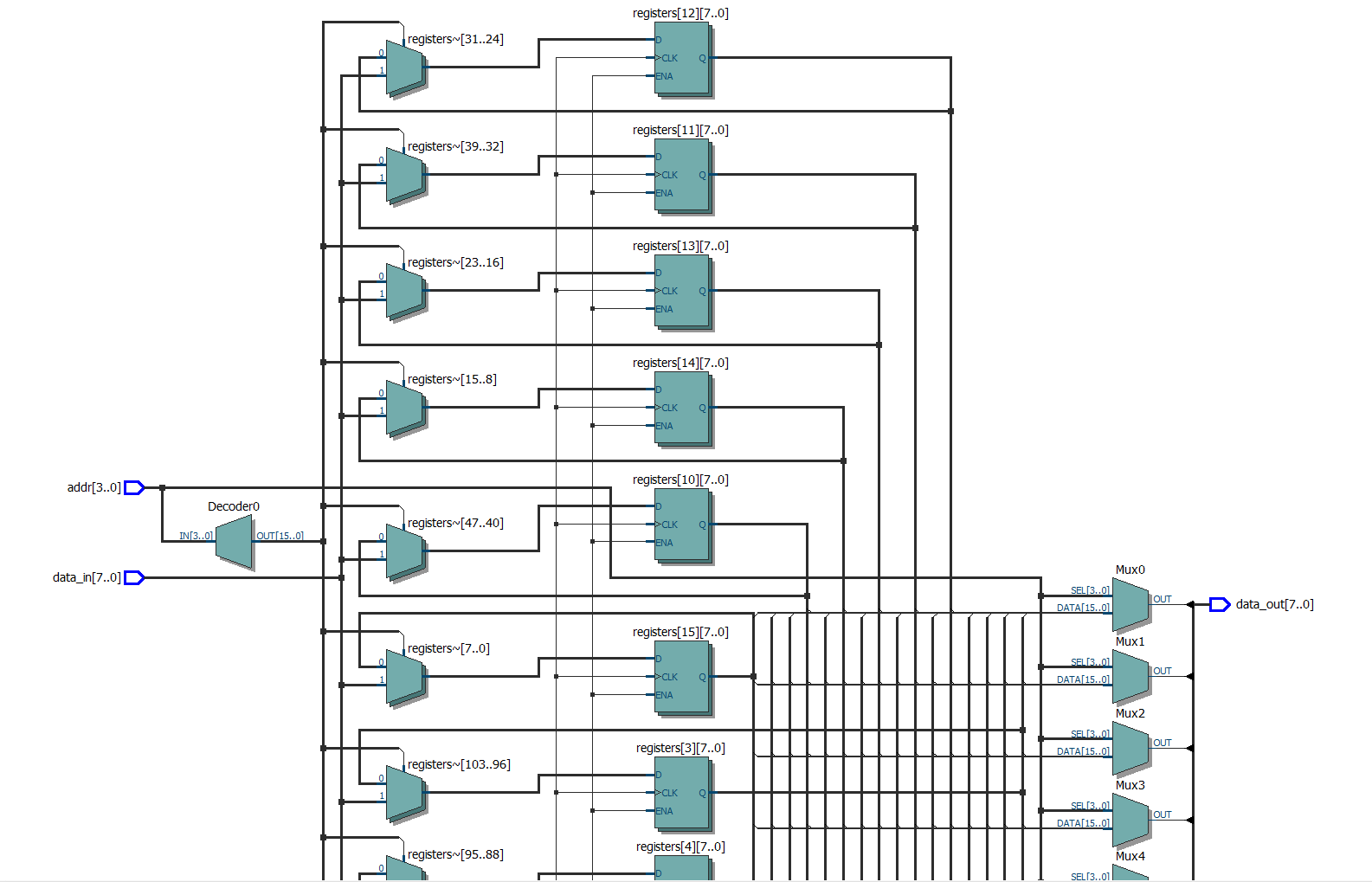
Instruction ROM:

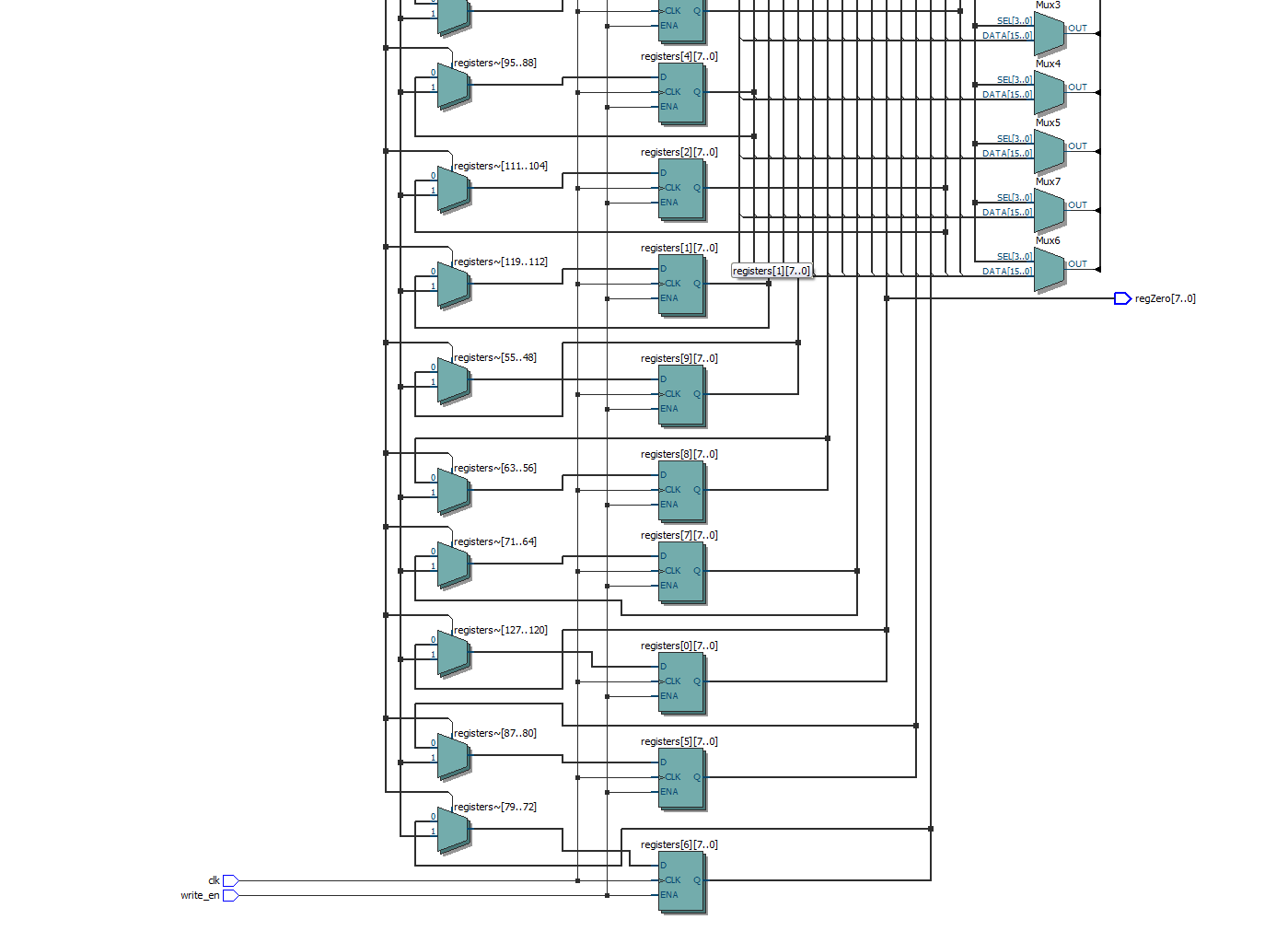


Our fetch design wasn’t really affected negatively, as it ended up not too different from a standard fetch unit design. The high number of operations led to longer case statements in the Instruction ROM, but this will make writing the assembly code easier later on.

### Register File

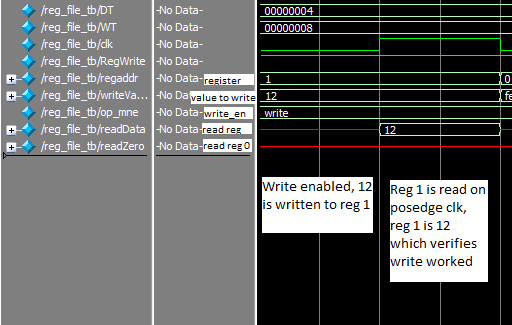




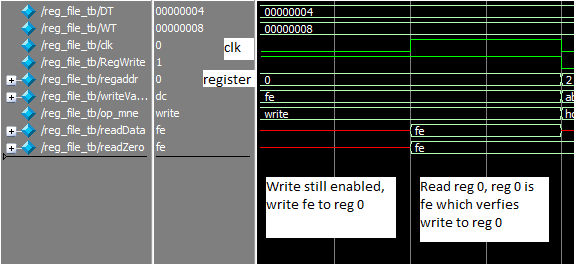


**Timing Diagram:**

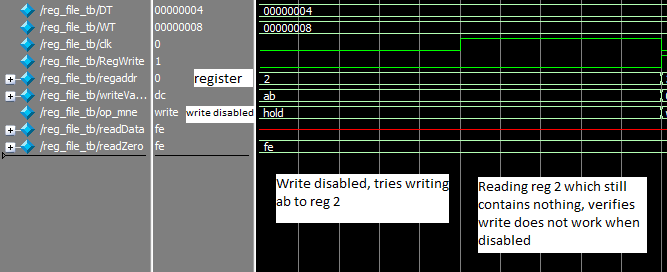
Test: writing 12 to reg 1



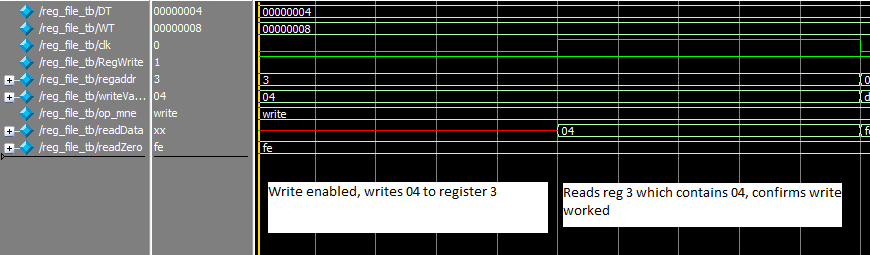
Test: writing fe to reg 0



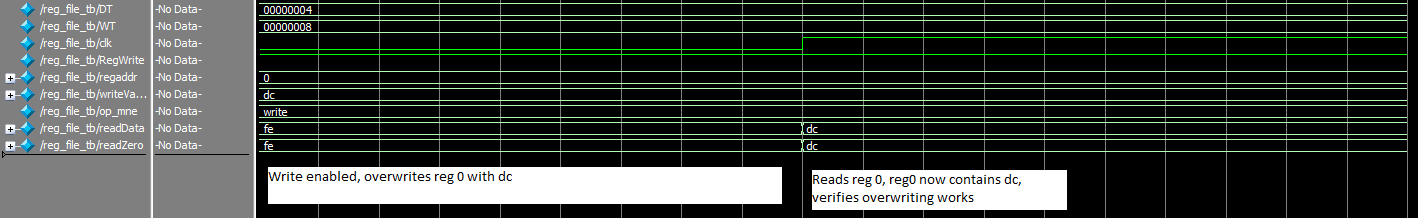
Test: writing to reg 2 while RegWrite is disabled



Test: writing 04 to reg 3



Test: overwriting a register



Our register file design is simple since our instructions only use one register that can be read or written to. So our register file only needs the register’s address, a control signal to determine whether the register is being written to, and the value to be written as input. It only outputs the values of the inputted register and register zero, thus this design is already simple enough to not have to change anything to make it easier to design.

#### ALU: Melissa

#### 

This ALU performs eight computations; many can be done in parallel, but some computations are many sequential steps strung together. They're all customized to perform as many operations as possible with the least number of instructions, so each output is mapped 1:1 to an instruction. To speed up the unit, they're all performed before the opcode has generated a signal, then the correct output is selected at the end. Here are the blocks performed, their descriptions, and their alu\_op codes.

* add: 000
  + adds stack0 and stack1
  + sets the overflow flag
  + writes the value into reg\_out
* branch less than (blt): 001
  + compares stack1 and stack0
  + sets the branch signal high if stack0 < stack1
* increment (inc): 010
  + increments the value of reg inputted by 1
  + writes the result in reg\_out
* & and shift (aas): 011
  + shifts stack1 right by the value stored in reg
  + &s the result with the value in stack0
  + stores the result in a temporary register
  + shifts the value in the temporary register right by 8-(reg\_val)
    - stores the result in stack1
  + shifts the value in the temporary register left by reg\_val
    - stores the result in stack0
* add overflow (aov): 100
  + adds the overflow and the value in stack0 to the value in reg
  + writes the result in reg\_out
* contains (con): 101
  + takes every possible four bits where stack0 may be a substring
    - performs an xor with stack0 to check for matching bits
    - performs a not so if there are no matching bits, a 1 is outputted
    - puts the result of all possibilities through an or gate
  + sets the branch signal high if any of possibilities outputs a match
* subtract (sub): 110
  + subtracts stack1 from stack0
  + writes the result to reg\_out
* absolute value (abs): 111
  + takes the absolute value of the value stored in the input register
  + writes the result back into the register
  + stores the result into stack0 as well

The block names indicate the instruction they perform for. The ALU does not perform branch instruction calculations because there are few enough locations to branch to that they can be coded directly into the instruction, as detailed in our first lab.

Of course, the ALU was made much more complicated by the fact that it performed convoluted operations in order to reduce our instruction count in our ISA; an ISA that used simpler operations (add, subtract, bitwise access) was certainly possible and would make our ALU much smaller. However, as one of the goals of the first lab was to reduce instruction count, we decided against this.

Given that the operations our ALU performs are so highly specialized, there aren't very many useful instructions that could be added. We could push more values at a time, given that currently only one instruction pushes two values and most don't make use of this extra output. We could also very easily add instructions that output in different locations.

There are two instructions that compete for the highest complexity in this ALU: "and and shift" and "contains." They are the most specialized and make our code less repetitive, but as a result, require many operations within our ALU. "Contains" requires five extra registers, and "and and shift" requires six black boxes of work.

