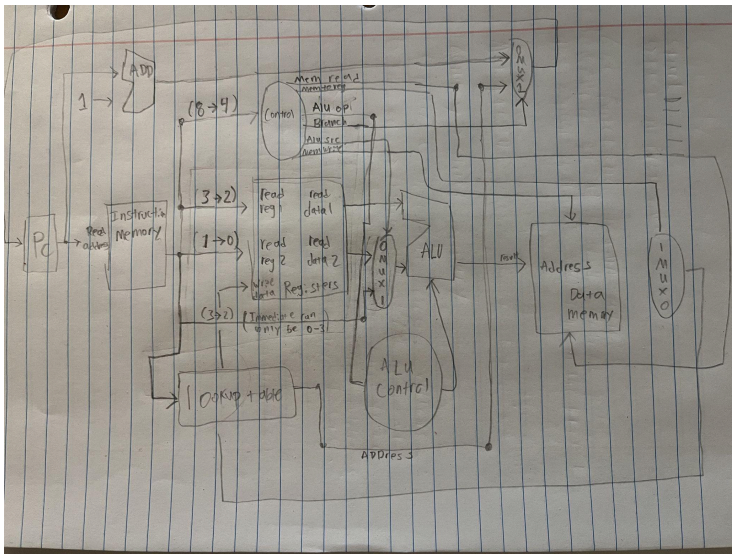
0. Team

Seth Chng

1. Introduction

My machine can be defined as a type of register-register/load store machine. I plan on having 32 different instructions and 4 registers to run my operations. Due to this, I will name my architecture memory pull, since I will pull from memory a lot of times with the limited amount of registers I will use. The philosophy behind my machine is to have a lot of instructions so most of my work is about handling my registers effectively. The larger amount of instructions to register should make the code longer, but I should have a variety of tools/instructions available to create the program required.

2. Architectural Overview



3. Machine Specification

Instruction format:

I have two types of instruction. R type instructions and J type instructions. This will take 1 bit. J will make the first bit in my instruction bits 1, while R instructions will not.

R type:

<u>Arithmetic (2)</u>

Add
Sub

Logical (4)
And
Or
Xor
Shift left logical

Data transfer (3)
Load address
Store address
move

Branch (1)
Branch on not equal

J type:

Jumps (1)
Jump

13 instructions total

5 bits required for instructions, 4 bits left.

Type	Format	Instructions
R	1 bit type 4 bits opcode, 4 bits free for registers or sys calls	Add Sub And Or Xor Shift left logical Load address Store address Move Branch on equal Branch on greater Branch on lesser Read int Exit return
J	1 bit type, 8 bit address	Jump

Operations:

Name	Type	Bit breakdown	Example	Notes
Add	R	1 bit type followed by 4 bit opcode 00000 followed by 2 2 bit registers	# Assume R0 has `0b0001_0001` # Assume R1 has `0b0000_0000` Add R0, R1 # after and instruction, R0 now holds `0b0001_0001`	R0 will be the destination register and update itself accordingly
Sub	R	1 bit type followed by 4 bit opcode 00001 followed by 2 2 bit	# Assume R0 has `0b0001_0001` # Assume R1	R0 will be the destination register and update itself accordingly

		registers	has `0b0000_0001` sub R0, R1 # after and instruction, R0 now holds `0b0001_0000`	
And	R	1 bit type followed by 4 bit opcode 00010 followed by 2 2 bit registers	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` And R0, R1 # after and instruction, R0 now holds `0b0001_0000`	R0 will be the destination register and update itself accordingly
or	R	1 bit type followed by 4 bit opcode 00011 followed by 2 2 bit registers	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` Or R0, R1 # after and instruction, R0 now holds `0b1001_0001`	R0 will be the destination register and update itself accordingly
xor	R	1 bit type followed by 4 bit opcode	# Assume R0 has `0b0001_0001`	R0 will be the destination register and update itself

		00100 followed by 2 2 bit registers	# Assume R1 has `0b1001_0000` Xor R0, R1 # after and instruction, R0 now holds `0b1000_0001`	accordingly
Shift left logical	R	1 bit type followed by 4 bit opcode 00100 followed by 2 2 bit registers	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` Sll r0, r1 # after and instruction, R0 now holds `0b0000_1000`	R0 will be the destination register and update itself accordingly
Branch on equal	R	1 bit type followed by 4 bit opcode 00111 followed by 2 2 bit registers	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` Beq r0,r1 # after and instruction, Reg hold the same value, but lookup table will be called	This just compares, so nothing too specific here besides it will check the lookup table to know where to branch to
Branch on Not equal	R	1 bit type followed by 4 bit opcode	# Assume R0 has	This just compares, so nothing too

		01000 followed by 2 2 bit registers	0b0001_0001 # Assume R1 has 0b1001_0000 Bne r0, r1 # after and instruction, Reg hold the same value, but lookup table will be called	specific here besides it will check the lookup table to know where to branch to
Load Address	R	1 bit type followed by 4 bit opcode 01010 followed by 1 2 bit register and null/dont care bits	# Assume R0 has 0b0001_0001 Lod r0 # after and instruction, R0 will hold the address from the destination in memory	Implied destination in memory inaccessible to the programmer to change to load and store memory addresses
Store Address	R	1 bit type followed by 4 bit opcode 01011 followed by 1 2 bit register and null/dont care bits	# Assume R0 has 0b0001_0001 Str r0 # after and instruction, R0 will hold the address from the destination in memory	Implied destination in memory inaccessible to the programmer to change to load and store memory addresses
Move	R	1 bit type followed by 4 bit opcode 01100 followed	# Assume R0 has 0b0001_0001	R0 will be the destination register and update itself accordingly

		by 2 2 bit registers	# Assume R1 has `0b1001_0000` Mov r0, r1 # after and instruction, R0 now holds `0b1001_0000`	

Internal Operands:
4 registers are supported by my design. All 4 of these registers are general purpose. Each register will be specified by two bits, allowing for the load load structure of my machine. There is also a dedicated memory location to store and load memory addresses from that the load and store methods will call.

Control Flow:
My branches will be supported by a lookup table. The table will store the memory address to branch to and it will let the program counter know if it must be incremented by one or just branch to the address in the lookup table. Once the instruction is into the lookup table it will also be rewritten to allow the bits required to store the address. The branches that are supported are the basic operations Greater than, less than, and equal to.

Addressing Modes
The addressing mode of my machine will be direct. I will use my lookup table, load store, and jump functions to do so. Load and store will be directly addressed by always checking the saved memory used to store memory addresses, while jump will be directly calling which address to go/jump to.

4. Programmer's Model

For my machine, a programmer should think about loading and writing to memory in between every calculation step. By only having 4 registers, a programmer is severely limited in how much can be stored in the registers. Due to this a huge priority of my system is to load and write between each step in memory because of the limited register space. On top of this, a programmer must keep in mind that loading and storing memory addresses is dangerous and requires a lot of thought due to the fact that it is actually accessing a memory address that points to other memory.

I have used a lot of the similar instructions from MIPS to create the instructions for my machine. The instructions are the same, but the encoding will be different. This is due to the fact that my machine has an instruction length of 9 bits, while MIPS has 32. I overcame this by looking at the requirements for the three programs we were told to code, and decided on 16 instructions that I will take from MIPS to my machine.

```
module HammingDistanceCalculator(
  input wire [N-1:0] input1,
  input wire [N-1:0] input2,
  output reg [N-1:0] hamming_distance
);

  reg [N-1:0] counter;
  reg [N-1:0] temp1, temp2;

  initial
        counter = 0;

  always @(*) begin
        temp1 = input1;
        temp2 = input2;
        hamming_distance = 0;
        for (int i = 0; i < N; i = i + 1) begin
        if (temp1[i] != temp2[i]) begin
        counter = counter + 1;
        end
        end
        hamming_distance = counter;
  end
endmodule


module ArithmeticDistanceCalculator(
  input wire [8:0] input1,
  input wire [8:0] input2,
  output reg [8:0] arithmetic_distance
);

  reg [8:0] result;

  always @(*) begin
        result = input1 - input2;
        arithmetic_distance = result;
  end
```

```
endmodule

module TwoComplementMultiplier(
  input wire [15:0] multiplicand,
  input wire [15:0] multiplier,
  output reg [31:0] product
);

  reg [31:0] accumulator;
  reg [31:0] temp;
  reg [15:0] multiplier_reg;
  reg [3:0] shift_count;

  initial begin
    accumulator = 0;
    multiplier_reg = multiplier;
    shift_count = 16;
  end

  always @(*) begin
    temp = {16'b0, multiplicand}; // Pad multiplicand with zeros
    if (multiplier_reg[15] == 1'b1) begin
      accumulator = accumulator + temp;
    end
    multiplier_reg = multiplier_reg << 1; // Shift multiplier left by 1
    shift_count = shift_count - 1;
    if (shift_count == 0) begin
      product = accumulator;
    end
  end
endmodule
```

Question 4.3:

Yes I will be using my ALU for for non-arithmetic instructions. This complicates my design by requiring more instructions to be coded into my ALU. I will be branching off of the ALU and this will cause 3 more things to worry about. Additional Control Logic, Conditional Execution, and Program Counter Manipulation. I plan on creating another lookup table for my branching to help stave off these issues.

CSE 141L Milestone 2 Add-on Template

Instructions

We are providing you with the template that should be copied and pasted into your Milestone 1 Report. The updated report, alongside your Verilog code, will be your submission for Milestone 2.

Steps

1. Change the title of your report from "CSE 141L Milestone 1" to "CSE 141L Milestone 2".
2. Please make appropriate changes to your Milestone 1 content based on feedback from the teaching staff. Make sure to update your Changelog with what you have changed.
3. Add your answer to the question "Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM-like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?" to your Programmer Model's section (Section 4). Label this question 4.3.
4. Change the Program Implementation number from 5 to 6.

(more steps on the next page)

5. Copy, paste, and complete the section templates we provide in the following pages to your report. Omit sections that you don't intend on using from your report (e.g. lookup table if you do not have one). We want you to test two modules: the Program Counter (to see if you can perform jumps/branches) and the ALU (to verify you have implementations for your specified instructions). Your report should consist of, *in order*:
    0. Team
    1. Introduction
    2. Architectural Overview
    3. Machine Specification
    4. Programmer's Model
    5. Individual Component Specification
        a. Top Level
        b. Program Counter
        c. Instruction Memory
        d. Control Decoder
        e. Register File
        f. ALU
        g. Data Memory
        h. Lookup Tables
        i. Muxes
        j. (if necessary) other modules - *name sections as appropriate*
    6. Program Implementation
    7. Changelog

(this page intentionally left blank to avoid confusion)

Individual Component Specification

***Disclaimer: These schematics are the starter code given, I am currently still working through the logic for some of my own edits and wanted to turn in a quartus diagram schematic that compiled.**

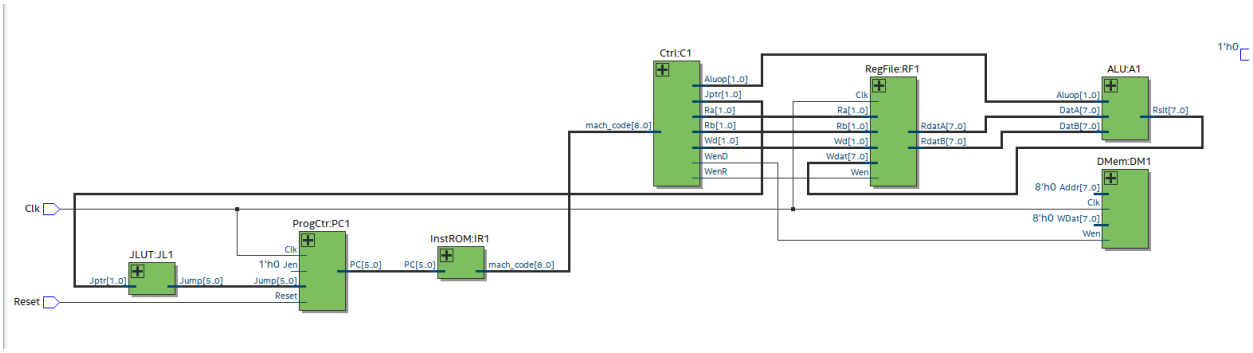
Top Level

Top.sv

Functionality Description

This module is responsible for running all of the programs in conjunction with each other.

Schematic



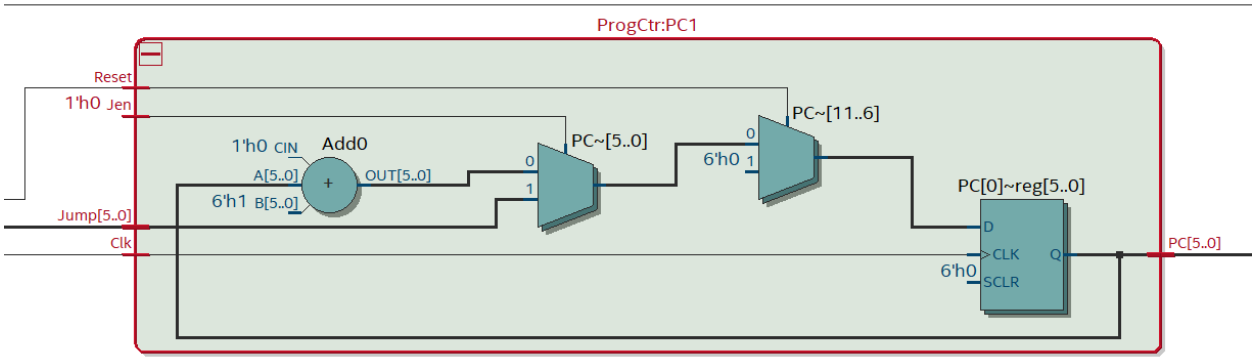
Program Counter

Module file name: ProgCtr.sv
Module testbench file name: ProgCtrtb.sv

Functionality Description

This module is responsible for incrementing the program counter and make sure every instruction is being run and when jumping instructions get incremented accordingly

Schematic



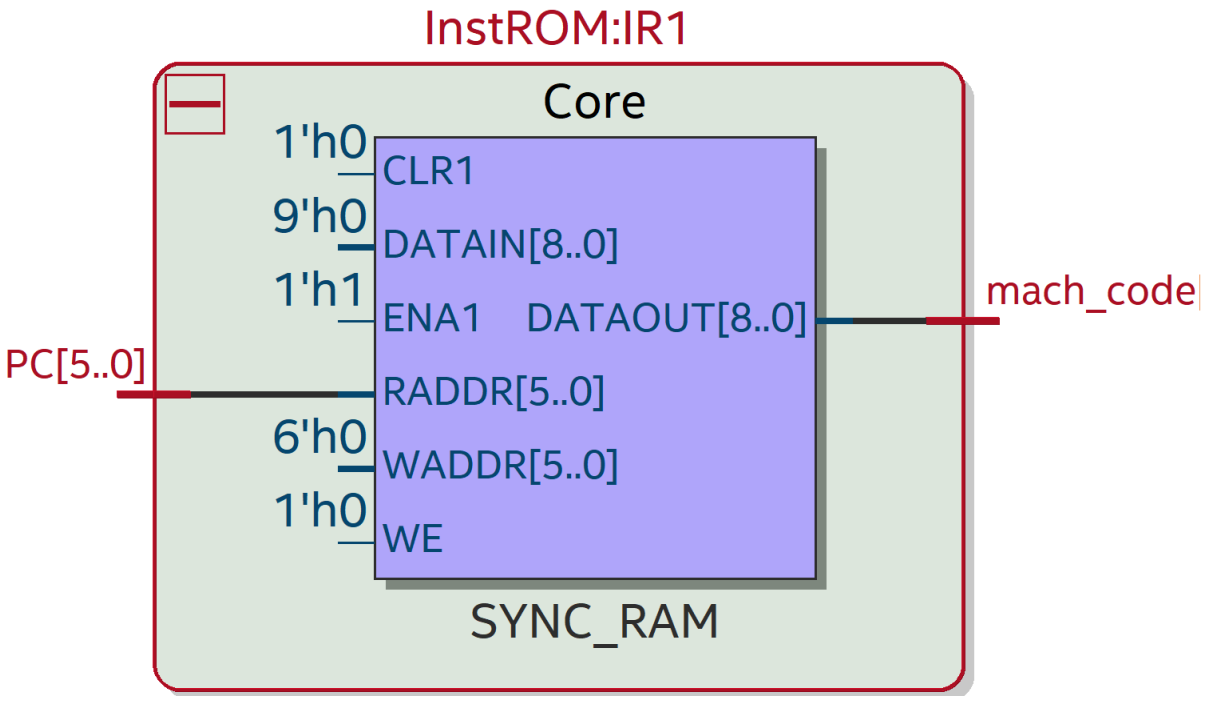
Instruction Memory

Module file name: InstROM.sv

Functionality Description

This module will be responsible for storing and defining all the possible instructions my computer will be running.

Schematic



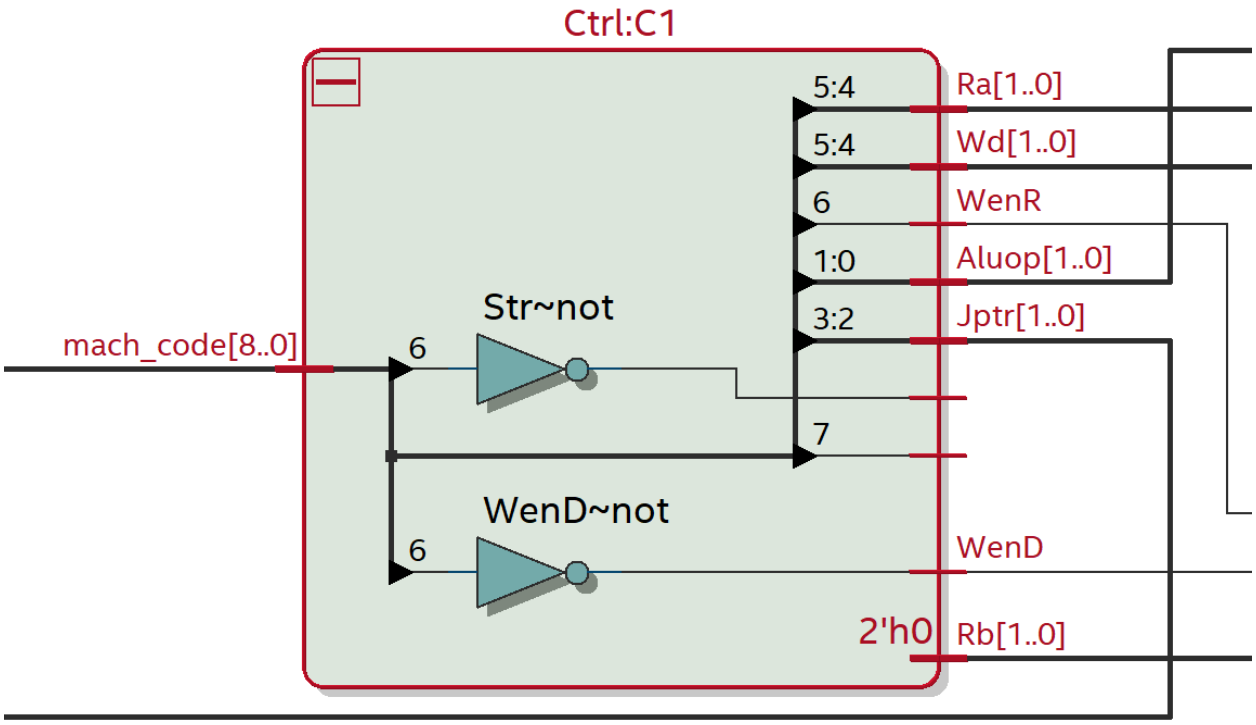
Control Decoder

Module file name: Ctrl.sv

Functionality Description

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

Schematic



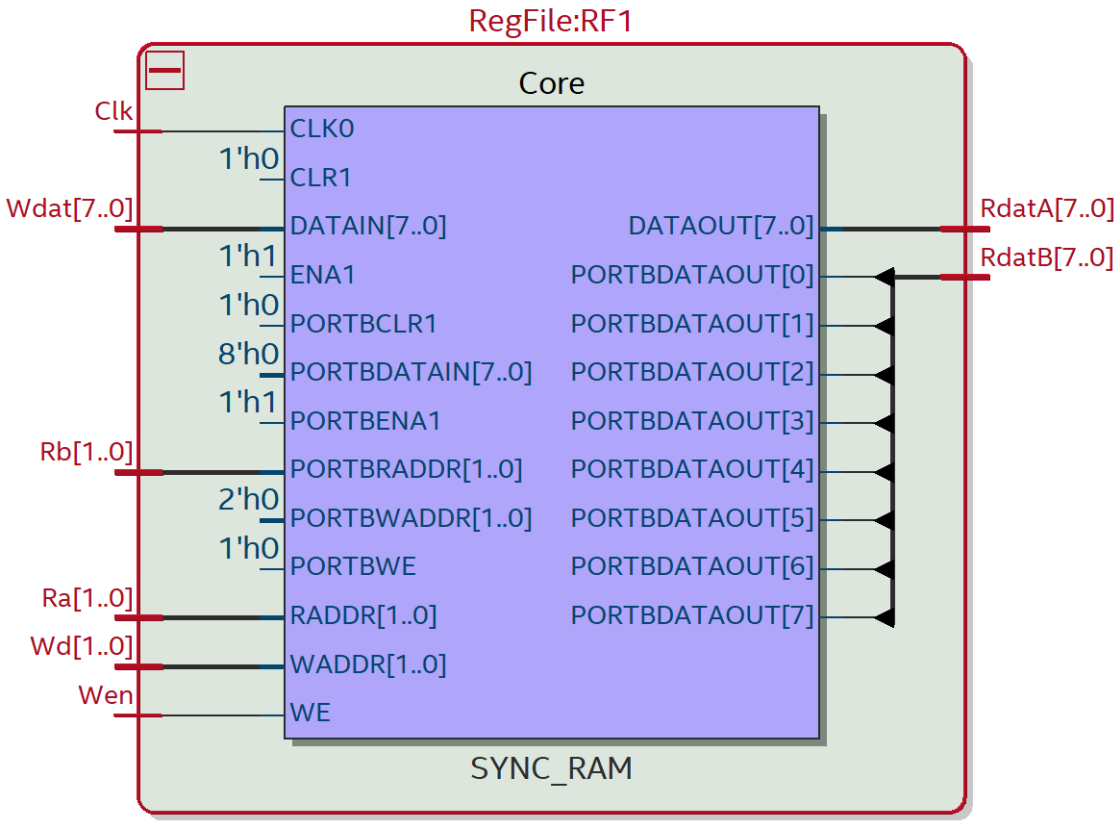
Register File

Module file name: RegFile.sv

Functionality Description

This file is responsible for storing and controlling the registers that my microprocessor will work with.

Schematic



ALU (Arithmetic Logic Unit)

Module file name: ALU.sv
Module testbench file name: ALUtb.sv

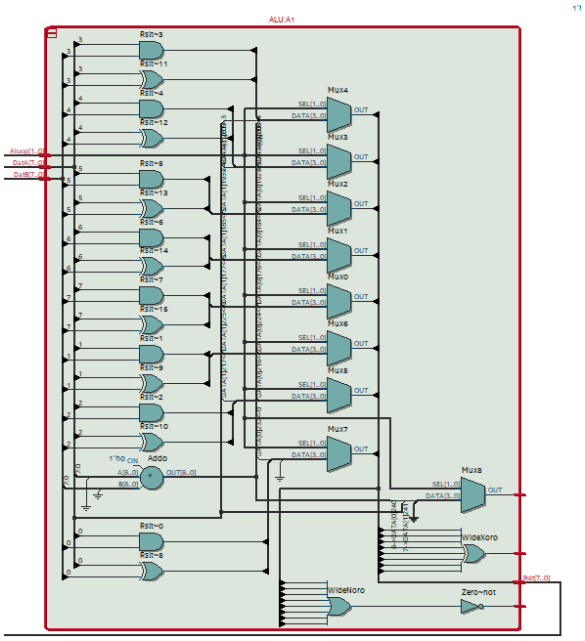
Functionality Description

This code is responsible for doing all the arithmetic operations my computer will have to run.

ALU Operations

I will be doing equals, not equals, add, sub, xor, and, shift and or operations within my ALU

Schematic



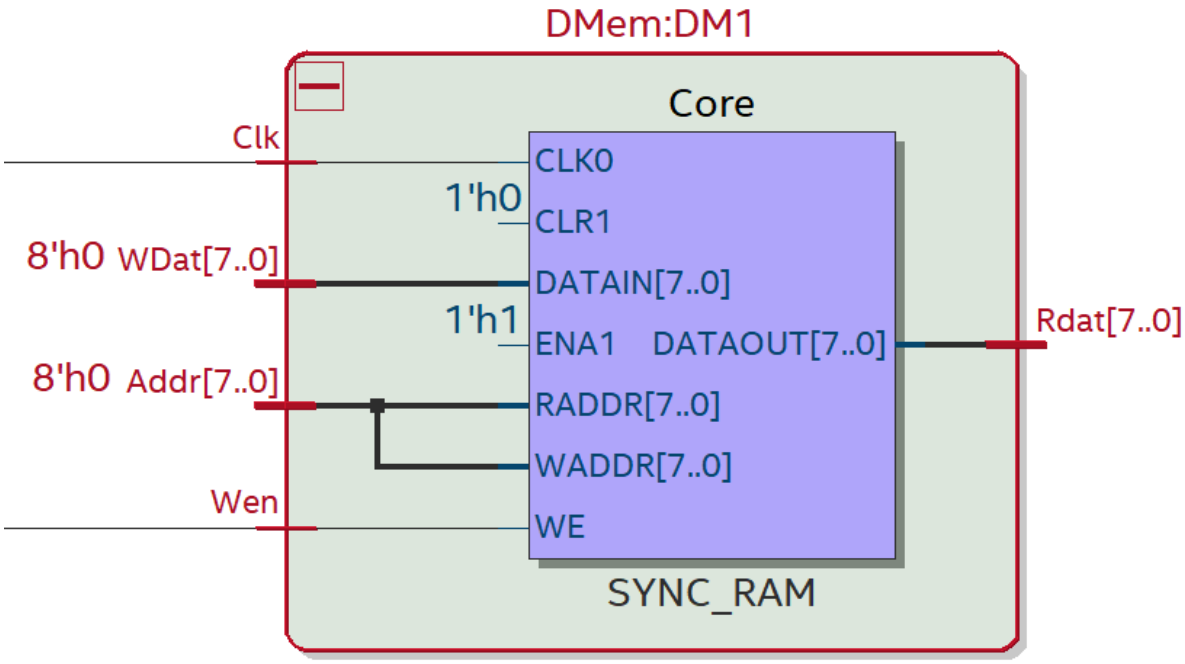
Data Memory

Module file name: DMem.sv

Functionality Description

A place for my limited number of registers to store and write information from. Also allows for branches.

Schematic



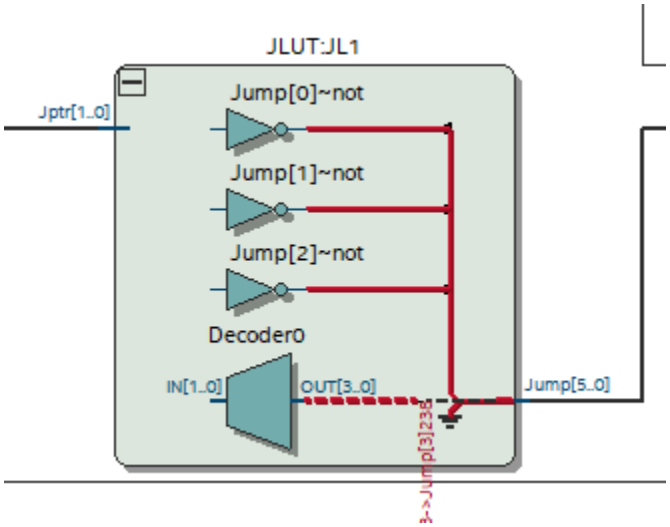
Lookup Tables

Module file name: BLUT.sv

Functionality Description

A lookup table that enables branches.

Schematic



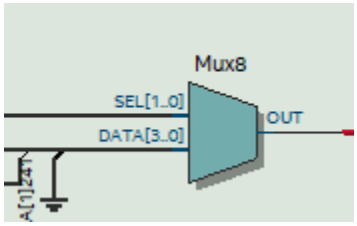
Muxes (Multiplexers)

Module file name: mux.sv

Functionality Description

Used as a decider between two options.

Schematic



7. Changelog

- Milestone 2
  - Initial version using schematics from starter code.
- Milestone 1
  - edited to change from a load/store architecture to accumulator architecture.
  - Also removed instructions
    - Removed a branch instruction to change branch instructions to beq and bne

        - Removed all my system call instructions
        - Removed Jump instruction.
    - Added some pseudo code to help me visualize the programs I need to run.
    - Added a signature to the AI form (“I excel with integrity”).

FINAL Project

Updated instruction list

Add
Sub
And
or
xor
Shift left logical
Branch on Not equal
Load Address
Store Address
Move

Added control unit logic to the opcodes I created in milestone 1 Also changed instructions from j type to i type. Branch on not equal and load are now i type instructions.

Opcode	RegDst	RegWrite	ALUSrc	MemWrite	MemRead	MemToReg	ALUOp	Branch
00000	1	1	0	0	0	0	000	0
00001	1	1	0	0	0	0	001	0
00010	1	1	0	0	0	0	010	0
00011	1	1	0	0	0	0	011	0
00100	1	1	0	0	0	0	100	0
00101	1	1	1	0	0	0	101	0
00110	0	1	1	0	1	1	000	0
00111	X	0	1	1	0	X	000	0
01000	1	1	0	0	0	0	110	0
01001	X	0	0	0	0	X	001	1
01010	X	0	X	0	0	X	XXX	0

Then I edited the given control unit starter code to work with this table.

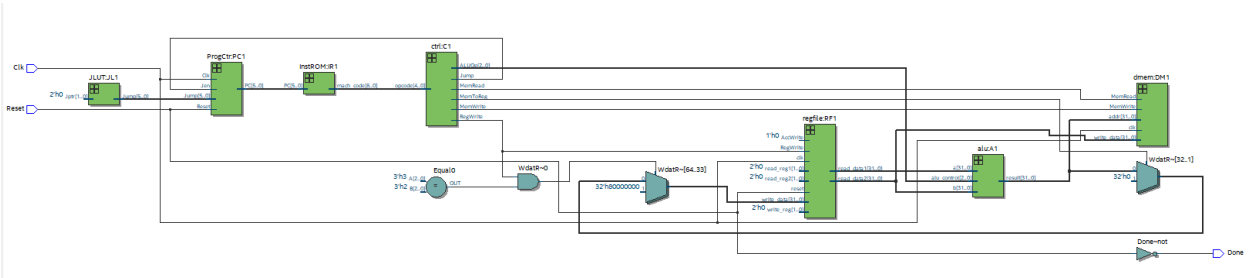
I started by piecing together all the verilog components I edited into this top level diagram.

```
module Top (
	input logic clk,
	input logic start,
	output logic done
);
	logic [7:0] address1, address2;
	logic [15:0] data_in;
	logic write_enable;
	logic [15:0] data_out1, data_out2;
	logic [4:0] hamming_distance;

	data_memory dm (
	.clk(clk),
	.address(address1),
	.data_in(data_in),
	.write_enable(write_enable),
	.data_out(data_out1)
	);

	data_memory dm2 (
	.clk(clk),
```

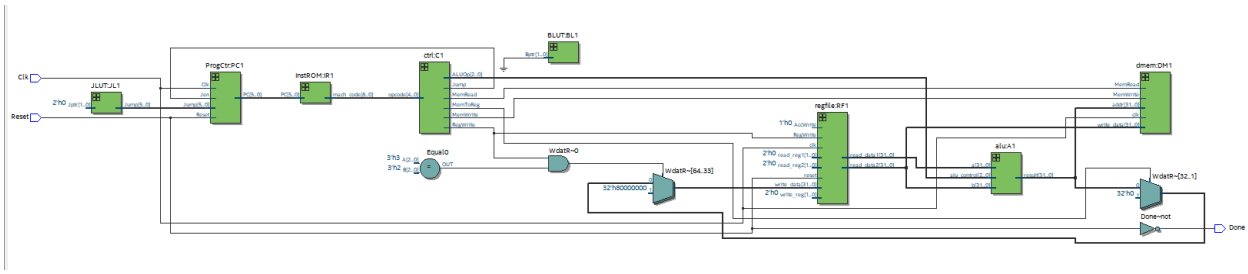
```
        .address(address2),
        .data_in(data_in),
        .write_enable(write_enable),
        .data_out(data_out2)
        );

        alu alu1 (
        .operand1(data_out1),
        .operand2(data_out2),
        .hamming_distance(hamming_distance)
        );

        control_unit cu (
        .clk(clk),
        .start(start),
        .done(done),
        .address1(address1),
        .address2(address2),
        .write_enable(write_enable),
        .data_in(data_in)
        );
endmodule
```

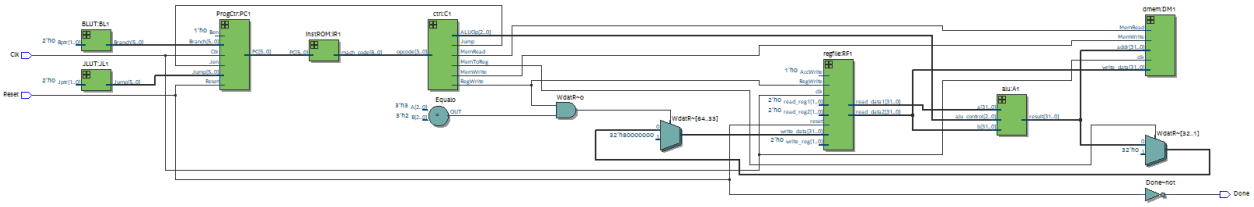
This was completely incorrect and my datamemory was a mess. I had to rethink my design.

After editing due to the problems listed above , I ended with this quartus diagram for my microprocessor design.



Realized I needed branch logic not jump changed design again got this





The idea is that there are branch and jump look up tables to allow for loops in the processor. The instrom reads the machine code I write and is dictated by the program counter. After reading an instruction, the control unit then will decide what needs to be done and use the register file or alu tools to make a decision on the subject. Data memory will be accessed as needed as well. This was my idea for developing a microprocessor and how I used verilog to create this design.

Now I started designing machine code to try to run programs.

My design allows for 16 registers with a 5 length opcode

Register 0 will be the acc
Register 8 will be the designated register for the memory address for memory access.

This was my pseudocode for calculating hamming distance

```
        ; Load the first number from memory address 0x20 into R1
        LOAD R1, 0x20

        ; Load the second number from memory address 0x24 into R2
        LOAD R2, 0x24

        ; XOR R1 and R2, store the result in R3
        XOR R3, R1, R2

        ; Initialize the count of 1s to 0 in R4
        MOVE R4, 0

        ; Initialize the bit mask to 1 in R5
        MOVE R5, 1

        ; Initialize the loop counter to 32 (assuming 32-bit numbers) in R6
        MOVE R6, 32

count_ones_loop:
        ; Check if the loop counter R6 is 0, if so, exit loop
        JUMP_IF_ZERO R6, end_loop
```

```
        ; AND R7 with R3 and R5 to check if the current bit is 1
        AND R7, R3, R5

        ; If R7 is not 0, increment the count in R4
        JUMP_IF_ZERO R7, skip_increment
        ADD R4, R4, 1

skip_increment:
        ; Left shift the bit mask in R5
        ADD R5, R5, R5

        ; Decrement the loop counter R6
        ADD R6, R6, -1

        ; Repeat the loop
        JUMP count_ones_loop

end_loop:
        ; Store the result (Hamming distance) from R4 to memory address 0x28
        STORE R4, 0x28
```

Rewrote to make easier to convert to accumulator form to be this

```
        ; Load the first number from memory address 0x20 into register R0
        LOAD R0, 0x20

        ; Load the second number from memory address 0x24 into register R1
        LOAD R1, 0x24

        ; XOR R0 and R1, store the result back in R0 (this now holds the differing bits)
        XOR R0, R0, R1

        ; Initialize the count of 1s to 0 in register R2
        MOVE R2, 0

        ; Initialize the bit mask to 1 in register R3
        MOVE R3, 1

        ; Initialize the loop counter to 32 (assuming 32-bit numbers) in register R4
        MOVE R4, 32

count_ones_loop:
        ; Check if the loop counter R4 is 0, if so, exit loop
```

```
        SUB R5, R4, 1           ; Decrement the loop counter
        Branch_on_equal R5, 0, end_loop ; If R4 is 0, jump to end_loop

        ; AND R6 with R0 and R3 to check if the current bit is 1
        AND R6, R0, R3

        ; If R6 is not 0, increment the count in R2
        Branch_on_equal R6, 0, skip_increment
        ADD R2, R2, 1

skip_increment:
        ; Left shift the bit mask in R3
        Shift left logical R3, 1

        ; Decrement the loop counter R4
        SUB R4, R4, 1

        ; Repeat the loop
        Branch_on_not_equal R4, 0, count_ones_loop

end_loop:
        ; Store the result (Hamming distance) from R2 to memory address 0x28
        STORE R2, 0x28
```

Assembly pseudocode for closest and farthest arithmetic pairs

```
        # Initialize variables and pointers
        la $t0, array     # Load address of array into $t0
        la $t1, min_diff   # Load address of min_diff into $t1
        la $t2, max_diff   # Load address of max_diff into $t2

        # Loop through all pairs of integers
        li $t3, 0          # Outer loop index
        li $t4, 0          # Inner loop index

        li $t5, 32767    # Initialize max_diff with maximum possible value (2^15 - 1)
        li $t6, -32768   # Initialize min_diff with minimum possible value (-2^15)

outer_loop:
        # Load A = array[t3]
        lw $s0, 0($t0)  # Load array[t3] into $s0
        addi $t3, $t3, 2   # Move to next element (increment by 2 bytes for 16-bit integer)

        li $t4, 0          # Reset inner loop index
```

```
        lw $s1, 0($t0)  # Load array[0] into $s1 (reset inner pointer)

inner_loop:
        beq $t3, $t4, next_outer  # If t3 equals t4, skip inner loop

        # Calculate absolute difference
        sub $s2, $s0, $s1       # $s2 = A - B
        abs $s2, $s2            # Compute absolute value of $s2

        # Check for minimum difference
        slt $t7, $s2, $t5       # Set $t7 to 1 if $s2 < $t5 (current min_diff)
        beq $t7, 1, update_min_diff

        # Check for maximum difference
        slt $t7, $t6, $s2       # Set $t7 to 1 if $s2 > $t6 (current max_diff)
        beq $t7, 1, update_max_diff

update_min_diff:
        move $t5, $s2           # Update min_diff
        j end_inner_loop        # Jump to end of inner loop

update_max_diff:
        move $t6, $s2           # Update max_diff

end_inner_loop:
        addi $t4, $t4, 2        # Move to next element in inner loop
        lw $s1, 0($t0)          # Load next element into $s1
        j inner_loop    # Jump back to inner loop

next_outer:
        addi $t3, $t3, 2        # Move to next element in outer loop
        lw $s0, 0($t0)          # Load next element into $s0
        j outer_loop    # Jump back to outer loop

# Store results in memory
end:
        # Store min_diff into memory locations 66-67 (little-endian format)
        sw $t5, 66($t1)
        # Store max_diff into memory locations 68-69 (little-endian format)
        sw $t6, 68($t2)

        # Exit program
        li $v0, 10      # System call for exit
        syscall
```

Assembly Pseudocode for 2's complement multiplication

```
        # Load operands into registers
        lw $t0, operand1        # Load operand 1 into $t0
        lw $t1, operand2        # Load operand 2 into $t1

        # Initialize variables
        li $t2, 0           # Initialize result to 0
        li $t3, 16          # Loop counter (16 bits)

        # Check sign of operand1 (t0)
        srl $t4, $t0, 31 # Extract sign bit of t0 (bit 31)
        beq $t4, $zero, pos_op1   # Branch if operand1 (t0) is positive

        # Operand1 (t0) is negative: negate it
        nor $t0, $t0, $zero # NOT t0
        addi $t0, $t0, 1          # t0 = -t0

pos_op1:
        # Check sign of operand2 (t1)
        srl $t4, $t1, 31 # Extract sign bit of t1 (bit 31)
        beq $t4, $zero, pos_op2   # Branch if operand2 (t1) is positive

        # Operand2 (t1) is negative: negate it
        nor $t1, $t1, $zero # NOT t1
        addi $t1, $t1, 1          # t1 = -t1

pos_op2:
        # Multiply operands using left shift and conditional addition
        li $t4, 0           # Initialize shift counter

mul_loop:
        sll $t0, $t0, 1   # Shift t0 left (multiply by 2)
        bne $t1, $zero, add_t0   # Branch if t1 is not zero

        bne $t3, $zero, sub_t0   # Branch if t3 is not zero
        j store_t0                # Jump to store result

add_t0:
        add $t2, $t2, $t0         # Add t0 to t2 (accumulate result)
```

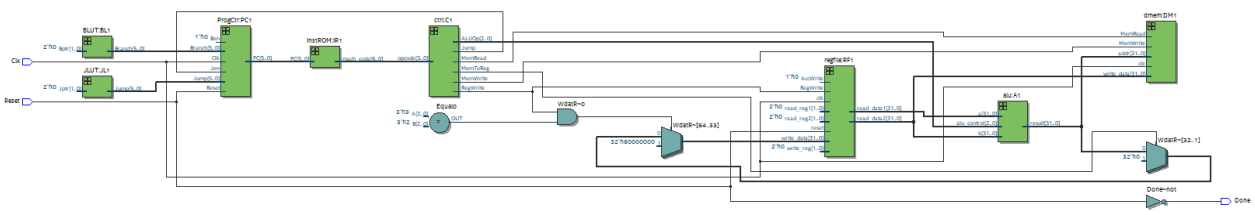
```
j next _shift
```

Hemming distance was not in accumulator form or in my language which my machine was so I had to edit the code

```
; Load the first number from memory address 0x0 (example) into accumulator(ACC)
MOV   0x00
MOV   R7
LOAD (will always load the value of the address in R7)

; Load the first number from memory address 0x0 (example)  into accumulator(ACC)
MOV R1

; Load the second number from memory address 0x16 (example)  into register R1
MOV   0x16
MOV   R7
LOAD (will always load the value of the address in R7)

; XOR ACC and R1, store the result back in R1
XOR R1
MOV R1

; Initialize the count of 1s to 0 in register R2
MOV 0
MOV R2

; Initialize the bit mask to 1111 in register R3
MOV 15
MOV R3

; Initialize the loop counter to 8 (assuming 8 bit numbers) in register R4
MOV 8
MOV R4

count_ones_loop:

MOV   0       set acc to be 0
MOV   R5      set r5 to be 0
MOV   0       set accumulator to be memory address 0
MOV   R7      set computer to load at this address (moves memory address 0 to r7)
Store   R4    stores loop number at address 0
Load          set acc to be R4's value which starts out as 16

Branch_not_equal R5  19, end_loop Checks if r4 is 0
```

```
        MOV   0       set accumulator to be memory address 0
        MOV   R7      set computer to load at this address (moves memory address 0 to r7)
        Store R1      stores R1 (the xor value) value at address 0
        MOV   0       set accumulator to be memory address 0
        Load          loads the xor value into my accumulator

        AND   R3      and with the xor value with a bit mask of 1

        MOV   R5      Moves this value into R5
        MOV   3       sets acc to be 2 so can store it in R9 for branching
        MOV   R9      Moves branch value to register saved for branching
        MOV   0       Moves 0 into ACC

        Branch_not_equal R5          (compares r5 with 0 and jumps forward 3 if fails)

        MOV   1       set acc to be   1
        ADD R2        add 1 to R2

skip_increment:
        Shift left logical R3

        Load          set acc to be R4
        SUB   1       ; Decrement the loop counter
        MOV   R4

        Branch_not_equal R4, -25,     count_ones_loop

end_loop:
        ; Store the result (Hamming distance) from R2 to memory address 0x28
        MOV   0       set accumulator to be memory address 0
        MOV   R7      set computer to load at this address (moves memory address 0 to r7)
        Store R2      stores R2 hemming dist at address 0
```

Edited verilog again to get this output because branching didnt make sense

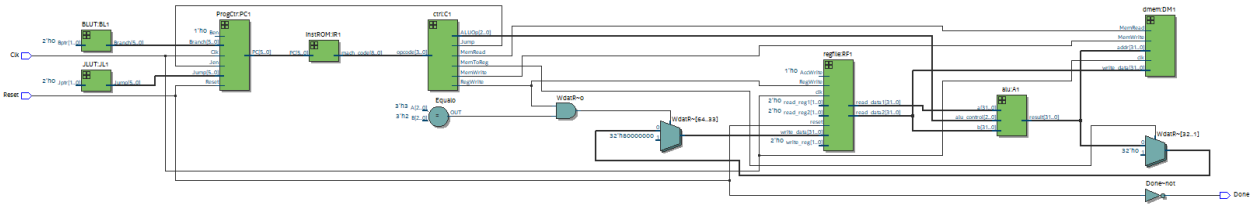


Added displays to all of my design components to try to debug why my machine wasn't running on the machine code and the verilog code I ran.

My design completely compiles in model sim and quartus. It is supposed to access data memory by loading and immediate value into the accumulator which then moves it to a designated register R7 so when loading values from data memory it uses the address stored there to load into my program. To store values, it uses the same register R7's address and takes a register's value to put into memory.

In my design Branching is performed absolutely. My BLUT has values that I manually counted from my machine code on how far to branch. Based on a value stored in register R9, the computer would then know which value to branch to.

Unfortunately I was caught up debugging and trying to get program one to run, so I only have hard coded machine code and values in the BLUT for hamming distance, but the above description is how the machine is supposed to work.



Quartus diagram of my project.

In order to edit the test bench to run on my program, I had to modify the names of most of the data memory calls, and also add mt top level as demonstrated in the code.

The new test bench for hamming distance

```
module test_bench;

  // Connections to DUT: clock, start (request), done (acknowledge)
  bit clk;
  logic reset, start;
  wire done;

  logic [4:0] Dist, Min, Max;   // current, min, max Hamming distances
  logic [4:0] Min1, Min2;       // addresses of pair w/ smallest Hamming distance
  logic [4:0] Max1, Max2;       // addresses of pair w/ largest Hamming distance
  logic [15:0] Tmp[32];         // cache of 16-bit values assembled from data_mem

  // Instantiate Top
```

```
Top D1 (
	.Clk(clk),
	.Reset(reset),
	.Done(done)
);

// Clock generation
initial begin
	clk = 0;
	$display("Time = %t, clk = %b", $time, clk);
	forever #50 clk = ~clk; // 100 ns period clock
end

// Number of tests
int itrs = 10;
int min_pass = 0;
int max_pass = 0;

initial begin
	// Reset system
	reset = 1;
	#100 reset = 0;
	$display("Reset asserted at time %t", $time);

	// Load operands for program 1 into data memory
	// 32 double-precision operands go into data_mem [0:63]
	// first operand = {data_mem[0],data_mem[1]}
	// endian order doesn't matter for program 1, as long as consistent for all values (why?)
	for (int loop_ct = 0; loop_ct < itrs; loop_ct++) begin
	#100;
	Min = 'd16;  // start test bench Min at max value
	Max = 'd0;   // start test bench Max at min value
	case(loop_ct)
	0: $readmemb("test0.txt", D1.DM1.memory);
	1: $readmemb("test1.txt", D1.DM1.memory);
	2: $readmemb("test2.txt", D1.DM1.memory);
	3: $readmemb("test3.txt", D1.DM1.memory);
	4: $readmemb("test4.txt", D1.DM1.memory);
	5: $readmemb("test5.txt", D1.DM1.memory);
	6: $readmemb("test6.txt", D1.DM1.memory);
	7: $readmemb("test7.txt", D1.DM1.memory);
	8: $readmemb("test8.txt", D1.DM1.memory);
	9: $readmemb("test9.txt", D1.DM1.memory);
	endcase
```

```
$display("Loading test%d.txt at time %t", loop_ct, $time);

for (int i = 0; i < 32; i++) begin
Tmp[i] = {D1.DM1.memory[2*i], D1.DM1.memory[2*i+1]};
$display("%d: %b", i, Tmp[i]);
end

// DUT data memory preloads beyond [63]
D1.DM1.memory[64] = 'd16;  // preset DUT final Min to max possible
for (int r = 65; r < 256; r++)
D1.DM1.memory[r] = 'd0;  // preset DUT final Max to min possible

$display("Data memory preloading complete at time %t", $time);

// Compute correct answers
for (int j = 0; j < 32; j++) begin
for (int k = j + 1; k < 32; k++) begin
#1 Dist = ham(Tmp[j], Tmp[k]);
if (Dist < Min) begin  // update Hamming minimum
Min = Dist;  // value
Min2 = j;         // location of data pair
Min1 = k;         // "
end
if (Dist > Max) begin  // update Hamming maximum
Max = Dist;  // value
Max2 = j;         // location of data pair
Max1 = k;         // "
end
end
end

$display("Computation complete at time %t", $time);

// Pulse start signal to begin the operation
start = 1;
#200 start = 0;
$display("Start pulse at time %t", $time);

// Wait for done signal from the DUT
wait (done);
$display("Done signal received at time %t", $time);
```

```
        // Check results in data_mem[64] and [65] (Minimum and Maximum distances,
respectively)
        if (Min == D1.DM1.memory[64]) begin
        $display("good Min = %d", Min);
        min_pass++;
        end else begin
        $display("fail Min: Correct = %d; Yours = %d", Min, D1.DM1.memory[64]);
        $display("Min addr = %d, %d", Min1, Min2);
        $display("Min valu = %b, %b", Tmp[Min1], Tmp[Min2]);
        end

        if (Max == D1.DM1.memory[65]) begin
        $display("good Max = %d", Max);
        max_pass++;
        end else begin
        $display("fail Max: Correct = %d; Yours = %d", Max, D1.DM1.memory[65]);
        $display("Max addr = %d, %d", Max1, Max2);
        $display("Max valu = %b, %b", Tmp[Max1], Tmp[Max2]);
        end

        if (loop_ct == itrs - 1) begin
        $display("Minimum correct %d/%d", min_pass, itrs);
        $display("Maximum correct %d/%d", max_pass, itrs);
        $stop;
        end

        #200 start = 0;
        $display("Test iteration %d complete at time %t", loop_ct, $time);
        end
 end

 // Hamming distance (anticorrelation) between two 16-bit numbers
 function [4:0] ham(input [15:0] a, b);
        ham = 'b0;
        for (int q = 0; q < 16; q++)
        if (a[q] ^ b[q]) ham++;  // count number of bits for which a[i] = !b[i]
 endfunction

Endmodule
```

My machine code also uploaded as machine_code.txt
0110 0000
1000 0111
0110 0111

1000 0001
0110 0001
1000 0111
0110 0111
0100 0001
1000 0001
0110 0000
1000 0010
0110 0000
1000 0011
0110 0000
1000 0100
0110 0000
1000 0101
0110 0000
1000 0111
0111 0100
0110 0111
1001 0101
0110 0000
1000 0111
0111 0001
0110 0000
0110 0111
0010 0011
1000 0101
0110 0000
1000 1001
0110 0000
1001 0101
0110 0000
0000 0010
0101 0011
0110 0111
0001 0001
1000 0100
1001 0100
0110 0000
1000 0111
0111 0010

Unfortunately, after trying to run my programs in model sim. Nothing happens. Everything compiles and I wrote machine code based off the pseudo code showed way up above. I can

cleanly compile in quartus and modelsim. I include all the text files from the class and the test bench. I included the mach_code.txt file and updated it in my instrom call. I have added display statements trying to find where in the design my code is running. None of these have printed anything out in the transcript section of model sim while I was simulating. I believe to get this machine up and running I would have to go through my verilog even more times especially in my control unit to figure out why nothing is running when I run the simulation. I even added display statements to the test bench to see if I changed the initialization of the clock. I strongly believe that my verilog is the weakness in my code, and that the logic/pseudocode behind my machine code is correct. Even if there are syntax errors I have yet to debug. So for next steps in my project, that is where I would go. I have changed instructions multiple times, tried writing data memory in verilog differently, but nothing loads or starts. I believe I am initializing the clock correctly, but it is very confusing. The best way of fixing this would probably be to double check how my verilog goes from section to section because I do not think it is passing through as intended.