# Lab7 – RISC-V CPU Implementation

## Introduction

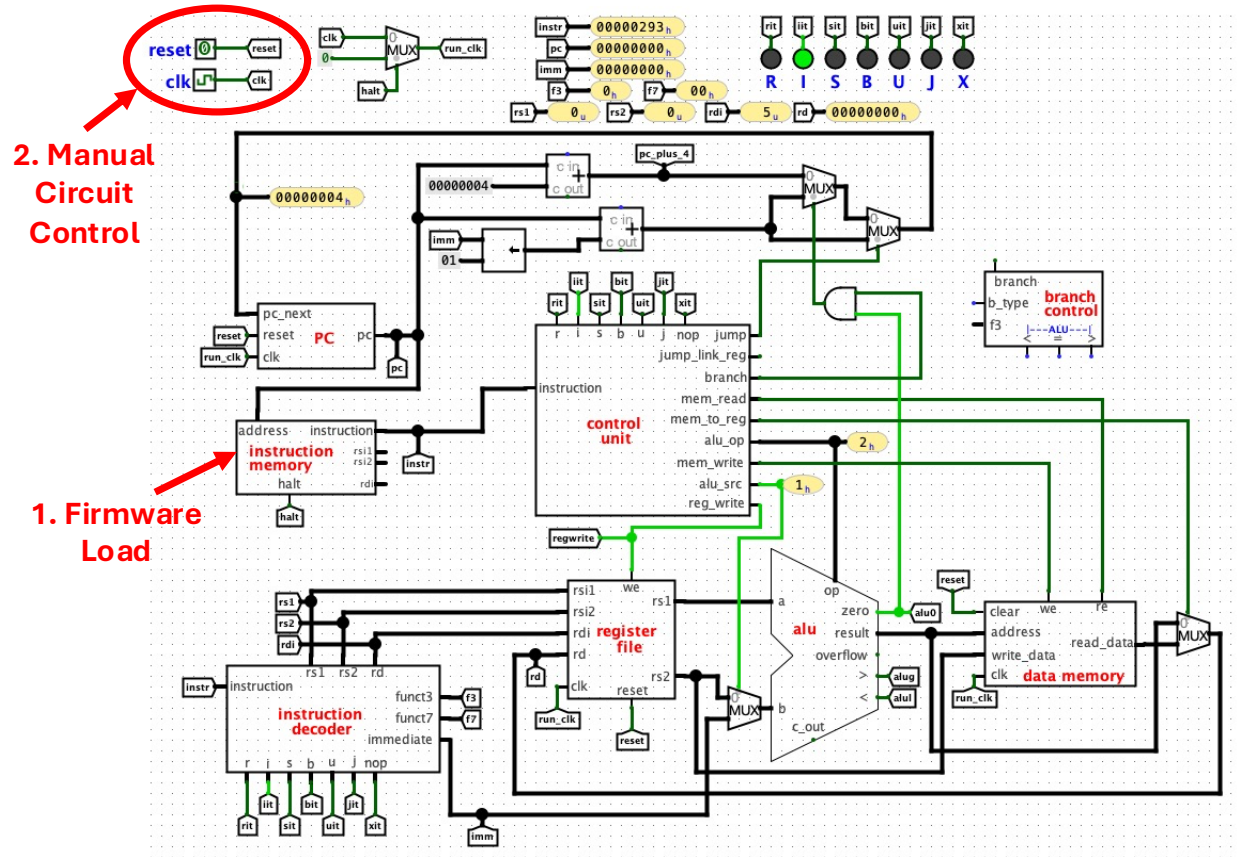
For this lab you will be using Logisim Evolution along with a starter file to extend a working RISC-V CPU that I have built and make it more robust.

## PART 0: Prerequisites

For this lab you will need the Venus Simulator and Logisim Evolution installed (you should already have this setup from previous labs). You will also need a working python environment for this lab. If you don't already have it installed on your machine, you can install it from: https://www.python.org/

## PART 1: Getting Familiar with the Starter File

The starter circuit file that I provided, "`rv5i_starter`" contains a functioning RISC-V CPU that follows the design of the single cycle processor we have been studying in class. Open this circuit and get familiar with its workings.




Let's start by loading and running a simple program to make sure things are working correctly.

To start, lets create a simple program that loops 5 times:

```
.text
    .globl  main      # assembly directive that makes the symbol main
                      # global and this is where execution starts

main:

    li t0, 0
    li t1, 5

    loop:
        beq t0, t1, end
        addi t0, t0, 1
        j loop
    end:

    mv t2, t1  #t2 should equal 5
```

Run this program in the Venus simulator to make sure it executes properly. At the end the `t2` register should contain the value of 5.

Now let's run it again and bring up the assembler view using the Venus tools:

```
0x00000000      0x00000293      addi x5 x0 0
0x00000004      0x00500313      addi x6 x0 5
0x00000008      0x00628663      beq x5 x6 12
0x0000000C      0x00128293      addi x5 x5 1
0x00000010      0xFF9FF06F      jal x0 -8
0x00000014      0x00030393      addi x7 x6 0
```

Copy this to a file by either copying to the clip board and then creating a new file, or choose "Save As..." Give the file the name **`count_5.d`** (we will use the .d for the disassembler file).

Now lets create a rom file from this file. Run the **`firmware.py`** file that I provided in the starter kit:

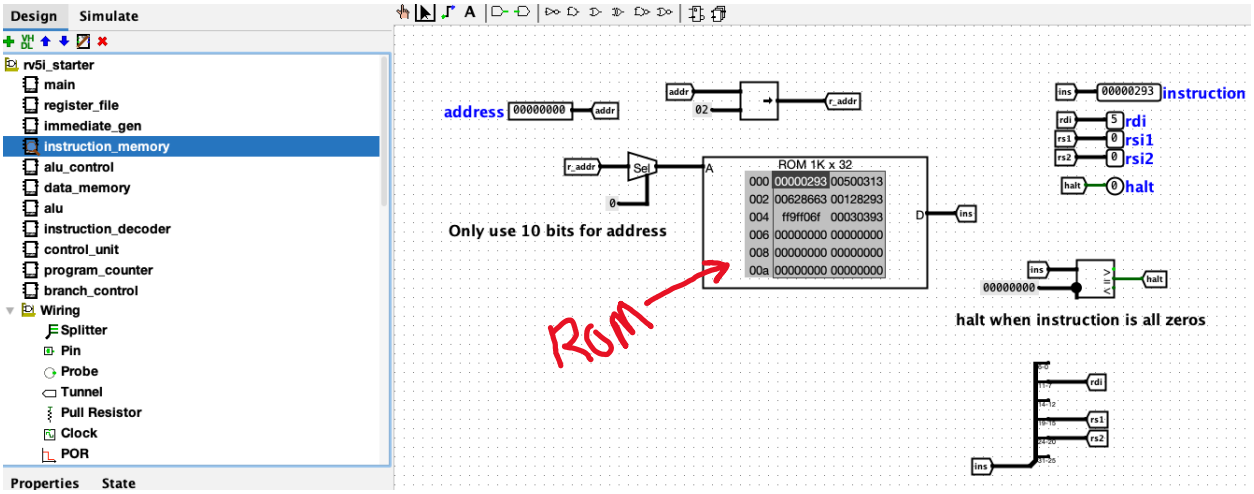
**`python firmware.py count_5.d count_5.rom`**

The python program that I provided takes the disassembly file as the first parameter and creates a file with the second parameter that will be the firmware rom image. This file should look like the below (but will be much longer):

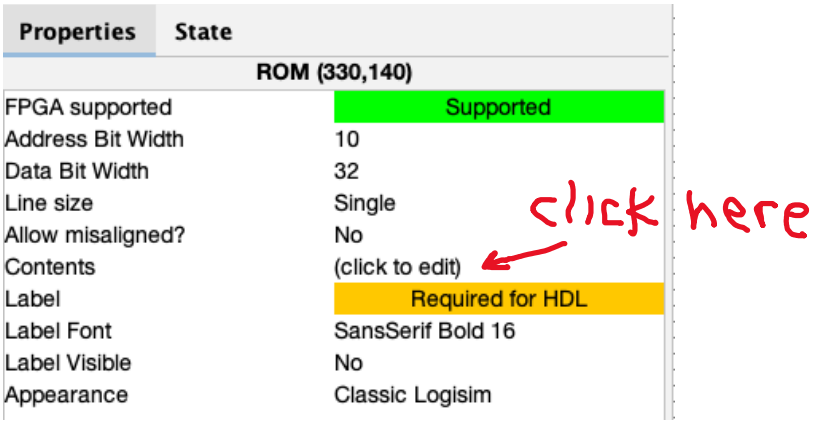
v3.0 hex words addressed
000: 00000293 00500313 00628663 00128293 FF9FF06F 00030393 00000000 00000000
008: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Now that you have created your firmware, lets load it and test it.

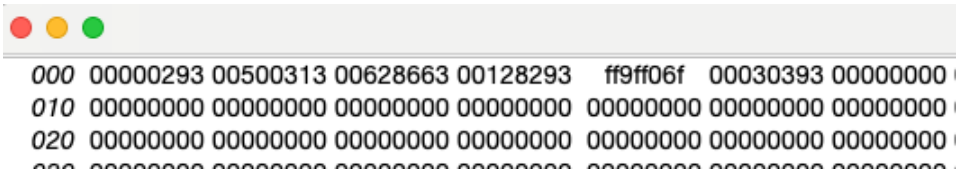
From Logisim, open the "`instruction_memory`" circuit:




Then click on the ROM component. Inspect the properties on the lower left, then click on the (click to edit) option for the Contents property:

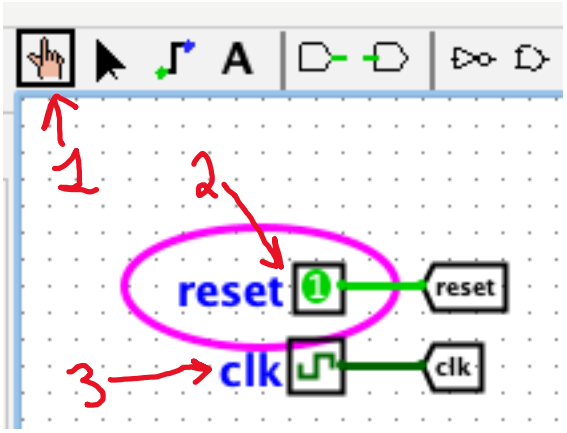



A big window should come up with a lot of zeros on it representing the ROM contents. Click on the "Open..." button on the bottom. Then select the ROM file that you created. If everything went OK you should see something like this:




Now hit "Close Window"

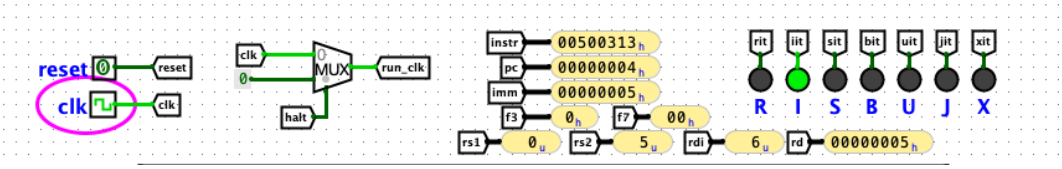
Now go back to the main circuit ->



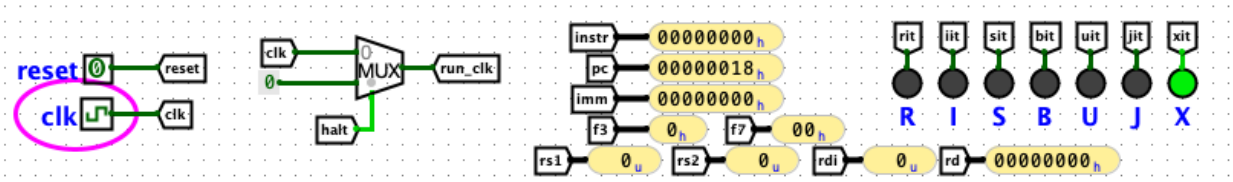

Now ensure the CPU is reset and ready to run:

1. Click on the “hand” icon to enter simulation mode
2. Click on the “reset” pin to reset the CPU
3. Click on the “clk” pin a few times to cycle the clock manually.

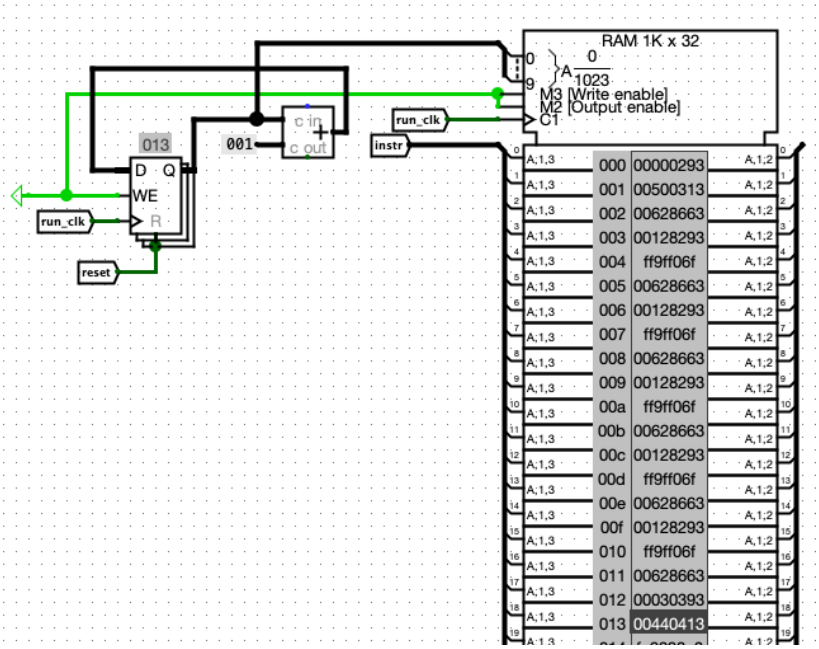
After you do this, click on the “`reset`” button again to put the CPU in run mode by ensuring it has the value of zero.




Now run the program by clicking on the `clk` button. Each time you click the program counter will advance and an instruction will be executed. You keep clicking on the `clk` until the program finishes running. You will know this when the `X LED` turns on:




You should also be able to see the instructions that were executed by scrolling down to the bottom of the circuit:




Now that you have successfully executed a program we can move onto the next part of the lab where you will be adding some features to the RISC-V CPU. Please be sure you are comfortable at this point before moving on. You will be creating different firmware for the next parts of the lab. So you need to be familiar with using Venus and creating the `.d` (disassembly) and `.rom` (firmware) files from above, as well as loading the ROM file into the instruction memory ROM.

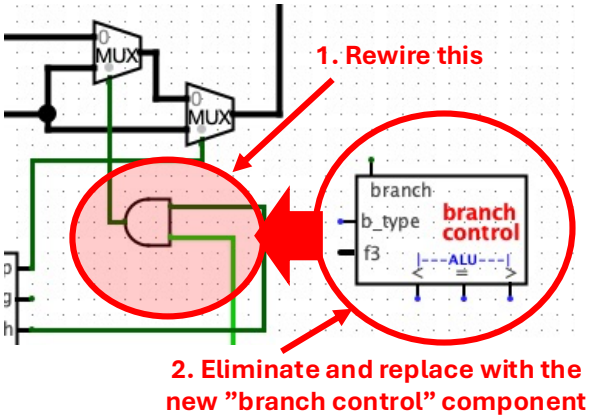
**PART 2: Making Branch More Robust**

Now that you have the lay of the land let's look at another program, in the starter file its called `count_5v2.s`. Work through the process we discussed in Part 1 and create a ROM file. Load this ROM file into the instruction memory. Then reset the CPU and manually advance the clock. The source code is here:

```
.text
    .globl  main      # assembly directive that makes the symbol main
                      # global and this is where execution starts

main:

    li t0, 0
    li t1, 5

    loop:
        addi t0, t0, 1
        bne t0, t1, loop
    end:
```

```
    mv t2, t1  #t2 should equal 5
```

Notice that this time it does not work. This is because the CPU knows how to process beq, but does not know how to process bne. Lucky for you, I created a new "branch control" component that knows how to deal with both bne and beq.

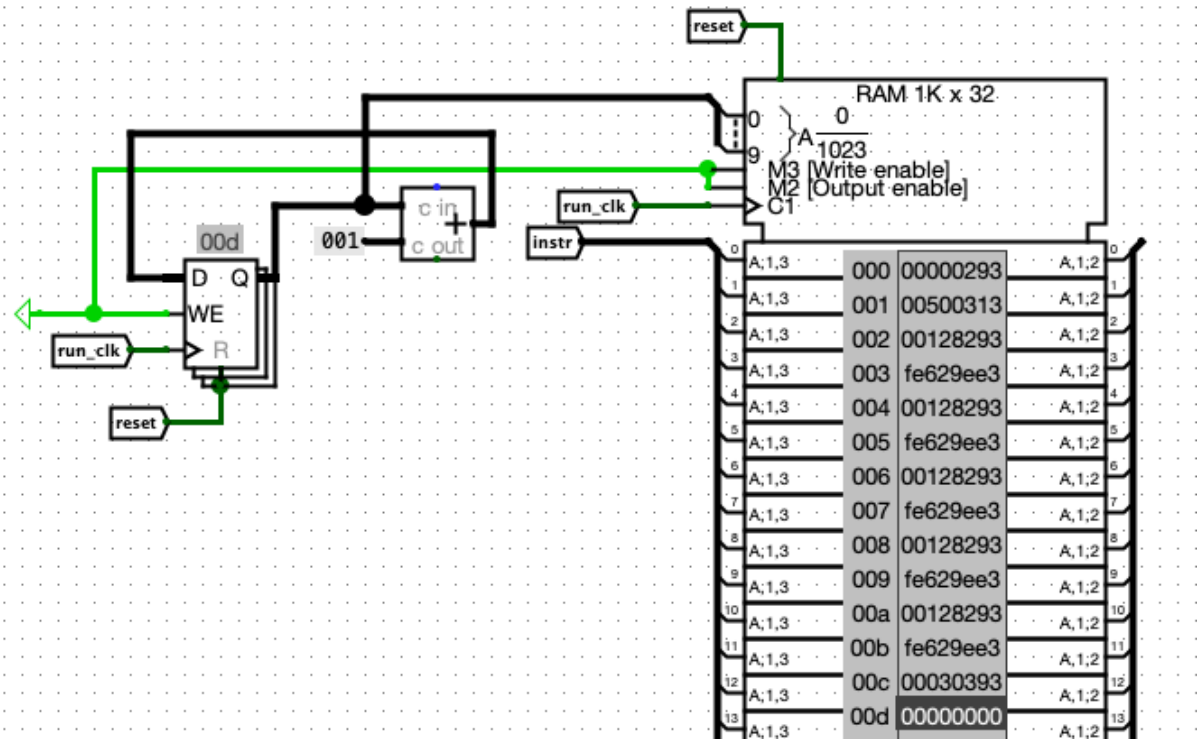



See above, for this part of the lab:

1. Eliminate the and gate that goes into the top mux. Recall that this mux has the branch address on `port 1` and pc+4 on `port 0`.
2. Rewire in the branch control component. This component has the following inputs:
    a. "`b_type`": This input receives the "branch" output from the control unit
    b. "`f3`": This input receives the function-3 input from the current instruction. It will contain the code for the type of the branch. See the data sheet. The beq instruction has `f3=000` and bne has `f3=001`. Notice that there is already a tunnel named "`f3`" that you can use for this input. It comes out of the "`instruction decoder`" component.
    c. "<": This input is 1 if a<b based on the ALU input. It is the "<" pin on the ALU. There is also an existing tunnel named "`alul`" (alu less) that you can use here.
    d. ">": This input is 1 if a>b based on the ALU input. It is the ">" pin on the ALU. There is also an existing tunnel named "`alug`" (alu greater) that you can use here.
    e. "=": This input is 1 if a==b based on the ALU input. It is the "zero" pin on the ALU. There is also an existing tunnel named "`alu0`" (alu zero) that you can use here.

After you rewire the "`branch`" control component rerun the `count_5v2` firmware and make sure it works properly.

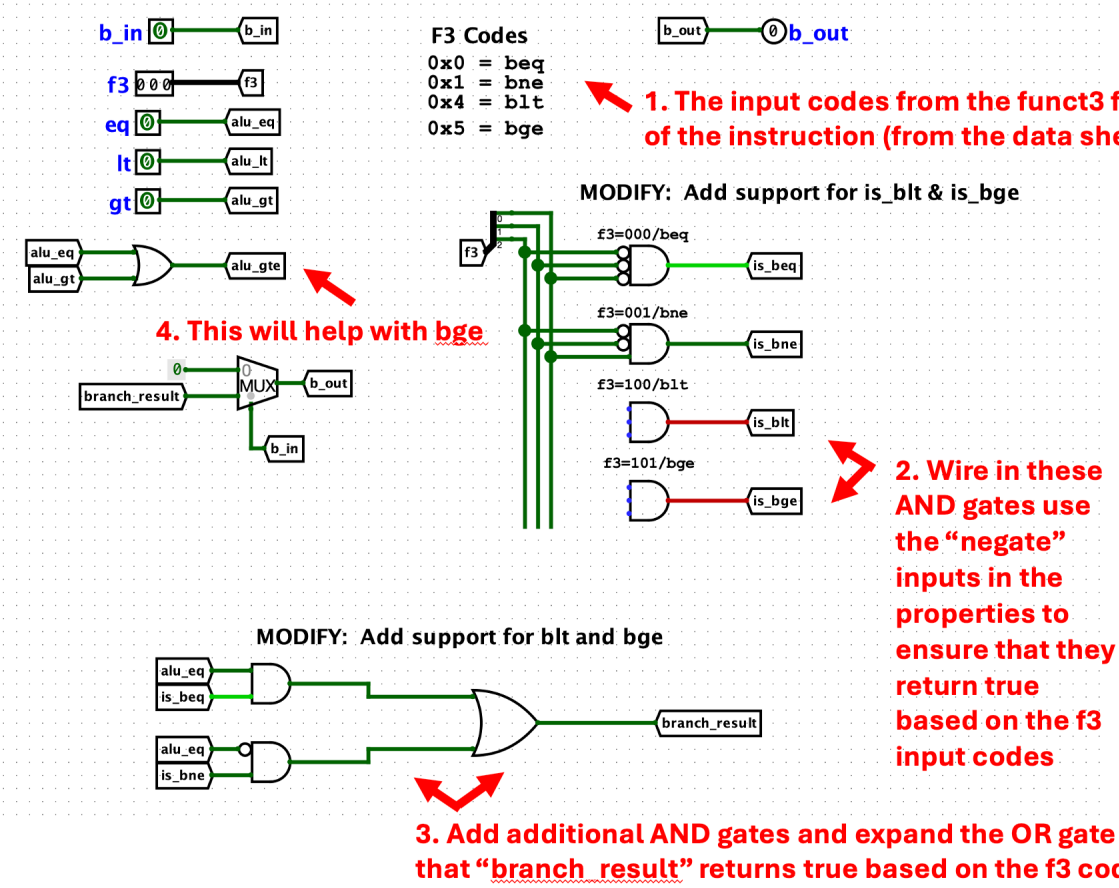
**What to hand in: Hand in an image showing the debug memory for the instructions that were executed once you fixed the ALU. It should look like the below:**




**PART 3: Continuing Making Branch More Robust**

Now we can handle both bne and beq. Let's add more features, specifically supporting the `blt` and bge instructions. Check your data sheet, with support for beq, bne, blt, and bge we get for free `beqz`, `bnez`, `blez`, `bgez`, `bltz`, `bgtz`, `bgt`, and `ble`. All these instructions are translated into the ones that we implemented.

To get started, open up the "`branch control`" component. See below:

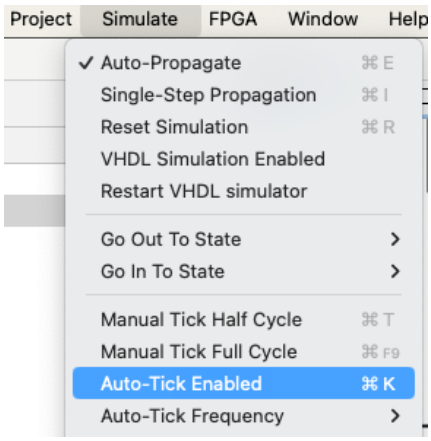



From the starter circuit (above):

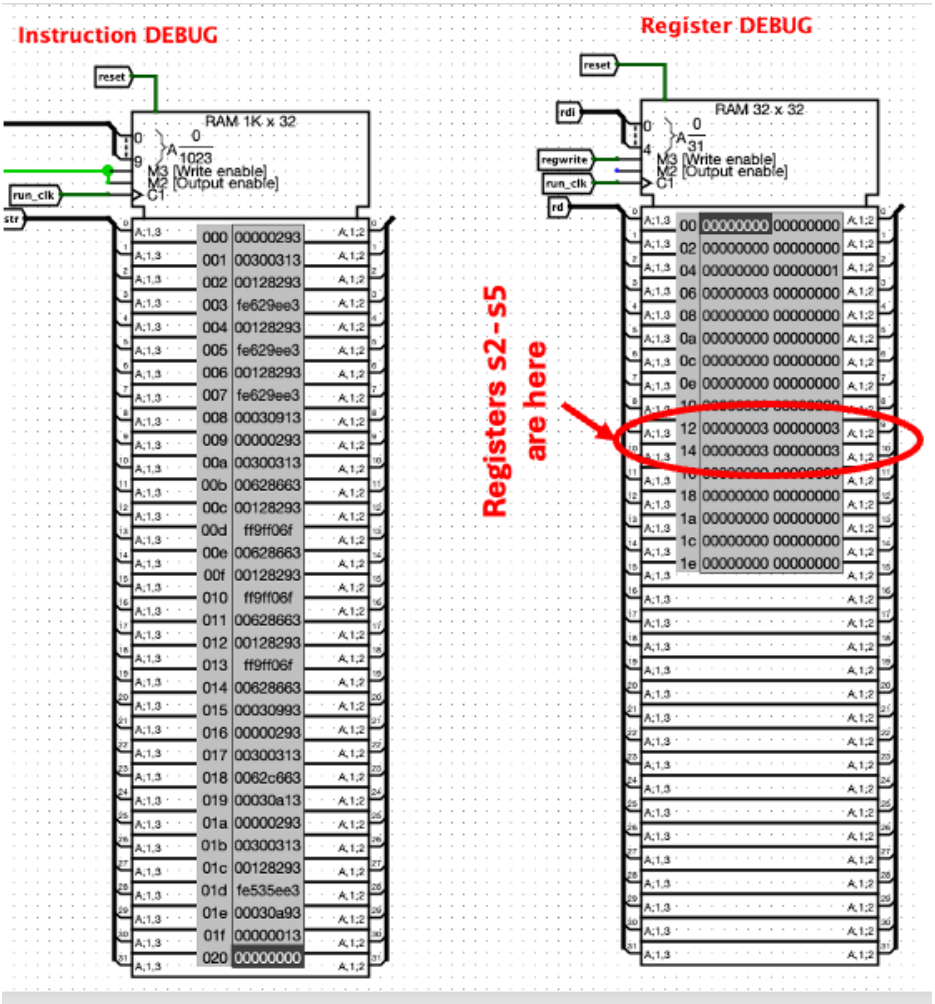
1. These are the `f3` codes for the various branches we need to support
2. Notice how I wired in beq and bne. Wire `blt` and bge properly. Use the proper `f3` bits and set the "bubbles" to negate the inputs to ensure that `is_blt` and `is_bge` returns true (1) if and only if the 3 `f3` bits represent a 4 and 5 respectively.
3. Now add 2 additional ANDs and wire into an OR (you will need to expand the number of inputs to 4 so that "`branch_result`" is properly set. This is the output of the component.

Now it's time to test again.

In the starter collection you will find a file named `branch_test.s`. This program runs all 4 branches and then dumps a counter out in registers `s2`, `s3`, `s4`, and `s5` respectively. To keep things shorter, the counters all loop 3 times. You can run manually by clicking the clock after resetting or use the "`Auto-Tick Enabled`" feature in Logisim to automate the process. I would suggest playing with the "`Auto-Tick Frequency`" to speed up and slow down the simulation. See below.



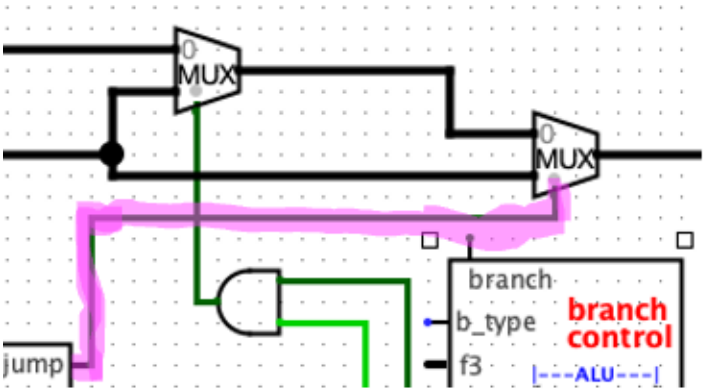

The final result, if everything is working correctly is that registers s2-s5 will have the value of 3 in them. You can check this by going into the register file component. There is also a Register Debug RAM at the bottom of the circuit that updates itself as register values change. You should have a result in the debugging RAMs that looks like:



**WHAT TO HAND IN FOR PART3 – A screenshot similar to above (no red annotations required) where the instruction execution loop and registers are clearly shown.**

**PART 4: Fixing Jumps**

We have used jump (`j`) instructions in several of our programs and everything seems to have worked properly. This is because we just used simple jumps and the `control_unit` has an output `jump` pin that is routed into a `mux` that correctly sets the PC value. Based on the starter file, any instruction that is simply `j label` will work properly.




If we study the data sheet we will see that the jump instruction that we use for simple control flow (example: `j loop`) and the `jal` instruction that we use to call functions (example: `jal call_function`) both get translated into another format. The format is an expanded version of `jal`, which is a J-Type of instruction. Specifically:

- `j offset` gets translated to `jal x0, offset`, and
- `jal offset` gets translated to `jal x1 offset`

Given the above we look into the definition of jal and see that the full version of jal looks like the following:

`jal Rd,immediate` which does 2 things:

1. `rd = pc+4`: changes the Rd register setting it to pc+4
2. `PC += immediate`: changes the pc to an offset using the immediate value

In other words, the J-Type of instruction also needs to modify the `Rd` register using the value of `pc+4`. If you look into the above you hopefully can connect how this works because the basic jump instruction tries to set the `zero` register to `pc+4` (this is ignored because the `zero` register cannot be modified). The other form that we use for function calls modifies the `x1` register, which is the `ra` register that we have talked about in class to handle properly returing from functions.

What you need to do is to fix the starter file to properly support jumps by placing pc+4 on the rd input line of the register file when we encounter a J-Type of instruction. Hints:

- You will need to insert a new mux into the data path that can control if the default value is placed into rd from the mem_to_reg mux.
- There are existing tunnels you can use to make the wiring less messy. Specifically, the ppp4 tunnel which has the current pc+4 address, and the jit tunnel which is true if the current instruction is J-Type

After you make the changes, prepare and run this simple program (function.s)to see if your changes work. Also, this is the first time we are seeing ecall in this lab. I have the starter circuit designed so that any ecall instruction just halts the processor.

```
.text
    .globl  main      # assembly directive that makes the symbol main
                      # global and this is where execution starts

main:

    li a0, 0

    jal add_one
    mv s2, a0
    ecall

add_one:
    addi a0, a0, 1
    jr ra
```

**WHAT TO HAND IN FOR PART4 – A screenshot showing the changes that you made to support injecting pc+4 properly into the rd input of the register file. Also, the instruction trace associated with executing the sample program above. Like in the previous section this would be a screen print of the "Instruction DEBUG" RAM.**

**PART 5: Fixing Jumps (Part 2)**

In the previous part of this lab we fixed the processor so that it properly supported basic jumps and jal instructions to make function calls. If you executed the sample program in part 4 hopefully you noticed that it almost but did not fully work properly. Specifically, the jr ra instruction in the add_one function did not set the pc to the address in ra. It just went to pc+4, which in the ROM file was an instruction of all zeros – 0x0000000. The sample circuit halts the processor when it encounters an instruction of all zeros.

Going back to our datasheet we see that the instruction:

`jr ra` gets translated into `jalr x0,ra,0` which is an I-Type of instruction

The good news is that our starter circuit already supports I-Type instructions properly out of the box. Reading the documentation for tells us `jalr x0,ra,0` that when this instruction executes it calculates `ra+immediate`, which in this case is `ra+0`, which is just `ra` as the output from the ALU. The problem is that the starter circuit does not have the ability to inject this address into the path that updates the program counter. Note: this is very similar to the change you had to do as part of Homework 6.

To fix this, what you need to do:

1. Capture the value out of the ALU, note that there is a tunnel already in place named `alu_out` where you can get this value.
2. Determine if the instruction is `jalr`, which is also trivial because a tunnel is already in place named `jalr` that comes out of the control unit that will be 1 if the instruction is jalr x0,ra,0 or 0 otherwise.
3. Update the data path to inject the value from the `alu_out` tunnel into the path that goes into the `pc_next` input of the PC component if the instruction is of type `jalr`.

After you make the changes, prepare and run the simple program we used in part 4 of this lab to see if your changes work. This time the `jr ra` instruction should properly return to the `main` function where the next instruction executed is the `mv s2,a0`.

**WHAT TO HAND IN FOR PART5 – You will hand in screen shots of the changes you made and the `Instruction DEBUG` RAM (basically the same thing as in Part 4, but this time showing that the entire program executes correctly.**

**PART 6: Final Test**

In this final part of the lab we will use a modified program from Lab 4 (recursion) to test our processor changes. The program below uses recursion to add the parameters passed to the better_multiply function. To get started:

1. Disassemble and create a ROM file for this program
2. Load the ROM file into the `instruction memory`
3. Run the program (I would suggest using the "`Auto-Tick Enabled`" feature of the clock

```
.text
    .globl  main      # assembly directive that makes the symbol main
                      # global and this is where execution starts

main:
```

```
   # Load the values of a and b from memory
    li s0, 5         # a = 5
    li s1, 3         #

    # Call the multiply function
    mv a0, s0
    mv a1, s1
    jal better_multiply

    # a0 should have the result

    ecall

#######################################
#ALGORITHM
#  multiply(a,b):
#    if b == 0:
#      return 0
#    else:
#      return a + multiply(a, b-1)
#######################################
better_multiply:
    bgt a0,a1, multiply
    nop                 #you need to figure this out
multiply:
    # Save the current state on the stack
    addi sp, sp, -8     # Push 2 words on stack
    sw a0, 4(sp)        # Save a
    sw ra, 0(sp)        # Save ra

    # Base case: b == 0
    bne a1, zero, recursive_case
    li   a0, 0          # Return 0
    addi sp, sp, 8      # Pop the stack
    jr   ra

recursive_case:
    # Recursive case: result += a, b--
    addi a1, a1, -1
     # Call the multiply function
    jal multiply        # Recursive call mult(a, b-1)

    lw t0, 4(sp)        # Load the address of result into t0
    add a0, a0, t0      # result in a0 = a + mult(a, b-1)

    #Get ready to return from recursive call
    lw ra, 0(sp)        # Restore ra
    addi sp, sp, 8      # Pop stack
```

```
    jr ra
```

Watch the program run as its executing, monitor the `Register` DEBUG RAM. When the program finishes you should have 0x0000000F in the a0 register, assuming the default inputs of 5 and 3.

**WHAT TO HAND IN – For this final part of the lab hand in a screen shot of the Instruction DEBUG and the Register DEBUG RAM – something like below:**