Dylan Thornburg
Anushri Selva

SANTA CLARA UNIVERSITY	**ECEN 122** **Winter 2024**	**Hoeseok Yang**
Laboratory #6: Control transfer **For lab sections on Tuesday/Friday February 27/March 1, 2024**		

I. OBJECTIVES

Add support for control transfer instructions.

PROBLEM STATEMENT

Our simple CPU, as it stands now, can only execute instructions in a straight line. This limits our ability to write programs that can take different actions depending on a set of conditions. It also causes inefficiencies in how programs can be written; if they have a repetitive aspect, it would be more efficient to execute the same instructions over again rather than consuming memory space with repeats of the same set of instructions.

The answer to these inefficiencies is to incorporate support for fetching an instruction from a memory location other than the location that sequentially follows the location containing the currently executing instruction. If we think of a currently executing instruction as "controlling" what the CPU is doing, we can also think of changing the flow of execution as transferring control of the CPU to a different point in the program that is running. The decisions of if and when to transfer control, and the identification of where in the program flow to continue execution, are all done by the program itself via special instructions that we categorize as control transfer instructions (CTIs).

CTIs can be unconditional or conditional. A terminology convention we will use is that unconditional CTIs are referred to as "jumps" and conditional CTIs are referred to as "branches".

When a CTI causes a transfer of control, we use the term "target" to indicate the memory location of the next instruction that should execute. A CTI needs a way to specify the target. We know that the PC contains the memory address used to fetch an instruction. So, to transfer control, the PC needs to be loaded with the target address. There are two approaches to generating the target address: a PC-relative approach takes the current PC value and adds a signed value to it, whereas an absolute approach takes
a completely new value to load into the PC.

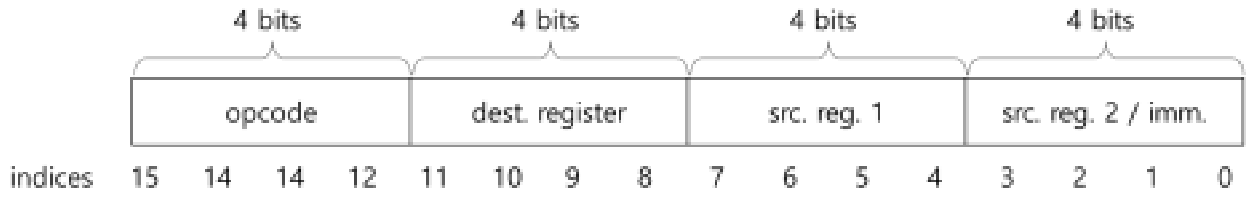
In this lab we will look at supporting just PC-relative conditional branches.

Branch instructions

Branch instructions come in many different forms, depending on the ISA. Some ISAs provide an assortment of comparisons, which give software programs a good bit of flexibility in exchange for more complicated hardware implementation. We are going to go in the opposite direction; we are going to implement the simplest solution we can, even if it means placing a burden on the software program to work around the limited functionality provided by the instruction set.

First, in terms of conditions that we will check, we will limit ourselves to the matching pair of branch if-zero (*bz*) and branch-if-not-zero (*bnz*). Executing these instructions will involve reading a source register and checking to see if the value is 0. Note that we really ought to have some form of comparison operations, but we are going to make do without.

For the branch target, we will use the 4-bit offset of our instruction format as a signed value to add to the PC+1 value of the executing branch instruction. That is, given the 16-bit instruction format below,

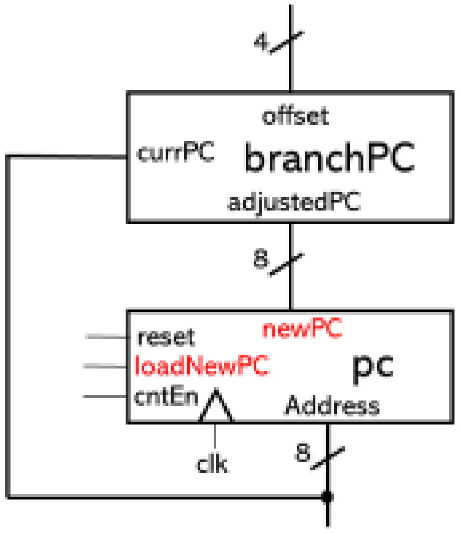


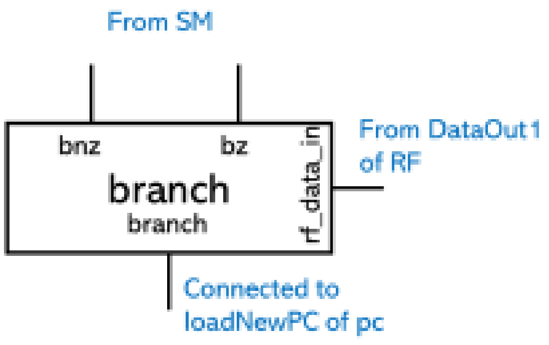
• *bz* sets the next PC value to (PC+1) + imm when the value src. reg. 1 holds is 0. • *bnz* sets the next PC value to (PC+1) + imm when the value src. reg. 1 holds is NOT 0.

Note that a 4-bit signed value only lets us move 7 instructions in either direction of the

branch. **<u>Description of the necessary changes to the datapath</u>**

Following two new modules will be added to the datapath





• A *branchPC* module, which will take the current PC and the offset from the instruction and output a new 8-bit PC value that is the adjusted PC using the offset.

• A *branch* module, which determines the branch will be taken or not. It takes two inputs from the control state machine (*bnz* and *bz*) and a 16-bit register data (*rf_data_in*). That is,

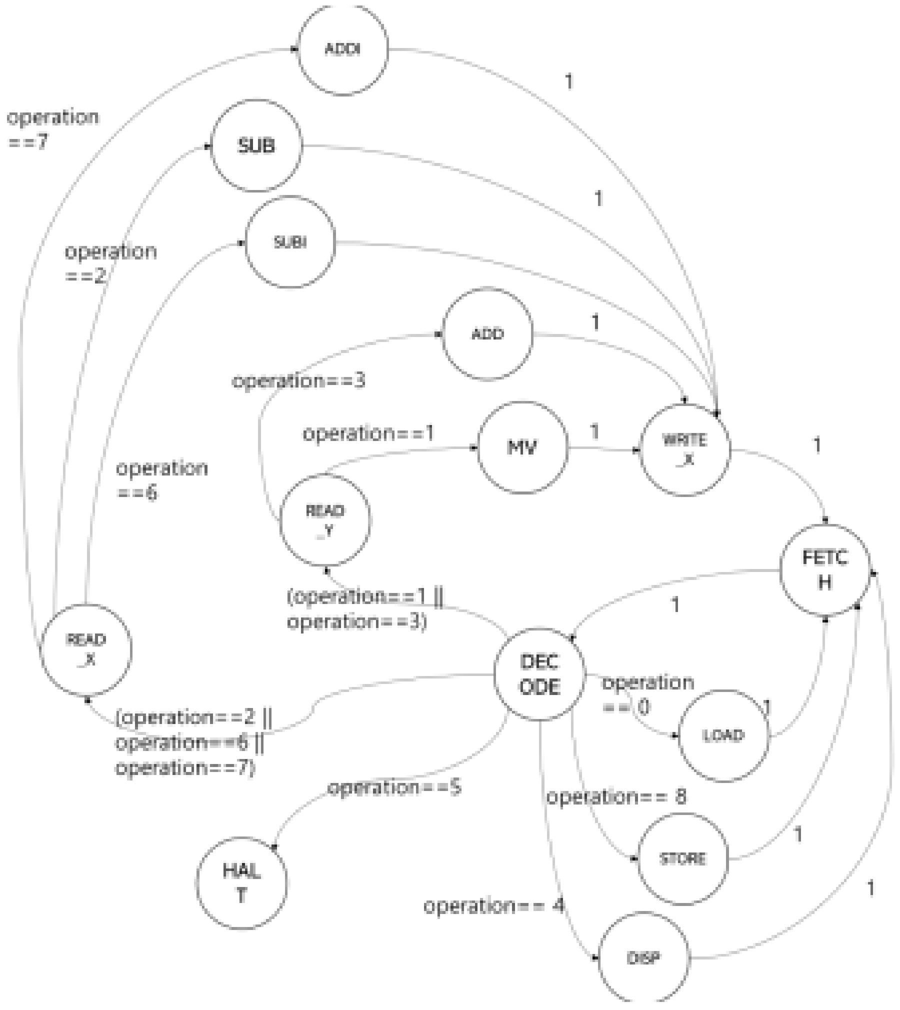
- *bnz*: the branch will take place (*branch*=1) if *rf_data_in* is not equal to zero.
- *bz*: the branch will take place (*branch*=1) if *rf_data_in* is equal to zero.

The PC module also need to be modified properly:

- An updated PC module, which will have two new inputs added to it: an 8-bit *newPC* port and an 1-bit *loadNewPC* port. When *loadNewPC*=1, *Address* is set to *newPC*.

Updates to the control state machine

We are going to reuse the state machine used in the previous lab, which is shown below:



Keep the basic structure of the state machine as it is and just <u>add two additional states</u> for the two new instructions (*bnz* and *bz*). Also, we need to add two 1-bit outputs (*bnz* and *bz*), which are connected to the *branch* module that is newly added to the datapath.

PRE-LAB

(i) Specify the opcodes you will use for the two new instructions, *bz* and *bnz*.

(ii) Draw a block diagram showing how the new hardware will need to be connected to correctly control the PC value. The diagram should include the two modules shown above (*branchPC* and

branch) and the PC module should also be properly modified. All modules should be clear about what signals are going into and them and what signals are coming out of them. Use the datapath diagram that we completed in Lab 5 (depicted on a separate page at the end of this material).

(iii)Draft a Verilog module for your branch control logic (the skeleton will be given). (iv)

Draw a new state diagram that accounts for the newly added instructions (*bnz* and *bz*).

(v) Write a program (mem.txt file) that implements the following higher-level pseudo-code.

```
          accum = 0;
          if (accum ==0)
            goto skip;
          accum = 10;
skip: address = 48;
         count = <value at address>;
          while(count !=0){
            address++;
          number = <value at address>
            accum += number;
            count--;}
          display accum;
```

Do not forget to put a halt instruction at the end of your program.

In the lab, the given memory wrapper file (l6_MEM.v) will be initializing memory with known values at address locations 48 and beyond, so that the results of the program can be checked in lab.

Prior to lab, create a short version of your program that just does the first few lines. Specifically

```
          accum = 0;
          if (accum == 0)
            goto skip;
          accum = 10;
skip: display accum;
```

You will use this short program for doing initial testing of your design.

II. LAB PROCEDURE

(i) After reconciling your pre-labs, split the work between you and partner to get all the necessary changes implemented.

(ii) First, focus on the short program and check if your CTI works correctly as

intended. (iii) Now work on getting the full program working.

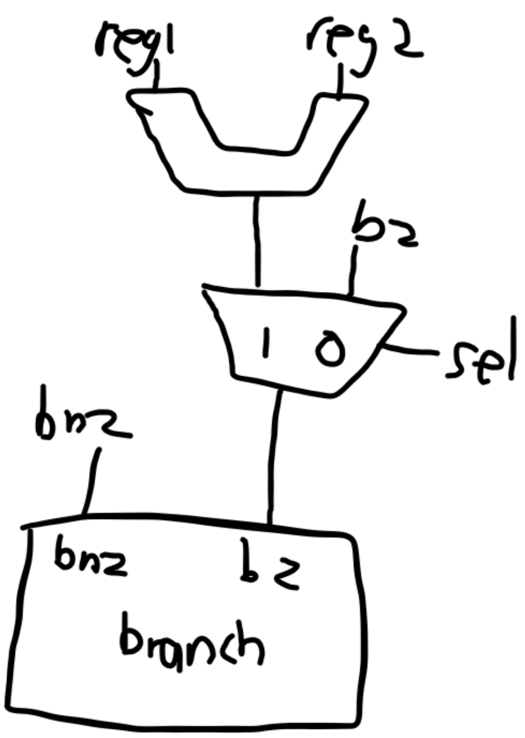
(iv)When you believe you have everything ready, start working to demonstrate that your program can be correctly executed.

(v) Once you have determined that your design appears to be working properly, demonstrate to the TA.

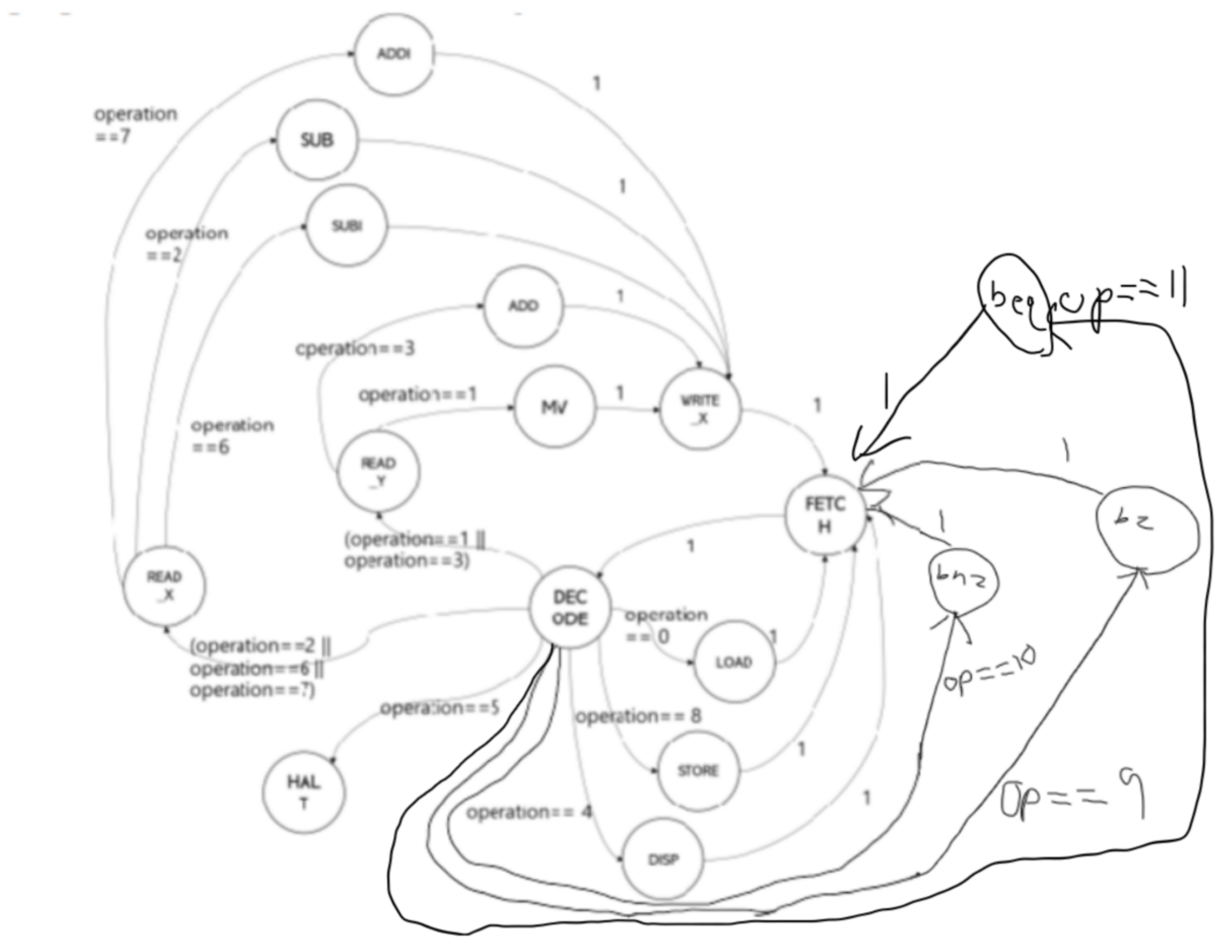
III. REPORT

For your lab report, include the source code for your working state machine and your working branch logic. In addition, include answers to the following questions.

1. **What problems did you encounter while implementing and testing your system?**
    - We initially set up the mem file incorrectly. We accidentally made it a design source instead of a simulation source. We also messed up sign extension but we fixed it quickly. The last problem (which wasn't a huge issue) was that we didn't make a true while loop that checks the condition at the beginning and instead made a do-while loop. Rikesh let this slide and didn't require us to add a "count" check instruction, but we still did.
2. **Did any problems arise when demonstrating for the TA? In other words, did the TA ask you to demonstrate something that you did not think of yourself? What was the scenario that you were asked to demonstrate? Provide some thoughts about why you didn't think of this yourself.**
    - Originally, our instructions had a do-while loop instead of the expected while loop. But we were able to change it, by checking the condition before running the while loop. If r3 (count) were ever to equal 0 initially, we would skip the five instructions, and go to Display and HALT.
3. **Pick another CTI operation that could have been implemented. Describe what changes to the datapath would have been necessary to support this operation, and how your state machine would need to be modified to correctly sequence the relevant control signals.**
    We would try to add Beq between two registers. We would add an ALU that takes two registers, subtracts them, and then use a mux to switch between Beq and Bz. If the output of ALU = 0 (which is what Bz tests for), it means r1=r2 and then it branches. For the state machine, we would make Beq come from decode with an opcode of 11 and go to fetch afterward. Because we aren't writing anything, we Beq can go straight from decode to fetch.



test if equal; new CTI is Beq



Code:

BranchPC:

```
module l6_branchPC(
        input [7:0] currPC,
        input [3:0] offset,
        output [7:0] adjustedPC
);

    /* TODO #1: calculate the target branch address */
    /* currPC + sign-extended offset */
        assign adjustedPC = currPC + { {4{offset[3]}}, offset};
endmodule
```

Branch.v:

```
module l6_branch(
        input bnz, /* asserted if the instruction is bnz */
```

```
              input bz,  /* asserted if the instruction is bz */
              input [15:0] rf_data_in,
              output reg branch  /* output: assert if the branch condition is fulfilled */
);

  /* TODO #2: complete the following */
  always@(*)
  begin
    if(bnz==1)  /* if the instruction is bnz? */
        if(rf_data_in==0) branch=0;
        else branch=1;
    else if (bz==1) /* if the instruction is bz */
        if(rf_data_in==0) branch=1;
        else branch=0;
    else /* if the instruction is not a branch-related one? */
        branch=0;

  end

endmodule
```

SM.v:

```
DECODE:
BNZ:
                      begin
                             _Extern = 1'b0;
                             Gout = 1'b0;
                             //Iout = 1'b1;
                             Ain = 1'b0;
                             Gin = 1'b0;
                             DPin = 1'b0;
                             RdX = 1'b0;
                             RdY = 1'b0;
                             WrX = 1'b0;
                             add_sub = 1'b0;
                             pc_en = 1'b0;
                             ILin = 1'b0;
                             rf_sel = 1'b0;
                                sw_sel = 1'b0;
                                MemWr = 1'b0;
                                AddrSel = 1'b0;
              /* TODO #3-3: output handling for bz and bnz */
                             bz = 1'b0;
                             bnz = 1'b1;
                          end
                      BZ:
                          begin
                             _Extern = 1'b0;
                             Gout = 1'b0;
                             //Iout = 1'b1;
                             Ain = 1'b0;
                             Gin = 1'b0;
                             DPin = 1'b0;
```

```
                    RdX = 1'b0;
                    RdY = 1'b0;
                    WrX = 1'b0;
                    add_sub = 1'b0;
                    pc_en = 1'b0;
                    ILin = 1'b0;
                    rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
            /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b1;
                    bnz = 1'b0;
                end

            if(operation == 4'b0000) state <= LOAD;
            else if(operation == 4'b0001) state <= READ_Y;
            else if(operation == 4'b0010) state <= READ_X;
            else if(operation == 4'b0011) state <= READ_Y;
            else if(operation == 4'b0100) state <= DISP;
            else if(operation == 4'b0101) state <= HALT;
            else if(operation == 4'b0110) state <= READ_X;
            else if(operation == 4'b0111) state <= READ_X;
            else if(operation == 4'b1000) state <= STORE;
            else if(operation == 4'b1001) state <= BZ;
            else if(operation == 4'b1010) state <= BNZ;
/* TODO #3-2: state transitions for bz and bnz */
            else state <= FETCH;


            BNZ:
                    state <= FETCH;
            BZ:
                    state <= FETCH;
```

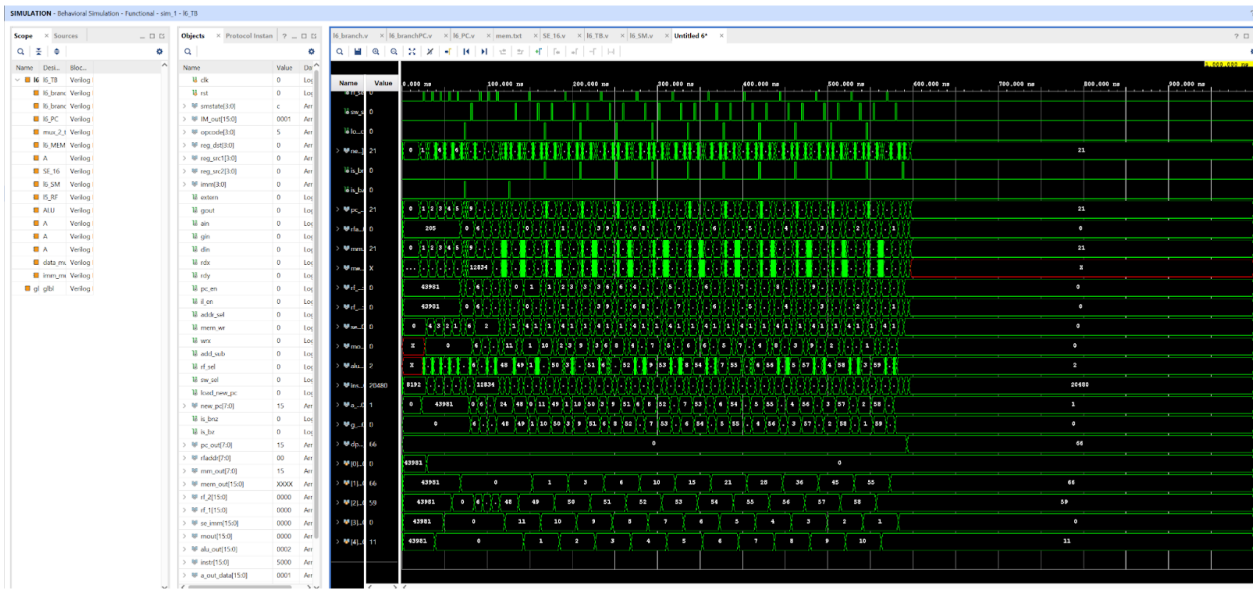
TB.v:

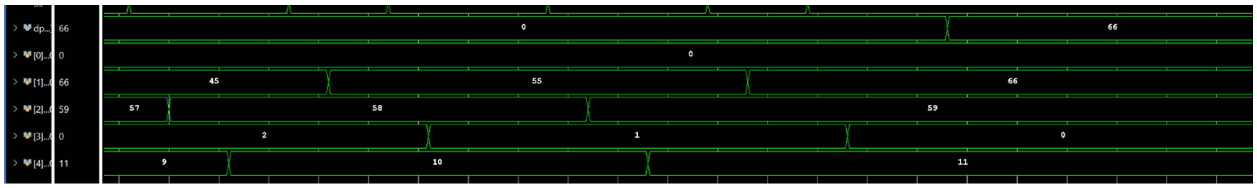
```
l6_branchPC branchPC(
            .currPC(pc_out),
            .offset(imm),
            .adjustedPC(new_pc)
);
   l6_branch branch(
   .bnz(is_bnz),
            .bz(is_bz),
            .rf_data_in(rf_1),
            .branch(load_new_pc)
            );
   l6_PC pc(
   .clk(clk),
            .countEn(pc_en),
            .reset(rst),
            .loadNewPC(load_new_pc),
            .newPC(new_pc),
            .Address(pc_out)
);
```

Mem.txt:

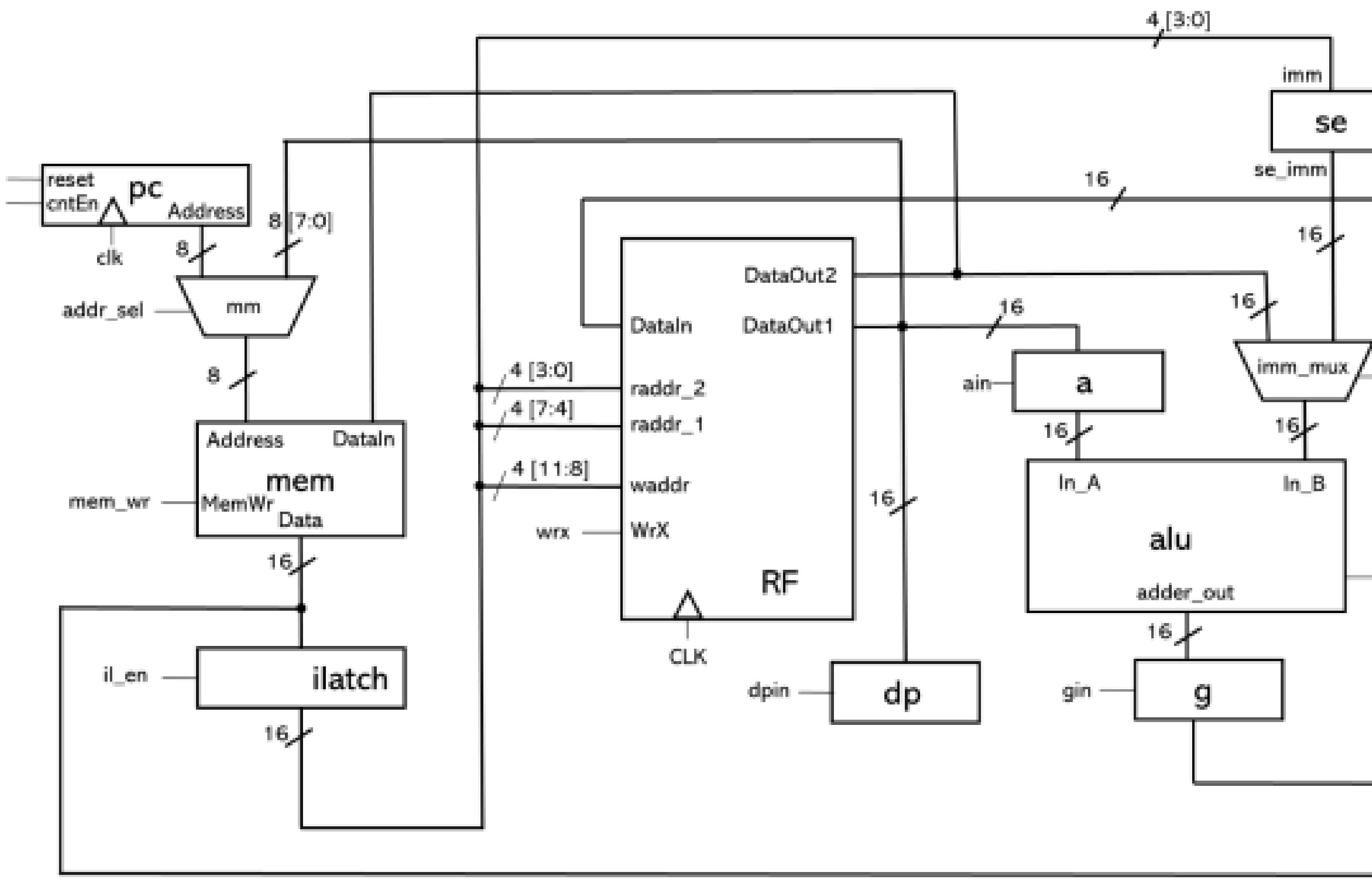
```
2000 // create r0=0 in case i need it
2444 // sub r4, r4,r4; r4= number
2333 // sub r3,r3,r3; r3=count
2222 // sub r2,r2,r2; r2 = address
2111 // sub r1, r1,r1; r1= accum = 0
9012 // branch to dis (skip two instruction) if r1 = 0
7115 // add 5 into r1
3111 // multiply r1*2 = 10
7226 // add 6 into r2
3222 // mult r2 by 2 (12)
3222 // mult r2 by 2 (24)
3222 // mult r2 by 2 (48)
0320 // count = <value at address>

//begin while loop

9035 // skip loop if r3 == 0
7221 // add 1 to r2 (address)
0420 // number = <value at address>
3114 // add r1, r1, r4; accum +=number
6331 // subi r3, r3, 1
A03b // bnz; branch up 4 if r3 !=0

//end while loop

4010 // displayr1/accum
5000 // HALT!
```

Simulation Picture:

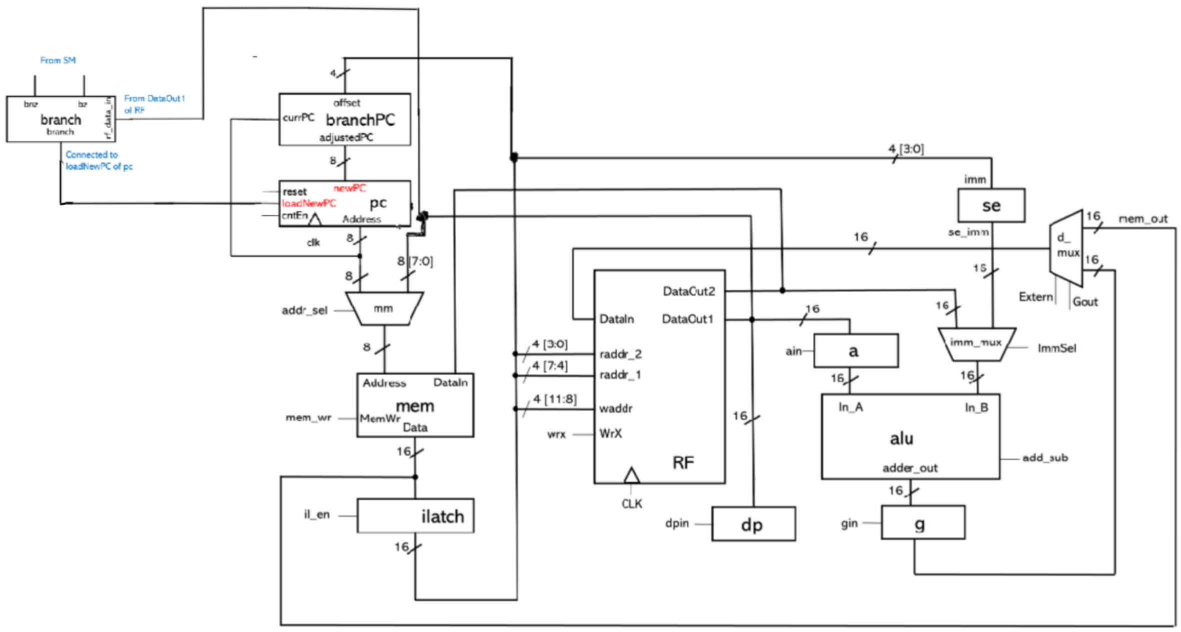

SIMULATION - Behavioral Simulation - Functional - sim_1 - I6_TB



[Datapath we implemented in Lab 5]



Modified data path:


From SM
bnz
bz
branch
branch
rf_data_in
From DataOut1 of RF
Connected to loadNewPC of pc
4
offset
currPC
branchPC
adjustedPC
8
reset
newPC
loadNewPC
pc
cntEn
Address
clk
8
8
8 [7:0]
addr_sel
mm
8
Address
DataIn
mem
mem_wr
MemWr
Data
16
il_en
ilatch
16
4 [3:0]
imm
se
se_imm
16
15
d_ mux
16
mem_out
16
Extern
Gout
DataOut2
DataIn
DataOut1
4 [3:0]
raddr_2
4 [7:4]
raddr_1
4 [11:8]
waddr
wrx
WrX
RF
CLK
16
dpin
dp
16
ain
a
16
imm_mux
ImmSel
16
In_A
In_B
alu
adder_out
add_sub
16
gin
g