SANTA CLARA UNIVERSITY	ELEN 122 Winter 2020	Jim Lewis

Laboratory #5: Control transfer

For lab section on Friday February 14, 2020

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 all of these characteristics.

Branch and jump instructions

Branch instructions come in many different forms, depending on the ISA. Some ISA's provide an assortment of comparisons, which give software programs a good bit of flexibility in exchange for more hardware implementation. We're going to go in the opposite direction; we're 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're 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. Note that a 4-bit signed value only lets us move 7 instructions in either direction of the branch. Moving farther than that will require the jump instruction.

Which brings us to the jump instruction. Our jump instruction will read a value from the register file, specified by the src1 field of the instruction, and load it into the PC, thus causing the next fetched instruction to come from the location specified by this new value. To ease the use of the jump instruction, we're going to add a companion instruction, called savePC. This instruction will just take the current PC value (after it has been incremented from the PC of the savePC instruction itself) and store it into a specified register. So anytime a program wants to establish a target for a jump instruction it should make sure there is a savePC instruction just prior to it in the instruction flow.

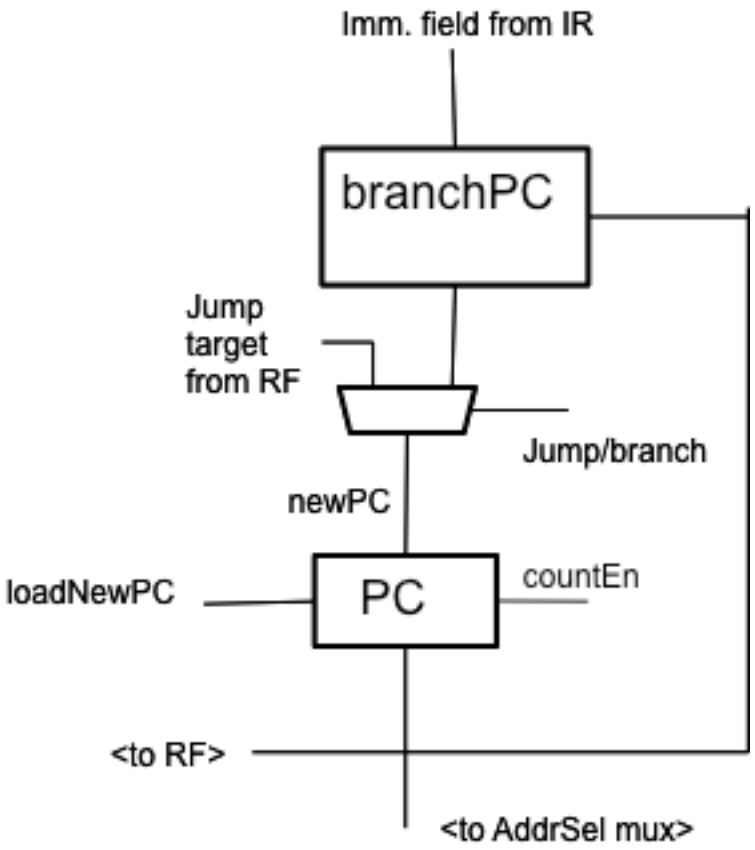
Note that this now allows us to overcome the limited range of our 4-bit PC-relative target for the branch instructions. If a program needs to establish a target for a conditional branch that is beyond the range, it should make sure the savePC instruction has been used to capture the target and then have the conditional branch transfer to a jump instruction that will then jump to the desired target.

Description of the necessary changes to the datapath

The TA will provide two new modules:

- An updated PC module, which will have two new inputs added to it: a 16-bit newPC port and a 1-bit loadNewPC port.
- A branchPC module, which will take the current PC and the offset from the instruction register and output a new 16-bit PC value that is the adjusted PC using the offset.

You will need to add a mux that allows for choosing between a register value (for jump instructions) and the adjusted PC (for branch instructions). The output of this mux will be the new PC value to load, when applicable. You should have something that looks like:



Another mux will be needed to allow for feeding the current PC value into the data input port of the register file.

Note that muxes have mux select signals that control their behavior. You will need to correctly generate and control these mux select signals as a function of what instruction is currently executing.

Controlling the loadNewPC signal will be a bit more involved. It should only assert if a CTI is executing. But if the CTI is a branch, then it should only assert if the branch condition turns out to be true. And that can't be determined just by decoding the instruction. So you will need to come up with your own branch logic module to generate the loadNewPC signal.

A simple assembler

Since our instruction format is very simple, it's pretty straightforward to have a script that creates the hex values for the instructions we want to execute. We just need to have a syntax for specifying our opcodes and our register values.

To make program generation a little easier, the TA will provide a simple assembler that we have developed. There is a format file that you will use to assign instruction names/acronyms to opcodes. Then you will use these acronyms, along with the stipulated format for specifying registers, to write a program in this simple assembly language. When you run the assembler, it will generate the

stream of hex values that you will then load into your instruction memory when you are ready to run your design.

PRE-LAB

(i) Specify the opcodes you will use for the bz, bnz, jump, and savePC instructions.

(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, at least: the PC, the newPC module, a mux to choose new PC values, and a module to represent your branch control logic. All modules should be clear about what signals are going into and them and what signals are coming out of them.

(iii) Draft a Verilog module for your branch control logic.

(iv) Explain the timing of the control signals relevant to the block diagram you created in (ii). Make sure it's clear: when you increment the PC, when you load the PC with a new value, and when you allow the IR to update.

(v) Create a syntax file for the simple assembler, which specifies acronyms for all of the opcodes you now support. (The TA will post a sample that demonstrates the syntax.)

(vi) Using the language you have now established, write a program that implements the following higher-level pseudo-code. (Yes it's a bit convoluted, but it's intended to make sure of all four of the new instructions):

```
            accum = 0
label1:     if (accum == 0)
                goto label2
            jump to label3
label2:     address = 48
            count = <value at address>
            while (count !=0)
                address++
                number = <value at address>
                accum += number
                count--
            jump to label1
label3:     display accum
```

A couple of points to note:

- The jumps to label1 and label3 are going to need correct usage of the savePC instruction.

- The jump to label3 is a bit odd in that it will rely on a savePC instruction that resides later in the instruction stream. But the algorithm is constructed in a way that this jump shouldn't execute until after the PC for label3 has been saved.
- You shouldn't need a savePC for label2, since this should be reachable with an immediate offset in a branch instruction. I just used a label to represent the target for the branch.

You might want to consider having the update of the display register inside of your loop so that you can see the value change as you're accumulating. And don't forget to put a halt instruction at the end of your program.

Again, the TA 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.

II. LAB PROCEDURE

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

(ii) When you believe you have everything ready to run on the board, download your design to the Nexys board, and start working to demonstrate that your program can be correctly executed.

(iii) 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.

- What problems did you encounter while implementing and testing your system?
- 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.
- If we had wanted to use an 8-bit value to calculate a new PC target for our branch operations, how could this have been done? Explain what datapath changes would be needed to support this.
- Pick another conditional branch 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.