SANTA CLARA UNIVERSITY	ELEN 122 Winter 2020	Jim Lewis

Laboratory #3: Fetch and execute instructions from memory

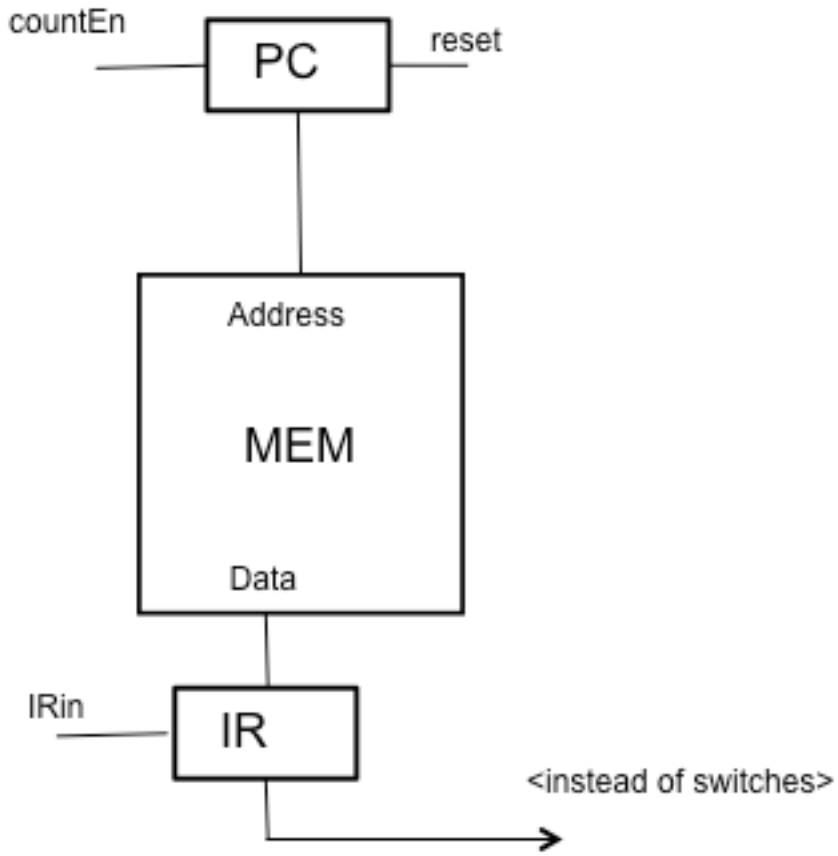
For lab section on Friday January 31, 2020

I. OBJECTIVES

Extend our CPU with a memory module that contains a sequence of instructions and demonstrate the execution of those instructions.

PROBLEM STATEMENT

Continuing our evolution of the CPU we have implemented over the previous labs, we are going to create a system that no longer uses switches for the inputs but rather fetches instructions from a memory module. You will be given the memory module but you will need to connect it to your CPU design in such a way that a) it fetches instructions at the appropriate time, and b) each fetched instruction correctly controls the remaining CPU hardware. You will also create the "program" that will be loaded into memory.



An update to our diagram

The diagram above represents what will take the place of the switches in the design we've been working with so far. The TA will provide the PC, MEM, and IR modules for you to instantiate; the Instruction Register (IR) is just another register with a load enable (in this case, IRin). But in this case it's going to be a 16-bit register rather than the 4-bit registers that are instantiated in other parts of the design.

The MEM module for this lab will not have any control signals. Whatever data is present at the location indexed by the Address input will show up on the Data output. As soon as Address changes, Data will change to the value at this new location.

Writing and executing a program

To write a "program", or a sequence of instructions to execute, you will create a simple text file where each line is an "instruction". An instruction, for our purposes, is just a binary representation of how you would be setting the switches if you were still controlling your CPU as you did in lab 2.

Your program is only going to be a handful of instructions, which will be fetched and executed one after another. Since the hardware support for fetching and executing instructions is going to be designed to continually fetch and execute, we'll need a way to tell it to stop. From lab 2, you have a 3-bit code that specifies what instruction to execute, but we only have 7 of the 8 encodings specified. To support the idea of causing a program to end, you will define your currently unused operation encoding to be a HALT instruction. When a HALT instruction is fetched, the "execution" of this instruction will cause the CPU to stop fetching further instructions and just sit until the system is reset.

You will put the sequence of instructions (your program) into a file called "instructions.mem". The MEM module provided by your TA will know how to load the instructions into the memory so that your specified list of instructions will be fetched as the PC starts counting up from 0.

A few points about the format of the instructions.mem file:

- Since each memory location is going to be 16 bits of information, yet our "instructions" only need 11 bits, we will need to fill in the missing 5 bits with 0's. So every line will have sixteen 1's and 0's.
- A given line can have arbitrary spaces between bits, which can help your brain keep track of what bits you're setting to what values. You can also put comments after the bit pattern, preceeded by "//". So you might have a line that looks like:

  00000 011 10 11 0010 // first 5 bits are 0, 011 is the opcode, AddrX=10, AddrY=11, imm=2

The TA can provide additional guidance if you have questions about formatting.

Updates to the state machine

Up to now, you have a state machine that is sitting in an Idle state until it sees the Go signal, and it has a Done state that it will stay in unless it sees that Go is no longer asserted. The states in between represent the execution of the current instruction. What we need to do here is amend this flow so that we cause an instruction to be loaded into the IR before we enter the states representing the execution of this instruction. We can repurpose our Idle state to do this. And we'll rename it the Fetch state.

Keep in mind that the contents of the IR are now going to be taking the place of the information that was coming from the pins. So, depending on how you have implemented your state machine up to now, you may actually need to insert another state, after the Fetch state, that allows for responding to the contents of the IR to determine how you traverse thru the execution portion of your state machine. If you need to add such another state, think of it as a Decode state.

When the execution of the current instruction is complete, we need to cause a new instruction to be loaded into the IR. In other words, we need to return to the Fetch state. But somewhere along the line between "fetching" one instruction and fetching the next instruction, we need to cause the PC to increment by having the countEn signal assert. And we need to make sure the count enable only asserts for one cycle, so we don't skip any instructions.

Note that we will still need the Done state at the end of each instruction execution, after which it will return to the Fetch state if the Go signal is asserted.

We also need to account for the "execution" of the HALT instruction, which should put the state machine into a state that does not nothing, and can only be exited by asserting a reset signal. In other words it will need to be a state other than the Done state because the Done state will go back to the Fetch state if the Go signal is asserted.

Manual controls

We will use two switches on the board: one for reset functionality and one to provide "single instruction" functionality.

- Reset: When the reset switch is set high, it will cause the reset signal to the PC to be asserted (causing it to go to 0), and it will cause your state machine to go to its initial state. When the reset switch is then moved low, causing the internal reset signal to deassert, your state machine will then be able to proceed with instruction execution.

- Single instruction mode: When this switch is low, the Go signal will just be tied high. This means that as soon as the execution of one instruction reaches the Done state it will proceed to the Fetch state in the very next cycle, this delivering continuous execution of the sequence of instructions in memory. But the switch is set high, the Go signal will be provided by a button. This means that, when an instruction reaches the Done state, the sequencing state machine will sit and wait until the user presses the button before fetching and executing the next instruction.

PRE-LAB

(i) Make sure that it's clear what encoding will be used to specify the new HALT instruction.

(ii) Draw out an updated state diagram that accounts for the introduction of the Fetch state and the execution of the HALT instruction. Make sure that it's clear when the 'countEn' and 'IRin' control signals will be asserted.

(iii) Write a "program" to execute in lab. The program will be a simple text file, called "instructions.txt", and each line will be sixteen 1's and 0's, to represent the encoding of an instruction. Your program should

   a. include at least one instance of each instruction type
   b. update (write to) every register at least once
   c. end with your HALT instruction.

(iv) Provide a description of what your program is doing, and what contents you expect to be in the four registers at the end of your program.

II. LAB PROCEDURE

(i) Once again, work with your lab partner to reconcile your pre-labs. In particular, decide which program you're going to execute, which will in turn dictate which instruction encodings you will need to work with. This will dictate how your instructions are decoded inside of your state machine implementation.

(ii) The person in your partnership who wrote the program that you chose in step (i) will be the Integrator for this lab and move to step (iv). The other person will be the Developer and perform step (iii).

(iii) The Developer will take the Integrator's state machine code from the previous lab and modify it to account for the agreed-upon changes to the state machine.

(iv) The Integrator will bring up Vivado and instantiate the new elements provided by the TA. Namely, the module for the PC, the MEM module, and a register to be used as the IR. The TA will also provide a module that will be generating the Go signal. All these modules will need to be connected, as appropriate, to the other entities in the system.

(v) Once the updated state machine code is ready, import it into Vivado, download to the Nexys board, and start working to demonstrate that your program can be correctly executed. Note that you will likely need to toggle the reset switch to get the program to execute.

(vi) If your program should not work properly, and running in single instruction mode isn't giving you enough insight into what is wrong, he TA will provide additional debug options.

(vii) 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. 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.
- What would happen if you didn't implement the HALT instruction? What would your system likely have done?