SANTA CLARA UNIVERSITY	ELEN 122 Winter 2020	Jim Lewis

Laboratory #4: Formatted instructions and support for stores

For lab section on Friday February 7, 2020

I. OBJECTIVES

Add structure to our instruction formatting, and add support for a store instruction.

PROBLEM STATEMENT

Up to now, the data on which our instructions have executed has been held solely in the register file. But register files tend to be fairly small, in terms of the number of registers, which means that we can only manage a small number of values if we don't have more locations to hold onto data. That's where memory comes in.

In the last lab, we added a memory module for holding instructions. In this lab we're going to add the support to use this memory to also hold onto data values as well. We will need to add a store instruction to write data to memory. And we will need to modify our existing load instruction to read data from memory rather than taking data from the instruction itself.

Because our memory module stores 16-bit values, we are also going to widen our datapath (the width of the registers and the width of the ALU) to 16 bits.

And, now that we're using 16-bit instructions, we're going to expand our register file to contain 16 registers. Since that will shake things up a bit relative to your previous formatting, we will take this opportunity to standardize on a new format that will be similar to what we've seen in class.

Specifying three registers, or two registers and an immediate value

With the elbow room that we now have with 16-bit instructions, we're going to do two things. First, we're going to expand our register file to contain 16 registers. That in turn means that we will need a 4-bit value to specify/address any given register. Also, we're going to move away from our previous instruction format, which only allowed us to specify two different registers and thus forced us to "overload" one of the addresses as both a source and a destination register, and now use three register identifiers: two source registers and a separate identifier for the destination of the instruction.

With three 4-bit fields as register identifiers, that consumes 12 bits out of our 16-bit instruction. The remaining 4 bits we will dedicate to specifying the opcode, the encoding that indicates what the instruction should actually be doing.

But some of our instructions still want to have immediate values (like addi/subi). Rather than implementing a different instruction format (R-format vs I-format), we're going to limit ourselves to a 4-bit immediate value, and just treat one of the fields as an immediate value for the instructions that use an immediate value.

That leads us to the following instruction format:

4-bit opcode	4-bit dest reg	4-bit src reg1	4-bit src reg2/imm

Note that this all implies some minor but important changes to our datapath:

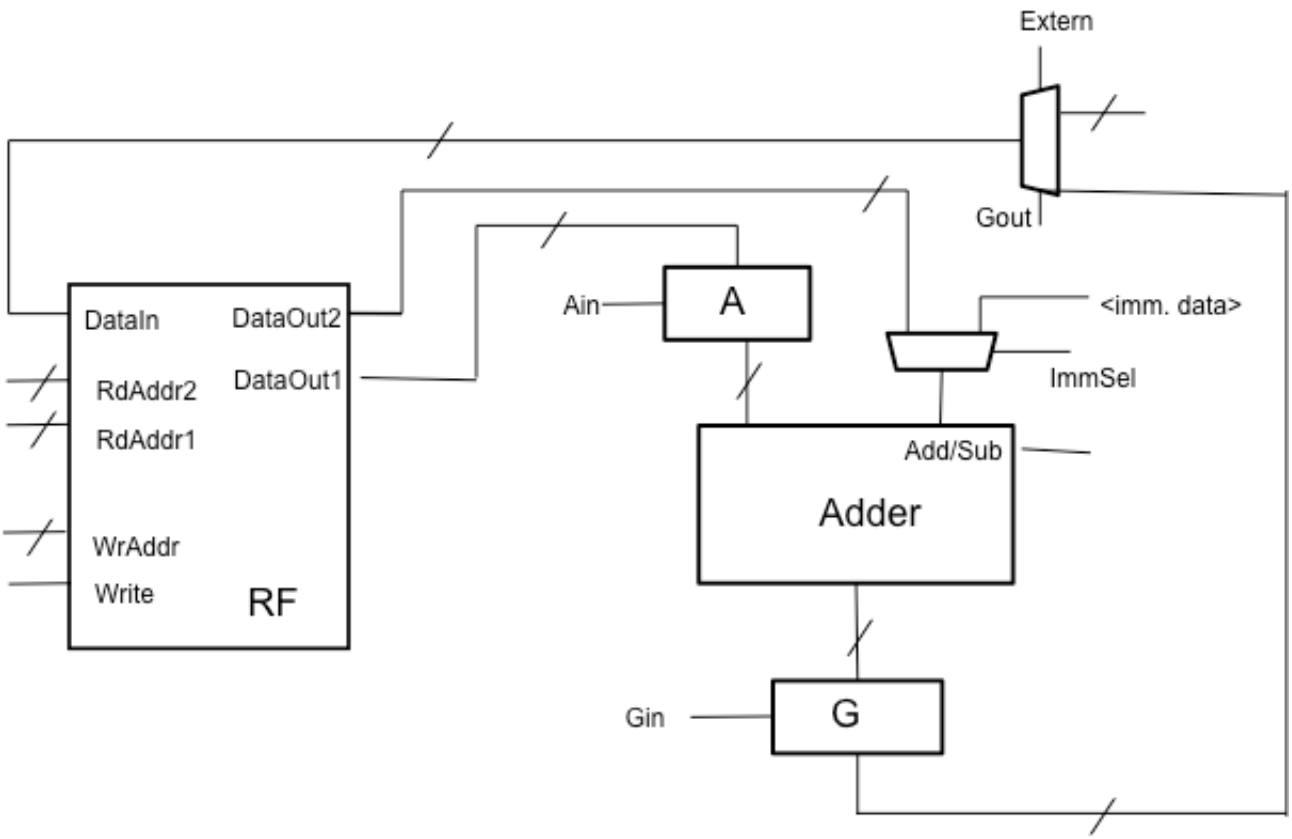
- The interface to the register file is going to change, by adding a third address port and a second read data port.
- Since we'll have two data read ports out of the register file, we'll need two separate buses connecting to the ALU.
- Since we're now dealing with 16-bit data values, the 4-bit "immediate" data from our instruction will need to sign extended into a 16-bit value before it is fed into the mux that feeds the second input port of the ALU.

Because we are introducing a register file with the ability to read our two source operands concurrently, out of two data ports, we no longer need the "read" control signals we had to apply to the register file. This new register file will just always read out whatever values are stored in the registers specified by the two read addresses.

Note also that we could get away with removing the staging registers, A and G. But that would then mean making substantial changes to the control state machine. We'll remove these vestiges in future labs, but we'll keep them in place for now to minimize the amount of work that needs to be done to get this lab working.

Also, we want to keep the display instruction that was previously implemented, and have it work the same as before. So the bus coming out of the RF's DataOut1 port should be fed to the display staging register, in addition to staging register A.

Given the changes described above, our datapath (minus the display staging register, and memory) is now going to look like this:



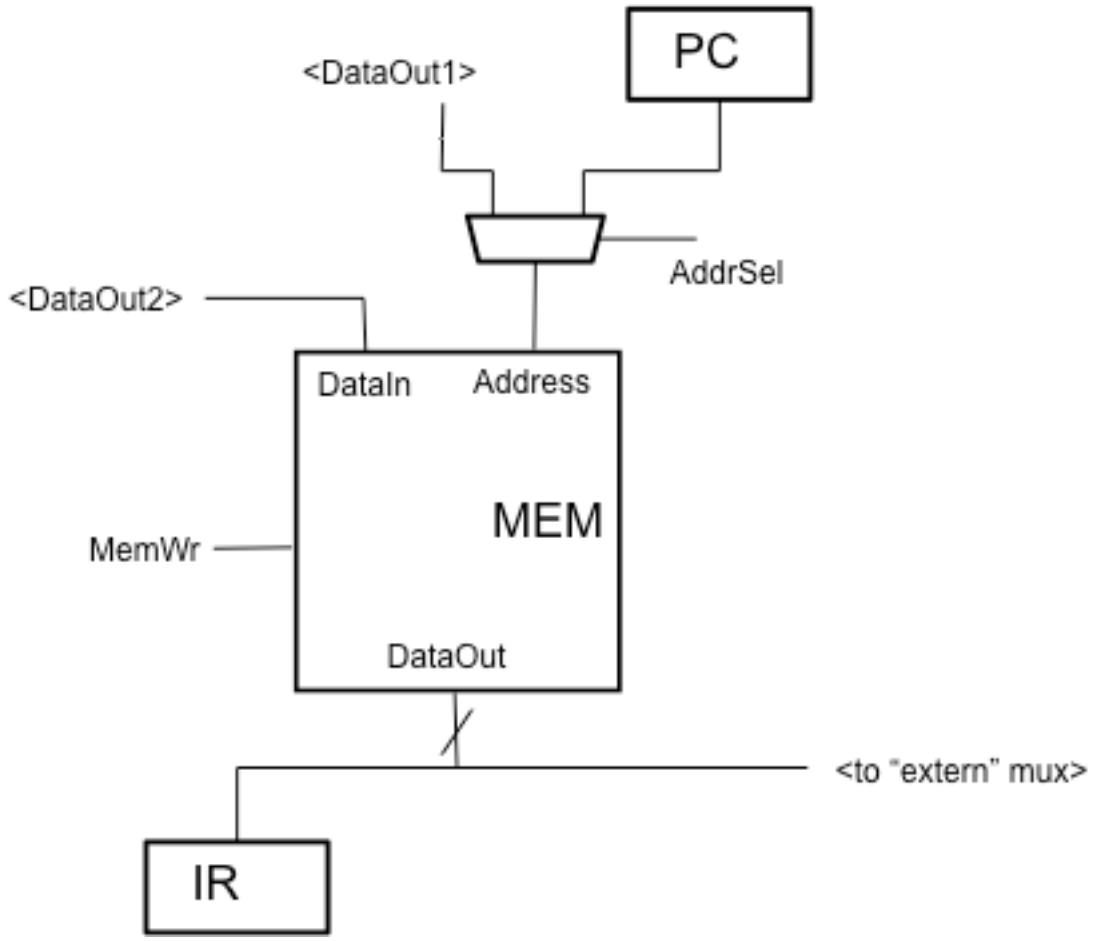
Adding a store instruction, and modifying the load instruction

Now let's turn our attention to what we need to support the use of memory for storing and loading data operands. For a store operation, we need to provide an address to identify the location in memory that we want to update, and we need to provide the data that we want to store at that location. Both of these values (the address and the data to store) will come from the register file. Specifically, the address will be supplied by source register 1 (i.e., the value we read out of port DataOut1) and the data to store will be supplied by source register 2.

The load operation is going to have some symmetry with the store operation: the address of the location from which we want to load will be supplied by source register 1. But now data will be going to the register file. So the data coming from source register 2 will be unused. The destination register specified by the load operation will be the register that is written with the data that is supplied by memory.

Note that we're using the same memory module that is holding our instructions. And for instruction fetches, the address supplied to the memory module is coming from the PC. So we have two different addresses (the PC for instructions, source register 1 for load/store operations), but only one address port into the memory module. So we will need a mux to select between these two addresses, and we will need to make sure we are selecting the right value to pass thru the mux depending on what we're trying to do.

All of the above description leads to a picture that looks like this:



Note the two control signals: AddrSel and MemWr. For the address mux, if you connect the PC to port 0, and source register 1 to port 1, then AddrSel should be asserted when performing either a load or a store operation. The MemWr signal should be asserted when performing a store operation.

Updates to the control state machine

It should be possible to correctly control this updated datapath with just a few changes to your state machine:

- Your opcode will be coming from slightly different bits from the Instruction Register (IR), but if you use the same instruction encodings then it should be fine.

- The RdX and RdY control signals have gone away, but you don't really have to prune them from your state machine, you can just leave them unconnected.
- The above change would impact the implementation of the move operation. But we no longer need the move operation. Given that our instructions can now specify separate source and destination registers, the move operation can be accomplished by using the addi (or subi) instruction and just setting the immediate field to 0. So you can completely remove the move operation.
- You'll need to define a new opcode encoding for the store operation, and then implement the correct transitions thru your state machine. Note that the store operation needs to assert both AddrSel and MemWr. Also note that you could reuse the operation encoding for move since it's going away, or you can allocate a new encoding.
- The load instruction will need to assert the AddrSel signal to make sure you load from the correct address.

Change to the writing of a program

For this lab, we will still be using a file called instructions.mem to specify and load a program into memory. But rather than specifying the instructions in binary, we will now specify them in hex. This works out reasonably well since the four fields in our instruction format are all 4-bit quantities, so they all align exactly with a hex digit.

PRE-LAB

(i) Identify the opcode you will be using for the store instruction; whether you are reusing the opcode previously used for move, or whether you are allocating a new encoding.

(ii) Draw out an updated state diagram that accounts for the store instruction, and any other relevant changes.

(iii) Write a program, to load and execute in lab, that accomplishes the following:
   a. Load the value at addresses 48, 49, and 50. We'll call these Val1, Val2, and Val3.
   b. Add Val1 and Val2 and store the result at the address indicated by Val3.
   c. Display the sum of Val1, Val2, and Val3.

   Make sure to end your program with a halt instruction.

   The TA will be initializing memory with known values at address locations 48, 49, and 50, so that the results of the program can be checked in lab.

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 and which state machine you will use.

(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 make all the necessary datapath changes described here.

(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.

(vi) 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.
- This design uses memory addresses directly from registers, rather than allowing for an offset to be added to a register value. How would you modify the datapath to allow for an offset, particularly for a store operation? How would this impact your state machine design?