ECE/CS 250: A Guide to Making and Debugging the Processor

BEFORE READING THIS GUIDE, PLEASE READ THE HOMEWORK SPECIFICATION IN ITS ENTIRETY SO YOU WILL HAVE CONTEXT FOR WHAT'S BEING DISCUSSED BELOW.

Before you start

While many people will find following along the slides to be the best approach (which is acceptable), you should understand that there are fundamental differences between the processor you're building in homework 4, and the one you've seen in lecture:

- **Word size:** The lecture slides (and the textbook) demonstrate the construction of a 32-bit processor. You're going to build a 16-bit processor in this homework.
- **Addressing:** The processor in the lecture slides (and the textbook) is a byte-addressed processor (just like in MIPS). The one you're building is a word-addressed processor. Make sure you understand exactly what this means (tldr: addresses are expressed in units of words rather than bytes), by reviewing relevant content. This fact has several implications:
    - **You won't have to increment the PC by 4 unlike in lecture slides.**
        - Instead, you'll increment the PC by 1 on the homework.
        - This is because PCs are always incremented by one word, but the processor in lecture was byte addressed, and 4 bytes = 1 word in the processor shown to you in lecture.
    - **You won't have to shift by 2 when jumping or branching to a specified address.**
        - In **byte-addressed** processors, you have a shifter that shifts the address to the left by 2 before jumping/branching because addresses are specified in **units of words** (no, this wasn't a typo) in the instructions' bitstrings. So, the shifter converts these words to bytes by multiplying by 4 (which is essentially the same as left-shifting by 2).
        - The processor in Homework 4 is word-addressed, so you can use the address that's specified in the bit-string as-is, without converting to bytes.
    - These are just examples of differences. Please make sure you're not blindly following the lecture slides and are aware of where you have to adjust things to satisfy this homework's requirements.

Make sure you're implementing each instruction incrementally (and testing it) before moving on to the next instruction. This means, if you build "add" first, for example, check that its datapath is completely correct before moving on to "addi" or "sub."

For now, treat your control signals (the ones that basically flip the switch on the multiplexers to select one of the many signals going into them) as if they're inputs, and have an input pin for all of your control signals. This allows you to test your instructions' datapaths individually, before you've built your controller. This approach is recommended, because once you've built all of your instructions and are confident that the datapaths are correct, building the controller will not introduce any errors, and any bugs you discover will be isolated to the controller only. For example, after you've built a datapath for the add instruction (and tested it), when you're building the datapath for addi, you'll have to introduce a mux at some point to select the ALU operand to be either the register (in add's case) or the immediate (in addi's case). For now, just treat this mux-selection signal as if it were inputted using a pin. **Later in the process (after all instructions' datapaths have been built), you will replace all these input pins by tunnels that come out of a controller.** The controller will take the opcode as its input, and compute values for all the control signals in your datapaths.

Many people prefer to combine the instruction-decoder (the box that takes in your 16-bit instruction and spits out the opcode, $rs, $rd, $rt, shamt, immediate, jump address and other fields) and controller (the box that takes in an opcode and spits out control signals) in the same subcircuit. For the sake of modularity and isolating errors to smaller surface areas, it is recommended that you **separate your instruction decoder and controller** into two different subcircuits and test them independently.

The instruction decoder shouldn't be any more complicated than a simple splitter that splits up the 16-bit instruction into 16 individual bits and recombines them (i.e. combines the most significant 4 bits into an opcode, the next 3 into $rs etc.) using many smaller splitters.

The controller will take in a 4-bit input (opcode). There are many ways to implement a controller, and the most obvious (and also the hardest to debug) approach would be: Write down a truth table that enumerates all 16 combinations of these 4 bits and what the control signals would be for each of those bits; then, write down logic functions for all of these control signals; optionally, simplify all logic; finally, use a splitter to split the 4-bit input into 4 individual bits, and implement the logic functions calculated in the previous steps, which will generate outputs which are your control signals.

While this is the most straightforward approach, an easier one might be as follows. Simply use your 4-bit input as the input to a 1-hot decoder, which will have 16 output wires (why?) out of which, only one will be turned on at any given time (corresponding to one of the 16 opcodes). Now, you'll put in as many output pins in your circuit as you have control signals. Lastly, for every control signal, you'll just have OR gates whose inputs are all the wires coming from your decoder that correspond to opcodes for which this control signal should be turned on, and the output of this OR gate would be that control signal. Essentially, this says "if the wire corresponding to any of these opcodes is on, set this control signal to be on, because that means we're dealing with one of those opcodes." This is the approach which was demonstrated in recitation.

At this point, you'll replace all your control signal input pins with actual control signals emitted from this subcircuit.

If you've followed all the steps correctly, and accurately set the settings for all components as required, you should have a functioning processor. If not, follow the debugging steps as outlined below.

Common issues

- **Before attempting any processor-debugging specific to Homework 4, make sure you have no Logisim-issues. Follow "A Guide to Debugging Logisim" on Sakai to eliminate any such errors before proceeding any further.**
- Be sure to explicitly enable your PC register. Not doing so can lead to non-deterministic behavior, which includes registers not being written to when running your processor using the autograder, but registers being written to when running your processor in Logisim.
- Double check that you've done everything listed under "Automated Testing" in the homework handout. If not, our autograder won't be able to interact with your circuit correctly even if it's functionally correct as determined by your own manual Logisim tests.
- You shouldn't use multiple clocks in a circuit. Even different subcircuits shouldn't have different clocks. Your main circuit should have a clock, and every clocked component in all your circuits/subcircuits should use the same clock to be in sync.
    - This is so important, that we're repeating this point in this guide even though it's mentioned in the Logisim Debugging guide mentioned in the first bullet.
- Make sure all your muxes have the "include enable" attribute set to "no" in the object's attributes.
    - This will reduce any potential non-determinism that could cause your circuit to fail the autograder tests.
- **If you have taken care of all these commons mistakes and bugs still persist, you need to debug your circuits by running them one clock tick at a time (this is analogous to stepping through programs line-by-line).**

Testing your CPU

Before you test your CPU as a whole, double-check that all of your instructions (individually) are all correct. To do so, manually poke in a 16-bit instruction into what would be your instruction-decoder's input, and take note of whether all the control signals are being generated as expected, and then as you tick through one clock cycle, note whether all state elements (registers, memory etc.) are updated as expected, and that all I/O is handled appropriately. **This step alone should resolve almost all bugs.**

At this point, you're ready to run programs on your CPU. To run any assembly program on your CPU, you'll need two files: an instruction memory file (which contains the raw bytes that correspond to instructions written in the ".text" section in the original code), and a data memory file (which contains raw bytes that correspond to data in the ".data" section in the original source code). You're given these files for all programs that come with the test kit and are used by the Python tester to check your CPU. All instruction memory files are named as "<test_name>.imem.lgsim" and all data memory files are named as "<test_name>.dmem.lgsim." Files whose names end in ".buffer" are only used in I/O tests, and contain text that will be inputted into the keyboard element in your circuit.

To load these instruction and data memory files into Logisim, right click on your instruction/data memory and click on "Load Image." Then select a memory file to be loaded into the ROM/RAM. Make sure that you only load instruction memory files into your instruction memory block, and data memory files into your data memory block.

When you load memory into your RAM/ROM, every row that's displayed corresponds to one word (16-bits) of memory and is displayed in hex. You should be able to convert these hex digits to instructions and back (you did this on homework 2). If you don't want to convert these by hand, you can always run your programs using the provided simulator (described in the next section), and the simulator will run your program instruction-by-instruction (using a correct CPU implementation you can compare against), and also show you to state of every register in every clock cycle. In addition, it'll also show you hex-representations of all instructions.

After loading your program into Logisim, click on Simulate > Ticks Enabled, to run your CPU. Alternatively, you can click on your clock input using the poke tool. Remember: two clicks = two ticks = 1 clock cycle. Now, all you have to do is observe the changes, and make sure that all registers contain expected values for that particular cycle, and that the TTY display outputs the correct values (for I/O tests). How do you figure out what the expected values in every clock cycle for all the registers are? Use the Simulator provided (see next section) for custom tests or simply look at the expected output file for provided tests.

The Assembler and Simulator

Read through the readme.txt in asm_sim for instructions on assembler and simulator usage.

The test kit comes with two important programs that will be useful when testing and debugging your processor after you've built it:

- **Assembler:** We provide a C++ based assembler that takes in an assembly file written in our variant of assembly, and produces the instruction memory ("<test_name>.imem.lgsim") and data memory ("<test_name>.dmem.lgsim") files that can be used for further debugging. These files can either be loaded into Logisim for testing as mentioned in the previous section. Alternatively, you can use these files with the Simulator to figure out what the expected values in registers are in each clock cycle.
- **Simulator:** This is a C++ based program that takes in an imem.lgsim file and a dmem.lgsim file (just like your Logisim CPU) and prints out every instruction on a separate line; one per clock cycle. It also shows you what the state of all the registers (i.e. what values are contained within every register in your register file) is on every clock cycle. You can use this simulator to see what you should expect to see in your register file in every clock cycle when you use Logisim to test your programs as described in the previous section.

To use the assembler and simulator, navigate to the `asm_sim` folder and run `make` to compile it. This should result in an `asm` and `sim` binary.

Writing Assembly Programs

Some things to note when writing assembly programs in the Duke 260/16 ISA:

- The first line of your program must be `.text` (do not start your program with a comment etc.).
- Save your assembly file as a `.s` file. While the Duke 260/16 ISA is not MIPS, MIPS syntax highlighting tools should work well with it since many instructions are similar.
- You can comment a line using "`#`".
- You can have a `.data` section just like in MIPS. For example:

```
.data
some_data:  .word      4       # Store 4 (one word of memory)
some_text:  .asciiz    "Hi!"   # Store "Hi!" (three words of memory)
```

- Because the processor is word addressed (and we don't have any load byte instruction), text in `.data` is stored one character per word instead of one character per byte.

Running the Assembler

Use the command `./asm PROGRAM_FILE` to assemble a file. For example, `./asm simple.s`. The `-v` flag will output some debug information, like the symbol table, when applicable (e.g. `./asm simple.s -v`).

Running the Simulator

Use the command `./sim PROGRAM_NAME.imem.lgsim PROGRAM_NAME.dmem.lgsim` to run the simulator. For example, `./sim example.imem.lgsim examples.dmem.lgsim`. You wil usually want to use `-v`, `-n`, `-d`, and `-r` flags (e.g. `./sim example.imem.lgsim examples.dmem.lgsim -v -n -d -r`) for most tests (but not all so read the flag descriptions below).

Simulator flags:

- `-v`: Print the dynamic instruction trace, along with the values of all registers at every clock cycle.
- `-n`: Print the number of dynamic instructions executed at the end of the simulation.
- `-d`: Print the register values in decimal rather than hex.
- `-r`: Add an extra cycle at the beginning for processor reset (i.e. for non-I/O tests).
- `-F`: Cause all input instructions to fail to return valid data. Useful to test invalid data case for the input instruction.

Note, the simulator outputs all values as unsigned (even though it can accept negative inputs and performs calculations on negative numbers). This is the same as Logisim. You'll need to convert an output to the expected negative number using 2's complement.

Sample Output

Below is a running of the assembler and simulator. You can see that apart from showing you the contents of all 8 registers in your register file in every clock tick, the simulator also shows you the raw 16-bit bitstring (represented in hex) for every instruction.

Before running the simulator or the assembler, be sure to compile both of those programs by running "make" in your terminal. Do so even if you can already see that you have the binaries associated with the two programs, because it's a good idea to recompile programs to account for OS differences.

From left to right the simulator outputs: Register values (0 to 7), the PC, the clock cycle, instruction in hex, instruction text.

```
ubuntu@cs250-az-00:~/hw4_test_kit/asm_sim $ make
g++ -g -o asm asm.cpp
g++ -g -o sim sim.cpp
ubuntu@cs250-az-00:~/hw4_test_kit/asm_sim $ ./asm simple.s
ubuntu@cs250-az-00:~/hw4_test_kit/asm_sim $ ./sim simple.imem.lgsim simple.dmem.lgsim -v -n -r
Extra cycle for initial reset of processor. All regs are 0.
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0000 1 2041 addi $r1,$r0,1
Regs: [0000 0001 0000 0000 0000 0000 0000 0000]  0001 2 2082 addi $r2,$r0,2
Regs: [0000 0001 0002 0000 0000 0000 0000 0000]  0002 3 20c3 addi $r3,$r0,3
Regs: [0000 0001 0002 0003 0000 0000 0000 0000]  0003 4 2104 addi $r4,$r0,4
Regs: [0000 0001 0002 0003 0004 0000 0000 0000]  0004 5 2145 addi $r5,$r0,5
Regs: [0000 0001 0002 0003 0004 0005 0000 0000]  0005 6 2186 addi $r6,$r0,6
Regs: [0000 0001 0002 0003 0004 0005 0006 0000]  0006 7 21c7 addi $r7,$r0,7
Regs: [0000 0001 0002 0003 0004 0005 0006 0007]  0007 8 c1ff bne $r0,$r7,-1
9 dynamic instructions executed
```