Learning Objectives

1. Understanding a simple computer datapath
2. Building an instruction decoder

Work that needs to be handed in (via SVN)

1. `arith_machine.v`: In this file, there are two modules that you need to implement:
   (a) `mips_decode`: This module takes in an instruction's **opcode** and **function** field and produces all of the control signals needed by the data path. You should use combinational design to do the implementation, much like we did for the keypad circuit.
   (b) `arith_machine`: This module implements the data path for the arithmetic machine. You implementation will largely be instantiating modules and wiring them together. There is little or no need for logical elements outside the various registers, memories, adders, muxes, and ALUs that we provide and the decoder that you are building above.

That's it as far as grading is concerned! (But, please continue to check in test benches like `arith_machine_tb.v`.)

We've provided a bunch of files for your use. None of these need to be checked in.

- `mips_defines.v`: defines of various MIPS related instruction fields.
- `rom.v`: holds the instruction memory.
- `alu32.v`: provides an ALU that you can use (but feel free to use yours).
- `mux_lib.v`: the same mux library from Lab4.
- `rf.v`: a 32 x 32b MIPS register file that you can use (but feel free to use yours).
- `test_alu.s` and `memory.text.dat`: an example MIPS input to test your code in both source and binary formats
- `Makefile`: for compiling and running with the provided input. Running with `make all` should recompile your verilog code if necessary and re-run it.

If you want to write your own tests (and we recommend that); here's how to do that.

1. Write a short assembly program (perhaps subsetting the provided `test_alu.s`.
2. On an EWS machines, run: `/class/cs232/Linux/bin/spim.vasm -vasm input.s outputname`
   where `input.s` and outputname are the names of your MIPS file and the desired output name, respectively.

This will create the following files:

```
outputname.data.dat
outputname.kdata.dat
outputname.ktext.dat
outputname.stack.dat
outputname.text.dat
```

Of these, you only care about `outputname.text.dat`. When you want that file to be loaded in your verilog simulation, rename it to `memory.text.dat` and run your verilog simulation.

Incremental Testing

The wrong way to do this assignment:

Step 1. Write all of the code
Step 2. Compile all of the code (and debug the compiler errors)
Step 3. Debug all of the code

One right way to do this assignment: (there are many)

Step 1. Implement the PC register, the PC+4 circuit, and the instruction memory. Hard-wire the exception output to zero, so the simulation doesn't end prematurely.
Step 2. Compile and debug this circuit. It should start at address 0, stepping by 4 each cycle. Check to make sure that the instructions printed out match those in `memory.text.dat`.
Step 3. Implement the MIPS instruction decoder; connect it to the output of the instruction memory.
Step 4. Compile and debug the full circuit. Make sure that for each instruction the right control signals are being generated **using gtkwave**. Refer to `test_alu.s` to know what the instructions are. *Alternatively, you could build a test bench to test the MIPS instruction decoder independently.*
Step 5. Implement the rest of the datapath.
Step 6. Compile and debug the complete design. Use the console output to validate whether your design is correct, but if it isn't use `gtkwave` to trace the signals at each step to see what you aren't getting right. **gtkwave** gives you a lot of visibility, so you should use it!

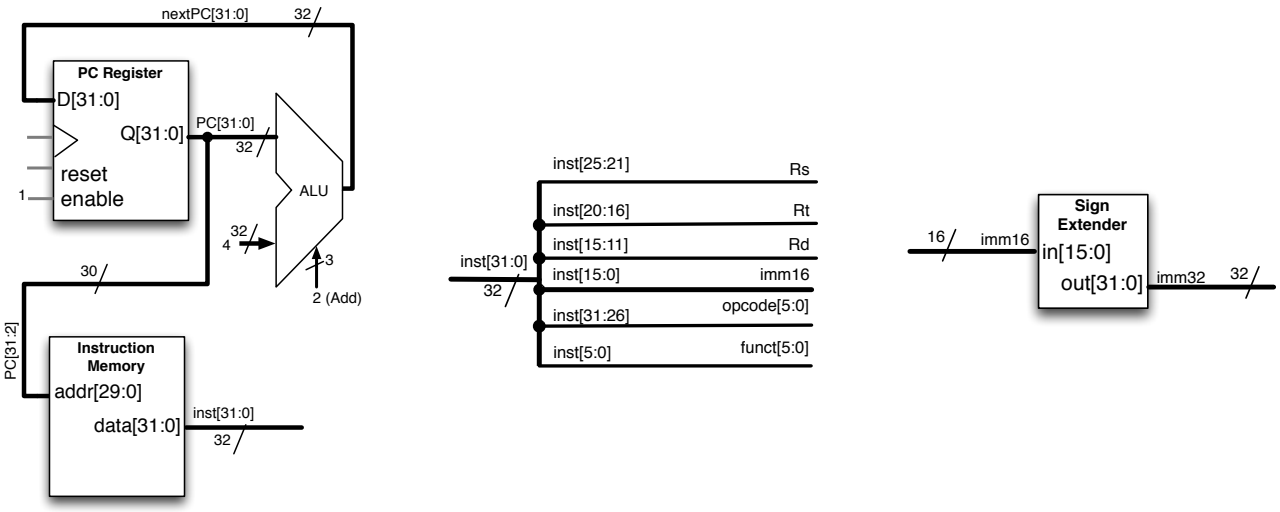
Debugging with gtkwave

When your design isn't doing what you think it should, the best way to figure out what is going wrong is with gtkwave. Here's how you should be using it (assuming you are already testing the smallest piece of the design that you haven't already validated).

1. Load your simulation output into gtkwave.
2. Plot all of the top level signals.
3. Find the first point in time that your circuit isn't doing what you think it should be doing. *If you debug a later point, some of the inputs might be wonky.*
4. Find the smallest circuit in the design where the inputs to the circuit are correct but the outputs are wrong.
5. If that circuit is a module, display the internal signals of that module and repeat step 4.
6. If that circuit is just logic, find the bug!

Wires, Buses, and Selection

We have a number of parts of the design where we are creating and selecting from groups of wires (called buses). For example:



Useful syntax for working with buses:

1. Declaring a bus:

   `wire [17:2] busname;` *// defines a 16-bit wide bus with indexes from 2 to 17, inclusive.*

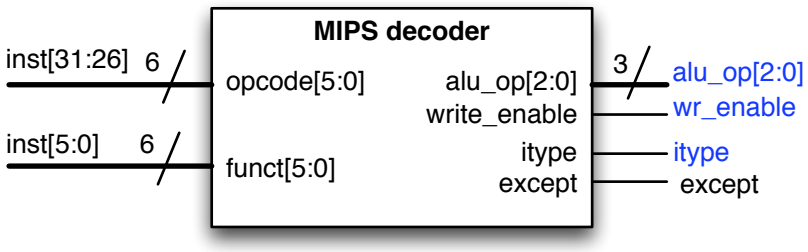
2. Attaching individual wires between buses:

   `assign x[15] = y[17];` *// connects wire 15 of bus* `x` *to wire 17 of bus* `y`*.*

3. Extracting some wires from a bus:

   `assign foo[4:0] = bar[15:11];` *// connects wires 0-4 of bus* `foo` *to wires 11-15 of bus* `bar`*.*

Decoder



The `exception` signal should be 1 when the opcode/func field pair is not recognized.

instruction	opcode	func	alu_op			itype	wr_enable
add							
sub							
and							
or							
xor							
nor							
addi							
andi							
ori							
xori							

assign itype =

assign alu_op[0] =