Learning Objectives

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

Work that needs to be handed in (via SVN)

1. `decoder.v`: This file contains the module `mips_decode`, which takes an instruction's `opcode` and `funct` fields and produces all of the control signals needed by the data path. You should use combinational design to do the implementation. **This is due by the first deadline.**
2. `decoder_tb.v`: A test bench for your decoder. As before, we can't autograde this, but we can use it to determine partial credit.
3. `arith_machine.v`: This file contains the module `arith_machine`, which implements the data path for the arithmetic machine. Your 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. **This is due by the second deadline.**
4. `test_alu.s`: A MIPS program to test your arithmetic machine – more details in the next section. Not autograded (see `decoder_tb.v`). This file contains comments indicating the value each register should contain after execution.

That's it as far as grading is concerned! (`arith_machine_tb.v` does not need to be modified – more on that below – and therefore does not need to be committed.)

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

- `mips_defines.v`: definition of various MIPS related instruction fields.
- `rom.v`: holds the instruction memory.
- `alu32.v`: provides an ALU.
- `mux_lib.v`: the same mux library from Lab4.
- `rf.v`: a 32 x 32b MIPS register file.

Compiling, Running and Testing

We've provided a Makefile you can use for compiling and running your code. The main rules of interest are `make decoder` and `make arith_machine`, and their names should be self-explanatory.

The decoder is tested traditionally; `decoder_tb.v` contains a small set of test cases, which you need to extend. On the other hand, `arith_machine_tb.v` is already complete; it runs all the instructions in `memory.text.dat`, which is compiled from `test_alu.s`, and prints out the subsequent register values. If you want to add more tests for your arithmetic machine, which we highly recommend, here's how to do it:

1. Modify `test_alu.s` to your heart's content.
2. The next time you run `make arith_machine`, it will automatically attempt to recompile `test_alu.s` using `spim-vasm`. You can install `spim-vasm` on your own machine following the instructions on the wiki:
   `https://wiki.cites.illinois.edu/wiki/display/cs233sp18/CS+233+on+Your+Own+Machine`

Incremental Testing

The ***wrong*** way to do this assignment, or any:

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 and test your decoder. Make sure the right control signals are generated for all valid instructions, and invalid instructions are handled correctly (that is, `except` should be 1 on an invalid instruction, and writing should not be enabled).
Step 2. Wire up the PC register, the PC+4 circuit, and the instruction memory. Hard-wire the `except` output to 0, so the simulation doesn't end prematurely. Notice that you do not need to connect all the output ports to a signal. For example, in the ALU that performs the operation PC+4, since we do not need the values of the zero, negative and overflow flags we can leave those ports unconnected by simply using spaces, as shown below:
`alu32 pcplus4(nextPC, , , , PC, 32'h4, `ALU_ADD);`
Step 3. 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 4. Connect the MIPS decoder to the output of the instruction memory. At this point, since the the MIPS instruction decoder provides the value for the `except` output, you should connect it appropriately and remove the hard-wiring you did in Step 2.
Step 5. Compile and debug the full circuit.
Step 6. Implement the rest of the datapath.
Step 7. 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

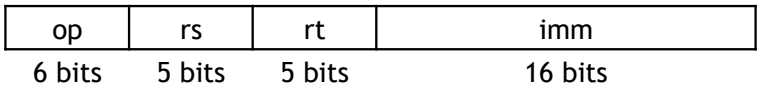
When your design isn't doing what you think it should, the best way to figure out what is going wrong is with gtkwave. Here's how you should be using it (assuming you are already testing the smallest piece of the design that you haven't already validated).

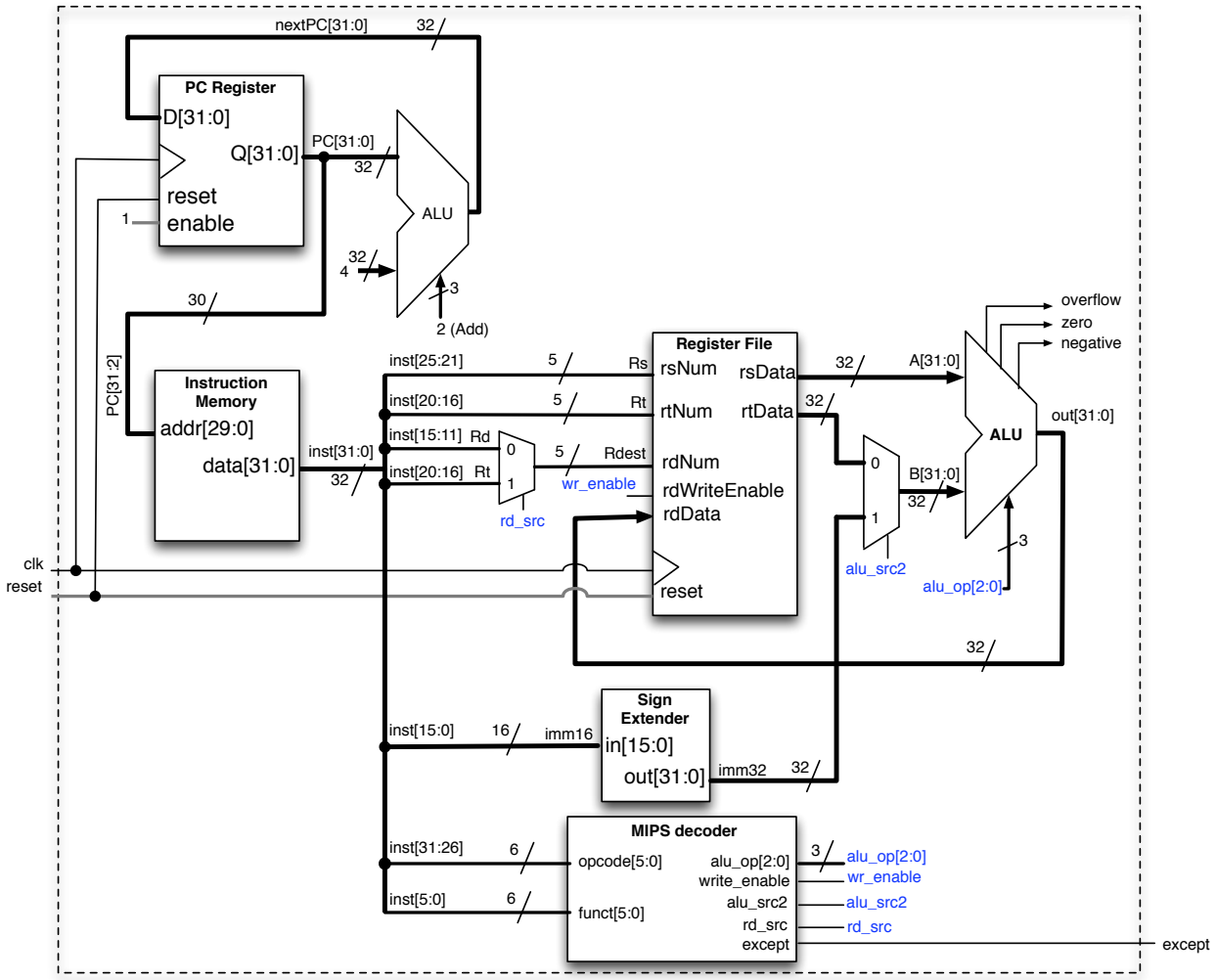
1. Load your simulation output into gtkwave.
2. Plot all of the top level signals.
3. Find the first point in time that your circuit isn't doing what you think it should be doing. *If you debug a later point, some of the inputs might be wonky.*
4. Find the smallest circuit in the design where the inputs 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!

Arithmetic machine circuit

R-type format:

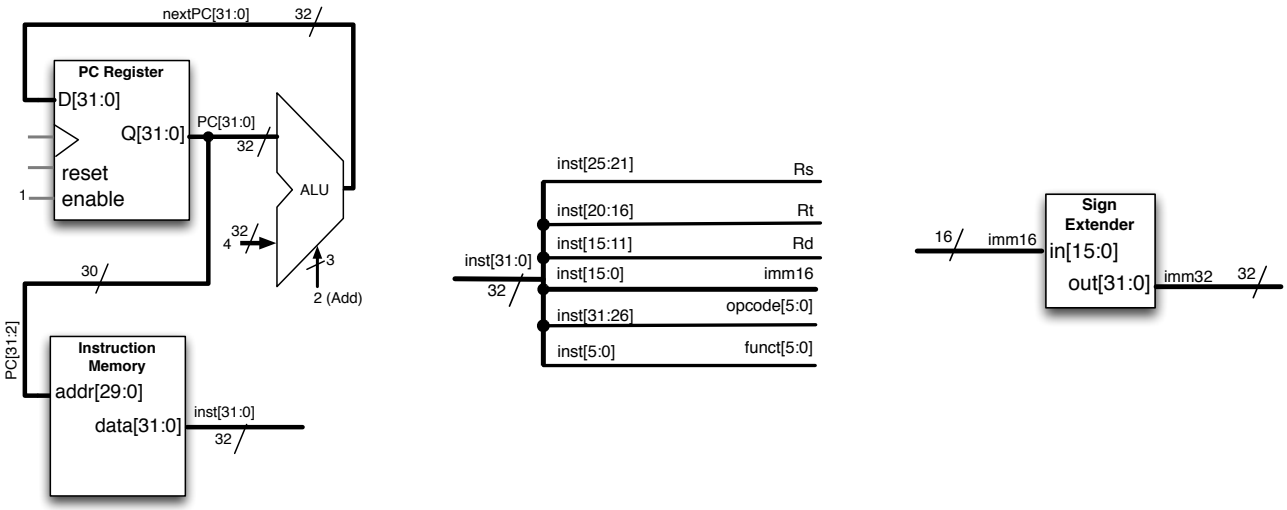
I-type format:




nextPC[31:0]
32
PC Register
D[31:0]
Q[31:0]
reset
enable
1
PC[31:0]
32
ALU
32
4
3
2 (Add)
30
PC[31:2]
Instruction Memory
addr[29:0]
data[31:0]
inst[31:0]
32
inst[25:21]
5
Rs
inst[20:16]
5
Rt
inst[15:11]
Rd
inst[20:16]
Rt
0
1
rd_src
5
Rdest
wr_enable
Register File
rsNum
rsData
rtNum
rtData
rdNum
rdWriteEnable
rdData
reset
32
A[31:0]
32
0
1
alu_src2
B[31:0]
32
ALU
overflow
zero
negative
out[31:0]
3
alu_op[2:0]
clk
reset
32
inst[15:0]
16
imm16
Sign Extender
in[15:0]
out[31:0]
imm32
32
inst[31:26]
6
inst[5:0]
6
MIPS decoder
opcode[5:0]
funct[5:0]
alu_op[2:0]
write_enable
alu_src2
rd_src
except
3
alu_op[2:0]
wr_enable
alu_src2
rd_src
except

Useful Verilog Syntax

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.*
4. Replicating some wires:
   `assign foo[2:0] = {3{bus[4]}};` *// replicates bus[4] 3 times.*
5. Concatenating some wires:
   `assign foo[1:0] = {bus[3],inst[4]};` *// concatenates bus[3] and inst[4].*
6. Checking opcode and funct:
   We have provided a file `mips_defines.v` with the values for the opcode and function code of all the instructions you will need. Below is an example of how you can use the values defined in that file:
   `opcode == `OP_ADDI` *// a 1 bit signal with value 1 if* `opcode` *matches the opcode of* `addi`.
7. Declare and assign in one go:
   (see next page)

For the decoder, you'll have to create a lot of internal wires which will then be `assign`ed to later. Instead of declaring and assigning separately, you can do it in one go, which is slightly more convenient. In other words, you can use the version on the right instead of the version on the left.

```
// don't do this
wire blah1, blah2, blah3;

// and then later
assign blah1 = ...;
assign blah2 = ...;
assign blah3 = ...;
```

```
// do this instead
wire blah1 = ...;
wire blah2 = ...;
wire blah3 = ...;
```