Learning Objectives

1. Understanding the implementation of branches, loads, and stores in a processor datapath.
2. Building a full machine capable of running MIPS programs containing a variety of instructions.

Work that needs to be handed in on GitHub

By the first deadline, 9/24 @ 8pm

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, much like the decoder in Lab 4. Your decoder needs to support **all the instructions from Lab 4**, plus the following instructions:

   ```
   bne, beq, j, jr, lui, slt, lw, lbu, sw, sb, addm
   ```

   We won't autograde your decoder output for `addm`, since different implementations of it can set the control signals differently.

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.

By the second deadline, 9/27 @ 8pm

1. `full_machine.v`: This file contains the module `full_machine`, which implements the data path for **all instructions from Lab 4**, plus the following instructions:

   ```
   bne, beq, j, jr, lui, slt, lw, lbu, sw, sb, addm
   ```

   See the final page for details about `addm`.
   Like Lab 4, we've provided all of the major components, so there will be little logic outside of wiring up these provided components and your MIPS decoder. There is significant overlap between this assignment and Lab 4, so you should **reuse portions of your Lab 4 Verilog in this Lab.**

2. `fm.s`: Additional tests for your full machine; the next section has the details. Like `decoder_tb.v`, this will not be autograded.

The files that we provide for Lab 5 include many of those we provided from Lab 4, with the following notable additions/extension. None of these need to be checked in.

- `rom.v`: has a data memory access in addition to the instruction memory from Lab 4.
- `all.s` and `lwbr.s`: two example MIPS assembly programs for testing your code
- `all.text.dat` and `lwbr.text.dat`: the machine language that corresponds to the two provided assembly programs.
- `all.data.dat` and `lwbr.data.dat`: the data memory images that correspond to the two provided assembly programs.

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 full_machine`, for compiling your decoder and full machine modules respectively.

Before you can run the full machine, **you need to set up your own test case**. We've provided two test cases in `lwbr.s` and `all.s`, which you can use by running `make lwbr_test` and `make all_test`, respectively. The assembly files contain comments specifying their expected behavior, which you can use to verify your implementation's correctness. `lwbr.s` is simpler, so we recommend getting that working first and then progressing to `all.s`.

Our provided tests aren't comprehensive, and in particular, they don't test most of the instructions carried over from last week, so it's a good idea to write your own. To do so, add your test cases to `fm.s` and then run `make fm_test` to assemble this file. The program needed for this compilation is already present on EWS machines. You can download it to your own machine by following the instructions on the wiki to download `spim-vasm`:

`https://cs233.github.io/onyourownmachine.html`

The full machine test bench will stop after 30 instructions by default. If you need to run more instructions, you should edit it accordingly.

Optional details, for those interested

Our simulated memories (instruction and data) get their initial values from loading files. Specifically, the instruction memory loads a file called `memory.text.dat` and the data memory gets its value from loading `memory.data.dat`.

Since we've provided two test cases, and neither of them are named `memory.text.dat` and `memory.data.dat`, you'll need copy or symbolic link one of the set of files. Specifically, we'd recommend that you use symbolic links. Symbolic links provide an alternate name for a given file. For example:

```
ln -s lwbr.text.dat memory.text.dat
ln -s lwbr.data.dat memory.data.dat
```

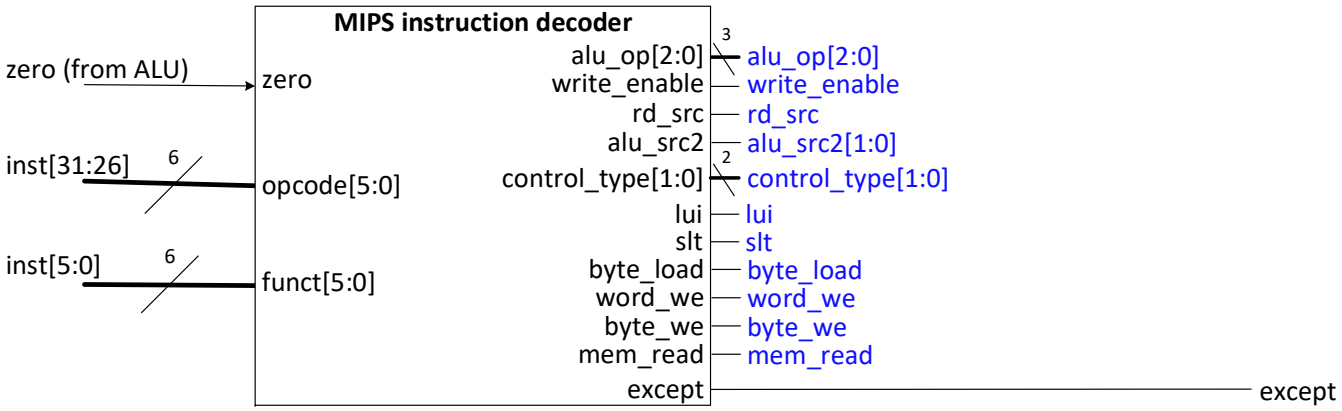
By typing `ls -l memory*`, we can see that now the `memory.*` files refer to their `lwbr` counterparts.

```
linux_machine$ ls -l memory*
lrwxr-xr-x  1 zilles  staff  14 Sep 29 13:00 memory.data.dat -> lwbr.data.dat
lrwxr-xr-x  1 zilles  staff  14 Sep 29 13:00 memory.text.dat -> lwbr.text.dat
```

We can switch to the `all.*` equivalents by removing the `memory.*` files and making new symbolic links.

```
rm memory.text.dat memory.data.dat
ln -s all.text.dat memory.text.dat
ln -s all.data.dat memory.data.dat
```

Decoder



The `exception` signal should be 1 when the opcode/func field pair is not recognized.
See the last page for details on addm.

inst	alu_op			rd_src	wr_ena	alu_src2		ctrl_type		mem_rd	word_we	byte_we	byte_ld	lui	slt
beq															
bne															
j															
jr															
lui															
slt															
lw															
lbu															
sw															
sb															
addm															

```
// mips_decode: a decoder for MIPS arithmetic instructions
//
// alu_op      (output) - control signal to be sent to the ALU
// writeenable (output) - should a new value be captured by the register file
// rd_src      (output) - should the destination register be rd (0) or rt (1)
// alu_src2    (output) - should the 2nd ALU source be a register (0) or an immediate (1)
// except      (output) - set to 1 when the opcode/func filed pair is unrecognized
// control_type[1:0] (output) - 00 = fallthrough, 01 = branch_target (taken),
//                              10 = jump_target, 11 = jump_register
// mem_read    (output) - the register value written is coming from the memory
// word_we     (output) - we're writing a word's worth of data
// byte_we     (output) - we're only writing a byte's worth of data
// byte_load   (output) - we're doing a byte load
// lui         (output) - the instruction is a lui
// slt         (output) - the instruction is an slt
// opcode       (input) - the opcode field from the instruction
// funct        (input) - the function field from the instruction
// zero         (input) - from the ALU
```

Full Machine

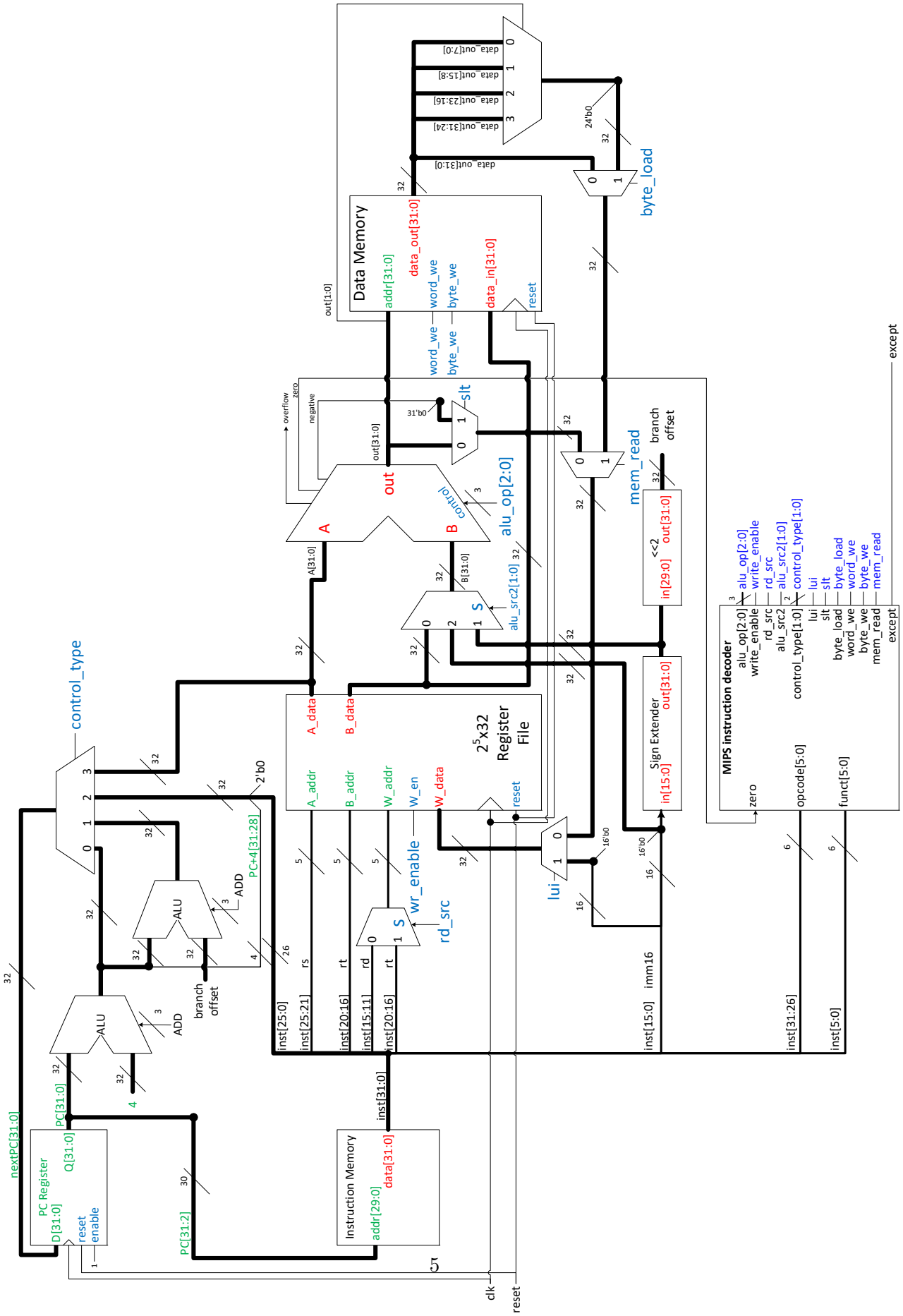
With the Decoder finished, you should be able to connect everything together to finish your MIPS datapath! As before, we have provided the implementations of many modules within the rf.v and rom.v files distributed with the lab.

In addition to the datapath for the arithmetic machine connected by your previous arith_machine.v, you will need to connect the modules representing your data memory, also including the muxes (with control bits) to choose between the outputs properly.

A few things to note:

1. We have shorthanded the Zero Extension module as 16'b0 in the following diagram. It's still there though!
2. Every label in the diagram is important! As a suggestion, look through each module individually and ensure the correct control bits/inputs/outputs are being passed in.
3. If you're unsure of what wires need to be passed into a given module, look through mux_lib.v, rom.v, or rf.v for their verilog headers.

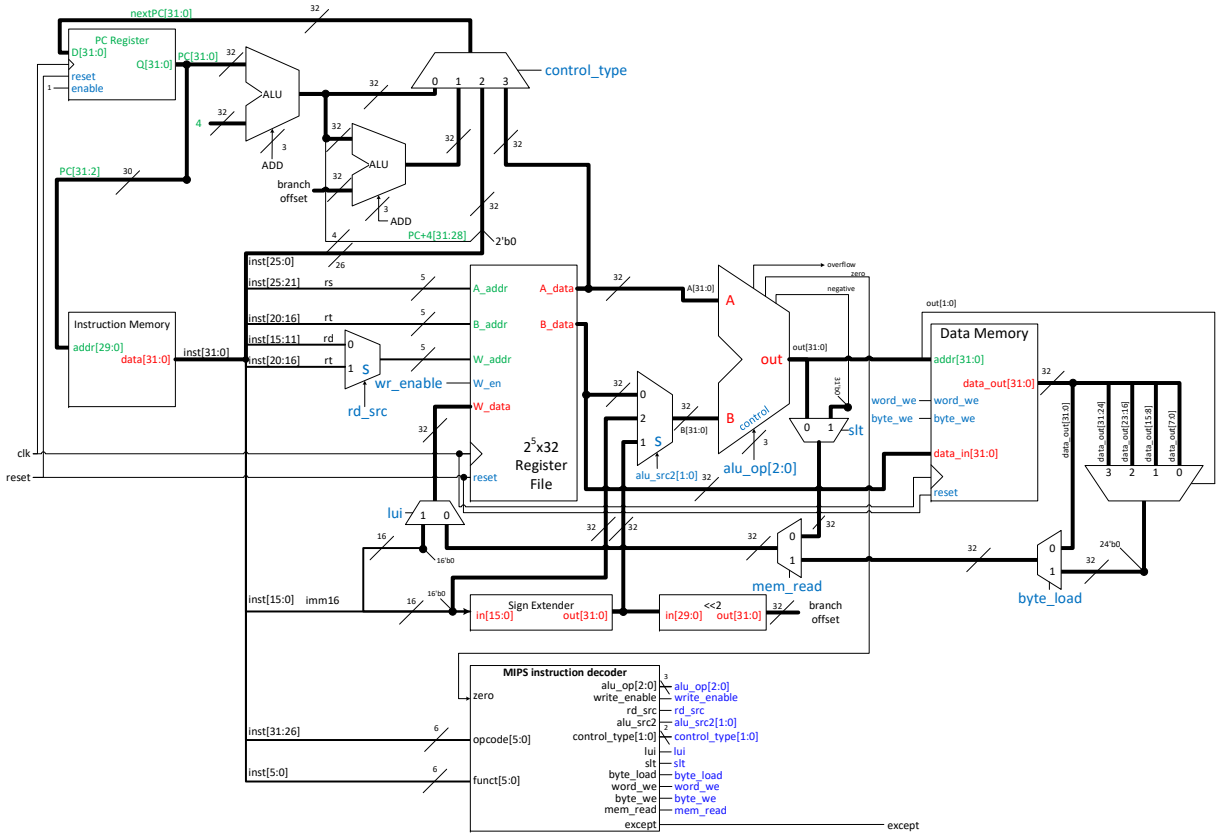
And finally, don't panic! While there are many things to connect, you already know how to instantiate modules and wire them together. So long as you're careful, everything should fall into place.


Data Memory
addr[31:0]
data_out[31:0]
word_we
byte_we
data_in[31:0]
reset
data_out[31:0]
data_out[7:0]
data_out[15:8]
data_out[23:16]
data_out[31:24]
24'b0
byte_load
mem_read
branch offset
alu_op[2:0]
slt
overflow
zero
negative
out[31:0]
out[1:0]
A
B
out
Control
A[31:0]
B[31:0]
alu_src2[1:0]
<<2
in[29:0]
out[31:0]
control_type
2'b0
A_data
B_data
A_addr
B_addr
W_addr
W_en
W_data
reset
2^5x32 Register File
Sign Extender
in[15:0]
out[31:0]
lui
16'b0
wr_enable
rd_src
ALU
ADD
PC+4[31:28]
branch offset
inst[25:0]
inst[25:21]
inst[20:16]
inst[15:11]
inst[20:16]
rs
rt
rd
rt
imm16
inst[15:0]
MIPS instruction decoder
alu_op[2:0]
write_enable
rd_src
alu_src2
control_type[1:0]
lui
slt
byte_load
word_we
byte_we
mem_read
except
zero
opcode[5:0]
funct[5:0]
inst[31:26]
inst[5:0]
nextPC[31:0]
PC Register
D[31:0]
Q[31:0]
reset
enable
PC[31:0]
PC[31:2]
4
Instruction Memory
addr[29:0]
data[31:0]
inst[31:0]
clk
reset

Datapath modification for ISA extension

Modify the datapath and decoder to extend the current MIPS ISA with the new `addm` instruction, as described on the last page of this week's discussion handout. In `mips_defines.v`, the `addm` instruction is an R-Type instruction with function code `6'h2c`. You will need to recognize this instruction in your decoder and add the new parts of the datapath in `full_machine.v`.

The semantics of the new `addm` instruction have been defined in the Discussion Section as `R[rd] = Memory[R[rs] + 0] + R[rt]`. For this extension you can use the design that your group constructed during the discussion section, or use a new design using the datapath below.



Fixing slt

Our slt implementation, as discussed in lecture, won't always give correct results. Figure out why and put a corrected implementation in your `full_machine.v` to get 5 points of extra credit.