

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 (via SVN)

By the first deadline

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 you did for the steering circuit. Your decoder needs to support **all the instructions from Lab 5**, 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

1. `full_machine.v`: This file contains the module `full_machine`, which implements the data path for **all instructions from Lab 5**, 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 5, 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 5, so you should reuse portions of your Lab 5 Verilog in this Lab.

2. `fm.s`: Additional tests for your full machine; the next section has the details. Not autograded (see `decoder_tb.v`).

The files that we provide for Lab 6 include many of those we provided from Lab 5, with the following notable additions/extension. None of these need to be checked in.

- `rom.v`: has a data memory in addition to the instruction memory from Lab 5.
- `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`, and their names should be self-explanatory.

Before you can run the full machine, however, you need to set up a test case to use. 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://wiki.cites.illinois.edu/wiki/display/cs233sp17/CS+233+on+Your+Own+Machine>

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.

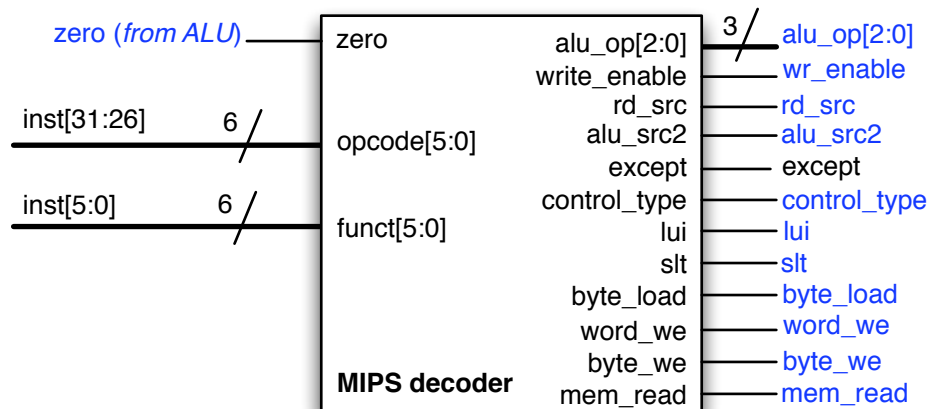
```
machine:Lab6 zilles$ ls -l memory*
lrwxr-xr-x  1 zilles  staff   14 Sep 29 13:00 memory.data.dat -> lwbr.data.dat
lrwxr-xr-x  1 zilles  staff   14 Sep 29 13:00 memory.text.dat -> lwbr.text.dat
```

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
```

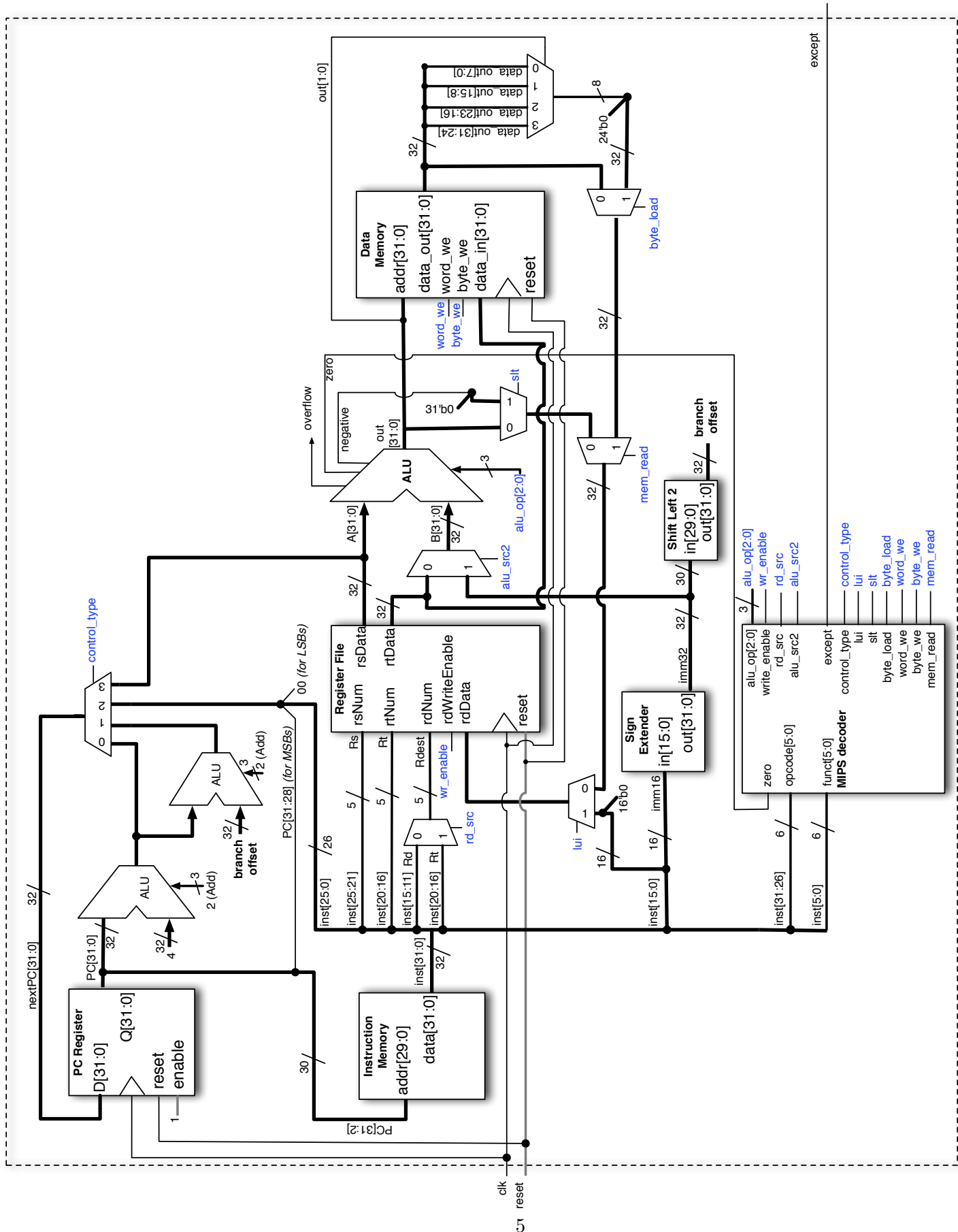
Decoder



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

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											

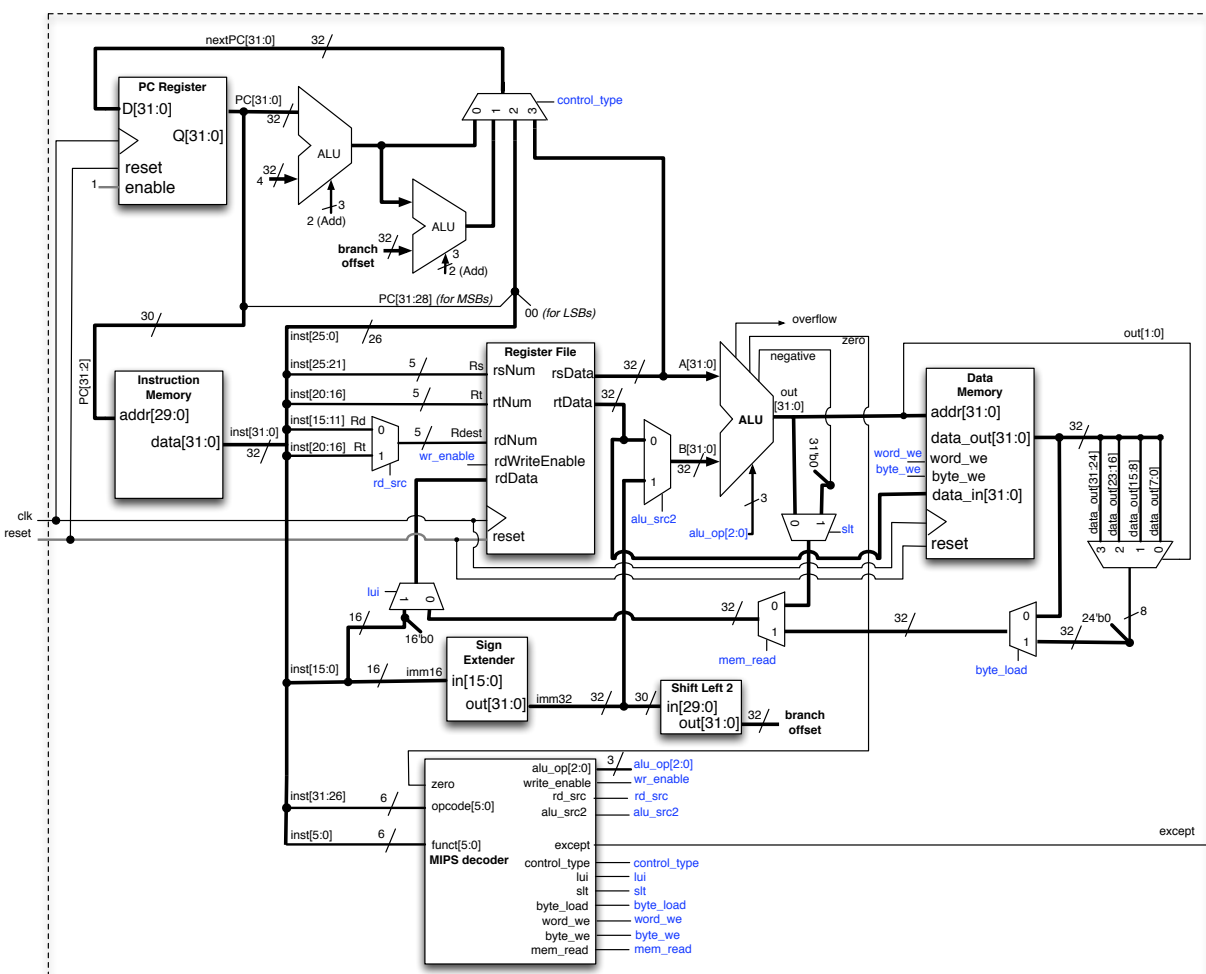
```
// mips_decode: a decoder for MIPS arithmetic instructions
//
// alu_op      (output) - control signal to be sent to the ALU
// writeenable (output) - should a new value be captured by the register file
// rd_src      (output) - should the destination register be rd (0) or rt (1)
// alu_src2     (output) - should the 2nd ALU source be a register (0) or an immediate (1)
// except      (output) - set to 1 when the opcode/func field 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
```



Datapath modification for ISA extension

Modify the datapath and decoder to extend the current MIPS ISA with the new `addm` instruction that we introduced in the discussion section. In `mips_defines.v`, we have defined the function code for the `addm` instruction as `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 $R[rd] = \text{Memory}[R[rs] + 0] + R[rt]$. 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.