Learning Objectives

1. Understanding the implementation of branches, loads, and stores in a processor datapath.
2. Building an instruction decoder

Work that needs to be handed in (via SVN)

1. `full_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) `full_machine`: This module implements the data path for all instructions from Lab 5, plus the following instructions:

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

   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. As there is significant overlap between this assignment and Lab 5, you will likely find it useful to reuse a portion of your Lab 5 Verilog in this Lab.

That's it.

The files that we provide for Lab 6 include many of those we provided from Lab 5, with the following notable additions/extension. Again, 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 `lwbr2.s`: two example MIPS assembly programs for testing your code
- `all.text.dat` and `lwbr2.text.dat`: the machine language that corresponds to the two provided assembly programs.
- `all.data.dat` and `lwbr2.data.dat`: the data memory images that correspond to the two provided assembly programs.

Setting up the test cases

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 (neither of which 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 lwbr2.text.dat memory.text.dat
ln -s lwbr2.data.dat memory.data.dat
```

By typing `ls -l memory*`, we can see that now the `memory.*` files refer to their `lwbr2` counterparts.

```
machine:Lab6 zilles$ ls -l memory*
lrwxr-xr-x  1 zilles  staff  14 Sep 29 13:00 memory.data.dat -> lwbr2.data.dat
lrwxr-xr-x  1 zilles  staff  14 Sep 29 13:00 memory.text.dat -> lwbr2.text.dat
```

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

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

We've provided these commands as part of the Makefile, accessible with `make all_test` and `make lwbr_test`. Be sure to run one of these two commands before `make`.

Load Address (la)

For programmer convenience, the MIPS assembler provides *pseudo-instructions*. These are instructions that aren't really implemented by the machine, that the assember translates into sequences of real instructions. One example is `load address`. The `la` pseudo-instruction writes into a register the address associated with a label.

```
        la      $2, array         # $2 = address associated with the label "array"
```

`la` works with labels from both the `.text` and `.data` segments.

What values will be in the registers after this code executes?

```
.data   # data segment begins at address 0x10010000
array:  .word   1       255      1024

.text
main:   addi    $10, $0, 255
        la      $2, array         # aka:  lui $2, 4097
        lw      $3, 0($2)
        lw      $4, 4($2)
        beq     $4, $10, equal

        addi    $11, $0, 1
        j       end

equal:  addi    $12, $0, 2
end:    addi    $13, $0, 3
```

reg	value
r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13	

What values will be in the registers after this code executes?

```
.data
array:  .word   1   255   1024  0xcafebabe

.text
main:   la      $2, array       # aka:  lui $2, 4097

        lw      $3, 12($2)
        slt     $8, $3, $0
        slt     $9, $2, $0
        slt     $10, $0, $2
        slt     $11, $2, $3

        lbu     $4, 12($2)
        lbu     $5, 13($2)
        lbu     $6, 14($2)
        lbu     $7, 15($2)

        sw      $0, 0($2)
        lw      $12, 0($2)
        sb      $4, 2($2)
        lw      $13, 0($2)
```

reg	value
r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13	

Endian-ness

There are two possible ways to store multi-byte data words (*e.g.*, 32-bit integers) in memory; in both cases when a word is stored to address N, the bytes are stored in byte addresses N, N+1, N+2, and N+3. *Little endian* stores the least significant byte (LSB) in the first byte (N, *i.e.*, little end first), while *big endian* stores the most significant byte (MSB) in the first byte (N, *i.e.*, big end first). For example, if storing the value `0xdeadbeef` to address `0x10000000`, you get the following:

ADDRESS:	**0x10000000**	**0x10000001**	**0x10000002**	**0x10000003**
Little Endian	`0xEF`	`0xBE`	`0xAD`	`0xDE`
Big Endian	`0xDE`	`0xAD`	`0xBE`	`0xEF`

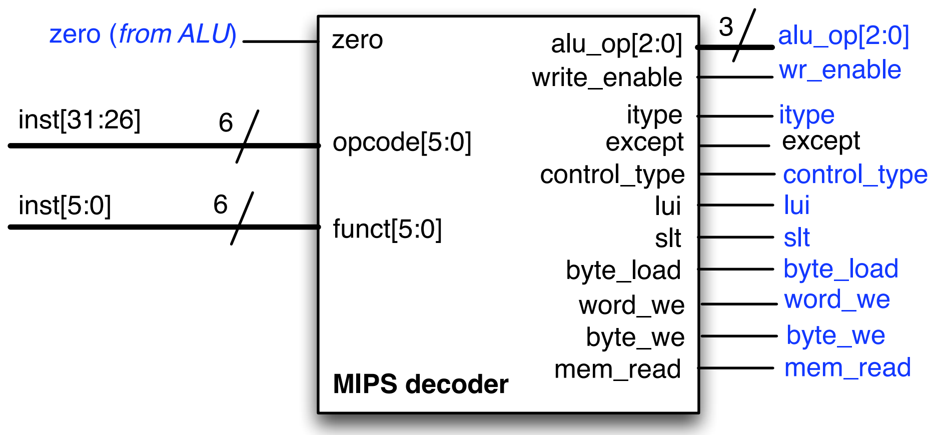
Both conventions are in active use by popular microprocessors: Alpha and x86 are little endian, SPARC and PowerPC are big endian. MIPS is actually bi-endian (*i.e.*, it can be run in either mode), but **our Verilog MIPS processor should implement little-endian.** SPIM (somewhat sadistically) uses the endianness of its host. That is, SPIM is big endian on a SPARC machine and little endian on an x86. Since the EWS machines are all x86, what you get from SPIM will correspond to our Verilog implementation.

As long as you don't cast one data type to another, you shouldn't have to worry about whether you are running on a big-endian or little-endian machine. That is unless you need to communicate over the network.

When two computers communicate, they must ensure that numbers are correctly interpreted even if one machine is little endian and the other is big endian. For example, remote procedure call (RPC) defines big endian as the canonical form (not surprising since Sun defined it and SPARC is big endian). Thus, little endian machines need to "swizzle" their bytes to big endian before sending them across the network.

```
unsigned swizzle_word(unsigned in) {
   unsigned temp = in >> 24;
   temp |= ((in >> 16) & 0xff) << 8;
   temp |= ((in >> 8) & 0xff) << 16;
   temp |= (in & 0xff) << 24;
   return temp;
}
```

Decoder



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

inst	alu_op			itype	wr_enable	ctrl_type		mem_read	word_we	byte_we	byte_load	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
// itype       (output) - ALU receives 2 reg values (0) or 1 reg. value and 1 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
```