← Lab 4: Bitfields

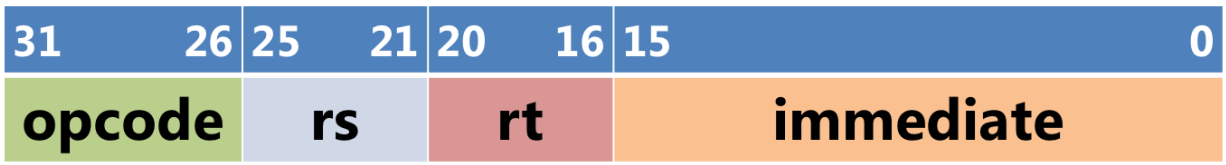
Finish by midnight on Sunday, 10/7

By now you know that machine language is the pattern of bits a CPU uses to encode its instructions. Instructions are usually encoded as **bitfields** in order to pack a lot of information into a small number of bits.

MIPS uses a few different *instruction formats* in order to encode its instructions. No matter what format, **all MIPS instructions are 32 bits.**

The one you'll be looking at today is the **I-type** format, where "I" stands for "immediate." This is used to encode instructions with two registers and an immediate such as `li`, `addi`, `lw`, and `beq`.

Here's how it looks:



- The **bit numbers** show the numbers of the first and last bits, **inclusive,** of each field.
- The **opcode** field says which instruction this is.
- The **rs** and **rt** fields encode the two registers used.
- The **immediate** field is the immediate value (the number in the instruction).

So for example, `addi t0, s1, 123` will be encoded as:

- **opcode** = 8 (the designers decided that 8 means `addi`)
- **rs** = 17 (`s1` is register 17)
- **rt** = 8 (`t0` is register 8)
- **immediate** = 123

First: think about it

Using the diagram above, write yourself some notes to answer these questions.

- How many bits are in each field (`opcode`, `rs`, `rt`, and `immediate`)?
  - Be careful, it's easy to mis-count.

- What is the **position** of each field?
  - This is the amount you'll be shifting by to encode/decode the bitfield.
- What is the **mask** for each field **in hexadecimal?**
  - Remember, the mask is the special value that you AND with **after shifting right.**
  - It's based on the number of bits in the field.
  - You can think about it in binary and then turn that to hex.

Now: write a program about it

Right click this link and save it. It's the skeleton/driver code for this lab.

The goal is to have your program output the following:

```
0x2228007b
0x1100fff8

opcode = 8
rs = 17
rt = 8
immediate = 123

opcode = 4
rs = 8
rt = 0
immediate = -8
```

Encoding instructions

The `encode_instruction` function takes four arguments in this order: opcode, rs, rt, and immediate. (Look at `main` to see how it's called.)

It should:

- encode the instruction using left-shifts (`sll`) and ORs (`or`)
- print the resulting instruction using syscall 34 (prints a hex number)
- print a newline

This function isn't too long. Once you write it, you might see this:

```
0x2228007b
0xfffffff8
```

The second instruction doesn't quite look right. Step through and have a look at what value is in `a3`. It's -8... negative numbers have a bunch of `1` bits at the beginning. That's not good.

The immediate should only be **16 bits.** So **how can you "filter out" the low 16 bits of `a3` and "turn off" the upper 16 bits?** Do that in `encode_instruction` before ORing everything together, and you should now get the correct output:

```
0x2228007b
0x1100fff8
```

Decoding instructions

Decoding isn't much more complicated, but you'll be printing it out with strings, so that will make this function a bit longer.

`decode_instruction` takes 1 argument: the encoded instruction to be decoded.

It should do the following:

- print the `opcode` string
- extract the value of the opcode and print it with syscall 1
- print the `rs` string
- extract the value of rs and print it with syscall 1
- print the `rt` string
- extract the value of rt and print it with syscall 1
- print the `immediate` string
- extract the value of the immediate and print it with syscall 1

Pretty straightforward, but there are a few things to note.

You have to reuse `a0`, so, uh, hm.

The syscalls expect their arguments in `a0`, but this function takes an argument in `a0`.

- Which register should you `move` the argument into to keep it safe?
- What do you have to do with that register to follow the calling convention?
- Once you copy the argument into that register, don't change that register's value. You'll need it multiple times.

Printing strings

I've given you 4 strings in the `.data` segment. To print a string, you use syscall 4, and you use `la` (not `li` or `lw`) to put the address of the string in `a0`:

```
        la      a0, some_string
        li      v0, 4
        syscall
```

The second instruction doesn't decode properly...

You might get this:

```
opcode = 4
rs = 8
rt = 0
immediate = 65528
```

The immediate *should* be -8. What's going on here?

It's because it's a negative number, but it needs to be *sign-extended* from 16 bits to 32 bits. Right now, it's `0x0000FFF8`; it needs to be `0xFFFFFFF8`.

We can do this with a funky trick (assuming that the value to be sign-extended is in `t0` to begin with):

```
        sll t0, t0, 16
        sra t0, t0, 16 # shift right *arithmetic*
```

`sra` is a new kind of shift, an *arithmetic* right shift. It's kinda like sign-extension: instead of shifting 0s into the left side, it smears the top bit into the new places. Just like sign extension!

If you do this after `AND`ing the immediate value, you should now get -8 in the output!

Submitting

Make sure your file is named `username_lab4.asm`, like `jfb42_lab4.asm`.

Submit here.

Drag your asm file into your browser to upload. **If you can see your file, you uploaded it correctly!**

You can also re-upload if you made a mistake and need to fix it.