Computer Architecture

Lab 3

The goals of this lab are to work with logic and shift instructions.

You should turn in the source file for the program you're asked to write. It should be submitted via the tool available on Sakai for this lab. It is due by the start of the next class period.

Bitwise

Every assembly language has ways to manipulate the individual bits in the 1's and 0's representation of data (and code.) Together, these are called **bitwise** instructions. They take their names from their Boolean Algebra counterparts and therefore the names are pretty poor descriptors of what they actually do. We just have to roll with that and remember how to use them.

Just know the following: the **`and`** instruction **multiplies** bits, the **`or`** instruction **adds** bits *inclusively* (1+1=1), and the **`xor`** instruction **adds** bits *exclusively* (1+1=0).

We use **`and`** to make bits 0 (0 times anything is 0.) We use **`or`** to make bits 1 (1 plus anything is 1.) We use **`xor`** to complement bits (1+1=0 and 0+1=1.) Together, we can use these instructions with **masks** to manipulate **bit strings**.

In addition to these **logic** instructions, there are **shift** instructions. Shifting a number moves the bits to the left or the right by 1 bit. So, the bit string `1101` shifting left becomes `1010` and shifted right becomes `1110` (if using an *arithmetic* shift) or `0110` (if using a *logical* shift.) The mathematical effect of shifting is to multiply or divide by powers of 2 and they can be used to do math more efficiently than relying on **`mul`** and **`div`** instructions. Usually, however, they are used to manipulate bits by moving them around based on position or field.

C and C++ can do all of this as well. The bitwise operators `&`, `|`, `^`, `<<`, and `>>` provide `and`, `or`, `xor`, and `shift` capabilities.

(Software development note: Java and Python can do these things, too, using basically the same syntax. That's because this syntax started with C, the greatest programming language in the history of the universe.)

Lab Exercise

In class we worked with the following C++ function to convert the 1's and 0's that make up a variable into a `string`.

```
string toBits(unsigned char c)
{
    string bits = "";
    for (int i = 0; i < 8; i++)
    {
        bits += (((c << i) & 128) ? '1' : '0');
    }

    return bits;
}
```

First of all, you should make sure to understand what is going on here. The **`&`** with **`128`** will perform the **`and`** operation with the bit string **`10000000`** which will result in isolating the most significant bit. Make sure you see that! Then the **`?`** operation looks at the result of the **`and`** and executes either the first thing in the **`:`** or the second thing based on whether the result is non-zero or zero. So, if there is a 1 in the most significant bit we end up adding a **`'1'`** to the bit string and if there is a 0 in the most significant bit we end up adding a **`'0'`** to the bit string. Finally, the **`c << i`** will shift the value in **`c`** to the left so in the next iteration we look at the next bit. In this way, we isolate each bit in the representation of **`c`** and output them into the string bits. We do all this 8 times because there are 8 bits in a **`char`**.

Your task is to write this in MIPS.

Let the user type in a number and store it in a location using **`.word`**. Then put the string in a memory location using **`.asciiz`** and then output it using the appropriate **`syscall`** we learned in last week's lab.

Since a word size in MIPS is 32 bits, you'll have to loop 32 times instead of 8 times, but the core of the loop remains the same.

The **`sll rd, rt, shmt`** instruction will shift left (we need 1 for **`shmt`**) and the **`and rd, rs, rt`** instruction will **`and`** the values in **`rs`** and **`rt`** and store the result in **`rd`**. You can store the appropriate **mask** value (it won't be 128) in a location called **`mask:`** and then load it into a register using our **`la`**/**`lw`** combo.

Finally, look again at the original C++ code. We shift the value in **`c`** while leaving the mask of 128 the same. We could do it another way. We could keep the value in **`c`** the same while shifting the mask each iteration. Be sure you know how to write the loop that way as well. Load up Visual Studio (or g++) and test it out.