Family Name:_____

Given Names:_____

ID#:_____

CMPT 295 Lab 3 (0.5%)

The purpose of this lab is to introduce you to the use of bitwise operators. You will also verify the endianness of your machine.

Part 1: Representing Binary Sequences

- Login to the CSIL machines in the Linux environment, download and unpack the care package into `sfuhome`, and change directories into `lab3`. A directory listing should reveal a `makefile` and the source files `main.c`, and `get_byte_by_addr.s`.
- Start by opening `main.c` in your favourite editor. The program contains two variable declarations: one for type `unsigned int` and one for a plain old `int`. They are both the same size (32 bits) but their values span different ranges.
- Though you could initialize each with a decimal value, the point of today's lab is to work with the bits themselves. Therefore, you will use hex format: assign to `x` the value `0x01234567`, and assign to `y` the value `0xfffffd6`.

  Note: The "`0x`" hex notation — pronounced "zero x" or "zero cross" — also works in x86-64 assembly. Capitalization on the hex digits is allowed, but doesn't affect the value. If you don't supply enough digits to fill the variable space, then the most significant bits are set to 0. E.g., `0xf` would be equivalent to `0x0000000f`.
- Within the mainline routine, complete the four `printf()` statements so they output:

  ```
  x = <hex equivalent of x>
  x = <decimal equivalent of x>
  y = <hex equivalent of y>
  y = <decimal equivalent of y>
  ```

  with the correct substitutions. Try the linux command `man format` to find the correct format strings.

  Hint: To print out all 8 hex digits, including the leading zeros, prepend your format character by "`.08`". To print out "`0x`", append the "`%`" character with a "`#`".

  Note: Traditionally, C format strings do not have to match their underlying types. This can lead to unexpected output bugs.
- Call your TA to your workstation to view your result before proceeding to Part 2.

Part 2: Extracting Binary Sequences

- The C bitwise operators `&` and `>>` can be used to retrieve sequences of bits. With just `&`, you can *mask* the portion of the binary that you wish to keep. For example, if you tried:

  ```
  lowest_byte = y & 0xff;
  ```

  it would effectively ignore the highest 24 bits of `y` and keep the lowest 8. Remember that the binary of `0xff` is `1111 1111`, and that logically `1 & x = x`. That's why it's called a mask: just like a real-life mask blocks out the parts that would prefer remain unseen, the 0s in the mask cause those columns of the binary to remain unseen.

- The next least significant byte could similarly be masked by

  ```
  byte1 = y & 0xff00;
  ```

  If you print out the values of `lowest_byte` and `byte1`, you should notice that `byte1` is still in its correct position: in bits 15 through 8. To correct this, shift the result to the right by 8 places.

  ```
  byte1 = ((y & 0xff00) >> 8) & 0xff;
  ```

- Generalize this idea and complete the function `get_byte_by_order(int x, int i)` which returns the `i`$^{\text{th}}$ least significant byte, where `i` is 0, 1, 2 or 3. Re-`make` your code and test that it works before proceeding to Part 3.

Part 3: Endianness

- Open `main.s` in your editor. You should see a declaration for both `x` and `y`: two `.long` directives, followed by their integer equivalents. Observe that the initial value for `y` is negative. Though this seems to go against the declaration of `unsigned int`, remember that the machine language executable carries with it no notion of types: the interpretation of the bits are left up to the program itself.

- Replace the initial value of `x` to be 233876875. Replace the initial value of `y` to be 2908479998. Then, save and close `main.s`.

- To prove the endianness of the machine, you will complete the function `get_byte_by_addr(int *base, int i)`, which gets the `i`$^{\text{th}}$ byte of the integer at address `base`, as they would appear in memory. As with the other function, `i` is 0, 1, 2 or 3, thus the result will be the byte at address `base` + 0, `base` + 1, `base` + 2 or `base` + 3.

  Though it is possible to write this function in C, it is awkward. By contrast, the function is almost trivial to write it in assembly.

  Open the file `get_byte_by_addr.s`, and complete the code.

  Variable Map:

  - `%rdi` contains the address `base`.
  - `%esi` contains the byte offset `i`.
  - place the return value into `%al`.

- This is the last step of Lab 3. Call the TA to your terminal to view your assembly source code. Then do a `make`, and run your code. After verifying your work, the TA will then collect this sheet.

Thinking Ahead

There are many other codes besides integers, and it is possible to do bitwise operators on all of them, albeit indirectly.

One data type that will be introduced shortly is the `float`, a 32-bit single-precision floating point number, basically designed to do real-valued arithmetic.

As an experiment, create a variable `x` and an output statement like so:

```
float x = 1.75;
printf("x = %f\n", x);
```

Bit 31, the most significant bit of `x`, is 0. To set it, you can use an *or mask*:

```
int *p = (int *)(&x);
*p = *p | 0x80000000;
```

What was the effect on `x`?

You can also use an *xor mask* to toggle bits, from 0 to 1 or from 1 to 0. Bit 23 of `x` is interesting to toggle. What do you think is the meaning of bit 23? You might also want to toggle bit 24.