CS061 – Lab 8
Advanced Bit Manipulation

1 High Level Description

The purpose of this lab is to give you some experience in doing some advanced bit manipulation techniques, making use of both left- and right-shifting.

2 Our Objectives for This Week

1. Exercise 1 – Review & extend assignment 4
2. Exercise 2 – Counting bits
3. Exercise 3 – Right Shift

Exercise 1

Hey, remember that Programming Assignment (assn 4) you finished last week? Sweet, huh? Let's turn it around:

Specs:

1. The main code block (the test harness) will just invoke subroutine 1, and then subroutine 2, using the register value created by sub 1 to set up input for sub 2.
2. Subroutine 1: This subroutine will just take a hard-coded (.FILL) value, and load it into a register of your choice.
3. (In the test harness) add 1 to the number and invoke subroutine 2:
4. Subroutine 2: Using your chosen output register from sub 1 as an input register for sub 2, print the new value out to the console as a decimal number *(i.e. the opposite of Programming Assignment 4)*.
   Work out the algorithm for yourself by studying the algorithm we give you for assn 4, and then working "backwards" from that.
   ALWAYS, ALWAYS, WRITE OUT YOUR ALGORITHM - express it in bullet points, in pseudo-code, in C++, however you like - but DO IT!
5. Just for fun, try entering the decimal number 32767 into this program; what do you get?

Exercise 2

Write a subroutine that counts the number of binary 1's in the value stored in a given register *(this is part of a technique known as a "parity check" - go look it up!)*

Specs:

1. The main code block (the test harness) should ask the user to input a single character at the keyboard.
2. The input should be "passed" as a parameter to the subroutine (i.e. the input character is copied to the subroutine's input register).
3. The subroutine should "return" the number of binary 1's in the input character in another register (i.e. it counts the number of 1's in the original character, and stores that count in a "return" register). The subroutine should not alter the input register.
4. The main code block should then print the result in a reasonably intelligent format:
   Example: The ASCII code for a semi-colon (';') is x3B == b0000 0000 0011 1011
   which contains five binary 1's, so your program will output:
   `The number of 1's in ';' is: 5`
   Hint: we will test this exercise only with ascii characters from the keyboard (x20 - x7F) - i.e. the maximum possible number of 1's will be 7, so your output will always be a *single digit numeric character*.

Exercise 3

You have to read & understand this exercise, but you don't need to code it - you just need to show your TA the algorithm you would use *(use the outline below)**.***

Ok, here we go. Are you feeling advanced? You *look* advanced. You're awesome! You can totally do this! Ready? Ok!

Build a subroutine that takes, as a parameter, the value of a register and **right-shifts** it by one bit, losing the trailing bit, and filling in the empty leading bit with 0:

Example:

```
(R1) ← xABCD     ; (b1010 1011 1100 1101)
[call the right-shift subroutine]
[now (R1) == x55E6 == b0101 0101 1110 0110]
```

Note how we shifted-in a 0 when we did the right-shift, and "lost" the 1 in the lsb?
This is called a *logical right shift*; there are two other variations, the arithmetic right-shift (which maintains the sign bit rather than always shifting-in a 0 to the msb); and a right-rotate, which shifts in into the msb the bit shifted out of the lsb. For now, we will just deal with the logical right-shift.

Discussion:

Alright, let's think about this thing. When we **left-shift**, we are doubling the number (aka: multiplying it by two - aka: adding it to itself - aka: ADD R1, R1, R1), right? So if left-shift is multiplication, then what might **right-shift** be?

Yep, you guessed it: Division. If you left-shift the number 2, you get 4. If you right-shift 4, you go back to having 2. So, division. Right. Got it.

"How on earth are we supposed to right shift?!" the TA said rhetorically.

[Whacky Student]: If left-shifting is ADDing a number to itself, then right-shift must be subtracting it from itself, right?

[TA]: *blank stare*

[Less Whacky Student]: *whispers* "Dude, that would make it zero."

[Whacky Student]: *facepalm*

Well, sadly, there is no super-simple way to perform a binary right-shift in software. Still, it's not insanely difficult. Let's think about it:

- Left-shifting is short and sweet: add the number to itself.
- Right-shifting is not quite as simple. There is no way to directly do it in software.
- Hmm… how else can we mess around with the bits? We can shove them left and shift-in a 0 every time to the lsb (that's a logical left-shift)… but what if instead we **rotated** the bits (i.e. take the msb being shifted out, and shift it in as the new lsb)?

Hmmm:

- Given the 4-bit (unsigned) number xC == b1100 == #12 (UNsigned magnitude)
- [Rotate left, noting that msb is 1, which gets rotated in to the lsb]
- Now we have b1001 == #9
- [Rotate left, noting that msb is again 1, which gets rotated in to the lsb]
- Now we have b0011 == #3
- [Rotate left, noting that msb is now 0, which gets rotated in to the lsb]
- Now we have b0110 == #6 - note that msb is now 0

Ok, this is boring. Let's stop.

Hey wait! Oh my gosh!
We have 6 == 12/2 now! How'd that happen?!?
[This is where you, the Less Whacky Student, fill in the blanks ☺]
*Note: This only works for **unsigned** representation. How could it be made to work for two's complement representation?*

3.2 Submission

Demo your lab exercises to your TA ***before you leave lab***.

If you are unable to complete all exercises in lab, show your TA how far you got, and request permission to complete it after lab.
Your TA will usually give you partial credit for what you have done, and allow you to complete & demo the rest later for full credit, so long as you have worked at it seriously for the full 3 hours.
When you're done, demo it to any of the TAs in office hours ***before your next lab***.
Office hours are posted on Piazza, under the "Staff" tab.

4 So what do I know now?

... more about bit-manipulation than you probably ever wanted to know :)