CS061 – Lab 07
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. Exercises 01 – Review & extend assignment 4
2. Exercise 02 – Counting bits
3. Exercise 03 – Right Shift

REMEMBER: ALL your programs must now be written as subroutines!!

Call your subroutines with the JSRR instruction *(it works with subroutines located anywhere in memory)*

And remember to include all four steps of subroutine construction, being *especially* careful to back up & restore R7 *(or your program won't work and you won't be able to figure out why!)*

Exercise 01

Hey, remember that Programming Assignment (assn 4) you just did? Sweet, huh? Let's reverse it::

Specs:

1. The main code block (test harness) should just invoke subroutine 1, and then use the resulting register value as input to subroutine 2:
2. Subroutine 1: ask the user to Input any decimal number between [-32768, 32767] from the keyboard; convert it into its numerical equivalent and put it into a register *(use your Programming Assignment 04 code for this)* The input should be able to recognize & handle a leading sign ('+' or '-') and/or leading 0's - e.g. "+000005"
3. (In the test harness) add 1 to the number and invoke subroutine 2:
4. Subroutine 2: Print the new value out to the console as a decimal number *(i.e. the inverse of Programming Assignment 04)*.
5. Just for fun, try entering the number #32767 into this program; what do you get?

Exercise 02

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

Specs:

6. The main code block (test harness) should ask the user to input a single character at the keyboard.
7. The input should be "passed" as a parameter to the subroutine (i.e. the input character is put into a register that is available to the subroutine).
8. The subroutine should "return" the number of binary 1's in the input character in another register (i.e. it places the number of 1's in a register and doesn't restore that register before returning).
9. 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 something like:
   "The number of 1's is: **5**" or "If you counted the 1's by hand, you would get **5**".

Exercise 03

- You have to read & understand this exercise, but you don't need to submit it.

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, filling in the empty leading bit with 0:

Example:

(R1) ← xABCD ; (b1010 1011 1100 1101)
[call the right-shift subroutine]
[now (R1) == x55E6 == b**0**101 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 to the msb the bit shifted out of the lsb. For the moment, 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 in blazes are we suppose 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*

[Whacky Student]: …

[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. 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.
- 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 xE == b1110 == #14 (UNsigned magnitude)
- [Rotate left, noting that msb is 1]
- Now we have b1101 == #13
- [Rotate left, noting that msb is 1]
- Now we have b1011 == #11
- [Rotate left, noting that msb is 1]
- Now we have b0111 == #7 - note that msb is now 0

Ok, this is boring. Let's stop.

> Hey wait! Oh my gosh!
> We have 14/2 == 7 now! How'd that happen?!?
> [This is where you, the Less Whacky Student, fill in the blanks ☺]
> *Note: This only works for **positive** values. How could it be made to work for negative values?*