Homework #2
Introduction to SPIM
Due October 15th, by electronic handin (use the "turnin" program)

Read: Chapter 3.5 - 3.8, Appendix A-6, A-8 & A-9,

Purpose: The point of this assignment is to ease you into SPIM and MIPS assembly programming. The next assignment will be to make you masters at it.

Start by downloading the extended SPIM simulator and installing it. You can find it at:

http://www.cs.washington.edu/education/courses/cse410/CurrentQtr/software410.html

The version of SPIM you will be using in this class has been extended by Doug Johnson to support file I/O operations. In this first assignment you will not need these extensions, but in Homework #3 you will make extensive use of them. So you might as well start with the extended SPIM binaries now and get ready for homework #3.

As a warm-up exercise, do problems 3.2 and 3.6 in the book. This will give you a chance to read and interpret someone elses's code (3.2) and correct someone else's bugs (3.6) before tackling your own. :-) Do the first exercise mentally, without using the SPIM simulator. The second can be done with the simulator.

Now you're ready for the main event!

Your task is to write two small MIPS assembly programs. The first program will prompt the user for input of a decimal number (potentially, but not necessarily, prefixed with a + or - sign). This program will then output the 32 bit 2's complement binary representation of this number. For example, the output of your program may look something like this:

```
Input> -2
Binary = 11111111111111111111111111111110
```

** NOTE: Use the syscall in SPIM to read a string and write your own string -> number converter (similar to an atoi() function in C). Do not use the "read integer" function of SPIM.

** DOUBLE NOTE: Be sure to check for erroneous input and tell the user to be good and enter numbers correctly.

The second program will do the opposite. It will prompt the user for a 2's complement binary number and then sign extend it and print out the decimal representation. Sign extending is a process that occurs often within a microprocessor. For example, consider the following C code:

```
char    c= -1;
int     i = c;
```

The first statement assigns 11111111 to the variable c. However, when i is assigned c what value should it get? Should it be 0x000000ff or should it be 0xffffffff? The answer is the latter. The way this is done is to take the top most bit of c and replicate it upwards throughout the upper

bits of i. This process (sign extending) ensures that the result (the value of i) has the meaningful value the programmer intended (that is -1). The output of your program should look something like this:

Input> 010
Decimal = 2

Or

Input> 10
Decimal = -2

As before, you should check for erroneous input and output some interesting message. Also use the syscall for reading a string, not for reading an integer (can't make this too easy!).

You are required to comment your code in this course and it will be part of the grade for your assignment. This is partly for the TAs' benefit, but it is also good programming practice and will help you when you are writing larger programs. Moreover, if your code doesn't work and it is well commented, you are more likely to get partial credit than if it has no comments.

HERE IS AN EXAMPLE OF EASY TO READ CODE:

```
        li      $t0, 0                  # use t0 as an index register, zero it out to start
        li      $t4, 0                  # Sum = 0
LoopStart:
        lw      $t1, DataSize           # use $t1 as the end point comparison
        bge     $t0, $t1, EndOfLoop     # while ( index < DataSize )
        lw      $t2, ArrayX
        sll     $t3, $t0, 2             # multiply index*4 to get offset into Array
        lw      $t3, 0($t2)             # t3 = ArrayX[Index]
        add     $t4, $t4, $t3           # Sum = Sum + ArrayX[Index]
        addi    $t0, $t0, 1             # index = index +1
        j       LoopStart               # jump back to start of loop
EndOfLoop                               # index >= DataSize here.
```

HERE IS THE SAME LOOP, BUT IN A DIFFICULT TO READ FORM:

```
        xor     $8, $8, $8
        xor     $9, $9, $9
        lw      $10, DataSize
        sll     $10, $10, 2
        lw      $12, ArrayX
LoopStart:
        slt     $11, $8, $10
        beq     $11, $0, EndOfLoop
        addi    $13, $12, $8
        lw      $13, 0($13)
        add     $9, $9, $13
        addi    $8, $8, 4
        j       LoopStart
EndOfLoop:
```