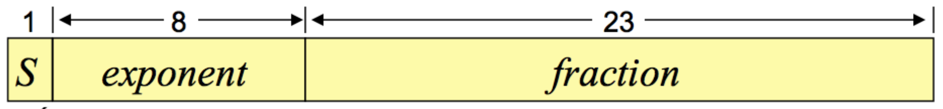
Question 4 (30 points): In decimal notation we can define the *decimal-point position* of an integer as the position of the most-significant digit of the integer (counting the position of the least-significant digit as position zero). This definition is convenient because the decimal-point position is equal the value of the exponent when the number is written into normalized scientific notation. Below are a few examples:

Decimal Integer Value	decimal-point position	normalized scientific notation
7	0	7.0×10^0
31	1	3.1×10^1
15624	4	1.5634×10^4

Similarly, we can define the *binary-point position* for a positive integer value represented in two-complement notation as the position of the most-significant non-zero bit.

A *casting* operation is an operation that transforms the data type of a value. Assume that a floating-point number is represented in the IEEE 754 format shown below:



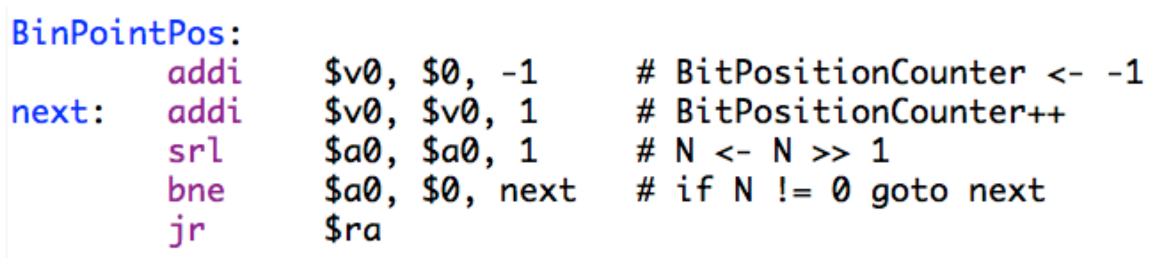
$$N = \begin{cases} (-1)^S \times 1.fraction \times 2^{exponent-127}, & 1 \leq exponent \leq 254 \\ (-1)^S \times 0.fraction \times 2^{-126}, & exponent = 0 \end{cases}$$

With this format the casting of a two-complement integer representation to a floating-point representation requires the computation of the sign, exponent and fraction and the assembling of the number into the binary floating point representation. Depending on the value of the integer this casting may incur in some loss of precision.

When writing the programs below you must follow all the MIPS function calling and parameter passing conventions. Also in your code you are not allowed to use pseudo instructions that manipulate constants that larger than 16 bits.

a. (**10 points**) Write MIPS assembly code for a function called `BinPointPos` that receives in `$a0` the value of a non-zero positive integer represented in two-complement notation and returns in `$v0` the *binary-point position* of the number.

b. (**20 points**) Write a MIPS assembly code for a function called `IntToFloatCast` that receives in `$a0` a memory address. The 4-byte word at that address contains an integer value represented in two-complement notation. When `IntToFloatCast` returns, the same memory position will contain a 32-bit binary representation of the floating-point representation of the same value with as much precision as is possible for the casting to maintain. `IntToFloatCast` **must call** `BinPointPos` to compute the position of the binary point.



```
BinPointPos:
        addi    $v0, $0, -1     # BitPositionCounter <- -1
next:   addi    $v0, $v0, 1     # BitPositionCounter++
        srl     $a0, $a0, 1     # N <- N >> 1
        bne     $a0, $0, next   # if N != 0 goto next
        jr      $ra
```

```
# Arguments:
#       $a0: memory address of a word in memory
# Return Value:
#       None
# Register Usage:
#       $s0: F --- floating point representation
#       $s1: I --- integer representation
#       $s2: A --- memory address of word
#       $t0: E --- exponent as it appears in floating point
# Side Effects:
#       Casting of integer at memory position to floating point
IntToFloatCast:
        addi    $sp, $sp, -16
        sw      $a0, 0($sp)
        sw      $s0, 4($sp)
        sw      $s1, 8($sp)
        sw      $s2, 12($sp)
        add     $s0, $0, $0     #  F <-- 0
        add     $s2, $a0, $0    #  A <-- $a0
        lw      $s1, 0($s2)     #  I <-- Mem[A]
        beq     $s1, $0, Done   #  if I == 0 goto Done
        bgt     $s1, $0, Positive
        lui     $s0, 0x8000     # sign(F) <-- 1
        sub     $s1, $0, $s1    # Now I is positive
Positive:
        add     $a0, $s1, $0    # $a0 <-- I
        jal     BinPointPos
        addi    $t0, $v0, 127   # E <-- BinPointPos(N) + 127
        sll     $t0, $t0, 23    # move E to right position
        or      $s0, $s0, $t0   # exponent(F) <-- E
        li      $t1, 32
        sub     $t1, $t1, $v0   # $t1 <-- 32 - BinPointPos(N)
        sllv    $t2, $s1, $t1   # $t2 <-- I shifted so that bin point at 32
        slr     $t2, $t2, 9     # $t2 <-- fraction
        or      $s0, $s0, $t2   # fraction(F) <-- fraction
Done:
        sw      $s0, 0($s2)     # Mem[A] <-- F
        lw      $a0, 0($sp)
        lw      $s0, 4($sp)
        lw      $s1, 8($sp)
        lw      $s2, 12($sp)
        addi    $sp, $sp, 16
        jr      $ra
```