**Part 2. Converting An Integer Number To Floating Point Number (15 points)**

Write an interactive program that asks the user to enter a positive integer and number converts this number to floating point representation by using bit manipulations and prints both integer number and its floating point equivalent. Provide a user interface that allows the user enter integers as long as the user wants to continue. For example, if the user enters 10 which is 1010 in binary, you have to remember that its scientific notation is 1.010 x 23 hence in a register like $t0 you have to construct the bit pattern "0**1000 0010** 01000000000000000000000" sign bit:0, exponent: **1000 0010**, mantissa: 01000000000000000000000. Note that in the exponent we use excess 127 notation by following the IEEE 754 standard and it is 127= 7F16, 7F + 3= 8216, in binary it is **1000 0010**. Use proper subprograms and make sure that you use the conventions of MIPS programming by properly using the stack and registers as needed. As an extension you may accept negative integers too.

**Hint**: In order to print an integer number stored in a variable, as a floating point number you may use the following code segment.

l.s $f0, a # $f.. are floating point registers

cvt.s.w $f12, $f0

li $v0, 2

syscall # Prints 10.0

.data

a: .word 10

.data

ask: .asciiz "Please enter a positive integer numbers"

continue:

a: .word 10

.text

li $v0, 4 # print ask question

la $a0, ask

syscall

li $v0, 5

syscall

j printInt

jal convert

j printInt

convert:

add $t1, $t1, $0 #counter

add $s1, $s1, 1 #and number

loop:

move $s0, $a0

and $s2, $s1, $s0

bne $s2, 0, loopEnd

add $t1, $t1, 1

sll $s0, $s0, 1

b loop

loopEnd:

sll $s0, $s0, 1

add $t1, $t1,1

add $s4, $s4, $zero

add $s4, $s4, $s0

#mantissa

srl $s0, $s0, 9

add $t1, $t1, -32

add $t1, $t1, 127

#exponent

sll $s4, $s4, 22

add $s0, $s0, $s4

b printfloat

printInt:

#print the number

move $a1, $v0

li $v0, 1

move $a0, $a1

syscall

printfloat:

mtc1 $s0, $f12

li $v0, 2

syscall

stop:

#stop

li $v0, 10

syscall

**Part 3. MIPS: Creating Linked List Utility Routines (55 points)**

Linked lists are important dynamic data structures, useful to a variety of algorithms because of their ability to grow and shrink.  Utilities libraries for linked lists are therefore useful, so that common functions can be available to users, pre-written, tested and ready to go. In this lab, you will write the utility functions defined below for linked lists, in MIPS assembly language.  Your utilities will be combined with other utility programs such as create\_list and display\_list, and called by a main program. [Any code other than the you are asked to write in this lab will be provided (see the Unilica folder)—you don’t need to write it]

In memory, the linked lists your utilities will work with are implemented as follows: each element consists of 2 parts: pointerToNext, and value. Each part is a 32-bit MIPS word in memory. The two parts are located in successive word addresses, with the pointerToNext being first.  For example, if the byte address of the pointerToNext is 100, then the byte address of the value will be 104.  *Remember, MIPS memory is byte addressable.*

In the last element of the linked list, the pointerToNext has value 0, in other words it is the null pointer. This means that there is not any next element. *Do not forget that the last element still has a value.*

Write the following linked list utility routines. Follow MIPS programming conventions in terms of using the stack and registers. Ignore the other methods not implemented. Modify the menu to include the following possibilities.

1. **Insert\_end (15 points):** Insert an element to the linked list at end: the pointer to the linked list is passed in $a0, and the integer value of the new element to be inserted is given in $a1. This utility will request space in memory from the operating system, use it to create a new element, and then insert the new element correctly into the linked list at the end.  Upon return, the value in $v0 =0 if successful, -1 if not. In either case, the pointer to the head of the linked list should be the same as it was before the call.
2. **Delete\_n (15 points)**: Delete an element from the linked list at position n: the pointer to the linked list is passed in $a0, and the position of the element to be deleted is given in $a1. The first element is in position 1, the second element in position 2, etc.  Return value in $v0 =0 if successful, -1 if not.  In either case, the return value in $v1 contains the pointer to the head of the linked list.
3. **Display\_Reverse\_Order (10 points)**: Display the elements of the linked list in reverse order. This means that if the list contains -> 10 -> 20 -> 30 it displays the list in the reverse order 30, 20, 10. This must be a non-recursive subprogram.
4. **Display\_Reverse\_Order\_Recursively (15 points)**: Display the elements of the linked list in reverse order. This must be a recursive subprogram: it must call itself with jal.

Insert\_end:

li $s0, 8

li $v0 9 #syscall 9 (sbrk)

syscall

move $s1, $a0

bne $s1, $zero, devam80

loop:

lw $s0, 0($s1)

add $s1, $s1, 8

bne $s0, $zero, loop

sw $v0, 0($s0)

sw $a1, 4($s0)

Delete\_n:

move $s1, $a0

bne $s1, $zero, devam80

loopD:

lw $s0, 0($s1)

add $s1, $s1, 8

bne $s0, $a1, loopD

sw $zero, 0($s1)

sw $zero, 4($s1)

li $v0, -1

move $v1, $a0

jr $ra

Insert\_n:

li $v0, -1

move $v1, $a0

jr $ra

Delete\_x:

li $v0, -1

move $v1, $a0

jr $ra

nop

Display\_Reverse\_Order:

move $s1, $a0

bne $s1, $zero, devam80

loopX:

lw $s0, 0($s1)

lw $a1, 4($s1)

add $s1, $s1, 8

bne $s0, $zero, loopX

loop2:

li $v0, 1

move $a0, $a1

syscall

add $s1, $s1, -8

b loop