Question 5 (30 points): A bit-vector is a useful compact data structure that is often used in optimizing compilers. In a 32-bit machine each 32-bit word in memory contains 32 bits of a bit vector. Therefore, given the initial address of a bit vector stored in memory, in order to read or write a single bit k , a program has to do the following:

- Compute the address of the word that contains bit k
- Read the word that contain bit k from memory
- Create a mask that contains a 1 in the position of bit k within the word
- Execute the read or write operation using the mask
- Either write the word back to memory (for a write) or return the value of the bit (for a read).

Assume that there is an arbitrary-length bit vector stored in memory. Given a bit position k, once the word that contains this bit is identified, the bits within the word are numbered from the least significative to the most significative as follows:

```
31        27        23        19               7 6 5 4   3 2 1 0
 b b b b   b b b b   b b b b   b b b b  ....   b b b b   b b b b
```

Bit vectors are often associated with the instructions of an assembly program with one bit associated with each instruction. For instance if we would like to record which instructions in an assembly program are basic-block leaders, we can assign a unique id to each instruction — the distance, measured in words, between the instruction and the start of the procedure that contains the instruction, for instance — and use a single bit vector for the whole procedure to record if each instruction is a basic-block leader. The bit vector will have a 1 in the positions corresponding to basic-block leaders and 0 in all other positions.

Given an assembly code where the leaders have been identified, the rule to form basic blocks is as follows:

- each basic block is formed by a leader plus all subsequent instructions up to, but not including, the next leader.

a. (**10 points**) Write a MIPS assembly code for the `ReadBit` function that receives in `$a0` the memory address of the first word of a memory region that contains a bit vector and in `$a1` a bit position k. `ReadBit` returns `$v0 = 1` if the bit at position k is 1 and returns `$v0 = 0` if the bit a position k is 0.

b. (**20 points**) Write a MIPS assembly code for a `FormBasicBlocks` function that reads a bit vector and generates a list of basic-blocks for a procedure. Each bit in this vector is associated with an instruction in an assembly program. A 1 in the bit vector indicates that the corresponding instruction is a basic-block leader. A 0 indicates that it is not. `FormBasicBlocks` receives the following parameters:

   - `$a0`: `vector` is the memory address of the first word of the memory region that contains the bit vector

- `$a1`: `lenght` is the number of instructions in the program
- `$a2`: `code` is the memory address of the first instruction of the procedure
- `$a3`: `blocks` is the memory address of a pre-allocated memory region that will contain the list of basic blocks.

`FormBasicBlocks` will write a list of basic blocks that starts in the memory address given in `blocks`. The first word will be the number of basic blocks in the procedure and then each subsequent pair of words will be the start address and the number of instructions in each basic block. For instance if `$a2 = 0x80000100`, `$a1 = 0x40007500`, and the procedure has three basic block, the first with 3 instruction the second with 4 instruction and the third with one instruction, then after `FormBasicBlocks` returns we will have the following content in memory:

```
  Address    :   Content
0x8000 0118 : 0x4000 0001
0x8000 0114 : 0x4000 7524
0x8000 0110 : 0x0000 0004
0x8000 010c : 0x4000 750c
0x8000 0108 : 0x0000 0003
0x8000 0104 : 0x4000 7500
0x8000 0100 : 0x0000 0003
```

```
# ReadBit receives a, the address of the first word where the bit vector
# is stored and a bit position k. It returns the value of the bit at that position
#
# Parameters:     $a0: a, the memory address of the first word of the bit vector
#                 $a1: k, the position of a bit in the vector
# Return value:   $v0: value of bit k
# Register usage: $t2: address of the word that contains bit k
#                 $t3: word that contains bit k
#                 $t5: mask for bit k
ReadBit:
# Compute address of the word that contains bit k
                srl     $t0, $a1, 5             # $t1 <-- k/32 = number of words before word with k
                sll     $t1, $t0, 2             # $t1 <-- number of bytes before word with k
                addi    $t2, $a0, $t1           # $t2 <-- address of word that has bit k
                lw      $t3, 0($t2)             # $t3 <-- word that has bit k
# Create a mask with a 1 in the position of bit k
                andi    $t1, $a1, 0x001F        # $t1 <-- k%32 (rest of division by 32)
                addi    $t5, $zero, 1           # $t5 <-- 0x0000 0001
                sllv    $t5, $t5, $t1           # $t5 <-- mask with a 1 in position corresponding to bit k
# Extract bit value from word and return
                and     $v0, $t3, $t5           # $v0 <-- value of bit k
                jr      $ra
```

```
# Parameters:   $a0: vector, the memory address of the first word that contains the bit vector
#               $a1: lenght, number of instructions in the program
#               $a2: code, the memory address of the first instruction of the procedure
#               $a3: blocks, the memory address of a pre-allocated memory region that
#                    will contain the list of basic blocks.
# Return value: None
# Register usage: $s0: block_ptr, pointer to list of blocks
#                 $s1: llenght, local copy of lenght
#                 $s2: lvector: local copy of vector
#                 $s3: icounter, counts the number of instructions
#                 $s4: bbcounter, counts the number of basic blocks
#                 $s5: last_leader, position of previous leader in bit vector
#                 $s6: laddress, address of the leader of current basic block
#                 $t0: bb_instr, number of instructions in a basic block
FormBasicBlocks:
                addi    $sp, $sp, -32
                sw      $a0, 0($sp)   # save $a0
                sw      $s0, 4($sp)   # save $s0
                sw      $s1, 4($sp)   # save $s1
                sw      $s2, 4($sp)   # save $s2
                sw      $s3, 4($sp)   # save $s3
                sw      $s4, 4($sp)   # save $s4
                sw      $s5, 4($sp)   # save $s5
                sw      $s6, 4($sp)   # save $s6
#
                addi    $s0, $a3, 4   # block_ptr <-- blocks+4
                add     $s1, $a1, $0  # llenght <-- lenght
                add     $s2, $a0, $0  # lvector <-- vector
                addi    $s3, $0, 1    # icounter <-- 1
                add     $s4, $0, $0   # bbcounter <-- 0
                add     $s5, $0, $0   # last_leader <-- 0
                add     $s6, $a2, $0  # laddress <-- code
NextInstruction:
                add     $a0, $s2, $0  # $a0 <-- lvector
                add     $a1, $s3, $0  # $a1 <-- icounter
                jal     ReadBit
                beqz    $v0, NotLeader
                sub     $t0, $s3, $s5 # bb_instr <-- icounter - last_leader
                sw      $s6, 0($s0)   # *block_ptr <-- laddress
                sw      $t0, 4($s0)   # *(block_ptr+1) <-- bb_instr
                addi    $s0, $s0, 8   # block_ptr++
                add     $s5, $s3, $0  # last_leader <-- icounter
                sll     $t1, $t0, 4   # $t1 <-- 4 x bb_instr
                add     $s6, $s6, $t1 # laddress += (4 x bb_instr)
                addi    $s4, $s4, 1   # bbcounter += 1
NotLeader:      addi    $s3, $s3, 1   # icounter++
                bne     $s3, $s1, NextInstruction  # if icounter != llenght goto next
#
                lw      $a0, 0($sp)   # save $a0
                lw      $s0, 4($sp)   # save $s0
                lw      $s1, 8($sp)   # save $s1
                lw      $s2, 12($sp)   # save $s2
                lw      $s3, 16($sp)   # save $s3
                lw      $s4, 20($sp)   # save $s4
                lw      $s5, 24($sp)   # save $s5
                lw      $s6, 28($sp)   # save $s6
                addi    $sp, $sp, 32
                jal     $ra
```