Question 4 (30 points):

In this question you will write two MIPS assembly functions: `ChangeBranch` and `FixBranches`. While writing both functions you need to follow the MIPS register-saving calling conventions. You are allowed to use any MIPS instructions, including SPIM pseudo instructions. But you are NOT allowed to use any instruction that takes as a parameter a constant that is larger than 16 bits. Even though SPIM allows you to use larger constants, the MIPS assembly does not. We want you to be thinking about the processor and not about the simulator.

1. (**10 points**) Write the MIPS assembly code for a function called `ChangeBranch`. This function changes the binary representation of a branch instruction so that it branches to a new target after `ChangeBranch` is executed.

   Invariants:

   - The new target is still within the range of the branch instruction.
   - The word stored in the address provided is the binary representation of a branch instruction

   Parameters:

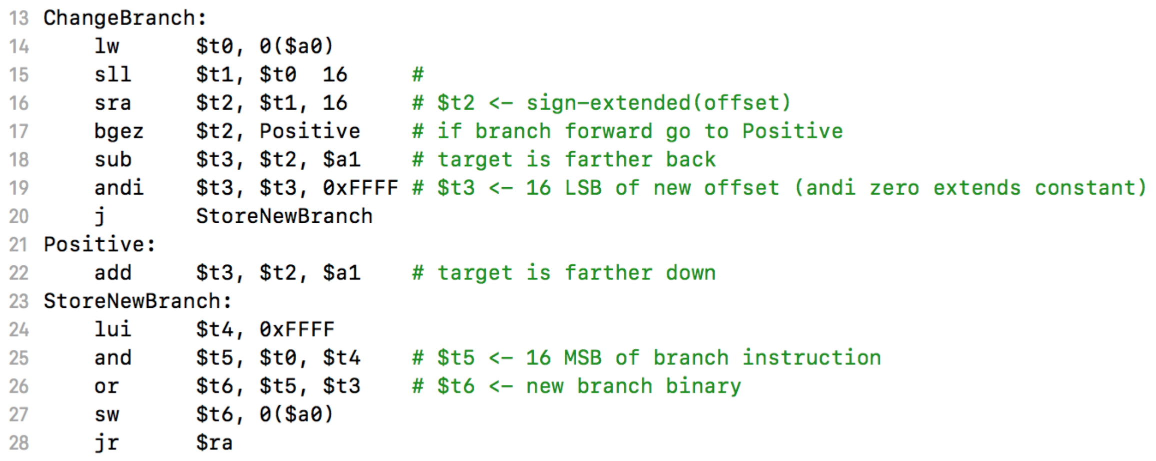
   - `$a0`: address of a branch instruction
   - `$a1`: Number of instructions inserted between the branch instruction and the target of the branch instruction

   Return Value:

   - None

   Side Effect:

   - The branch instruction is changed to reach the new target



```
ChangeBranch:
    lw      $t0, 0($a0)
    sll     $t1, $t0  16     #
    sra     $t2, $t1, 16     # $t2 <- sign-extended(offset)
    bgez    $t2, Positive    # if branch forward go to Positive
    sub     $t3, $t2, $a1    # target is farther back
    andi    $t3, $t3, 0xFFFF # $t3 <- 16 LSB of new offset (andi zero extends constant)
    j       StoreNewBranch
Positive:
    add     $t3, $t2, $a1    # target is farther down
StoreNewBranch:
    lui     $t4, 0xFFFF
    and     $t5, $t0, $t4    # $t5 <- 16 MSB of branch instruction
    or      $t6, $t5, $t3    # $t6 <- new branch binary
    sw      $t6, 0($a0)
    jr      $ra
```

2. (**20 points**) Write MIPS assembly code for a function called `FixBranches` that changes all the branch instructions that needed to be fixed in the binary representation of a MIPS program to which instructions have been inserted. The parameters to `FixBranches` are the address of the first instruction of the MIPS program and the address of a vector of 32-bit integers. Each element of this vector corresponds to a branch instruction in the MIPS program. The number in each element of the vector corresponds to the number of instructions that have been inserted between the branch instruction and its target. The vector has as many elements as the number of branch instructions in the input code.

   `FixBranches` only has to detect and fix the following four types of branches:

Opcode	Branch
`001000`	`beq`
`001001`	`bne`
`001010`	`blez`
`001011`	`bgtz`

   Hint: Note that the four most significant bits of these opcodes are the same for the four types of branches and that this subset of instructions includes all combinations of the two least significant bits of the opcode. This observation should be helpful to decide which instructions are branches in this binary representation.

   Invariants:

   - all the elements in the vector are non-negative integers
   - after the insertion of the instructions, each branch is still within range
   - there is a sentinel value `0xFFFF FFFF` after the last instruction of the program
   - `FixBranches` must invoke `ChangeBranch` to obtain the new binary representation for each branch instruction

   Parameters:

   - `$a0`: Address of the first instruction
   - `$a1`: Address of the vector of integers

   Return Value:

   - None

   Side Effect:

   - All branch instructions are changed according to the specification

   The example shown in Figure 1 is only intended to illustrate how `FixBranches` is supposed to work — Your code for `FixBranches` must work with any valid MIPS binary input. The `xor` instructions that appear in the transformed code were added to the original code. Assume that `FixBranches` is working with a version of the transformed code where new instructions have already been inserted, but the binary representation of the branches is still the same as it was in the original code, and thus they need to be fixed. For this example the vector of integers would be `( 1, 2)` because one instruction was inserted between the branch at line 106 and its target at line 102 and two instructions were inserted between the branch in line 107 and its target at line 112 of the transformed code.

```
0x4000 0000           add  $a0, $0 $0
0x4000 0004  TargetA: lui  $a1, 0xFFFF
0x4000 0008           srl  $t2, $t1, 8
0x4000 000C           sllv $t3, $t1, $t2
0x4000 0010           beq  TargetA
0x4000 0014           bne  $v0, $0, TargetB
0x4000 0018           sllv $t0, $v0, $0
0x4000 001C           nor  $t1, $a0, $t0
0x4000 0020  TargetB: or   $v0, $v0, $t1
0x4000 0028           jr   $ra
```

(a) Original Code

```
0x4000 0000           add  $a0, $0 $0
0x4000 0004  TargetA: lui  $a1, 0xFFFF
0x4000 0008           srl  $t2, $t1, 8
0x4000 000C           xor  $t2, $t1, $t2
0x4000 0010           sllv $t3, $t1, $t2
0x4000 0014           beq  TargetA
0x4000 0018           bne  $t0, $0, TargetB
0x4000 001C           sllv $t0, $t0, $0
0x4000 0020           xor  $t2, $t3, $t1
0x4000 0024           xor  $t0, $t1, $t2
0x4000 001C           nor  $t1, $a0, $t0
0x4000 0020  TargetB: or   $v0, $v0, $t1
0x4000 0028           jr   $ra
```

(b) Transformed Code

Figure 1: Code before and after insertion of instructions.

```
FixBranches:
    add     $sp, $sp, -20
    sw      $ra, 0($sp)
    sw      $s0, 4($sp)
    sw      $s1, 8($sp)
    sw      $s2, 12($sp)
    sw      $s3, 16($sp)
    move    $s0, $a0            # $s0 <- Address of first instruction
    move    $s1, $a1            # $s1 <- Vector address
    lui     $s2, 0xF000         # $s2 <- 0xF000 0000
    lui     $s3, 0x2000         # $s3 <- 4 MSB of branch instructions
    li      $s4, -1             # $s4 <- 0xFFFF FFFF
NextInstr:
    lw      $a0, 0($s0)         # $a0 <- instruction
    beq     $a0, $s4, Done      # if $a0 = setinel go to done
    and     $t0, $s2, $a0       # $t0 <- 4 MSB of instruction
    bne     $t0, $s3, NotBranch # if 4 MSB = 0010 then it is a branch
    lw      $a1, 0($s1)         # $a1 <- number of instructions inserted
    jal     ChangeBranch
    add     $s1, $s1, 4         # $s1 <- $s1 + 4 (next vector position)
NotBranch:
    add     $s0, $s0, 4         # $s0 <- $s0 + 4 (next instruction)
    j       NextInstr
Done:
    lw      $ra, 0($sp)
    lw      $s0, 4($sp)
    lw      $s1, 8($sp)
    lw      $s2, 12($sp)
    lw      $s3, 16($sp)
    add     $sp, $sp, 20
    jr      $ra
```