1. **Pseudo-instructions [15 points]**

You want to create new pseudo-instructions for easy programming.

(a) **[8 points]** You decide to extend the MIPS assembler to provide support for a new rori pseudo-instructions that performs a rotate logical right. Unlike a standard shift logical right operation, the most significant bits of the resulting value come from the least significant bits of the source register. The instruction would look like:

rori $t0, $t1, 3 # rotate right $t1 by 3 into $t0

to indicate that the value in $t1 should be rotated logically right by 3 places and stored into $t0. A sample input and output is shown below:

$t1 0000000000000**11**000000000000000**111**

$t0 **111**0000000000000**11**000000000000000

Write the MIPS instructions into which the assembler might convert the above pseudo-instruction. Use the minimum number of MIPS instructions. (Hint: shift instruction has the following format: sll, register1, register2, immediate)

sll $t2, $t1, 29
slr $t0, $t1, 3
add $t0, $t0, $t2

(b) **[7 points]** You also decide to extend the MIPS assembler to support a method of memory addressing known as memory indirect by way of another pseudo-instruction. In memory indirect addressing, the memory address is dereferenced, as if following a pointer. The instruction load memory indirect (lmi) would look like:

lmi $t0, 8($t1) # $t0 ← Mem[Mem[$t1+8]]

Write the MIPS instructions that the assembler might convert the above pseudo-instruction. Use the minimum number of MIPS instructions.

2. MIPS assembly [15 points]

Please consider the following C and assembly code. The following codes are the C and assembly code for Fibonacci number generator.

```
int fib(int n){
    if(n==0)
        return 0;
    else if (n==1){
        return 1;
    else
        return fib(n-1)+fib(n-2);
}
```

```
0x0000 1000   fib:    addi $sp,$sp, -12       # make room on stack
                      sw $ra 8($sp)           #
                      sw $s0, 4($sp)          # push $s0
                      sw $a0, 0($sp)          # push $a0 (N)
                      bgt $a0,$0, test        # if n>0, test if n=1
                      add $v0, $0, $0         # else fib(0) = 0
                      j rtn                   #
              test:   addi $t0, $0, 1         #
0x0000 1000 + 8*4(decimal)
= 0x0000 1020         bne $t0, $a0, gen       # if n>1, gen
                      add $v0, $0, $t0        # else fib(1) = 1
                      j rtn
              gen:    addi $a0,$a0,-1         # n-1
                      jal fib                 # call fib(n-1)
                      add $s0,$v0, $0         # copy fib(n-1)
                      sub $a0,$a0,1           # n-2
                      jal fib                 #
                      add $v0, $v0, $s0       # fib(n-1)+fib(n-2)
              rtn:    lw $a0, 0($sp)          # pop $a0
```

```
lw $s0, 4($sp)          # pop $s0
lw $ra, 8($sp)          # pop $ra
addi $sp $sp 12         # restore sp
jr $ra
```

Note that bgt is a pseudo-instruction performing branch if one number is greater than the other.

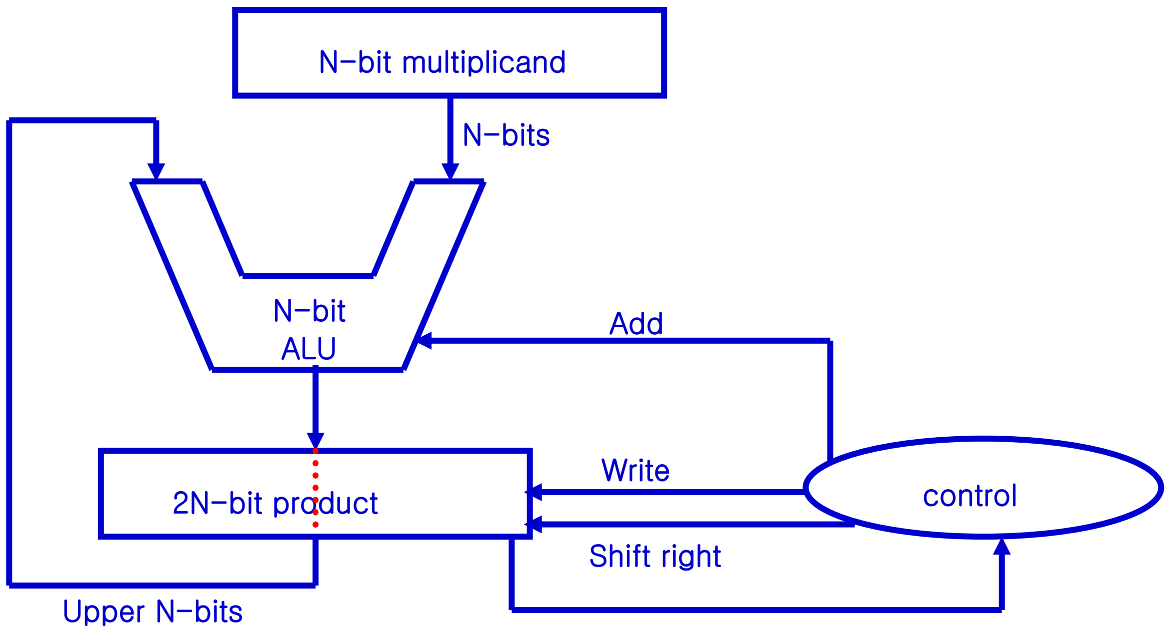
(a) [5 points] fill in the blanks with necessary codes.

(b) [5 points] The address of the label ‘fib’ 0x00001000. Please write down the immediate operand (branch offset) of ‘bne $t0, $a0, gen’ in hex decimal number.

0x0002

(c) [5 points] The address of the label ‘fib’ 0x00001000. Please write down the all possible value for $ra while the code is being executed. (Please exclude the return address of the original caller function of fib.)

3. Multiplication [15 points]



(a) **[5 points]** Determine the minimum number N that can compute two number multiplication for 5*-9.

5

(b) **[10 points]** Show how the above multiplier computes 5*-9. Show the contents of 2N-bit product register each cycle using the number you found for N in (a). Each cycle consists of an addition step and a shift step. Show only the content of the register after the shift is completed. Assume the multiplicand is -9 and the multiplier is 5. Cycle 0 is initial state. Fill as many blanks as necessary.