1.)
a.)

```
sltiu $t1, $t0, -1
bne $t1, $0, SOMEPLACE
addiu $t2, $t1, -1
j EXIT

SOMEPLACE:
add $t2, $t2, $t1

EXIT:
```

-1 = 0xFFFFFFFF so t0 (0x10000000) is less than $t1 = 1

$t0 = 0x10000000 and $t1 is 0x00000001 so jump

$t2 = $t2 + $t1 (1) so $t2 = 0x00000001

b.) How many instructions are executed if N=0, N=1, N=10, N=100, N=1000?
(I assume all are unsigned bits)
a = $s0, c =$s1, i= $s2, and N = $s3 and all variables/arrays are word-sized.

```
for(i=1; i<=N; i++) {
      a[i] = c[i]*16;// Do not use the mul inst
}
------------------------------
li $s2, 1 # Load i=1
j COND # jump to condition

LOOP:

addiu $s2, $s2, 1       # increment i by 1

sll $t0, $s2, 2
addu $t1, $s1, $t0      # c[i] memory address
addu $t2, $s0, $t0      # a[i] memory address

lw $t0, 0($t1)          # load c[i] value
sll $t0, $t0, 4         # c[i] * 16

sw $t0, 0($t2)          # store a[i] = c[i] * 16

COND:

addiu $t0, $s3, 1       # $t0 = N+1
sltu $t0, $s2, $t0      # t0 = 1 if i<(N+1) or i<=N else 0

bne $t0, $zero, LOOP    # if $t0 is 1 (meaning i is less than N+1) go to LOOP

EXIT:                   # Exits once $t0 is 0 (i => N+!1 or i > N)
----------------------------
```

Total instructions:
2 initial, 3 for COND, 7 for LOOP
Equation: 5 + 10 * N
N = 0: 5 instructions
N = 1: 15 N=10: 105
N = 100: 1005 N=1000: 10005

1.)
c.)
score = $a0 grade = $v0. Use conditional branches.

```
if (score >= 90) {grade='A';}
else if (score >= 80) {grade = 'B';}
else if (score >= 70) {grade = 'C';}
else if (score >= 60) {grade = 'D';}
else {grade = 'F';}

------------------------------
li $v0, 'F'             # set grade to F
sltiu $t0, $a0, 60      # if Grade < 60 then $t0 = 1
bne $t0, $zero, EXIT    # return grade = F

li $v0, 'D'             # set grade to D
sltiu $t0, $a0, 70      # if Grade < 70 then $t0 = 1
bne $t0, $zero, EXIT    # return grade = D

li $v0, 'C'             # set grade to C
sltiu $t0, $a0, 80      # if Grade < 80 then $t0 = 1
bne $t0, $zero, EXIT    # return grade = C

li $v0, 'B'             # set grade to B
sltiu $t0, $a0, 90      # if Grade < 80 then $t0 = 1
bne $t0, $zero, EXIT    # return grade = B

li $v0, 'A'             # set grade to A

EXIT:
------------------------------
```

2.)
a.)
Psuedo : not $t1, $t2 // bit-wise invert

Actual : xori $t1, $t2, 0xFFFFFFFF

b.)
Should this instruction produce any exceptions (i.e., can it produce any errors)? If so, what are they and does your implementation handle them?

Psuedo : ttc $t1 // takes the two's complement of $t1

Actual : xori $t1, $t1, 0xFFFFFFFF
addiu $t1, $t1, 1

This program wouldn't have any errors to handle. Neither xori or andiu could cause an arithmetic overflow.

2.)
c.)
MIPS is a RISC load/store architechure. This means that it can only take registers as operands and destination. And does not have the datapath to connect values from memory directly to arithmetic (ALU).

Additionally, MIPS instructions are only 32 bits longs and has 32 bits to address memory with. This means such an add instruction using memory addresses would not be possibly unless its address range is shrunk significantly.

Basically, having this instruction would be needless and significantly complicate the design of the ISA. As MIPS has a way do this through loading, adding registers, and then storing the computed value.

3.)
a.) Submitted C code
b.) Submitted assembly code
c.) Submitted assembly code

Provide three reasonable test cases for your MIPS assembly (inputs and expected outputs) and justify why you have included each one.

In my code I tested three things:

First input	Second input	Output
Don’t	Stop	Don’tStop
Stop		Stop
Believin’	Hold On	Believin’Hold On

What I actually did in the code was add a buffer of null terminators to the Don’t string so that when it concats Stop to it no overflow occurs. As well, this shows adequate usage of the strcat function where the program is the 2 strings fit the buffer.

The Stop test sees if the program will do when concating only a null terminator to a string. This is to make sure that the program will stop concating exactly at the null terminator.

The Believin’ test is used to test inappropriate usage of the program where the buffer does not fit the two strings added together. This overflows the buffer and will modify memory that it shouldn’t. As well, f the strings are placed next to each other, then the output that shows “First input + Second input” will show an incorrect Second input value in my program.

(Another, useful tests could be the first input as empty to test if the progam’s first loop stops exactly at the null terminator of the string.)