

Question 1: Disassembling the Loop Write down the corresponding C code for the assembly snippet shown in class.

Question 2: RISC-V Programming Translate the following code fragment written in C to RISC-V assembly language. You don't have to come up with the most optimal solution, just something that works. You can assume that the necessary declaration and storage allocation for variables and arrays has already been done. Please use descriptive label names to reference variables and branch targets. Include comments when variables are assigned to a register. Furthermore, you can assume that all variables have been allocated into the lower 2K bytes of memory, and that all variables and arrays are C integers, i.e., 32-bit values.

```
1 sum = 0;
2 i = 0;
3
4 while (arr[i] != 0) {
5     if (arr[i] % 2 == 1) {
6         sum = sum + arr[i];
7     }
8     i = i + 1;
9 }
```

Question 3: RISC-V and Memory Tracing Consider the following recursive implementation of GCD:

```
# Addressed RISC-V Code (4-byte instructions)

# -----
# reset
# -----
0:  lui  sp, 0xc0000
4:  addi sp, sp, 0xff0
8:  jal  ra, main
12: j    halt

# -----
# gcd
# -----
16: addi sp, sp, -4    # push stack
20: sw   ra, 0(sp)    # save ra
24: beq  a0, a1, return
28: blt  a0, a1, else
32: sub  a0, a0, a1
36: jal  ra, gcd
40: beq  x0, x0, return

44: else:
44: sub  a1, a1, a0
48: jal  ra, gcd

# -----
# return
# -----
52: return:
52: lw   ra, 0(sp)
56: addi sp, sp, 4
60: ret

# -----
# data
# -----
64: y: .word 0

# -----
# main
# -----
68: main:
68: addi sp, sp, -4
72: sw   ra, 0(sp)
76: li   a0, 300
80: li   a1, 465
84: jal  ra, gcd
88: sw   a0, y
92: lw   ra, 0(sp)
96: addi sp, sp, 4
100: ret
```

After the initial call `gcd(300, 465)` what are the subsequent calls?

When `gcd()` is called the 4th time, the stack has the following contents (each row shows the address, contents in hex and decimal) before `addi, sp, sp-4`

0xBFFFFFF0	0x00000000	0
0xBFFFFFFC	0x0000000C	12
0xBFFFFFF8	0x00000058	88
0xBFFFFFF4	0x00000034	52
sp->0xBFFFFFF0	0x00000028	40

3.1. At what address is `return`?

- ☐ 0
- ☐ 12
- ☐ 88
- ☐ 52
- ☐ 40

3.2. At what address is `else`?

- ☐ 40
- ☐ 44
- ☐ 48
- ☐ 52
- ☐ 56

3.3. At what address is `main`?

- ☐ 68
- ☐ 72
- ☐ 80
- ☐ 84
- ☐ 88

Question 4: Stacks and Procedures (RISC-V)

```
# C-like pseudo-code
#
# int factorial(int n) {
#     if (n == 0) { return 1; }
#     return n * factorial(n - 1);
# }
factorial:
1  addi sp, sp, -16
2  sw  ra, 12(sp)
3  sw  s0, 8(sp)
4  addi s0, sp, 16
5  sw  a0, 4(sp)

6  beq a0, x0, recur
7  addi a0, x0, 1
8  j   end

9 recur:
10 addi a0, a0, -1
11 j   factorial
12 lw  t0, 4(sp)
13 mul a0, t0, a0

14 end:
15 lw  ra, 12(sp)
16 lw  s0, 8(sp)
17 addi sp, sp, 16
18 jr  ra
```

Consider the RISC-V instructions that compute the factorial of a non-negative integer.

You may assume that when the `factorial` procedure is called, register `a0` holds the argument value, and when it returns, register `a0` holds the return value.

These instructions are intended to correctly apply the RISC-V calling convention.

However, the instructions contain **precisely two errors** that prevent the correct result.

Identify the two errors and explain how to fix them.

4.1. Error 1 and fix

4.2. Error 2 and fix