RISC-V Calling Conventions:
- Caller places arguments in registers a0-a7
- Caller transfers control to callee using jal (jump-and-link) to capture the return address in register ra
  - jal ra, label: R[ra] <= pc + 4; pc <= label
  - jal label
- Callee runs, and places results in registers a0 and a1
- Callee transfers control to caller using jr (jump-register) instruction
  - ret: pc <= R[ra]
  - jr ra
  - jalr x0, 0(ra)

Push register xi onto stack
```
addi sp, sp, -4
sw xi, 0(sp)
```

Pop value at top of stack into register xi
```
lw xi, 0(sp)
addi sp, sp, 4
```
Assume 0(sp) holds valid data.

**Stack discipline**: can put anything on the stack, but leave stack the way you found it

- Always save s registers before using them
- Save a and t registers if you will need their value after procedure call returns.
- Always save ra if making nested procedure calls.
Problem 1.

For the following C functions, does the corresponding RISC-V assembly obey the RISC-V calling conventions? If not, rewrite the function so that it does obey the calling conventions.

(A) int function_A(int a, int b) {
    some_other_function();
    return a + b;
}

function_A:
    addi sp, sp, -8
    sw a0, 8(sp)
    sw a1, 4(sp)
    sw ra, 0(sp)
    jal some_other_function
    lw a0, 8(sp)
    lw a1, 4(sp)
    add a0, a0, a1
    lw ra, 0(sp)
    addi sp, sp, 8
    ret

(B) int function_B(int a, int b) {
    int i = foo((a + b) ^ (a - b));
    ret (i + 1) ^ i;
}

function_B:
    addi sp, sp, -4
    sw ra, 0(sp)
    add t0, a0, a1
    sub a0, a0, a1
    xor a0, t0, a0
    jal foo
    addi t0, a0, 1
xor a0, t0, a0
lw ra, 0(sp)
addi sp, sp, 4
ret

(C) int function_C(int x) {
    foo(1, x);
    bar(2, x);
    baz(3, x);
    return 0;
}

function_C:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv a1, a0
    li a0, 1
    jal foo
    li a0, 2
    jal bar
    li a0, 3
    jal baz
    li a0, 0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret

(C) int function_C(int x) {
    foo(1, x);
    bar(2, x);
    baz(3, x);
    return 0;
}

function_C:
    addi sp, sp, -8
    sw ra, 0(sp)
    mv a1, a0
    sw a1, 4(sp)
    li a0, 1
    jal foo
    lw a1, 4(sp)
    li a0, 2
    jal bar
    lw a1, 4(sp)
    li a0, 3
    jal baz
    li a0, 0
    lw ra, 0(sp)
    addi sp, sp, 8
    ret
(D) int function_D(int x, int y) {
    int i = foo(1, 2);
    return i + x + y;
}

function_D:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv s0, a0
    mv s1, a1
    li a0, 1
    li a1, 2
    jal foo
    add a0, a0, s0
    add a0, a0, s1
    lw ra, 0(sp)
    addi sp, sp, 4
    ret

function_D:
    addi sp, sp, -12
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw s1, 8(sp)
    mv s0, a0
    mv s1, a1
    li a0, 1
    li a1, 2
    jal foo
    add a0, a0, s0
    add a0, a0, s1
    lw ra, 0(sp)
    lw s0, 4(sp)
    lw s1, 8(sp)
    addi sp, sp, 12
    ret

yes...no
Problem 2.

Our RISC-V processor does not have a multiply instruction, so we have to do multiplications in software. The C code below shows a recursive implementation of multiplication by repeated addition of unsigned integers (in C, unsigned int denotes an unsigned integer). Ben Bitdiddle has written and hand-compiled this function into the assembly code given below, but the code is not behaving as expected. Find the bugs in Ben’s assembly code and write a correct version.

```
unsigned int mul(unsigned int x,
                 unsigned int y) {
    if (x == 0) {
        return 0;
    } else {
        unsigned int lowbit = x & 1;
        unsigned int p = lowbit? y : 0;
        return p + (mul(x >> 1, y) << 1);
    }
}

Buggy assembly code
mul:
    addi sp, sp, -8
    sw s0, 0(sp)
    sw ra, 4(sp)
    beqz a0, mul_done
    andi s0, a0, 1  // lowbit in s0
    mv t0, zero  // p in t0
    beqz s0, lowbit_zero
    mv t0, a0
lowbit_zero:
    slli a0, a0, 1
    jal mul
    slli a0, a0, 1
    add a0, t0, a0
    lw s0, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 8
mul_done:
    ret
mul:
    beqz a0, mul_done
    addi sp, sp, -8
    sw s0, 0(sp)
    sw ra, 4(sp)
    andi t0, a0, 1  // lowbit in t0
    mv s0, zero  // p in s0
    beqz t0, lowbit_zero
    mv s0, a1
lowbit_zero:
    slli a0, a0, 1
    jal mul
    slli a0, a0, 1
    add a0, s0, a0
    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
mul_done:
    ret
```
Errors (intentional, there may be unintentional ones too...):

1. s0 and ra are saved and restored from different offsets – should be lw ra, 4(sp); lw s0, 0(sp)
2. beqz a0, mul_done should be before sp is decremented (or mul_done label should be moved up 3 instructions)
3. p cannot be in t0 because it's caller-saved and used after call; store lowbit in t0 and p in s0 instead, or use an s1 register, or add code before and after jal mul to save and restore t0.
4. Slli and srli are switched (first one should be slri, second slli)
5. p should come from a1 not a0.
Problem 3.

The following C program computes the log base 2 of its argument. The assembly code for the procedure is shown on the right, along with a stack trace showing the execution of ilog2(10). The execution has been halted just as it’s about to execute the instruction labeled “rtn:” The SP label on the stack shows where the SP is pointing to when execution halted.

```c
/* compute log base 2 of arg */
int ilog2(unsigned x) {
    unsigned y;
    if (x == 0) return 0;
    else {
        /* shift x right by 1 bit */
        y = x >> 1;
        return ilog2(y) + 1;
    }
}
```

(A) Please fill in the values for the two blank locations in the stack trace shown on the right. Please express the values in hex.

**Fill in values (in hex!) for 2 blank locations**

(B) What are the values in a0, s0, sp, and pc at the time execution was halted? Please express the values in hex or write “CAN’T TELL”.

Value in a0: 0x__________ in s0: 0x__________
Value in sp: 0x__________ in pc: 0x__________

a0 = 2, s0 = 2, sp = Can’t tell, pc = 0x250

(C) What was the address of the original ilog2(10) function call?

**Original ilog2(10) address: 0x__________**

0x1104
Problem 4.

You are given an incomplete listing of a C program (shown below) and its translation to RISC-V assembly code (shown on the right):

\[
\begin{align*}
\text{int } \text{fn}(\text{int } x) &= \{} \\
\text{int } \text{lowbit} &= x \& 1; \\
\text{int } \text{rest} &= x >> 1; \\
\text{if } (x == 0) &\text{ return 0; } \\
\text{else } &\text{return ???}; \\
\}\n\]

(C) What is the missing C source corresponding to ??? in the above program?

C source code: ________________________________

\[
\text{fn(rest) + lowbit}
\]
The procedure fn is called from an external procedure and its execution is interrupted just prior to the execution of the instruction tagged ‘yy’. The contents of a region of memory are shown on the left below. If the answer to any of the below problems cannot be deduced from the provided information, write “CAN’T TELL”.

(B) What was the argument to the most recent call to fn?

Most recent argument (HEX): x=_______ 0x11

(C) What is the missing value marked ???? for the contents of location 1D0?

Contents of 1D0 (HEX): _______CAN’T TELL

(D) What is the hex address of the instruction tagged rtn:? 

Address of rtn (HEX): _______0x50

(E) What was the argument to the first recursive call to fn?

First recursive call argument (HEX): x=_______ 0x23

(F) What is the hex address of the jal instruction that called fn originally?

Address of original call (HEX): _______0xC0

(G) What were the contents of s1 at the time of the original call?

Original s1 contents (HEX): _______0x22

(H) What value will be returned to the original caller if the value of a0 at the time of the original call was 0x47?

Return value for original call (HEX): _______0x4

counts the number of 1’s in original number