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.
Note: A small subset of essential problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1.

For the following Python 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) `def function_A(a, 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`

yes ... no

(B) `def function_B(a, b):
    i = foo((a + b)^(a - b))
    return (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) def function_C(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

(D) def function_D(x, y):
    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
Problem 2.

Write assembly program that computes square of the sum of two numbers (i.e. \( \text{squareSum}(x,y) = (x + y)^2 \)) and follows RISC-V calling convention. Note that in your assembly code you have to call assembly procedures for `mult` and `sum`. They are not provided to you, but they are fully functional and obey the calling convention.

**Python code for square of the sum of two numbers**

```python
def squareSum(x, y):
    return mult(sum(x, y), sum(x, y))
```

# start of the assembly code

`squareSum:`
Problem 3. ★

Our RISC-V processor does not have a multiply instruction, so we have to do multiplications in software. The Python code below shows a recursive implementation of multiplication by repeated addition of unsigned integers. 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.

<table>
<thead>
<tr>
<th>Python for unsigned multiplication</th>
<th>Buggy assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td># x, y are unsigned integers</td>
<td>mul:</td>
</tr>
<tr>
<td>def mul(x, y):</td>
<td>addi sp, sp, -8</td>
</tr>
<tr>
<td>if x == 0:</td>
<td>sw s0, 0(sp)</td>
</tr>
<tr>
<td>return 0</td>
<td>sw ra, 4(sp)</td>
</tr>
<tr>
<td>else :</td>
<td>beqz a0, mul_done</td>
</tr>
<tr>
<td>lowbit = x &amp; 1</td>
<td>andi s0, a0, 1</td>
</tr>
<tr>
<td>p = y if lowbit else 0</td>
<td>// lowbit in s0</td>
</tr>
<tr>
<td>return p + (mul(x &gt;&gt; 1, y)</td>
<td>mv t0, zero</td>
</tr>
<tr>
<td>)&lt;&lt; 1)</td>
<td>// p in t0</td>
</tr>
<tr>
<td></td>
<td>beqz s0, lowbit_zero</td>
</tr>
<tr>
<td></td>
<td>mv t0, a0</td>
</tr>
<tr>
<td></td>
<td>lowbit_zero:</td>
</tr>
<tr>
<td></td>
<td>slli a0, a0, 1</td>
</tr>
<tr>
<td></td>
<td>jal mul</td>
</tr>
<tr>
<td></td>
<td>srli a0, a0, 1</td>
</tr>
<tr>
<td></td>
<td>add a0, t0, a0</td>
</tr>
<tr>
<td></td>
<td>lw s0, 4(sp)</td>
</tr>
<tr>
<td></td>
<td>lw ra, 0(sp)</td>
</tr>
<tr>
<td></td>
<td>addi sp, sp, 8</td>
</tr>
<tr>
<td></td>
<td>mul_done:</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>