1 Calling Convention

You are given a function `mystery_func` that violates calling convention. It takes in two arguments: `a0, a1`. The return value is stored in `a0`. It calls two functions, `function_1` and `function_2`, which you may assume follow calling convention.

```
mystery_func:
    # Prologue
    addi sp sp -8
    sw s0 0(sp)
    sw s1 4(sp)

    mv s0 a0
    mv s1 a1
    slli t0 a1 2
    add a0 a0 t0

    jal ra function_1

    mv s2 a0
    add a0 a0 t0
    addi a1 x0 23

    jal ra function_2

    add a0 s0 s1
    add a0 a0 s2

    # Epilogue
    lw s0 0(sp)
    lw s1 4(sp)
    addi sp sp 8

    jr ra
```

Identify all calling convention errors in `mystery_func`. For each error, briefly explain the issue that it creates, and how it can be fixed.
2 RISC-V Translation

2.1 In this question, we will be translating between RISC-V code and binary/hexadecimal values.

Translate the following RISC-V instructions into binary and hexadecimal notations.

1. \texttt{addi s1 x0 -24} = \texttt{0b______________________________} = \texttt{0x____________}
2. \texttt{sh s1 4(t1)} = \texttt{0b______________________________} = \texttt{0x____________}

2.2 In this question, we will be translating between RISC-V code and binary/hexadecimal values.

Translate the following hexadecimal values into the relevant RISC-V instruction.

1. \texttt{0x234554B7} = _________________________________
2. \texttt{0xFE050CE3} = _________________________________
3 RISC-V Addressing

We have several addressing modes to access memory (immediate not listed):

1. Base displacement addressing adds an immediate to a register value to create a data memory address (used for lw, lh, sw, sb).

2. PC-relative addressing uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an instruction address (used by branch and jump instructions).

3. Register Addressing uses the value in a register as an instruction address. For instance, jalr, jr, and ret, where jr and ret are just pseudoinstructions that get converted to jalr.

3.1 What is the range of 32-bit instructions that can be reached from the current PC using a branch instruction? Recall that RISC-V supports 16b instructions via an extension.

3.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

3.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you’ll need your RISC-V reference sheet!). Each field refers to a different block of the instruction encoding.

1. `0x002cff00`: loop: add t1, t2, t0 |________|________|________|________|________|__0x33__|
2. `0x002cff04`: jal ra, foo |__________________________|_________________|__0x6F__|
3. `0x002cff08`: bne t1, zero, loop |________|________|________|________|________|__0x63__|
4. ...
5. `0x002cff2c`: foo: jr ra ra = ________________
4 CALL

The following is a diagram of the CALL stack detailing how C programs are built and executed by machines.

Source files: foo.c

C Preprocessor

Intermediate files: foo.i, foo.ii

Compiler

Assembly files: foo.s

Assembler

Object files: foo.o

Linker

Libs

Executable files: foo.out

Loader (OS)

Memory

Please answer true/false to the following questions, and include an explanation.

4.1 The compiler may output pseudoinstructions.

4.2 The main job of the assembler is to perform optimizations on the assembly code.

4.3 The object files produced by the assembler are only moved, not edited, by the linker.

4.4 The destination of all jump instructions is completely determined after linking.
Please answer the following questions with a short answer.

4.5 How many passes through the code does the Assembler have to make? Why?

4.6 Which step in CALL resolves relative addressing? Absolute addressing?

4.7 Describe the six main parts of the object files outputted by the Assembler (Header, Text, Data, Relocation Table, Symbol Table, Debugging Information).