CS 61C — RISC-V, CALL

Summer 2023 — Discussion 5

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 The compiler may output pseudoinstructions.

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

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

1.4 The destination of all jump instructions is completely determined after linking.

2 Calling Convention Again

2.1 Write a function `sumSquare` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

For this problem, you are given a RISC-V function called `square` that takes in a single integer and returns its square.

First, let's implement the meat of the function: the squaring and summing. We will be abiding by the caller/callee convention, so in what register can we expect the parameter `n`? What registers should hold `square`'s parameter and return value? In what register should we place the return value of `sumSquare`?

2.2 Since `sumSquare` is the callee, we need to ensure that it is not overriding any registers that the caller may use. Given your implementation above, write a prologue and epilogue to account for the registers you used.

3 Translation

3.1 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

```
addi  s1  x0  -24  =      0b______________________________   =  0x___________
sh  s1  4(t1)      =      0b______________________________   =  0x___________
```

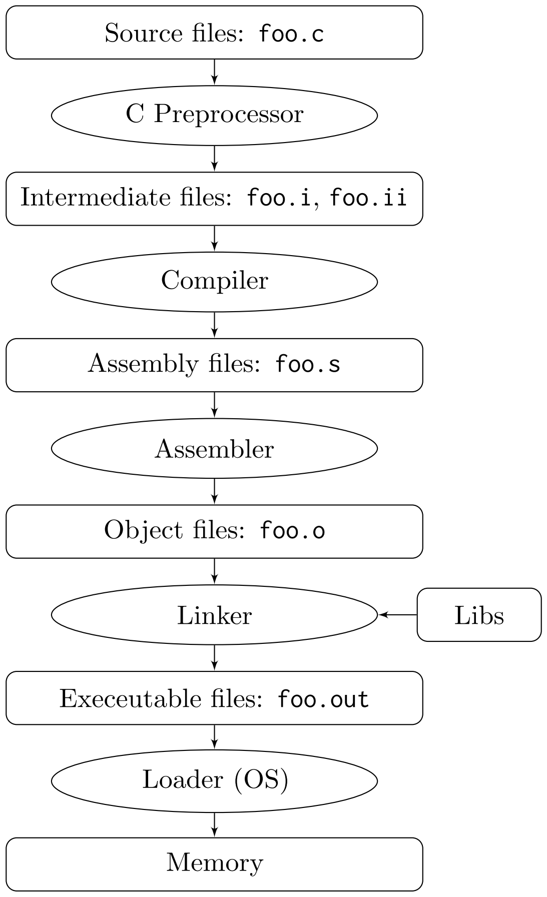
3.2 In this question, we will be translating between RISC-V code and binary/hexadecimal values.

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

```
0x234554B7 =  _________________________________
0xFE050CE3 = _________________________________
```

4 CALL

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



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

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

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

5 Assembling RISC-V

Let's say that we have a C program that has a single function `sum` that computes the sum of an array. We've compiled it to RISC-V, but we haven't assembled the RISC-V code yet.

```
1  .import print.s             # print.s is a different file
2  .data
3          array: .word 1 2 3 4 5
4  .text
5  .globl sum
6  sum:    la t0, array
7          li t1, 4
8          mv t2, x0
9  loop:   blt t1, x0, end
10         slli t3, t1, 2
11         add t3, t0, t3
12         lw t3, 0(t3)
13         add t2, t2, t3
14         addi t1, t1, -1
15         j loop
16 end:    mv a0, t2
17         jal ra, print_int   # Defined in print.s
```

5.1 Which lines contain pseudoinstructions that need to be converted to regular RISC-V instructions?

5.2 For the branch/jump instructions, which labels will be resolved in the first pass of the assembler? The second? Assume that the code is processed from top to bottom.

Let's assume that the code for this program starts at address `0x00061C00`. The code below is labelled with its address in memory (think: why is there a jump of 8 between the first and second lines?).

```
0x00061C00: sum:    la t0, array
0x00061C08:         li t1, 4
0x00061C0C:         mv t2, x0
0x00061C10: loop:   blt t1, x0, end
0x00061C14:         slli t3, t1, 2
0x00061C18:         add t3, t0, t3
0x00061C1C:         lw t3, 0(t3)
0x00061C20:         add t2, t2, t3
0x00061C24:         addi t1, t1, -1
0x00061C28:         j loop
0x00061C2C: end:    mv a0, t2
```

```
0x00061C30:         jal ra, print_int
```

5.3 What is in the symbol table after the assembler makes its passes?

5.4 What's contained in the relocation table?