CS 61C Summer 2023 — RISC-V, CALL — 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.

True. It is the job of the assembler to replace these pseudoinstructions.

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

False. That's the job of the compiler. The assembler is primarily responsible for replacing pseudoinstructions and resolving offsets.

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

False. The linker needs to relocate all absolute address references.

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

False. Jumps relative to registers (i.e. from jalr instructions) are only known at run-time. Otherwise, you would not be able to call a function from different call sites.

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`?

```
      add  s0, a0, x0   # Set s0 equal to the parameter n
      add  s1, x0, x0   # Set s1 (accumulator) equal to 0
loop: beq  s0, x0, end  # Branch if s0 reaches 0
      add  a0, s0, x0   # Set a0 to the value in s0, setting up
                        # args for call to function square
      jal  ra, square   # Call the function square
      add  s1, s1, a0   # Add the returned value into s1
```

```
          addi s0, s0, -1   # Decrement s0 by 1
          jal  x0, loop     # Jump back to the loop label
     end: add  a0, s1, x0   # Set a0 to s1 (desired return value)
```

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.

```
prologue: addi sp, sp -12   # Make space for 3 words on the stack
          sw   ra, 0(sp)    # Store the return address
          sw   s0, 4(sp)    # Store register s0
          sw   s1, 8(sp)    # Store register s1


epilogue: lw   ra, 0(sp)    # Restore ra
          lw   s0, 4(sp)    # Restore s0
          lw   s1, 8(sp)    # Restore s1
          addi sp, sp, 12   # Free space on the stack for the 3 words
          jr   ra           # Return to the caller
```

Note that `ra` is stored in the prologue and epilogue even though it is a call*er*-saved register. This is because if we call multiple functions within the body of `sumSquare`, we'd need to save `ra` to the stack on every call, which would be redundant — we might as well save it in the prologue and restore it in the epilogue along with the callee-saved registers. For this reason, in functions that don't call other functions, it is generally safe to refrain from saving/restoring `ra` in the prologue/epilogue as long as nothing else is overwriting it.

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___________
```

For this question, use the reference sheet to get information about the instructions and convert them to binary representation. One thing that helps is splitting the parsing into parts. For question 1:

```
addi  s1  x0  x4:
rd= s1 = 0b01001
rs1 = x0 = 0b00000
immediate = -24 = 0b1111 1110 1000
opcode = 001 0011
funct3 = 000
Bringing it together - 0b1111 1110 1000 0000 0000 0100 1001 0011 = 0xFE800493
```

For question 2, with a similar method we get the answer: 0b0000 0000 1001 0011 0001 0010 0010 0011 = 0x00931223

$\boxed{3.2}$ 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 = _________________________________
```

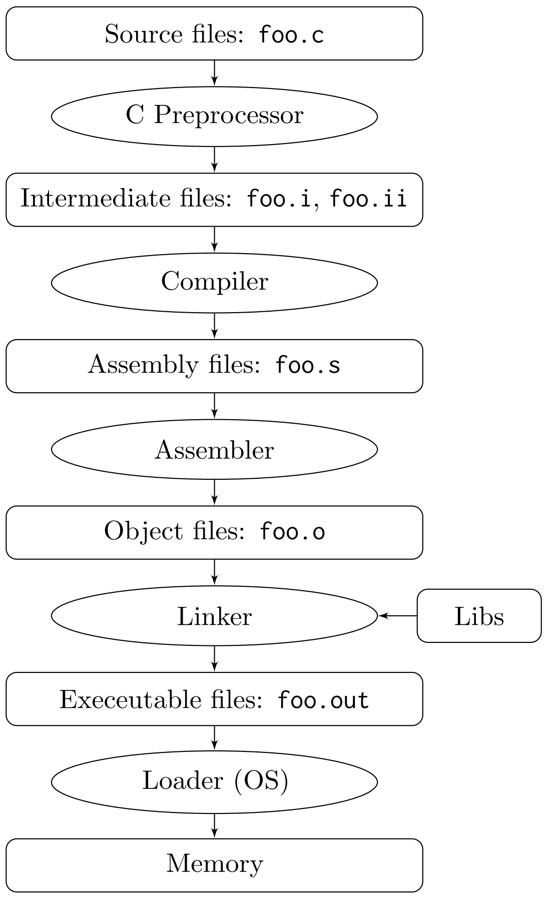
For the reverse conversion, we want to first determine the instruction type. In order to do that, we first look at the opcode (and then func3/func7 if necessary). Let's start with the first one:

```
0x234554B7 = 0b0010 0011 0100 0101 0101 0100 1011 0111, the opcode is always the last 7 bits so opcode = 011 0111, which corresponds to the operation lui!
Looking at lui, we can see that the first 20 bits correspond to the immediate, and the next 5 ones are the register ones. So:
0b0010 0011 0100 0101 0101 = 0x23455 So, the immediate input was indeed 0x23455.
Looking at the next 5 bits, they must be the rd register values. So, we have
rd = 0b01001
That is equal to 9, which is the register x9 = s1. Thus, overall we have
lui s1 0x23455
```

For question 2, with a similar approach: beq a0, x0, -8

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?

Two: The first finds all the label addresses, and the second resolves forward references while using these label addresses.

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

Assembler, Linker

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

- Header: Sizes and positions of the other parts
- Text: The machine code
- Data: Binary representation of any data in the source file
- Relocation Table: Identifies lines of code that need to be "handled" by the Linker (jumps to external labels (e.g. lib files), references to static data)
- Symbol Table: List of file labels and data that can be referenced across files
- Debugging Information: Additional information for debuggers

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?

6, 7, 8, 15, 16.

`la` becomes the `auipc` and `addi` instructions.

`li` becomes an `addi` instruction here (e.g. `li t0, 4` → `addi t0, x0, 4`).

`mv` becomes an `addi` instruction (i.e. `mv rd, rs` → `addi rd, rs, 0`).

`j` becomes a `jal` instruction (e.g. `j loop` → `jal x0, loop`).

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.

`loop` (in `j loop`) will be resolved in the first pass since it's a backward reference. Since the assembler will have kept note of where `end` is in the first pass, it will resolve `end` in `blt t1, x0, end` in the second pass. (`print_int` in `jal ra, print_int` will be resolved by the Linker.)

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?).

There's a jump of 8 because `la` is a pseudoinstruction that gets translated to two regular RISC-V instructions!

```
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?

Label	**Address**
sum	`0x00061C00`

The only labels that are put in the symbol table are the ones which external programs can reference, declared using the `.globl` directive.

5.4 What's contained in the relocation table?

`array` and `print_int`.

Since `array` is defined in the static portion of memory, there's no way the assembler could know where it will be located (relative to the program counter) until the program actually executes. Recall that the static portion of memory is above the code portion of memory. Since we haven't linked other files with this one yet (that's done in the linker phase!), we don't know how much code we'll have, so we don't know where the static portion of memory will begin! Also, other files may declare items in static memory, and the assembler won't know how these are specifically ordered when the program is finally loaded.

Similarly, `print_int` is defined in a different file, so the assembler doesn't know where it will be in the final executable. That will be decided in the linking stage.