CS 61C — CALL, RISC-V Procedures

Spring 2022 — 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 generate optimized machine 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.

1.5 After calling a function and having that function return, the `t` registers may have been changed during the execution of the function, while `a` registers cannot.

1.6 Let `a0` point to the start of an array `x`. `lw s0, 4(a0)` will always load `x[1]` into `s0`.

1.7 Assuming no compiler or operating system protections, it is possible to have the code jump to data stored at `0(a0)` (offset 0 from the value in register `a0`) and execute instructions from there.

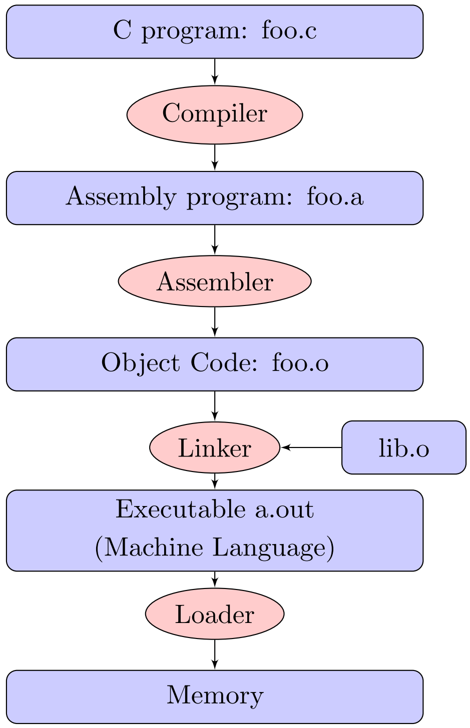
1.8 Assuming integers are 4 bytes, adding the ASCII character `'d'` to the address of an integer array would get you the element at index 25 of that array (assuming the array is large enough).

1.9 `jalr` is a shorthand expression for a `jal` that jumps to the specified label and does not store a return address anywhere.

1.10 Calling `j label` does the exact same thing as calling `jal label`.

2 CALL

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



2.1 What is the Stored Program concept and what does it enable us to do?

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

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

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

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

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

3.2 For the branch/jump instructions, which labels will be resolved in the first pass of the assembler? The second?

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?).

```
1  0x00061C00: sum:    la t0, array
2  0x00061C08:         li t1, 4
3  0x00061C0C:         mv t2, x0
4  0x00061C10: loop:   blt t1, x0, end
5  0x00061C14:         slli t3, t1, 2
6  0x00061C18:         add t3, t0, t3
7  0x00061C1C:         lw t3, 0(t3)
8  0x00061C20:         add t2, t2, t3
9  0x00061C24:         addi t1, t1, -1
10 0x00061C28:         j loop
11 0x00061C2C: end:    mv a0, t2
```

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

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

3.4 What's contained in the relocation table?

4 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 memory address (used for lw, lb, sw, sb).
2. PC-relative addressing uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an address (used by branch and jump instructions).
3. Register Addressing uses the value in a register as a memory address. For instance, `jalr`, `jr`, and `ret`, where `jr` and `ret` are just pseudoinstructions that get converted to `jalr`.

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

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

4.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!).

```
0x002cff00: loop: add t1, t2, t0         |________|________|________|________|________|__0x33__|
0x002cff04:       jal ra, foo             |__________________________|_________________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |________|________|________|________|________|__0x63__|
...
0x002cff2c: foo:  jr ra                 ra = ______________________________
```

5 Writing RISC-V Functions

5.1 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`?

5.2 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.