CS 61C Fall 2020 — CALL, RISC-V Procedures

Discussion 5: Sept 28, 2020

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.

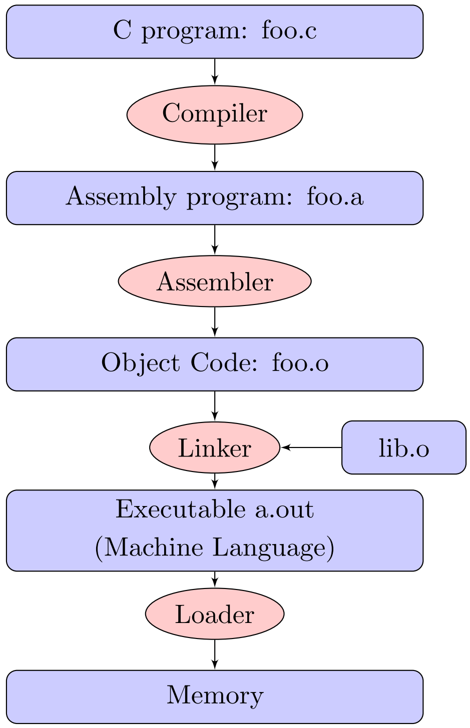
1.2 The main job of the assembler is to generate optimized machine code.

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.

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.

```
.import print.s            # print.s is a different file
.data
array: .word 1 2 3 4 5
.text
sum:    la t0, array
        li t1, 4
        mv t2, x0
loop:   blt t1, x0, end
        slli t3, t1, 2
        add t3, t0, t3
        lw t3, 0(t3)
        add t2, t2, t3
        addi t1, t1, -1
        j loop
end:    mv a0, t2
        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?).

```
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
```

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 = ______________________________
```