CS 61C — CALL, RISC-V Procedures

Fall 2020 — 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.

True , it's the job of assembler to replace pseudoinstructions

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

True False, that's the job of complier The assembler is responsible for replacing pseudoinstructions and resolving offsets

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

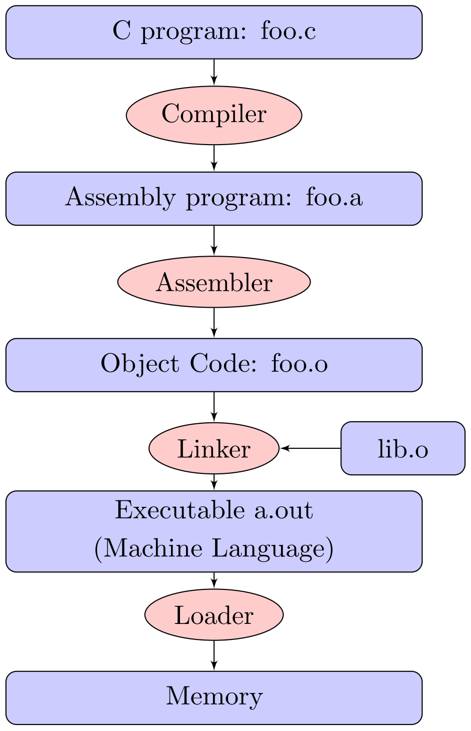
False The linker needs to relocate all absolute address references

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

True False, Jump relative to register are only known at run-time

2 CALL C (ompiling. Assembling. Linking, Loading)

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?

It's the idea that instructions are data so we can write programs to manipulate other programs without modifying the physical hardware

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

2, to solve "forward reference" problem
one to find all label address
the other to convert all instructions

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

Header: size and position of other pieces of the object file
Text: machine code Data: static data Relocation Table: code that need to be fixed later
Symbol Table: list of label and data that can be referenced
DI: standard format

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

Assembling Linking

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?

5 . 6 . 7, 15. ~~16~~
14

la → auipc, addi j → jal
li → addi
mv → addi
(j loop → jal x0,loop)

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

first: end second: loop

loop will be resolved in the first pass since it's a backword refrence
end will be resolved in the second pass

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?).

Header

la is translated to 2 Risc V instructions

```
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
```

```
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?

Sum : 0x00061C00 loop: 0x00061C10 end: 0x00061C2C

3.4 What's contained in the relocation table?

0x00061C30 array and print_int

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?

$[PC - 4096 \times 4, PC + 4094 \times 4]$

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

$PC + (2^{12} - 1) \times 4$

0010 1100
0100
00101000

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!).

```
                         rd  rs1 rs2
0x002cff00: loop: add t1, t2, t0        |_0x0____|_0x5____|_0x7____|_0x0____|_0x6____|__0x33__|
0x002cff04:       jal ra, foo           |_0____0x14____0___0_____|________1__________|__0x6F__|
0x002cff08:       bne t1, zero, loop    |_1 1 0x3F|____0___|___6____|____1___|0xC 1 1_|__0x63__|
...                                 rs1 rs2     pc-8
0x002cff2c: foo:  jr ra                 ra = __0x002cff08________________
```

0x002cff2c − 0x002cff04

Jal	imm	discard	imm[0] rd	opcode
	0 | 0x14 | 0 | 0		00001	110111

Jal saves pc+4 in rd

0000000000 _ _ _ _ _ _ _ 1 0 0 0

0000 0010 100 | 0 0000 0000 | 0000 | 1101111

rd opco

jal ra, foo