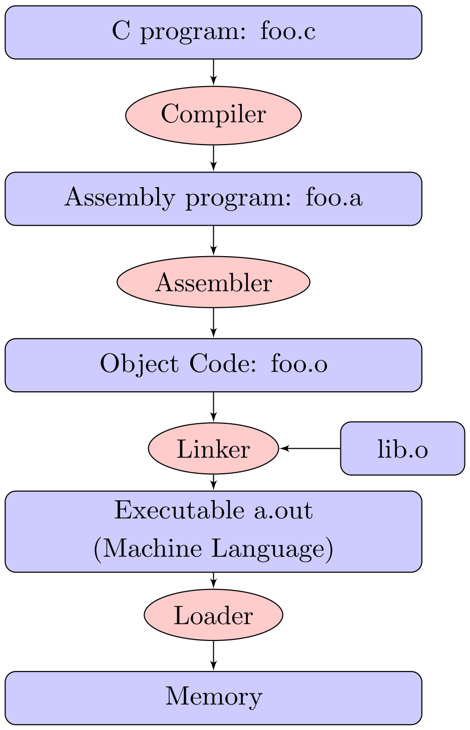
CS 61C — Fall 2019

CALL, RISC-V Procedures

Discussion 5: September 30, 2019

1 CALL

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



1.1 What is the Stored Program concept and what does it enable us to do? Instructions = Data

It is the idea that instructions are just the same as data, and we can treat them as such. This enables us to write programs that can manipulate other programs! Program may modify other programs,

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

Two, one to find all the label addresses and another to convert all instructions while resolving any forward references using the collected label addresses.

1.3 Describe the six main parts of the object files outputed by the Assembler (Header, Text, Data, Relocation Table, Symbol Table, Debugging Information). 6 total items

- Header: Size and position of other parts
- Text: The machine code
- Data: Binary representation of any data in the source file (think static memory)

- Relocation Table: Identifies lines of code that need to be "handled" by Linker (jumps to external labels (e.g. lib files), reference to static data)
- Symbol Table: List of file labels and data that can be referenced across files
- Debugging Information: Additional information for debuggers (-g flag)

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

Assembler, Linker

↳ since we know where in our code we want to jump to, it is just a static offset in our code thus we can calculate this in the Assembler. For absolute addressing we will not know where libraries, etc will be stored till the linker.

2 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          addi 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
```

← pseudo for auipc + addi (we want to get location which should be wrt the current location plus a well defined offset)

← pseudo for addi t1, t1, 4 since 4 can fit here. you would need lui + addi if it is larger than 0x7FF. or smaller than 0x800

← pseudo

why 0x7FF & 0x800? since we have 12 bits in an addi immediate AND we sign extend since it is twos comp. with 12 bits the largest value is 0x7FF + the smallest value is 0x800

← pseudo jal x0, loop Laka. disregard next instructions address.

← pseudo! e.g. add a0, t2, x0 All we want is to move the value of t2 into a0

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

5, 6, 7, 14, 15.

`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`).

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

`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.

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!

auipc and addi

```
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:         addi 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
```

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

Label	Address
sum	`0x00061C00`

or

Label	Address
sum	`0x00061C00`
loop	`0x00061C10`
end	`0x00061C2C`

Normally, one would assume that both the `loop` and `end` labels would be included in the symbol table—and that's perfectly valid answer given that an isolated assembler would have no way to tell the difference between the three labels.

However, we stated at the beginning of this problem that this file is compiled from C code. If we have a integrated compiler, assembler, and linker (e.g. `gcc`), then it will know from the compilation phase which labels are for functions and which ones aren't. As such, it will only put the function labels in the symbol table since those are the only ones that other files can reference.

2.4 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. We 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.

3 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`.

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

12 bits to address byte addresses
$\downarrow$
$2^{12} \cdot 2 \leftarrow$ convert from byte addr to half-word addr (aka the implicit zero)

branch 12-bit imm represents half-words

The immediate field of the branch instruction is 12 bits. This field only references addresses that are divisible by 2, so the immediate is multiplied by 2 before being added to the PC. Therefore, the branch immediate can move PC in the range of $[-2^{12}, 2^{12} - 1]$ bytes. If we're in a version of RISC-V that has 2-byte instructions, then this corresponds to a range of $[-2^{-11}, 2^{11} - 1]$ instructions. The instructions we use, however, are 4 bytes so they reside at addresses that are divisible by 4 not 2. Therefore, we can only reference half as many 4-byte instructions as before, and the range of 4-byte instructions is $[-2^{10}, 2^{10} - 1]$

So we have 2^{13} addresses, so if we use twos comp, we will have $[-2^{12}, 2^{12}-1]$ bytes which we can address from current PC. If we restrict this to halfwords we will divide this by 2 since two bytes fit in a halfword thus $[-2^{11}, 2^{11}-1]$. This ~~[illegible]~~ derivation forward $[-2^{10}, 2^{10}-1]$

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

The immediate field of the jump instruction is 20 bits. Similar to above, this immediate is multiplied by 2 before added to the PC to get the final address. Since the immediate is signed, we have a range of $[-2^{20}, 2^{20} - 1]$ bytes, or $[-2^{19}, 2^{19} - 1]$ 2-byte instructions. As we actually want the number of 4-byte instructions, we actually can reference those within $[-2^{18}, 2^{18} - 1]$ instructions of the current PC.

Jumps have 20 bits w/ implicit zero thus 21 bits in reality so we can address $[-2^{20}, 2^{20}-1]$ bytes or $[-2^{19}, 2^{19}-1]$ half words or $[-2^{18}, 2^{18}-1]$ words.

3.3 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!).

zero = 0
t1 = 6
t2 = 7
t0 = 5
ra = 1

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

func7	rs2	rs1	func3	rd	opcode

```
0x002cff00: loop: add t1, t2, t0          |   0    |   5   |   7   |  0  |  6  |  0x33  |
0x002cff04:       jal ra, foo             |  0  |  0x14  |  0  |     0     |    1     |  0x6F  |
0x002cff08:       bne t1, zero, loop      |  1  |  0x3F  |  0  |  6  |  1  |  0xC  |  1  |  0x63  |
...
```

jal: 20 | 10:1 | 11 | 19:12 | rd

bne: 12 | 10:5 | rs2 | rs1 | func3 | 4:1 | 11

0x2C
② Jal imm = $\underline{-0x04}$
So: 0x28

$$\overset{20}{0}\,\underbrace{0000\ 0000}_{19:12}\ 0\,000\ \underbrace{0\ 001}_{0x\ 1}\ \underbrace{0\ 100}_{4}\ 0$$

0x0 0x0

0 ← this is cut out! It is implicit in the instruction

```
0x002cff2c: foo:  jr ra                    ra = 0x002cff08
```