CS 61C Fall 2021

CALL, RISC-V Procedures

Discussion 5: September 27, 2021

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 generate optimized machine 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.

1.1 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.

False. `a0` and `a1` registers are often used to store the return value from a function, so the function can set their values to the its return values before returning.

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

False. This only holds for data types that are four bytes wide, like `int` or `float`. For data-types like `char` that are only one byte wide, `4(a0)` is too large of an offset

to return the element at index 1, and will instead return a `char` further down the array (or some other data beyond the array, depending on the array length).

1.3 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.

True. If your compiler/OS allows it (some do not, for security reasons), it is possible for your code to jump to and execute instructions passed into the program via an array. Conversely, it's also possible for your code to treat itself as normal data (search up self-modifying code if you want to see more details).

1.4 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).

True. There is no fundamental difference between integers, strings, and memory addresses in RISC-V (they're all bags of bits), so it's possible to manipulate data in this way. (We don't recommend it, though).

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

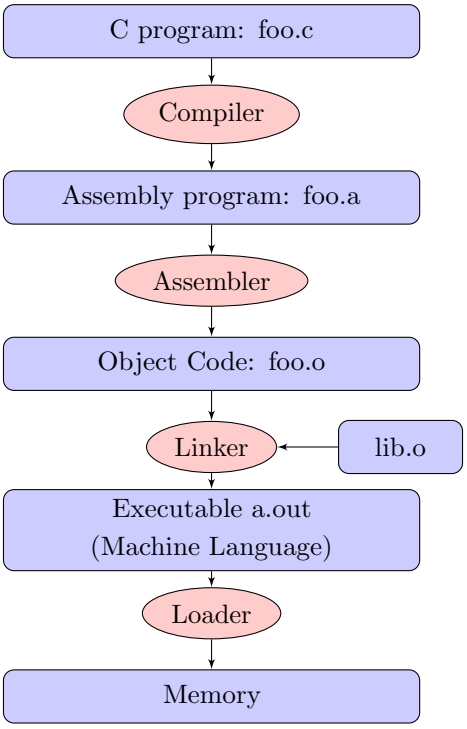
False. `j label` is a pseudo-instruction for `jal x0, label`. `jalr` is used to return to the memory address specified in the second argument. Keep in mind that `jal` jumps to a label (which is translated into an immediate by the assembler), whereas `jalr` jumps to an address stored in a register, which is set at runtime.

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

False. As from the previous problem, `j label` is short for `jal x0, label` — since it's writing the return address to `x0`, it's effectively discarding it since we have no need to jump back to the original PC. `jal label` is short for `jal ra, 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?

It is the idea that instructions are really just data, so we can treat them as such. This enables us to write programs that can manipulate other programs without modifying the physical hardware!

2.2 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 using these label addresses to resolve any forward references.

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: 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

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

Assembler, Linker

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, 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`).

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

Note: This answer assumes that the assembler goes from top to bottom. The answer changes if it goes in reverse.

`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!

```
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
12 0x00061C30:         jal ra, print_int
```

3.3 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.

3.4 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.

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?

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. Since it is signed, the branch immediate can therefore move the PC in the range of $[-2^{12}, 2^{12} - 2]$ 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 2-byte instructions, and the range of 4-byte instructions is $[-2^{10}, 2^{10} - 1]$

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

The immediate field of the jal instruction is 20 bits, while that of the jalr instruction is only 12 bits, so jal can reach a wider range of instructions. Similar to above, this 20-bit immediate is multiplied by 2 and added to the PC to get the final address. Since the immediate is signed, we have a range of $[-2^{20}, 2^{20} - 2]$ bytes, or $[-2^{19}, 2^{19} - 1]$ 2-byte instructions. As we actually want the number of 4-byte instructions, we can reference those within $[-2^{18}, 2^{18} - 1]$ instructions of the current PC.

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!).

```
1  0x002cff00: loop: add t1, t2, t0          |________|________|________|________|________|__0x33__|
2  0x002cff04:       jal ra, foo             |_____________________________|_________________|__0x6F__|
3  0x002cff08:       bne t1, zero, loop      |________|________|________|________|________|__0x63__|
4  ...
5  0x002cff2c: foo:  jr ra                  ra = ____________________________________
```

```
1  0x002cff00: loop: add t1, t2, t0       |   0   |   5   |   7    |  0  |  6  |  0x33  | → 0x00538333
2  0x002cff04:       jal ra, foo          |  0  |  0x14  |  0  |   0   |   1   |  0x6F  | → 0x028000ef
3  0x002cff08:       bne t1, zero, loop   | 1 |  0x3F  | 0 | 6 | 1 |  0xC  | 1 |  0x63  | → 0xfe031ce3
4  ...
5  0x002cff2c: foo:  jr ra                 ra = _________0x002cff08_________
```

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`?

```
        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)
```

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.

```
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.