CS 61C Fall 2022

CALL, RISC-V Procedures

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 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 Because `a0` and `a1` are the return values of a function, the other `a` registers will be unchanged after returning from a function.

False. While `a0` and `a1` are the return values, all `a` registers must be caller saved. The function may have manipulated the argument registers in its execution, so it is unknown that `a` registers will stay the same.

1.3 The stack should only be manipulated at the beginning and end of functions, where the callee saved registers are temporarily saved.

False. While it is a good idea to create a separate 'prologue' and 'epilogue' to save callee registers onto the stack, the stack is mutable anywhere in the function. A good example is if you want to preserve the current value of a temporary register, you can decrement the `sp` to save the register onto the stack right before a function call.

1.4 The compiler may output pseudoinstructions.

True. It is the job of the assembler to replace these pseudoinstructions.

1.5 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.

1.6 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.

1.7 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.

2 Calling Convention Practice

2.1 In a function called `myfunc`, we want to call two functions called `generate_random` and `reverse_and_multiply`.

`myfunc` takes in 3 arguments: `a0, a1, a2`

`generate_random` takes in no arguments and returns a random integer to `a0`.

`reverse_and_multiply` takes in 4 arguments: `a0, a1, a2, a3` and doesn't return anything.

```
1  myfunc:
2      # Prologue (omitted)
3
4      # assign registers to hold arguments to myfunc
5      addi t0 a0 0
6      addi s0 a1 0
7      addi a7 a2 0
8
9      jal generate_random
10
11     # store and process return value
12     addi t1 a0 0
13     slli t5 t1 2
14
15     # setup arguments for reverse
16     add a0 t0 x0
17     add a1 s0 x0
18     add a2 t5 x0
19     addi a3 t1 0
20
21     jal reverse
22
23     # additional computations
24     add t0 s0 x0
25     add t1 t1 a7
26     add s9 s8 s7
27     add s3 x0 t5
28
29     # Epilogue (omitted)
30     ret
```

2.1 Which registers, if any, need to be saved on the stack in the prologue?

`s0`, `s3`, `s9`, `ra` We must save all s-registers we modify, and it is conventional to store ra in the prologue (rather than just before calling a function) when the function contains a function call.

2.2 If generate_random follows calling conventions, which registers do we need to save on the stack before calling generate_random?

`t0`, `a7`

Under calling conventions, all the t-registers and a-registers may be changed by generate_random, so we must store all of these which we need to know the value of after the call. t0 is used on line 16 and a7 is used on line 25. Note that while t1 and t5 are used later, we don't care about its value before calling generate_random (they are set after the call, on lines 12-13), so we don't need to store them.

2.3 If reverse follows calling conventions, which registers do we need to save on the stack before calling `reverse`?

`t1`, `t5`, `a7`

As before, we must save t-registers and a-registers we need to read later.

2.4 Which registers need to be recovered in the epilogue before returning?

`s0`, `s3`, `s9`, `ra`

This mirrors what we saved in the prologue.

3 Translation

3.1 In this question, we will be translating between RISC-V code and binary/hexadecimal values.

Translate the following Risc-V instructions into binary and hexadecimal notations

```
1 addi  s1  x0  -24  =      0b_____________________________    =  0x___________
2 sh  s1  4(t1)      =      0b_____________________________    =  0x___________
```

For this question, use the reference sheet to get information about the instructions and convert them to binary representation. One thing that helps is splitting the parsing into parts. For question 1:

```
1 addi  s1  x0  x4:
2 rd= s1 = 0b01001
3 rs1 = x0 = 0b00000
4 immediate = -24 = 0b1111 1110 1000
5 opcode = 001 0011
6 funct3 = 000
7 Bringing it together - 0b1111 1110 1000 0000 0000 0100 1001 0011 = 0xFE800493
```

For question 2, with a similar method we get the answer: 0b0000 0000 1001 0011 0001 0010 0010 0011 = 0x00931223

3.2 In this question, we will be translating between RISC-V code and binary/hexadecimal values.

Translate the following hexadecimal values into the relevant RISC-V instruction.

```
1 0x234554B7 =  _____________________________
2 0xFE050CE3 = _____________________________
```

For the reverse conversion, we want to first determine the instruction type. In order to do that, we first look at the opcode (and then func3/func7 if necessary). Let's start with the first one:

```
0x234554B7 = 0b0010 0011 0100 0101 0101 0100 1011 0111, the opcode is always the last 7 bits so
    opcode = 011 0111, which corresponds to the operation lui!
Looking at lui, we can see that the first 20 bits correspond to the immediate, and the next 5 ones
    are the register ones. So:
0b0010 0011 0100 0101 0101 = 0x23455 So, the immediate input was indeed 0x23455.
Looking at the next 5 bits, they must be the rd register values. So, we have
rd = 0b01001
That is equal to 9, which is the register x9 = s1. Thus, overall we have
lui s1 0x23455
```

For question 2, with a similar approach: beq a0, x0, -8

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? Recall that RISC-V supports 16b instructions via an extension.

The B-format instruction encoding has space for a 12 bit immediate field. Since RISC-V supports 16b instructions, the byte offset needed to branch to any instruction will always be divisible by 2. Thus, we can assume an implicit 0 at bit 0, allowing a 13 bit byte offset (that's why the reference sheet describes the immediate as `imm[12:1]`!). Since the byte offset is signed, the branch immediate can move the PC in the range of $[-2^{12}, 2^{12} - 1]$ bytes. As we are looking for the range of 32-bit instructions, we look for only offsets divisible by 4, leading to a total of $[-2^{10}, 2^{10} - 1]$ 32-bit instructions to branch to.

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.

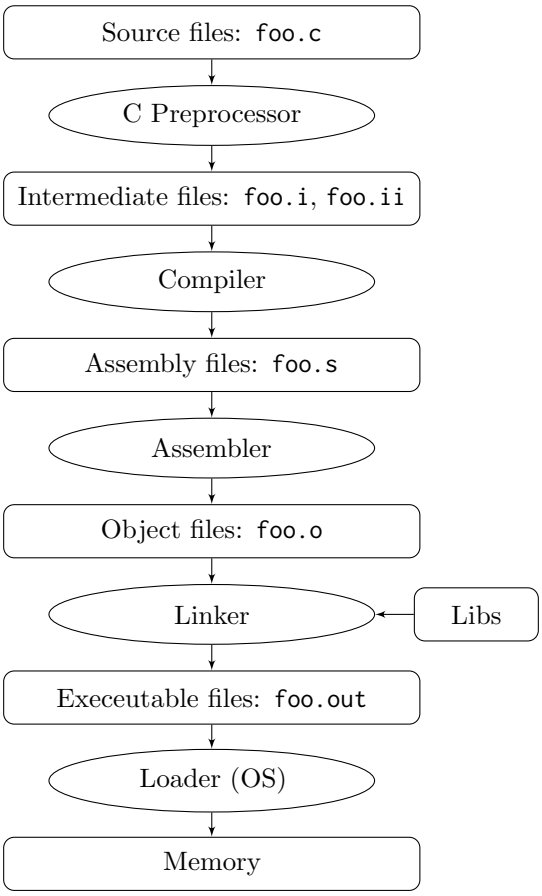
4.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V reference sheet!). Each field refers to a different block of the instruction encoding.

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

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

5 CALL

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



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

Two: The first finds all the label addresses, and the second resolves forward references while using these label addresses.

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

Assembler, Linker

5.3 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

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

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

6, 7, 8, 15, 16.

`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`).

6.2 For the branch/jump instructions, which labels will be resolved in the first pass of the assembler? The second? Assume that the code is processed from top to bottom.

`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
```

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

Label	**Address**
sum	`0x00061C00`

The only labels that are put in the symbol table are the ones which external programs can reference, declared using the `.globl` directive.

6.4 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. 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.