CS 61C Summer 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 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.

2 Translation

2.1 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

```
addi  s1  x0  -24  =      0b______________________________    =  0x___________
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:

```
addi  s1  x0  x4:
rd= s1 = 0b01001
```

```
3 rs1 = x0 = ob00000
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 1010 0010 0011 = 0x00931A23

2.2 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. You can assume that each hexadecimal value does represent an 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:

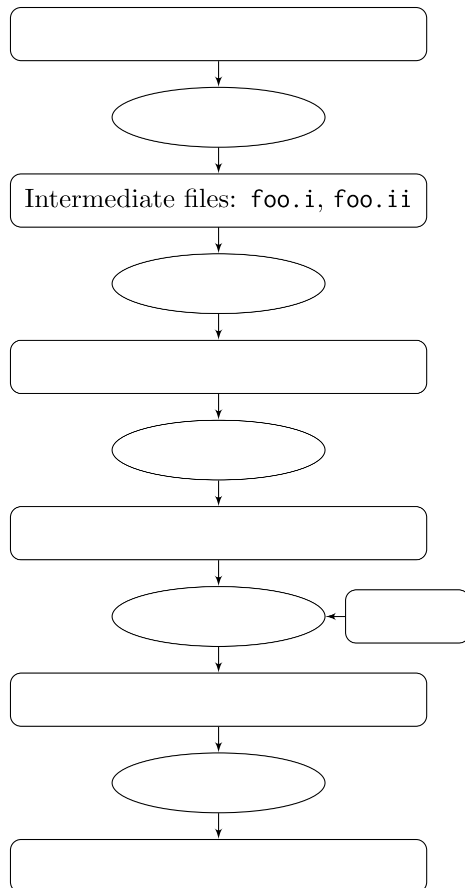
```
1 0x234554B7 = 0b0010 0011 0100 0101 0101 0100 1011 0111, the opcode is    always the last 7 bits so
      opcode = 011 0111, which corresponds to the operation lui!
2 Looking at lui, we can see that the first 20 bits correspond to the immediate, and the next 5 ones
      are the register ones. So:
3 0b0010 0011 0100 0101 0101 = 0x23455 So, the immediate input was indeed 0x23455.
4 Looking at the next 5 bits, they must be the rd register values. So, we have
5 rd = 0b01001
6 That is equal to 9, which is the register x9 = s1. Thus, overall we have
7 lui s1 0x23455
```

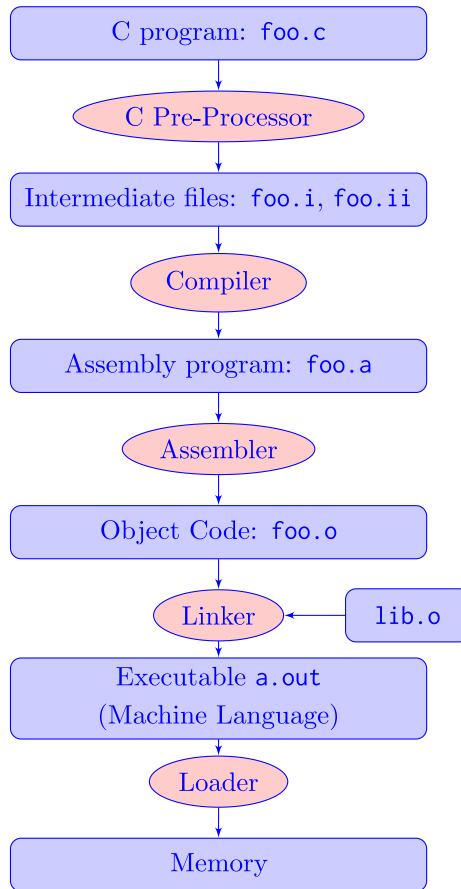
For question 2, with a similar approach: beq a0, x0, -8

3 CALL

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

3.1 Fill in the diagram such that the oval blanks hold the program/tool name (e.g. interpreter) and rectangular boxes hold what goes into each program and the filetype (e.g. high-level code: `foo.py`.





3.2 For each step, describe briefly the program's overall job and generally how it's done. Then describe why we're not done at this stage.

C Pre-Processor: the C Pre-Processor helps integrate macros in C with the rest of the code to help improve efficiency of code. It does this by locating parts of C code marked with pre-processor directives (e.g. `define, ifdef, include` before fully finishing compilation of code. **Note that the C preprocessor in real life executes *after* the first few steps of compilation (that we don't cover and is not in scope for 61C), but effectively for our purposes, we can consider it to execute "first".** We're not done at this stage because the code still resembles source C code which is non-readable by the system.

Compilation: the compiler turns higher level code, like C, into optimised assembly language. It's generally done by the compiler taking into account our overarching code and deciding what's efficient, what's not, and what can be "fixed". We're not done at this stage because the resulting assembly file still contains pseudoinstructions and has not resolved memory addresses, so it cannot be translated to binary.

Assembly: the assembler attempts to resolve non-relocated addresses, and produces additional information from our file to use later on by other programs, such as the symbol and relocation tables. It does this by making 1 or 2 passes over the file, filling in the two tables, and using them to determine addresses that need resolving and where labels are defined. We're not done at this stage because the resulting object file still contains unresolved addresses.

Linking: the linker stitches together the same segments from each of the object files and libraries needed for the program and resolves absolute addressing at this point. It does the first part by identifying the text and data components' start and end indices using the object file header and appending text-to-text and data-to-data segments. The does the second part by using the symbol tables included in object files to "request" file-relative address from other files for each file's own unresolved relocation entries. We're not done at this stage because the program has not been loaded into executable parts of memory and the system has not been set up to run the program. This is also assuming we're using statically linked libraries, which will be fully present for each program at this point in the process. However, if we use dynamically linked library, some addresses and setup will only be fully resolved at the beginning of program loading.

Loading: the loader loads the executable into executable memory, and preps the system for running the program by setting up system arguments, hardware components (e.g. register states), the stack, and more. It does this by basically working with (or being incorporated into) the operating system, so it understands and knows what is happening in a finer grain detail than we as users do. At this point, we're done since the program is ready to be run!

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

Two, one to find all the label addresses, and another to resolve forward references while using these label addresses.

3.4 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

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

Assembler, Linker

4 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
```

4.1 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`).

4.2 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
```

4.3 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.

4.4 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.

5 More Calling Convention

In a function called `array`, we want to call a function called `reverse_and_multiply`, which takes in an `array` and reverses the array while multiplying each element by a random number. `array` takes in 3 arguments: `a0` - the address of the original array `a1` - the address of a new array with the same length as `a0` `a2` - the length of the array at address `a0` `reverse_and_multiply` takes in 3 arguments: `a0` - the address of the original array `a1` - the address of a new array with the same length as `a0` `a2` - the length of the array at address `a0` `a3` - the random number `generate_random` takes in 0 arguments and returns a random integer to `a0`

```
1  array:
2      # Prologue
3
4      addi t0 a0 0    # t0 is now the address of the original array
5      addi s0 a1 0    # s0 is now the address of a new array w/ same length as a0
6      addi a7 a2 0    # a7 now contains the length of the array
7
8      jal generate_random
9
10     addi t1 a0 0    # t1 now contains the random number
11
12     add a0 t0 x0    # a0 now contains the address of the original array
13     add a1 s0 x0    # a1 now contains the address of a new array with same length as a0
14     add a2 a7 x0    # a2 now contains the length of the array
15     addi a3 t1 0    # a3 now contains the random number
16
17     jal reverse
18
19     add t0 s0 x0
20     add t1 t1 t1
21     add a7 a6 a5
22     add s9 s8 s7
23     add s3 x0 t5
24     # Epilogue
25     ret
```

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

`s0`, `s3`, `s7`, `s8`, `s9`, `ra`

TODO Full explanation.

5.2 Assuming generate_random uses all the t registers and all the a registers, what registers, if any, do we need to save on the stack before calling generate_random?

`t0`, `a7`

5.3 Now let's assume `generate_random` only uses s registers. Which registers do we need to save on the stack before calling `generate_random`? What registers does

`generate_random` need to save on the stack in its prologue?

No registers need to be save on the stack before calling `generate_random`. All the `s` registers need to be saved on the stack for `generate_random`'s prologue.

5.4 Assuming reverse uses the following registers: `t0`, `t5`, `s0`, `s3`, `s7`, `s9`, `a5`. Which registers do we need to save on the stack before calling `reverse`?

`a5`, `t5`

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

`s0`, `s3`, `s7`, `s8`, `s9`, `ra`

6 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`.

6.1 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 $[-2^{10}, 2^{10} - 1]$

6.2 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.

6.3 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________
```