CS 61C Fall 2023 — RISC-V Functions, ISA, CALL — Discussion 5

1 Calling Convention

You are given a function `mystery_func` that violates calling convention. It takes in two arguments: a0, a1. The return value is stored in a0. It calls two functions, `function_1` and `function_2`, which you may assume follow calling convention.

```
1  mystery_func:
2      # Prologue
3      addi sp sp -8
4      sw s0 0(sp)
5      sw s1 4(sp)
6
7      mv s0 a0
8      mv s1 a1
9      slli t0 a1 2
10     add a0 a0 t0
11
12     jal ra function_1
13
14     mv s2 a0
15     add a0 a0 t0
16     addi a1 x0 23
17
18     jal ra function_2
19
20     add a0 s0 s1
21     add a0 a0 s2
22
23     # Epilogue
24     lw s0 0(sp)
25     lw s1 4(sp)
26     addi sp sp 8
27
28     jr ra
```

1.1 Identify all calling convention errors in `mystery_func`. For each error, briefly explain the issue that it creates, and how it can be fixed.

There are three calling convention errors in `mystery_func`.

1. `ra` is not saved and restored, even though other functions are called within `mystery_func`. When a function is called (say, `jal ra function_1`), ra will be overwritten and the old value of ra will be lost. When `mystery_func` tries

to return at the instruction `jr ra`, the ra that it is returning to is not the original one we wanted it to return to, but a new unintended ra generated by a function call. It is conventional (and convenient in this situation, since we have two function calls) to save and restore ra in the prologue and epilogue when needed, so this issue could be fixed by adding ra to the prologue/epilogue.

2. `s2` is not saved and restored, even though it is assigned a value (`mv s2 a0`). Calling convention requires that saved registers be preserved by the function being called (the callee). If a function uses register `s2` and then calls `mystery_func`, `s2` may not have the same value after the call, which creates issues if that function uses `s2` expecting it to be the same. This issue can be fixed by adding `s2` to the prologue and epilogue.

3. `t0` is used right after a function call (`add a0 a0 t0`). Values stored in non-preserved registers (`t0-t6, a0-a7, ra`) may be overwritten and not restored by a function being called (a callee). This means that `t0`'s value may have been overwritten by the function, and should be treated as garbage. This will create issues as `t0` is not what we think it is. This can be resolved by using a saved register instead, or saving to and restoring from the stack.

Below is one example of `mystery_func` with all calling convention errors fixed.

```
1  mystery_func:
2      # Prologue
3      addi sp sp -16
4      sw s0 0(sp)
5      sw s1 4(sp)
6      sw s2 8(sp)
7      sw ra 12(sp)
8
9      mv s0 a0
10     mv s1 a1
11     slli t0 a1 2
12     add a0 a0 t0
13
14     addi sp sp -4
15     sw t0 0(sp)
16
17     jal ra function_1
18
19     lw t0 0(sp)
20     addi sp sp 4
21
22     mv s2 a0
23     add a0 a0 t0
24     addi a1 x0 23
25
26     jal ra function_2
27
```

```
28      add a0 s0 s1
29      add a0 a0 s2
30
31      # Epilogue
32      lw s0 0(sp)
33      lw s1 4(sp)
34      lw s2 8(sp)
35      lw ra 12(sp)
36      addi sp sp 16
37
38      jr ra
```

2 RISC-V 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
rs1 = x0 = 0b00000
immediate = -24 = 0b1111 1110 1000
opcode = 001 0011
funct3 = 000
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

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.

```
0x234554B7 =  ___________________________________
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

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 data 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 instruction address (used by branch and jump instructions).
3. Register Addressing uses the value in a register as an instruction address. For instance, `jalr`, `jr`, and `ret`, where `jr` and `ret` are just pseudoinstructions that get converted to `jalr`.

3.1 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.

3.2 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.

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 reference sheet!). Each field refers to a different block of the instruction encoding.

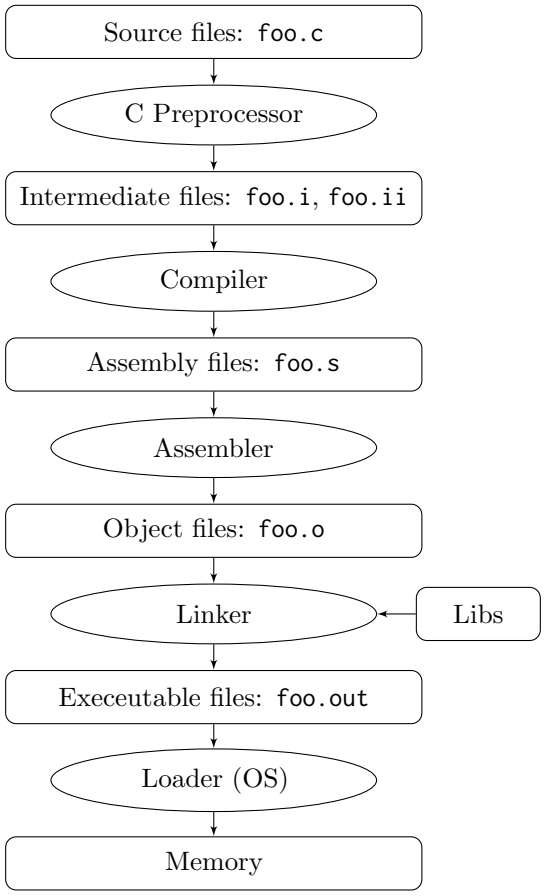
```
0x002cff00: loop: add t1, t2, t0          |_______|_______|_______|_______|_______|__0x33__|
0x002cff04:       jal ra, foo             |_____________________________|_____________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |_______|_______|_______|_______|_______|__0x63__|
...
0x002cff2c: 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__________
```

4 CALL

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



Please answer true/false to the following questions, and include an explanation.

4.1 The compiler may output pseudoinstructions.

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

4.2 The main job of the assembler is to perform optimizations on the assembly code.

False. That's the job of the compiler. The assembler is primarily responsible for replacing pseudoinstructions and resolving offsets.

4.3 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.

4.4 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.

Please answer the following questions with a short answer.

4.5 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.

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

Assembler, Linker

4.7 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