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.

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.

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

1.4 The compiler may output pseudoinstructions.

1.5 The main job of the assembler is to generate optimized machine code.

1.6 The object files produced by the assembler are only moved, not edited, by the linker.

1.7 The destination of all jump instructions is completely determined after linking.

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.

```
myfunc:
    # Prologue (omitted)

    # assign registers to hold arguments to myfunc
    addi t0 a0 0
    addi s0 a1 0
    addi a7 a2 0

    jal generate_random

    # store and process return value
    addi t1 a0 0
    slli t5 t1 2

    # setup arguments for reverse
    add a0 t0 x0
    add a1 s0 x0
    add a2 t5 x0
    addi a3 t1 0

    jal reverse

    # additional computations
    add t0 s0 x0
    add t1 t1 a7
    add s9 s8 s7
    add s3 x0 t5

    # Epilogue (omitted)
    ret
```

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

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

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

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

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

```
addi  s1  x0  -24  =      0b______________________________   =  0x___________
sh  s1  4(t1)      =      0b______________________________   =  0x___________
```

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.

```
0x234554B7 =  ________________________________
0xFE050CE3 = ________________________________
```

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.

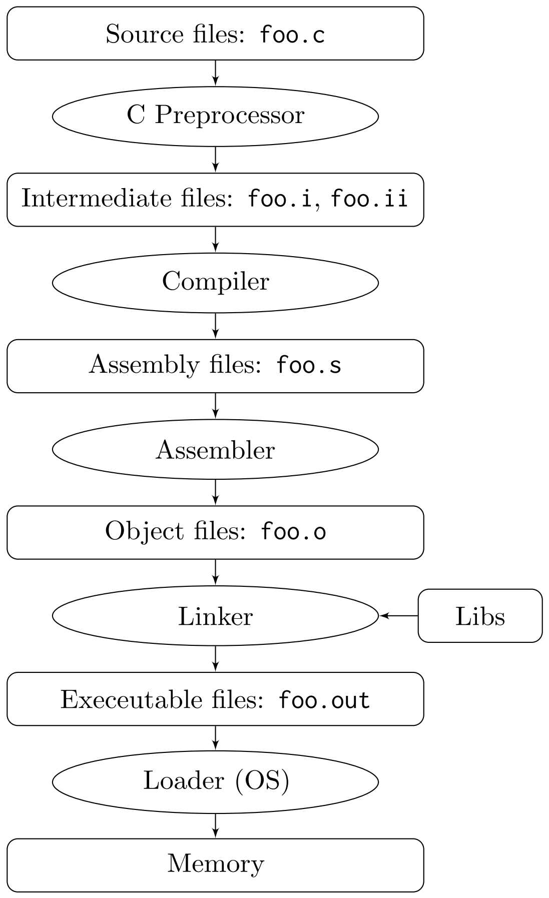
4.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

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 = _____________________________
```

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?

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

5.3 Describe the six main parts of the object files outputted by the Assembler (Header, Text, Data, Relocation Table, Symbol Table, Debugging Information).

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?

6.2 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.

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?).

```
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:         add 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
```

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

6.4 What's contained in the relocation table?