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.

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

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

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

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__________
```

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

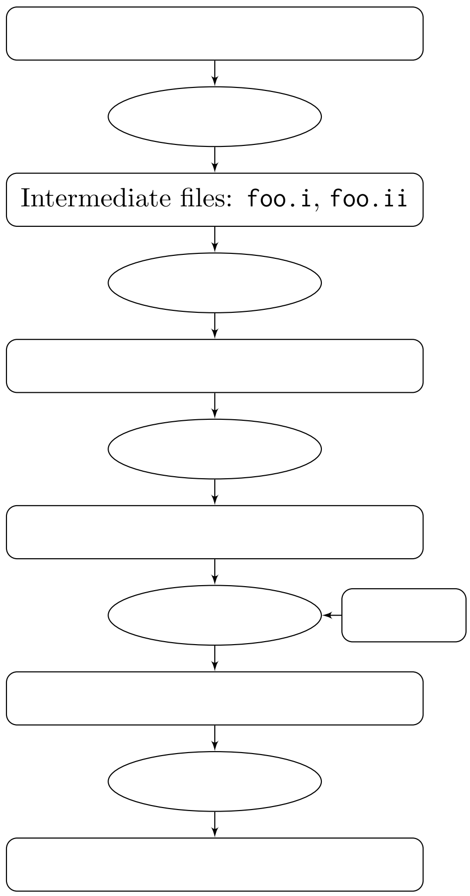
Translate the following hexadecimal values into the relevant RISC-V instruction. You can assume that each hexadecimal value does represent an instruction.

```
0x234554B7 =  _________________________________
0xFE050CE3 = _________________________________
```

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.

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

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

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

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.

```
.import print.s             # print.s is a different file
.data
array: .word 1 2 3 4 5
.text
sum:    la t0, array
        li t1, 4
        mv t2, x0
loop:   blt t1, x0, end
        slli t3, t1, 2
        add t3, t0, t3
        lw t3, 0(t3)
        add t2, t2, t3
        addi t1, t1, -1
        j loop
end:    mv a0, t2
        jal ra, print_int   # Defined in print.s
```

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

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

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
```

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

4.4 What's contained in the relocation table?

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`

```
array:
    # Prologue

    addi t0 a0 0    # t0 is now the address of the original array
    addi s0 a1 0    # s0 is now the address of a new array w/ same length as a0
    addi a7 a2 0    # a7 now contains the length of the array

    jal generate_random

    addi t1 a0 0    # t1 now contains the random number

    add a0 t0 x0    # a0 now contains the address of the original array
    add a1 s0 x0    # a1 now contains the address of a new array with same length as a0
    add a2 a7 x0    # a2 now contains the length of the array
    addi a3 t1 0    # a3 now contains the random number

    jal reverse

    add t0 s0 x0
    add t1 t1 t1
    add a7 a6 a5
    add s9 s8 s7
    add s3 x0 t5
    # Epilogue
    ret
```

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

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?

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?

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`?

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

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?

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

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!).

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