CS61C Discussion 4 – RISC-V Addressing, Inst. Formats, and CALL

1 RISC-V Addressing

- We have several **addressing modes** to access memory (immediate not listed):
  (a) **Base displacement addressing:** Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
  (b) **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)
  (c) **Register Addressing:** Uses the value in a register as a memory address (jr)

1. What is 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. Thus, the branch immediate can move the reference 2-byte instructions that are within $[-2^{11}, 2^{11}-1]$ instructions of the current PC. 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 before, and the range of 4-byte instructions is $[-2^{10}, 2^{10}-1]$

2. What is the range of 32-bit instructions that can be reached from the current PC using a jump instruction?
   The immediate field of the jump instruction is 20 bits. Similar to above, this immediate is multiplied by 2 before added to the PC to get the final address. Since the immediate is signed, the range of 2-byte instructions that can be referenced is $[-2^{19}, 2^{19}-1]$. As we actually want the number of 4-byte instructions, we actually can reference those within $[-2^{18}, 2^{18}-1]$ instructions of the current PC.

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         |    0    |     5    |     7     |  0  |  6  |  0x33  |
0x002cff04:       jal ra, foo            |  0  |   0x14   |  0  |     0     |     1     |  0x6F  |
0x002cff08:       bne t1, zero, loop     |  1  |  0x3F  |  0  |  6  |  1  |  0xC  |  1  |  0x63  |
...
0x002cff2c: foo:  jr ra                 ra=__0x002cff08___
```

2 Powerful RISC-V Functions

1. Write a function `double` in RISC-V that, when given an integer x, returns $2x$.

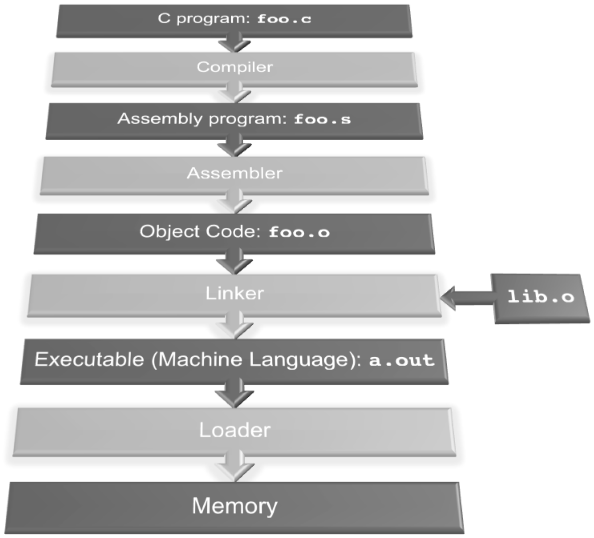
```
double: add a0, a0, a0
        jr ra
```

2. Write a function `power` in RISC-V that takes in two numbers x and n, and returns x^n. You may assume that $n \geq 0$ and that multiplication will always result in a 32-bit number.

```
power: li   t0, 0           # Set t0 to be a 0 (counter variable)
       addi t1, a0, 0       # Set t1 to be a0, which represents x
       addi a0, x0, 1       # Set a0, the return value, to 1
loop:  bge  t0, a1, end     # End the loop if the counter is greater than or equal to a1 (repres
       mul  a0, a0, t1      # Multiply the running product a0 by t1 (which holds x)
       addi t0, t0, 1       # Increment the counter
       jal  x0, loop        # Jump back to the while condition
end:   jr ra                # Return to caller
```

3 Compile, Assemble, Link, Load, and Go!

3.1 Overview



3.2 Exercises

a. What is the Stored Program concept and what does it enable us to do?
It is the idea that instructions are just the same as data, and we can treat them as such. This enables us to write programs that can manipulate other programs!

b. How many passes through the code does the Assembler have to make? Why?
Two, one to find all the label addresses and another to convert all instructions while resolving any forward references using the collected label addresses.

c. What are the different parts of the object files output by the Assembler?
Header: Size and position of 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 Linker
Symbol Table: List of the files labels and data that can be referenced
Debugging Information: Additional information for debuggers

d. Which step in CALL resolves relative addressing? Absolute addressing? Assembler, Linker.

e. What does RISC stand for? How is this related to pseudoinstructions?
Reduced Instruction Set Computing. Minimal set of instructions leads to many lines of code. Pseudoinstructions are more complex instructions intended to make assembly programming easier for the coder. These are converted to TAL by the assembler.

4 Writing RISC-V Functions

Write a function `sumSquare` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

For this problem, you are given a RISC-V function called `square` that takes in an integer and returns its square. Implement `sumSquare` using `square` as a subroutine.

```
sumSquare: addi sp, sp -12   # Make space for 3 words on the stack
           sw   ra, 0(sp)    # Store the return address
           sw   s0, 4(sp)    # Store register s0
           sw   s1, 8(sp)    # Store register s1
           add  s0, a0, x0   # Set s0 equal to the parameter n
           add  s1, x0, x0   # Set s1 equal to 0 (this is where we accumulate the sum)
     loop: bge  x0, s0, end  # Branch if s0 is not positive
           add  a0, s0, x0   # Set a0 to the value in s0 to prepare for the function square
           jal  ra, square   # Call the function square
           add  s1, s1, a0   # Add the returned value into the accumulator s1
           addi s0, s0, -1   # Decrement s0 by 1
           jal  x0, loop     # Jump back to the loop label
      end: add  a0, s1, x0   # Set a0 to s1, which is the desired return value
           lw   ra, 0(sp)    # Restore ra
           lw   s0, 4(sp)    # Restore s0
           lw   s1, 8(sp)    # Restore s1
           addi sp, sp, 12   # Free space on the stack for the 3 words
           jr   ra           # Return to the caller
```