CS61C Summer 2018 Discussion 3 – RISC-V

1 RISC-V with Arrays and Lists

Comment each snippet with what the snippet does. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which is starts at memory address `0xBFFFFF00`, and a linked list struct (as defined below), `struct ll* lst;`, whose first element is located at address `0xABCD0000`. `s0` then contains `arr`'s address, `0xBFFFFF00`, and `s1` contains `lst`'s address, `0xABCD0000`. You may assume integers and pointers are 4 bytes and that structs are tightly packed.

```
struct ll {
    int val;
    struct ll* next;
}
```

1.
```
lw  t0, 0(s0)    # Loads arr[0] into register t0
lw  t1, 8(s0)    # Loads arr[2] into register t1
add t2, t0, t1   # Sets t2 equal to t0 plus t1
sw  t2, 4(s0)    # Sets arr[1] equal to value in t2
```

Sets `arr[1]` to `arr[0]` + `arr[2]`

2.
```
loop: beq  s1, x0, end    # Branch to the end if struct pointer (s1) is NULL
      lw   t0, 0(s1)      # Load the value of the node into t0
      addi t0, t0, 1      # Increment t0 by 1
      sw   t0, 0(s1)      # Store the incremented value back into the node
      lw   s1, 4(s1)      # Load the address of the next element into s1
      jal  x0, loop       # Jump back to the loop label
 end:
```

Increments all values in the linked list by 1.

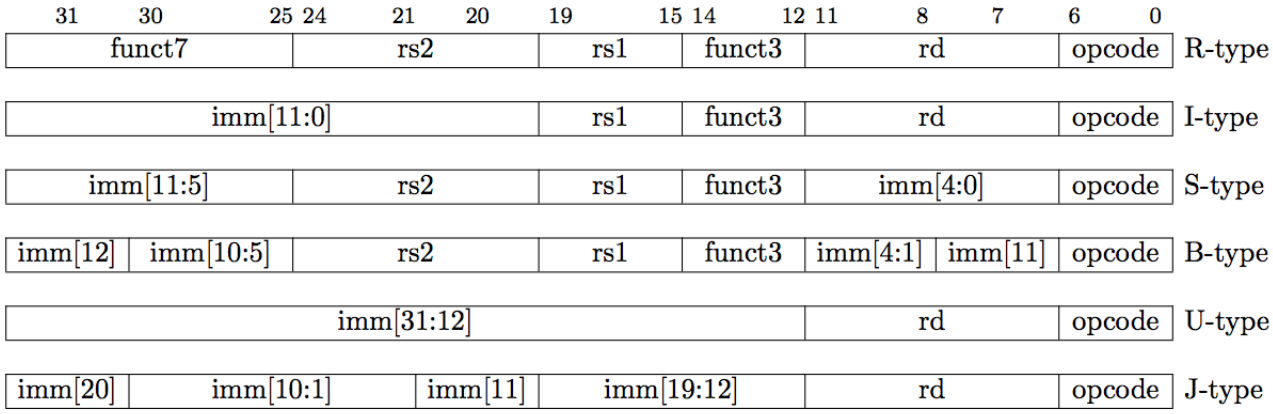
3.
```
       add  t0, x0, x0     # Sets register t0 to 0
loop:  slti t1, t0, 6      # Sets t1 to 1 if t0 < 6, 0 otherwise
       beq  t1, x0, end    # Branches to the end if t1 is 1 (t0 >= 6)
       slli t2, t0, 2      # Sets t2 to t0 * 4 (4 is number of bytes in an integer)
       add  t3, s0, t2     # Sets t3 to the address of arr[t0] (added t2 bytes to arr)
       lw   t4, 0(t3)      # Load arr[t0] into register t4
       sub  t4, x0, t4     # Sets t4 to its negative
       sw   t4, 0(t3)      # Stores this updated value back at arr[t0]
       addi t0, t0, 1      # Increments t0 to move to the next element
       jal  x0, loop       # Jump back to the loop label
 end:
```

Negates all elements in `arr`

2 RISC-V Instruction Formats

2.1 Overview

Instructions in RISC-V can be turned into binary numbers that the machine actually reads. There are different formats to the instructions, based on what information is need. Each of the fields above is filled in with binary



that represents the information. Each of the registers takes a 5 bit number that is the numeric name of the register (i.e. zero = 0, ra = 1, s1 = 9). See your reference card to know which register corresponds to which number.
I type instructions fill the immediate into the code. These numbers are signed 12 bit numbers.

2.2 Exercises

1. Expand `addi s0 t0 -1`
   111111111111 00101 000 01000 0010011 = 0xFFF28413
2. Expand `lw s4 5(sp)`
   000000000101 00010 010 10100 0000011 = 0x00512A03
3. Write the format name of the following instructions:
   (a) `jal` UJ
   (b) `lw` I
   (c) `beq` SB
   (d) `add` R
   (e) `jalr` I
   (f) `sb` S
   (g) `lui` U

3 Translating between C and RISC-V

Translate between the C and RISC-V code. You may want to use the RISC-V Green Card as a reference. We show you how the different variables map to registers – you don't have to worry about the stack or any memory-related issues.

C	RISC-V

`// Nth_Fibonacci(n):`
`// s0 -> n, s1 -> fib`
`// t0 -> i, t1 -> j`
`// Assume fib, i, j are already these values`
`int fib = 1, i = 1, j = 1;`
`if (n==0)  return 0;`
`else if (n==1) return 1;`
`n -= 2;`
`while (n != 0) {`
`  fib = i + j;`
`  j = i;`
`  i = fib;`
`  n--;`
`}`
`return fib;`
	`  ...`
	`  beq  s0, x0, Ret0`
	`  addi t2, x0, 1`
	`  beq  s0, t2, Ret1`
	`  addi s0, s0, -2`
	`Loop: beq  s0, x0, RetF`
	`  add  s1, t0, t1`
	`  addi t1, t0, 0`
	`  addi t0, s1, 0`
	`  addi s0, s0, -1`
	`  jal  x0, Loop`
	`Ret0: addi a0, x0, 0`
	`  jal  x0, Done`
	`Ret1: addi a0, x0, 1`
	`  jal  x0, Done`
	`RetF: add  a0, x0, s1`
	`Done: ...`

4 RISC-V Calling Conventions

1. How do we pass arguments into functions?
   Use the 8 arguments registers `a0` - `a7`

2. How are values returned by functions?
   Use `a0` and `a1` as the return value registers as well

3. What is `sp` and how should it be used in the context of RISC-V functions?
   `sp` stands for stack pointer. We subtract from `sp` to create more space and add to free space. The stack is mainly used to save (and later restore) the value of registers that may be overwritten.

4. Which values need to saved before using `jal`?
   Registers `a0` - `a7`, `t0` - `t6`, and `ra`

5. Which values need to be restored before using `jalr` to return from a function?
   Registers `sp`, `gp` (global pointer), `tp` (thread pointer), and `s0` - `s11`. Important to note that we don't really touch `gp` and `tp`