CS 61C Fall 2023 — RISC-V Assembly, Functions

Discussion 4

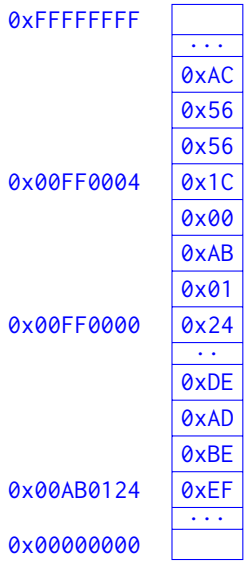
1 Data Transfer

Using the given instructions and the sample memory array, what will happen when the RISC-V code is executed? For load instructions (`lw, lb, lh`), write out what each register will store. For store instructions (`sw, sh, sb`), update the memory array accordingly. Recall that RISC-V is little-endian and byte addressable.

```
1  li x5 0x00FF0000
2  lw x6 0(x5)
3  addi x5 x5 4
4  lhu x7 1(x5)
5  lh x8 1(x5)
6  lb x9 3(x6)
7  sh x8 2(x5)
```

0xFFFFFFFF	
	...
	0x00
	0xAC
	0x56
0x00FF0004	0x1C
	0x00
	0xAB
	0x01
0x00FF0000	0x24
	..
	0xDE
	0xAD
	0xBE
0x00AB0124	0xEF
	...
0x00000000	

- Line 1: `x5` will hold `0x00FF0000`
- Line 2: `x6` will hold 0x00AB0124, the word at the address `0x00FF0000` + 0
- Line 3: `x5` will hold `0x00FF0004`
- Line 4: `x7` will hold `0x0000AC56`. `0xAC56` is the 2 bytes of data stored starting at address `0x00FF0004` + 1. Because the instruction is lhu, `x7` will hold `0xAC56` zero-extended. Recall, registers store 32 bits
- Line 5: `x8` will hold 0xFFFFAC56. The instruction is lh, so `0xAC56` is sign-extended
- Line 6: `x9` will hold `0xFFFFFFDE`. Byte `0xDE` is located at address 0x00AB0124 + 3. Register `x9` will hold `0xDE` sign-extended.
- Line 7: The last two bytes that x8 holds are `0xAC56`. These two bytes will be stored in memory starting at address `0x00FF0004` + 2


0xFFFFFFFF
...
0xAC
0x56
0x56
0x00FF0004
0x1C
0x00
0xAB
0x01
0x00FF0000
0x24
..
0xDE
0xAD
0xBE
0x00AB0124
0xEF
...
0x00000000

2 Arrays in RISC-V

Comment what the following code block does. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which starts at memory address `0xBFFFFF00`. Let `s0` contain `arr`'s address `0xBFFFFF00`. You may assume integers and pointers are 4 bytes.

2.1

```
       add  t0, x0, x0
loop:  slti t1, t0, 6
       beq  t1, x0, end
       slli t2, t0, 2
       add  t3, s0, t2
       lw   t4, 0(t3)
       sub  t4, x0, t4
       sw   t4, 0(t3)
       addi t0, t0, 1
       jal  x0, loop
 end:
```

Negates all elements in `arr`.

2.2 **Conceptual check:** Let `a0` point to the start of an array `x`. `lw s0, 4(a0)` will always load `x[1]` into `s0`.

False. This only holds for data types that are four bytes wide, like `int` or `float`. For data-types like `char` that are only one byte wide, `4(a0)` is too large of an offset to return the element at index 1, and will instead return a `char` further down the array (or some other data beyond the array, depending on the array length).

3 Calling Convention Practice

Function `myfunc` takes in two arguments: a0, a1. The return value is stored in a0. In `myfunc`, `generate_random` is called. It takes in 0 arguments and stores its return value in `a0`.

```
1  myfunc:
2      # Prologue (omitted)
3
4      addi t0 x0 1
5      slli t1 t0 2
6      add t1 a0 t1
7      add s0 a1 x0
8
9      jal generate_random
10
11     add t1 t1 a0
12     add a0 t1 s0
13
14     # Epilogue (omitted)
15     ret
```

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

`s0`, `ra`. We must save all s-registers we modify. In addition, if a function contains a function call, register ra will be overwritten when the function is called (i.e. jal ra label). ra must be saved before a function call. It is conventional to store ra in the prologue (rather than just before calling a function) when the function contains a function call. myfunc contains the function call generate_random.

3.2 Which registers do we need to save on the stack before calling `generate_random`?

`t1`.

Under calling conventions, all the t-registers and a-registers may be changed by generate_random, so we must store all of these which we need to know the value of after the call. A total of 2 t-registers are used before calling generate_random, t0 and t1, but only t1's value is referenced again after the function call.

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

`s0`, `ra`. This mirrors what we saved in the prologue.