CS 61C Spring 2024 — RISC-V Assembly, Functions — Discussion 4

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 In order to use the saved registers (`s0-s11`) in a function, we must store their values before using them and restore their values before returning.

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

2 Arrays in RISC-V

Comment what each code block does. Each block runs in isolation. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which starts at memory address `0xBFFFFF00`, and a linked list struct (as defined below), `struct ll* lst`, whose first element is located at address `0xABCD0000`. Let `s0` contain `arr`'s address `0xBFFFFF00`, and let `s1` contain `lst`'s address `0xABCD0000`. You may assume integers and pointers are 4 bytes and that structs are tightly packed. Assume that `lst`'s last node's `next` is a NULL pointer to memory address `0x00000000`.

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

2.1
```
lw  t0, 0(s0)
lw  t1, 8(s0)
add t2, t0, t1
sw  t2, 4(s0)
```

2.2
```
loop: beq  s1, x0, end
      lw   t0, 0(s1)
      addi t0, t0, 1
      sw   t0, 0(s1)
      lw   s1, 4(s1)
      jal  x0, loop
 end:
```

2.3
```
       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:
```

3 Memory Access

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.

3.1

```
li t0 0x00FF0000
lw t1 0(t0)
addi t0 t0 4
lh t2 2(t0)
lw s0 0(t1)
lb s1 3(t2)
```

0xFFFFFFFF	
	...
0x00FF0004	0x000C561C
0x00FF0000	36
	...
0x00000036	0xFDFDFDFD
0x00000024	0xDEADB33F
	...
0x0000000C	0xC5161C00
	...
0x00000000	

What value does each register hold after the code is executed?

3.2

```
li t0 0xABADCAFE
li t1 0xF9120504
li t2 0xBEEFCACE
sw t0 0(t1)
addi t1 t1 4
addi t0 t0 4
sh t1 2(t0)
sb t2 1(t2)
sb t2 3(t1)
sb t2 3(t0)
```

0xFFFFFFFF	
0xF9120504	
0xABADCAFE	
0x00000004	
0x00000000	0x00000000

Update the memory array with its new values after the code is executed. Some memory addresses may not have been labeled for you yet.

4 Calling Convention Practice

4.1 In a function called `myfunc`, we want to call two functions called `generate_random` and `reverse`.

`myfunc` takes in 3 arguments: `a0, a1, a2`

`generate_random` takes in no arguments and returns a random integer to `a0`.

`reverse` takes in 4 arguments: `a0, a1, a2, a3` and doesn't return anything.

```
1 myfunc:
2     # Prologue (omitted)
```

```
3
4       # assign registers to hold arguments to myfunc
5       addi t0 a0 0
6       addi s0 a1 0
7       addi a7 a2 0
8
9       # Save the registers in 4.2
10      jal generate_random
11      # Load the registers stored from 4.2
12
13      # store and process return value
14      addi t1 a0 0
15      slli t5 t1 2
16
17      # setup arguments for reverse
18      add a0 t0 x0
19      add a1 s0 x0
20      add a2 t5 x0
21      addi a3 t1 0
22
23      # Save the registers in 4.3
24      jal reverse
25      # Load the registers stored from 4.2
26
27      # additional computations
28      add t0 s0 x0
29      add t1 t1 a7
30      add s9 s8 s7
31      add s3 x0 t5
32
33      # Epilogue (omitted)
34      ret
```

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

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

4.3 Which registers do we need to save on the stack before calling `reverse`?

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