CS 61C RISC-V Assembly, Functions

Spring 2023 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 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).

1.2 Assuming no compiler or operating system protections, it is possible to have the code jump to data stored at `0(a0)` (offset 0 from the value in register `a0`) and execute instructions from there.

True. If your compiler/OS allows it (some do not, for security reasons), it is possible for your code to jump to and execute instructions passed into the program via an array. Conversely, it's also possible for your code to treat itself as normal data (search up self-modifying code if you want to see more details).

1.3 `jalr` is a shorthand expression for a `jal` that jumps to the specified label and does not store a return address anywhere.

False. `jalr` is used to return to the memory address specified in the second argument. Keep in mind that `jal` jumps to a label (which is translated into an immediate by the assembler), whereas `jalr` jumps to an address stored in a register, which is set at runtime. Related, `j label` is a pseudo-instruction for `jal x0, label` (they do the same thing).

1.4 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.

False. `a0` and `a1` registers are often used to store the return value from a function, so the function can set their values to the its return values before returning.

1.5 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.

True. The saved registers are callee-saved, so we must save and restore them at the beginning and end of functions. This is frequently done in organized blocks of code called the ”function prologue” and ”function epilogue”.

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

False. While it is a good idea to create a separate 'prologue' and 'epilogue' to save callee registers onto the stack, the stack is mutable anywhere in the function. A good example is if you want to preserve the current value of a temporary register, you can decrement the `sp` to save the register onto the stack right before a function call.

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)
```

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

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:
```

Increments all values in the linked list by 1.

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:
```

Negates all elements in `arr`.

3 Memory Access

Using the given instructions and the sample memory arrays provided, 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 x5 0x00FF0000
lw x6 0(x5)
addi x5 x5 4
lh x7 2(x5)
lw x8 0(x6)
lb x9 3(x7)
```

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

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

`x5` will hold `0x00FF0004`, adding 4 to the initial address. `x6` will hold 36, loading the word from the address `0x00FF0000`. `x7` will hold `0xC`, loading the upper half of the address `0x00FF0004`. `x8` will hold the word at 36 = 0x24, so `0xDEADB33F`. Finally, `x9` will hold `0xFFFFFFC5`, taking the most significant byte and sign-extending it.

3.2

```
li x5 0xABADCAFE
li x6 0xF9120504
li x7 0xBEEFCACE
sw x5 0(x6)
addi x6 x6 4
addi x5 x5 4
sh x6 2(x5)
sb x7 1(x7)
sb x7 3(x6)
sb x7 3(x5)
```

`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.

0xFFFFFFFF	
0xF9120508	0xCE000000
0xF9120504	0xABADCAFE
0xBEEFCAD2	
0xBEEFCACE	0x0000CE00
0xABADCB02	0xCE080000
0xABADCAFE	
0x00000004	
0x00000000	0x00000000

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.

```
myfunc:
    # Prologue (omitted)

    # assign registers to hold arguments to myfunc
    addi t0 a0 0
    addi s0 a1 0
    addi a7 a2 0

    jal generate_random

    # store and process return value
    addi t1 a0 0
    slli t5 t1 2

    # setup arguments for reverse
    add a0 t0 x0
    add a1 s0 x0
    add a2 t5 x0
    addi a3 t1 0

    jal reverse

    # additional computations
```

```
24      add t0 s0 x0
25      add t1 t1 a7
26      add s9 s8 s7
27      add s3 x0 t5
28
29      # Epilogue (omitted)
30      ret
```

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

`s0`, `s3`, `s9`, `ra` We must save all s-registers we modify, and it is conventional to store ra in the prologue (rather than just before calling a function) when the function contains a function call.

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

`t0`, `a7`

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. t0 is used on line 16 and a7 is used on line 25. Note that while t1 and t5 are used later, we don't care about its value before calling generate_random (they are set after the call, on lines 12-13), so we don't need to store them.

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

`t1`, `t5`, `a7`

As before, we must save t-registers and a-registers we need to read later.

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

`s0`, `s3`, `s9`, `ra`

This mirrors what we saved in the prologue.