

## 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`.
- 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.
- 1.3 `jalr` is a shorthand expression for a `jal` that jumps to the specified label and does not store a return address anywhere.
- 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.
- 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.
- 1.6 The stack should only be manipulated at the beginning and end of functions, where the callee saved registers are temporarily saved.

## 2 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.

2.1	<code>li x5 0x00FF0000</code>	<code>0xFFFFFFFF</code>	
	<code>lw x6 0(x5)</code>		...
	<code>addi x5 x5 4</code>	<code>0x00FF0004</code>	<code>0x000C561C</code>
	<code>lh x7 2(x5)</code>	<code>0x00FF0000</code>	36
	<code>lw x8 0(x6)</code>		...
	<code>lb x9 3(x7)</code>	<code>0x00000036</code>	<code>0xFDFDFDFD</code>
		<code>0x00000024</code>	<code>0xDEADB33F</code>
			...
		<code>0x0000000C</code>	<code>0xC5161C00</code>
		<code>0x00000000</code>	...

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

2.2	<code>li x5 0xABADCAFE</code>	<code>0xFFFFFFFF</code>	
	<code>li x6 0xF9120504</code>		
	<code>li x7 0xBEEFCACE</code>		
	<code>sw x5 0(x6)</code>	<code>0xF9120504</code>	
	<code>addi x6 x6 4</code>		
	<code>addi x5 x5 4</code>		
	<code>sh x6 2(x5)</code>	<code>0xABADCAFE</code>	
	<code>sb x7 1(x7)</code>	<code>0x00000004</code>	
	<code>sb x7 3(x6)</code>	<code>0x00000000</code>	<code>0x00000000</code>
	<code>sb x7 3(x5)</code>		

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

### 3 Lost in Translation

3.1 Translate between the C and RISC-V verbatim.

C	RISC-V
<pre>// s0 -&gt; a, s1 -&gt; b // s2 -&gt; c, s3 -&gt; z int a = 4, b = 5, c = 6, z; z = a + b + c + 10;</pre>	
<pre>// s0 -&gt; int * p = intArr; // s1 -&gt; a; *p = 0; int a = 2; p[1] = p[a] = a;</pre>	

<pre>// s0 -&gt; a, s1 -&gt; b int a = 5, b = 10; if(a + a == b) {     a = 0; } else {     b = a - 1; }</pre>	
	<pre>addi s0, x0, 0 addi s1, x0, 1 addi t0, x0, 30 loop:     beq s0, t0, exit     add s1, s1, s1     addi s0, s0, 1     jal x0, loop exit:</pre>
<pre>// s0 -&gt; n, s1 -&gt; sum // assume n &gt; 0 to start for(int sum = 0; n &gt; 0; n--) {     sum += n; }</pre>	

## 4 Calling Convention Practice

Let's review what special meaning we assign to each type of register in RISC-V.

Register	Convention	Saver
x0	Stores <b>zero</b>	N/A
sp	Stores the <b>stack pointer</b>	N/A
ra	Stores the <b>return address</b>	Caller
a0 - a7	Stores <b>arguments</b> and <b>return values</b>	Caller
t0 - t7	Stores <b>temporary</b> values that <i>do not persist</i> after function calls	Caller
s0 - s11	Stores <b>saved</b> values that <i>persist</i> after function calls	Callee

To save and recall values in registers, we use the `sw` and `lw` instructions to save and load words to and from memory, and we typically organize our functions as follows:

- 1 # Prologue
- 2 `addi sp sp -8` # Room **for** two registers. (Why?)

```

3 sw s0 0(sp) # Save s0 (or any saved register)
4 sw s1 4(sp) # Save s1 (or any saved register)
5
6 # Code omitted
7
8 # Epilogue
9
10 lw s0 0(sp) #Load s0 (or any saved register)
11 lw s1 4(sp) #Load s1 (or any saved register)
12 addi sp sp 8 #Restore the stack pointer

```

Now, let's see what happens if we ignore calling convention.

4.1 Consider the following blocks of code:

<pre> 1 main: 2 # Prologue 3 # Saves ra 4 5 # Code omitted 6 addi s0 x0 5 7 # Breakpoint 1 8 jal ra foo 9 # Breakpoint 3 10 mul a0 a0 s0 11 # Code omitted 12 13 # Epilogue 14 # Restores ra 15 j exit </pre>	<pre> 1 foo: 2 # Preamble 3 # Saves s0 4 5 # Code omitted 6 addi s0 x0 4 7 # Breakpoint 2 8 9 # Epilogue 10 # Restores s0 11 jr ra </pre>
---	---

(a) Does `main` always behave as expected, as long as `foo` follows calling convention?

(b) What does `s0` store at breakpoint 1? Breakpoint 2? Breakpoint 3?

(c) Now suppose that `foo` didn't have a prologue or epilogue. What would `s0` store at each of the breakpoints? Would this cause errors in our code?

In part (c) above, we saw one way how not following calling convention could make our code misbehave. Other things to watch out for are: assuming that `a` or `t` registers will be the same after calling a function, and forgetting to save `ra` before calling a function.

4.2 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     jal generate_random
10
11    # store and process return value
12    addi t1 a0 0
13    slli t5 t1 2
14
15    # setup arguments for reverse
16    add a0 t0 x0
17    add a1 s0 x0
18    add a2 t5 x0
19    addi a3 t1 0
20
21    jal reverse
22
23    # 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?

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?