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 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.

2.1

```
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?

`t0` will hold `0x00FF0004`, adding 4 to the initial address. `t1` will hold 36, loading the word from the address `0x00FF0000`. `t2` will hold `0xC`, loading the upper half of the address `0x00FF0004`. `s0` will hold the word at $36 = 0x24$, so `0xDEADB33F`. Finally, `s1` will hold `0xFFFFFFC5`, taking the most significant byte and sign-extending it.

2.2

```
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.

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

3 Lost in Translation

3.1 Translate between the C and RISC-V verbatim.

<table>
<tr><th>C</th><th>RISC-V</th></tr>
<tr><td><pre>
// s0 -> a, s1 -> b
// s2 -> c, s3 -> z
int a = 4, b = 5, c = 6, z;
z = a + b + c + 10;
</pre></td><td><pre>
addi s0, x0, 4
addi s1, x0, 5
addi s2, x0, 6
add s3, s0, s1
add s3, s3, s2
addi s3, s3, 10
</pre></td></tr>
<tr><td><pre>
// s0 -> int * p = intArr;
// s1 -> a;
*p = 0;
int a = 2;
p[1] = p[a] = a;
</pre></td><td><pre>
sw   x0, 0(s0)
addi s1, x0, 2
sw   s1, 4(s0)
slli t0, s1, 2
add t0, t0, s0
sw   s1, 0(t0)
</pre></td></tr>
<tr><td><pre>
// s0 -> a, s1 -> b
int a = 5, b = 10;
if(a + a == b) {
    a = 0;
} else {
    b = a - 1;
}
</pre></td><td><pre>
    addi s0, x0, 5
    addi s1, x0, 10
    add t0, s0, s0
    bne t0, s1, else
    xor s0, x0, x0
    jal x0, exit
else:
    addi s1, s0, -1
exit:
</pre></td></tr>
<tr><td><pre>
// computes s1 = 2^30
// assume int s1, s0; was declared above
s1 = 1;
for(s0 = 0; s0 != 30; s0++) {
    s1 *= 2;
}
</pre></td><td><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></td></tr>
</table>

```c
// s0 -> n, s1 -> sum
// assume n > 0 to start
for(int sum = 0; n > 0; n--) {
  sum += n;
}
```

```
    addi s1, x0, 0
loop:
    beq  s0, x0, exit
    add  s1, s1, s0
    addi  s0, s0, -1
    jal  x0, loop
exit:
```

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 **zero**	N/A
`sp`	Stores the **stack pointer**	Callee
`ra`	Stores the **return address**	Caller
`a0 - a7`	Stores **arguments** and **return values**	Caller
`t0 - t7`	Stores **temporary** values that *do not persist* after function calls	Caller
`s0 - s11`	Stores **saved** values that *persist* 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:

```
# Prologue
addi sp sp -8 # Room for two registers. (Why?)
sw s0 0(sp) # Save s0 (or any saved register)
sw s1 4(sp) # Save s1 (or any saved register)

# Code omitted

# Epilogue

lw s0 0(sp) #Load s0 (or any saved register)
lw s1 4(sp) #Load s1 (or any saved register)
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:

```
main:
# Prologue
# Saves ra

# Code omitted
addi s0 x0 5
# Breakpoint 1
jal ra foo
```

```
# Breakpoint 3
mul a0 a0 s0
# Code omitted

# Epilogue
# Restores ra
j exit
```

```
foo:
# Preamble
# Saves s0

# Code omitted
addi s0 x0 4
# Breakpoint 2

# Epilogue
# Restores s0
jr ra
```

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

Yes, since `foo` saves the saved registers, and `main` saves the return address

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

5, then 4, then 5

(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?

5, then 4, then still 4. This would cause errors, since we use the value of `s0` in our calculations.

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.

```
myfunc:
    # Prologue (omitted)

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

    # Save the registers in 4.2
    jal generate_random
    # 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?

`s0`, `s3`, `s9`, `ra`, `s7`, and `s8` We must save all s-registers we modify (note that since `s7` and `s8` were used, it is assumed that they were modified in omitted code), 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`, `s7`, and `s8`

This mirrors what we saved in the prologue.