1   RISC-V: A Rundown

RISC-V is an assembly language, which is comprised of simple instructions that each do a single task such as addition or storing a chunk of data to memory.

For example, on the left is a line of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

```
int x = 5;
y[2];
y[0] = x;
y[1] = x * x;
```

```
// x -> s0, &y -> s1
addi s0, x0, 5
sw   s0, 0(s1)
mul  t0, s0, s0
sw   t0, 4(s1)
```

1.1   Can you figure out what each line in the RISC-V code is doing?

2   Registers

In RISC-V, we have two methods of storing data: main memory and registers. Registers are much faster than using main memory, but are very limited in space (32 bits each). You should ALWAYS use the names of registers, e.g. s0 rather than x8; the one exception to this rule is the zero register x0, as it is often shorter to write x0 than its name zero, and the purpose of the register is still easy to tell with either identifier. The below table of register names is reproduced from the RISC-V green card.

Register(s)	Alt.	Description
x0	zero	The zero register, always zero
x1	ra	The return address register, stores where functions should return
x2	sp	The stack pointer, where the stack ends
x5-x7, x28-x31	t0-t6	The temporary registers
x8-x9, x18-x27	s0-s11	The saved registers
x10-x17	a0-a7	The argument registers, a0-a1 are also return value

2.1   Can you convert each instruction's registers to the other form?

```
add s0, zero, a1    -->
or  x18, x1, x30    -->
```

3   Basic Instructions

For your reference, here are some of the basic instructions for arithmetic operations and dealing with memory (Note: ARG1 is argument register 1, ARG2 is argument register 2, and DR is destination register):

[inst]	[destination register] [argument register 1] [argument register 2]
add	Adds the two argument registers and stores in destination register
xor	Exclusive or's the two argument registers and stores in destination register
mul	Multiplies the two argument registers and stores in destination register
sll	Logical left shifts ARG1 by ARG2 and stores in DR
srl	Logical right shifts ARG1 by ARG2 and stores in DR
sra	Arithmetic right shifts ARG1 by ARG2 and stores in DR
slt/u	If ARG1 < ARG2, stores 1 in DR, otherwise stores 0, u does unsigned comparison
[inst]	[register] [offset]([register containing base address])
sw	Stores the contents of the register to the address+offset in memory
lw	Takes the contents of address+offset in memory and stores in the register
[inst]	[argument register 1] [argument register 2] [label]
beq	If ARG1 == ARG2, moves to label
bne	If ARG1 != ARG2, moves to label
[inst]	[destination register] [label]
jal	Stores the next instruction's address into DR and moves to label

You may also see that there is an "i" at the end of certain instructions, such as addi, slli, etc. This means that ARG2 becomes an "immediate" or an integer instead of using a register. There are also immediates in some other instructions such as **sw** and **lw**. Note that the size (maximum number of bits) of an immediate in any given instruction depends on what type of instruction it is (more on this soon!).

3.1   Assume we have an array in memory that contains int *arr = {1,2,3,4,5,6,0}. Let register s0 hold the address of the element at index 0 in arr. You may assume integers are four bytes and our values are word-aligned. What do the snippets of RISC-V code do? Assume that all the instructions are run one after the other in the same context.

```
a) lw   t0, 12(s0)        -->

b) sw   t0, 16(s0)        -->

c) slli t1, t0, 2
   add  t2, s0, t1
   lw   t3, 0(t2)         -->
   addi t3, t3, 1
   sw   t3, 0(t2)

d) lw   t0, 0(s0)
   xori t0, t0, 0xFFF     -->
   addi t0, t0, 1
```

4   C to RISC-V

4.1   Translate between the C and RISC-V verbatim.

C	RISC-V

<pre>// s0 -> a, s1 -> b
// s2 -> c, s3 -> z
int a = 4, b = 5, c = 6, z;
z = a + b + c + 10;</pre>	

<pre>// s0 -> int * p = intArr;
// s1 -> a;
*p = 0;
int a = 2;
p[1] = p[a] = a;</pre>	

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

5   RISC-V with Arrays and Lists

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;
}
```

5.1
```
lw  t0, 0(s0)
lw  t1, 8(s0)
add t2, t0, t1
sw  t2, 4(s0)
```

5.2
```
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:
```

5.3
```
        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:
```

6   RISC-V Calling Conventions

6.1   How do we pass arguments into functions?

6.2   How are values returned by functions?

6.3   What is `sp` and how should it be used in the context of RISC-V functions?

6.4   Which values need to saved by the caller, before jumping to a function using `jal`?

6.5   Which values need to be restored by the callee, before returning from a function?

6.6   In a bug-free program, which registers are guaranteed to be the same after a function call? Which registers aren't guaranteed to be the same?

7   Writing RISC-V Functions

7.1 Write a function sumSquare in RISC-V that, when given an integer n, returns the summation below. If n is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

For this problem, you are given a RISC-V function called square that takes in a single integer and returns its square.

First, let's implement the meat of the function: the squaring and summing. We will be abiding by the caller/callee convention, so in what register can we expect the parameter n? What registers should hold square's parameter and return value? In what register should we place the return value of sumSquare?

7.2 Since sumSquare is the callee, we need to ensure that it is not overriding any registers that the caller may use. Given your implementation above, write a prologue and epilogue to account for the registers you used.