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 \( x \). \( \text{lw } s0, 4(a0) \) will always load \( x[1] \) into \( s0 \).

1.2 Assuming integers are 4 bytes, adding the ASCII character ’d’ to the address of an integer array would get you the element at index 25 of that array (assuming the array is large enough).

1.3 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.

1.4 \( \text{jalr} \) is a shorthand expression for a \( \text{jal} \) that jumps to the specified label and does not store a return address anywhere.

1.5 Calling \( \text{j label} \) does the exact same thing as calling \( \text{jal label} \).
2 Instructions

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 snippet of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

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

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

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):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>Adds the two argument registers and stores in destination register</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive or's the two argument registers and stores in destination register</td>
</tr>
<tr>
<td>mul</td>
<td>Multiplies the two argument registers and stores in destination register</td>
</tr>
<tr>
<td>sll</td>
<td>Logical left shifts ARG1 by ARG2 and stores in DR</td>
</tr>
<tr>
<td>srl</td>
<td>Logical right shifts ARG1 by ARG2 and stores in DR</td>
</tr>
<tr>
<td>sra</td>
<td>Arithmetic right shifts ARG1 by ARG2 and stores in DR</td>
</tr>
<tr>
<td>slt/u</td>
<td>If ARG1 &lt; ARG2, stores 1 in DR, otherwise stores 0, u does unsigned comparison</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>Stores the contents of the register to the address+offset in memory</td>
</tr>
<tr>
<td>lw</td>
<td>Takes the contents of address+offset in memory and stores in the register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>If ARG1 == ARG2, moves to label</td>
</tr>
<tr>
<td>bne</td>
<td>If ARG1 != ARG2, moves to label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jal</td>
<td>Stores the next instruction's address into DR and moves to label</td>
</tr>
</tbody>
</table>

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!).

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) \texttt{lw\ t0, 12(s0)} \rightarrow

b) \texttt{sw\ t0, 16(s0)} \rightarrow

c) \texttt{slli t1, t0, 2}
   \quad \texttt{add\ t2, s0, t1}
   \quad \texttt{lw\ t3, 0(t2)} \rightarrow
   \quad \texttt{addi\ t3, t3, 1}
   \quad \texttt{sw\ t3, 0(t2)}

d) \texttt{lw\ t0, 0(s0)}
   \quad \texttt{xori\ t0, t0, 0xFFF} \rightarrow
   \quad \texttt{addi\ t0, t0, 1}

2.2 Assume that \textit{s0} and \textit{s1} contain signed integers. Without any pseudoinstructions, how can we branch on the following conditions to jump to some LABEL?

\begin{align*}
\text{s0} < \text{s1} & \quad \text{s0} \neq \text{s1} & \quad \text{s0} \leq \text{s1} & \quad \text{s0} > \text{s1}
\end{align*}

3 Lost in Translation

3.1 Translate between the C and RISC-V verbatim.

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{// s0 -&gt; a, s1 -&gt; b}</td>
</tr>
<tr>
<td>\texttt{// s2 -&gt; c, s3 -&gt; z}</td>
</tr>
<tr>
<td>\texttt{int a = 4, b = 5, c = 6, z;}</td>
</tr>
<tr>
<td>\texttt{z = a + b + c + 10;}</td>
</tr>
<tr>
<td>\texttt{// s0 -&gt; int * p = intArr;}</td>
</tr>
<tr>
<td>\texttt{// s1 -&gt; a;}</td>
</tr>
<tr>
<td>\texttt{*p = 0;}</td>
</tr>
<tr>
<td>\texttt{int a = 2;}</td>
</tr>
<tr>
<td>\texttt{p[1] = p[a] = a;}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lw\ t0, 0(s0)}</td>
</tr>
<tr>
<td>\texttt{xori t0, t0, 0xFFF} \rightarrow</td>
</tr>
<tr>
<td>\texttt{addi t0, t0, 1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{// s0 -&gt; a, s1 -&gt; b}</td>
</tr>
<tr>
<td>\texttt{int a = 5, b = 10;}</td>
</tr>
<tr>
<td>\texttt{if(a + a == b) {}}</td>
</tr>
<tr>
<td>\texttt{\quad a = 0;}</td>
</tr>
<tr>
<td>\texttt{}}</td>
</tr>
<tr>
<td>\texttt{else {}}</td>
</tr>
<tr>
<td>\texttt{\quad b = a - 1;}</td>
</tr>
<tr>
<td>\texttt{}}</td>
</tr>
</tbody>
</table>
\[\text{addi } s0, x0, 0\]
\[\text{addi } s1, x0, 1\]
\[\text{addi } t0, x0, 30\]
\[\text{loop:}\]
\[\text{beq } s0, t0, \text{exit}\]
\[\text{add } s1, s1, s1\]
\[\text{addi } s0, s0, 1\]
\[\text{jal } x0, \text{loop}\]
\[\text{exit:}\]

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

4 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.

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

4.2 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:
4.3

```
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:
```

5 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.

5.1

```
li x5 0x00FF0000
lw x6 0(x5) 0x000F0000
addi x5 x5 4 0x00FF0004
lh x7 2(x5) 0x00FF0000 0x561C
lw x8 0(x6) 0x00000036 0xFDFDFDFD
lb x9 3(x7) 0x00000024 0xDEADB33F
```

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

```
0x00000000
```

5.2

```
li x5 0xABADCAFE
li x6 0x9F120504
li x7 0xBEFCAE
sw x5 0(x6) 0xF9120504
addi x6 x6 4 0xABADCAFE
addi x5 x5 4
sh x6 2(x5)
sh x7 1(x7)
sb x7 3(x6)
sb x7 3(x5)
```

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