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 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).

True. There is no fundamental difference between integers, strings, and memory addresses in RISC-V (they’re all bags of bits), so it’s possible to manipulate data in this way. (We don’t recommend it, though).

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.

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).

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

False. j label is a pseudo-instruction for jal x0, label. 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.

1.5 Calling j label does the exact same thing as calling jal label.

False. As from the previous problem, j label is short for jal x0, label — since it’s writing the return address to x0, it’s effectively discarding it since we have no need to jump back to the original PC. jal label is short for jal ra, 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;
```

```risc-v
// 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>add</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>
<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>
<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>
<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) lw  t0, 12(s0) --> Sets t0 equal to arr[3]
b) sw  t0, 16(s0) --> Stores t0 into arr[4]
c) slli t1, t0, 2
   add  t2, s0, t1
   lw  t3, 0(t2) --> Increments arr[t0] by 1
   addi t3, t3, 1
   sw  t3, 0(t2)
d) lw  t0, 0(s0)
   xori t0, t0, 0xFFF --> Sets t0 to -1 * arr[0]
   addi t0, t0, 1

Sets t0 equal to arr[3]
Stores t0 into arr[4]
Increments arr[t0] by 1
Sets t0 to -1 * arr[0]

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

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

blt s0, s1, LABEL  bne s0, s1, LABEL  bge s1, s0, LABEL  blt s1, s0, LABEL

Note that RISC-V does not provide a bgt instruction because you can manipulate the blt instruction to get an equivalent result. Also note that the above solutions assume that s0 and s1 contained signed integers. If they are unsigned, then we would use the unsigned variants of the above commands (namely, bltu, bgeu).

3 Lost in Translation

3.1 Translate between the C and RISC-V verbatim.

<table>
<thead>
<tr>
<th>C</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>// s0 -&gt; a, s1 -&gt; b</td>
<td>addi s0, x0, 4</td>
</tr>
<tr>
<td>// s2 -&gt; c, s3 -&gt; z</td>
<td>addi s1, x0, 5</td>
</tr>
<tr>
<td>int a = 4, b = 5, c = 6, z;</td>
<td>addi s2, x0, 6</td>
</tr>
<tr>
<td>z = a + b + c + 10;</td>
<td>add s3, s0, s1</td>
</tr>
<tr>
<td></td>
<td>add s3, s3, s2</td>
</tr>
<tr>
<td></td>
<td>addi s3, s3, 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>// s0 -&gt; int * p = intArr;</td>
<td>sw  x0, 0(s0)</td>
</tr>
<tr>
<td>// s1 -&gt; a;</td>
<td>addi s1, x0, 2</td>
</tr>
<tr>
<td>*p = 0;</td>
<td>sw  s1, 4(s0)</td>
</tr>
<tr>
<td>int a = 2;</td>
<td>slli t0, s1, 2</td>
</tr>
<tr>
<td>p[1] = p[a] = a;</td>
<td>add t0, t0, s0</td>
</tr>
<tr>
<td></td>
<td>sw  s1, 0(t0)</td>
</tr>
</tbody>
</table>
// s0 -> a, s1 -> b
int a = 5, b = 10;
if(a + a == b) {
    a = 0;
} else {
    b = a - 1;
}

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:

// computes s1 = 2^30
// assume int s1, s0; was declared above
s1 = 1;
for(s0 = 0; s0 != 30; s0++) {
    s1 *= 2;
}

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:

// 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
    add s0, s0, -1
    jal x0, loop
exit:

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.

struct ll {
    int val;
    struct ll* next;
}

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

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

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:

Increments all values in the linked list by 1.

4.3  add  t0, x0, x0
     loop: slti t1, t0, 6
           beq t1, x0, end
           slti 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.

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)
     addi x5 x5  4
     lh  x7  2(x5)
     lw  x8  0(x6)
     lb  x9  3(x7)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x6</td>
<td>0x00FF0004</td>
</tr>
<tr>
<td>x7</td>
<td>0x00FF0000</td>
</tr>
<tr>
<td>x8</td>
<td>0x00000036</td>
</tr>
<tr>
<td>x9</td>
<td>0x00000024</td>
</tr>
</tbody>
</table>

What value does each register hold after the code is executed?
x5 will hold 0x00FF0004, after 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 0xFFFFFFFFC5, taking the most significant byte and sign-extending it.

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>0xF9120508</td>
<td>0xCE000000</td>
</tr>
<tr>
<td>0xF9120504</td>
<td>0xABADCAFE</td>
</tr>
<tr>
<td>0xBEEFCAD2</td>
<td></td>
</tr>
<tr>
<td>0xBEEFCACE</td>
<td>0x0000CE00</td>
</tr>
<tr>
<td>0xABADCB02</td>
<td>0xCE080000</td>
</tr>
<tr>
<td>0xABADCAFE</td>
<td></td>
</tr>
<tr>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>