

1 RISC-V Instructions

1.1 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 following 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)`
`addi t3, t3, 1`
`sw t3, 0(t2)`

Increments `arr[4]` by 1.

1st line sets `t1 = 16`

2nd line adds it to `s0` so that it now points at `arr[4]`

3rd-5th line loads the value at `arr[4]`, increments by one, and stores it back

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

Sets `t0` to `-1 * arr[0]`

2 Lost in Translation

2.1 Translate the code verbatim between C and RISC-V. The comments above the code indicate which registers to store the variables.

C	RISC-V
<pre>// s0 -> a // s1 -> b // s2 -> c // s3 -> z int a = 4, b = 5, c = 6; int z = a + b + c + 10;</pre>	<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>
<pre>// int *p = intArr; // s0 -> p; // s1 -> a; *p = 0; int a = 2; p[1] = p[a] = a;</pre>	<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>
<pre>// s0 -> a, // s1 -> b int a = 5; int b = 10; if (a + a == b) { a = 0; } else { b = a - 1; }</pre>	<pre>start: addi s0, x0, 5 addi s1, x0, 10 add t0, s0, s0 bne t0, s1, else add s0, x0, x0 jal x0, exit else: addi s1, s0, -1 exit: ...</pre>
<pre>// Compute s1 = 2^30 int s0 = 0; int s1 = 1; for (; s0 != 30; s0 += 1) { s1 *= 2; }</pre>	<pre>start: addi s0, x0, 0 addi s1, x0, 1 addi t0, x0, 30 loop: beq s0, t0, exit slli s1, s1, 1 addi s0, s0, 1 jal x0, loop exit: ...</pre>

C	RISC-V
<pre>// s0 -> n // s1 -> sum for (int sum = 0; n > 0; n--) { sum += n; }</pre>	<pre>start: addi s1, x0, 0 loop: beq s0, x0, exit add s1, s1, s0 addi s0, s0, -1 jal x0, loop exit: ...</pre>

3 RISC-V Memory Access

For Q3.1 – Q3.2, use the instructions and memory to figure out what the code does. Recall that RISC-V is little-endian and byte addressable. For any unknown instructions, use the [CS 61C reference card!](#)

- 3.1 Fill in the registers with the values they contain after the code finishes executing.

```
li t0 0x00FF0000
lw t1 0(t0)
addi t0 t0 4
lh t2 2(t0)
lw s0 0(t1)
lb s1 3(t2)
```

t0	0x00FF0004	0xFFFFFFFF	
t1	36	0x00FF0004	0x000C561C
t2	0x0000000C	0x00FF0000	36
s0	0xDEADB33F	0x00000036	0xFDFDFDFD
s1	0xFFFFFC5	0x00000024	0xDEADB33F
		0x0000000C	0xC5161C00
		0x00000000	

- **t0**: Line 3 adds 4 to the initial address.
- **t1**: Line 2 loads the 4-byte word from address 0x00FF0000.
- **t2**: Line 4 loads two bytes starting at the address $0x00FF0004 + 2 = 0x00FF0006$. This returns 0x000C, sign-extended to 0x0000000C.
- **s0**: Line 5 loads the word starting at address $36 = 0x24$ which is 0xDEADB33F.
- **s1**: Line 6 loads the MSB starting of the 4-byte word at address 0xC. The value is 0xC5 which is sign-extended to 0xFFFFFC5.

- 3.2 Fill in the memory diagram and `t3` register with the values contained in them after the code finishes executing. The values in the `t0`, `t1`, and `t2` registers at the start of program execution have been provided to you. Assume that all memory starts out initialized to zeros.

```
sw t0 0(t1)
addi t0 t0 4
sh t1 2(t0)
sh t2 0(t0)
lw t3 0(t1)
sb t1 1(t3)
sb t2 3(t3)
```

t0	0xABADCAF8	0xFFFFFFFF	0x00000000
			...
t1	0xF0120504	0xF0120504	0xABADCAF8
			...
t2	0xBEEFDAB0	0xBEEFDAB0	0x00000000
			...
t3	0xABADCAF8	0xABADCAFC	0x0504DAB0
		0xABADCAF8	0xB0000400
			...
		0x00000000	0x00000000