RISC-V

Discussion 3

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`.

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.

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

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, RISC-V code accomplishes the same task as the C code, on the right, with its streamlined instructions.

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

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

For your reference, here are some of the basic instructions for arithmetic/bitwise operations and memory access (Note: rs1 is argument register 1, rs2 is argument register 2, and rd 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 rs1 by rs2 and stores in rd
srl	Logical right shifts rs1 by rs2 and stores in rd
sra	Arithmetic right shifts rs1 by rs2 and stores in rd
slt/u	If rs1 < rs1, stores 1 in rd, 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 rs1 == rs2, moves to label
bne	If rs1 != rs2, moves to label

[inst]	[destination register] [label]
jal	Stores the next instruction's address into rd 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 rs2 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!).

2.1 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
```

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?

$s0 < s1$ $s0 \neq s1$ $s0 \leq s1$ $s0 > s1$

3 Lost in Translation

3.1 Translate between RISC-V and C verbatim.

<table>
<tr><th>RISC-V</th><th>C</th></tr>
<tr><td></td><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></tr>
<tr><td></td><td><pre>
// s0 -> int * p = intArr;
// s1 -> a;
*p = 0;
int a = 2;
p[1] = p[a] = a;
</pre></td></tr>
<tr><td></td><td><pre>
// s0 -> a, s1 -> b
int a = 5, b = 10;
if(a + a == b) {
    a = 0;
} else {
    b = a - 1;
}
</pre></td></tr>
<tr><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><td></td></tr>
<tr><td></td><td><pre>
// s0 -> n, s1 -> sum
// assume n > 0 to start
for(int sum = 0; n > 0; n--) {
  sum += n;
}
</pre></td></tr>
</table>

4 C Generics

4.1 **True or False:** In C, if the variable `ptr` is a generic pointer, then it is still possible to dereference `ptr` when used on the right-hand side of an assignment operator, e.g., `*ptr =...`

4.2 Generic functions (i.e., generics) in C use void * pointers to operate on memory without the restriction of types. Such generics pointers do not support dereferencing, as the number of bytes to access from memory is not known at compile-time. They instead use byte handling functions such as `memcpy` and `memmove`.

Implement `rotate`, which will prompt the following program to generate the provided output.

```
int main(int argc, char *argv[]) {
  int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  print_int_array(array, 10);
  rotate(array, array + 5, array + 10);
  print_int_array(array, 10);
  rotate(array, array + 1, array + 10);
  print_int_array(array, 10);
  rotate(array + 4, array + 5, array + 6);
  print_int_array(array, 10);
  return 0;
}
```

Output:

```
$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
```

Your Solution:

```
void rotate(void *front, void *separator, void *end) {
}
```