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 $x$. $\text{lws0, 4(a0)}$ 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 $\text{jalr}$ is a shorthand expression for a $\text{jal}$ 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):
### 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw t0, 12(s0)</code></td>
<td>Stores the contents of address <code>12</code> at offset <code>s0</code> in memory and stores in <code>t0</code></td>
</tr>
<tr>
<td><code>sw t0, 16(s0)</code></td>
<td>Stores the contents of register <code>t0</code> to the address <code>16</code> at offset <code>s0</code> in memory</td>
</tr>
<tr>
<td><code>slli t1, t0, 2</code></td>
<td>Logical left shifts <code>t1</code> by <code>2</code> and stores in <code>t0</code></td>
</tr>
<tr>
<td><code>sw t3, 0(t2)</code></td>
<td>Stores the contents of address <code>0</code> at offset <code>t2</code> in memory and stores in <code>t0</code></td>
</tr>
<tr>
<td><code>addi t3, t3, 1</code></td>
<td>Adds <code>1</code> to <code>t3</code> and stores in <code>t3</code></td>
</tr>
<tr>
<td><code>sw t3, 0(t2)</code></td>
<td>Stores the contents of address <code>0</code> at offset <code>t2</code> in memory and stores in <code>t0</code></td>
</tr>
</tbody>
</table>

### 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?
3 Lost in Translation

3.1 Translate between RISC-V and C verbatim.

<table>
<thead>
<tr>
<th>RISC-V</th>
<th>C</th>
</tr>
</thead>
</table>
| ``` | ```
| // s0 -> a, s1 -> b  
| // s2 -> c, s3 -> z  
| ```
| ``` | ```
| int a = 4, b = 5, c = 6, z;  
| z = a + b + c + 10;  
| ```
| ``` | ```
| // s0 -> int * p = intArr;  
| // s1 -> a;  
| ```
| ``` | ```
| *p = 0;  
| int a = 2;  
| p[1] = p[a] = a;  
| ```
| ``` | ```
| // s0 -> a, s1 -> b  
| int a = 5, b = 10;  
| if(a + a == b) {  
| a = 0;  
| } else {  
| b = a - 1;  
| ```
| ``` | ```
| ``` | ```
| 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;  
| ```
| ``` | ```
| ``` | ```
| ``` | ```
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.

```c
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, 2, 1, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
```

Your Solution:

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