1.) Consider the original, single-cycle (i.e. non-pipelined) Laundry-o-matic machine discussed in class. The version in class had 3 phases (washing, drying, and folding) that were all performed in a single magical machine.

Suppose that the folding phase has increased dramatically because it now includes ironing. The phases now take the following number of minutes:

<table>
<thead>
<tr>
<th>Phase</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Washing</td>
<td>30</td>
</tr>
<tr>
<td>Drying</td>
<td>40</td>
</tr>
<tr>
<td>Folding</td>
<td>70</td>
</tr>
</tbody>
</table>

What is the latency for a single load of laundry on this machine? What is the latency for 3 loads of laundry?

\[
\text{Single load} = 30 + 40 + 70 = 140 \\
\text{Three loads} = 3 \times 140 = 420
\]

2.) Now suppose we divided the Laundry-o-matic into 3 distinct units (i.e. washing, drying, and folding) as we did in class to create a pipelined version, where it again takes 6 minutes to switch between stages.

What is the latency for a single load of laundry to complete on this pipelined machine? What is the latency for 3 loads of laundry?

\[
\text{Single load} = (70+6) \times 3 = 228 \\
\text{Three loads} = 228 + 76 + 76 = 380
\]

3.) Suppose we were considering dividing our folding stage into 2 stages, one for ironing and one for folding. Our pipelined machine now has the following 4 units/stages with the following times:

<table>
<thead>
<tr>
<th>Phase</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Washing</td>
<td>30</td>
</tr>
<tr>
<td>Drying</td>
<td>40</td>
</tr>
<tr>
<td>Ironing</td>
<td>50</td>
</tr>
<tr>
<td>Folding</td>
<td>20</td>
</tr>
</tbody>
</table>

The time to switch between stages is still 6 minutes.

What is the latency for a single load of laundry to complete on this pipelined machine with 4 stages? What is the latency for 4 loads of laundry?

\[
\text{Single load} = (50+6) \times 4 = 224 \\
\text{Four loads} = 224 + 56 + 56 + 56 = 392
\]
4.) Rewrite the following x86-64 instruction in equivalent Y86-64 assembly.

```
addq %r8, 4(%rsp)
    mrmovq 4(%rsp), %r9
    addq %r8, %r9
    rmmovq %r9, 4(%rsp)
```
5.) Rewrite the following function written in Y86-64 assembly into equivalent C code.

```
fcn:
    irmovq $0, %rax
    andq %rdi, %rdi
    je done
    jmp cnd
loop:
    addq %r8, %rax
    irmovq $16, %r9
    addq %r9, %rdi
cnd:
    mrmovq (%rdi), %r8
    andq %r8, %r8
    jne loop
done:
    ret
```

You should assume that `fcn` is passed the address of an array of longs.

`Notes: using the standard C calling conventions, the first function argument is always stored in register %rdi, the second function argument in register %rsi, and the return value in register %rax.`

```
long fcn(long *array)
{
    long result = 0;
    if(array == 0){
        return result;
    }
    while(*array != 0){
        result += *array;
        array = array+2;
    }
    return result;
}
```