Q1. Explain the following assembly instructions in words:

```assembly
mov %rax, %r15
add %r9, %rdx
mov $10, %rax
add $0xF, %rdx
mov $20, (%rax)
movl %rax, -16(%rbp)
```

Q2. Let's try some more examples:

What will the state of registers and memory look like after executing these instructions?

```assembly
sub  $16, %rsp
movq  $3, -8(%rbp)
mov  $10, %rax
sal  $1, %rax
add  -8(%rbp), %rax
movq %rax, -16(%rbp)
ad   $16, %rsp
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Address</td>
</tr>
<tr>
<td>%rax</td>
<td>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x1FFF000AD8</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</td>
</tr>
</tbody>
</table>

x is stored at rbp-8
y is stored at rbp-16
What will the state of registers and memory look like after executing these instructions?

```assembly
... mov %rbp, %rcx sub $8, %rcx movq (%rcx), %rax or %rax, -16(%rbp) neg %rax
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Value</td>
</tr>
<tr>
<td>%rax</td>
<td>0</td>
</tr>
<tr>
<td>%rcx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</td>
</tr>
</tbody>
</table>

How might you implement the following C code in assembly?

\[ z = x \oplus y \]

x is stored at %rbp-8
y is stored at %rbp-16
z is stored at %rbp-24

A: movq -8(%rbp), %rax movq -16(%rbp), %rdx xor %rax, %rdx movq %rax, -24(%rbp)

B: movq -8(%rbp), %rax movq -16(%rbp), %rdx xor %rdx, %rax movq %rax, -24(%rbp)

C: movq -8(%rbp), %rax movq -16(%rbp), %rdx xor %rax, %rdx movq %rax, -8(%rbp)

D: movq -24(%rbp), %rax movq -16(%rbp), %rdx xor %rdx, %rax movq %rax, -8(%rbp)
How might you implement the following C code in assembly?

\[ x = y \gg 3 \mid x \ast 8 \]

- \( x \) is stored at %rbp-8
- \( y \) is stored at %rbp-16
- \( z \) is stored at %rbp-24

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Value</td>
</tr>
<tr>
<td>%rax</td>
<td>0</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rip</td>
<td>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</td>
</tr>
</tbody>
</table>

Which flags would this \texttt{cmp} set?

Suppose %rax holds 5, %rcx holds 7

\texttt{cmp %rcx, %rax}

- If the result is zero (ZF)
- If the result’s first bit is set (negative if signed) (SF)
- If the result overflowed (assuming unsigned) (CF)
- If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF
How could we use jumps/CCs to implement this C code?

```c
long userval;
scanf("%ld", &userval);

if (userval == 42) {
    userval = userval + 5;
} else {
    userval = userval - 10;
}
```

(A) `cmp $42, %rax` `jne L2`
    `L1:` `sub $10, %rax` `jmp DONE`
    `L2:` `add $5, %rax` `DONE:
```

(B) `cmp $42, %rax` `jne L2`
    `L1:` `sub $10, %rax` `jmp DONE`
    `L2:` `add $5, %rax` `DONE:
```

(C) `cmp $42, %rax` `jne L2`
    `L1:` `add $5, %rax` `jmp DONE`
    `L2:` `sub $10, %rax` `DONE:
```

Convert to C goto:

```c
x = 0;
for(i=0; i < 10; i++) {
    x = x + 1;
}
z = x * 3;
```

<table>
<thead>
<tr>
<th>for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(init; cond; step){</td>
</tr>
<tr>
<td>loop body</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

| init code |
|<fill in your answer here>