Problem 1 (15 pts): Nearby is a C function `col_update()` with associated data and documentation. Re-implement this function in x86-64 assembly according to the documentation given. Follow the same flow provided in the C implementation. The comments below the `colinfo_t` struct give information about how it lays out in memory and as a packed argument.

Indicate which registers correspond to which C variables.

### SOLUTION:

```assembly
.globl col_update

# YOUR CODE BELOW

col_update:
    movl 0(%rdi),%esi      # cur = info->cur
    movl 4(%rdi),%edx     # step = info->step
    cmpl $0,%esi          # if(cur < 0)
    jle .ERROR
    addl $1,%edx          # step++
    testl $0x01,%esi     # if(cur%2 == 1)
    jz .EVEN             # go to even case
    ## ODD CASE (fall through)
    imull $3,%esi        # odd: cur *= 3
    addl $1,%esi         # odd: cur += 1
    jmp .RETURN          # jump over even

# EVEN:
    sarl $1,%esi         # even: cur /= 2

.RETURN:
    movl %esi,0(%rdi)    # info->cur = cur;
    movl %edx,4(%rdi)    # info->step = step;
    movl $0,%eax         # success
    ret

.ERROR:
    movl $1,%eax         # error case
    ret
```

```c
typedef struct{
  int cur;
  int step;
} colinfo_t;

int col_update(colinfo_t *info){
  // Updates current value and step in colinfo_t pointed by param info.
  int cur = info->cur;
  int step = info->step;
  if(cur <= 0){
    return 1;
  }
  step++;
  if(cur % 2 == 1){
    cur = cur*3+1;
  } else{
    cur = cur / 2;
  }
  info->cur = cur;
  info->step = step;
  return 0;
}
```
Problem 2 (15 pts): Below is an initial register/memory configuration along with snippets of assembly code. Each snippet is followed by a blank register/memory configuration which should be filled in with the values to reflect changes made by the preceding assembly. The code is continuous so that POS A is followed by POS B.

**SOLUTION:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>Value</td>
<td>REG</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>rax</td>
<td>10</td>
<td>rax</td>
</tr>
<tr>
<td>rdi</td>
<td>20</td>
<td>rdi</td>
</tr>
<tr>
<td>rsi</td>
<td>30</td>
<td>rsi</td>
</tr>
<tr>
<td>rsp</td>
<td>#3032</td>
<td>rsi</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>MEM</td>
<td>Value</td>
<td>MEM</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>#3032</td>
<td>250</td>
<td>#3032</td>
</tr>
<tr>
<td>#3028</td>
<td>100</td>
<td>#3028</td>
</tr>
<tr>
<td>#3024</td>
<td>300</td>
<td>#3024</td>
</tr>
<tr>
<td>#3020</td>
<td>3</td>
<td>#3020</td>
</tr>
</tbody>
</table>

**Problem 3 (10 pts):** Rover Witer is writing an assembly function called `compval` which he will use in C programs. He writes a short C `main()` function to test `compval` but is shocked by the results which seem to defy the C and assembly code. Valgrind provides no insight for him. **Identify why** Rover’s code is behaving so strangely and fix `compval` so it behaves correctly.

**Sample Compile / Run:**

```bash
> gcc compval_main.c compval_asm.s
> a.out
expect: 0
actual: 19
expect: 0
actual: 50
```

**SOLUTION:** The `movq` instruction at line 7 of `compval` writes 8 bytes. This is inappropriate as a 4-byte int is supposed to be written. Apparently the stack layout in `main()` has the variable `actual` at a memory address immediately below variable `expect` so that on writing 8 bytes, the low order 4 bytes correctly get written to `actual` but the high order 4 bytes (all 0's for small values) overwrite the variable `expect` leaving it as 0. The fix for this is to use `movl %eax, (%rdx)` which will write 4 bytes, filling only actual.