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.

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

```c
int col_update(colinfo_t *info){
    // Updates current value and step in
    // colinfo_t pointed by param info. If
    // info->cur is invalid, makes no changes
    // and returns 1 to indicate an
    // error. Otherwise performs odd or even
    // update on cur and increments step
    // returning 0 for success.
    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.

```
addl %edi, %esi
subq $8, %rsp
movl $100, 4(%rsp)
addl %esi, (%rsp,%rdi,4)
movl $300, 0(%rsp)
leaq 8(%rsp), %rdi
addl (%rsp), %eax
```

|-------+-------| |-------+-------| |-------+-------|
| REG | Value | | REG | Value | | REG | Value |
|-------+-------| |-------+-------| |-------+-------|
| rax | 10 | | rax | | | rax |
| rdi | 20 | | rdi | | | rdi |
| rsi | 30 | | rsi | | | rsi |
| rsp | #3032 | | rsp | | | rsp |

|-------+-------| |-------+-------| |-------+-------|
| MEM | Value | | MEM | Value | | MEM | Value |
|-------+-------| |-------+-------| |-------+-------|
| #3032 | 250 | | #3032 | | | #3032 |
| #3028 | 1 | | #3028 | | | #3028 |
| #3024 | 2 | | #3024 | | | #3024 |
| #3020 | 3 | | #3020 | | | #3020 |

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:

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

```
// compval_main.c
#include <stdio.h>

void compval(int x, int y, int *val);
// compute something based on x and y
// store result at int pointed to by val
int main(){
    int expect, actual;
    expect = 7 * 2 + 5; // expected value
    compval(7, 2, &actual); // actual result
    printf("expect: %d\n",expect);
    printf("actual: %d\n",actual);
    expect = 5 * 9 + 5; // expected value
    compval(5, 9, &actual); // actual result
    printf("expect: %d\n",expect);
    printf("actual: %d\n",actual);
    return 0;
}
```

```
# compval_asm.s
.text
.global compval
compval:
imulq %rdi,%rsi
addq $5,%rsi
movq %rsi,(%rdx)
ret
```