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;

// | Field | Size | Offset | Bits |
// |-------+------+--------+--------|
// | cur   | 4    | +0     | 0-31   |
// | step  | 4    | +4     | 32-64  |

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)
movl $300, 0(%rsp)
addl (%rsp), %eax
```

<table>
<thead>
<tr>
<th>REG</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>10</td>
</tr>
<tr>
<td>rdi</td>
<td>20</td>
</tr>
<tr>
<td>rsi</td>
<td>30</td>
</tr>
<tr>
<td>rsp</td>
<td>#3032</td>
</tr>
</tbody>
</table>

```
movq $1, %rdi
addl %esi, (%rsp,%rdi,4)
leaq 8(%rsp), %rdi
addl (%rdi), %rax
```

<table>
<thead>
<tr>
<th>REG</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td></td>
</tr>
<tr>
<td>rdi</td>
<td></td>
</tr>
<tr>
<td>rsi</td>
<td></td>
</tr>
<tr>
<td>rsp</td>
<td></td>
</tr>
</tbody>
</table>

INITIAL | POS A | POS B
---------|-------|-------
|-------+-------| |-------+-------|
| REG  | Value | | REG  | Value |
|-------+-------| |-------+-------|
| rax  | 10    | | rax  |       |
| rdi  | 20    | | rdi  |       |
| rsi  | 30    | | rsi  |       |
| rsp  | #3032 | | rsp  |       |

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
```

```
1 // compval_main.c
2 #include <stdio.h>
3
4 void compval(int x, int y, int *val);
5 // compute something based on x and y
6 // store result at int pointed to by val
7 int main(){
8    int expect, actual;
9
10    expect = 7 * 2 + 5; // expected value
11    compval(7, 2, &actual); // actual result
12    printf("expect: %d\n",expect);
13    printf("actual: %d\n",actual);
14
15    expect = 5 * 9 + 5; // expected value
16    compval(5, 9, &actual); // actual result
17    printf("expect: %d\n",expect);
18    printf("actual: %d\n",actual);
19
20    return 0;
21}
```

```
1 // compval_asm.s
2 .text
3 .global compval
4 compval:
5   imulq %rdi,%rsi
6   addq $5,%rsi
7   movq %rsi,(%rdx)
8   ret
```