

**CMSC216: Practice Exam 2B SOLUTION**

Spring 2025

University of Maryland

Exam period: 20 minutes

Points available: 40

**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.

```

1 ### SOLUTION:
2 .text
3 .globl      col_update
4
5 # YOUR CODE BELOW
6 col_update:
7     movl     0(%rdi),%esi    # cur = info->cur
8     movl     4(%rdi),%edx    # step= info->step
9     cmpl     $0,%esi        # if(cur < 0)
10    jle      .ERROR
11    addl     $1,%edx         # step++
12    testl    $0x01,%esi     # if(cur%2 == 1)
13    jz       .EVEN          # go to even case
14 ## ODD CASE (fall through)
15    imull    $3,%esi        # odd: cur *= 3
16    addl     $1,%esi        # odd: cur += 1
17    jmp      .RETURN        # jump over even
18 .EVEN:
19    sarl     $1,%esi        # even: cur /= 2
20 .RETURN:
21    movl     %esi,0(%rdi)    # info->cur = cur;
22    movl     %edx,4(%rdi)    # info->step= step;
23    movl     $0,%eax        # success
24    ret
25 .ERROR:
26    movl     $1,%eax        # error case
27    ret                    # return 1

```

```

1 typedef struct{
2     int cur;
3     int step;
4 } colinfo_t;
5 // |      | Byte |   Byte | Packed |
6 // | Field | Size | Offset | Bits |
7 // |-----+-----+-----+-----|
8 // | cur   |   4 |    +0 | 0-31 |
9 // | step  |   4 |    +4 | 32-63 |
10
11 int col_update(colinfo_t *info){
12     // Updates current value and step in
13     // colinfo_t pointed by param info. If
14     // info->cur is invalid, makes no changes
15     // and returns 1 to indicate an
16     // error. Otherwise performs odd or even
17     // update on cur and increments step
18     // returning 0 for success.
19
20     int cur = info->cur;
21     int step = info->step;
22     if(cur <= 0){
23         return 1;
24     }
25     step++;
26     if(cur % 2 == 1){
27         cur = cur*3+1;
28     }
29     else{
30         cur = cur / 2;
31     }
32     info->cur = cur;
33     info->step = step;
34     return 0;
35 }

```

**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:			addl %edi, %esi			movq \$1, %rdi		
			subq \$8, %rsp			addl %esi, (%rsp,%rdi,4)		
			movl \$100, 4(%rsp)			leaq 8(%rsp), %rdi		
			movl \$300, 0(%rsp)			addl (%rdi), %eax		
			addl (%rsp), %eax					
INITIAL			# POS A			# POS B		
-----+-----			-----+-----			-----+-----		
REG   Value			REG   Value			REG   Value		
-----+-----			-----+-----			-----+-----		
rax   10			rax   310			rax   560		
rdi   20			rdi   20			rdi   #3032		
rsi   30			rsi   50			rsi   50		
rsp   #3032			rsp   #3024			rsp   #3024		
-----+-----			-----+-----			-----+-----		
MEM   Value			MEM   Value			MEM   Value		
-----+-----			-----+-----			-----+-----		
#3032   250			#3032   250			#3032   250		
#3028   1			#3028   100			#3028   150		
#3024   2			#3024   300			#3024   300		
#3020   3			#3020   3			#3020   3		
-----+-----			-----+-----			-----+-----		

**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 ...
> a.out
expect: 0
actual: 19
expect: 0
actual: 50
```

	REGS	VALS	
	rdi	#2032	-+
	STACK		
expect	expect	#2036	19->0
actual	actual	#2032	19

**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`.

```
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
8 int main(){
9     int expect, actual;
10
11     expect = 7 * 2 + 5; // expected value
12     compval(7, 2, &actual); // actual result
13     printf("expect: %d\n", expect);
14     printf("actual: %d\n", actual);
15
16     expect = 5 * 9 + 5; // expected value
17     compval(5, 9, &actual); // actual result
18     printf("expect: %d\n", expect);
19     printf("actual: %d\n", actual);
20
21     return 0;
22 }
```

```
1 # compval_asm_corrected.s
2 .text
3 .global compval
4 compval:
5     imulq    %rdi,%rsi # imul/add should be 32-bit
6     addq     $5,%rsi   # biggest problem is movq
7     movl     %esi, (%rdx) # likely: imull %edi, %esi
8     ret          # likely: addl $5, %esi
9                  # was movq %rsi, (%rdx)
10                 # MUST do a movl, not movq
```