Consider the following C code, similar to what we considered in slideset 6.

```c
long array[] = {0xd, 0xc0, 0xb00, 0xa000, 0};

/* Count elements in null-terminated list */
long len1( long a[] )
{
    long len;
    for (len = 0; a[len]; len++ );
    return len;
}

main()
{
    long ans;
    ans = len1( array );
    printf( "Answer: %ld\n", ans );
}
```

1. Assemble this code to generate x86-64 assembly language with minimal optimization (-Og). Please delete any assembler directives, other than .pos, .quad, .string, and labels. Annotate the assembly with comments. Now, using the code examples from the slides and book, rewrite this into assembly code for the Y86. For this part, you don’t have to do main, just len1. What substantive differences do you see between the Y86 and x86-64 code.

   BTW: whenever you’re asked to annotate assembler, point out what’s going on, but not trivia. “Move 1 to %rax” is not a useful comment.

2. Now assemble it again with higher optimization (-O2). Compare the two compilations. What differences do you observe when the optimization is more advanced? Again concentrate on len1.


4. (extra credit) Below is the x86 code for the recursive pcount function in slideset 9. Assume that when this function is called %rbx contains 0x200, %rdi contains 0x5 and the top of the stack contains the return address from the caller. List (in order) what will have been added to the stack when the base case is executed (i.e., at the beginning of the recursive call where %rdi is 0). You can use .L5 as an element on the stack. Assume that you’re not creating explicit stack frames with %rbp.
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq  %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   $1, %rdi
    call   pcount_r
.L5:
    addq   %rbx, %rax
    popq   %rbx
.L6:
    ret