BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED TO THE TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY.

THIS IS THE README FILE FOR LAB 5.

Name: Yifan Yao.740

When answering the questions in this file, make a point to take a look at whether the most significant bit (remembering it can be bit 7, 15, 31 or 63 depending upon what size value we are working with) to see if the results you see change based on whether it is a 0 or a 1.

```
.file "lab5.s"
.globl main
    .type    main, @function

.text
main:
    pushq %rbp        #stack housekeeping
    movq %rsp, %rbp

Label1:
#as you go through this program note the changes to %rip
    movq    $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211, %rip: 0x401193
    movb    $-1, %al                    # the value of %rax is: 0x88776655443322ff, %rip: 0x401195
    movw    $-1, %ax                    # the value of %rax is: 0x887766554433ffff, %rip: 0x401199
    movl    $-1, %eax                   # the value of %rax is: 0x00000000ffffffff, %rip: 0x40119e
    movq    $-1, %rax                   # the value of %rax is: 0xffffffffffffffff, %rip: 0x4011a5

    movl    $-1, %eax                   # the value of %rax is: 0x00000000ffffffff, %rip: 0x4011aa
    cltq                                # the value of %rax is: 0xffffffffffffffff, %rip: 0x4011ac

    movl    $0x7fffffff, %eax           # the value of %rax is: 0x000000007fffffff, %rip: 0x4011b1
    cltq                                # the value of %rax is: 0x000000007fffffff, %rip: 0x4011b3
    movl    $0x8fffffff, %eax           # the value of %rax is: 0x000000008fffffff, %rip: 0x4011b8
    cltq                                # the value of %rax is: 0xffffffff8fffffff, %rip: 0x4011ba
                                        # what do you think the cltq instruction does? When cltq executed, the value in %eax were extended to %rax.

    movq    $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211, %rip:0x4011c4

    # the value of %rdx *before* movb $0xAA, %dl executes is: 0x00007fffffffe3c8
    movb    $0xAA, %dl          # the value of %rdx is: 0x00007fffffffe3aa, %rip: 0x4011c6
    movb    %dl, %al            # the value of %rax is: 0x88776655443322aa, %rip: 0x4011c8
    movsbw    %dl, %ax          # the value of %rax is: 0x887766554433ffaa, %rip: 0x4011cc
    movzbw    %dl, %ax          # the value of %rax is: 0x8877665544330aa, %rip: 0x4011d0

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211, %rip: 0x4011da
    movb    %dl, %al            # the value of %rax is: 0x88776655443322aa, %rip: 0x4011dc
    movsbl    %dl, %eax         # the value of %rax is: 0x00000000ffffffaa, %rip: 0x4011df
    movzbl    %dl, %eax         # the value of %rax is: 0x00000000000000aa, %rip: 0x4011e2

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211, %rip: 0x4011ec
    movb    %dl, %al            # the value of %rax is: 0x88776655443322aa, %rip: 0x4011ee
    movsbq  %dl, %rax           # the value of %rax is: 0xffffffffffffffaa, %rip: 0x4011f2
    movzbq  %dl, %rax           # the value of %rax is: 0x00000000000000aa, %rip: 0x4011f6

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211, %rip: 0x401200
    # the value of %rdx *before* movb $0x0000000000000055, %dl executes is: 0xaa
    movb    $0x55, %dl          # the value of %rdx is: 0x8877665544332211, %rip: 0x401202
    movb    %dl, %al            # the value of %rax is: 0x8877665544332255, %rip: 0x401204
    movsbw  %dl, %ax            # the value of %rax is: 0x8877665544330055, %rip: 0x401208
    movzbw  %dl, %ax            # the value of %rax is: 0x8877665544330055, %rip: 0x40120c

    movq    $0x8877665544332211, %rax # the value of %rax is: 0x8877665544332211, %rip: 0x401216
    movb    %dl, %al            # the value of %rax is: 0x8877665544332255, %rip: 0x401218
    movsbl  %dl, %eax           # the value of %rax is: 0x0000000000000055, %rip: 0x40121b
    movzbl  %dl, %eax           # the value of %rax is: 0x0000000000000055, %rip: 0x40121e

    movq    $0x8877665544332211, %rax # the value of %rax is: 0x8877665544332211, %rip: 0x401228
    movb    %dl, %al            # the value of %rax is: 0x8877665544332255, %rip: 0x40122a
    movsbq  %dl, %rax           # the value of %rax is: 0x0000000000000055, %rip: 0x40122e
    movzbq  %dl, %rax           # the value of %rax is: 0x0000000000000055, %rip: 0x401232

#    movq    $0x8877665544332211, %rax
#    pushb   %al
#    movq    $0, %rax
#    popb    %al

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211, the value of %rsp is: 0x00007fffffffe2d0
    pushw   %ax                          # the value of %rsp is: 0x00007fffffffe2ce
                                         # the difference between the two values of %rsp is: 2
    movq    $0, %rax                     # the value of %rax is: 0x0000000000000000
    popw    %ax                          # the value of %rax is: 0x0000000000002211, How did the value of %rsp change? 0x00007fffffffe2d0, added by 2

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211, the value of %rsp is: 0x00007fffffffe2d0
    pushw   %ax                          # the value of %rsp is: 0x00007fffffffe2ce
                                         # the difference between the two values of %rsp is: 2
    movq    $-1, %rax                    # the value of %rax is: 0xffffffffffffffff
    popw    %ax                          # the value of %rax is: 0xffffffffffff2211, How did the value of %rsp change? 0x00007fffffffe2d0, returned to original value


#    movq    $0x8877665544332211, %rax
#    pushl   %eax
#    movq    $0, %rax
#    popl    %eax

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211, the value of %rsp is: 0x7fffffffe2d0
    pushq   %rax                         # the value of %rsp is: 0x00007fffffffe2c8
                                         # the difference between the two values of %rsp is: 8
    movq    $0, %rax                     # the value of %rax is: 0x0
    popq    %rax                         # the value of %rax is: 0x8877665544332211, How did the value of %rsp change? Changed back to 0x00007fffffffe2d0

                                         # what eflags are set? 0x246    [ PF ZF IF ]

    movq    $0x500, %rax                 # the value of %rax is: 0x0000000000000500
    movq    $0x123, %rcx                 # the value of %rcx is: 0x0000000000000123
    # 0x123 - 0x500
    subq    %rax, %rcx                   # the value of %rax is: 0x0000000000000500
            # the value of %rcx is: 0xffffffffffffc23

          # what eflags are set? 0x283    [ CF SF IF ]

    movq    $0x500, %rax                 # the value of %rax is: 0x0000000000000500
    movq    $0x123, %rcx                 # the value of %rcx is: 0x0000000000000123
    # 0x500 - 0x123
    subq    %rcx, %rax                   # the value of %rax is: 0x00000000000003dd
                  # what eflags are set? 0x216    [ PF AF IF ]

    movq    $0x500, %rax                 # the value of %rax is: 0x0000000000000500
    movq    $0x500, %rcx                 # the value of %rcx is: 0x0000000000000500
    # 0x500 - 0x500
    subq    %rcx, %rax                   # the value of %rax is: 0x0000000000000000
                # what eflags are set? 0x246    [ PF ZF IF ]

    movb    $0xff, %al      # the value of %rax is: 0x00000000000000ff
    # 0xff +=1 (1 byte)
    incb    %al             # the value of %rax is: 0x0000000000000000, what eflags are set? 0x256    [ PF AF ZF IF ]

    movb    $0xff, %al      # the value of %rax is: 0x00000000000000ff
    # 0xff +=1 (4 bytes)
    incl    %eax            # the value of %rax is: 0x0000000000000100, what eflags are set? 0x216    [ PF AF IF ]

    movq    $-1, %rax           # the value of %rax is: 0xffffffffffffffff
    # 0xff +=1 (8 bytes)
    incq    %rax            # the value of %rax is: 0x0000000000000000, what eflags are set? 0x256    [ PF AF ZF IF ]

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211
    movq    $0x8877665544332211, %rcx    # the value of %rax is: 0x8877665544332211, what eflags are set? 0x256    [ PF AF ZF IF ]
    addq    %rcx, %rax                   # the value of %rax is: 0x10eeccaa88664422, what eflags are set? 0xa07    [ CF PF IF OF ]

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211
    andq    $0x1, %rax                   # the value of %rax is: 0x0000000000000001

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211, explain why the values for AND/OR/XOR are
    andq    %rax, %rax                   # the value of %rax is: 0x8877665544332211, what they are
    orq     %rax, %rax                     # the value of %rax is: 0x8877665544332211
    xorq    %rax, %rax                   # the value of %rax is: 0x0000000000000000

    movq    $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211
    andw    $0x3300, %ax                 # the value of %rax is: 0x8877665544332200, explain the value in the 8 byte register vs
                                         # the value in the 2 byte register

    salq    $4, %rax                        # the value of %rax is: 0x8776655443322000, Why? We moved 4 bits to left and the missing bits were filled by 0 at last.

    movq    $0xff0000001f000000, %rax   # the value of %rax is: 0xff0000001f000000, what do these 6 values look like in binary???      1111 1111 0000 0000 0000 0000 0000 0000 0001 1111 0000 0000 0000 0000 0000 0000
    sall    $1, %eax                    # the value of %rax is: 0x000000003e000000  do these shift instructions do what you expected?  0000 0000 0000 0000 0000 0000 0000 0000 0001 1111 0000 0000 0000 0000 0000 0000
    sall    $1, %eax                    # the value of %rax is: 0x000000007c000000  The binary value did the left shift as expected.   0000 0000 0000 0000 0000 0000 0000 0000 0011 1110 0000 0000 0000 0000 0000 0000
    sall    $1, %eax                    # the value of %rax is: 0x00000000f8000000  Left shift starts from %eax.                       0000 0000 0000 0000 0000 0000 0000 0000 0111 1100 0000 0000 0000 0000 0000 0000
    sall    $1, %eax                    # the value of %rax is: 0x00000000f0000000                                                     0000 0000 0000 0000 0000 0000 0000 0000 1111 1000 0000 0000 0000 0000 0000 0000
    sall    $1, %eax                    # the value of %rax is: 0x00000000e0000000                                                     0000 0000 0000 0000 0000 0000 0000 0000 1110 0000 0000 0000 0000 0000 0000 0000

    movq    $0xff000000ff000000, %rax   # the value of %rax is: 0xff000000ff000000, what do these 6 values look like in binary???      1111 1111 0000 0000 0000 0000 0000 0000 1111 1111 0000 0000 0000 0000 0000 0000
    salq    $1, %rax                    # the value of %rax is: 0xfe000001fe000000  do these shift instructions do what you expected?  1111 1110 0000 0000 0000 0000 0000 0001 1111 1110 0000 0000 0000 0000 0000 0000
    salq    $1, %rax                    # the value of %rax is: 0xfc000003fc000000  The binary value did the left shift as expected.   1111 1100 0000 0000 0000 0000 0000 0011 1111 1100 0000 0000 0000 0000 0000 0000
    salq    $1, %rax                    # the value of %rax is: 0xf8000007f8000000  Left shift starts from %rax.                       1111 1000 0000 0000 0000 0000 0000 0111 1111 1000 0000 0000 0000 0000 0000 0000
    salq    $1, %rax                    # the value of %rax is: 0xf000000ff0000000                                                     1111 0000 0000 0000 0000 0000 0000 1111 1111 0000 0000 0000 0000 0000 0000 0000
    salq    $1, %rax                    # the value of %rax is: 0xe000001fe0000000                                                     1110 0000 0000 0000 0000 0000 0001 1111 1110 0000 0000 0000 0000 0000 0000 0000

    movq    $0xff0000000000ff, %rax   # the value of %rax is: 0xff0000000000ff, what do these 6 values look like in binary???      1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111
```

```
158     sarq    $1, %rax                    # the value of %rax is: 0xff8000000000007f  do these shift instructions do what you expected?  1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111 1111
159     sarq    $1, %rax                    # the value of %rax is: 0xffc000000000003f  The binary value did the left shift as expected.   1111 1111 1100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1111
160     sarq    $1, %rax                    # the value of %rax is: 0xffe000000000001f  Arithmetic right shift starts from %rax, missing   1111 1111 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1111
161     sarq    $1, %rax                    # the value of %rax is: 0xfff000000000000f  bits were filled by sign at left.                 1111 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111
162     sarq    $1, %rax                    # the value of %rax is: 0xfff8000000000007                                                  1111 1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111
163
164     movq    $0xff000000000000ff, %rax   # the value of %rax is: 0xff000000000000ff, what do these 6 values look like in binary???      1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111
165     shrq    $1, %rax                    # the value of %rax is: 0x7f8000000000007f  do these shift instructions do what you expected?  0111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111 1111
166     shrq    $1, %rax                    # the value of %rax is: 0x3fc000000000003f  The binary value did the left shift as expected.   0011 1111 1100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1111
167     shrq    $1, %rax                    # the value of %rax is: 0x1fe000000000001f  Logic right shift starts from %rax, missing bits   0001 1111 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1111
168     shrq    $1, %rax                    # the value of %rax is: 0x0ff000000000000f  were filled by 0 at left.                         0000 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111
169     shrq    $1, %rax                    # the value of %rax is: 0x07f8000000000007                                                  0000 0111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111
170
171     movq    $0xff000000000000ff, %rax   # the value of %rax is: 0xff000000000000ff, what do these 6 values look like in binary???      1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111
172     sarw    $1, %ax                     # the value of %rax is: 0xff0000000000007f  do these shift instructions do what you expected?  1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111 1111
173     sarw    $1, %ax                     # the value of %rax is: 0xff0000000000003f  The binary value did the left shift as expected.   1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1111
174     sarw    $1, %ax                     # the value of %rax is: 0xff0000000000001f  Arithmetic right shift starts from %ax, missing    1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1111
175     sarw    $1, %ax                     # the value of %rax is: 0xff0000000000000f  bits were filled by sign at left.                 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111
176     sarw    $1, %ax                     # the value of %rax is: 0xff0000000000007                                                   1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111
177
178     movq    $0xff000000000000ff, %rax   # the value of %rax is: 0xff000000000000ff, what do these 6 values look like in binary???      1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111
179     shrw    $1, %ax                     # the value of %rax is: 0xff0000000000007f, do these shift instructions do what you expected?  1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111 1111
180     shrw    $1, %ax                     # the value of %rax is: 0xff0000000000003f  The binary value did the left shift as expected.   1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1111
181     shrw    $1, %ax                     # the value of %rax is: 0xff0000000000001f  Logic right shift starts from %ax, missing bits   1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1111
182     shrw    $1, %ax                     # the value of %rax is: 0xff0000000000000f  were filled by 0 at left.                         1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111
183     shrw    $1, %ax                     # the value of %rax is: 0xff0000000000007                                                   1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111
184
185
186     leave                       #post function stack cleanup
187     ret
188
189 .size main, .-main
```

1. Write a paragraph that describes what you observed happen to the value in register **%rax** as you watched **mov**X (where X is 'q', 'l', 'w', and 'b') instructions executed. Describe what data changes occur (and, perhaps, what data changes you expected to occur that didn't). Make a point to address what happens when moving less than 8 bytes of data to a register.

   When the movX executed, the data from `%rax` will also change, the difference between 'q', 'l', 'w', and 'b' is the changes of last 8, 4, 2, 1 byte. When `movb $-1, %al` and `movw $-1, %ax` executed, the last 1 and 2 bytes of the value were changed. However, when `movl $-1, %eax` and `movq $-1, %rax` executed, the value of `%rax` are exactly as the immediate. From the observation, when we are moving less than 8 bytes of data to a register, the last 1 byte and 2 bytes will be replaced and other bytes will remain same, and when we are moving 4 bytes and 8 bytes, the value will be replaced as the exact immediate value.

2. What did you observe happens when the **cltq** instruction is executed? Did it matter what value is in **%eax**? Does **cltq** have any operands?

   When `cltq` executed, the value in `%eax` were extended to `%rax`. The value in `%eax` matter, when the value is 0x7fffffff, the 32 bits binary representation is (0)111 1111 1111 1111 1111 1111 1111 1111, therefore, the 64 bits signed extension will be (0000 0000 0000 0000 0000 0000 0000 0000 0)111 1111 1111 1111 1111 1111 1111 1111. Compare to 0x8fffffff, the 32 bits binary representation is 1000 1111 1111 1111 1111 1111 1111 1111, the 64 bits signed extension will be 1111 1111 1111 1111 1111 1111 1111 1111 1000 1111 1111 1111 1111 1111 1111 1111.

3. Write a paragraph that describes what you saw with respect to what happens as you use the **movs**XX and **movz**XX instructions with different sizes of registers. What do you observe with respect to the source and destination registers used in each instruction? Is there a relationship between them and the XX values? Describe what data changes occur (and, perhaps, what data changes you expected to occur that didn't).

   The size of the source and the destination register are different (source > destination). The difference between `movsXX` and `movzXX` is that `mov`**`s`**`XX` will provide a **s**ign-extended value for extra bytes, and `mov`**`z`**`XX` will have **z**ero-extended value for extra bytes in destination register.

   As we know, `b` = 1 byte, `w` = 2 bytes, `l` = 4 bytes, `q` = 8 bytes. Therefore, we can find a relationship between them and the XX values, each X represents the size of the data and by observe line 32, 37, 42, 49, 54, 59 (all **s**ign-extended instructions). For instance, in line 37 (`mov`**`s`**`bl %dl, %eax`), `%dl` hold 1 byte and `%eax` hold 4 bytes. To confirm my statement, let's use line 42 as another example, the source is `%dl` (1 byte = **`b`**) and the destination is `%rax` (8 bytes = **`q`**) which means the instruction should be `mov`**`sbq`**. By compare to original line 42 (`movsbq %dl, %rax`), I confirmed my guess.

   However, it is not always the case, from line 49 and 50, line 54 and 55, line 59 and 60, problem occurs (the thing value were expected did not occur). The value of `%rax` become same when `%dl` changed to 0x55[0b 0101 0101] (from 0xAA [0b 1010 1010]). When the leading digit is 0, the sign is 0, and 0 will be extended in both sign-extended and zero-extended value.

4. Write a paragraph that describes what you observed as you watched different push/pop instructions execute. What values were actually put on the stack? How did the value in %rsp change? Use the command **help x** from the command line in gdb. This will give you the format of the **x** instruction that allows you to see what is in specific addresses in memory. Note that a **word** means 2 bytes in x86-64, but it means 4 bytes when using the **x** command in gdb. To print 2 byte values with x, you must specify **h** for

halfword. If you wish to use an address located in a register as an address to print from using **x**, use **$** rather than **%** to designate the register. For example, if you wanted to print, in hexadecimal format, 1 2-byte value that is located in memory starting at the address located in register **rsp**, then you could use **x/1xh $rsp**. If you wanted to print, in hexadecimal format, 1 8-byte value that is located in memory starting at the address located in register **rsp**, then you could use **x/1xg $rsp**. You might want to play with this command a little. ☺

As we know, `b` = 1 byte, `w` = 2 bytes, `l` = 4 bytes, `q` = 8 bytes. Similar case here, when we execute `pushX/popX`, the `X` is corresponded with the value of the `%rsp`. When we executed line 68 (`pushw %ax`), the address stored the value of `%ax` was pushed into the stack, and the value of `%rsp` were decreased by 2; Oppositely, when we executed line 71 (`popw %ax`), the value of `%rsp` were increased by 2. However, the line 85 to 89 do the same thing here, except they are manipulating 8 bytes value (0x8877665544332211). Since the instruction `pushX/popX` is corresponding the register by size, therefore, the value in the register will be `push/pop` to the stack correctly.

5. What did you observe happened to the condition code values as instructions that process within the ALU executed? What instructions caused changes? Were the changes what you expected? Why or why not?

   Smaller Value – Larger Value = Negative Value, CF (Carry Flag) and SF (Sign Flag) were turned on. This change was expected since a smaller number minus a larger number must be negative and there must be a carry out.
   X – X = 0, ZF (Zero Flag) was turned on. This change was expected since both numbers were same.
   0xff += 1 (1 byte), the value of 1 byte register becomes to 0x00, ZF was turned on, OF (Overflow Flag) did not turned on. This change was expected because 0xff represents -1 in 1 byte and when we increment 1 in 1 byte, it will become 0.
   0xff += 1 (4 bytes), the value of 4 bytes register becomes to 0x00000100, there was no overflow occur, and no carry out. This change was expected since we are adding 4 bytes, which gives a space to store the result correctly.
   0xffffffffffffffff += 1 (8 bytes), the value of 8 bytes register becomes to 0x0000000000000000, there was no overflow occur, and no carry out. This change was expected since the original value was representing -1 in 8 bytes and when we increment by 1 in 8 bytes, it becomes to 0.
   When 0x8877665544332211 + 0x8877665544332211, CF and OF were turned on. This change was expected since there the largest representation is 8 bytes and there is no space to store the carryout which caused the overflow.

6. There were some instructions that caused bitwise AND/OR/XOR data manipulation. What did you observe?

   When the source and the destination do not have same bytes, the smaller value will zero-extended to the same size of the larger value and make the bitwise operation. If both source and destination have the same bytes, bitwise operation will be operated directly.

7. There were some instructions that executed left or right bit shifting. What did you observe with respect to the register data? Did the size of the data being shifted change the result in the register? How?

   Similar as pervious instructions, bit shifting instructions were corresponded with the suffix. If we are shifting with a 2 bytes value in the register, the instruction will be '`w`', same as 4 bytes ('`l`') and 8 bytes ('`q`'). The size of the data being shifted change the result in the register, compare line 164 to 169 (first group) and line 178 to 183 (second group), the first group of instructions were shifted from the beginning

of the 8 bytes register (`%rax`) and the second group of instructions were shifted from the beginning of the last 2 bytes register (`%ax`).

8. What did you observe happening to the value in register **%rip** over the course the program? Did it always change by the same amount as each instruction executed?

   The value of `%rip` will be increased(number(`X`)) by 2(‘`b`’), 4(‘`w`’), 5(‘`l`’), 7(‘`q`’) when `movX` executed, and the value of `%rip` will be increased by 2 when `cltq` executed. And there is an exception, for instance line 27 (`movq $0x8877665544332211, %rax`) the value of `%rip` was increased by 10. The popq instruction will increase the `%rip` by 1, `incb/incl/sall` by 2, `subq/addq/orq/salq/sarq/shrq/sarw/shrw` by 3, `andq/andw/xorq` by 4.

9. What did you observe when you took the comments away from the two different instruction sets and tried to reassemble the program? There were questions in item **L** and **M** in the Lab 5 Description; include your answers to those questions here.

   After I took the comments away from the two different instruction sets and tried to reassemble the program as the direction asked, as result, the compiler promotes error message in both step M and the error message consistent in step N.

   With suffixes ‘b’ and ‘l’, it is not valid option for 64 bits processors.

```
lab5-1.s: Assembler messages:
lab5-1.s:63: Error: invalid instruction suffix for `push'
lab5-1.s:65: Error: invalid instruction suffix for `pop'

lab5-1.s: Assembler messages:
lab5-1.s:81: Error: invalid instruction suffix for `push'
lab5-1.s:83: Error: invalid instruction suffix for `pop'
```

10. Any other comments about what you observed?

    There are so many ways to move data.