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: Vishal Kumar

When answering the questions in this file, make a point to take a look at whether the most significant bit (remembering it can be in bit position 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


# Recall that -1 is represented as 0xff, 0xffff, etc. depending upon the size of the value

movb                    $-1, %al                  # the value of %rax is: 0x88776655443322ff

movw                    $-1, %ax                  # the value of %rax is: 0x887766554433ffff

movl                    $-1, %eax                 # the value of %rax is: 0x88776655ffffffff

movq                    $-1, %rax                 # the value of %rax is: 0xffffffffffffffff
```

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


movl    $0x7fffffff, %eax           # the value of %rax is:  0x000000007fffffff
cltq                                # the value of %rax is: 0x000000007fffffff
movl    $0x8fffffff, %eax           # the value of %rax is: 0x000000008fffffff
cltq                                # the value of %rax is: 0xffffffff8fffffff
                                    # What is the difference between the values 0x7fffffff and 0x8fffffff
`                                   0x8fffffff must be negative
                                    # what do you think the cltq instruction does?
                                    Adds leading zeros or fs to fill register
movq    $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211
                                    # the value of %rdx *before* movb $0xAA, %dl executes is:
0x00007fffffffdaf8


        # Note the value of the 8-byte register values vs the 1, 2, or 4-byte register values
        # How does each size instruction suffix affect the 8-byte register?  Don't write answers here; you'll
need this info later.
movb    $0xAA, %dl                  # the value of %rdx is: 0x00007fffffffdaaa
movb    %dl, %al                    # the value of %rax is: 0x88776655443322aa
movsbw  %dl, %ax                    # the value of %rax is: 0x88776655443322aa
movzbw  %dl, %ax                    # the value of %rax is: 0x88776655443300aa


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


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

```
movb      %dl, %al                    # the value of %rax is: 0x88776655443322aa

movsbq    %dl, %rax                   # the value of %rax is: 0xffffffffffffffaa

movzbq    %dl, %rax                   # the value of %rax is: 0x00000000000000aa


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

                                      # the value of %rdx *before* movb $0x55, %dl executes is:
0x00007fffffffdaaa

movb      $0x55, %dl                  # the value of %rdx is: 0x00007fffffffda55

movb      %dl, %al                    # the value of %rax is: 0x8877665544332255

movsbw    %dl, %ax                    # the value of %rax is: 0x8877665544330055

movzbw    %dl, %ax                    # the value of %rax is: 0x8877665544330055


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

movb      %dl, %al                    # the value of %rax is: 0x8877665544332255

movsbl    %dl, %eax                   # the value of %rax is: 0x0000000000000055

movzbl    %dl, %eax                   # the value of %rax is: 0x0000000000000055


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

movb      %dl, %al                    # the value of %rax is: 0x8877665544332255

movsbq    %dl, %rax                   # the value of %rax is: 0x0000000000000055

movzbq    %dl, %rax                   # the value of %rax is: 0x0000000000000055


#movq     $0x8877665544332211, %rax

#pushb    %al

#movq     $0, %rax

# popb    %al


movq      $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211          the value
of %rsp is: 0x00007fffffffda00

pushw     %ax                         # the value of %rsp is: 0x00007fffffffd9fe
```

```
                                        # the difference between the two values of %rsp is:

movq        $0, %rax                    # the value of %rax is: 0x0000000000000000

popw        %ax                         # the value of %rax is: 0x0000000000002211        How did
the value of %rsp change? 0x7fffffffda00


movq        $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211        the value
of %rsp is: 0x00007fffffffda00

pushw       %ax                         # the value of %rsp is: 0x00007fffffffd9fe

                                        # the difference between the two values of %rsp is:

movq        $-1, %rax                   # the value of %rax is: 0xffffffffffffffff

popw        %ax                         # the value of %rax is: 0xffffffffffff2211    How did the value of %rsp
change? 0x00007fffffffda00



#movq       $0x8877665544332211, %rax

#pushl      %eax

#movq       $0, %rax

#popl       %eax


movq        $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211        the value
of %rsp is: 0x00007fffffffda00

pushq       %rax                        # the value of %rsp is: 0x00007fffffffd9f8

                                        # the difference between the two values of %rsp is:

movq        $0, %rax                    # the value of %rax is: 0x0000000000000000

popq        %rax                        # the value of %rax is:  0x8877665544332211        How did
the value of %rsp change? 0x7fffffffda00


                                        # what rflags are set? 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: 0xfffffffffffffc23

                              # what rflags are set? 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 rflags are set? 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 rflags are set? 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
rflags are set? P A Z I

movb        $0xff, %al        # the value of %rax is: 0x00000000000000ff
# 0xff +=1 (4 bytes)
incl        %eax              # the value of %rax is: 0x0000000000000100        what
rflags are set? 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
rflags are set? P A Z I

movq      $0x8877665544332211, %rax    # the value of %rax is: 0x8877665544332211
movq      $0x8877665544332211, %rcx    # the value of %rax is: 0x8877665544332211        what
rflags are set? P A Z I
addq      %rcx, %rax                   # the value of %rax is: 0x10eeccaa88664422        what
rflags are set? C P I 0

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?

movq      $0xff0000001f000000, %rax    # the value of %rax is: 0xff0000001f000000
                                       # to help you understand what's happening in this part of the code,
write the value in %rax in binary
                                       # on a piece of scratch paper for the remaining instructions in this file
                                       # and watch the bits move as each shift instruction occurs.
                                       # You should notice how each of the 1-, 2-, 4-, and 8-byte shift
instructions works
```

```
                                            # within the 8-byte register.
sall     $1, %eax                    # the value of %rax is: 0x3e000000          do these shift
instructions do what you expected?
sall     $1, %eax                    # the value of %rax is: 0x000000007c000000
sall     $1, %eax                    # the value of %rax is: 0x00000000f8000000
sall     $1, %eax                    # the value of %rax is: 0x00000000f0000000
sall     $1, %eax                    # the value of %rax is: 0x00000000e0000000

movq     $0xff000000ff000000, %rax   # the value of %rax is: 0xff000000ff000000
salq     $1, %rax                    # the value of %rax is: 0xfe000001fe000000
salq     $1, %rax                    # the value of %rax is: 0xfc000003fc000000
salq     $1, %rax                    # the value of %rax is: 0xf8000007f8000000
salq     $1, %rax                    # the value of %rax is: 0xf000000ff0000000
salq     $1, %rax                    # the value of %rax is: 0xe000001fe0000000

movq     $0xff000000000000ff, %rax   # the value of %rax is: 0xff000000000000ff
sarq     $1, %rax                    # the value of %rax is: 0xff8000000000007f
sarq     $1, %rax                    # the value of %rax is: 0xffc000000000003f
sarq     $1, %rax                    # the value of %rax is: 0xffe000000000001f
sarq     $1, %rax                    # the value of %rax is: 0xfff000000000000f
sarq     $1, %rax                    # the value of %rax is: 0xfff8000000000007

movq     $0xff000000000000ff, %rax   # the value of %rax is: 0xff000000000000ff
shrq     $1, %rax                    # the value of %rax is: 0x7f8000000000007f
shrq     $1, %rax                    # the value of %rax is: 0x3fc000000000003f
shrq     $1, %rax                    # the value of %rax is: 0x1fe000000000001f
shrq     $1, %rax                    # the value of %rax is: 0xff000000000000f
shrq     $1, %rax                    # the value of %rax is: 0x7f8000000000007
```

```
movq    $0xff000000000000ff, %rax   # the value of %rax is: 0xff000000000000ff
sarw    $1, %ax                     # the value of %rax is: 0xff0000000000007f
sarw    $1, %ax                     # the value of %rax is: 0xff0000000000003f
sarw    $1, %ax                     # the value of %rax is: 0xff0000000000001f
sarw    $1, %ax                     # the value of %rax is: 0xff0000000000000f
sarw    $1, %ax                     # the value of %rax is: 0xff00000000000007

movq    $0xff000000000000ff, %rax   # the value of %rax is: 0xff000000000000ff
shrw    $1, %ax                     # the value of %rax is: 0xff0000000000007f
shrw    $1, %ax                     # the value of %rax is: 0xff0000000000003f
shrw    $1, %ax                     # the value of %rax is: 0xff0000000000001f
shrw    $1, %ax                     # the value of %rax is: 0xff0000000000000f
shrw    $1, %ax                     # the value of %rax is: 0xff00000000000007

leave                               #post function stack cleanup
ret

.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.

movX moves a value from one destination to another. The X denotes the size of data in which we want to move, because we used movX on eight byte registers, if the size was less than a quad word (8 bytes), then the bigger data would still remain while the smaller amount of bytes denoted by the X would change.

2. What did you observe happens when the **cltq** instruction is executed? Did it matter what value is in **%eax**? What is the difference between 0x7fffffff and 0x8fffffff ? Does **cltq** have any operands?

The cltq instruction extended the sign of the data to fill the register, essentially converting a long to a quad word. 0x8fffffff must be negative because cltq extended it to 0xffffffff8fffffff. Cltq did not seem to have any operands but only operated on eax and rax, perhaps those are the default operands for the function.

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 is the difference between the value 0xAA and the value 0x55? 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).

I observed that the movsXX and movzXX moved values from one source to the other. movs was used on different sized registers like ax and eax and when the value was smaller than the register size it extended the sign of the value (0 or f). Movz, in contrast, did not extend the sign instead it extended the value with zeros in front. The difference between 0xAA and 0x55 is that 0xAA must be negative as the movsXX instructions filled the register with leading fs. I obsered that sometimes the source was smaller than the destination registers. The size of the destination register size corresponded with s or w or l or q as the last character in the the movsbX and movzbX instructions. An interesting data change that i saw occurred was even when we exectued movsbl on eax and the value was negative, it extended the sign of the value until it filled eax but also removed the leading data or rax and set it to 0 instead of leaving it there or replacing the data to Fs to extend the sign.

4. Write a paragraph that describes what you observed as you watched different push/pop instructions execute. What values are put on the stack based on the suffix used? (Use the instructions further down in this question to see stack values.) 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. J It will be well worth your time to do so as the semester continues.

We pushed and popped different sized values from the registers which was designated with pushX and popX where X was b, w, l, or q. The values put on the stack were the values in registers given in the pushX argument and popX removed them from the stack and stored them in the designation argument. rsp decreased when we pushed something onto the stack and increased when we popped something from the stack. The increase in value corresponds to the size of the data that was designated (b, w, l, q). For example when we performed popq rsp increased by 8. And when we performed pushq on it rsp decreased by 8.

5. What did you observe happen to the condition code values as instructions that process within the ALU executed? What instructions caused changes? What instructions within this program did not cause condition codes to change? When changes occurred, were the changes what you expected? Why or why not?

The condition codes changed whenever we executed different kinds of arithmetic and popX instructions. Instructions that were different than the previous set off different rflags in the ALU. I notified that the rflags did not change when we did the same instruction twice with the same arguments. I noticed that a ZF condition code was added when we performed incb on %al. This made sense to me as the ALU was handling zeros within the register.

6. There were some instructions that performed bitwise AND/OR/XOR data manipulation. What did you observe as the suffix changed? Is it consistent with respect to what you learned about these bitwise instructions in class?

As the suffix changed with the bitwised and or and xor instructions the register in which we performed the operation changed accordingly to the suffix size. The performance of these operations was consistent with what we learned in class as it followed the logic and outcome of the bitwise logical operations.

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? Is it consistent with respect to what you learned about these bitwise instructions in class?

The left and right bit-shifting performed arithmetic and logical right and left shifting on the values within the registers. The size of the data being shifted did affect the result in the register as when we performed it on the long size within eax, the “bit bucket” behavior was present and the data did not shift into the larger rax register. The shifting behavior is consistent with what we learned in class.

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?

rip was incremented over the course of the program as it pointed to each next instruction. The incrementation was not consistent, instead, it was incremented by the byte size of the instruction.

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 **M** and **N** in the Lab 5 Description; include your answers to those questions here. Based upon your experiences with this exercise, what can you conclude with respect to push/pop instructions when used with the q, l, w, and b suffixes?

When taking the # away from both sets of instructions, I received the error messages:
lab5.s:67: Error: invalid instruction suffix for `push'
lab5.s:69: Error: invalid instruction suffix for `pop'. And
lab5.s:85: Error: invalid instruction suffix for `push'
lab5.s:87: Error: invalid instruction suffix for `pop'

These error messages are consistent and This likely means that pushX and popX can only have suffixes w and q so we can only push/pop 4 bytes and 8 bytes size values on and off the stack. I believe this is likey for maintaining size properties of the stack as long and byte sizes could result in tricky math as they are not 4 or 8 bytes in size.

10. Any other comments about what you observed?

Overall a very informative lab! The stack pointer incrementing and decrementing with pop and push felt a little counterintuitive but that was because I was conceptualizing the stack backwards.