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: Yoyi Liao

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.

It's best that you present all register values in hexadecimal rather than decimal. You will be able to understand what is happening more easily.

Monitor the RFLAGS. What instructions change RFLAGS, what instructions don't? At the end of the program could you make a list of which do, and which don't? Could you describe characteristics of what instructions do that don't change condition flags vs what characteristics instructions have that don't?

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

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

movl                    $0x7fffffff, %eax           # the value of %rax is: 0x7fffffff
cltq                                                # the value of %rax is: 0x7fffffff
movl                    $0x8fffffff, %eax           # the value of %rax is: 0x8fffffff
cltq                                                # the value of %rax is: 0xffffffff8fffffff
                                                    # What is the difference between the values 0x7fffffff and 0x8fffffff
                                                    # what do you think the cltq instruction does?

movq                    $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211
                                                    # the value of %rdx *before* movb $0xAA, %dl executes is: 0x7ffffffdad8

                        # 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: 0x7ffffffdaaa
movb                    %dl, %al                    # the value of %rax is: 0x88776655443322aa
movsbw                  %dl, %ax                    # the value of %rax is: 0x887766554433ffaa
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: 0xffffffaa
movzbl                  %dl, %eax                       # the value of %rax is: 0xaa

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: 0xaa

movq                    $0x8877665544332211, %rax   # the value of %rax is: 0x8877665544332211
                                                    # the value of %rdx *before* movb $0x55, %dl executes is: 0x7ffffffdaaa
```

```
movb    $0x55, %dl              # the value of %rdx is: 0x7fffffffda55
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: 0x55
movzbl  %dl, %eax                       # the value of %rax is: 0x55

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: 0x55
movzbq  %dl, %rax                       # the value of %rax is: 0x55

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

movq    $0x8877665544332211, %rax # the value of %rax is: 0x8877665544332211    the value of %rsp is: 0x7fffffffd9e0
pushw   %ax                     # the value of %rsp is: 0x7fffffffd9de
                                # the difference between the two values of %rsp is: 0x00000000001
movq    $0, %rax                # the value of %rax is: 0x0
popw    %ax                     # the value of %rax is: 0x2211   How did the value of %rsp change? Changed to 0x7fffffffd9e0

movq    $0x8877665544332211, %rax # the value of %rax is: 0x8877665544332211    the value of %rsp is: 0x7fffffffd9e0
pushw   %ax                     # the value of %rsp is: 0x7fffffffd9de
                                # the difference between the two values of %rsp is: 0x00000000001
movq    $-1, %rax               # the value of %rax is: 0xffffffffffffffff
popw    %ax                     # the value of %rax is: 0xffffffffffff2211    How did the value of %rsp change? Changed to
0x7fffffffd9e0


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

movq    $0x8877665544332211, %rax # the value of %rax is: 0x8877665544332211    the value of %rsp is: 0x7fffffffd9e0
pushq   %rax                    # the value of %rsp is: 0x7fffffffd9d8
                                # the difference between the two values of %rsp is: 0x000000000008
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
to 0x7fffffffd9e0

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

movq    $0x500, %rax            # the value of %rax is: 0x500
movq    $0x123, %rcx            # the value of %rcx is: 0x123
# 0x123 - 0x500
subq    %rax, %rcx              # the value of %rax is: 0x500
                                # the value of %rcx is: 0xfffffffffffffc23

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

movq    $0x500, %rax            # the value of %rax is: 0x500
movq    $0x123, %rcx            # the value of %rcx is: 0x123
# 0x500 - 0x123
subq    %rcx, %rax              # the value of %rax is: 0x3dd
                                # what rflags are set? 0x216 [ PF AF IF ]

movq    $0x500, %rax            # the value of %rax is: 0x500
movq    $0x500, %rcx            # the value of %rcx is: 0x500
# 0x500 - 0x500
subq    %rcx, %rax              # the value of %rax is: 0x0
                                # what rflags are set? 0x246 [ PF ZF IF ]

movb    $0xff, %al                      # the value of %rax is: 0xff
# 0xff +=1 (1 byte)
incb    %al                     # the value of %rax is: 0x0       what rflags are set? 0x256 [ PF AF ZF IF ]
```

movb **$0xff, %al** **# the value of %rax is:** 0xff
 # 0xff +=1 (4 bytes)
incl **%eax** **# the value of %rax is:** 0x100 **what rflags 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:** 0x0 **what rflags 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 rflags are set?** 0x256 [PF AF ZF IF]
addq **%rcx, %rax** **# the value of %rax is:** 0x10eeccaa88664422 **what rflags are set?** 0xa07 [CF PF IF OF]

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

movq **$0x8877665544332211, %rax** **# the value of %rax is:** 0x8877665544332211 **explain why the values for AND/OR/XOR are what they are:** A number bitwise AND with 1 is 1
andq **%rax, %rax** **# the value of %rax is:** 0x887766554432211
orq **%rax, %rax** **# the value of %rax is:** 0x8877665544332211
xorq **%rax, %rax** **# the value of %rax is:** 0x0

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:** bitwise AND only applies to last 2 byte (0x2211 for ax). 2211 AND 3300 is 2200. Hence, the value of last 2 bytes in ax changes from 2211 to 2200

salq **$4, %rax** **# the value of %rax is:** 0x8776655443322000 **Why?:** The salq $4, %rax operation shifts all the bits of %rax 4 bit positions to the left, or 1 digit in hex.

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:** 0x7c000000 **when bits shift left, and pass between bytes, do the bits**
sall **$1, %eax** **# the value of %rax is:** 0xf8000000 **end up where you expected?**
sall **$1, %eax** **# the value of %rax is:** 0xf0000000
sall **$1, %eax** **# the value of %rax is:** 0xe0000000

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:** 0xff000000000007f
sarw **$1, %ax** **# the value of %rax is:** 0xff000000000003f
sarw **$1, %ax** **# the value of %rax is:** 0xff000000000001f
sarw **$1, %ax** **# the value of %rax is:** 0xff000000000000f
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 movX (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.**

   A: For movq, it move all of the bits from source to location. For movb, it only moves the last byte from source to destination while other bytes of destination remain unchanged. Movl will zero out the top 4 bytes. Movw is same as movb except for 2 bytes of destination.

2. **What did you observe happens when the cltq instruction is executed? Did it matter what value is in %eax? What is the difference in the result of the cltq instruction (in %rax) when %eax holds 0x7fffffff vs when it holds 0x8fffffff ? Does cltq have any operands?**

   A: it just sign extend eax to rax. Since 7 has 0 as its most significant bit, when it sign extends, it ignores leading zeros. However, 8 has 1 as its most significant bit, so it'll be pad with 1s when sign extend. Cltq doesn't have any operands.

3. **Write a paragraph that describes what you saw with respect to what happens as you use the movsXX and movzXX instructions with different sizes of registers. What is the difference between the value 0xAA and the value 0x55? (note the most significant bit of 0xAA vs 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).**

   A: movsXX and movzXX both zeros out top 4 registers if the last "X" is l. If the last X is l, it extends the source to 4 bytes. If the last X is q, it extends the source to 8 bytes. They'll have the same result if working with unsinged, but movz does not do extension (always pad with 0).

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. ☺ It will be well worth your time to do so as the semester continues.**

A: if no suffix in front of the register, we're just pushing the last byte into the stack. If we're working with eax, we're pushing the bottom 4 bytes into the stack. If rax, we're pushing the entire things into the stack. After push, the value will now be pushed into $rsp.

For pop, we're popping what's in the stack to the described register. If no suffix in front, we're just writing to the last byte. If we're working with eax, we're popping to the bottom 4 bytes from the stack. If rax, we're popping the entire things from the stack. After popping, that value will be popped out (removed) from stack (rsp).

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?**

   Different flags were raised based on different condition values. Subq, incb, incl, incq, addq are instructions that caused changes. Movq and pop doesn't caused changes to the flag.

   Z flag is set when the value in register is zero. O flag is when the value overflow. S flag is when the value is signed. These were the changes I expected. For example, when I set the ax value to 0, it did indeed raise the O flag.

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?**

   A: the suffix indicates how many bytes are we performing the bitwise manipulation on. For example, when doing andq, all bytes are being AND with the value. However, when doing andw, only the last 2 bytes are performing the manipulation and other bytes remained unchanged.

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?**

   A: The register data changes as bits are shifted left or right. The size of the register remains the same, but the value changes based on the number of bits shifted. Left shifts multiply the value, while right shifts divide the value. These behaviors align with the standard understanding of bitwise shift operations.

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?**

   A: it changes by the same amount for same instruction, but the amount of change is different between the instructions.

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?**

   A: when I took off the comments for section 1, I can't even compile it due to invalid instruction suffix for

both push and pop. Same error shows up when I took off comments for section 2. Hence, I concluded that only q and w are valid suffix for push and pop instructions.

10. Any other comments about what you observed?

A: I think it's interesting to see how the value of $rip changed. After executing the instruction of movq, for example, I expect it to add 8 to the current value since q is 8 bytes. However, it's actually adding 10 to it instead.