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.

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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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).

1. 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.

1. 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'

1. Any other comments about what you observed?

There are so many ways to move data.