CS 210 A1Part2

Jeong Yong Yang

TOTAL POINTS

55 / 61

QUESTION 1

Question 1 19 pts

1.1 a **2 / 2**

✓ **- 0 pts** Correct

1.2 b **5 / 5**

✓ **- 0 pts** Correct

1.3 c **0 / 5**

✓ **- 5 pts** **Answer is not clear / missing.**

The question is "Which flags get newly set to 1?", you have not specified which flags get changed newly to 1.

1.4 d **2 / 2**

✓ **- 0 pts** Correct

1.5 e **5 / 5**

✓ **- 0 pts** Correct

QUESTION 2

Question 2 14 pts

2.1 a **2 / 2**

✓ **- 0 pts** Correct

2.2 b **2 / 2**

✓ **- 0 pts** Correct

2.3 c **2 / 2**

✓ **- 0 pts** Correct

2.4 d **2 / 2**

✓ **- 0 pts** Correct

2.5 e **2 / 2**

✓ **- 0 pts** Correct

2.6 f **2 / 2**

✓ **- 0 pts** Correct

2.7 g **2 / 2**

✓ **- 0 pts** Correct

QUESTION 3

Question 3 28 pts

3.1 a **4 / 4**

✓ **- 0 pts** Correct

3.2 b **4 / 4**

✓ **- 0 pts** Correct

3.3 c **4 / 4**

✓ **- 0 pts** Correct

3.4 d **4 / 4**

✓ **- 0 pts** Correct

3.5 e **4 / 4**

✓ **- 0 pts** Correct

3.6 f **4 / 4**

✓ **- 0 pts** Correct

3.7 g **3 / 4**

✓ **- 1 pts** Minor mistakes

CS 210, Fall 2020
Assignment 1: Assembly Programming Part 2
Assigned: Friday Jan. 31 Due: Friday Feb. 7 11:59pm

1 Introduction

Part 2

1. Explore how unsigned integers are represented and basic operations on them.
2. Explore how signed integers are represented as 2's complement numbers.
3. Explore how all data is represented as bytes; what changes is how we interpret them.

2 How to submit

Please provide your answers in the space provide. Once you are done, upload a copy to gradescope.

1. Implication of fixed sizes.

   Goal: Using kex, load rax with the largest unsigned integer and then have the cpu add one to it using a small assembly program. Use this setup to explore implications of fixed sized registers.

   The following walks you though how the get setup and then there are some questoins for you to answer.

   ### Setup

   ADVICE: THINK!!!! Don't treat this as instructions but rather as guidance on how we use the basic tools and facilities of a computer to explore how things work. EXPLORE!

   The following single instruction assembly program, or any equivalent of it, is what we want to use to add 1 to the current value of the rax register.

   ```
   addq $1, %rax
   ```

   You will need to create an assembly source file (using an editor), run the assembler and use a tool to create a binary memory image from it. The following is an example of two commands which do the last steps. You are free to name the files anything you like (eg. you don't have to call them myfile).

$ as myfile.s -o myfile.o

$ objcopy myfile.o --output-format=binary myfile.bin

Given how simple our program is, unlike in class, we don't have to use the linker (ld). Rather we can use the 'objcopy' command to extract out the raw opcode bytes that the assembler created for us in myfile.o (the object file) and place them in myfile.bin (a raw memory image file). We can use the unix 'od' command to print the bytes of the myfile.bin to see what they are:

$ od -A x -t x1 myfile.bin 000000 48 83 c0 01

000004

od is a very old command but very useful. It can flexibly print the values of the bytes of any file in many different formats. The left column is the location in the file for which the reset of the row is the contents. By specifying '-A x' we are telling od to print the locations in hex. As such the first row is is displaying the bytes starting at position 0 in the file. '-t x1' says that each unit of 1 byte should be displayed as hex (for more details see man od). In the above we see that the myfile.bin has 4 bytes in it with the displayed values in hex. This is the opcode values that the assembler converted our assembly program into. By default the assembler assumes that you will place the opcodes at address zero in memory. By loading this file myfile.bin into memory at address 0 of kex we will have placed our "program" into kex's memory. We can play with it using gdb commands to manipulate and direct our kex computer.

To load this into kex using gdb:

(gdb) loadmem myfile.bin 0

Restoring binary file myfile.bin into memory (0x0 to 0x4)

To confirm that your "code" is loaded you can ask gdb to display the memory contents in hex bytes and have gdb read kex's memory and try to "disassemble" the byte values back into mnemonics. eg

(gdb) x/4xb 0

0x0: 0x48 0x83 0xc0 0x01

(gdb) x/1i 0

0x0: add $0x1,%rax

This last command 'x/1i' is using the fact that gdb has a built in 'disassembler' for x86. It can do the reverse of the assembler and interpret opcodes and print out the corresponding human readable mnemonics. (See https://en.wikipedia.org/wiki/Disassembler)

For your program to be useful, you will need to load your input value into rax prior to executing the instruction that makes up you program.

(gdb) print /x $rax

(gdb) set $rax = <the max unsigned int value> (gdb) print /x $rax

(gdb) set $pc = 0 (gdb) stepi

(gdb) print /x $rax

In the above, the 'print/x' commands are printing the values of kex's rax register so that we can see how it changes as we give gdb commands or when the kex cpu executes an instruction. The 'set $rax' command uses gdb to manually set the value of kex's rax register. This is like preparing the

arguments for your program. By setting your kex computer's pc (called rip on an Intel processor) register to the location of where you program is loaded (the address of its first opcode in memory) you can then let your kex computer run one fetch, decode, execute loop with the 'stepi' command. Note: In GDB, is is possible to shorten each command to its shortest unique prefix (eg. 'print' -¿
'p'). Also, certain commands have special abbreviations (eg. 'stepi' -¿ 'si').

Now you can play around and explore how kex behaves as you put different values into rax and re- execute your program (eg. set rax to something, set the pc back to your starting memory location and then stepi, all the while looking at how the various registers change).

Specifically you will want to see how your kex cpu behaves when RAX contains the maximum unsigned integer value and kex then adds one to that (by you directing it to run the add instruction).

Questions

Please answer the following questions in the space provided. You may use gdb, the intel manuals and the textbook.

(a) (2 points) What do you think the result of the sum (the value in rax after we add 1 to it) should be?

I believe the answer would be 0x01.

(b) (5 points) What was the actual result of the sum (the value in rax after we add 1 to it)?

The actual result was 0.

(c) (5 points) What eflag single bit fields gets newly set to 1? Hints: Using gdb you can print the value of eflags with p $eflags. This command will print the names of the eflag single bit fields that are set to 1. Try doing so before and after executing your addition. You will want to read a little on eflags. eflags is also called the condition code register. See 3.6.1 of your text and you can also read about eflags in vol 1 3-15 3.4.3 "EFLAGS Register" of the Intel manuals).

Prior to the addition, only PF and SF were set to one. However, after the addition, ZF, AF, CF, and PF were set to one and SF value turned into 0. Therefore, PF remained the same but SF turned into 0.

(d) (2 points) When an instruction, like add, results in an update to a register in this unexpected way what is this condition called? You might have to do some research.

The condition is called overflow.

(e) (5 points) Without using the eflags what is a mathematical statement that indicates that this strange condition has happened given $x + y = s$? eg your job is to determine what mathematical symbol to fill in for '?' that makes this statement true when the condition has occurred. $s ? x$. Hint: see 2.3.1 in the text discusses how the sum relates to the addends.

The strange condition happens when there is an overflow where s $<$ x or s $<$ y despite the fact that x + y = s because if s is the addition of x and y, s cannot be less than x or y when x and y are unsigned.

2. 2's Complement

Using the textbook and your computer please answer the following questions. Hint gdb commands that are useful:

p/t $rax p/x
$rax p/u $rax p/d
$rax

Prints the value of rax in binary, hex, interpreted as a unsigned integer, interpreted as a signed integer (2's complement number).

(a) (2 points) What is the binary representation of -1 as a single byte?

0b11111111, 0xff, 255, -1

(b) (2 points) What is the binary representation of -1 as a 8 byte value?

0b1111111111111111111111111111111, 0xffffffff, 2147483647, -1

(c) (2 points) What is the most negative number you can represent on your kex computer in a register under 2's complement? Hint read about 2's complement and figure out how the most negative number is represented. You might also learn a lot by writing a single line assembly program that subtracts one from rax and explore how rax changes when you start it at 0.

-9223372036854775808

(d) (2 points) What is the register value (expressed in hex) that represents the most negative number that kex can represent?

0x8000000000000000

(e) (2 points) What is the largest signed integer (using 2's complement) that can be represented in an 8 byte register?

9223372036854775807

(f) (2 points) What is binary register value (expressed in hex) represents it?

0x7fffffffffffffff

(g) (2 points) What is the value of adding the most negative number with the most positive number?

-1

Interesting…is that what you expected? Yes

3. Everything is about interpretation! Given the following output, please fill in the following tables:

```
(gdb) x/64tb 0
0x0: 01010100 01101111 00100000 01100010 01100101 00101100 0010000001101111
0x8: 01110010 00100000 01101110 01101111 01110100 00100000 0111010001101111
0x10:  00100000 01100010 01100101 00101100 00100000 01110100 01101000 01100001
0x18:  01110100 00100000 01101001 01110011 00100000 01110100 01101000 01100101
0x20:  00100000 01110001 01110101 01100101 01110011 01110100 01101001 01101111
0x28:  01101110 00111010 00001010 01010111 01101000 01100101 01110100 01101000
0x30:  01100101 01110010 00100000 00100111 01110100 01101001 01110011 00100000
0x38:  01101110 01101111 01100010 01101100 01100101 01110010 00100000 01101001
```

(a) (4 points) Write the values as single bytes in hex. Hint: This is a bit of manual work but it is worth practicing binary to hex conversions.

0x0	54	6F	20	62	65	2C	20	6F
0x8	72	20	6E	6F	74	20	74	6F
0x10	20	62	65	2C	20	74	68	61
0x18	74	20	69	73	20	74	68	65
0x20	20	71	75	65	73	74	69	6F
0x28	6E	3A	0A	57	68	65	74	68
0x30	65	72	20	27	74	69	73	20
0x38	6E	6F	62	6C	65	72	20	69

(b) (4 points) As 2-byte little endian unsigned integers in hex. Hint: you will be expected to know how to do this by hand. Eg not having access to a computer.

0x0	6F54	6220	2C65	6F20	2072	6F6E	2074	6F74
0x10	6220	2C65	7420	6168	2074	7369	7420	6568
0x20	7120	6575	7473	6F69	3A6E	570A	6568	6874
0x30	7265	2720	6974	2073	6F6E	6C62	7265	6920

(c) (4 points) As 4-byte little endian unsigned integers in hex. Hint: you will be expected to know how to do this by hand. Eg not having access to a computer.

0x0	62206F54	6F202C65	6F6E2072	6F742074
0x10	2C656220	61687420	73692074	65687420
0x20	65757120	6F697473	570A3A6E	68746568
0x30	27207265	20736974	6C626F6E	69207265

(d) (4 points) As 4-byte unsigned little-endian integers in decimal. Hint: This would be a lot of work by hand while you should understand the representation I would use gdb to help with the answers.

0x0	1646292820	1864379493	1869488242	1869881460
0x10	744841760	1634235424	1936269428	1701344288
0x20	1702195488	1869182067	1460288110	1752458600
0x30	656437861	544434548	1818390382	1763734117

(e) (4 points) As 4-byte signed little-endian 2's complement integers in decimal. Hint: This would be a lot of work by hand while you should understand the representation I would use gdb to help with the answers.

0x0	1646292820	1864379493	1869488242	1869881460
0x10	744841760	1634235424	1936269428	1701344288
0x20	1702195488	1869182067	1460288110	1752458600
0x30	656437861	544434548	1818390382	1763734117

(f) (4 points) As 8-byte signed little-endian 2's complement integers in hex. Hint: you will be expect to know how to do this by hand. Eg not having access to a computer.

0x0	6F202C6562206F54	6F7420746F6E2072
0x10	616874202C656220	6568742073692074
0x20	6F69747365757120	68746568570A3A6E
0x30	2073697427207265	692072656C626F6E

(g) (4 points) Write the values as ascii characters using the following format:
For standard lower and upper case alphabet simply write the character eg. 'a','b',... For space use ' '
For newline use '\n' For tab use '\t' For ' use '\''
For " use '\"'
For all other ascii or non-ascii values use '\?'

0x0	'T'	'o'	' '	'b'	'e'	'\?'	' '	'o'
0x8	'r'	' '	'n'	'o'	't'	' '	't'	'o'
0x10	' '	'b'	'e'	'\?'	' '	't'	'h'	'a'
0x18	't'	' '	'i'	's'	' '	't'	'h'	'e'
0x20	' '	'q'	'u'	'e'	's'	't'	'i'	'o'
0x28	'n'	'\?'	'\t'	'w'	'h'	'e'	't'	'h'
0x30	'e'	'r'	' '	'\''	't'	'i'	's'	' '
0x38	'n'	'o'	'b'	'l'	'e'	'r'	' '	'i'