University of Washington – Computer Science & Engineering

Autumn 2016	Instructor:	Justin Hs	ia	2016-2	11-02		
CSE351	M	R	Ţ		31		B
Last Na	ame:		Pe	erfect			
First Na	ime:		Р	erry			
Student ID Num	ber:		12	34567			
Section you attend (circ	cle): Chris	lohn	Kevin	Sachin	Suraj Waylon	Thomas	Xi
Name of person to your Left R	ight <mark>Sa</mark>	amantha St	tudent		Larry	Learner	
All work is my own. I had no prior knowledge of the contents nor will I share the contents with oth CSE351 who haven't taken it yet. (please	ers in						

Do not turn the page until 2:30.

Instructions

- This exam contains 10 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Feel free to detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed one page (US letter, double-sided) of *handwritten* notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 50 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read *all* questions first and start where you feel the most confident.
- Relax. You are here to learn.

Question	1	2	3	4	5	Total
Possible Points	8	12	11	12	12	55

Question 1: Computer Architecture Design [8 pts]

Answer the following questions in the boxes provided with a **single sentence fragment**. Please try to write as legibly as possible.

(A) Why can't we upgrade to more registers like we can with memory? [2 pt]

Registers are part of the CPU (and the architecture) and are not modular like RAM.

(B) Why don't we see new assembly instruction sets as frequently as we see new programming languages? [2 pt]

Hard to implement/get adopted – need to build new hardware. (by comparison, a new programming language only needs a new compiler – software)

(C) Name one reason why a program written in a CISC language might run slower than the same program written in a RISC language and one reason why the reverse might be true: [4 pt]

CISC slower:	RISC slower:
Complicated instructions take longer to	Need more instructions to do complicated
execute (fewer instructions, but each is	computations (faster instructions, but
slower).	more numerous).

SID: 1234567

Question 2: Number Representation [12 pts]

- (A) What is the value of the char 0b 1101 1010 in decimal? [1 pt] If x = 0xDA, $-x = 0x26 = 2^5 + 6 = 38$ Also accepted unsigned: 0xDA = 16*13+10 = 218
- (B) What is the value of **char** z = (0xB << 6) in decimal? [1 pt] $0xB << 6 = 0b \ 1100 \ 0000 = -128 + 64 = -64$ Also accepted unsigned: 0xC0 = 192

overflow. Unsigned overflow comes naturally along with this.

(C) Let char x = 0xC0. Give one value (in hex) for char y that results in *both* signed and unsigned overflow for x+y. [2 pt] x < 0, so need large enough (in magnitude) neg num for signed

For the rest of this problem we are working with a floating point representation that follows the same conventions as IEEE 754 except using 8 bits split into the following vector widths:

Sign (1)

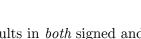
- (D) What is the *magnitude* of the **bias** of this new representation? [2 pt] Bias = $-(2^{4-1} - 1) = -7$ 7
- (E) Translate the floating point number 0b 1100 1110 into decimal. [3 pt]

 $S = 1, E = 1001_2, M = 110_2$. Notice that E indicates this is *not* a special case. Exp = 9 + (-7) = 2, $Man = 1.110_2$. $(-1)^1 \times 1.110_2 \times 2^2 = -111_2 = -7.$

(F) What is the smallest positive integer that can't be represented in this floating point encoding scheme? Hint: For what integer will the "one's digit" get rounded off? [3 pt]

Looking for number such that the $2^0=1$ bit is just off the end of the mantissa. So of the form 1.0001×2^{Exp} , with the underlined bit being 2° . Counting to the left, we find that Exp = 4, and $1.0001 \times 2^4 = 17$.

(1)	Exponent (4)	Mantissa (3)	

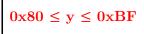


-38 or 218

-64 or 192

-7

17



Question 3: C & Assembly [11 pts]

We are writing the function toLower, which takes a char pointer and converts a string of letters (assume only letters and spaces) to lowercase, leaving spaces as spaces. <u>Example</u>: If the pointer p points to "TeST oNe", then after toLower(p), p now points to "test one".

ASCII	`A'	`Z′	Space
Binary	0b 0100 0001	0b 0101 1010	0b 0010 0000
Binary	0b 0110 0001	0b 0111 1010	0b 0010 0000
ASCII	`a'	`z′	Space

(A) Using the table of ASCII values (in binary) above, complete the function using a bitwise operator: [2 pt]

```
void toLower (char * p) {
    while(*p != 0) {
        *p = _*p | 0x20____;
        p++;
    }
}
```

(B) Fill in the blanks in the x86-64 code below with the correct instructions and operands. Remember to use the proper size suffixes and correctly-sized register names! You may assume that Lines 4, 7, and 8 are correctly filled in. [9 pt]

```
toLower(char*):
 1
               (%rdi), %rax
                                   # get *p
       movzbq
 2
       testb
                %al,
                        %al
                                  # conditional
 3
                .Exit
                                   # conditional jump
       je
   .Loop:
 4
       <<answer to part (A)>>
                                  # to lowercase
 5
                                  # update char in memory
       movb
                %al,
                        (%rdi)
 6
                $1,
                        %rdi
                                   # increment p
       addq
 7
       <<same as Line 1>>
                                   # get new *p
 8
       <<same as Line 2>>
                                   # conditional
 9
       ine
                .Loop
                                   # conditional jump
   .Exit:
10
       ret
                                   # return
```

Grading Notes for Question 3:

Line 1: must be dereference, must be 64-bit register name, p is first argument (%rdi).

- Line 2: any width specifier accepted as long as register names match
 (testq/%rax, testl/%eax, testw/%ax).
 Also accepted compq \$0, \$rax (same idea with width specifiers).
- Line 5: points awarded as long as it matched the Line 1 blank.
- Line 6: must be q width specifier because destination is %rdi.
- Line 9: points awarded as long as it was the *opposite* of the Line 3 blank.
- Line 10: retq also accepted.

Question 4: Pointers & Memory [12 pts]

char* cp = 0x10 **short*** sp = 0x08unsigned* up = 0x24

For this problem we are using a 64-bit x86-64 machine (little endian). The initial state of memory (values in hex) is shown below:

Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
0x00	AC	AB	03	01	BA	5E	BA	11
0x08	5E	00	AB	0C	BE	Α7	CE	FA
0x10	1D	в0	99	DE	AD	60	BB	40
0x18	14	CD	FA	1D	DO	41	ED	77
0x20	BA	в0	FF	20	80	AA	BE	EF

(A) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Remember to use the appropriate bit widths. [6 pt]

	Register	Value (hex)
	%rdi	0x0000 0000 0000 0003
	%rsi	0x0000 0000 0000 0005
leaw (%rsi, %rdi), %ax	%ax	0x0008
movb 8(%rdi), %bl	%bl	0x0C
movswl (,%rdi,4), %ecx	%rcx	0x0000 0000 FFFF A7BE

movb instruction pulls byte from memory at address $8+3 = 11 = 0 \times 0B$. movswl instruction pulls 2 bytes from memory starting at addresses $4^*3 = 12 = 0 \times 0$ C. Remember little-endian! Then sign extended to 32 bits, zero out top 32 bits.

(B) It's a memory scavenger hunt! Complete the C code below to fulfill the behaviors described in the comments using pointer arithmetic. [6 pt]

long v1 = (long) *(cp + ___5__); // set v1 = 0x60 unsigned* v2 = up + __7_; // set v2 = 64 long v3 = *(long *)(sp + ___3_); // set v3 = 0xB01DFACE

- v1: Byte 0x60 is at address 0x15. 0x15 cp = 5.
- v2: No dereferencing, just pointer arithmetic (scaled by sizeof(unsigned)=4). up = 0x24 = 36. To get to 64, need to add 28 (7 by pointer arithmetic).
- v3: The correct bytes can be found (in little-endian order) in addresses 0x0E-0x11. Want (0x0E - sp)/sizeof(short) = 3.

Question 5: The Stack [12 pts]

The recursive factorial function fact() and its x86-64 disassembly is shown below:

int fact(int n) {
 if(n==0 || n==1)
 return 1;
 return n*fact(n-1);

0000000000	40052d <fact>:</fact>	
40052d:	83 ff 00	cmpl \$0, %edi
400530:	74 05	je 400537 <fact+0xa></fact+0xa>
400532:	83 ff 01	cmpl \$1, %edi
400535:	75 07	jne 40053e <fact+0x11></fact+0x11>
400537:	b8 01 00 00 00	movl \$1, %eax
40053c:	eb 0d	<pre>jmp 40054b <fact+0x1e></fact+0x1e></pre>
40053e:	57	pushq %rdi
40053f:	83 ef 01	subl \$1, %edi
400542:	e8 e6 ff ff ff	call 40052d <fact></fact>
400547:	5f	popq %rdi
400548:	Of af c7	imull %edi, %eax
40054b:	f3 c3	rep ret

- (A) Circle one: [1 pt] fact() is saving %rdi to the Stack as a (Caller)// Callee
- (B) How much space (in bytes) does this function take up in our final executable? [2 pt]
 Count all bytes (2nd column) or subtract address of next instruction (0x40054d) from 0x40052d.
- (C) **Stack overflow** is when the stack exceeds its limits (i.e. runs into the Heap). Provide an argument to fact(n) here that will cause stack overflow. [2 pt]

Any negative int

We did mention in the lecture slides that the Stack has 8 MiB limit in x86-64, so since 16B per stack frame, credit for anything between 2^{19} and TMax $(2^{31}-1)$.

(D) If we use the main function shown below, answer the following for the execution of the entire program: [4 pt]

```
void main() {
    printf("result = %d\n", fact(4));
}

Total frames
    main \rightarrow fact(4) \rightarrow fact(3) \rightarrow fact(2) \rightarrow fact(1)
    \rightarrow printf
```

(E) In the situation described above where main() calls fact(4), we find that the word 0x2 is stored on the Stack at address 0x7fffdc7ba888. At what address on the Stack can we find the return address to main()? [3 pt]

0x7fffdc7ba8b0

5

Only %rdi (current n) and return address get pushed onto Stack during fact().

Address	Contents
	<rest of="" stack=""></rest>
0x7fffdc7ba8b0	Return addr to main()
0x7fffdc7ba8a8	Old %rdi (n=4)
0x7fffdc7ba8a0	Return addr to fact()
0x7fffdc7ba898	Old %rdi (n=3)
0x7fffdc7ba890	Return addr to fact()
0x7fffdc7ba888	Old %rdi (n=2)
0x7fffdc7ba880	Return addr to fact()