University of Washington – Computer Science & Engineering

Autumn 2018 Instructor: Justin Hsia 2018-10-29

CSE351 MIDTERM

Last Name:	
First Name:	
Student ID Number:	
Name of person to your Left | Right	|
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade. **(please sign)**	

Do not turn the page until 5:10.

Instructions

- This exam contains 5 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. *Please* 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 70 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	20	20	12	24	24	**100**

Question 1: Number Representation [20 pts]

(A) What is the value of the `signed char` **`0b 1000 0100`** in decimal? [2 pt]

(B) If **`a = 0x2C`**, complete the *bitwise* C statement so that **`b = 0x1F`**. [4 pt]

`b = a __ 0x____`

(C) Find the *smallest 8-bit numeral* `c` (answer in hex) such that **`c + 0x71`** causes *signed* overflow, but NOT *unsigned* overflow in 8 bits. [4 pt]

`0x`

For the rest of this problem we are working with a floating point representation that follows the same conventions as IEEE 754 except using 7 bits split into the following fields:

Sign (1)	Exponent (3)	Mantissa (3)

(D) What is the *magnitude* of the **bias** of this new representation? [2 pt]

(E) What is the decimal value encoded by **`0b1110101`** in this representation? [4 pt]

(F) What value will be read after we try to store **-18** in this representation? (Circle one) [4 pt]

–16 –NaN –∞ –18

SID: _______________

Question 2: Pointers & Memory [20 pts]

For this problem we are using a 64-bit x86-64 machine (**little endian**). The current 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	68	0C	BE	A7	CE	FA
0x10	1D	B0	99	DE	AD	60	BB	40
0x18	14	1D	EC	AF	EE	FF	C0	70
0x20	BA	B0	41	20	80	DD	BE	EF

```
char*  charP  = 0x1B
short* shortP = 0xE
```

(A) Using the values shown above, complete the C code below to fulfill the behaviors described in the comments using pointer arithmetic. [8 pt]

```
char v1 = charP[________];                  // set v1 = 0xEC

int* v2 = ((int*)shortP) + ________;        // set v2 = 0x1A
```

(B) What are the values (in hex) stored in each register shown after the following x86-64 instructions are executed? We are still using the state of memory shown above. *Remember to use the appropriate bit widths.* [12 pt]

```
leaw   (,%rsi,2),   %r15w
movswl (%rdi,%rsi), %ebp
addb   5(%rdi),     %dil
```

Register	**Data (hex)**
%rdi	0x 0000 0000 0000 000C
%rsi	0x 0000 0000 0000 0008
%r15w	0x
%rbp	0x
%dil	0x

Question 3: Design Questions [12 pts]

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

(A) What values can `S` take in an x86-64 memory operand? *Briefly* describe why these choices are useful/important. [4 pt]

Values:
Importance:

(B) Until very recently (Java 8/9), Java did not support *unsigned* integer data types. Name one advantage and one disadvantage to this decision to omit `unsigned`. [4 pt]

Advantage:
Disadvantage:

(C) **Condition codes** are part of the *processor/CPU state*. Would our instruction set architecture (ISA) still work if we got rid of the condition codes? *Briefly* explain. [4 pt]

Circle one: Yes No
Explanation:

Question 4: C & Assembly [24 pts]

Answer the questions below about the following x86-64 assembly function:

```
mystery:
        movl    $0, %eax                # Line 1
        jmp     .L2                     # Line 2
.L3:    movslq  %eax, %rdx              # Line 3
        leaq    (%rdi,%rdx,8), %rcx     # Line 4
        movq    (%rcx), %rdx            # Line 5
        xorq    $-1, %rdx               # Line 6
        addq    $1, %rdx                # Line 7
        movq    %rdx, (%rcx)            # Line 8
        addl    $2, %eax                # Line 9
.L2:    movzwl  %si, %edx               # Line 10
        cmpl    %eax, %edx              # Line 11
        jg      .L3                     # Line 12
        retq                            # Line 13
```

(A) What **variable type** would `%rdi` be in the corresponding C program? [4 pt]

____

(B) What **variable type** would the 2nd argument be in the corresponding C program? [4 pt]

____

(C) This function uses a `for` loop. Fill in the corresponding parts below, using register names as variable names (no declarations necessary). None should be blank. [8 pt]

`for ( ____________ ; ____________ ; ____________ )`

(D) If we call this function with the value **1 as the second argument**, how many jump instructions are executed (taken or untaken) in this function? [4 pt]

(E) Describe at a high level what you think this function *accomplishes* (not line-by-line). [4 pt]

Question 5: Procedures & The Stack [24 pts]

The recursive function `sum_r()` calculates the sum of the elements of an `int` array and its x86-64 disassembly is shown below:

```
int sum_r(int *ar, unsigned int len) {
    if (!len) {
        return 0;
    else
        return *ar + sum_r(ar+1,len-1);
}
```

```
0000000000400507 <sum_r>:
  400507:  41 53            pushq  %r12
  400509:  85 f6            testl  %esi,%esi
  40050b:  75 07            jne    400514 <sum_r+0xd>
  40050d:  b8 00 00 00 00   movl   $0x0,%eax
  400512:  eb 12            jmp    400526 <sum_r+0x1f>
  400514:  44 8b 1f         movl   (%rdi),%r12d
  400517:  83 ee 01         subl   $0x1,%esi
  40051a:  48 83 c7 04      addq   $0x4,%rdi
  40051e:  e8 e4 ff ff ff   callq  400507 <sum_r>
  400523:  44 01 d8         addl   %r12d,%eax
  400526:  41 5b            popq   %r12
  400528:  c3               retq
```

(A) The addresses shown in the disassembly are all part of which section of memory? [2 pt]

(B) *Disassembly* (as shown here) is different from *assembly* (as would be found in an assembly file). Name two major differences: [4 pt]

Difference 1:
Difference 2:

(C) What is the return address to `sum_r` that gets stored on the stack? Answer in hex. [2 pt]

0x

(D) What value is saved across each recursive call? Answer using a *C expression*. [2 pt]

(E) Assume `main` calls `sum_r(ar,3)` with `int ar[] = {3,5,1}`. Fill in the snapshot of memory below the top of the stack **in hex** as this call to `sum_r` returns to `main`. For unknown words, write "`0x unknown`". [6 pt]

0x7fffffffde20	<ret addr to main>
0x7fffffffde18	<original r12>
0x7fffffffde10	0x
0x7fffffffde08	0x
0x7fffffffde00	0x
0x7fffffffddf8	0x
0x7fffffffddf0	0x
0x7fffffffdde8	0x

(F) Assembly code sometimes uses *relative addressing*. The last 4 bytes of the `callq` instruction encode an integer (in *little endian*). This value represents the difference between which two addresses? Hint: both addresses are important to this `callq`. [4 pt]

value (decimal):	
address 1:	0x
address 2:	0x

(G) What could we change in the assembly code of this function to **reduce the amount of Stack memory used** while keeping it *recursive* and *functioning properly*? [4 pt]

THIS PAGE PURPOSELY LEFT BLANK

CSE 351 Reference Sheet (Midterm)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

IEEE 754 FLOATING-POINT STANDARD

Value: ±1 x Mantissa x 2^{Exponent}

Bit fields: $(-1)^S$ x 1.M x $2^{(E-\text{bias})}$

where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

E	M	Meaning
all zeros	all zeros	± 0
all zeros	non-zero	± denorm num
1 to MAX-1	anything	± norm num
all ones	all zeros	± ∞
all ones	non-zero	NaN

IEEE Single Precision and Double Precision Formats:

S (bit 31)	E (bits 30–23)	M (bits 22–0)
1 bit	8 bits	23 bits

S (bit 63)	E (bits 62–52)	M (bits 51–0)
1 bit	11 bits	52 bits

Assembly Instructions

mov a, b	Copy from a to b.
movs a, b	Copy from a to b with sign extension. Needs *two* width specifiers.
movz a, b	Copy from a to b with zero extension. Needs *two* width specifiers.
lea a, b	Compute address and store in b. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
push src	Push src onto the stack and decrement stack pointer.
pop dst	Pop from the stack into dst and increment stack pointer.
call <func>	Push return address onto stack and jump to a procedure.
ret	Pop return address and jump there.
add a, b	Add from a to b and store in b (and sets flags).
sub a, b	Subtract a from b (compute b−a) and store in b (and sets flags).
imul a, b	Multiply a and b and store in b (and sets flags).
and a, b	Bitwise AND of a and b, store in b (and sets flags).
sar a, b	Shift value of b *right* (*arithmetic*) by a bits, store in b (and sets flags).
shr a, b	Shift value of b *right* (*logical*) by a bits, store in b (and sets flags).
shl a, b	Shift value of b *left* by a bits, store in b (and sets flags).
cmp a, b	Compare b with a (compute b−a and set condition codes based on result).
test a, b	Bitwise AND of a and b and set condition codes based on result.
jmp <label>	Unconditional jump to address.
j* <label>	Conditional jump based on condition codes (*more on next page*).
set* a	Set byte a to 0 or 1 based on condition codes.

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	b-a > 0U
jb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	b-a < 0U

Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – Caller saved	%eax	%ax	%al
%rbx	Callee saved	%ebx	%bx	%bl
%rcx	Argument #4 – Caller saved	%ecx	%cx	%cl
%rdx	Argument #3 – Caller saved	%edx	%dx	%dl
%rsi	Argument #2 – Caller saved	%esi	%si	%sil
%rdi	Argument #1 – Caller saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	Callee saved	%ebp	%bp	%bpl
%r8	Argument #5 – Caller saved	%r8d	%r8w	%r8b
%r9	Argument #6 – Caller saved	%r9d	%r9w	%r9b
%r10	Caller saved	%r10d	%r10w	%r10b
%r11	Caller saved	%r11d	%r11w	%r11b
%r12	Callee saved	%r12d	%r12w	%r12b
%r13	Callee saved	%r13d	%r13w	%r13b
%r14	Callee saved	%r14d	%r14w	%r14b
%r15	Callee saved	%r15d	%r15w	%r15b

Sizes

C type	x86-64 suffix	Size (bytes)
char	b	1
short	w	2
int	l	4
long	q	8