CSE 351 Midterm Exam

Winter 2019

Thursday, February 14, 2019

Name: ______________________________

UW NetID: ______________________________

First, a quick note. This is a **take-home** exam. We are giving you certain liberties and restrictions on how you complete it (see below). Above all, we are trusting you to comply with the stated rules and to complete the exam honestly. Failure to do so will result in a **failing grade** and **disciplinary action**.

Good luck! And remember, if you're stupid enough to cheat, I'm stupid enough to catch you.

- Max

Instructions

- **You may not collaborate.** You must complete the exam alone.
- You may ask clarifying questions on Piazza. Use the `midterm` tag and make the question **only visible to instructors**.
- Show scratch work for partial credit, but put your final answers in the blanks provided.
- Write your UW NetID on the top right corner of **every page**.
- The last page is a reference sheet. Please detach it from the rest of the exam. **Do not scan the reference sheet**.
- The exam is **open course material**. You may use content from the course website and the book, including slides, lectures, and section material.
- The exam should take just under 1 hour.
- If you can't get something, relax. Show what you know and you'll get partial credit.
- The exam totals 100 regular points and 10 extra credit points.
- **The exam is due Thursday, February 14 at 11:59pm. You must scan and submit it to Gradescope according to the instructions on Piazza.**

Question:	Number representation	Pointers	x86-64	Procedures	*Extra Credit*	Total
Points:	20	30	25	25	10	110
Score:						

UW NetID: ____________________

Question 1: Number representation **(20 total points)**

(a) (6 points) If we have seven (7) bits to represent integers, what is largest unsigned number and what is largest 2s complement signed number we can represent (in decimal **and** binary)?

Largest unsigned: ____________________

Most positive signed: ____________________

Most negative signed: ____________________

(b) (3 points) Complete the code below using only the space underlined (no extra lines of C). The function `is_negative` should return 1 if `x` is negative, and 0 otherwise. You **may not use the comparison operators** < and >.

```
int is_negative(int x)
{
  return ________________________________________;
}
```

(c) (2 points) Is floating point addition associative? (Does $a + (b + c) = (a + b) + c$?) Explain in 1 sentence.

(d) (2 points) Is floating point addition commutative? (Does $a + b = b + a$?) Explain in 1 sentence.

(e) (2 points) Explain in 1-2 sentences why testing for float equality (ex: `f1 - 1.0 == f2`) is rarely useful and should be done with caution.

(f) (5 points) Does the following function always return? If so, explain what it returns. If not, explain why.

```
float add_loop() {
    float f1 = 1E30;    // this is 1 * 10^30 in decimal
    float f2 = 1E-30;
    while (f1 > 1E29) {
        f1 -= f2;
    }
    return f1;
}
```

UW NetID: ______________________________

Question 2: Pointers **(30 total points)**

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	BD	28	ED	02	35	72	3A	AF
0x08	66	6F	B1	E9	00	FF	5D	4D
0x10	86	06	04	30	64	31	8C	B3
0x18	63	78	1E	1C	25	34	EE	93
0x20	42	6C	65	67	DE	AD	BE	EF
0x28	CA	FE	D0	0D	1E	93	FA	CE

(a) (16 points) Write the value **in hexadecimal** of each expression within the commented lines at their respective state in the execution of the given program. Write UNKNOWN in the blank if the value cannot be determined.

```
int main(int argc, char** argv) {
    char *charP;
    short *shortP;
    int *intP = 0x00;
    long *longP = 0x28;

    // The value of intP is:              0x______________________

    // *intP                              0x______________________

    // &intP                              0x______________________

    // longP[-2]                          0x______________________

    charP = 0x20;
    shortP = (short *) intP;
    intP++;
    longP--;

    // *shortP                            0x______________________

    // *intP                              0x______________________

    // *((int*) longP)                    0x______________________

    // (short*) (((long*) charP) - 2)     0x______________________
}
```

(b) Classic arcade games such as PacMan displayed ranked player scores after a game over. Below we define a struct (`Score`) to store the information.

```
struct Score {
    char name[4];
    int  rank;
    long score;
};
```

Answer the following questions using the current state of memory (from previous page):

i. (2 points) What is the size (in bytes) of our struct `Score`? ______________

ii. (4 points) Given a `struct Score *p;`, write an expression equivalent to `p->rank` that uses pointer arithmetic and casting instread of struct field access notation (dot and arrow).

____

Suppose we have some array of scores (defined below) that begins at address `0x00` in the table on the previous page.

```
Score scores[3];    // Address of scores = 0x00
```

iii. (2 points) What is the value (in hex) of `scores[1].score`? 0x____________________________

iv. (2 points) Which value is greater? (Circle one)

`scores[0].name[3]`   `scores[2].name[1]`

Suppose we were to switch the order in which the fields of `Score` are declared to the following:

```
struct Score {
    char name[4];
    long score;
    int  rank;
};
```

v. (2 points) What is the size (in bytes) of our new struct `Score`? ______________

vi. (2 points) What is the size (in bytes) of our new array `scores`? ______________

Question 3: x86-64 **(25 total points)**

Suppose we have the following assembly code for a C function called `mystery`:

```
1  mystery:
2      movl $0, %eax
3  .L1:
4      movb (%rdi), %cl
5      cmpb %cl, (%rsi)
6      je .L2
7      jl .L3
8      addb $1, (%rdi)
9      jmp .L4
10 .L3:
11     addb $1, (%rsi)
12 .L4:
13     addb $1, %eax       // this line is buggy
14     jmp .L1
15 .L2:
16     ret
```

(a) (3 points) The assembler would reject line 13 because of a problem. Describe the problem in 1 sentence and provide a fix.

(b) (4 points) How does the `je` on line 6 "know" whether to jump or not? What about the `jl` on line 7? Explain in at most 1 sentence each.

(c) (18 points) Fill in the C skeleton below so that the C definition of `mystery` has the same behavior as the (fixed) assembly version above.

```
int mystery(_____________________p1, _____________________ p2) {

    int x = ________________________________________________________;

    while(________________________________________________________) {

        if(________________________________________________________) {

            ________________________________________________________;

        } else {

            ________________________________________________________;
        }

      ________________________________________________________;
    }
    return x;
}
```

Question 4: Procedures **(25 total points)**

Consider the following implementation of `accumulate` which recursively adds the first n numbers. The C and assembly are shown below.

In the questions below, **use the line numbers given to refer to parts of the assembly**.

```c
unsigned accumulate(unsigned n, unsigned total_so_far) {
    if (n == 0)
        return total_so_far;
    return accumulate(n - 1, n + total_so_far);
}
```

```
1  accumulate:
2      subq    $8, %rsp
3      movl    %esi, %eax
4      testl   %edi, %edi
5      je     .L1
6      addl    %edi, %esi
7      subl    $1, %edi
8      call    accumulate
9  .L1:
10     addq    $8, %rsp
11     ret
```

(a) (2 points) List the **callee**-saved registers (if any) used by `accumulate`.

(b) (4 points) The x86-64 ABI states (and the hardware itself prefers) that `%rsp` should be 16-byte aligned right before a `call` instruction. Which instructions are responsible for ensuring this? In a sentence or less, explain why these instructions are necessary to satisfy the 16-byte alignment of `call`s.

(c) Consider the behavior of `accumulate(6, 0)`

i. (1 point) What does it return? ____________

ii. (2 points) How many stack frames will it use in total? ____________

iii. (2 points) What is the size of a single stack frame (in bytes)? ____________

iv. (2 points) How much total stack space does this call use (in bytes)? ____________

(d) (4 points) How much total stack space does `accumulate(n, 0)` use? Write your answer in terms of n.

You wisely observe that calling and returning immediately (lines 8-11 in the assembly) seems wasteful. So you decide to replace the `call` instruction on line 8 with `jmp accumulate`. You also note that this new version **returns the same answer**. The remaining questions refer to this modification.

(e) (4 points) The modified version now violates stack discipline. After calling the modified `accumulate` procedure, a register will be different than before (and shouldn't be). What is that register? Briefly state how to fix this problem by adding, removing, or changing a small number of instructions, and **why** you can do this to this modified version of the program but not the original.

(f) (4 points) After the fix in part (e), how much stack total space will a call to `accumulate(n, 0)` procedure use? State your answer as a number of bytes in terms of n.

UW NetID: ____________________________

Question 5: *Extra Credit* *(10 total points)*

```
unsigned foobar(unsigned x) {
    x = ((x & 0x55555555) <<  1) | ((x & 0xAAAAAAAA) >>  1);
    x = ((x & 0x33333333) <<  2) | ((x & 0xCCCCCCCC) >>  2);
    x = ((x & 0x0F0F0F0F) <<  4) | ((x & 0xF0F0F0F0) >>  4);
    x = ((x & 0x00FF00FF) <<  8) | ((x & 0xFF00FF00) >>  8);
    x = ((x & 0x0000FFFF) << 16) | ((x & 0xFFFF0000) >> 16);
    return x;
}
```

What does the above function `foobar` return when given the following inputs:

(a) (2 points) `foobar(1) =` ____________________________

(b) (2 points) `foobar(0xFFF00F00) =` ____________________________

(c) (6 points) Explain in 1-2 sentences what `foobar` does in general.

This page intentionally 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: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

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

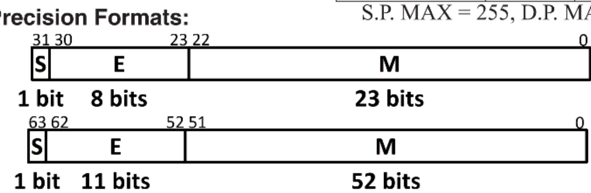
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:



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.
leaq 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 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 based on condition codes.

Conditionals

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

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