CSE351 Spring 2018, Midterm Exam
April 27, 2018

Please do not turn the page until 11:30.

Last Name:	
First Name:	
Student ID Number:	
Name of person to your left:	
Name of person to your right:	
Signature indicating: All work is my own. I had no prior knowledge of the exam contents nor will I share contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade.	

Rules:

- The exam is closed-book, closed-note, etc.
- But it contains two useful reference pages at the end that were also posted in advance. *Please remove this last piece of paper and do not turn it in.*
- **Please stop promptly at 12:20.**
- There are **100 points**, distributed **unevenly** among **6** questions (all with multiple parts):
- **The exam is printed double-sided.**

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (**18** points) (32-Bit Integers and Bit Operations)

   (a) In hex notation, write +30 (base-10) as a 32-bit twos-complement number.

   (b) In hex notation, write -30 (base-10) as a 32-bit twos-complement number.

   (c) In hex notation, write the most-positive 32-bit twos-complement number.

   (d) In hex notation, write the most-negative 32-bit twos-complement number.

   Now suppose `x` is a C `int` and a signed 32-bit twos-complement number. For each of the following C expressions answer:

   - *always zero* if the expression evaluates to 0 for every value of `x`
   - *sometimes zero* if the expression evaluates to 0 for some values of `x` but not all values of `x`
   - *never zero* if the expression evaluates to 0 for no values of `x`

   (e) `(~x) | x`

   (f) `(x << 1) & !!x`

   (g) `(x & 0x00FF) ^ (x & 0xFF00)`

   (h) `x & (x << 1)`

   (i) `~x + x + 1`

Name:____

2. (**18** points) (Pointers, Arrays, and C) For this problem assume:

- `int` is 4 bytes
- `char` is 1 byte
- machine is ***little-endian***
- The array `a` begins at address `0x100`.

Throughout this problem, if any byte of memory holds 0, you can leave it blank — this saves you some writing if you wish.

Consider this C code, which does nothing useful:

```
int a[6];
for(int i=0; i < 6; i++) {
  a[i] = i;
}
// part a
int * ip = a;
for(int i=0; i < 6; i++) {
  *ip = *ip + 2;
  ip++;
}
// part b
char * cp = (char *) a;
for(int i=0; i < 6; i++) {
  *cp = *cp + 2;
  cp++;
}
// part c
a[0] = a[0] + a[1] + a[2];
// part d
```

(a) What is the value of each *byte* of `a` when control reaches the line `// part a`? Enter a hex value in each (non-zero) square below — you don't need to write `0x`.

0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107	0x108	0x109	0x10A	0x10B	0x10C	0x10D	0x10E	0x10F	0x110	0x111	0x112	0x113	0x114	0x115	0x116	0x117

(b) What is the value of each *byte* of `a` when control reaches the line `// part b`? Enter a hex value in each (non-zero) square below — you don't need to write `0x`.

0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107	0x108	0x109	0x10A	0x10B	0x10C	0x10D	0x10E	0x10F	0x110	0x111	0x112	0x113	0x114	0x115	0x116	0x117

(c) What is the value of each *byte* of `a` when control reaches the line `// part c`? Enter a hex value in each (non-zero) square below — you don't need to write `0x`.

0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107	0x108	0x109	0x10A	0x10B	0x10C	0x10D	0x10E	0x10F	0x110	0x111	0x112	0x113	0x114	0x115	0x116	0x117

(d) What is the value of each *byte* of `a` when control reaches the line `// part d`? Enter a hex value in each (non-zero) square below — you don't need to write `0x`.

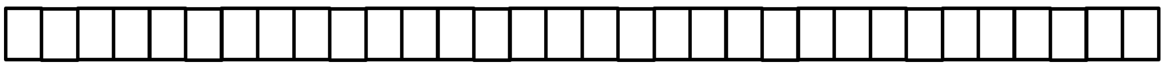
0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107	0x108	0x109	0x10A	0x10B	0x10C	0x10D	0x10E	0x10F	0x110	0x111	0x112	0x113	0x114	0x115	0x116	0x117

Name:____

3. (**17** points) (Floating-Point Numbers)

Throughout this problem, we assume single-precision (i.e., 32-bit) IEEE-754 floating-point numbers.

(a) Consider the decimal number 10.75. Give the IEEE-754 representation of this number filling in the diagram below. Hint: Remember bias and the implicit bit. Consider explaining your work for potential partial credit but explanation is not required.



(b) Consider the range of numbers between 2.0 and 12.0.

i. What is the smallest gap between any two representable numbers in this range? (You can give your answer in the form a^b. For example, 3^{-5} would be in this form.)

ii. What is the largest gap between any two representable numbers in this range? (Again you can give your answer in the form a^b.)

iii. If `x` and `y` are two representable numbers in this range and we subtract them, will we get rounding error? Answer *yes* if there will be rounding error for all such `x` and `y`, *maybe* if it depends on `x` and `y`, and *no* if rounding error is impossible.

iv. Repeat the previous rounding-error question but assume `x` and `y` are two representable numbers in the range between 10.0 and 12.0.

(c) Consider this C code:

```
void floaty_mcfloatface(float x) {
   float inf = 3.0 / 0.0; // positive infinity
   while(x < inf) {
      x += 1.0;
   }
}
```

If we call this function with a "normal" floating-point number for `x` (ignore infinities, NaN, etc.), will it terminate ("yes" for always, "maybe" for depends on `x`, "no" for never)? ***Explain your answer*** *in approximately 1–2 English sentences.*

Name:____

4. (**15** points) (x86-64 Assembly) This problem considers this assembly implementation of a C function of the from `long mystery(long x) { ... }`

```
mystery:
        movq    $0, %rax
        testq   %rdi, %rdi
        jle     .L2
.L1:
        addq    %rdi, %rdi
        addq    $1, %rax
        testq   %rdi, %rdi
        jg      .L1
.L2:
        ret
```

In parts (a)-(c) we ask you to modify the assembly code in ways that have *no effect* on the answers it produces, i.e., it should perform the same overall computation after any of your changes.

(a) Give a use of a `cmpq` instruction that could be used instead of either of the `testq` instructions.

(b) Give a use of a `shlq` instruction could be used instead of one of the `addq` instructions and indicate which instruction it is replacing.

(c) Suppose we replace the `jle .L2` with `jg .L1`. Insert an additional instruction to complete this change correctly: indicate what instruction you are adding and where.

Now we ask about what `mystery` is actually computing.

(d) Complete this description of what `mystery` computes with 1–2 English sentences: "It takes the number in `%rdi` and returns...".

(e) What is the largest number `mystery` could possibly return? Answer in base-10.

Name:____

5. (**25** points) (Assembly, Procedures, Stacks) This problem considers an assembly implementation of these two C functions:

```
long f(long s) {
  long y = s;
  g(&y,3);
  return y;
}
```

```
void g(long * p, long i) {
  if(i==0)
    return;
  *p += i;
  g(p,i-1);
  *p += i;
}
```

(a) What does f(7) return?

(b) Fill in the blanks to complete these implementations of f and g in assembly. Note some blanks give the instruction but not the operand(s) and others you choose both instruction and operand(s).

```
f:
        pushq %rdi

        movq _______, %rdi

        movq _______, _______

        call g

        ____________________

        ret
```

```
g:
        testq %rsi, %rsi
        jnz .L5
        ret
.L5:
        addq %rsi, (%rdi)
        pushq %rdi
        pushq %rsi
        subq  $1, %rsi
        call g

        ____________________

        ____________________

        addq %rsi, _________
        ret
```

(c) Suppose we call f(7) and immediately before the first instruction of f is executed, %rsp contains 0xFFFF0000. Fill in this table to give the contents of registers immediately before the first instruction of g is executed. Use hex. (Note 0xFFFF0000 is actually too small a 64-bit address to be realistic, but it works fine for an exam problem.)

	%rdi	%rsi	%rsp
First call to g			
Second call to g			
Third call to g			
Fourth call to g			

(d) Does g "do anything" to save-and-restore any caller-save registers? (yes/no without explanation)

(e) Does g "do anything" to save-and-restore any callee-save registers? (yes/no without explanation)

(f) Does g "follow the rules" for the x86-64/Linux calling convention? (yes/no without explanation)

Name:____

6. (**7** points) (Instruction-Set Architecture Design) Suppose we decide to change x86-64 to have 100 registers instead of 16. Give one-word answers to the following questions.

   (a) Would this change make it harder or easier to implement hardware that executes instructions as quickly?

   (b) Would this change make it harder or easier for software to use less stack space?

   (c) Would you expect a revised calling convention to have more caller-save registers or fewer caller-save registers?

   (d) Would you expect a revised calling convention to have more callee-save registers or fewer callee-save registers?

   (e) Would it be possible to make this change in a way that existing x86-64 executables could still run without modifying them (yes or no)?

This page intentionally blank. You can use it if you need more room than you have on another page, but please indicate on the other page to look here! Write, "see extra page!"

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

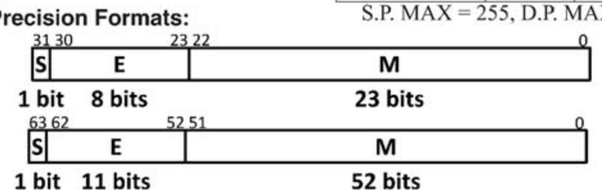
where Single Precision Bias = 127,
Double Precision Bias = 1023.

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.
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 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
jb	“Below” (unsigned >)	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