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`

   Solution:

   (a) 0x1E
   (b) 0xFFFFFFE2
   (c) 0x7FFFFFFF
   (d) 0x80000000
   (e) never zero [will always be all 1 bits, i.e., -1]
   (f) always zero [`x << 1` always has a 0 in the low-bit and `!!x` always has 0s in all but the low-bit]
   (g) sometimes zero [equivalent to `x & 0xFFFF`]
   (h) sometimes zero [zero if and only if input x does not have two consecutive 1 bits not including the left-most bit]
   (i) always zero [This is equivalent to x - x in twos-complement]

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

Solution:

(a) `0 0 0 0 1 0 0 0 2 0 0 0 3 0 0 0 4 0 0 0 5 0 0 0`

(b) `2 0 0 0 3 0 0 0 4 0 0 0 5 0 0 0 6 0 0 0 7 0 0 0`

(c) `4 2 2 2 5 2 0 0 4 0 0 0 5 0 0 0 6 0 0 0 7 0 0 0`

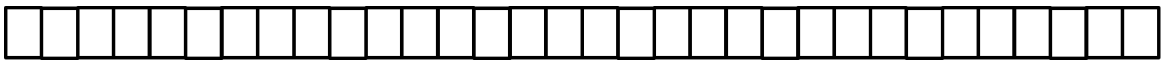
(d) `D 4 2 2 5 2 0 0 4 0 0 0 5 0 0 0 6 0 0 0 7 0 0 0`

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.*

Solution:

(a) 0 10000010 01011000000000000000000 (sign bit of 0, then exponent of 3 which means 130 biased, which is 10000010, then mantissa of 1.01011 but we do not represent the implicit bit.

(b) i. 2^{-22} (gap between all numbers in range 2.0 to 4.0)

ii. expected answer: 2^{-20} (gap between all numbers in range 8.0 to 16.0) because we meant between any two *consecutive* representable numbers, but the English is definitely vague, so we accepted 10 as a correct answer (the gap between 2.0 and 12.0).

iii. maybe

iv. no

(c) No, this function will never terminate unless called with positive infinity or NaN. For normal numbers, `x` will keep holding larger and larger values until the exponent is high enough that adding 1.0 does not increase the value of `x` due to rounding, after which point `x` never changes and the loop continues forever.

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.

Solution:

(a) `cmpq $0, %rdi`

(b) `shlq $1, %rdi` for the first `addq` instruction

(c) Intended answer: `ret` or `jmp .L2` needs to be added after this first jump, i.e., immediately before `.L1`. But it also works to put [back] `jle .L2` either before or after the `.L1` or even before the `jg .L1`, so that also receives full credit.

(d) It takes the number in `%rdi` and returns the number of times it needs to be doubled before the repeated doubling produces a non-positive number. (If the original number was positive, this will be due to overflow.) An alternate description is it returns 63 minus the bit position of the left-most 1-bit in `%rdi` where the least-significant bit is position 0 and returning 0 if there is no 1-bit.

(e) 63

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)

Solution:

(a) 19

(b)

```
f:                        g:
        pushq %rdi                testq %rsi, %rsi
                                  jnz .L5
        movq %rsp, %rdi           ret
                          .L5:
        movq $3, %rsi             addq %rsi, (%rdi)
                                  pushq %rdi
        call g                    pushq %rsi
                                  subq  $1, %rsi
        popq %rax                 call g

        ret                       popq %rsi

                                  popq %rdi # can be another caller-save register
                                            # but same register must be on next line
                                  addq %rsi, (%rdi)
                                  ret
```

(c)

	%rdi	%rsi	%rsp
First call to g	0xFFFEFFF8	0x3	0xFFFEFFF0
Second call to g	0xFFFEFFF8	0x2	0xFFFEFFD8
Third call to g	0xFFFEFFF8	0x1	0xFFFEFFC0
Fourth call to g	0xFFFEFFF8	0x0	0xFFFEFFA8

(d) Yes

(e) No

(f) Yes (The previous "no" is not a problem because it does not use any callee-save registers, so they're implicitly preserved.)

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)?

Solution:

(a) harder

(b) easier

(c) more

(d) more

(e) yes

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!"