University of Washington – Computer Science & Engineering

Summer 2018 Instructor: Justin Hsia 2018-07-18

CSE351 MIDTERM

Last Name:	**Perfect**	
First Name:	**Perry**	
Student ID Number:	1234567	
Name of person to your Left | Right	Samantha Student	Larry Learner
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 10:50.

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 60 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	22	20	14	24	20	**100**

Question 1: Number Representation [22 pts]

Consider the **signed char x = 0b 1010 1000**.

(A) What is the value of `x`? You may answer as the sum of powers of 2. [2 pt]

$\boxed{-2^7+2^5+2^3 = -88}$

(B) In theory (math), what is the difference in the values of **(unsigned char)x** and **x**? Your answer should be a *decimal number*. [2 pt]

`(unsigned char)x` has *value* $2^7+2^5+2^3$.

$\boxed{2^n = 2^8 = 256}$

(C) In C, what is the value of **char y = (unsigned char)x – x;**? [2 pt]

Subtraction is done at the bit level, so the result is all zeros.
Also, the "result" of 256 can't be represented by an 8-bit integer.

$\boxed{0}$

(D) Which of the following expressions will result in a *positive* (non-negative, non-zero) result? (Circle one) [4 pt]

`x<<4`	[✓ `x^0x81`]	`x|~x`	`!(x>>1)`
`0b10000000`	`0b00101001`	`0b11111111`	`0`

For the rest of this problem we are working with a new floating point datatype (**flo**) that follows the same conventions as IEEE 754 except using 8 bits split into the following fields:

Sign (1)	Exponent (3)	Mantissa (4)

(E) What is the value of the numeral from above **0b 1010 1000** in this representation? [4 pt]

S = 1, E = 0b010, M = 0b1000. bias=2^{3-1}-1=3.
$(-1)^1 \times 1.1000_2 \times 2^{2-3} = -1.1_2 \times 2^{-1} = -0.11_2 = -(0.5 + 0.25)$

$\boxed{-0.75}$

(F) What is the *encoding* of the **most negative real number that we can represent** (∞ is not a real number) in this floating point scheme (binary)? [4 pt]

Largest *normalized* number, but negative: `0b` **1 110 1111**

(G) What will occur if we cast **flo f = (flo)x** (*i.e.* try to represent the value stored in `x` as a `flo`)? (Circle one) [4 pt]

Rounding Underflow [✓ Overflow] None of these

$-88 = -(64+16+8) = -(2^6+2^4+2^3) = -1011000_2 = -1.0110\times 2^6$. Mantissa fits, but max exponent is 0b110 – bias = 6 – 3 = 3.

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:

```
char* charP = 0x12
int*  intP  = 0x8
long* longP = 0x30
```

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	AA	BE	EF

(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 + ___9___);      // set v1 = 0xAF

int* v2 = &intP[___3___];          // set v2 = 0x14
```

v1: Byte `0xAF` is at address `0x1B`. `0x1B - charP = 9`.

v2: No dereferencing; just pointer arithmetic (scaled by `sizeof(int) = 4`).
`intP = 0x8`. To get to `0x14`, need to add `12` (3 by pointer arithmetic).

(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]

```
leab    7(%rsi),        %r9b
movl    (%rdi,%rsi),    %eax
movzbq -2(,%rsi,8),     %r8
```

Register	Data (hex)
%rdi	0x 0000 0000 0000 0019
%rsi	0x 0000 0000 0000 0003
%r9b	0x 0A
%eax	0x 70C0 FFEE
%r8	0x 0000 0000 0000 0000 00BB

`leab` calculates address `7 + 0x3`.

`movl` pulls *four* bytes starting at memory address `0x1C` (remember little endian!).

`movzbq` instruction pulls the byte at memory address `0x3*8-2` = `22 = 0x16`, which is `0xBB`. Then zero-extend out to 8 bytes.

Question 3: Design Questions [14 pts]

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

(A) Name the two issues with Sign and Magnitude that Two's Complement fixed. [4 pt]

- Two zeros.

- Bad arithmetic.
- Negatives incrementing in the wrong direction.

(B) *Briefly* describe an advantage of making addresses and registers both the width of a *word*. [4 pt]

- Addresses are guaranteed to be able to fit/be stored in a register.
- Memory operands can be specified easily using registers.

(C) *Briefly* explain your answers to the following questions if we moved 1 bit from the mantissa field (now 22 bits) to the exponent field (now 9 bits) in floating point: [6 pt]

Will the total number of *representable floats* (normalized + denormalized + special cases) change? Circle one: Yes [✓ No]

Explanation: We still have the same fixed width (32 bits), so we can still only represent 2^{32} floats. We've changed the *numbers* we represent, but not how many we represent.

Credit also given if Yes was argued by saying that there are more NaN's than before, so fewer floats representable. This relies on counting all NaN's as a single "special case," which wasn't the intention of the question, but is a reasonable interpretation.

The frequency of *rounding* will (circle one): [✓ Increase,] Decrease, or Stay the same

Explanation: By shortening the M field, we lose precision. There are now more numbers than before that will have some value rounded off when encoded (any number whose mantissa has 1's at least 23 digits apart instead of at least 24 digits apart).

The gaps between representable numbers doubled for the same exponent, so it's easier to fall in-between representable numbers and get rounded.

Question 4: C & Assembly [24 pts]

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

```
mystery:
        movl    $0, %eax             # Line 1
.L2:    cmpl    %esi, %eax           # Line 2
        jge     .L1                  # Line 3
        movslq  %eax, %rdx           # Line 4
        leaq    (%rdi,%rdx,2), %rcx  # Line 5
        movzwl  (%rcx), %edx         # Line 6
        andl    $1, %edx             # Line 7
        movw    %dx, (%rcx)          # Line 8
        addl    $1, %eax             # Line 9
        jmp     .L2                  # Line 10
.L1:    retq                         # Line 11
```

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

`%rcx` is calculated from `%rdi` with scale 2 (Line 5) and then dereferenced with a `movzwl` instruction (Line 6).

__**short***__ `rdi`

(B) *Briefly* describe why Line 4 is needed before Line 5. [4 pt]

Memory operands (Line 5) must take 64-bit register names, since addresses are 8 bytes wide. So the 4-byte value in `%eax`, must be extended to 8 bytes beforehand.

(C) This function uses a `for` loop. Fill in the corresponding parts below, using register names as variable names. None should be blank. [8 pt]

`for ( __`**`eax = 0`**`___ ; __`**`eax < esi`**`__ ; __`**`eax++`**`__ )`

Init is from Line 1, Test is from Lines 2-3, Update is from Line 9.

(D) If we call this function with the value **3 as the second argument**, what value is returned? [4 pt]

Return value is `%rax` and we exit the loop when `%eax` = `%esi`.

3

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

Overrides an array of `short`s with the parity of the entries (1 for odd, 0 for even – given by the least significant bit).

Question 5: Procedures & The Stack [20 pts]

The recursive power function `power()` calculates `base^pow` and its x86-64 disassembly is shown below:

```
int power(int base, unsigned int pow) {
  if (pow) {
    return base * power(base,pow-1);
  }
  return 1;
}
```

```
00000000004005a0 <power>:
  4005a0:  85 f6           testl  %esi,%esi
  4005a2:  74 10           je     4005b4 <power+0x14>
  4005a4:  53              pushq  %rbx
  4005a5:  89 fb           movl   %edi,%ebx
  4005a7:  83 ee 01        subl   $0x1,%esi
  4005aa:  e8 f1 ff ff ff  call   4005a0 <power>
  4005af:  0f af c3        imull  %ebx,%eax
  4005b2:  eb 06           jmp    4005ba <power+0x1a>
  4005b4:  b8 01 00 00 00  movl   $0x1,%eax
  4005b9:  c3              ret
  4005ba:  5b              popq   %rbx
  4005bb:  c3              ret
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pt]

Count all bytes (middle columns) or subtract address of next instruction (`0x4005bc`) from `0x4005a0`.

28 B

(B) Circle one: The label `power` will show up in which table(s) in the object file? [4 pt]

Symbol Table **Relocation Table** [✓ **Both Tables**] **Neither Table**

`power` is called in this file (recursively) and can be called by external files, so in both.

(C) Which register is being saved on the stack? [2 pt]

See `pushq` instruction (`0x4005a4`).

`%rbx`

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

The address of the instruction *after* `call`.

0x4005af

(E) Assume `main` calls `power(8,3)`. Fill in the snapshot of memory below the top of the stack **in hex** as this call to `power` returns to `main`. For unknown words, write "unknown". [6 pt]

0x7fffeca3f748	<ret addr to main>	power(8,3)
0x7fffeca3f740	<original rbx>	
0x7fffeca3f738	**0x4005af <ret addr>**	power(8,2)
0x7fffeca3f730	**0x8 <base>**	
0x7fffeca3f728	**0x4005af <ret addr>**	power(8,1)
0x7fffeca3f720	**0x8 <base>**	
0x7fffeca3f718	**0x4005af <ret addr>**	power(8,0)
0x7fffeca3f710	**unknown**	

The base case doesn't push `%rbx` onto the stack, so `0x7fffeca3f710` remains unknown.

(F) Harry the Husky claims that we could have gotten away with not pushing a register onto the stack in `power`. Is our intrepid school's mascot correct or not? Briefly explain. [4 pt]

Harry is correct! `base` doesn't change between recursive calls and `power` doesn't call other procedures, so there is no need to save `%rdi` in `%rbx`.

In fact, if you compile the C function with an optimization flag of `-O2`, it doesn't push `%rbx` onto the stack!