University of Washington – Computer Science & Engineering

Autumn 2019 Instructor: Justin Hsia 2019-10-28

CSE351 MIDTERM

Last Name:	**Perfect**	
First Name:	**Perry**	
Student ID Number:	1234567	
Name of person to your Left | Right	Samantha Student	Samantha Student
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 5:30.

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) Convert the decimal number **-12** into *5-bit* two's complement. Answer in binary. [2 pt]

MSB has weight $-2^4 = -16$. $-12 = -16 + 4$.

`0b` **`10100`**

(B) If **`signed char a = 0x88`**, complete the *bitwise* C statement so that **`b = 0xF1`**. The first blank should be an operator and the second blank should be a numeral. [4 pt]

```
   0b 1000 1000
^  0b 0111 1001      OR      0b 1000 1000 >> 3
   0b 1111 0001            (arithmetic right shift)
```

```
b = a ^  0x79
b = a >> 0x3
```

(C) Find the *largest 8-bit unsigned numeral* `c` (answer in hex) such that **`c + 0x80`** causes NEITHER *signed* nor *unsigned* overflow in 8 bits. [4 pt]

Unsigned overflow will occur for `c > 0x80`.
Signed overflow can only happen if `c` is negative (also `> 0x80`).

`0x` **`7F`**

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

Sign (1)	Exponent (5)	Mantissa (2)

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

$2^{5-1}-1 = 15$

(E) What is the decimal value encoded by **`0b 1100 1001`** in this representation? [4 pt]

`-10`

$S = 1$, $E = 0b10010 = 18$, $M = 0b01$
Value $= (-1)^1 \times 1.01_2 \times 2^{18-15} = -1.01_2 \times 2^3 = -1010_2 = -10$

(F) What is the **smallest positive integer** that *can't be represented* in this floating point encoding scheme? <u>Hint</u>: for what integer will the "one's digit" get rounded? [4 pt]

`9`

Look for the number such that the first bit off the right end of the mantissa has the value $2^0 = 1$. In this case, that means $1.00\underline{1}_2 \times 2^{Exp}$, with the underlined bit being 2^0. The underlined bit has the value $2^{-3} \times 2^{Exp} = 2^{Exp-3} = 2^0$, meaning Exp = 3 and $1.00\underline{1}_2 \times 2^3 = 1001_2 = 9$.

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  = 0xD;
short* shortP = 0x1E;
```

Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
0x00	20	F6	EF	EA	A2	5E	9F	1A
0x08	A2	D0	4F	C4	A0	0C	F7	27
0x10	B8	BD	1A	CA	35	95	CB	80
0x18	84	3F	02	4F	8E	F3	F6	E5
0x20	CD	4A	F6	48	1A	6F	7E	63

(A) Using the values shown above, fill in the C type and hex value for each of the following C expressions. Leading zeros are not required for the hex values. [8 pt]

C Expression	C Type	Hex Value
`*(charP + 6)`	**char**	0x **CA**
`(int**)shortP - 2`	**int****	0x **E**

charP: 0xD + 6 (scaled by `sizeof(char) = 1`) yields 0x13. Address 0x13 holds the char 0xCA.

shortP: 0x1E – 2 (scaled by `sizeof(int*) = 8`) yields 0xE.

(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 data widths.* [12 pt]

```
leal    2(,%rsi,8),     %r10d
movswq  (%rdi,%rdi,2),  %r11
cmpb    0x19(%rsi),     %dil
```

Register	Data (hex)
%rdi	0x 0000 0000 0000 0004
%rsi	0x 0000 0000 0000 0001
%r10d	0x **0000 000A**
%r11	0x **0000 0000 0000 0CA0**
%dil	0x **04**

`leal` calculates address `2 + 0x1 × 8 = 10 = 0xA` in 4 bytes.

`movswq` instruction pulls two bytes starting at memory address `0x4 + 0x4 × 2 = 0xC`, which is `0x0CA0` (remember little endian!). Then sign-extend (copies MSB of 0) out to 8 bytes.

cmp doesn't store its result, so there's no change to `%dil` (lowest byte of `%rdi`)!

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) While the floating point special cases seem arbitrary, there is a method to the madness. *Briefly* describe why the following choices were made: [4 pt]

The `E` encoding for ∞: Some possible answers:

- Largest `E` to be larger than normalized numbers in comparisons.
- One higher than largest normalized `E` so that overflow naturally results in ∞.

Multiple encodings for NaN: Some possible answers:

- To parallel the fact that there are multiple ways to generate a NaN.
- To help with debugging the cause of the NaN.

(B) When we cast between an integer data type and a floating point one, the conversion is done by encoding the original *value*, changing the stored bits. Imagine if we instead did the conversion by leaving the bits the same, but interpreting them differently. Name an advantage and a disadvantage of this change. [4 pt]

Advantage: Some possible answers:

- No loss of data if we cast from between integer and floating point representations and then back again.
- The hardware conversion between integer and floating point becomes faster/easier/simpler.

Disadvantage: Some possible answers:

- No well-defined relationship between the converted values.
- Breaks compatibility with code that relies on the preservation of the value.

(C) Assume we have an address space of 2^w **bytes**. If we decided to **assign an address to every 4 bits** instead of every byte, what is the new width of an address? Also, name one change we would need to make to the existing x86-64 instruction syntax. [4 pt]

New address width: **`w+1`** bits

Instruction syntax change: Some possible answers:

- Introduce a new instruction suffix/width specifier for 4 bits.
- Allow a scale factor of 16 for memory operands.

Question 4: C & Assembly [24 pts]

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

```
mystery:
        jmp     .L2                 # Line 1
.L4:    addq    $1, %rdi            # Line 2
        movb    %al, (%rsi)         # Line 3
        leaq    1(%rsi), %rsi       # Line 4
.L2:    movzbl  (%rdi), %eax        # Line 5
        testb   %al, %al            # Line 6
        je      .L3                 # Line 7
        cmpb    %dl, %al            # Line 8
        jne     .L4                 # Line 9
.L3:    movb    $0, (%rsi)          # Line 10
        retq                        # Line 11
```

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

Line 5: we read a byte out of memory by dereferencing the value in `%rdi`. **`unsigned char *`** also accepted due to zero-extension.

_____**char ***_____

(B) What **variable type** would the 3rd argument be in the corresponding C program? [4 pt]

Line 8: `%dl` (lowest byte of `%rdx`) is compared to the byte read out of memory.

______**char**______

(C) This function uses a `while` loop. Fill in the two conditionals below, using register names as variable names (no declarations necessary). [8 pt]

```
                   al
                 al != 0
                  *rdi              al != dl
while ( _*rdi != 0__ && _*rdi != dl_ )
```

Conditional 1 is from Lines 6-7, which exit the loop if `%al = 0`.
Conditional 2 is from Lines 8-9, which loop back if `%al - %dl != 0`.

(D) Taking the variable types into account, describe at a high level what the *purpose* of Line 10 is (not just what it does mechanically). [4 pt]

Adds a null terminator (char with value 0) to the end of `*rsi` (the destination string).

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

It copies all of the characters from a source string (in `%rdi`) to a destination string (in `%rsi`) until it sees a specified character (in `%dl`) or the end of the source string. The destination string is then null-terminated.

Question 5: Procedures & The Stack [24 pts]

The recursive function `pcount_r()` from lecture calculates the "popcount" of `x`, returning the number of 1's in the unsigned binary representation. However, a faulty compiler has produced the **BUGGY** x86-64 disassembly shown below:

CORRECT

```
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    return (x & 1) + pcount_r(x >> 1);
}
```

BUGGY

```
00000000004005d7 <pcount_r>:
  4005d7:  b8 00 00 00 00  movl   $0x0,%eax
  4005dc:  48 85 ff        testq  %rdi,%rdi
  4005df:  75 02           jne    4005e3 <pcount_r+0xc>
  4005e1:  f3 c3           repz retq
  4005e3:  48 d1 ef        shrq   $1, %rdi
  4005e6:  e8 ec ff ff ff  callq  4005d7 <pcount_r>
  4005eb:  83 e7 01        andl   $0x1,%edi
  4005ee:  48 01 f8        addq   %rdi,%rax
  4005f1:  c3              retq
```

(A) What is the address of the code that comes after the *function* `pcount_r` (not the label) in memory? [2 pt]

The last instruction in `pcount_r` is `retq`, which is at address `0x4005f1` and is only 1 byte long.

0x **4005f2**

(B) Circle one: The variable `x` will show up in which table(s) in the object file? [2 pt]

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

`x` is a local variable and doesn't have an associated label, so it won't be in either table.

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

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

0x **4005eb**

(D) To see what's going wrong with this implementation, trace the execution for `pcount_r(1)`. What are the expected and actual return values? [4 pt]

There is one 1 in 1. `%rdi` and `%rax` are both 0 after the recursive call.

Expected: **1**

Actual
: **0**

(E) Assume `main` calls our buggy `pcount_r(5)`. Fill in the snapshot of memory below the top of the stack **in hex** as this call to `pcount_r` returns to `main`. For unknown words, write "`0x unknown`". [4 pt]

`0x7fffffffdc98`	`<ret addr to main>`	`pcount_r(5)`
`0x7fffffffdc90`	`0x` **`4005eb`**	`pcount_r(2)`
`0x7fffffffdc88`	`0x` **`4005eb`**	`pcount_r(1)`
`0x7fffffffdc80`	`0x` **`4005eb`**	`pcount_r(0)`
`0x7fffffffdc78`	`0x` **`unknown`**	
`0x7fffffffdc70`	`0x` **`unknown`**	

All minimal stack frames, including the base case.

(F) Now let's go about fixing the assembly code. During your execution trace in part D, think about when you encountered a value that you didn't expect. Which register did this happen to? As a *C expression*, what value was supposed to be held in this register? Is this caller- or callee-saved? [4 pt]

Register: **`%rdi`** Expression: **`x`** Type: caller-saved

In part D, at `0x4005eb` we expected `%edi = 1` (old value of `x`), but it was actually `0`.

(G) Now we need to add a `pushq` and `popq` instruction for the register you identified above. Pay attention to the register saving convention you identified above. Give the addresses of the instruction you would place the new instruction *just before*. For example, write "`0x 4005e1`" if you wanted to place an instruction just before the `repz retq`. [6 pt]

`pushq` address: `0x` **`4005e3`**

`popq` address: `0x` **`4005eb`**

Because `%rdi` is caller-saved, the saving and restoring should happen around the recursive call. The push has to happen before `call` *and* before we modify `x`/`%rdi`. The pop has to happen after `call` but before we use `%edi`.

End of Exam

Did you write your Student ID Number on the top-right corner of every odd page?