University of Washington – Computer Science & Engineering

Autumn 2018 Instructor: Justin Hsia 2018-10-29

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 5:10.

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) What is the value of the `signed char` **`0b 1000 0100`** in decimal? [2 pt]

-128+4 = -124

(B) If **`a = 0x2C`**, complete the *bitwise* C statement so that **`b = 0x1F`**. [4 pt]

```
  0b 0010 1100
^ 0b 0011 0011
  0b 0001 1111
```

`b = a ^ 0x33`

(C) Find the *smallest 8-bit numeral* c (answer in hex) such that **`c + 0x71`** causes *signed* overflow, but NOT *unsigned* overflow in 8 bits. [4 pt]

For signed overflow, need (+) + (+) = (−).
For no unsigned overflow, need no carryout from MSB.
The first (−) encoding we can reach from 0x71 is 0x80.
0x80 − 0x71 = 0xF.

0x **0F**

For the rest of this problem we are working with a floating point representation that follows the same conventions as IEEE 754 except using 7 bits split into the following fields:

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

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

$2^{3-1}-1 = 3$

(E) What is the decimal value encoded by **`0b1110101`** in this representation? [4 pt]

-13

S = 1, E = 0b110 = 6, M = 0b101

Value = $(-1)^1 \times 1.101_2 \times 2^{6-3} = -1.101_2 \times 2^3 = -1101_2 = -13$

(F) What value will be read after we try to store **-18** in this representation? (Circle one) [4 pt]

−16 −NaN [✓ −∞] −18

$-18 = -(16+2) = -(2^4 + 2^1) = -1.001_2 \times 2^4$.

The largest normalized exponent we can encode is 0b110 → Exp = 3, so this causes overflow, resulting in −∞ being stored (as 0b1111000).

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:

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	DD	BE	EF

```
char*  charP  = 0x1B
short* shortP = 0xE
```

(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[__-1___];                  // set v1 = 0xEC

int* v2 = ((int*)shortP) + ___3___;        // set v2 = 0x1A
```

v1: Byte 0xEC is at address 0x1A. 0x1A – charP = -1.

v2: No dereferencing; just pointer arithmetic (scaled by `sizeof(int) = 4`).
`shortP=0xE=14`. To get to `0x1A=26`, 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]

```
leaw    (,%rsi,2),     %r15w
movswl  (%rdi,%rsi),   %ebp
addb    5(%rdi),       %dil
```

Register	**Data (hex)**
%rdi	0x 0000 0000 0000 000C
%rsi	0x 0000 0000 0000 0008
%r15w	0x **0010**
%rbp	0x **0000 0000 0000 60AD**
%dil	0x **BC**

`leaw` calculates address `0x8 × 2`. Can use left shifting to do this multiplication.

`movswl` instruction pulls two bytes starting at memory address `0xC+0x8 = 0x14`, which is `0x60AD` (remember little endian!). Then sign-extend out to 32 bits, with the upper 4 bytes being automatically zeroed out.

`addb` pulls the byte from memory at address `0xC+5=0x11` (`0xB0`) and adds it to the lowest byte of `%rdi` (`0x0C`).

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) What values can `S` take in an x86-64 memory operand? *Briefly* describe why these choices are useful/important. [4 pt] – a memory operand is of the form `D(Rb,Ri,S)`.

Values: 1, 2, 4, 8

Importance: These values represent the different scaling factors used in pointer arithmetic based on the data type sizes.

(B) Until very recently (Java 8/9), Java did not support *unsigned* integer data types. Name one advantage and one disadvantage to this decision to omit `unsigned`. [4 pt]

Advantage: Some possible answers:

- Less confusing/more consistent arithmetic interpretations for the programmer.
- Fewer cases of implicit casting.
- Fewer data types to worry about.

Disadvantage: Some possible answers:

- Need to use larger data widths for numbers in the range (TMax, UMax] for a given width.
- More difficult to do unsigned comparisons.
- More difficult to do zero-extension.

(C) **Condition codes** are part of the *processor/CPU state*. Would our instruction set architecture (ISA) still work if we got rid of the condition codes? *Briefly* explain. [4 pt]

Circle one: Yes [✓ No]

Explanation: Our jump and set instructions, which rely on the values of the condition codes, would no longer work. Without jump instructions, we couldn't implement most of our program's control flow.

Question 4: C & Assembly [24 pts]

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

```
mystery:
         movl    $0, %eax             # Line 1
         jmp     .L2                  # Line 2
.L3:     movslq  %eax, %rdx           # Line 3
         leaq    (%rdi,%rdx,8), %rcx  # Line 4
         movq    (%rcx), %rdx         # Line 5
         xorq    $-1, %rdx            # Line 6
         addq    $1, %rdx             # Line 7
         movq    %rdx, (%rcx)         # Line 8
         addl    $2, %eax             # Line 9
.L2:     movzwl  %si, %edx            # Line 10
         cmpl    %eax, %edx           # Line 11
         jg      .L3                  # Line 12
         retq                         # Line 13
```

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

Line 4: we compute array index address `%rcx` from `%rdi` with a scale factor of **8** (`long`). Line 5: address is dereferenced (pointer).

____**`long *`**____

(B) What **variable type** would the 2nd argument be in the corresponding C program? [4 pt]

Line 10: we use a `mov`**`zw`**`l` on `%si`.

`unsigned short`

(C) This function uses a `for` loop. Fill in the corresponding parts below, using register names as variable names (no declarations necessary). None should be blank. [8 pt]

`eax < si`

`for ( `__**`eax = 0`**___` ; `__**`eax < edx`**_` ; `__**`eax += 2`**__` )`

Init is from Line 1, Test is from Lines 2-4, Update is from Line 9.

Both `%si` (Line 10) and `%edx` (Line 11) were accepted in the Test comparison.

(D) If we call this function with the value **1 as the second argument**, how many jump instructions are executed (taken or untaken) in this function? [4 pt]

Line 2 once (unconditional), Line 12 twice (taken when `%eax = 0`, then untaken when `%eax = 2`).

3

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

It negates (–x = (x ^ -1) + 1) every even index of an array (*i.e.* every other starting with index 0).

Question 5: Procedures & The Stack [24 pts]

The recursive function `sum_r()` calculates the sum of the elements of an `int` array and its x86-64 disassembly is shown below:

```
int sum_r(int *ar, unsigned int len) {
    if (!len) {
        return 0;
    else
        return *ar + sum_r(ar+1,len-1);
}
```

```
0000000000400507 <sum_r>:
  400507:  41 53           pushq  %r12
  400509:  85 f6           testl  %esi,%esi
  40050b:  75 07           jne    400514 <sum_r+0xd>
  40050d:  b8 00 00 00 00  movl   $0x0,%eax
  400512:  eb 12           jmp    400526 <sum_r+0x1f>
  400514:  44 8b 1f        movl   (%rdi),%r12d
  400517:  83 ee 01        subl   $0x1,%esi
  40051a:  48 83 c7 04     addq   $0x4,%rdi
  40051e:  e8 e4 ff ff ff  callq  400507 <sum_r>
  400523:  44 01 d8        addl   %r12d,%eax
  400526:  41 5b           popq   %r12
  400528:  c3              retq
```

(A) The addresses shown in the disassembly are all part of which section of memory? [2 pt]

Text or `.text` also accepted.

Instructions/Code

(B) *Disassembly* (as shown here) is different from *assembly* (as would be found in an assembly file). Name two major differences: [4 pt]

Differences: Some possible answers include:

- No machine code (middle column) would be shown in the assembly (*i.e.* the code hasn't been assembled yet).
- Finalized addresses would not be found in the assembly (left column).
- All labels would still be symbolic/named in the assembly instructions (*e.g.* `jne`, `jmp`, `callq`).

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

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

0x **400523**

(D) What value is saved across each recursive call? Answer using a *C expression*. [2 pt]

The instruction at address `0x400514` dereferences `%rdi` and stores the value in `%r12d`.

`*ar`

(E) Assume `main` calls `sum_r(ar,3)` with `int ar[] = {3,5,1}`. Fill in the snapshot of memory below the top of the stack **in hex** as this call to `sum_r` returns to `main`. For unknown words, write "`0x unknown`". [6 pt]

`0x7fffffffde20`	`<ret addr to main>`	`sum_r(ar,3)`
`0x7fffffffde18`	`<original r12>`	
`0x7fffffffde10`	`0x` **`400523 <ret addr>`**	`sum_r(ar+1,2)`
`0x7fffffffde08`	`0x` **`3 <*ar>`**	
`0x7fffffffde00`	`0x` **`400523 <ret addr>`**	`sum_r(ar+2,1)`
`0x7fffffffddf8`	`0x` **`5 <*ar>`**	
`0x7fffffffddf0`	`0x` **`400523 <ret addr>`**	`sum_r(ar+3,0)`
`0x7fffffffdde8`	`0x` **`1 <*ar>`**	

The base case DOES still push `%r12` onto the stack.

(F) Assembly code sometimes uses *relative addressing*. The last 4 bytes of the `callq` instruction encode an integer (in *little endian*). This value represents the difference between which two addresses? Hint: both addresses are important to this `callq`. [4 pt]

`0xffffffe4` = `-(0x1b + 1)` = `-28` value (decimal): **-28**

This corresponds to the address we jump to. address 1: 0x **400507**

This corresponds to the return address. address 2: 0x **400523**

(G) What could we change in the assembly code of this function to **reduce the amount of Stack memory used** while keeping it *recursive* and *functioning properly*? [4 pt]

The issue with recursive functions is that no matter what kind of register you use to save a value (caller-saved or callee-saved), the recursive call will overwrite that value because it's an identical function! So we actually *can't* avoid pushing something to the stack without making the function iterative. So any potential saving of Stack space will come from the base case. Keep reading for two possible solution types:

Callee-saved: `%r12` is a *callee*-saved register. This means that its old value just needs to be saved before we overwrite its value; it does not need to be saved at the very top of `sum_r`.

1) Move the `pushq` instruction into the recursive case (below the `jmp` instruction).
2) Either make the `jmp` go to address `0x400528` instead OR
   move the `movl $0,%eax` above the `jne` and change the `jne` to `je 0x400528`.

Caller-saved: The value we really care about saving across the recursive call (`ar` or `*ar`), already starts in a caller-saved register in `%rdi`! This value must then be saved before we make a recursive call to `sum_r` and restored once it returns:

1) Convert the `pushq %r12` to `pushq %rdi` and move it down to *replace* the `movl (%rdi),%r12d` instruction.
2) Convert the `popq %r12` to `popq %rdi` and move it right after/below the `callq`.
3) Convert the `addl %r12d,%eax` to `addl (%rdi),%eax`.