University of Washington – Computer Science & Engineering

Autumn 2017 Instructor: Justin Hsia 2017-10-30

CSE351 MIDTERM

Last Name:	
First Name:	
Student ID Number:	
Name of person to your Left | Right	|
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 8 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	11	10	6	12	11	**50**

Question 1: Number Representation [11 pts]

(A) Convert the number **-25** into a *6-bit* signed representation. Answer in binary. [1 pt]

0b

(B) What is the stored result of **`signed char c = (0x79 ^ (~0)) >> 2`** in hex? [2 pt]

0x

(C) For **`char m = 0xCD`**, find the *smallest positive integer* `n` (in decimal) such that **`m+n`** causes *unsigned* overflow but NOT *signed* overflow. [2 pt]

For the rest of this problem we are working with a floating point representation that follows the same conventions as IEEE 754 except using 9 bits split into the following fields:

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

(D) What is the *magnitude* of the **bias** of this new representation? [1 pt]

(E) Encode the number $\mathbf{2^{2}} + \mathbf{2^{-1}} + \mathbf{2^{-3}}$ into this floating point scheme (binary). [2 pt]

0b

(F) Let `f1 = 5.0` using this encoding. What is the *smallest positive integer value* of `f2` such that `f1*f2` overflows? [3 pt]

Question 2: Pointers & Memory [10 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 = 0x10
int*  intP  = 0x20
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	CD	FA	1D	D0	41	EE	77
0x20	BA	B0	FF	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. [4 pt]

```
char  v1 = charP[_______];      // set v1 = 0xEE

long* v2 = longP + _______;     // set v2 = 0x68
```

(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.* [6 pt]

```
movb   (%rsi),        %al
leal   2(,%rdi,4),    %ebx
movzwq -3(%rdi,%rsi), %rcx
```

Register	Data (hex)
%rdi	0x 0000 0000 0000 000F
%rsi	0x 0000 0000 0000 0002
%al	0x
%ebx	0x
%rcx	0x

Question 3: Design Questions [6 pts]

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

(A) We have repeatedly stated that Intel is big on legacy and backwards-compatibility. Name one example of this that we have seen in this class. [2 pt]

(B) Name one programming consequence if we decided to assign an address to every 4 bytes of memory (instead of 1 byte). [2 pt]

(C) If we changed the x86-64 architecture to use 24 registers, how might we adjust the *register conventions*? [2 pt]

One thing that should remain the same:
One thing that should change:

Question 4: C & Assembly [12 pts]

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

```
mystery:
        movq    %rdi, %rdx      # Line 1
.L4:    movb    (%rdi), %al     # Line 2
        testb   %al, %al        # Line 3
        je      .L2             # Line 4
        movb    %al, (%rdx)     # Line 5
        cmpb    $32, %al        # Line 6
        je      .L3             # Line 7
        addq    $1, %rdx        # Line 8
.L3:    addq    $1, %rdi        # Line 9
        jmp     .L4             # Line 10
.L2:    movb    %al, (%rdx)     # Line 11
        retq                    # Line 12
```

(A) What **variable type** would `%rdi` be in the corresponding C program? [2 pt]

__________ `rdi`

(B) Give the following labels more intuitive/functional names: [1 pt]

`.L4` ____________ `.L2` ____________

(C) Convert lines 6-8 into C code. Use variable names that correspond to the register names (e.g. `al` for the value in `%al`). [3 pt]

`if ( ____________ ) _______________;`

(D) This function uses a `for` loop. Fill in the corresponding parts below, again using register names as variable names. None should be blank. [4 pt]

`for ( ____________ ; ____________ ; ____________ )`

(E) Describe at a high level what you think this function *accomplishes* (not line-by-line). [2 pt]

Question 5: Procedures & The Stack [11 pts]

The recursive function `count_nz` counts the number of *non-zero* elements in an `int` array. Example: if `int a[] = {-1,0,1,255}`, then `count_nz(a,4)` returns `3`. The function and its x86-64 *disassembly* are shown below:

```
int count_nz(int* ar, int num) {
     if (num>0)
          return !!(*ar) + count_nz(ar+1,num-1);
     return 0;
}
```

```
0000000000400536 <count_nz>:
  400536:  85 f6            testl  %esi,%esi
  400538:  7e 1b            jle    400555 <count_nz+0x1f>
  40053a:  53               pushq  %rbx
  40053b:  8b 1f            movl   (%rdi),%ebx
  40053d:  83 ee 01         subl   $0x1,%esi
  400540:  48 83 c7 04      addq   $0x4,%rdi
  400544:  e8 ed ff ff ff   callq  400536 <count_nz>
  400549:  85 db            testl  %ebx,%ebx
  40054b:  0f 95 c2         setne  %dl
  40054e:  0f b6 d2         movzbl %dl,%edx
  400551:  01 d0            addl   %edx,%eax
  400553:  eb 06            jmp    40055b <count_nz+0x25>
  400555:  b8 00 00 00 00   movl   $0x0,%eax
  40055a:  c3               retq
  40055b:  5b               popq   %rbx
  40055c:  c3               retq
```

(A) How much space (in bytes) does this function take up in our final executable? [1 pt]

(B) The compiler automatically creates labels it needs in assembly code. How many labels are used in `count_nz` (including the procedure itself)? [1 pt]

(C) In terms of the *C function*, what value is being saved on the stack? [1 pt]

(D) What is the return address to `count_nz` that gets stored on the stack (in hex)? [1 pt]

`0x`

(E) Assume `main` calls `count_nz(a,5)` with an appropriately-sized array and then prints the result using `printf`. Starting with (including) `main`, answer the following *in number of stack frames*. [2 pt]

Total created:	Max depth:

(F) Assume `main` calls `count_nz(a,6)` with **`int`** `a[] = {3,5,1,4,1,0}`. We find that the return address to `main` is stored on the stack at address `0x7fffeca3f748`. What data will be stored on the stack at address **`0x7fffeca3f720`**? *You may use the provided stack diagram, but you will be graded primarily on the answer box to the right.* [3 pt]

`0x`

`0x7fffeca3f748`	`<ret addr to main>`
`0x7fffeca3f740`	
`0x7fffeca3f738`	
`0x7fffeca3f730`	
`0x7fffeca3f728`	
`0x7fffeca3f720`	

(G) A similar function `count_z` that counts the number of *zero* elements in an array is made by making a single change to `count_nz`. What is the address of the changed assembly instruction? [2 pt]

`0x`