University of Washington – Computer Science & Engineering

Autumn 2017 Instructor: Justin Hsia 2017-10-30

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 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]

-25 = -32+4+2+1 = $-2^5+2^2+2^1+2^0$ (two's complement)
[sign & mag also accepted: 0b 111001]

0b **100111**

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

`0x79^0xFF=0x86`, `0x86>>2=0xE1` (arithmetic right shift).

0x **E1**

(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]

`UMax` is `0xFF`, so unsigned overflow is 1 past this.
`0xFF - 0xCD + 1 = 0x33 = 3*(16+1) = 51.`
In two's complement, this is -51+51=0, so no overflow.

51

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]

bias = $2^{n-1}-1 = 2^3-1 = 7$.

7

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

0b **0 1001 0010**

$2^2+2^{-1}+2^{-3} = 100.101_2 = 1.00101_2 \times 2^2$, so Exp = 2, Man = 1.00101_2.
S = 0. E = Exp+bias = 9 = 0b1001. M = 0b0010 (last 1 gets rounded off).

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

52

Overflow occurs in floating point when our arithmetic result requires an exponent that can't be represented in the E field. In this case, this corresponds to the encoding E=0b1111, which is not coincidentally reserved for `Inf` and `NaN`.
This corresponds to the value $2^{15-7} = 256$. The first multiple of 5 that exceeds this is $5 \times 52 = 260$.

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[__14___];      // set v1 = 0xEE

long* v2 = longP + ___7___;     // set v2 = 0x68
```

v1: Byte 0xEE is at address 0x1E. 0x1E – charP = 14.

v2: No dereferencing; just pointer arithmetic (scaled by `sizeof(long)=8`).
`longP=0x30`. To get to `0x68`, need to add `0x38` (7 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.* [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 **03**
%ebx	0x **0000 003E**
%rcx	0x **0000 0000 0000 FACE**

`movb` pulls byte from memory at address `0x2`.

`leal` calculates address `0xF × 4 + 2`. Can use left shifting to do multiplication.

`movzwq` instruction pulls two bytes starting at memory address `0xF+0x2-3` = `0xE`, which is `0xFACE` (remember little endian!). Then zero-extend out to 64 bits.

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.

Many different answers were accepted for these questions, including some not listed here.

(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]

- Naming of first 8 registers (`%rax`, etc.) comes from IA32.
- Any 32-bit result stored in a register will zero-out the upper 32-bits (so IA32 programs run correctly on 64-bit machines).
- The "word" instruction suffix in x86-64 (e.g. mov**w**) still refers to 16 bits.
- Use of CISC design philosophy: keeps old instructions in newer instruction sets.

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

- For the same word size, your address space will be 4 times larger now.
- For same address space, addresses could be 2 bits shorter now.
- Difficult to access data for small datatypes *in memory* (alternatively, much more padding needed when storing small datatypes).
- Might not be able to use `b` and `w` assembly instruction suffixes when accessing memory.

(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:

- Only need 1 stack pointer and 1 return value.
- Still have both callee-saved and caller-saved registers.
- Keep the names of the existing 16 registers.

One thing that should change:

- Probably increase the number of argument registers.
- Anything related to defining which of the new registers are callee-saved or caller-saved was given credit.

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]

In Line 2, `%rdi` is dereferenced in a **`movb`** instruction. ___**`char*`**__ `rdi`

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

Other variants accepted. `.L4` ____**`Loop`**____ `.L2` ____**`Exit`**____

(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 ( __`**`al != 32`**`__ ) __`**`rdx = rdx+1`**`__;`

(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 ( __`**`rdx = rdi`**`__ ; _`**`*rdi != 0`**`_ ; _`**`rdi = rdi+1`**`_ )`

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

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

Removes all occurrences of the character 32 (space `' '`) from a string.
Anything mentioning skipping or overwriting the character 32 in a char array received some credit.

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]

Count all bytes (middle columns) or subtract address of next instruction (`0x40055d`) from `0x400536`.

39 B

(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]

The addresses 0x400536, 0x400555 (BaseCase:), 0x40055b (Exit:).

3

SID: ___________

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

`movl` instruction at 0x40053b puts `*ar` into `%rbx`, which is pushed onto the stack by the `pushq` instruction at 0x40053a.

`*ar`

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

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

0x **400549**

(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: **8**	Max depth: **7**

`main→count_nz(a,5)→(a+1,4)→(a+2,3)→(a+3,2)→(a+4,1)→(a+5,0)`
`→printf`

(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 **5**

0x7fffeca3f748	<ret addr to main>
0x7fffeca3f740	**<original rbx>**
0x7fffeca3f738	**0x400549 <ret addr>**
0x7fffeca3f730	**0x3 <*a>**
0x7fffeca3f728	**0x400549 <ret addr>**
0x7fffeca3f720	**0x5 <*(a+1)>**

(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]

Changing the `setne` to a `sete` changes the double bang in the C code to a single bang and counts the zero elements instead.

0x **40054b**