Midterm Exam

601.229 Computer Systems Fundamentals

October 1, 2021

Complete all questions.

Use additional paper if needed.

Time: 50 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: Solution

Print name: ______________________

Date: ______________________

Reference

Powers of 2 ($y = 2^x$):

x	0	1	2	3	4	5	6	7	8	9	10	11	12
y	1	2	4	8	16	32	64	128	256	512	1,024	2,048	4,096

x	13	14	15	16
y	8,192	16,384	32,768	65,536

Note that in all questions concerning C:

- `uint8_t` is an 8-bit unsigned integer type
- `uint16_t` is a 16-bit unsigned integer type
- `uint32_t` is a 32-bit unsigned integer type
- `int8_t` is an 8-bit signed two's complement integer type
- `int16_t` is a 16-bit signed two's complement integer type
- `int32_t` is a 32-bit signed two's complement integer type

x86-64 registers:

Callee-saved: `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`

Caller-saved: `%r10`, `%r11`

Return value: `%rax`

Arguments: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

Note that argument registers and return value register are effectively caller-saved.

Registers and sub-registers:

Register	Low 32 bits	Low 16 bits	Low 8 bits
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbp	%ebp	%bp	%bpl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Stack alignment: `%rsp` must contain an address that is a multiple of 16 when any `call` instruction is executed.

Operand size suffixes: **b** = 1 byte, **w** = 2 bytes, **l** = 4 bytes, **q** = 8 bytes (Examples: `movb`, `movw`, `movl`, `movq`)

Question 1. [10 points] Show the binary (base 2) representation of the following integer values:

- 15 = 8 + 4 + 2 + 1 = 1111
- 225 = 128 + 64 + 32 + 1 = 11100001

Question 2. [10 points] What output is printed by the following C code? Explain briefly.

128
107

21

```
uint8_t a = 197;
uint8_t b = 65;
uint8_t sum = a + b;
printf("%u\n", (unsigned) sum);
```

197
+ 65
---
262 mod 256 = 6

output is 6

Question 3. [10 points] Show the 8-bit two's complement representation of the following integer values:

- 51 = 32 + 16 + 2 + 1 → 00110011
- -107 = -128 + 21 = -128 + 16 + 4 + 1 → 10010101

Question 4. [10 points] What output is printed by the following code? (Hint: `|` means bitwise or.) Explain briefly.

```
int16_t x = 32767;
printf("%d\n", x);
x = x | 0x8000;
printf("%d\n", x);
```

32767 → 0111111 11111111

0x8000: sets sign bit to 1 (adding $-2^{15} = -32768$)

output is -1

Question 5. [10 points] A 32-bit IEEE 754 single precision floating point value has the following representation:

Sign	Exponent	Fraction
1 bit	8 bits	23 bits

Recall that normalized floating point numbers have values $\pm 1.x \times 2^y$, where x is specified by the fraction bits, and y is value of the exponent (which has a value between -126 and 127.)

This format allows all integer values in the range $-q$ to q (inclusive) to be represented exactly.

State the value of $q = 1.x \times 2^y$. First, specify the fraction (x) *in base 2* (i.e., a sequence of 23 0s and 1s):

$x =$ 00000000000000000000000

Next, specify the exponent (y) *in base 10*:

$q = 1.0 \times 2^{24}$

$y =$ 24

Optional: explain briefly.

an exponent of 23 allows the decimal point to be moved to the right past all digits of the fraction. So, all integers with magnitudes 2^{24} and less can be represented exactly.

Question 6. [10 points] What output is printed by the following C program? Assume that `sizeof(int) == 4`. Explain briefly.

```
int a[4] = { 6, 7, 8, 9 };
printf("%d\n", (int) (&a[2] - &a[0]));
printf("%d\n", (int) ((char *)&a[2] - (char *)&a[0]));
```

2

8

Arithmetic on C pointers is in terms of number of array elements, regardless of element size. Because char elements are 1 byte, arithmetic on char pointers is in terms of bytes. Elements 0 and 2 of an array of int elements are 8 bytes apart.

Question 7. [40 points] Consider the following C function prototype:

```
void add_to_vec_if_even(int32_t *vec, unsigned len, int32_t value);
```

This function takes an array of `len` `int32_t` values and adds `value` to each of the *even* values in the array. Its behavior is described by the following unit test:

```
int32_t data[] = { 247, -550, 582, 181 };
add_to_vec_if_even(data, 4, 10);
ASSERT(247 == data[0]);  // original value was odd
ASSERT(-540 == data[1]); // original value was even
ASSERT(592 == data[2]);  // original value was even
ASSERT(181 == data[3]);  // original value was odd
```

Show an x86-64 assembly language implementation of the `add_to_vec_if_even` function. (Continue on next page if necessary.) **Hint**: `andl $1,` *Reg* is a useful way to check whether the 32-bit value in *Reg* is odd.

```
    .globl add_to_vec_if_even:
add_to_vec_if_even:
      subq $8, %rsp                // not really necessary
      leaq (%rdi, %rsi, 4), %r10   // %r10 points one element past
                                   // end of array

.Lloop:
      cmpq %r10, %rdi              // reached end?
      jae .Ldone                   // if so, done

      movl (%rdi), %r11d           // get array element
      andl $1, %r11d               // see if low bit is set
      jne .Ladvance                // if it is, don't modify (b/c value is odd)
      addl %edx, (%rdi)            // add value to array element
 .Ladvance:
      addq $4, %rdi                // advance to next element
      jmp .Lloop                   // continue loop
 .Ldone:
      addq $8, %rsp
      ret
```

[Continue your answer to Question 7 here if necessary.]