Exam 1

601.229 Computer Systems Fundamentals

October 6, 2025

Complete all questions.

Time: 50 minutes.

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

Signed: _____

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. [20 points] Assume that a two-element array of `uint32_t` values can be used to represent a single 64-bit unsigned integer value, with the first element containing the low 32 bits, and the second element containing the high 32 bits. The `add64` function takes pointers to two such arrays, `a` and `b`, computes the 64-bit sum of the values in those arrays, and stores the sum in the array pointed to by the `sum` argument. The function should return 1 if overflow occurred, and 0 if no overflow occurred. If overflow does occur, the elements of `sum` should contain the lowest 64 bits of the correct sum.

Example unit test:

```
uint32_t left[] = { 0xFFFFFFFF, 0 }, right[] = { 1, 0 };
uint32_t out[2];
assert( 0 == add64( left, right, out ) );
assert( 0 == out[0] );
assert( 1 == out[1] );
```

Complete the implementation of the `add64` function as described above. **Important**: your implementation may *not* use any 64-bit operations. E.g., you may not use the `uint64_t` data type. 32-bit operations (e.g., on `uint32_t` values) are allowed.

```
int add64( const uint32_t *a, const uint32_t *b, uint32_t *sum ) {
```

Question 2. [15 points] Complete the following `sub32` function so that it returns the difference `a - b` where `a` and `b` are the `int32_t` arguments. Example unit tests:

```
assert( sub32( 11, 4 ) == 7 );
assert( sub32( 4, 11 ) == -7 );
```

In general, it should be the case that `sub32(x,y) == (x-y)` for all `int32_t` values `x`, `y`.

Important: the body of the function must *not* have any occurrences of the minus character ("-"). Hint: you should assume that `int32_t` values are represented using two's complement. The bitwise complement operator (~) could be useful.

```
int32_t sub32( int32_t a, int32_t b ) {
```

Question 3. [10 points] Convert the value $\boxed{-17.90625}$ to an IEEE 754 single precision floating point value. Show the exact bits used to represent the sign, exponent, and fraction. Recall that

- A normalized floating point value has the form $\pm 1.x \times 2^y$, where x is the fraction and y is the exponent
- The exponent is stored in biased form, as $y + 127$, where y is the true value of the exponent

Hint: note that $0.90625 = 1/2 + 1/4 + 1/8 + 1/32$.

Write your answer here:

Sign (1 bit)	Exponent (8 bits)	Fraction (23 bits)

You can use the rest of the page for scratch work.

Question 4. [10 points] The `is_palindrome` function is intended to check a C character string to determine whether its sequence of characters is the same in both directions. Function prototype and example unit tests:

```
// function prototype
int is_palindrome( const char *s );
```

```
// unit tests
assert( 1 == is_palindrome( "risetovotesir" ) );
assert( 0 == is_palindrome( "foobar" ) );
assert( 1 == is_palindrome( "" ) );
assert( 1 == is_palindrome( "Q" ) );
assert( 0 == is_palindrome( "Qq" ) );
```

Identify *exactly three* distinct errors in the following x86-64 assembly language implementation of this function, and *briefly* explain how to fix each one. ("Distinct" means that each error should be a different kind of error.) It is possible that there are more than three errors, but you should only state three.

```
    .globl is_palindrome
is_palindrome:
    movl %edi, %r12d        /* save ptr to string */
    call strlen             /* determine string length */
    movl %r12d, %r8d        /* copy ptr to string to %r8 */
    addl %eax, %r8d         /* advance by string length */
    decl %r8d               /* subtract 1 (%r8 now points to end) */
    jmp .Lcheck             /* jump to loop condition */
.Lloop:
    movb (%r12d), %cl       /* get "left" character */
    movb (%r8d), %dl        /* get "right" character */
    cmpb %dl, %cl           /* compare characters */
    jne .Lnot_palindrome    /* if different, not a palindrome */
    incl %r12d              /* move left ptr forward */
    decl %r8d               /* move right ptr back */
.Lcheck:
    cmpl %r8d, %r12d        /* right ptr > left ptr? */
    ja .Lloop               /* if so, continue loop */
    movl $1, %eax           /* string is a palindrome */
    jmp .Ldone              /* done */
.Lnot_palindrome:
    movl $0, %eax           /* return 0 */
.Ldone:
    ret
```

Question 5. [20 points] In x86-64 assembly language, complete the implementation of the `make_abs` function below, which has the following function prototype:

```
void make_abs( int32_t *p );
```

If the value in the `int32_t` variable pointed to by the pointer argument `p` is negative, then the function should update the variable to contain the absolute value of the original negative value. Example unit tests are shown on the right.

```
// Example unit tests

int32_t a = 4, b = -17;
make_abs( &a );
assert( 4 == a );
make_abs( &b );
assert( 17 == b );
```

```
    .globl make_abs
make_abs:
```

Question 6. [25 points] In x86-64 assembly language, complete the implementation of the `make_all_abs` function below, which has the following function prototype:

```
void make_all_abs( int32_t *arr, int32_t n );
```

The function should invoke the `make_abs` function (from Question 5) on each of the `n` elements in the array `arr`. An example unit test is shown on the right.

```
// Example unit test

int32_t a[] =
  { 4, -17, -3 };
make_all_abs( a, 3 );
assert( 4 == a[0] );
assert( 17 == a[1] );
assert( 3 == a[2] );
```

```
    .globl make_all_abs
make_all_abs:
```

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]