

**Computer Systems Organization
CSCI-UA.0201 Spring 2022**

Mid-Term Exam

Write all answers on these exam sheets. If you need more space or scratch paper, write on the back of a sheet. Answer the easy questions first!

10 pts

1. True/False. Please circle the correct response. There are no trick questions.
 - a. **T**☒ **F** The term “instruction set architecture” (ISA) refers to how the circuits in the computer are designed.
 - b. **T**☒ **F** Because addition is commutative (that is, $x+y = y+x$), in two’s complement you can negate a number either by flipping the bits and then adding 1 or by adding 1 and then flipping the bits.
 - c. **T**☒ **F** In C, dereferencing a NULL pointer generally causes a program to crash because NULL is 0, and the hardware and operating system will not allow a program to access the memory location at address 0.
 - d. **T**☒ **F** In C, the expression “**p->val**” is not equivalent to “**(*p).val**”.
 - e. **T**☒ **F** $\log(a \times b) = \log a + \log b$
 - f. **T**☒ **F** In 64-bit signed (two’s complement) number representation, there are 2^{63} possible negative numbers and 2^{63} possible non-negative numbers.
 - g. **T**☒ **F** The code “**if (1 & 2) printf("Yes"); else printf("No");**” will print “No”.
 - h. **T**☒ **F** In an n-bit two’s complement number, the value of the number is the usual non-negative interpretation of the rightmost n-1 bits (i.e. bits 0 through n-2), plus the value of the leftmost bit times $-(2^{n-1})$.
 - i. **T**☒ **F** In x86-64 assembly, the instructions “**add (%rcx), %rax**” and “**add 0(%rcx), %rax**” mean different things.
 - j. **T**☒ **F** In C, the code “**int x = -10; x = x >> 32;**” will cause **x** to be zero, since all the original bits of **x** will have been shifted out of **x**.

10 pts
(2 pts each)

2. For each code snippet, below, indicate what will be printed as the result of executing the code. If the code won’t compile, indicate so.
 - a.

```
int x = 10;
int *p = &x;
(*p)--;
printf("%d\n", x);
```

Answer: 9

- b. `int c[5] = {0,2,4,6,8};`
`int sum = 0;`
`for (int i = 0; i < 5; i++){`
`sum += *c;`
`c++;`
`}`
`printf("%d\n", sum);` Answer: Won't compile (c is a constant)
- c. `int x = -3;`
`printf("%x\n", x); // HEX!` Answer: FFFFFFFFD
- d. `#define MASK 0xF`
`int x = 0xABCDEF;`
`printf("%d\n", (x >> 16) & MASK);` Answer: 11
- e. `int x = 0x3C;`
`int y = 0x66;`
`x = x & ~y;`
`printf("%x\n", x); // HEX!` Answer: 18

5 pts

3. Multiply the following two binary numbers (without converting to decimal), and then show the result in binary and in hex. Show all work and write neatly.

$$\begin{array}{r}
 1110 \\
 \times 1100 \\
 \hline
 0000 \\
 11100 \\
 111000 \\
 \hline
 10101000
 \end{array}$$

Result in binary: 10101000Result in hex: A8

10 pts

4. Given the following type declaration,

```
typedef struct cell {
    int val;
    struct cell *next;
} CELL;
```

write in C a function whose prototype is

```
int strictly_decreasing(CELL *head);
```

where **head** points to a linked list. The function should return true if the **val** fields of the elements of the linked list are in strictly decreasing order and should return false otherwise. If the linked list is empty or only has one element, the function should return true. You do not need to create the linked list, of course.

```
#define true 1 // you didn't have to do this
#define false 0
```

```
int strictly_decreasing(CELL *head)
{
    if ((head == NULL) || (head->next == NULL))
        return true;

    // there are other ways to do this, of course.
    int last_val = head->val;
    for(CELL *p = head->next; p != NULL; p = p->next) {
        if (p->val >= last_val)
            return false;
        last_val = p->val;
    }
    return true;
}
```

or, even easier (using recursion):

```
int strictly_decreasing(CELL *head)
{
    if ((head == NULL) || (head->next == NULL))
        return true;
    if (head->val > head->next->val)
        return strictly_decreasing(head->next);
    else
        return false;
}
```

10 pts

5. Fill in the missing X86-64 assembly code, below, for a function `count_not_n()` that takes three parameters – an integer array (in `%rdi`), the size of the array (in `%esi`), and an integer value `n` (in `%edx`) – and returns (in `%eax`) the count of the number of elements in the array that are not equal to `n`. Assume that you are free to overwrite any registers you want. Important: The last sheet of this exam is a brief assembly language reference sheet.

`_count_not_n:`

```

    pushq    %rbp

    movq     %rsp, %rbp

    movq     $0,%rcx                # i = 0 (use a 64-bit register)
    movl     $0,%eax,%eax          # count = 0 (result, put in %eax)
TOP:
    cmpl     %esi,%ecx              # compare i to size
    jge      OUT                   # if i >= size, jump out of loop

    cmpl     %edx, (%rdi,%rcx,4)    # if a[i]= n, don't increment
                                     # count, otherwise increment count.
    je       NEXT
    incl     %eax

NEXT:
    incq     %rcx                  # i++
    jmp      TOP                   # jump back to top of loop
OUT:
    # result is already in %eax

    pop     %rbp
    ret

```

10 pts
(5 pts each)

6. 32-bit IEEE floating numbers have one sign bit, 8 exponent bits (with a 127 bias), and 23 mantissa bits.

- a. Fill in the following blank to compute the actual mantissa (not the stored mantissa) of the floating point number in the **float** variable **f**, using the unsigned integer variable **x**. Assume that the “point” is to the left of bit 22 of the result.

```
unsigned int x = *((unsigned int*) &f);    // assume f is a float
unsigned int actual_mantissa = (x & 0x7FFFFFFF) | (1 << 23);
```

- b. Fill in the sign, exponent, and mantissa bits in the IEEE floating point representation of the number **26.5** (be sure to show the stored bits, not the actual exponent and mantissa values).

sign bit = 0

exponent bits = 10000011

mantissa bits = 10101000000000000000000

Show your work here.

$$26.5 = 16 + 8 + 2 + (1/2) = 11010.1 \times 2^0 = 1.10101 \times 2^4$$

Since the number is positive, the sign bit is 0.

The stored exponent = $4 + 127 = 131 = 128 + 2 + 1 = 10000011$

The 23-bit stored mantissa (without the 1 before the point) is 10101 followed by 18 zeros.

x86-64 Assembly Reference Sheet

Registers:

- **64 bit:** %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15
- **32 bit (half of 64-bit register):** %eax, %ebx, %ecx, %edx, %esi, %edi, %r8d, %r9d, %r10d, %r11d, %r12d, %r13d, %r14d, %r15d

Instructions:

Note: A “q” suffix indicates a 64-bit instruction, an “l” suffix indicates a 32-bit instruction. These are generally optional unless the assembler can’t tell the type of the instruction.

- **Add:** add
- **Subtract:** sub
- **Multiply:** imul
- **Increment:** inc
- **Decrement:** dec
- **Jump:** jmp
- **Comparison:** cmp
- **Conditional Jump:** jle, jl, je, jne, jg, jge

Addressing Modes:

These are just examples.

- **Immediate:** \$23
- **Register:** %rbx
- **Pointer:** (%rsi)
- **Pointer+offset:** 12(%rdx)
- **Indexed:** (%rcx,%rbx,8)
- **Indexed+Offset:** 16(%rax,%r10,4)