Name SOLUTION

Computer Systems Organization
CSCI-UA.0201 Spring 2022

Mid-Term Exam

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

10pts
(1 pt each)

1. True/False. Please circle the correct response. There are <u>no</u> trick questions.

   a. 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.

   b. [✓ 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.

   c. [✓ T] F In C, the expression "`p->val`" is equivalent to "`(*p).val`".

   d. T [✓ F] log (a+b) = log a × log b

   e. T [✓ F] In 64-bit signed (two's complement) number representation, there are 2^{32} possible negative numbers and 2^{32} possible non-negative numbers.

   f. [✓ T] F The code "`if (1 & 2) printf("Yes"); else printf("No");`" will print "No".

   g. [✓ 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 $-(2^{n-1})$.

   h. [✓ T] F In x86-64 assembly, the instructions "`add  %rcx,%rax`" and "`add (%rcx),%rax`" mean different things.

   i. T [✓ F] In C, the code "`int x = -1; x = x >> 32;`" will cause **x** to be zero, since all the original bits of **x** will have been shifted out of **x**.

   j. [✓ T] F The term "instruction set architecture" (ISA) refers to how the hardware looks to an assembly language programmer, including the registers, instructions, etc., rather than how the circuits are designed.

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 = 30;
int *p = &x;
(*p)++;
printf("%d\n", x);
```
   Answer: 31

b.
```
int a[5] = {0,1,2,3,4};
int sum = 0;
for (int i = 0; i < 5; i++){
     sum += *a;
     a++;
}
printf("%d\n", sum);
```
Answer: Won't compile ("a" is a constant)

c.
```
int x = -4;
printf("%x\n", x);  // HEX!
```
Answer: FFFFFFFC

d.
```
#define MASK 0xF
int x = 0xABCDEF;
printf("%d\n",(x >> 8) & MASK);
```
Answer: 13

e.
```
int x = 0x66;
int y = 0x3C;
x = x & ~y;
printf("%x\n", x); // HEX!
```
Answer: 42

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.

```
      1010
  ×   1110
         0
     1010
    1010
   1010
  --------
  10001100
```

Result in binary: 10001100

Result in hex: 8C

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_increasing(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 increasing 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_increasing(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_increasing(CELL *head)
{
  if ((head == NULL) || (head->next == NULL))
      return true;
  if (head->val < head->next->val)
      return strictly_increasing(head->next);
  else
      return false;
}
```

10 pts

5. Fill in the missing X86-64 assembly code, below, for a function **`count_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 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_n:

        pushq   %rbp

        movq    %rsp, %rbp

        movq    $0,%rcx              # i = 0 (use a 64-bit register)

        movl    $0        ,%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.

        jne     NEXT

        incl    %eax

NEXT  :

        incq    %rcx                 # i++

        jmp     TOP                  # jump back to top of loop

OUT:  :

        # result is already in %eax

        pop     %rbp

        ret
```

The "q" and "l" suffixes aren't necessary in any of the above instructions. However, they are helpful in catching errors, in case, for example, you intended to use 64-bit registers but accidentally used 32-bit registers.

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 exponent (not the stored exponent) of the floating point number in the **float** variable **f**, using the unsigned integer variable **x**.

```
unsigned int x = *((unsigned int*) &f);   // assume f is a float

int actual_exponent = ((x >> 23) & 0xFF) - 127__________;
```

   b. Fill in the sign, exponent, and mantissa bits in the IEEE floating point representation of the number **-12.25** (be sure to show the stored bits, not the actual exponent and mantissa values).

      sign bit = 1

      exponent bits = 10000010

      mantissa bits = 10001000000000000000000

   Show your work here.

$-12.25 = -(8 + 4 + (1/4)) = -1100.01 \times 2^0 = -1.10001 \times 2^3$

Since it is negative, the sign bit is 1.

The stored 8-bit exponent = 3 + 127 = 130 = 128 + 2 = 10000010

The stored 23-bit mantissa (without the 1 before the point) = 10001 followed by 18 zeros (total of 23 bits)

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)`