

# Computer Organization

Fall 2003

Test #1

Name \_\_\_\_\_

There are 6 pages - make sure you have all of them.  
 Answer all questions - pay attention to the # of points for each question.  
 Don't leave anything blank - partial credit is always possible!

**Question 1a (10 pts):** Complete the following tables by filling in each blank space with the appropriate value.

Decimal	8 bit 2's complement binary	Hex
1	0000 0001	01
-63	1100 0001	C1
109	0110 1101	6D
-125	1000 0011	83

Decimal	8 bit unsigned binary	Hex
138	1000 1010	8A
126	0111 1110	7E

**Question 1b (5 points):** Determine the decimal value of the following IEEE 32 bit floating point number:

IEEE Single Precision (32 bit)		
Sign (1)	Exponent (8)	Significand (23)
0	1000 0010	1111 1000 0000 0000 0000 000

exponent is  $130-127=3$

number is  $1.11111 \times 2^3$  which is  $1111.11_2$  which is  $15.75_{10}$

Number is 15.75

**Question 3 (20 pts):** We are using a computer with the following characteristics:

- The C type `int` is represented using 32 bits two's complement representation.
- The C type `char` is signed, 8 bit two's complement.
- All signed integers are right shifted arithmetically.

We have the following code to declare and initialize some variables:

```
int x = random();           /* x can have any value */
int y = random();           /* y can have any value */
unsigned int ux = (unsigned) x;
char c = (char) x;
unsigned char uc = (unsigned char) c;
```

For each of the C expressions listed below, determine whether the expression is *always* true, *always* false, or if the value can't be determined without knowing the values of `x`, `y`, `ux`, `c`, `uc`.

<i>expression</i>	<i>value of expression is</i>		
<code>(x &lt; y) == (ux &lt; uy)</code>	Always True	Always False	Can't Tell
<code>(x &amp; 0x80000000) == (ux &amp; 0x80000000)</code>	Always True	Always False	Can't Tell
<code>(x &gt;&gt; 8) == (ux &gt;&gt; 8)</code>	Always True	Always False	Can't Tell
<code>(x &amp; ux)</code>	Always True	Always False	Can't Tell
<code>(x &amp; -x)</code>	Always True	Always False	Can't Tell
<code>(x &amp; ~x)</code>	Always True	Always False	Can't Tell
<code>(( (x &gt;&gt; 1) &lt;&lt; 1) == x)</code>	Always True	Always False	Can't Tell
<code>(-x == (~x + 1))</code>	Always True	Always False	Can't Tell
<code>((x    c) == (ux    uc))</code>	Always True	Always False	Can't Tell
<code>((uc &gt;&gt; 4) &amp; (~uc &lt;&lt; 4))</code>	Always True	Always False	Can't Tell

**Question 3 ( 20 pts):** Write a C function named `countbits()` with the following prototype:

```
int countbits(char x);
```

The function should return the number of bits in `x` that have the value 1 (so the return value should be between 0 and 8 inclusive).

You should assume that the C data type `char` is signed (8 bit 2's complement).

```
int countbits(char x) {
    int tot=0;
    int i;
    for (i=0;i<8;i++) {
        if (x & 0x01) tot++;
        x = x >> 1;
    }
    return(tot);
}
```

**Question 4 (20pts):** The C function named `compute()` (shown below) is compiled using `gcc` and part of the resulting IA32 assembly language code generated is shown below. Your job is to complete the assembly language code - provide instructions that complete the assembly code for the function `compute()`. Comment each line!

Notes:

- the value of parameter `x` is at 8 (`%ebp`) and the value of `y` is at 12 (`%ebp`)
- the value to be returned should be in register `%eax` when the function completes.

```
int compute(int x, int y)
{
    int z;
    if (x>y) {
        z = x - y;
    } else {
        z = y - x;
    }
    return(z);
}
```

```
compute:
    pushl %ebp
    movl  %esp, %ebp
    movl  12(%ebp), %edx
    movl  8(%ebp), %eax
    cmpl  %edx, %eax          # CC <= x - y
    jle  .L2                  # jump if x <= y
    subl  %edx, %eax          # z = x - y
    jmp  .L3

.L2:
    subl  %eax, %edx          # z = y - x
    movl  %edx, %eax          # eax is z

.L3:

    leave
    ret
```

**Question 5 (10 pts):** We want to create a (small) instruction set for a robot. The individual instructions that the robot needs are as follows:

- move forward one or two meters (2 different instructions!).
- turn to the right 90°
- turn to the left 90°
- sing a song
- say "Hello"
- bark like a dog.
- go back 1, 2, or 3 instructions (three different instructions!)  
(back 1 would go to the previous instruction)

Your job is to develop a 4-bit binary *machine code* that encodes these instructions (each instruction must be 4 bits) and to then write a machine code program that instructs the robot to do the following (in the order shown):

- move 3 meters forward and 2 meters to the right.
- bark like a dog 2 times.
- repeat the following forever:
  - sing a song.
  - move forward one meter
  - turn left

Show the machine language for each of the possible instructions, then show the machine code for the program. Use the back of this page if you need more room.

One possible machine code:

```
000x: move forward
      x=0 means 1 meter, x=1 means 2 meters
001d: turn.
      d=0 means turn left, d=1 means turn right
0100: sing
0101: bark
0110: say Hello
10yy: go back yy instructions
      (yy=00 is nop, yy=01 is one instruction, etc.)
```

PROGRAM

```
0001 forward 2 meters
0000 forward 1 meter
0011 turn right
0001 forward 2 meters
0101 bark
0101 bark
0100 sing
0000 forward 1 meter
0010 turn left
1011 back 3 instructions
```

**Question 6 (15 pts):** Consider the computation expressed as a C assignment statement shown below:

$$x = 3*y + (2*z - 12);$$

**Part a (5pts):** Write IA32 assembly language for the computation (). You should assume that the value of  $y$  is in register  $\%edx$ , the value of  $z$  is in register  $\%ecx$  and we want the result to be placed in register  $\%eax$  at the end of the computation. NOTE: Feel free to use the `imull` instruction (which we have not discussed), but you don't need it for this computation.

Here is one solution:

```
leal    (%edx,%edx,2), %edx    # %edx = 3*y
leal    -12(%edx, %ecx,2), %eax # %eax=3*y+2*z-12
```

**Part b (5pts):** Show how this computation might look when using a processor based on a *stack-based instruction set architecture* (generic assembly is fine, with instructions like "push  $y$ ", "add" and "pop tmp").

Here is one solution:

```
push 3
push y
mult
push 2
push z
mult
push -12
add
add
pop x
```

**Part c (5pts):** Show how the computation might look when using a processor based on an *accumulator based instruction set*. Generic assembly is fine here also, instructions like "add  $y$ ", "load  $z$ " and "store  $x$ " are expected.

Here is one solution:

```
load y
mult 3
store tmp
load x
mult 2
subtract 12
add tmp
store x
```