351 Section 5 - Midterm Review

1. Number Representation – Integers (13 Autumn)

a) Explain why we have a Carry-Flag and an Overflow-Flag in x86 condition codes.
What is the difference between the two? (Explain in at most two sentences.)

b) Add **11011001** and **01100011** as two's complement 8-bit integers & convert the result to decimal notation.

c) Convert your answer from the previous problem to a 2-digit hex value.

2. Floating-Point Number Representation (based on 12 Spring)

A new pizzeria has opened on the Ave. It is mysteriously called "Pizza 0x40490FDB". Given that you are in CSE351, you have a hunch what the mystery might be. Consider the string of hex digits as a 32-bit IEEE floating point number (8-bit exponent and 23-bit fraction).

a) Fill in the hexadecimal digits in the bytes below and then translate them to individual bits.

8 hex digits in 4 bytes: **40 49 0F DB**

32 bits: __

b) Is this number positive or negative?

c) What is the exponent?
(exponents are biased in this representation so make sure to make this adjustment)

d) What is the significand in binary?
(only use the first 7 bits of the fraction, ignore the lower-order 16 bits)

e) What is the value of the number in binary?

f) What is the decimal number represented?
(only show two decimal digits after the decimal point)

g) What is the pizzeria's mystery name?

3. Arrays – C to Assembly (based on 14 Autumn)

Given the following C function:

```
long sum_pair(long *z, long index) {
    return z[index] + z[index + 1];
}
```

a) Write **x86-64** assembly code for this function here. You can assume that **z** points to an array of 16 elements, and 0 <= **index** < 15.
Comments are not required but could help for partial credit.

4. Assembly and C (15 Winter)

Consider the following x86-64 assembly and C code:

```
<do_something>:
      xor   %rax,%rax
      cmp   $0x0,%rsi
      _____ <end>
      sub   $0x1,%rsi

<loop>:
      lea   (%rdi,%rsi, _____),%rdx
      add   (%rdx),%ax
      sub   $0x1,%rsi
      jns   <loop>

<end>:
      ret

short do_something(short* a, int len) {
      short result = 0;
      for (int i = _____; i >= 0; _____) {
            ____________________ ;
      }
      return result;
}
```

a) Both code segments are implementations of the unknown function do_something. Fill in the missing blanks in both versions. (Hint: **%rax** and **%rdi** are used for **result** and **a** respectively. **%rsi** is used for both **len** and **i**)

b) Briefly describe the value that do something returns and how it is computed. Use only variable names from the C version in your answer.

5. Stack Discipline (14 Spring)

The following function recursively computes the greatest common divisor of the integers a, b:

```
int gcd(int a, int b) {
      if (b == 0) {
            return a;
      } else {
            return gcd(b, a % b);
      }
}
```

Here is the x86_64 assembly for the same function:

```
4006c6 <gcd>:
4006c6:     sub         $0x18, %rsp
4006ca:     mov         %edi, 0x10(%rsp)
4006ce:     mov         %esi, 0x08(%rsp)
4006d2:     cmpl        $0x0, %esi
4006d7:     jne         4006df <gcd+0x19>
4006d9:     mov         0x10(%rsp), %eax
4006dd:     jmp         4006f5 <gcd+0x2f>
4006df:     mov         0x10(%rsp), %eax
4006e3:     cltd
4006e4:     idivl       0x08(%rsp)
4006e8:     mov         0x08(%rsp), %eax
4006ec:     mov         %edx, %esi
4006ee:     mov         %eax, %edi
4006f0:     callq       4006c6 <gcd>
4006f5:     add         $0x18, %rsp
4006f9:     retq
```

Note: **`cltd`** is an instruction that sign extends **`%eax`** into **`%edx`** to form the 64-bit signed value represented by the concatenation of [**`%edx | %eax`**].

Note: **`idivl <mem>`** is an instruction divides the 64-bit value [**`%edx | %eax`**] by the long stored at **`<mem>`**, storing the quotient in **`%eax`** and the remainder in **`%edx`**.

a) Suppose we call gcd(144, 64) from another function (i.e. main()), and set a breakpoint just before the statement “return a”. When the program hits that breakpoint, what will the stack look like, starting at the top of the stack and going all the way down to the saved instruction address in main()? Label all return addresses as "ret addr", label local variables, and leave all unused space blank.

Memory address on stack line)	**Value (8 bytes per line)**
0x7fffffffffffad0	Return address back to main
0x7fffffffffffac8	
0x7fffffffffffac0	
0x7fffffffffffab8	
0x7fffffffffffab0	
0x7fffffffffffaa8	
0x7fffffffffffaa0	
0x7fffffffffffa98	
0x7fffffffffffa90	
0x7fffffffffffa88	
0x7fffffffffffa80	
0x7fffffffffffa78	
0x7fffffffffffa70	

<-**%rsp** points here at start of procedure

b) How many total bytes of local stack space are created in each frame (in decimal)?

c) When the function begins, where are the arguments (a, b) stored?

d) From a memory-usage perspective, why are iterative algorithms generally preferred over recursive algorithms?