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.)

The carry flag is used for unsigned numbers and indicates a carry-out of 1 during addition from the most-significant-bit. The overflow flag applies to signed arithmetic and indicates that the addition yielded a number that was too large a positive or too small a negative value.

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

```
     11011001  =     -39
+    01100011  = +   +99
     --------
     00111100  =     +60
```

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

60 = 0x3c

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: **01000000 01001001 00001111 11011011**

b) Is this number positive or negative?

Positive

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

1000 0000   128-Bias = 128-127 = 1

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

frac = 1001001...011   So significand = 1.1001001

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

$1.1001001 * 2^1$=11.001001 (shifted the binary point left by 1)

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

11.001001 = 2 + 1 + .125 + .015625 = 3.14...

g) What is the pizzeria's mystery name?

Pizza Pi

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];
}
```

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.

```
sum_pair:
      movq  (%rdi,%rsi,8), %rax
      addq  8(%rdi,%rsi,8), %rax
      ret
```

4. Assembly and C (15 Winter)

Consider the following x86-64 assembly and C code:

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

<loop>:
      lea   (%rdi,%rsi, 2 ),%rdx
      add   (%rdx),%ax
      sub   $0x1,%rsi
      jns   <loop>

<end>:
      ret
```

```
short do_something(short* a, int len) {
      short result = 0;
      for (int i = len - 1 ; i >= 0; i-- ) {
            result += a[i] ;
      }
      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.

> *do_something returns the sum of the shorts pointed to by a. It does so by traversing the array backwards.*

5. Stack Discipline (14 Spring)

a)

Memory address on stack line)	Value (8 bytes per line)	
`0x7fffffffffffad0`	Return address back to main	<-**`%rsp`** points here at start of procedure
`0x7fffffffffffac8`	*1st of 3 local variables on stack (argument a= 144)*	
`0x7fffffffffffac0`	*2nd of 3 local variables on stack (argument b = 64)*	
`0x7fffffffffffab8`	*3rd of 3 local variables on stack (unused)*	
`0x7fffffffffffab0`	*Return address back to gcd(144, 64)*	
`0x7fffffffffffaa8`	*1st of 3 local variables on stack (argument a = 64)*	
`0x7fffffffffffaa0`	*2nd of 3 local variables on stack (argument b = 16)*	
`0x7fffffffffffa98`	*3rd of 3 local variables on stack (unused)*	
`0x7fffffffffffa90`	*Return address back to gcd(64,16)*	
`0x7fffffffffffa88`	*1st of 3 local variables on stack (argument a = 16)*	
`0x7fffffffffffa80`	*2nd of 3 local variables on stack (argument b = 0)*	
`0x7fffffffffffa78`	*3rd of 3 local variables on stack (unused)*	*<-%rsp at "return a" in 3rd recursive call*
`0x7fffffffffffa70`		

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

32 (24 allocated explicitly and 8 for the return address.)

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

They are stored in the registers %rdi and %rsi, respectively.

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

Recursive algorithm continue to grow the stack for the maximum number of recursions which may be hard to estimate.