5. ***Bit Off More Than You Can Chew? (10 points):*** Consider the code fragment below:

```
union {
  int x;
  unsigned int u;
  float f;
  char s[4];
} testout;

testout.x=0x40000000;
```

What would be printed for each of the following statements:

a. `printf("%d", testout.x);`

2^{30}

b. `printf("%u", testout.u);`

2^{30}

c. `printf("%f", testout.f);`

2.000

d. `printf("%c %c %c %c", testout.s[3], testout.s[2], testout.s[1], testout.s[0]);`

"@ "

How many bytes would `testout` occupy in memory?:

e. # of bytes: 4

6. ***Let Me EAX Another Question (15 points)***: Consider the following array reference:

```
hash_table[(index&255)^((index>>8)&255)];
```

We will implement this reference in an assembly code fragment. Assume that we want to store the value of this reference in register *%eax*. The code fragment will be run on a 32-bit little-endian machine. The assembly code fragment is below – with some blanks left for you to fill in.

```
8048368:    0f b6 55 f8             movzbl  _____-8_____(%ebp),%edx
804836c:    8b 45 f8                mov     _____-8_____(%ebp),%eax
804836f:    c1 f8 08                sar     ____$0x8____,%eax
8048372:    25 ff 00 00 00          and     ____$0xff___,%eax
8048377:    31 d0                   xor     _____%edx___,%eax
8048379:    8b 84 85 f8 fb ff ff    mov     ___-0x408___(%ebp,%eax,4),%eax
                                            or -1032 or 0xfffffbf8
```

To help you fill in the blanks – here's some interaction with gdb to get some key values you will need. This interaction takes place immediately before the assembly fragment above is executed. The following interaction takes place before the code is executed:

```
(gdb) print $esp
$3 = (void *) 0xffffd4b4
(gdb) print $ebp
$4 = (void *) 0xffffd8c8
(gdb) print &hash_table
$5 = (int (*)[256]) 0xffffd4c0
(gdb) print &index
$6 = (int *) 0xffffd8c0
```

8. ***I Cannot Function in this Environment (15 points):*** The following two procedure fragments are part of a program compiled on an x86-64 architecture.

```
int func2(int x, long y, short z          )/*same arg list as func1*/
{
  return  x + y - z                       ;
}

int func1(int x, long y, short z          )/*same arg list as func2*/
{
  x*=     256                             ;
  y*=     18                              ;
  z*=     2                               ;
  return func2(x,y,z);
}
```

Clearly some of the code is missing – your job is to fill in the blanks. Note that the blanks may be larger than necessary. The procedure *func1* is called by some other procedure using *callq*. These procedures will be compiled to the following assembly code:

```
00000000004004a0 <func2>:
  4004a0:       8d 04 37                lea    (%rdi,%rsi,1),%eax
  4004a3:       0f bf d2                movswl %dx,%edx
  4004a6:       29 d0                   sub    %edx,%eax
  4004a8:       c3                      retq

00000000004004b0 <func1>:
  4004b0:       0f bf d2                movswl %dx,%edx
  4004b3:       48 8d 34 f6             lea    (%rsi,%rsi,8),%rsi
  4004b7:       c1 e7 08                shl    $0x8,%edi
  4004ba:       01 d2                   add    %edx,%edx
  4004bc:       0f bf d2                movswl %dx,%edx
  4004bf:       48 01 f6                add    %rsi,%rsi
  4004c2:       e9 d9 ff ff ff          jmpq   4004a0 <func2>
```