

1. **Lost at C? (12 points):** The following problem assumes the following declarations:

```
int x = rand();
float f = foo(); // f is not NaN
unsigned ux = rand();
```

For the following C expressions, circle either Y or N (but not both). If you circle the right answer, you get +2 points. If you circle the wrong answer, you get -1 point. If you do not circle anything, you get 0 points. So do not just guess wildly.

	Always True?	
a. $x > 0 \Rightarrow ((x < 4) >> 5) > 0$	Y	N
b. $f > 0 \Rightarrow ((f < 4) >> 5) > 0$	Y	N
c. $(x >> 20) == (\sim(x >> 20) + 1) \Rightarrow x == (\text{int})(\text{float}) x$	Y	N
d. $x \leq 0, f \leq 0 \Rightarrow x * f \leq 0$	Y	N
e. $x > ux \Rightarrow (\sim x + 1) < 0$	Y	N
f. $ux - 2 \geq -2 \Rightarrow ux \leq 1$	Y	N

Note that “ \Rightarrow ” represents an *implication*. $A \Rightarrow B$ means that you assume A is true, and your answer should indicate whether B should be implied by A – i.e. given that A is true, is B always true?

2. **Boole's Foolery (8 points):** Consider the following sequence of operations on integers x and y:

$$\begin{aligned}x &= x \wedge (\sim y); \\ y &= y \wedge x;\end{aligned}$$

Which of the following expresses the value of y after this?

- A) x
- B) $\sim x$
- C) $\neg x$
- D) y
- E) $\sim y$
- F) $\neg y$

Your answer should be a single character (A-F).

3. **Lucky Number Seven (7 points):** Consider the following expression:

$\sim((\sim(7 < 7) + 7) \wedge \sim 7) >> 7$

What decimal integer value would this evaluate to? Simplify as much as possible.

4. **Structured Play (8 points):** Consider the following structure definition:

```
struct S76 {  
    unsigned int ID;  
    char name[20];  
    short zip;  
    long misc;  
} my_data[10];
```

How many bytes would my_data consume in memory on a:

a. IA32 Linux machine?

b. x86-64 Linux machine?

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);`

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

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

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: _____

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 _____, %edx
804836c:    8b 45 f8             mov     _____, %eax
804836f:    c1 f8 08             sar     _____, %eax
8048372:    25 ff 00 00 00       and     _____, %eax
8048377:    31 d0               xor     _____, %eax
8048379:    8b 84 85 f8 fb ff ff mov     _____, %eax, 4, %eax
```

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
```

7. **Magic 8 Ball says “Success is not likely” (10 points):** You are debugging an application in execution using gdb on a 32-bit (i.e. pointers use 32 bits), little-endian architecture. The application has a variable called *magic8ball* - defined as

```
char magic8ball[8][8][8];
```

Using gdb you find the following information at a particular stage in the application:

```
(gdb) p &magic8ball
$1 = (char (*)[8][8][8]) 0xffffd448
```

And:

```
(gdb) x/256x 0xffffd428
0xffffd428: 0x6279614d 0x00a50065 0x6e6f7257 0x08040067
0xffffd438: 0x65727553 0x00000000 0x656b694c 0x0000796c
0xffffd448: 0x6576654e 0x00000072 0x656b694c 0x0000796c
0xffffd458: 0x0068614e 0x00000000 0x00006f4e 0x00000000
0xffffd468: 0x00736559 0x00000000 0x0068614e 0x00000000
0xffffd478: 0x6279614d 0x00a50065 0x6e6f7257 0x08040067
0xffffd488: 0x6279614d 0x00a50065 0x6576654e 0x00000072
0xffffd498: 0x68676952 0x08040074 0x6e6f7257 0x08040067
0xffffd4a8: 0x6576654e 0x00000072 0x6279614d 0x00a50065
0xffffd4b8: 0x00006f4e 0x00000000 0x68616559 0x00000000
0xffffd4c8: 0x656b694c 0x0000796c 0x0068614e 0x00000000
0xffffd4d8: 0x0068614e 0x00000000 0x00736559 0x00000000
0xffffd4e8: 0x656b694c 0x0000796c 0x68616559 0x00000000
0xffffd4f8: 0x0068614e 0x00000000 0x68616559 0x00000000
0xffffd508: 0x6279614d 0x00a50065 0x68616559 0x00000000
0xffffd518: 0x6576654e 0x00000072 0x6e6f7257 0x08040067
0xffffd528: 0x6e6f7257 0x08040067 0x00006f4e 0x00000000
0xffffd538: 0x6279614d 0x00a50065 0x6e6f7257 0x08040067
0xffffd548: 0x0068614e 0x00000000 0x68676952 0x08040074
0xffffd558: 0x65727553 0x00000000 0x00006f4e 0x00000000
0xffffd568: 0x68616559 0x00000000 0x0068614e 0x00000000
0xffffd578: 0x0068614e 0x00000000 0x68676952 0x08040074
0xffffd588: 0x00736559 0x00000000 0x68616559 0x00000000
0xffffd598: 0x00006f4e 0x00000000 0x68616559 0x00000000
0xffffd5a8: 0x68616559 0x00000000 0x656b694c 0x0000796c
0xffffd5b8: 0x68676952 0x08040074 0x00006f4e 0x00000000
0xffffd5c8: 0x6576654e 0x00000072 0x6e6f7257 0x08040067
0xffffd5d8: 0x00736559 0x00000000 0x6576654e 0x00000072
0xffffd5e8: 0x0068614e 0x00000000 0x656b694c 0x0000796c
0xffffd5f8: 0x65727553 0x00000000 0x00736559 0x00000000
0xffffd608: 0x65727553 0x00000000 0x65727553 0x00000000
0xffffd618: 0x6576654e 0x00000072 0x656b694c 0x0000796c
0xffffd628: 0x6279614d 0x00a50065 0x6e6f7257 0x08040067
0xffffd638: 0x65727553 0x00000000 0x656b694c 0x0000796c
0xffffd648: 0x6576654e 0x00000072 0x656b694c 0x0000796c
0xffffd658: 0x0068614e 0x00000000 0x00006f4e 0x00000000
0xffffd668: 0x00736559 0x00000000 0x0068614e 0x00000000
0xffffd678: 0x6279614d 0x00a50065 0x6e6f7257 0x08040067
```

0xffffd688:	0x6279614d	0x00a50065	0x6576654e	0x00000072
0xffffd698:	0x68676952	0x08040074	0x6e6f7257	0x08040067
0xffffd6a8:	0x6576654e	0x00000072	0x6279614d	0x00a50065
0xffffd6b8:	0x00006f4e	0x00000000	0x68616559	0x00000000
0xffffd6c8:	0x656b694c	0x0000796c	0x0068614e	0x00000000
0xffffd6d8:	0x0068614e	0x00000000	0x00736559	0x00000000
0xffffd6e8:	0x656b694c	0x0000796c	0x68616559	0x00000000
0xffffd6f8:	0x0068614e	0x00000000	0x68616559	0x00000000
0xffffd708:	0x6279614d	0x00a50065	0x68616559	0x00000000
0xffffd718:	0x6576654e	0x00000072	0x6e6f7257	0x08040067
0xffffd728:	0x6e6f7257	0x08040067	0x00006f4e	0x00000000
0xffffd738:	0x6279614d	0x00a50065	0x6e6f7257	0x08040067
0xffffd748:	0x0068614e	0x00000000	0x68676952	0x08040074
0xffffd758:	0x65727553	0x00000000	0x00006f4e	0x00000000
0xffffd768:	0x68616559	0x00000000	0x0068614e	0x00000000
0xffffd778:	0x0068614e	0x00000000	0x68676952	0x08040074
0xffffd788:	0x00736559	0x00000000	0x68616559	0x00000000
0xffffd798:	0x00006f4e	0x00000000	0x68616559	0x00000000
0xffffd7a8:	0x68616559	0x00000000	0x656b694c	0x0000796c
0xffffd7b8:	0x68676952	0x08040074	0x00006f4e	0x00000000
0xffffd7c8:	0x6576654e	0x00000072	0x6e6f7257	0x08040067
0xffffd7d8:	0x00736559	0x00000000	0x6576654e	0x00000072
0xffffd7e8:	0x0068614e	0x00000000	0x656b694c	0x0000796c
0xffffd7f8:	0x65727553	0x00000000	0x00736559	0x00000000
0xffffd808:	0x65727553	0x00000000	0x65727553	0x00000000
0xffffd818:	0x6576654e	0x00000072	0x656b694c	0x0000796c

Hint – don't forget gdb's trick about reversing byte ordering within each 4-byte chunk.

If the application were to output the value of `magic8ball[3][2]` – what would it be? i.e. what would be returned from the statement `printf(“%s”, magic8ball[3][2]);`

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(_____)/*same arg list as func1*/
{
    return _____;
}

int func1(_____)/*same arg list as func2*/
{
    x*=_____ ;
    y*=_____ ;
    z*=_____ ;
    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 ffff    jmpq 4004a0 <func2>
```


9. **Now That's a Switch (15 points):** A switch statement is described via the following assembly code fragments on x86-64:

```

4004d7: 83 f8 07      cmp  $0x7,%eax
4004da: 77 09         ja  4004e5 <func0+0x45>
4004dc: 89 c0         mov  %eax,%eax
4004de: ff 24 c5 28 06 40 00 jmpq *0x400628(,%rax,8)
4004e5: 89 ea         mov  %ebp,%edx
4004e7: d3 e2         shl  %cl,%edx
....
4004fd: 8d 14 29      lea  (%rcx,%rbp,1),%edx
400500: eb e7         jmp  4004e9 <func0+0x49>
400502: 89 ca         mov  %ecx,%edx
400504: 21 ea         and  %ebp,%edx
400506: eb e1         jmp  4004e9 <func0+0x49>
400508: 89 ca         mov  %ecx,%edx
40050a: 31 ea         xor  %ebp,%edx
40050c: eb db         jmp  4004e9 <func0+0x49>
40050e: 89 ca         mov  %ecx,%edx
400510: 09 ea         or   %ebp,%edx
400512: eb d5         jmp  4004e9 <func0+0x49>
400514: 89 ea         mov  %ebp,%edx
400516: 29 ca         sub  %ecx,%edx
400518: eb cf         jmp  4004e9 <func0+0x49>

```

And the following gdb interaction:

(gdb) x/32x 0x400628

```

0x400628: 0x004004e5  0x00000000  0x004004fd  0x00000000
0x400638: 0x00400514  0x00000000  0x0040050e  0x00000000
0x400648: 0x0040050e  0x00000000  0x004004e5  0x00000000
0x400658: 0x00400508  0x00000000  0x00400502  0x00000000

```

Three variables (i , j , k) are used in the switch statement – all three are declared as type *int*. Before the start of the code fragment above, variable i is in `%eax`, variable j is in `%ebp`, and variable k is in `%ecx`. Using this information, fill in the cases for the switch statement on the solution page.