1. ***Lost at C? (8 points)***: The following problem assumes the following declarations:

```
int x = random();
int y = random();
double d = foo();  // d is not NaN
unsigned ux = (unsigned) x;
```

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. $((x+y)<<4) + y-x == 17*y+15*x$ | (Y) | N |
| b. $((x >> 1) << 1) <= x$ | (Y) | N |
| c. $x < 0 \Rightarrow ((\text{int}) (ux >> 1)) < 0$ | Y | (N) |
| d. $d == (\text{float}) d \Rightarrow d < 4 \times 10^{38}$ | Y | (N) |

always > 0

d = ∞ is false

Also accepted Y as an answer for part d

Note that "⇒" represents an *implication*. A ⇒ 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):*** Suppose *A* and *B* are single bit values.

   a. Which of the following is equivalent to *A xor B*? Answer: A, B, D

      A) *(A+B) % 2*
      B) *(A | B) && !(A & B)*
      C) *!(A & B)*
      D) *(!A & B) | (A & !B)*

      Note that % is the modulo operator.

   b. Which one of the following expressions is equivalent to *A | (B & C)*? Answer: B

      A) *!A & !(B & C)*
      B) *!(!A & !(B & C))*
      C) *!A & !(!B & !C)*
      D) *!( !A & !(!B & !C))*
      E) *!A & (!B & !C)*
      F) *!( !A & (!B & !C))*

3. ***Three Times the Fun (8 points):*** Consider the following C functions:

```
int fun1(int a, int b)
      {
            if (a < b)
                  return a;
            else
                  return b;
      }

int fun2(int a, int b)
      {
            if (b < a)
                  return b;
            else
                  return a;
      }

int fun3(int a, int b)
      {
            unsigned ua = (unsigned) a;
            if (ua < b)
                  return b;
            else
            return ua;
      }
```

Which of the functions would compile into the assembly code shown here?

Answer: Fun 1 (1, 2, or 3)

```
      pushl %ebp
      movl %esp,%ebp
      movl 8(%ebp),%edx  a
      movl 12(%ebp),%eax  b
      cmpl %eax,%edx  a-b
      jge .L9
      movl %edx,%eax  return a
.L9:  movl %ebp,%esp  return b
      popl %ebp
      ret
```

a-b ≥ 0 becomes
a ≥ b

code:

```
if (a≥b)
  return b;
else //a<b
  return a;
```

⇓ switch if/else statements

```
if (a<b)
  return a;
else //a≥b
  return b;
```

4. ***Bit Off More Than You Can Chew? (12 points):*** Consider the 8-bit value 10110010. For the following different interpretations, provide the decimal equivalent of this 8-bit value. Assume that the word size of the machine interpreting this value is 8 bits – and in fact, all declared types only occupy 8 bits of space.
   a. An 8-bit signed integer

      $-78$

   b. An 8-bit unsigned integer

      $178$

   c. An 8-bit floating point value. It follows the IEEE format for encoding: assume that the sign takes 1 bit, the exponent takes 4 bits, and the fraction takes 3 bits.

      $-1.01 \cdot 2^{-1} = -6.25 \cdot 10^{-1} = -\frac{5}{8}$

5. ***Let Me EAX Another Question (10 points)***: Consider the following code fragment:

```
            movl $0x4, %edx
            movl $0x0, %eax
            movl $0x2, %ebx
start:      cmpl $0xC, %eax
            je end
            addl %edx, %eax
            imull %ebx, %eax
            j start
end:        nop
```

What is the value of *%eax* at the time that the nop is executed?

Accepted either 0xC or infinite.

6. ***Vexing Hexing (15 points):*** You are examining a memory dump (in hexadecimal) from a 32-bit (i.e. pointers use 32 bits), little-endian architecture for memory addresses from 0x400500 to 0x4005DF . Somewhere in this memory space there are five elements of a linked list – each element of the list has three fields: a *char* tag, a pointer to a 32-bit *float*, and a pointer to the next element of the list. You know that the first element of the linked list is located at address 0x40051C. Find the element that contains the tag 'B' and provide the 32-bit *float* value to which that node refers. Note – we want you to interpret the IEEE format of the *float* to tell us what number it actually represents.

| | | | | |
|---|---|---|---|---|
| 0x400500 | 41102413 | 42DBE044 | E7B56423 | 0411460A |
| 0x400510 | 24915123 | AAF41591 | 91F05420 | 468C0540 |
| 0x400520 | 00740540 | 2455CDDF | 41980540 | 00000000 |
| 0x400530 | 00243F50 | 04512242 | 428A0041 | 18210041 |
| 0x400540 | 42AA431D | F00D4E41 | FEEDBEEF | 24971021 |
| 0x400550 | 004005A0 | 42700540 | EACFEAE2 | 00000000 |
| 0x400560 | 2465E24D | 60210720 | F423E8D9 | 1BABE4FE |
| 0x400570 | 86A4B000 | 42900540 | 00A40540 | 00EC2342 |
| 0x400580 | 2458C931 | 8542C176 | AFE543DD | 91412124 |
| 0x400590 | FF800000 | 66124455 | 55442166 | 2254E243 |
| 0x4005A0 | DEFEC8ED | 45940540 | 00280540 | 0014DE9F |
| 0x4005B0 | 4121E890 | 01234567 | 24890ABC | DEF64322 |
| 0x4005C0 | 42400580 | 7013225A | 1BADBEAD | EBB90042 |
| 0x4005D0 | 49880540 | 001C0540 | 00000000 | 24400530 |

The address (in hex) of the linked list element that contains the tag 'B': 0x400574

The *float* contained inside that element : $4.63 \times 10^{-41}$

Float = 000080FF => s = 0

exp = 0

frac = 0x80FF

Float = $4.63 \times 10^{-41}$

7. *I Cannot Function in this Environment (15 points):*

```
call_func:
            ...
            ### PREP ###
            call func
            ...
```

```
func:
      ### SETUP ###
      movl 8(%ebp),%esi
      movl 12(%ebp), %edx
      leal (%esi,%edx,4), %esi
      push %esi
      call foo
      ### FINISH ###
      ret
```

The two code fragments above show an example of a caller and callee – function *call_func* will call function *func*. Note that there are three sections with ###'s: PREP, SETUP, and FINISH. This problem will involve you putting the correct code in these sections. You may assume that function *foo* modifies *only* register %eax – it takes one parameter. Note that the value returned by *foo* is the value we want to return by *func*. The function *func* takes two parameters (arg1 and arg2 – in that order), which are currently located in %ebx (arg1) and %edi (arg2). Before the call to *func*, all registers have values that we will need after the call has completed. This code is targeted for ia32 (i.e. a 32-bit x86 architecture). Your code should be efficient – every instruction you add should have a useful purpose (you may ignore cache alignment and memory performance for now), even if it has no effect on the system.

a. ***Preparing for the call to func*** – what code (if any) needs to be placed where the ### PREP ### placeholder is located? You should supply the *minimum* code required to satisfy the procedure call convention (i.e. stack discipline) and the specific requirements above.

```
pushl %edx
pushl %edi
pushl %ebx
```

b. ***Setup code for func*** – what code (if any) needs to be placed where the ### SETUP ### placeholder is located? You should supply the *minimum* code required to satisfy the procedure call convention (i.e. stack discipline) and the specific requirements above.

```
pushl %ebp
movl %esp, %ebp
pushl %esi
```

c. ***Finishing the call to func*** – what code (if any) needs to be placed where the ### FINISH ### placeholder is located? You should supply the *minimum* code required to satisfy the procedure call convention (i.e. stack discipline) and the specific requirements above.

```
movl -4(%ebp) %esi
movl %ebp, %esp
popl %ebp
```

8. ***Flipping Through Your Notes? (8 points):*** Answer the following questions with 3 words or less to name what is being described.

   a. An ISA that typically has many different variable length instructions, usually allowing them to access memory directly rather than via explicit data movement instructions.

      CISC

   b. An effective technique for handling switch statements in C code when the case statements are small integer constants.

      jump table

   c. The 32-bit register in ia32 that holds values returned by procedures.

      %eax

   d. Floating point values that are very close to 0.

      Denormalized numbers

9. ***Power Two, you People (12 points):*** Just like Lab #1, your task here is to use Your assignment is to complete a function skeleton using only straightline code (i.e., no loops or conditionals) and a limited number of C arithmetic and logical operators. Specifically in this problem, you are only allowed to use the following seven operators:

    ! ~ & ^ | + << >>

    Also, you are not allowed to use any constants longer than 8 bits.

```
/*
 * isPower2 - returns 1 if x is a power of 2, and 0 otherwise
 *   Examples: isPower2(5) = 0, isPower2(8) = 1, isPower2(0) = 0
 *   Note that no negative number is a power of 2.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 60
 *   Rating: 4
 */
int isPower2(int x) {
  int result;
  // count the number of 1 bits
  // Sum group of 4 bits each
  int bitcount;
  int m1 = 0x11 | (0x11<<8);
  int mask = m1 | (m1 << 16);
  int s = x & mask;
  s += x >> 1 & mask;
  s += x >> 2 & mask;
  s += x >> 3 & mask;
  // Combine high and lower order sums
} s = s + (s>>16);
// Low order 16 bits now consists of 4 sums, each ranging between 0 and 8.
//Split into two groups and sum
mask = 0xF | (0xF << 8);
s = (s & mask) + ((s >> 4) & mask);
bitcount = (s + (s >> 8)) & 0x3F;
// If bitcount is 1 and x is not Tmin, then x is a power of 2
result = !(~bitcount + 2) & ~(x >> 31);
return result;
```