Written Exercise Set 1 Solutions

Problem 1

1. d. Formula (d) is a pure bitwise operation (i.e., it does the same thing to each bit position), and this combination of OR, AND, and NOT is equivalent to XOR: it starts with the inclusive OR `a | b`, but then it also requires either `a` or `b` to be false, which makes the case when both a and b are true give false.

2. a. TMIN + TMAX = -1, because TMIN has only highest bit set, and TMAX has all but the highest bit set: when you add them bit by bit each bit in the result is 1 with no carries, and the all bits set value is -1. XORing with a mask that has all bits set is the same as flipping each bit, and that is also what `~a` does.

3. i. Formula (i) is the expression for division by 8 that rounds towards zero (floor rounding for positive inputs, ceiling rounding for negative inputs), which is the same as the C behavior of `a / 8`. Adding 7 when `a` is negative doesn't change the result when `a` is a multiple of 8, but it causes all other values to be rounded up towards 0 instead of down towards negative infinity.

4. l. The expression `((a << 31) >> 31)` first moves the lowest bit into the highest bit position, and then it duplicates that high bit back into all other bit positions, because `>>` on `int`s is arithmetic shift. Thus if `a` was odd, it gives the all 1s value -1, while if `a` was odd it gives zero. The bitwise complement exchanges those values, and then adding 1 turns -1 and 0 into 0 and 1 respectively. Thus the whole result is 0 if `a` was even and 1 if `b` was odd, which is the same as `a & 1`.

5. e. TMIN and TMAX are bitwise complements of each other, since TMIN as only the highest bit set, while TMAX has all but the highest bit set.

6. k. `a >> 31` duplicates the sign bit of a into all other bit positions, so it gives -1 if a was negative, and 0 if a was non-negative. The left shift is the same as multiplication by 2, so you get -2 for negative and still zero for non-negative. The bitwise complements of these are 1 and 0 respectively.

Descriptions of the unused right-column formulas:

"b". `~a | ~b` is equivalent to `~(a & b)`, which is another bitwise function of a and b, sometimes called NAND. But it's not equivalent to XOR (1).

"c". `a & TMIN` is `TMIN` if a is negative, and 0 otherwise. Like (6) it is a value that tells you whether a was negative, but the resulting values are different.

"f". `1 << 31` is TMIN. Arithmetic right shift gives the value whose top two bits are 1 and the rest 0, `0xc0000000`.

"g". `a | TMAX` is -1 if a is negative, and `TMAX` otherwise. Like (6) it is a value that tells you whether a was negative, but the resulting values are different.

"h". `1 << 30` is a value in which only the second-highest bit is set, `0x40000000`.

"j". `a >> 3` with an arithmetic right shift is related to signed division by 8, but it is division that always rounds towards negative infinity (floor rounding), which is different from the round-toward-zero rule that C uses (3).

You will also see that most of these expressions are different when you plug in any random values for `a` and `b`. For instance if `a` is `0x0x80ff00fe` and `b` is `0xffff0000`, the various expressions give:

```
0x00000000 4: a & 1
0x00000000 l: ~((a << 31) >> 31) + 1
0x00000001 6: (a < 0) ? 1 : -1
0x00000001 k: ~((a >> 31) << 1)
0x40000000 h: 1 << (31 - 1)
```

```
0x7f0000fe 1: a ^ b
0x7f0000fe d: (a | b) & (~a | ~b)
0x7f00ff01 2: ~a
0x7f00ff01 a: a ^ (TMIN + TMAX)
0x7f00ffff b: ~a | ~b
0x80000000 5: TMIN
0x80000000 c: a & TMIN
0x80000000 e: ~TMAX
0xc0000000 f: (1 << 31) >> 1
0xf01fe01f j: a >> 3
0xf01fe020 3: a / 8
0xf01fe020 i: ((a < 0) ? (a + 7) : a) >> 3
0xffffffff g: a | TMAX
```

Problem 2

The output will be:

```
starting x = 2
starting y = 7
a) 0x800
b) 0x804
c) 7
d) 0x802
e) -30
f) -30
g) 2
h) -30
```

At first x and y contain 2 and 7, as they were initialized. The addresses of `x`, `px`, `y`, and `py` are `0x800`, `0x802`, `0x804`, and `0x806` respectively, since each is 2 bytes long on this platform and they are allocated sequentially.

(a). Because `px` is initialized as `&x`, its initial value is the address of `x`, namely `0x800`.

(b). Because `py` is initialized as `&y`, its initial value is the address of `y`, namely `0x804`.

(c). The statement `px = py` makes `px` a pointer to the same location that `py` had been pointing to. Specifically `px` now points at `y`. `*px` is the value that `px` points to, namely the value of `y`, or 7.

(d). This statement prints the address of `px`. This never changes, it is still `0x802`.

(e). The statement `*px = 25` updates the value that `px` points to, to have the value 25. At this point `px` is still pointing at `y`, so this statements makes `y` have the value 25. However the following statement `y = -30` changes `y` again, this time to -30. `px` is still pointing at `y`, so `*px` is still the value of `y`, -30.

(f). The pointer `py` is still pointing at `y`, so the value of `*py` is the value of `y`, -30.

(g). The variable `x` has not been changed, so it still has the value 2.

(h). From the earlier assignment, `y` has the value -30.

Problem 3

A. In binary, TMax is always single 0 bit followed by 1 bits, and TMin is always single 1 bit followed by 0 bits. So in 8-bit two's complement, TMax is 01111111, and Tmin is 10000000.

B. TMax is $2^7 - 1 = 128 - 1 = 127$. TMin is $-2^7 = -128$.

C.

```
Carries:     11
            10010001   -128+16+1      =  -111
          + 00110110   32+16+4+2      = +  54
          ----------                    -----
            11000111   -128+64+4+2+1 =   -57
```

No overflow. There is never overflow when adding a negative and a positive number.

D.

```
Carries:    1111  1
            00011001   16+8+1          =    25  107+25=132
          + 01101011   64+32+8+2+1     = + 107          132-256 = -124
          ----------                     -----
            10000100   -128+4          =  -124
```

This is an overflow case. The sum of two positive numbers should be positive, but 107+25=132 is too large (greater than TMax). Instead the result is the negative value -124, which is $132 - 2^8$.

E.

```
Carries:   111
            01100110  64+32+4+2          =   102
          + 11111001  -128+64+32+16+8+1 = +  -7
          ----------                      -----
            01011111  64+16+8+4+2+1      =    95
```

As in C, adding a positive number and a negative number can never cause a signed overflow. (The carry out from the last column would be an unsigned overflow.) The result is the correct one.

F.

```
Carries:
            10000000   -128               =  -128
          + 01111111    64+32+16+8+4+2+1 = + 127
          ----------                       -----
            11111111                      =    -1
```

Again because TMin is negative and TMax is positive, there can be no overflow. Since TMin and TMax are complements, their sum is always negative 1.

G. To negate TMin, we complement it and add 1.

```
TMin      = 10000000 = -128
~TMin     = 01111111 = 127

Carries:    1111111
            01111111
          + 00000001
          ----------
            10000000 = -128
```

The negation of -128 should be +128, but +128 is not representable, so it overflows back to -128.

H. Multiplying by 2 is the same as shifting left by one place.

```
TMax   = 01111111
2*TMax = 11111110 = -128+64+32+16+8+4+2 = -2
```

Another way to see that the result is -2 in decimal is to observe that if you added 1 to it, you would get -1. Normally multiplying a positive number by a positive number should give a positive result, in particular 2*127 = 254. But 254 is too large to be representable and it overflows to -2.

Problem 4

A. Rather than going via decimal, the easiest way to convert from hexadecimal to octal is to go via binary: you just have to regroup the bits.

```
0xAA29 = 1010  1010  0010  1001
         1 010 101 000 101 001 (regroup by 3s)
         1   2   5   0   5   1
= 0125051 (octal)
```

But to check, here's the conversion from hex to decimal:

```
0xAA29 = 10*16**3 + 10*16**2 + 2*16 + 9
       =    40960 +     2560 +   32 + 9
       = 43561
```

To convert this to octal, you compute octal digits by repeatedly dividing by 8: the remainders are the digits right to left.

```
43561 = 8*5445+1                                43561/8 = 5445.125 (.125 = 1/8)
      = 8*(8*680 + 5) + 1                        5445/8 =  680.625 (.625 = 5/8)
      = 8*(8*(8*85 + 0) + 5) + 1                  680/8 =   85
      = 8*(8*(8*(8*10 + 5) + 0) + 5) + 1           85/8 =   10.625 (.625 = 5/8)
      = 8*(8*(8*(8*(1*8 + 2) + 5) + 0) + 5) + 1  10/8 =    1.25  (.25  = 2/8)
= 0125051 (octal)
```

B. Same approach as in A.

```
0xBF8A = 1011  1111  1000  1010
         1 011 111 110 001 010 (regroup by 3s)
         1  3    7   6   1   2
= 0137612 (octal)
```

C.

```
0x25C0 = 2*16**3 + 5*16**2 + 12*16 + 0
       =    8192 +    1280 +   192 + 0
       = 9664
```

D.

```
88888_9 = 8*9**4 + 8*9**3 + 8*9**2 + 8*9 + 8
        =  52488 +   5832 +    648 +  72 + 8
        = 59048
```

As a shortcut, you can instead observe that this value is one less than $100000_9 = 9^5$. $9^5 - 1 = 59048$.

E.

```
 32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
     1     0    0    0    1    0   0   0   1  0  0  0 1 0 0 0

32768+2048+128+8 = 34952
```

You may notice another pattern here: the repeated 1000 bits mean that this is equivalent to `0x8888`. You can get the value of any sequence of repeated digits in any base by applying the formula for the sum of a finite geometric series. Here:

$$8\sum_{k=0}^{4-1} 16^k = 8 \cdot \left(\frac{16^4 - 1}{16 - 1}\right) = 8 \cdot 65535/15 = 34952$$

Or, written purely in hex: `0x8888 = 8 * (0xffff/0xf)` where `0xffff = 0x10000 - 0x1`.

Problem 5

a. -1 is $-1 \cdot 1 \cdot 2^0$, so s=1, M=1, and E=0. E=0 is encoded as exp = 15 = 01111. M=1 has only the leading 1 bit which is not encoded, so it turns into frac = 0000000000. To convert to hex, write the fields in order in binary and then regroup the bits by 4s:

```
1 01111 0000000000
1011 1100 0000 0000
   b    c    0    0
0xBC00
```

b. The smallest value greater than -2 will be the negative of the largest value less than +2. 2 is the smallest number with E=1, so the largest value less than it will be the largest value with E=0. That means its significand should be almost 2: the encoding frac = 1111111111 means 1.1111111111. With 10 bits to the right of the radix point, this is $(2^{11} - 1)/2^{10}$. Thus M = 2047/1024 or equivalently M = $2047 \cdot 2^{-10}$. s = 1 and exp = 15 = 01111, so converting to hex gives:

```
1 01111 1111111111
1011 1111 1111 1111
   b    f    f    f
0xBFFF
```

In decimal that is -1.9990234375 exactly, or -1.999023 rounded.

c. The smallest value of the exponent field exp = 0 = 00000 is reserved for denormalized numbers, so the smallest exponent for a normalized number is exp = 1 = 00001, which corresponds to E = -14 since -14 + 15 = 1. The smallest significand is the one with all zero bits, 0000000000, which represents M = 1. Combining with s = 0 gives:

```
0 00001 0000000000
0000 0100 0000 0000
   0    4    0    0
0x0400
```

2^{-14} is 0.00006103515625 exactly, or 0.000061 rounded.

d. To convert from hex, we change to binary and then regroup into the appropriate fields:

```
0x5BE2
   5    b    e    2
0101 1011 1110 0010
0 10110 1111100010
```

This gives s = 0, exp = 10110 = 22, and a significand of 1.111110001. Removing the bias, E = exp - 15 = 7. The significand has 9 digits after the decimal, so we convert 1111110001_2 = 512+256+128+64+32+16+1 = to give M = 1009/512 or M = $1009 \cdot 2^{-9}$. Multiplying by 2^7 to get V cancels all but 2^2 in the denominator, leaving 1009/4 or $1009 \cdot 2^{-2}$. In decimal that is 252.25.

e. Infinite values are represented with the largest possible value of exp = 11111 = 31. Subtracting the bias this corresponds to E = 16, though that value is not used in computation. The sign should be s = 0 since this is positive not negative infinity. And infinity values are distinguished from NaNs by the frac field being all zero bits

0. It's not really defined whether infinite values have an implied leading 1 digit for the significand, so this could be thought of as either M = 0 or M = 1. Converting to hex gives:

```
0 11111 0000000000
0111 1100 0000 0000
   7    c    0    0
0x7C00
```

f. NaN values have the same exp = 11111 (E = 16) as infinite values, but any non-zero frac field. For instance we can take frac = 1111111111. The sign can be either 0 or 1. Converting to hex gives:

```
0 11111 1111111111
0111 1111 1111 1111
   7    f    f    f
0x7FFF
```