1.  
Determine how many digits are necessary to represent all possible values of the

A. sum of 512 integers, S = X1+X2+…+X512, with each Xi in the range from 0 to 127:  
 ­­ ­\_ \_  
With k = 7, n = 512, S = k + |log2 n|  
 \_ \_  
Radix-2: so 7 + |log2 512| = 7 + 9 = 16  
Radix-10: so 3 + |\_log10 512\_| = 3 + 2 = 5  
  
B. product of 100 integers, P = Y1·Y2·…·Y100, with each Yi the range from 0 to 999  
In order to represent all possible values (i.e. including the case of all values being 1000):  
Radix-2: It takes 10 bits to represent 1000 different values, so if we do 10\*100, we have 1000 bits required to be able to represent all possible values (although in many cases this would just lead to leading 0’s before the first 1).  
Radix-10: The same thing takes place here; we need 3 decimal digits to represent 1000 different values, and so we do 3\*100 = 300 decimal digits required.

2.  
a. With (1765432.432104)8, we first convert to binary:  
001 111 110 101 100 011 010 . 100 011 010 001 000 100  
From here, we can easily group into four bits to find the hex (i.e. 23 = 8, 24 = 16 [hex]):  
0000 0111 1110 1011 0001 1010 . 1000 1101 0001 0001 0000  
Now we can simply convert each set of four bits into a hexadecimal value:  
(07EB1A.8D110)16  
b.357246.1234561  
Again we repeat this process:  
357246.1234561  
011 101 111 010 100 110 . 001 010 011 100 101 110 001  
to:  
0001 1101 1110 1010 0110 . 0010 1001 1100 1011 1000 1000  
Which is: (1DEA6.29CB88)163.  
k = 8, l = 8

a. -103.95703125  
The positive value in binary is: 01100111.11110101

Signed Magnitude:  
-(64 + 32 + 4 + 2 + 1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/64 + 1/256) 🡪  
11100111.11110101  
One’s Complement:  
From 01100111.11110101, we simply invert the bits to get the negative:  
10011000.00001010  
Two’s Complement:  
10011000.00001011  
Biased with Base B = 128:  
-128 + 16 + 8 + 1/32 + 1/128 + 1/256 🡪  
00011000.00001011  
Biased with base B = 128-ulp:  
We bias with B = 127:  
-127 + 16 + 7 + 4 + 2 + 1/32 + 1/128 + 1/256 🡪   
00010111.00001011  
  
b. -61.43359375  
Representing 61.43359375 in binary:  
00111101.01101111

Signed Magnitude:  
-(32+16+8+4+ 1 + 1/4 + 1/8 + 1/32 + 1/64 + 1/128 + 1/256) 🡪  
10111101.01101111  
One’s Complement:

11000010.10010000  
Two’s Complement:  
11000010.10010001  
Biased with Base B = 128:  
-128 + 64 + 2 + 1/2 + 1/16 + 1/256 🡪  
01000010.10010001  
Biased with base B = 128-ulp:  
-127 + 64 + 1 + 1/2 + 1/16 + 1/256 🡪  
We bias with B = 127:  
01000001.10010001  
  
4. Now, with k’ = 12, l’ = 12:  
a. -103.95703125  
The positive value in binary is: 000001100111.111101010000

Signed Magnitude:  
100001100111.111101010000  
One’s Complement:  
111110011000.000010101111  
Two’s Complement:  
111110011000.000010110000  
Biased with Base B = 2048:  
-2048 + 1024 + 512 + 256 + 128 + 16 + 8 + fractions 🡪  
011110011000.000010110000  
Biased with base B = 2048-ulp:  
-2047 + 1024 + 512 + 256 + 128 + 16 + 4 + 2 + 1 + fractions 🡪  
011110010111.000010110000  
  
b. -61.43359375  
Representing 61.43359375 in binary:  
000000111101.011011110000

Signed Magnitude:  
100000111101.011011110000  
One’s Complement:

111111000010.100100001111

Two’s Complement:  
111111000010.100100010000

Biased with Base B = 2048:  
-2048 + 1024 + 512 + 256 + 128 + 64 + 2 + fractions 🡪  
011111000010.100100010000  
Biased with base B = 2048-ulp:  
-2047 + 1024 + 512 + 256 + 128 + 64 + 1 + fractions 🡪  
011111000001.100100010000  
5.  
Determine all bits of the ANSI/IEEE standard single-precision floating-point representation of the following numbers (Hint: use the default rounding scheme if necessary):  
range for exponent: [-126, 127] (e + bias)  
A. -8.56BCF8016 × 2-130  
-1000.0101 0110 1011 1100 1111 1000 0000\* 2-130  
-.10000101 0110 1011 1100 1111 1000 0000\* 2-126  
1, 00000000, 10000101011010111100111110000000🡪   
1, 00000000, 10000101011010111101000  
B. 8.56BCF8016 × 21251000.0101 0110 1011 1100 1111 1000 0000 \* 2125This number is too large for normal representation so we must express it as +infinity:  
0, 11111111, 00000000000000000000000  
C. -CD.EFAB16 × 2-134  
-1100 1101 . 1110 1111 1010 1011 🡪  
-0. 1100 1101 1110 1111 1010 1011 \* 2-126 🡪  
1, 00000000, 11001101111011111010110  
D. -CD.EFAB16 × 2120  
1100 1101 . 1110 1111 1010 1011 \* 2120  
1, 11111110, 10011011110111110101100E. (-infinity) / (+infinity) = - NaN  
1, 11111111, 11111111111111111111111

F. (+infinity) - (-infinity) = +infinity  
0, 11111111, 00000000000000000000000

G. 0/(-infinity)  
This results in zero, which can be represented as:  
0, 00000000, 00000000000000000000000  
6.  
What numbers are represented by the following hexadecimal strings, if these strings are treated as the ANSI/IEEE standard single-precision representations of real numbers.

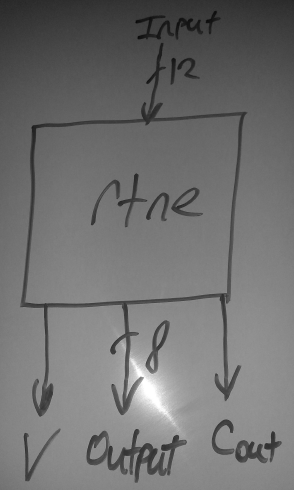
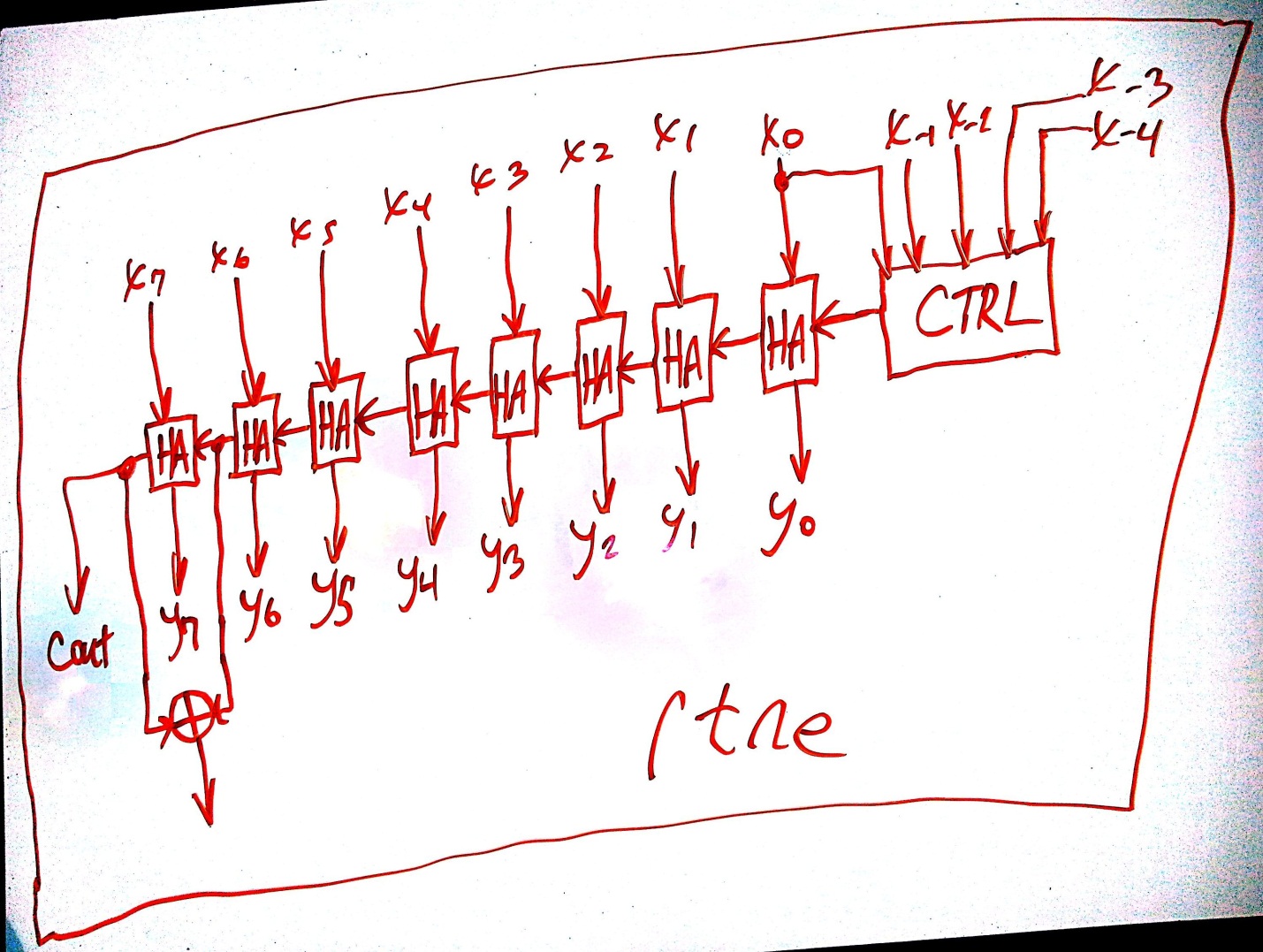
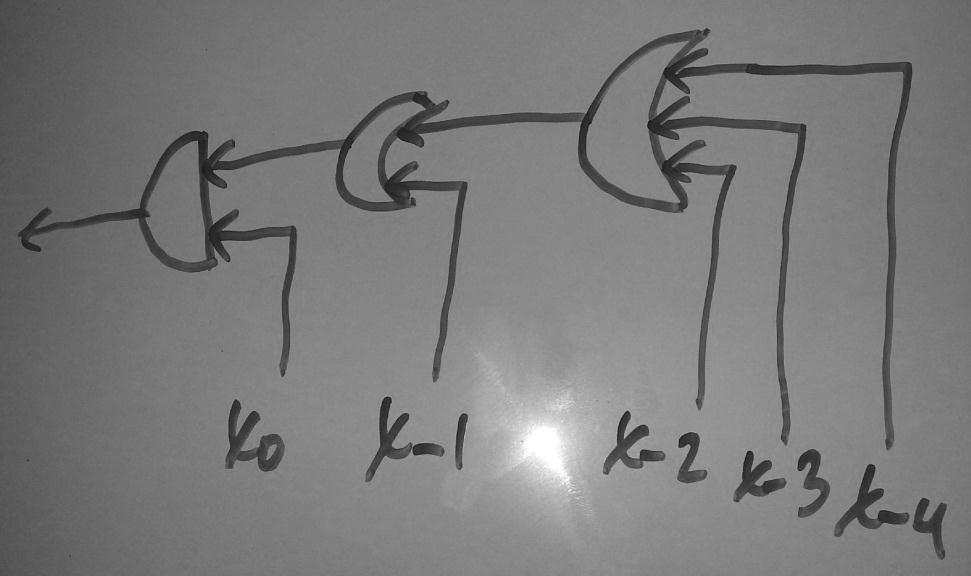
Please express the integral and fractional part of each number, as well as its exponent using decimal representation.

A. EDCB0000  
1110 1101 1100 1011 0000 0000 0000 0000  
1, 11011011, 10010110000000000000000  
Sign: Negative  
E + Bias: 11011011  
F: 10010110000000000000000  
Result: -(1 + 1/2 + 1/16 + 1/64 + 1/128)\*292 = -1.5859375\*292 = -7.853182\*1027

B. 80120000 (DENORMAL)  
1000 0000 0001 0010 0000 0000 0000 0000

1 00000000 00100100000000000000000

Sign: Negative  
E + Bias: 00000000 = 2-127  
F: 00100100000000000000000, we compute the mantissa differently: 1/4 + 1/32 = 0.28125  
Result: Sign \* 2e \* mantissa = -1 \* 2-127 \* 0.28125 = -1.6530389\*10-39  
C. FF800000  
1111 1111 1000 0000 0000 0000 0000 0000  
1 11111111 00000000000000000000000  
-Infinity

D. FF400000  
1111 1111 0100 0000 0000 0000 0000 0000  
1 11111110 10000000000000000000000  
Sign: Negative  
E + Bias: 11111110 = 2127  
F: 10000000000000000000000  
Result: -1 \* 2127 \* 1.5 = -2.5521178\*10387.  
Along with having a carry out bit, we will also need an overflow bit to perform these signed roundings with the rounding having a tiebreaker to even (i.e. 1.5 rounds to 2 and 0.5 rounds to 0). Otherwise, the result is the same between signed and unsigned rounding under this scenario.  
Adders are used to propagate a ‘1’ from a truncation circuit (although it could simply pass a zero) depending upon whether we round up or down.  
Cout = 1 for the case where we have “11111111” and we must round up, and V = 1 for the case of having 01111111 rounding up to 10000000 (i.e. we are incrementing by 1 and truncating the fractional bits).  
The bits we must consider are bits x0, x-1, x-2,and x-3, as these will decide if we have a simple truncation of if we must also pass a 1 to the half adders.  
Below is the external view of the circuit:  
  
  
  
  
Taking inputs as x7 to x0 for the integer parts and x­-1 and x-4 as the fractional parts, and the CTRL (control) module used to determine if we increment or not, we have the following circuit:  
  
  
The CTRL module will produce a ‘1’ as output under the following circumstances:  
(The + operator is used to express the logical OR operator)  
x0 = 1  
x-1 = 1  
x-2 + x-3 + x-4 = 1,  
  
x0 = 1  
x-1 = 1  
x-2 + x-3 + x-4 = 0,  
  
x0 = 0  
x-1 = 1  
x-2 + x-3 + x-4 = 1,  
  
For example, 0.5 will round to 0, in which case we do not need to increment, but for 1.5, we will need to increment to 2 for the rounding. Case 1 is greater than 1.5, case 2 is a tie-breaker at 1.5 (both rounding to 2), and case 3 is greater than 0.5, which in turn rounds to 1.  
We will need the inside of our CTRL module to look like this:  
  
8.  
//In C:  
#include <stdio.h>

#include <stdint.h>

int main()

{

int x = 1;

char \*first = (char\*) & x;

if((\*first ? 0 : 1) == 1)

printf("Big Endian\n");

else

printf("Little Endian\n");

}

//Also, in Java it is even simpler:  
import java.nio.ByteOrder;

public class Endian

{

public static void main(String args[])

{

ByteOrder x = ByteOrder.nativeOrder();

if (x.equals(ByteOrder.BIG\_ENDIAN))

{

System.out.println("Big-endian");

}

else

{

System.out.println("Little-endian");

}

}

}  
  
The result of my test is that the processor of my system is little endian. The brand is Intel and the model is the i7 processor.

9.  
P(x) = x5 + x2 + 1.

A. '02' · '1F'

B. '05' · '1B'

C. '1F' · '1F'.  
a. Given P(x) = x5 + x2 + 1, we calculate 02 \* 1F. The operands can be expressed in binary for simplification with:  
00000010 \* 00011111 = 00111110,   
Then,  
00111110  
00100101  
00011011 = x4 + x3 + x + 1 = ‘1B’

b. Next, we compute 05 \* 1B  
We perform the multiplication  
as follows: (((x\*2)\*2)+x), with x = 1B:  
00011011 \* 2 = 00110110  
00110110  
00100101  
00010011  
We then multiply this value by 2 after having reduced it by P(x):  
00010011 \* 2 = 00100110  
00100110  
00100101  
00000011  
Finally, we add 1B back in:  
00000011 + 1B = 00011000, which is x4 + x3 = ‘18’When I use the method discussed in class, I am able to verify these solutions.

c. To compute 1F \* 1F, we can do the following:  
Already knowing 2\*1F = 00011011, we must compute 31\*x:  
((((2\*x)\*2)\*2)\*2) + 8\*x + 4\*x + 2\*x + x  
00011011 \* 2 = 00110110  
00110110  
00100101  
00010011  
00010011 \* 2 = 00100110  
00100110  
00100101  
00000011  
00000011 \* 2 = 00000110  
So far we have:  
00000110 + 8\*x + 4\*x + 2\*x + x  
2x = 00011011  
4x = 00010011  
8x = 00000011  
16x = 00000110  
So we perform addition:  
00000110 +   
00000011 +   
00010011 +   
00011011 +   
00011111  
00010010 = x4 + x = ‘12’  
B1.  
By deduction, we need to replicate the values of the most significant bit when extending a two’s complement number as we have ak-1\*xk-1th bit increasing the magnitude of the negative number when it is a 1 and not affecting the final value when it is 0. In other words, if we have 8 bits, we need to overcome -128 to reach our value, and if we have 12 bits, we must overcome a negative value of -2048. Thus, for every bit we extend to the left, we need to ensure that it is a negative value (msb stays as a 1 for the case of the original number being negative), and that we have the other bits set to bring this higher magnitude negative number back to the original value. In an extension of the fractional bits, we must have these bits set to 0 to represent the original number. This is because our fractional value is not affected by increasing the number of bits used to represent the number.B2. Yes, this is true. By logical deduction, if any of the “fractional part” bits are selected, with no “e+bias” bits selected, we will have a denormal, which is greater than zero, provided our sign bit is 0, and denormal representations are lower than ordinary number representations. If at least one bit of the “e + bias” field is selected, we no longer have denormalized values and so our binary representation as well as our converted one are larger. If we actually select all of the bits of the “e + bias” field, we again increase our binary representation and our converted value (the latter being a representation of infinity). Even if we have the maximum representable real number before infinity, our binary representation cannot exceed that of the representation for infinity. The same applies for the case of having the sign bit negative – the negative denormals will be less than the ordinary numbers, which are less than –infinity.   
In other words, in terms of magnitude, 0 < denormals < ordinary numbers < infinity in both the actual value we are trying to represent as well as the signed magnitude format of bit placement.  
B3.  
Big Endian: Xilinx Microblaze, IBM POWER, Motorola 6800  
Little Endian: MCS-48, 8051, DEC Alpha, Altera Nios II, Atmel AVR  
Bi-Endian: PowerPC, Alpha, MIPS, PA-RISC, SuperH SH-4   
How to switch between big and little-endian conventions:  
Modes of operation can be changed in the software (for example at boot-time), but this can also be done at the hardware level with bit-shifting. PowerPC, for example, does a simple byte-swapping when a different representation is required, for example when dealing with 32-bit fields of network packets. This can prove costly though as this increases the complexity to process the lengths of fields and to allow for this byte-swapping or bit-shifting whenever it is necessary.