Problem 1:
How would you test for overflow, the result of an addition of two 8-bit operands if the operands were (i) unsigned (ii) signed with 2s complement representation.
Add the following 8-bit strings assuming they are (i) unsigned (ii) signed and represented using 2's complement. Indicate which of these additions overflow.
A. 0110 1110 + 1001 1111

- Unsigned addition: 110 + 159 = 269. 269 > 2^8 - 1 => overflow has occurred since 269 cannot be represented with 8 bits
- Signed addition: 110 - 97 = 13. 13 < 2^7 - 1 => no overflow has occurred since 13 can be represented with 8 bits in signed form

B. 1111 1111 + 0000 0001

- Unsigned addition: 255 + 1 = 256 > 2^8 - 1 => overflow has occurred
- Signed addition: -1 + 1 = 0, 0 can be represented in 2's complement form in 8 bits => no overflow has occurred

C. 1000 0000 + 0111 1111

- Unsigned addition: 128 + 127 = 255 = 2^8 -1 => no overflow has occurred since 255 can be represented with 8 bits
- Signed addition: - 128 + 127 = -1, -1 can be represented in 2's complement form in 8 bits => no overflow has occurred

D. 0111 0001 + 0000 1111

- Unsigned addition: 113 + 15 = 128 < 2^8 -1 => no overflow has occurred since 128 can be represented with 8 bits
- Signed addition: 113 + 15 = 128, cannot be represented in 2's complement form => an overflow will occur

Problem 2:
One possible performance enhancement is to do a shift and add instead of an actual multiplication. Since 9×6, for example, can be written (2×2×2+1)×6, we can calculate 9×6 by shifting 6 to the left three times and then adding 6 to that result. Show the best way to calculate 0xABhex × 0xEFhex using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

- To represent the result of the multiplication, we will need at least 16 bits
- ABhex = 171, EFhex = 239
- 239 = (256 - 17), 17 = (16 + 1)
- 256 = 2^8, 16 = 2^4
- To obtain the result,
    - Left shift 171 four times and add 171 to it to obtain 171 x 17 - a
    - Left shift 171 eight times to obtain 171 x 256 - b
    - Subtract a from b

Problem 3:
What decimal number does the 32-bit pattern 0×DEADBEEF represent if it is a floating-point number? Use the IEEE 754 standard

- 32 bits => Single Precision Floating Point
- Binary representation of 0xdeadbeef:
- 1101 1110 1010 1101 1011 1110 1110 1111
- S: 1 => -1
- Exponent: 10111101 => 189, Exponent Bias = 189 - 127 = 62
- Fraction: 01011011011111011101111 => 1 + 2^-2 + 2^-4 + 2^-5 + 2^-7 + 2^-8 + 2^-10 + 2^-11 + 2^-12 + 2^-13 + 2^-14 + 2^-16 + 2^-17 + 2^-18 + 2^-20 + 2^-21 + 2^-22 + 2^-23 =
  1.35738933086395262997999
- Decimal representation: -1.35738933086395262997999 * 2^62 =
  -6,259,853,398,707,797,984.9229547264623

Problem 4:
Write down the binary representation of the decimal number 78.75 assuming the IEEE 754 single precision format. Write down the binary representation of the decimal number 78.75 assuming the IEEE 754 double precision format

- Number is positive, sign bit is zero
- 78 in binary representation: 1001110
- 0.75 in binary representation: 110
- 78.75 = 1001110.110 = 1.00111011 x 2^6
- Exponent, 32 bits = 6 + 127 = 133 = 10000101
- Exponent, 64 bits = 6 + 1023 = 1029 = 10000000101
- 32 bit : 01000010100111011000000000000000
- 64 bit: 0100000001010011101100000000000000000000000000000000000000000000

Problem 5:
Write down the binary representation of the decimal number 78.75 assuming it was stored using the single precision IBM format (base 16, instead of base 2, with 7 bits of exponent).

- Number is positive, sign bit is zero
- 78 in base16 representation: 4E
- .75 in base16 representation: .C
- 78.75 in base16 representation: 4E.C = 4.EC * 16
- 78.75 in binary representation: 01001110.110
- Shift right by two hexadecimal digits: 0.01001110110 * 16^2
- Bias of 64 added to exponent of 2, 66: 1000010 binary rep
- 32 bit rep with base 16: 01000010010011101100000000000000

Problem 6:
IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa (fractional field) is 10 bits long. A hidden 1 is assumed.

(a) Write down the bit pattern to represent −1.3625 ×10−1. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

- -1.3625 * 10^-1 = -0.13625
- Number is negative => sign bit is turned on
- 0.13625 binary representation: 0.001000101110000101
- = 1.000101110000101 * 2^-3
- Exponent = -3 + 15 = 12
- 16 bit representation: 1011000001011110000101
- Bits marked in red will not be stored due to the limited precision of 10 bits, truncation error
- In case of a 32 bit representation, -0.13625 could've been represented perfectly
- Range of numbers in single precision : 2^(-126) to 2^(+127)
- Range of numbers in single precision : 2^(-14) to 2^(+15)

(b) Calculate the sum of 1.6125 ×101 (A) and 3.150390625 ×10−1 (B) by hand, assuming operands A and B are stored in the 16- bit half precision described in problem a. above Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps.

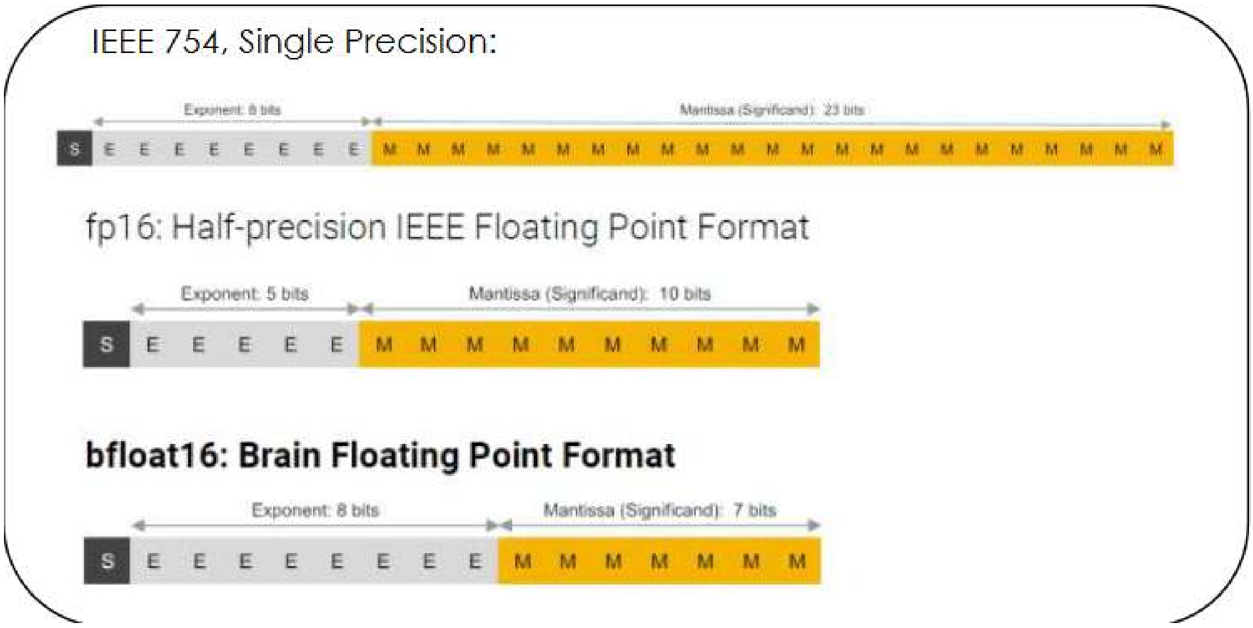
- 1.6125 * 10^1 = 16.125
- Binary representation: 10000.001 = 1.0000001 * 2^4 (A)
- Exponent with bias: 15+4 = 19, binary representation: 10011
- 16 bit representation: 0100110000001000

- 3.150390625 ×10^-1 = 0.3150390625
- Binary representation: 0.0101000010100110011 = 1.01000010100110011 * 2^-2 (B)
- Exponent with bias: 15 -2 = 13, binary representation: 1101
- 16 bit representation: 0011010100001010

We can't directly add the binary representations because they don't have the same exponents, we can shift (A) to left by 6 bits

- 1.0100001010
- + .00010000001 (Truncation error)
- 1.0101001010 * 2^4
- Already normalized, no errors while adding the numbers. I do not know what to do with the Truncation and representation errors

Problem 7:
What is the range of representation and relative accuracy of positive numbers for the following 3 formats:
(i) IEEE 754 Single Precision (ii) IEEE 754 – 2008 (described in Problem 6 above) and (iii) 'bfloat16' shown in the figure below


IEEE 754, Single Precision:
Exponent: 8 bits
Mantissa (Significand): 23 bits
S E E E E E E E E M M M M M M M M M M M M M M M M M M M M M M M
fp16: Half-precision IEEE Floating Point Format
Exponent: 5 bits
Mantissa (Significand): 10 bits
S E E E E E M M M M M M M M M M
bfloat16: Brain Floating Point Format
Exponent: 8 bits
Mantissa (Significand): 7 bits
S E E E E E E E E M M M M M M M

Name	Mantissa bits	Exponent bits	Exponent bias	Smallest positive number
Single precision	**23**	**8**	**127**	**2^(-126-(23))**
fp16	**10**	**5**	**15**	**2^(-14-(10))**
bfloat16	**7**	**8**	**127**	**2^(-126-(7))**

Problem 8:

Suppose we have a 7-bit computer that uses IEEE floating-point arithmetic where a floating point number has 1 sign bit, 3 exponent bits, and 3 fraction bits. All of the bits in the hardware works properly. Recall that denormalized numbers will have an exponent of 000, and the bias for a 3-
bit exponent is
2^3-1 – 1 = 3.
(a) For each of the following, write the binary value and the corresponding decimal value of the 7-bit floating point number that is the closest available representation of the

requested number. If rounding is necessary, use round-to-nearest. Give the decimal values either as whole numbers or fractions. The first few lines are filled in for you.

Number	Binary	Decimal
0	0 000 000	0
-0.125	1 000 000	-0.125
Smallest positive normalized number	0 001 000	0.25
largest positive normalized number	0 111 111	90
Smallest positive denormalized number > 0	0 000 001	0.125
largest positive denormalized number > 0	0 000 111	0.875

(b) The associative law for addition says that a + (b + c) = (a + b) + c. This holds for regular arithmetic, but does not always hold for floating-point numbers. Using the 7-bit floating-point system described above, give an example of three floating-point numbers a, b, and c for which the associative law does not hold, and show why the law does not hold for those three numbers.