NYU Tandon School of Engineering

Fall 2022, ECE 6913

Homework Assignment 4

Instructor: Azeez Bhavnagarwala, email: ajb20@nyu.edu

Course Assistants

Varadraj Kakodkar (vns2008), Kartikay Kaushik (kk4332), Siddhanth Iyer (si2152), Swarnashri Chandrashekar (sc8781), Karan Sheth (kk4332), Haotian Zheng (hz2687), Haoren Zhang (kk4332), Varun Kumar (vs2411)

Homework Assignment 4 [released Wednesday October 5th 2022] [due Wednesday October 12th by 11:59PM]

You *are allowed* to discuss HW assignments with anyone. You are *not allowed* to share your solutions with other colleagues in the class. Please feel free to reach out to the Course Assistants or the Instructor during office hours or by appointment if you need any help with the HW. Please enter your responses in this Word document after you download it from NYU Classes. *Please use the Brightspace portal to upload your completed HW.*

---

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

(i) 0110 1110 + 1001 1111 = 1 0000 1101, there is an overflow

(ii) 0110 1110 + 1001 1111 = 1 0000 1101 = $(13)_{10}$, there is not an overflow

B. 1111 1111 + 0000 0001

(i) 1111 1111 + 0000 0001 = 1 0000 0000, there is an overflow

(ii) 1111 1111 + 0000 0001 = 1 0000 0000 = $(0)_{10}$, there is not an overflow

C. 1000 0000 + 0111 1111

(i) 1000 0000 + 0111 1111 = 1111 1111 = $(255)_{10}$, there is not an overflow

(ii) 1000 0000 + 0111 1111 = 1111 1111 = $(-1)_{10}$, there is not an overflow

D. 0111 0001 + 0000 1111

(i) 0111 0001 + 0000 1111 = 1000 0000 = $(255)_{10}$, there is not an overflow

(ii) 0111 0001 + 0000 1111 = 1000 0000 = $(-128)_{10}$, there is an overflow

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 $0xAB_{hex} \times 0xEF_{hex}$ using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

0xAB = 1010 1011 = $(171)_{10}$

0xEF = 1110 1111 = $(239)_{10}$ = 2^8 - 2^4 - 1

So first we can left shift 0xAB 4 bits, record this number as n1. Then we left shift 4 more bits, record this number as n2. Thus the result can be represented by n2 - b1 - 0xAB. The result is 40869.

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

0xDEADBEEF = 1101 1110 1010 1101 1011 1110 1110 1111

the sign bit is 1

The exponent part is 1011 1101 = 189, E-bias = 189 - 127 = 62

The fractional part is 010 1101 1011 1110 1110 1111 = 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) = 1.3573893308639526

Thus the floating number is:

(-1)^1 * 2^62 * 1.3573893308639526 = -6.259853398707798016 * 10^18

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

Single precision:

The sign bit is 0.

78.75 = 0100 1110.11 = 1.0011 1011 * 2^6

The exponent part is: 127 + 6 = 133 = 1000 0101

The fractional part is: 001 1101 1000 0000 0000 0000

Thus the single precision format is:

0100 0010 1001 1101 1000 0000 0000 0000

Double precision:

The sign bit is 0.

The exponent part is: 1023 + 6 = 1029 = 100 0000 0101

The fractional part is: 0011 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Thus the double precision format is:

0100 0000 0101 0011 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

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).
The sign bit is 0.
78.75 = 0x4E.C = 0100 1110.1100
The normalized result is:
0.0100 1110 1100 * 16^2

According to https://en.wikipedia.org/wiki/IBM_hexadecimal_floating-point, the bias is 64
Thus, the exponent is 2 + 64 = 66 =100 0010

Finally, the IBM format is 0100 0010 0100 1110 1100 0000 0000 0000

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

The sign bit is 1.

0.13625 = 0.0001011100 = 1.0001 0111 00 * 2^(-4)

Thus the exponent part is -3 + 15 = 12 = 0 1100

Thus, the 16-bit half precision format is

1011 0000 0101 1100

If we use IEEE 754 standard format:

the exponent part is -3 + 127 = 124 = 01111100

The complete format is:

1011 1110 0000 1011 1000 0101 0001 1111

The accuracy of 16-bit floating point format is 2 ^(-10). The range is [2 ^(-14), 2^16].

The accuracy of IEEE 754 floating point format is 2 ^(-23). The range is [2 ^(-126), 2^128].

Therefore, the accuracy and the range of 16-bit floating point format are both weaker than IEEE 754 format.

(b) Calculate the sum of **1.6125 ×10**^{1} (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 = 1 0000.001 = 1. 0000 001 * 2^4
The exponent part is 4 + 15 = 19 = 10011
Thus the 16-bit half precision format is
0 10011 0000 0010 00

3.150390625 * 10^(-1) = 0.3150390625 = 1.0100 0010 1001 1001 100 * 2^(-2)

Now we have

1.0000 001* 2^4
+ 1.0100 0010 1001 1001 100 * 2^(-2)

1.0000 0010 0000 0000 0000 0000 0 * 2^4
+ 0.0000 0101 0000 1010 0110 0110 0 * 2^4

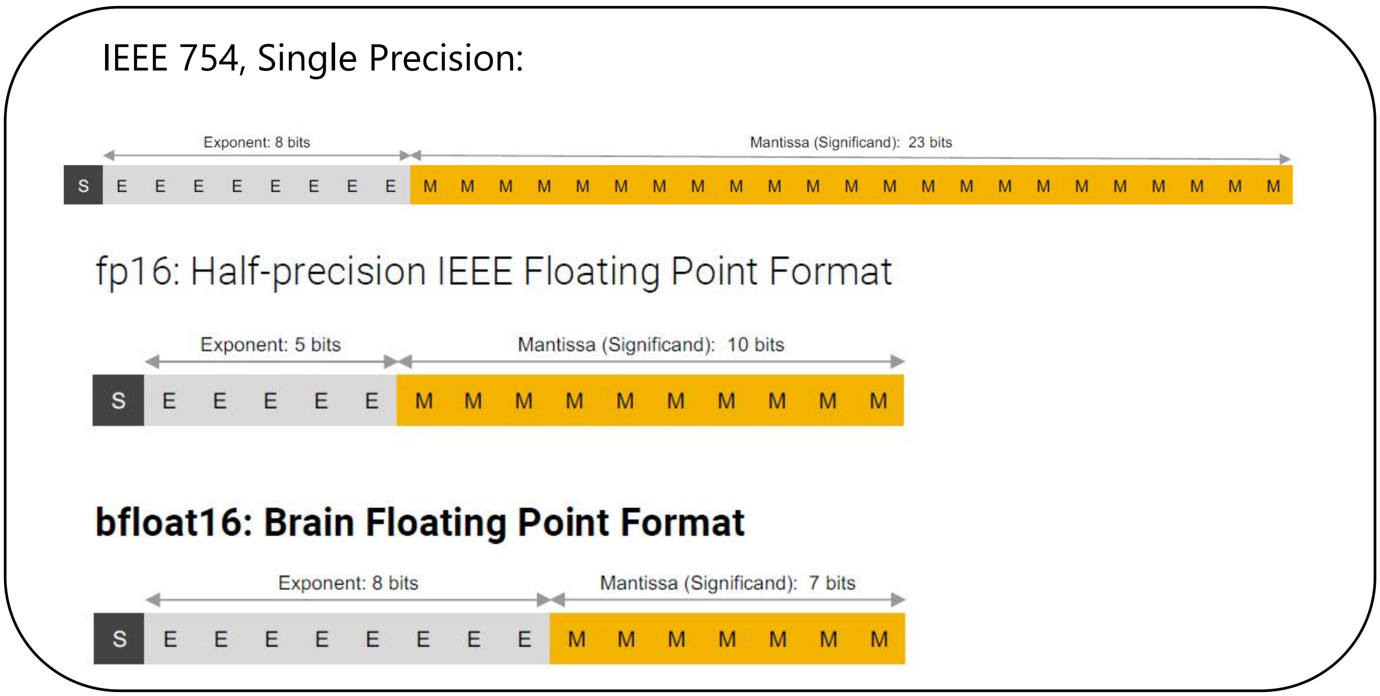
= 1.0000 0111 0000 1010 0110 0110 0 * 2^4

Thus, we don't need the guard bit.
The round bit is 0. The sticky bit is 1.
Thus the final result is:

1.0000 0111 00 * 2^(4) = 16.4375
The exponent part is 4 +15 = 19 = 1 0011
The half precision format is
0100 1100 0001 1100

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



(i) The accuracy is 2 ^(-23). The range is [2 ^(-126), 2^128].

(ii) The accuracy is 2 ^(-10). The range is [2 ^(-14), 2^16].

(iii) The accuracy is 2 ^(-7). The range is [2 ^(-126), 2 * 2^127].

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 work 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
-0.125	1 000 000	-0.125

Smallest positive normalized number	0 001 000	0.25
largest positive normalized number	0 110 111	15
Smallest positive denormalized number > 0	0 000 001	0.015625
largest positive denormalized number > 0	0 000 111	0.109375

(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.

Because when operating bitwise floating numbers, they can be overflow so that the associative law may not be applied.

For example,

let a = 0 111 111, b = 0 111 100, c = 1 111 111

if we calculate (a+b) first, the result will be overflow and we cannot get the right answer.