**NYU Tandon School of Engineering**

**Fall 2021, ECE 6913**

**Homework Assignment 3**

**Jincheng Tian Sep 30**

**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**

**B. 1111 1111 + 0000 0001**

**C. 1000 0000 + 0111 1111**

**D. 0111 0001 + 0000 1111**

**Answers:**

**For unsigned numbers**

1. 0110 1110 + 1001 1111 = 0100001101
   1. This is overflow
2. 1111 1111 + 0000 0001 = 10000000
   1. This is overflow
3. 1000 0000 + 0111 1111= 11111111
   1. Equals to 255 in decimal
4. 0111 0001 + 0000 1111 = 10000000

For signed and in 2’s complement

1. 0110 1110 + 1001 1111 = 00001101
   1. This is 110 + -97 = 13(10), it is not overflow
2. 1111 1111 + 0000 0001 = 0
   1. -1 + 1 = 0 , it is not overflow
3. 1000 0000 + 0111 1111= 11111111
   1. -128 + 127 = -1, ther is no overflow
4. 0111 0001 + 0000 1111 = 1111 1111
   1. This is 113 + 15 = + 128, there is 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 0xABhex × 0xEFhex using shifts and adds/subtracts. Assume both inputs**

**are 8-bit unsigned integers.**

**Answer:**

**0xABhex = 171(decimal) = 1010 1011**

**0xEFhex = 239(decimal) = 1110 1111**

**In decimal: 171 \* 239 = 40869 = 9FA5 (hex)**

**Since 0xABhex = 171(decimal) = 2^ 8 + 2^5 + 2 ^3 + 2^1 + 2 ^0**

**We could shift 0xEF to the left by 7 places to make it 111011110000000 = 7780(hex)**

**Append one that 0xEF shift left by 5 places 11101 1110 0000 = 1DE0(hex)**

**Then add the one that shifted left 3 places which is 011101111000 = 778(hex)**

**Then add the one that shifted one place which is 000011101 1110 = 1DE(hex)**

**Then add 0xEF**

**Add these up, we get 0x9FA5(hex) equal to the mutlple result.**

**Therefore， we use 5 shifts and 4 adds operation**

**3. What decimal number does the 32-bit pattern 0×DEADBEEF represent if it is a**

**floating-point number? Use the IEEE 754 standard**

**Answer:**

0×DEADBEEF(hex) = 3735928559 （10）

In binary, it is 011011110101011011011111011101111

With the IEEE 754 standard, it is 1 10111101 01011011011111011101111

Exponent field = 1011 1101 = DB, which is 189 in decimal

The exponent value would be E – 127 = 62

Therefore, the number would be

1.01011011011111011101111 x 2^ 62 = 6.259853398708 \* 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:

78.75 = 78.75 \* 1 = 78.75 ^ 10 ^ 0 = 1001110.11000000 \* 2 ^ 0

Shift the binary point by six places to the left we have 1.00111011 \* 2 ^ 6

The exponent value would be 127 + 6 = 133

Hence the representation of 78.75 in IEEEE 754 single precision format would be

0x429D8000

The double precision format would be that:

Exponent value is 1023 + 6 = 1029 (10) = 10000000101 (binary)

The final representation would be 0 100 0000 0101 0011 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 = 0x4053B00000000000

**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).**

**Using the similar methodology, 78.75(10) is 01001110.11000000 in decimal and 4E.C0 (hex)**

**Shift right the number by 1 hex digit which si four bits per time until the leftmost digit is 0.4EC0 \* 16^2**

**Exponent value = bias + 2 = 62 + 1 = 66**

**66 = 1000010**

**Therefore , the single precision IBM format for decimal number 78.75 would be01000 010 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.**

**(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.**

**Answers:**

* 1. **−1.3625 ×10^ −1 = -0.13625 \* 10 ^ 0**

**In binary way, it is =0.0010001100 \* 2 ^ 0**

**To normalize, we need to shift the number to the right by three bits, which make it -1.00011 \* 2 ^ -3**

**Sign is negative,**

**The bias is 2 ^ 5 -1 = 15**

**Exponent value is field – bias = -3**

**Hence we get that field = -3 + 15 = 12**

**Hence, we get that 16 bit representation is 101100 00011 00000**

* 1. **To get the sum of. 1.6125 \* 10 ^ 1 + 3.150390625 ×10^ −1**

**We need to know that 1.6125 \* 10 ^ 1 = 10000.001 = 10000001 \* 2 ^4**

**3.150390625 \* 10 ^ -1 = 0.3150390625 = 0.010100001010 = 1.0100001010 \* 2 ^ -2**

**Shift B to the same exponents would be that 0.0000010100001010 \* 2 ^ 4**

**After rounding, we can get that the sum would be 1.0000011100 \* 2 ^4 = 16.4375**

**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**

**for the IEEE 754 single precision,**

**Range is ±1.18×10-38 to ±3.4×1038**

**For Relative accuracy,**

**machine epsilon = 2-23 = 1.19\*10-7 and**

**precision=7**

**for IEEE 754 – 2008**

**the Range is ±5.96×10-8 to ±65504**

**For Relative accuracy,**

**machine epsilon = 2-10 = 9.76\*10-4 and**

**precision = 4**

**for bfloat16**

**Range is ±1.18×10-38 to ±3.4×1038**

**For Relative accuracy,**

**machine epsilon = 2-7 = 7.81\*10-3 and**

**precision = 3**

**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**

**23-1 – 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 | 1 000 100 | -0.125 |
| Smallest positive normalized number | 0 000 010 | 0.25 |
| A smallest positive denormalized number | 0 000 100 | 0.125 |
| Largest positive number | 0 111 111 | 15 |

**(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.**

**Suppose that**

**A : 1110 111**

**B : 0110 111**

**C: 0000001**

**Hence, we have a + b + c = C**