EE 3755

Computer Arithmetic

Handout # 5

Floating Point Arithmetic: Addition, Subtraction.

Here we will show how floating point addition and subtraction can take place. We will assume binary FLP arithmetic ($r=2$). All our FLP numbers will be consisting of normalized fractions and biased exponents.

Floating point addition and subtraction

The problem is defined as follows: Consider two binary ($r=2$) FLP numbers $A_1 = (-1)^{s_1} \times f_1 \times 2^{e_1}$ and $A_2 = (-1)^{s_2} \times f_2 \times 2^{e_2}$ where s_1 and s_2 are the sign bits of A_1 and A_2, f_1 and f_2 their n-bit fractions which are considered to be unsigned and normalized and e_1 and e_2 their m-bit exponents which are considered to be in biased form. We are interested in computing $A_3 = A_1 \pm A_2 = (-1)^{s_3} \times f_3 \times 2^{e_3}$ where s_3 is the sign bit, f_3 the n-bit unsigned normalized fraction and e_3 the m-bit biased exponent of the result A_3.

Obviously there are two cases: The case of a <u>true addition</u> and the case of a <u>true subtraction</u>. A true FLP addition will take place when adding numbers of the same sign or subtracting numbers of different signs. A true subtraction will take place when subtracting numbers of the same sign or adding numbers of different signs. In either case the correct sign bit has to be returned.

The following show the steps involved in floating point addition or subtraction:

1. Align fractions and adjust the smaller exponent

This means that the two exponents have to be made equal to the larger exponent by shifting the fraction of the FLP number corresponding to the smaller exponent by k bits to the right where

$k = e_{larger} - e_{smaller}$.

2. Add or subtract the fractions and set exponent of result to the larger exponent.

3. Postnormalize the result if necessary

This means that we have to make sure that the resulting fraction f_3 is an n-bit normalized fraction.

4. Check for exponent overflow or exp. underflow.

Here the following observations are in place:
A true addition can never result in an exponent underflow. However, a true addition can result in an exponent overflow if the two equalized exponents are equal to the maximum allowable exponent value and the addition of the two fractions creates a fraction overflow.
On the other hand a true subtraction can never result in an exponent overflow. A true subtraction, however, could sometimes create an exponent underflow since the fraction difference f_3 can have a certain number of leading zeros in which case normalizing the fraction (shifting it to the left/decrementing exp) might result in an exponent with value below the minimum allowable exponent value.

Some numerical examples of FLP additions and subtractions follow. For all exsuples normalized fractions and biased exponents are assumed.

Example 1: Consider the following two floating point numbers with 4-bit exponents in biased form and 8-bit unsigned normalized fractions:

Compute $A_3 = A_1 + A_2$. Return A_3 in a form consisting of a normalized fraction and exponent in biased form.

Solution:

1. Align fractions / adjust smaller exp.

Here we need to find the larger exponent as well as the difference $e_{larger} - e_{smaller}$. We can do all the above using a 4-bit adder. We should perform $e_1 - e_2 = e_1 + 2$'s complement of e_2

$$\begin{array}{r} 0101 \\ +\ 0111 \\ \hline 0\ 1100 \end{array}$$

$\hookrightarrow c=0 \Rightarrow e_1 - e_2 < 0$ or $e_2 > e_1$

So $e_2 > e_1$ and $e_2 - e_1$ = 2's complem. of $(1100) = (0100)_2 = (4)_{10}$.
We then have to shift the fraction of the FLP number corresponding to the smaller exponent 4 bits to the right so that both exponents become equal to the larger exp. e_2. So the fraction $f_1 = .10000101$ needs to be right-shifted by 4 bits to give $f_1' = .00001000$
Observe that a fraction underflow occured here since some non zero information was lost as a result of the 4-bit shift of f_1.

After alignment/adjustment the two FLP numbers are:

	s_1	e_2	f_1'
A_1:	0	1001	00001000

	s_2	e_2	f_2
A_2:	0	1001	11100000

2. Add fractions

Here a true addition between fractions needs to take place. $f_1' + f_2 =$

$$\begin{array}{r} .00001000 \\ +) \; .11100000 \\ \hline .11101000 \end{array}$$

So A_3:

s_3	e_3	f_3
0	1001	11101000

3. Postnormalize

No postnormalization necessary ($f_3 = .11101000$ is normal. fraction)

4. Check for exp ovf

No exponent ovf occured ($e_3 = (1001)_2 = (9)_{10} \in [0 \quad 15]$).

Example 2: Consider the following two floating point numbers:

	s_1	e_1	f_1
A_1:	0	1000	11100011

	s_2	e_2	f_2
A_2:	0	0110	10000001

Compute $A_3 = A_1 + A_2$. All fractions are normalized and all exponents biased.

Solution:

1. Align/adjust

Here the larger exponent is $e_1 = (1000)_2 = (8)_{10}$ and $e_1 - e_2 = 8 - 6 = 2$. So f_2 has to be shifted 2 bits to the right and A_2 becomes

	s_2	e_1	f_2'
A_2:	0	1000	00100000

A fraction underflow occured.

2. Add fractions

Here a true addition between f_1 and f_2' needs to take place $f_1 + f_2' = \cdot 11100011 + \cdot 00100000 = 1 \cdot 00000011$

↳ fraction overflow (result exceeds its 8-bit space).

3. Postnormalize

The summation of the fractions 1.00000011 should be shifted 1 bit to the right (to get the normalized 8-bit fraction $\cdot 10000001$) while the common exponent $(1000)_2$ should be incremented by 1. This way A_3 becomes

A_3:

s_3	e_3	f_3
0	1001	10000001

observe that the correction of the frac. overflow (namely the postnormalization) resulted in a fraction underflow.

4. Check for exp ouf

No exp. ouf. occured ($e_3 = (1001)_2 = (9)_{10} \in [0 \quad 15]$).

Example 3: Consider the following two floating point numbers

A_1:

s_1	e_1	f_1
0	1111	11000001

A_2:

s_2	e_2	f_2
0	1111	10000000

Compute $A_3 = A_1 + A_2$. All fractions should be considered normalized and all exponents biased.

Solution:

1. Align/adjust: No alignment/adjustment needed.

The two exponents are equal.

2. Add fractions : Here a true addition between f_1 and f_2 needs to take place.

$f_1 = .11000001$
$f_2 = .10000000$
+) 1.01000001
↳ fraction overflow (postnormalization needed)

3. Postnormalize and check for exp ovf.

The obtained $f_1 + f_2 = 1.01000001$ should be normalized to get $.10100000$ while exponent incrementation will give $(1111)_2 + (0001)_2 = (10000)_2$ which indicates an exponent overflow. In this case an exponent overflow flag has to be set.

- Question :

Consider the addition/subtraction between two FLP numbers. What happens if the difference between exponents $e_{larger} - e_{smaller} \geq$ fraction length ?

Answer : The result is the number with the larger exponent (of course the correct sign bit has to be returned). So if this case (of exponent diff. $\geq$ frac. length) can be detected by the FLP hardware, then no actual addition/subtr. needs to take place

The following example clarifies the above issue

Example 4 : Consider the following two floating point numbers

A_1 :

s_1	e_1	f_1
0	1100	11000011

A_2 :

s_2	e_2	f_2
0	0010	11011011

Compute $A_3 = A_1 + A_2$. All fractions should be considered normalized and all exponents biased.

Solution: Here the larger exponent is $e_1 = 12$, the smaller exponent is $e_2 = 2$ and $e_1 - e_2 = 12 - 2 = 10 > 8$ (8 is the fraction length). The exponent difference being 10 suggests that f_2 should be shifted 10 bits to the right or the number A_2 would become

A_2:

s_2	e_1	f_2'
0	1100	00000000

. Obviously $A_3 = A_1 + A_2 = (f_1 + f_2') \times 2^{e_1}$ $= A_1$ which means no actual addition is needed.

Example 5: Consider the following two floating point numbers

A_1:

s_1	e_1	f_1
0	1000	11111100

A_2:

s_2	e_2	f_2
1	1010	10010010

Compute $A_3 = A_1 + A_2$. All fractions should be considered normalized and all exponents biased.

Solution:

1. Align/adjust: Here the larger exponent is $e_2 = (1010)_2 = (10)_{10}$, the smaller exp is $e_1 = (1000)_2 = (8)_{10}$ and $e_2 - e_1 = 10 - 8 = 2$. Thus f_1 needs to be shifted 2 bits to the right and A_1 becomes

A_1:

s_1	e_2	f_1'
0	1010	00111111

2. Subtract fractions: Here, since A_1 and A_2 are of different signs and the addition $A_1 + A_2$ is desired, a true subtraction between fractions has to take place. Instead of using an 8-bit magnitude comparator to find the largest between f_1' and f_2

and an 8-bit adder to subtract the smaller from the larger fraction, we can do all the above using only an 8-bit adder.

We perform $f_1' - f_2$ = f_1' + 2's complement of f_2 =

$$\begin{array}{r} = \ 00111111 \\ +) \ 01101110 \\ \hline 0 \ 10101101 \end{array}$$

↳ $c=0 \Rightarrow f_1' - f_2 < 0 \Rightarrow f_1' < f_2$. Since f_2 is the larger fraction the number $A_3 = A_1 + A_2$ will have as a sign bit the sign bit of A_2 (negative sign). The fraction of A_3 will now be

2's complement of $(f_1' - f_2)$ = 2's complem. of (10101101)
$= (01010011)_2$. So the result will be

$A_3 = A_1 + A_2$:

1	1010	01010011

3. Postnormalize:

Since the resulting fraction has one leading zero, a 1-bit left shift of the fraction and decrementation of the exponent by one are needed. So the normalized result is A_3:

s_3	e_3	f_3
1	1001	10100110

4. Check for exp. underflow.

No exponent underflow occured $\left(e_3 = (1001)_2 = 9 \in [0\ 15]\right)$.

Example 6: Consider the following two floating point numbers

A_1:

s_1	e_1	f_1
0	0010	11110000

A_2:

s_2	e_2	f_2
1	0010	11100000

Compute $A_3 = A_1 + A_2$. All fractions should be considered normalized and all exponents biased.

Solution:

1. Align/adjust : Here the two exponents are equal so no alignment/adjustment is needed.

2. Subtract fractions: Here since $A_1 > 0$ and $A_2 < 0$ and the addition $A_1 + A_2$ is desired, a true subtraction between fractions has to take place. We perform $f_1 - f_2 = f_1 +$ 2's complement of f_2 =

$$
\begin{array}{rr}
= & 11110000 \\
+) & 00100000 \\
\hline
1 & 00010000
\end{array}
$$

↳ $c = 1 \Rightarrow f_1 - f_2 > 0 \Rightarrow f_1 > f_2$. Since f_1 is the larger fraction the result A_3 will have the same sign as A_1 (positive sign bit). The fraction of A_3 will be $f_1 - f_2 = .00010000$. Thus A_3 will be

A_3: | 0 | 0010 | 00010000 |

3. Postnormalize and check for exp. underflow:

The resulting fraction is .00010000 so postnormalization is necessary. Since the fraction has three leading zeros a 3-bit left shift of the fraction and decrementation of the exponent by three is necessary. The biased exponent (before postnormalization) is $(0010)_2 = (2)_{10}$. Decrementing it by three would mean a biased exponent of $-1 < 0$ which means that an exponent underflow occured. (Recall that the range of 4-bit biased exponents is [0 15]). Since an exponent underflow occured the result is forced to the unique FLP zero or

A_3: | s_3 | e_3 | f_3 |
| 0 | 0000 | 00000000 |