Unit 8 Binary Arithmetic

Structure:

8.1 Introduction

In this unit we focus on how various arithmetic operations like addition, subtraction, multiplication, and division of binary numbers are performed in computers. It also discusses floating point numbers and rational numbers representations.

Objectives:

By the end of Unit 8, you should be able to:

1. Compute the addition of signed and unsigned integers.
2. Compute the addition of floating point numbers
3. Compute the multiplication and division of signed and unsigned integers.

8.2 Binary Arithmetic

Inside a computer system, all operations are carried out on fixed-length binary values that represent application-oriented values. The schemes used to encode the application information have an impact on the algorithms for carrying out the operations. The unsigned (binary number system) and signed (2's complement) representations have the advantage that addition and subtraction operations have simple implementations, and that the same algorithm can be used for both representations. This note discusses arithmetic operations on fixed-length binary strings, and some issues in using these operations to manipulate information.

It might be reasonable to hope that the operations performed by a computer always result in correct answers. It is true that the answers are always correct but we must always be careful about what is meant by *correct*. Computers manipulate fixed-length binary values to produce fixed-length binary values. The computed values are *correct* according to the algorithms that are used; however, it is not always the case that the computed value is correct when the values are interpreted as representing application information. Programmers must appreciate the difference between application information and fixed-length binary values in order to appreciate when a computed value correctly represents application information!

A limitation in the use of fixed-length binary values to represent application information is that only a finite set of application values can be represented by the binary values. What happens if applying an operation on values contained in the finite set results in an answer that is outside the set? For example, suppose that 4-bit values are used to encode counting numbers, thereby restricting the set of represented numbers to 0 .. 15. The values 4 and 14 are inside the set of represented values. Performing the operation 4 + 14 should result in 18; however, 18 is outside the set of represented numbers. This situation is called *overflow*, and programs must always be written to deal with potential overflow situations.

Overflow in Integer Arithmetic

Using four bits, the range of numbers that can be represented is -8 through +7. When the result of an arithmetic operation is outside the representable range, an *arithmetic overflow* has occurred.

When adding unsigned numbers, the carry-out, *cn*, from the most significant bit position serves as the overflow indicator. However, this does not work for adding signed numbers. For example, when using 4-bit signed numbers, if we try to add the numbers +7 and +4, the output sum vector, *S*, is 1011, which is the code for -5, an incorrect result. The carry-out signal from the MSB position is 0. Similarly, if we try to add -4 and -6, we get *S* = 0110 = +6, another incorrect result, and in this case, the carry-out signal is 1. Thus, overflow may occur if both summands have the same sign. Clearly, the addition of numbers with different signs cannot cause overflow. This leads to the following conclusions:

1. Overflow can occur only when adding two numbers that have the same sign.
2. The carry-out signal from the sign-bit position is not a sufficient indicator of over- flow when adding signed numbers.

A simple way to detect overflow is to examine the signs of the two summands *X* and *Y* and the sign of the result. When both operands *X* and *Y* have the same sign, an overflow occurs when the sign of *S* is not the same as the signs of *X and Y*.

Binary Addition

The binary addition of two bits (*a* and *b*) is defined by the table:

a	*b*	*a + b*	
0	0	0	
0	1	1	carry 0
1	0	1	
1	1	0	carry 1

When adding n-bit values, the values are added in corresponding bit-wise pairs, with each carry being added to the next most significant pair of bits. The same algorithm can be used when adding pairs of unsigned or pairs of signed values.

4-Bit Example A:

$$\begin{array}{rrl} & 0\;1\;0 & \longleftarrow \text{carry values} \\ \text{value 1:} & 1\;0\;1\;1 & \multirow{2}{*}{$\Big\}$ bit-wise pairs} \\ \text{+ value 2:} & +\;\;0\;0\;1\;0 & \\ \hline \text{result:} & 1\;1\;0\;1 & \end{array}$$

Since computers are constrained to deal with fixed-width binary values, any carry out of the most significant bit-wise pair is ignored.

<u>4-Bit Example B:</u>

```
                 1 1 1 0  <------ carry values

   value 1:        1 0 1 1  }
                            }  bitwise pairs
 + value 2:      + 0 1 1 0  }
   -------         -------
   result:       1 0 0 0 1
                 ^    ^
          ignored    4-bit result
```

The binary values generated by the addition algorithm are always *correct* with respect to the algorithm, but what is the significance when the binary values are intended to represent application information? Will the operation yield a result that accurately represents the result of adding the application values?

First consider the case where the binary values are intended to represent unsigned integers (i.e. counting numbers). Adding the binary values representing two unsigned integers will give the correct result (i.e. will yield the binary value representing the sum of the unsigned integer values) providing the operation does not overflow – i.e. when the addition operation is applied to the original unsigned integer values, the result is an unsigned integer value that is inside of the set of unsigned integer values that can be represented using the specified number of bits (i.e. the result can be represented under the fixed-width constrains imposed by the representation).

<u>Reconsider 4-Bit Example A</u> (above) as adding unsigned values:

$$\begin{array}{rrr} & & 0\,1\,0 \leftarrow \text{carry values} \\ \text{value 1:} & 11_{10} & 1\,0\,1\,1 \\ \underline{+\text{ value 2:}} & \underline{+\ 2_{10}} & \underline{+\ 0\,0\,1\,0} \\ \text{result:} & 13_{10} \longrightarrow & 1\,1\,0\,1 \end{array}$$

In this case, the binary result (1101_2) of the operation accurately represents the unsigned integer sum (13) of the two unsigned integer values being added (11 + 2), and therefore, the operation did not overflow. But what about 4-Bit Example B (above)?

			1 1 1 0 $\longleftarrow$	carry values
value 1:	11_{10}		1 0 1 1	
+ value 2:	+ 6_{10}		+ 0 1 1 0	
result:	17_{10}	$\longrightarrow$	0 0 0 1	???? 11 + 6 = 1 ?????

When the values added in Example B are considered as unsigned values, then the 4-bit result (1) does not accurately represent the sum of the unsigned values (11 + 6)! In this case, the operation has resulted in overflow: the result (17) is outside the set of values that can be represented using 4-bit binary number system values (i.e. 17 is not in the set {0 , … , 15}). The result (0001_2) is *correct* according to the rules for performing binary addition using fixed-width values, but truncating the carry out of the most significant bit resulted in the loss of information that was important to the encoding being used. If the carry had been kept, then the 5-bit result (10001_2) would have represented the unsigned integer sum correctly.

But more can be learned about overflow from the above examples! Now consider the case where the binary values are intended to represent signed integers.

Reconsider 4-Bit Example A (above) as adding signed values:

			0 1 0 $\longleftarrow$	carry values
value 1:	-5_{10}		1 0 1 1	
+ value 2:	+ 2_{10}		+ 0 0 1 0	
result:	-3_{10}	$\longrightarrow$	1 1 0 1	no overflow !

In this case, the binary result (1101_2) of the operation accurately represents the signed integer sum (– 3) of the two signed integer values being added (– 5 + 2) → therefore, the operation did not overflow. What about 4-Bit Example B?

$$\begin{array}{rrrl} & & 1\,1\,1\,0 \longleftarrow & \text{carry values} \\ \text{value 1:} & -\,5_{10} & 1\,0\,1\,1 & \\ \text{+ value 2:} & +\ 6_{10} & +\ 0\,1\,1\,0 & \\ \hline \text{result:} & 1_{10} \longrightarrow & 0\,0\,0\,1 & \text{no overflow!} \end{array}$$

In this case, the result (again) represents the signed integer answer correctly, and therefore, the operation did not overflow.

Recall that in the unsigned case, Example B resulted in overflow. In the signed case, Example B did not overflow. This illustrates an important concept: overflow is interpretation dependent! The concept of overflow depends on how information is represented as binary values. Different types of information are encoded differently, yet the computer performs a specific algorithm, regardless of the possible interpretations of the binary values involved. It should not be surprising that applying the same algorithm to different interpretations may have different overflow results.

Subtraction

The binary subtraction of two bits (*a* and *b*) is defined by the table:

a	*b*	*a* – *b*	
0	0	0	
1	0	1	borrow 0
1	1	0	
0	1	1	borrow 1

When subtracting n-bit values, the values are subtracted in corresponding bit-wise pairs; with each borrow rippling down from the more significant bits as needed. If none of the more significant bits contains a 1 to be borrowed, then 1 may be borrowed into the most significant bit.

<u>4-Bit Example C:</u>

must borrow from second digit

$$\begin{array}{rr} & 1\,0\,1\,0 \\ - & 0\,0\,0\,1 \end{array}$$

becomes: Interpretations

$$\begin{array}{rrrl} 0 & \text{unsigned} & \text{signed} & \text{no overflow in either case} \\ 1\,0\,\cancel{1}\,{}^{1}0 & 10 & -6 & \\ -\;0\,0\,0\,1 & -\;\underline{1} & \underline{-1} & \\ \hline 1\,0\,0\,1 & 9 & -7 & \end{array}$$

<u>4-Bit Example D:</u>

must borrow from above most significant digit

$$\begin{array}{rr} & 0\,0\,0\,1 \\ - & 1\,1\,1\,1 \end{array}$$

becomes: Interpretations

$$\begin{array}{lrrrl} & 1\;\;1 & \text{unsigned} & \text{signed} & \text{overflow in unsigned case} \\ \text{borrow} & {}^{1}\cancel{0}\,{}^{1}\cancel{0}\,{}^{1}0\;\;1 & 1 & 1 & \text{no overflow in signed case} \\ \text{from} & -\;1\;\;1\;\;1\;\;1 & -15 & -\;\;-1 & \\ \text{above} & \overline{0\;\;0\;\;1\;\;0} & \overline{\;\;2} & \overline{\;\;2} & \end{array}$$

Most computers apply the mathematical identity:

$$a - b = a + (-b)$$

to perform subtraction by negating the second value (b) and then adding. This can result in a saving in transistors since there is no need to implement a subtraction circuit.

Another Note on Overflow

Are there easy ways to decide whether an addition or subtraction results in overflow? Yes but we should be careful that we understand the concept, and don't rely on memorizing case rules that allow the occurrence of overflow to be identified.

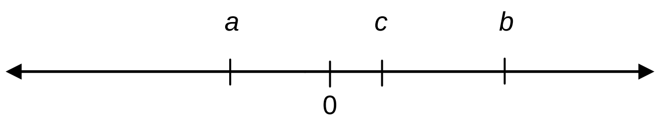
For unsigned values, a carry out of (or a borrow into) the most significant bit indicates that overflow has occurred.

For signed values, overflow has occurred when the sign of the result is impossible for the signs of the values being combined by the operation. For example, overflow has occurred if:

- Two positive values are added and the sign of the result is negative
- A negative value is subtracted from a positive value and the result is negative (a positive minus a negative is the same as a positive plus a positive, and should result in a positive value, i.e. $a - (-b) = a + b$)

These are just two examples of some of the possible cases for signed overflow.

Note that it is not possible to overflow for some signed values. For example, adding a positive and a negative value will never overflow. To convince you of why this is the case, picture the two values on a number line as shown below. Suppose that *a* is a negative value, and *b* is a positive value. Adding the two values, *a* + *b* will result in *c* such that *c* will always lie between *a* and *b* on the number line. If *a* and *b* can be represented prior to the addition, then *c* can also be represented, and overflow will never occur.



Multiplication

Multiplication is a slightly more complex operation than addition or subtraction. Multiplying two n-bit values together can result in a value of up to 2n-bits. To help to convince you of this, think about decimal numbers: multiplying two 1-digit numbers together result in a 1- or 2-digit result, but cannot result in a 3-digit result (the largest product possible is 9 x 9 = 81). What about multiplying two 2-digit numbers? Does this extrapolate to n-digit numbers? To further complicate matters, there is a reasonably simple algorithm for multiplying binary values that represent unsigned integers, but the same algorithm cannot be applied directly to values that represent signed values (this is different from addition and subtraction where the same algorithms can be applied to values that represent unsigned or signed values!).

Overflow is not an issue in n-bit unsigned multiplication, proving that 2n-bits of results are kept.

Now consider the multiplication of two unsigned 4-bit values *a* and *b*. The value *b* can be rewritten in terms of its individual digits:

$b = b_3 * 2^3 + \mathrm{b2} * 2^2 + b_1 * 2^1 + b_0 * 2^0$

Substituting this into the product $a * b$ gives:

$a * (b_3 * 2^3 + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0)$

Which can be expanded into:

$a * b_3 * 2^3 + a * b_2 * 2^2 + a * b_1 * 2^1 + a * b_0 * 2^0$

The possible values of bi are 0 or 1. In the expression above, any term where $b_i = 0$ resolves to 0 and the term can be eliminated. Furthermore, in any term where $b_i = 1$, the digit b_i is redundant (multiplying by 1 gives the same value, and therefore the digit b_i can be eliminated from the term. The resulting expression can be written and generalized to n-bits:

$$a * b = \sum_{i=0}^{n-1} a * 2^i \quad \text{where } b_i = 1$$

This expression may look a bit intimidating, but it turns out to be reasonably simple to implement in a computer because it only involves multiplying by 2 (and there is a trick that lets computers do this easily!). Multiplying a value by 2 in the binary number system is analogous to multiplying a value by 10 in the decimal number system. The result has one new digit: a 0 is injected as the new least significant digit, and all of the original digits are shifted to the left as the new digit is injected.

Think in terms of a decimal example, say: $37 * 10 = 370$. The original value is 37 and the result is 370. The result has one more digit than the original value, and the new digit is a 0 that has been injected as the least significant digit. The original digits (37) have been shifted one digit to the left to admit the new 0 as the least significant digit.

The same rule holds good for multiplying by 2 in the binary number system. For example:

$101_2 * 2 = 1010_2$. The original value of 5 (101_2) is multiplied by 2 to give 10 (1010_2). The result can be obtained by shifting the original value left one digit and injecting a 0 as the new least significant digit.

The calculation of a product can be reduced to summing terms of the form $a * 2^i$. The multiplication by 2^i can be reduced to shifting left i times! The shifting of binary values in a computer is very easy to do, and as a result, the calculation can be reduced to a series of shifts and adds.

Unsigned Integer Multiplication: Straightforward Method

To compute: $a * b$

Where

- Register A contains $a = a_{n-1}a_{n-2}...a_1a_0$
- Register B contains $b = b_{n-1}b_{n-2}...b_1b_0$

and where register P is a register twice as large as A or B

Simple Multiplication Algorithm: Straightforward Method

Steps:

1. If LSB(A) = 1, then set P_{upper} to $P_{upper} + b$
2. Shift the register A right
3. Using zero sign extension (for unsigned values)
4. Forcing the LSB(A) to fall off the lower end
5. Shift the double register P right
6. Using zero sign extension (for unsigned values)
7. Pushing the LSB(P_{upper}) into the MSB(P_{lower})

After *n* times (for *n*-bit values),

The full contents of the double register P = $a * b$

Example of Simple Multiplication Algorithm (Straightforward Method)

Multiply $b = 2 = 0010_2$ by $a = 3 = 0011_2$

(Answer should be $6 = 0110_2$)

P	A	B	Comments
0000 0000	0011	0010	Start: A = 0011; B = 0010; P = (0000, 0000)
0010 0000	0011	0010	LSB(A) = l ==> Add *b to P*
0010 0000	0001	0010	Shift A right
0001 0000	0001	0010	Shift P right
0011 0000	0001	0010	LSB(A) = l ==> Add *b to P*
0011 0000	0000	0010	Shift A right
0001 1000	0000	0010	Shift P right
0001 1000	0000	0010	LSB(A) = 0 ==> Do nothing
0001 1000	0000	0010	Shift A right
0000 1100	0000	0010	Shift P right
0000 1100	0000	0010	LSB(A) = 0 ==> Do nothing
0000 1100	0000	0010	Shift A right
0000 0110	0000	0010	Shift P right
0000 0110	0000	0010	Done: P is product

Unsigned Integer Multiplication: A More Efficient Method

To compute: $a * b$

where

- Register A contains $a = a_{n-1}a_{n-2}...a_1a_0$
- Register B contains $b = b_{n-1}b_{n-2}...b_1b_0$

  and where register P is connected to register A to form a register twice as large as A or B and register P is the upper part of the double register (P, A)

  Simple Multiplication Algorithm: A More Efficient Method

Steps:

1. If LSB(A) = 1, then set P to $P_{upper} + b$
2. Shift the double register (P, A) right
3. Using zero sign extension (for unsigned values)

4. Forcing the LSB(A) to fall off the lower end
5. Pushing the LSB(P_{upper}) into MSB(P_{lower}) = MSB(A)

After *n* times (for *n*-bit values),

The contents of the double register (P, A) = *a* * *b*

Example of Simple Multiplication Algorithm (A More Efficient Method)

Multiply *b* = 2 = 0010_2 by *a* = 3 = 0011_2

(Answer should be 6 = 0110_2)

P	A	B	Comments
0000	0011	0010	Start: A = 0011; B = 0010; P = (0000, 0000)
0010	0011	0010	LSB(A) = l ==> Add *b to P*
0001	0001	0010	Shift (P, A) right
0011	0001	0010	LSB(A) = l ==> Add *b to P*
0001	1000	0010	Shift (P, A) right
0001	1000	0010	LSB(A) = 0 ==> Do nothing
0000	1100	0010	Shift (P, A) right
0000	1100	0010	LSB(A) = 0 ==> Do nothing
0000	0110	0010	Shift (P, A) right
0000	0110	0010	Done: P is product

By combining registers P and A, we can eliminate one extra shift step per iteration.

Positive Integer Multiplication

Shift-Add Multiplication Algorithm

(Same as Straightforward Method above)

- For unsigned or positive operands
- Repeat the following steps *n* times:
  1. If LSB(A) = 1, then set P_{upper} to P_{upper} + *b* else set P_{upper} to P_{upper} + 0
  2. Shift the double register (P, A) right

3. Using zero sign extension (for positive values)
4. Forcing the LSB(A) to fall off the lower end

Signed Integer Multiplication

Simplest:

1. Convert negative values of *a* or *b* to positive values
2. Multiply both positive values using one of the two algorithms above
3. Adjust sign of product appropriately Alternate: Booth algorithm

Introduction to Booth's Multiplication Algorithm

A powerful algorithm for signed-number multiplication is the booth algorithm. It generates a 2n-bit product and treats both positive and negative numbers uniformly.

Consider a positive binary number containing a run of ones e.g., the 8-bit value: 00011110. Multiplying by such a value implies four consecutive additions of shifted multiplicands

```
            00010100                       20
          x 00011110                     x 30
              00000000                      00
             00010100      first          60
            00010100       second
           00010100       third
          00010100       fourth
         00000000
        00000000
       00000000
    ----------------                   -----
    0000001001011000                     600
```

Now 00011110 can be expressed as a difference of powers of two:

```
   00100000          32
 – 00000010        –  2
 __________      ________
  00011110           30
```

This means that the same multiplication can be obtained using only two additions:

- + 2^5 x multiplicand of 00010100
- - 2^1 x multiplicand of 00010100

Since the 1s complement of 00010100 is 11101011,

Then -00010100 is 11101100 in 2s complement

In other words, using sign extension and ignoring the overflow,

```
           00010100                        00010100
         x 00011110                      x 00011110
          _________                      __________
           00000000                        00000000
        – 00010100    – 2 x     111111111101100
         00000000                        00000000
        00000000                        00000000
       00000000                        00000000
  + 00010100      + 32 x        00010100
     00000000                        00000000
    00000000                        00000000
 ________________              __________________
 0000001001011000               1|0000001001011000
```

Booth's Recoding Table

Booth recodes the bits of the multiplier *a* according to the following table:

a_i	a_{i-1}	**Recoded** a_i
0	0	0
0	1	+1
1	0	-1
1	1	0

Always assume that there is a zero to the right of the multiplier

i.e. that a_{-1} is zero

so that we can consider the LSB (= a_0)

and an imaginary zero bit to its right

So, if *a* is 00011110 ,

the recoded value of *a* is

0 0 +1 0 0 0 -1 0

Example of Booth's Multiplication

A. Multiply 00101101 (*b*) by 00011110 (*a*) using normal multiplication

```
                 00101101                        45
            x 00011110                         x 30
           ----------                          -----
            00000000                             00
           00101101       first                135
             00101101        second
            00101101         third
           00101101          fourth
          00000000
         00000000
        00000000
     ----------------                     ----------
     0000010101000110                           1350
```

B. Multiply 00101101 (*b*) by 00011110 (*a*) using Booth's multiplication

Recall the recoded value of 00011110 (*a*) is 0 0 +1 0 0 0 -1 0

The 1s complement of 00101101 (*b*) is 11010010

The 2s complement of 00101101 (*b*) is 11010011

Hence, using sign extension and ignoring overflow:

```
            00101101                     45
              x 00100010                x 30
                  __________          __________
                    00000000                  00
        111111111010011    - b         135
              00000000
             00000000
            00000000
           00101101     + b
          00000000
         00000000
 _________________                 __________
1|0000010101000110                     1350
```

Booth's Multiplication Algorithm

Booth's algorithm chooses between + *b* and - *b* depending on the two current bits of *a* .

Algorithm:

1. Assume the existence of bit $a_{-1} = 0$ initially
2. Repeat *n* times (for multiplying two *n*-bit values):
   a. At step *i*:
   a. If $a_i = 0$ and $a_{i-1} = 0$, add 0 to register P
   b. If $a_i = 0$ and $a_{i-1} = 1$, add *b* to register P i.e. treat a_i as +1
   c. If $a_i = 1$ and $a_{i-1} = 0$, subtract *b* from register P i.e. treat a_i as -1
   d. If $a_i = 1$ and $a_{i-1} = 1$, add 0 to register P
   b. Shift the double register (P, A) right one bit with sign extension

Justification of Booth's Multiplication Algorithm

Consider the operations described above:

a_i	a_{i-1}	$a_{i-1} - a_i$	**Operation**
0	0	0	Add 0x*b* to P
0	1	+1	Add +1x*b* to P
1	0	-1	Add -1x*b* to P
1	1	0	Add 0x*b* to P

Equivalently, the algorithm could state

2. Repeat *n* times (for multiplying two *n*-bit values): At step *i*, add $(a_{i-1} - a_i) x b$ to register P

The result of all *n* steps is the sum:

$(a_{-1} – a_0) x 2^0 x b$

$+ (a_0 – a_1) x 2^1 x b$

$+ (a_1 - a_2) x 2^2 x b$ ...

$+ (a_{n-3} - a_{n-2}) x 2^{n-2} x b$

$+ (a_{n-2} - a_{n-1}) x 2^{n-1} x b$

where a_{-1} is assumed to be 0

This sum is equal to $b x SUM_{i=0}^{n-1} ((a_{i-1} - a_i) x 2^i)$

$= b(-2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + ... + 2a_1 + a_0) + ba_{-1}$

$= b(-2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + ... + 2a_1 + a_0)$

since $a_{-1} = 0$

Now consider the representation of *a* as a 2s complement

It can be shown to be the same as

$-2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + ... + 2a_1 + a_0$

where a_{n-1} represents the sign of *a*

- If $a_{n-1} = 0$, *a* is a positive number
- If $a_{n-1} = 1$, *a* is a negative number

Example:

- Let w = 1011_2
- Then w = $-1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = -8 + 2 + 1 = -5$

Thus the sum above,

$b(-2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + ... + 2a_1 + a_0)$,

is the same as the 2s complement representation of *b x a*

Example of Booth's Multiplication Algorithm (Positive Numbers)

Multiply *b* = 2 = 0010_2 by *a* = 6 = 0110_2

(Answer should be 12 = 1100_2)

The 2s complement of *b* is 1110

P	A	B	Comments
0000	0110 [0]	0010	Start: $a_{-1} = 0$
0000	0110 [0]	0010	00 ==> Add 0 to P
0000	0011 [0]	0010	Shift right
1110	0011 [0]	0010	10 ==> Subtract *b* from P
1111	0001 [1]	0010	Shift right
1111	0001 [1]	0010	11 ==> Add 0 to P
1111	1000 [1]	0010	Shift right
0001	1000 [1]	0010	01 ==> Add *b* to P
0000	1100 [0]	0010	Shift right
0000	1100 [0]	0010	Done: (P, A) is product

Example of Booth's Multiplication Algorithm (Negative Numbers)

Multiply *b* = -5 = $-(0101)_2$ = 1011_2 by *a* = -6 = $-(0110)_2$ = 1010_2

(Answer should be 30 = 11110_2)

The 2s complement of *b* is 1110

P	A	B	Comments
0000	1010 [0]	1011	Start: $a_{-1} = 0$
0000	1010 [0]	1011	00 ==> Add 0 to P
0000	0101 [0]	1011	Shift right

0101	0101 [0]	1011	10 ==> Subtract *b* from P
0010	1010 [1]	1011	Shift right
1101	1010 [1]	1011	01 ==> Add *b* to P
1110	1101 [0]	1011	Shift right
0011	1101 [0]	1011	10 ==> Subtract *b* from P
0001	1110 [1]	1011	Shift right
0001	1110 [1]	1011	Done: (P, A) is product

Advantages and Disadvantages of Booth's Algorithm

Advantages:

Handles positive and negative numbers uniformly

Efficient when there are long runs of ones in the multiplier

Disadvantages:

Average speed of algorithm is about the same as with the normal multiplication algorithm.

Worst case operates at a slower speed than the normal multiplication algorithm.

Division

Terminology: dividend ÷ divisor = quotient & remainder

The implementation of division in a computer raises several practical issues:

- For integer division there are two results: the quotient and the remainder.
- The operand sizes (number of bits) to be used in the algorithm must be considered (i.e. the sizes of the dividend, divisor, quotient and remainder).
- Overflow is not an issue in unsigned multiplication, but is a concern with division.
- As with multiplication, there are differences in the algorithms for signed vs. unsigned division.

Recall that multiplying two n-bit values can result in a 2n-bit value. Division algorithms are often designed to be symmetrical with this by specifying:

- the dividend as a 2n-bit value
- the divisor, quotient and remainder as n-bit values

Once the operand sizes are set, the issue of overflow may be addressed. For example, suppose that the above operand sizes are used, and that the dividend value is larger than a value that can be represented in n bits (i.e. $2^n - 1 <$ dividend). Dividing by 1 (divisor = 1) should result with quotient = dividend; however, the quotient is limited to n bits, and therefore is incapable of holding the correct result. In this case, overflow would occur.

Sign Extension / Zero Extension

When loading a 16-bit value into a 32-bit register,

How is the sign retained?

By loading the 16-bit value into the lower 16 bits of the 32-bit register and

By duplicating the MSB of the 16-bit value throughout the upper 16 bits of the 32-bit register

This is called sign extension

If instead, the upper bits are *always* set to zero,

It is called zero extension

Unsigned Integer Division

To compute: a / b

where

- Register A contains $a = a_{n-1}a_{n-2}...a_1a_0$
- Register B contains $b = b_{n-1}b_{n-2}...b_1b_0$

and where register P is connected to register A to form a register twice as large as A or B and register P is the upper part of the double register as done in many of the Multiplication methods.

Simple Unsigned Division

– for unsigned operands

Steps:

1. Shift the double register (P, A) one bit left
2. Using zero sign extension (for unsigned values)
3. And forcing the MSB(P) to fall off the upper end
4. Subtract *b* from P
5. If result is negative, then set LSB(A) to 0 else set LSB(A) to 1
6. If result is negative, set P to P + *b*

After repeating these steps *n* times (for *n*-bit values), the contents of register A = *a / b*, and the contents of register P = *remainder (a / b)*

This is also called restoring division

Restoring Division Example

Divide 14 = 1110_2 by 3 = 0011_2

P	A	B	Comments
+00000	1110	0011	Start
+00001	1100	0011	Shift left
-00010	1100	0011	Subtract *b*
+00001	1100	0011	Restore
+00001	1100	0011	Set LSB(A) to 0
+00011	1000	0011	Shift left
+00000	1000	0011	Subtract *b*
+00000	1001	0011	Set LSB(A) to 1
+00001	0010	0011	Shift left
-00010	0010	0011	Subtract *b*

+00001	0010	0011	Restore
+00001	0010	0011	Set LSB(A) to 0
+00010	0100	0011	Shift left
-00001	0100	0011	Subtract *b*
+00010	0100	0011	Restore
+00010	0100	0011	Set LSB(A) to 0
+00010	0100	0011	Done

Restoring versus Non-restoring Division

Let r = the contents of (P, A)

At each step, the restoring algorithm computes

P = $(2r - b)$

If $(2r - b) < 0$,

then

- (P, A) = $2r$ (restored)
- (P, A) = $4r$ (shifted left)
- (P, A) = $4r - b$ (subtract b for next step)

  If $(2r - b) < 0$

and there is no restoring,

then

- (P, A) = $2r - b$ (restored)
- (P, A) = $4r - 2b$ (shifted left)
- (P, A) = $4r - b$ (add b for next step)

Non-restoring Division Algorithm

Steps:

1. If P is negative,
   a) Shift the double register (P, A) one bit left
   b) Add b to P

2. Else if P is not negative,
   a) Shift the double register (P, A) one bit left
   b) Subtract *b* from P
3. If P is negative, then set LSB(A) to 0 else set LSB(A) to 1

   After repeating these steps *n* times (for *n*-bit values),

   if P is negative, do a final restore

   i.e. add *b* to P

   Then

   the contents of register A = *a / b*, and

   the contents of register P = *remainder (a / b)*

Non-restoring Division Example

Divide 14 = 1110_2 by 3 = 0011_2

P	A	B	Comments
00000	1110	0011	Start
00001	1100	0011	Shift left
11110	1100	0011	Subtract *b*
11110	1100	0011	P negative; Set LSB(A) to 0
11101	1000	0011	Shift left
00000	1000	0011	P negative; Add *b*
00000	1001	0011	P positive; Set LSB(A) to 1
00001	0010	0011	Shift left
11110	0010	0011	Subtract *b*
11110	0010	0011	P negative; Set LSB(A) to 0
11100	0100	0011	Shift left
11111	0100	0011	P negative; Add *b*
11111	0100	0011	P negative; Set LSB(A) to 0
00010	0100	0011	P=remainder negative; Need final restore
00010	0100	0011	Done

Division by Multiplication

There are numerous algorithms for division many of which do not relate as well to the division methods learned in elementary school One of these is *Division by Multiplication* It is also known as *Goldschmidt's algorithm*.

If we wish to divide N by D, then we are looking for Q such that Q = N/D

Since N/D is a fraction, we can multiply both the numerator and denominator by the same value, *x*, without changing the value of Q. This is the basic idea of the algorithm. We wish to find some value *x* such that D*x* becomes close to 1 so that we only need to compute N*x* To find such an *x*, first scale D by shifting it right or left so that 1/2 <= D < 1

Let s = #shifts be positive if to the left

and negative, otherwise

Call this D' (= D · 2^s)

Now compute *x* such that *x* = 1 - D'

Call this Z

Notice that 0 < Z <= 1/2

since Z = 1 - D'

Furthermore D' = 1 - Z

Then

$$
\begin{aligned}
Q &= N / D \\
&= N (1+Z) / D (1+Z) \\
&= 2^s N (1+Z) / 2^s D (1+Z) \\
&= 2^s N (1+Z) / D' (1+Z) \\
&= 2^s N (1+Z) / (1-Z) \cdot (1+Z) \\
&= 2^s N (1+Z) / (1-Z^2)
\end{aligned}
$$

Similarly,

$$\begin{aligned} Q &= N / D \\ &= 2^s N (1+Z) / (1-Z^2) \\ &= 2^s N (1+Z) (1+Z^2) / (1-Z^2) (1+Z^2) \\ &= 2^s N (1+Z) (1+Z^2) / (1-Z^4) \end{aligned}$$

And,

$$\begin{aligned} Q &= N / D \\ &= 2^s N (1+Z) (1+Z^2) / (1-Z^4) \\ &= 2^s N (1+Z) (1+Z^2) (1+Z^4) / (1-Z^4) (1+Z^4) \\ &= 2^s N (1+Z) (1+Z^2) (1+Z^4) / (1-Z^8) \end{aligned}$$

continuing to

$$\begin{aligned} Q &= N / D \\ &= 2^s N (1+Z) (1+Z^2) (1+Z^4) \cdot \cdot \cdot (1+Z^{2n-1}) / (1-Z^{2n}) \end{aligned}$$

Since $0 < Z <= 1/2$,

Z^i goes to zero as i goes to infinity. This means that the denominator, $1-Z^{2n}$, goes to 1 as n gets larger. Since the denominator goes to 1, we need only compute the numerator in order to determine (or approximate) the quotient, Q

So, $Q = 2^s N (1+Z) (1+Z^2) (1+Z^4) \cdot \cdot \cdot (1+Z^{2n-1})$

Examples of Division by Multiplication

Example 1:
Let's start with a simple example in the decimal system
Suppose N = 1.8 and D = 0.9
(Clearly, the answer is 2)
Since D = 0.9, it does not need to be shifted
so the number of shifts, s, is zero

And since we are in the decimal system,

we use 10^s in the equation for Q

Furthermore, $Z = 1 - D = 1 - 0.9 = 0.1$

For $n = 0$,

$$\begin{aligned} Q &= N / D \\ &= 1.8 / 0.9 \\ &= 10^0 \cdot 1.8 \\ &= 1.8 \end{aligned}$$

For $n = 1$,

$$\begin{aligned} Q &= N / D \\ &= 1.8 / 0.9 \\ &= 10^0 \cdot 1.8 (1 + Z) \\ &= 1.8 (1 + 0.1) \\ &= 1.8 \cdot 1.1 \\ &= 1.98 \end{aligned}$$

For $n = 2$,

$$\begin{aligned} Q &= N / D \\ &= 1.8 / 0.9 \\ &= 10^0 \cdot 1.8 (1 + Z) (1 + Z^2) \\ &= 1.8 (1 + 0.1) (1 + 0.01) \\ &= 1.8 \cdot 1.1 \cdot 1.01 \\ &= 1.98 \cdot 1.01 \\ &= 1.9998 \end{aligned}$$

For $n = 3$,

$$\begin{aligned} Q &= N / D \\ &= 1.8 / 0.9 \\ &= 10^0 \cdot 1.8 (1 + Z) (1 + Z^2) (1 + Z^4) \\ &= 1.8 (1 + 0.1) (1 + 0.01) (1 + 0.0001) \\ &= 1.8 \cdot 1.1 \cdot 1.01 \cdot 1.0001 \end{aligned}$$

= $1.98 \cdot 1.01 \cdot 1.0001$

= $1.9998 \cdot 1.0001$

= 1.99999998

And so on, getting closer and closer to 2.0 with each increase of n

8.3 Floating Point Numbers

Instead of using the obvious representation of rational numbers presented in the previous section, most computers use a different representation of a subset of the rational numbers. We call these numbers *floating-point* numbers.

Floating-point numbers use inexact arithmetic, and in return require only a fixed-size representation. For many computations (so-called scientific computations, as if other computations weren't scientific) such a representation has the great advantage that it is fast, while at the same time usually giving adequate precision.

There are some (sometimes spectacular) exceptions to the "adequate precision" statement in the previous paragraph, though. As a result, an entire discipline of applied mathematics, called numerical analysis, has been created for the purpose of analyzing how algorithms behave with respect to maintaining adequate precision, and of inventing new algorithms with better properties in this respect.

The basic idea behind floating-point numbers is to represent a number as *mantissa* and an *exponent*, each with a fixed number of bits of precision. If we denote the mantissa with *m* and the exponent with *e*, then the number thus represented is $m * 2^e$.

Again, we have a problem that a number can have several representations. To obtain a canonical form, we simply add a rule that *m* must be greater than or equal to 1/2 and strictly less than 1. If we write such a mantissa in binal (analogous to *decimal*) form, we always get a number that starts with 0.1. This initial information therefore does not have to be represented, and we represent only the remaining “binals”.

The reason floating-point representations work well for so-called scientific applications are that we more often need to *multiply* or *divide* two numbers. Multiplication of two floating-point numbers is easy to obtain. It suffices to multiply the mantissas and add the exponents. The resulting mantissa might be smaller than 1/2, in fact, it can be as small as 1/4. In this case, the result needs to be canonicalized. We do this by shifting the mantissa left by one position and subtracting one from the exponent. Division is only slightly more complicated. Notice that the imprecision in the result of a multiplication or a division is only due to the imprecision in the original operands. No additional imprecision is introduced by the operation itself (except possibly 1 unit in the least significant digit). Floating-point addition and subtraction do not have this property.

To add two floating-point numbers, the one with the smallest exponent must first have its mantissa shifted right by *n* steps, where *n* is the difference of the exponents. If *n* is greater than the number of bits in the representation of the mantissa, the second number will be treated as 0 as far as addition is concerned. The situation is even worse for subtraction (or addition of one positive and one negative number). If the numbers have roughly the same absolute value, the result of the operation is roughly zero, and the resulting representation may have no correct significant digits.

The two's complement representation that we have mentioned above is mostly useful for addition and subtraction. It only complicates things for multiplication and division. For multiplication and division, it is better to use a representation with sign + absolute value. Since multiplication and division is more common with floating-point numbers, and since they result in multiplication and division of the mantissa, it is more advantageous to have the mantissa represented as sign + absolute value. The exponents are added, so it is more common to use two's complement (or some related representation) for the exponent.

Usually, computers manipulate data in chunks of 8, 16, 32, 64, or 128 bits. It is therefore useful to fit a single floating-point number with both mantissa and exponent in such a chunk. In such a chunk, we need to have room for the sign (1 bit), the mantissa, and the exponent. While there are many different ways of dividing the remaining bits between the mantissa and the exponent, in practice most computers now use a norm called IEEE, which mandates the formats as shown in figure 8.1

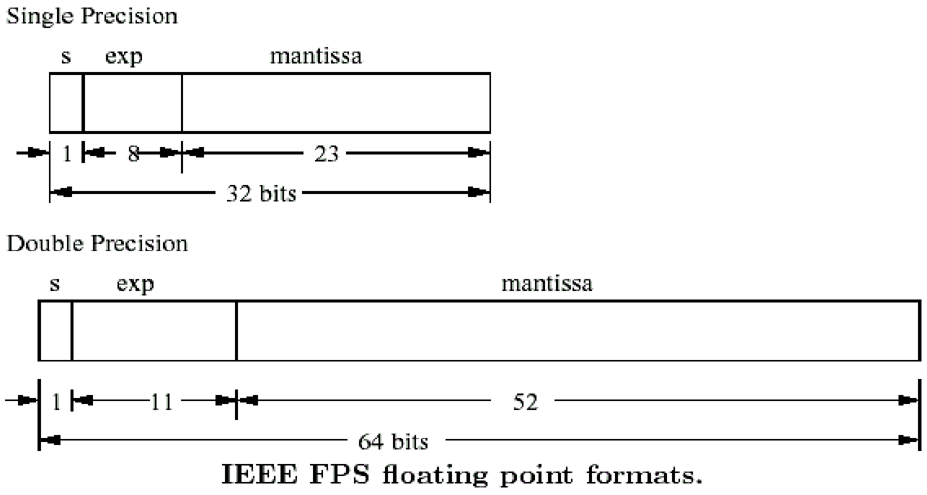


Figure 8.1: Formats of floating point numbers

Floating Point Variables

Floating point variables have been represented in many different ways inside computers of the past. But there is now a well adhered to standard for the representation of floating point variables. The standard is known as the IEEE Floating Point Standard (FPS). Like scientific notation, FPS represents numbers with multiple parts, a sign bit, one part specifying the mantissa and a part representing the exponent. The mantissa is represented as a signed magnitude integer (i.e., not 2's Compliment), where the value is normalized. The exponent is represented as an unsigned integer which is biased to accommodate negative numbers. An 8-bit unsigned value would normally have a range of 0 to 255, but 127 is added to the exponent, giving it a range of -126 to +127.

Follow these steps to convert a number to FPS format.

1. First convert the number to binary.
2. Normalize the number so that there is one nonzero digit to the left of the binary place, adjusting the exponent as necessary.
3. The digits to the right of the binary point are then stored as the mantissa starting with the most significant bits of the mantissa field. Because all numbers are normalized, there is no need to store the leading 1.

   Note: Because the leading 1 is dropped, it is no longer proper to refer to the stored value as the mantissa. In IEEE terms, this mantissa minus its leading digit is called the *significant*.
4. Add 127 to the exponent and convert the resulting sum to binary for the stored exponent value. For double precision, add 1023 to the exponent. Be sure to include all 8 or 11 bits of the exponent.
5. The sign bit is a one for negative numbers and a zero for positive numbers.
6. Compilers often express FPS numbers in hexadecimal, so a quick conversion to hexadecimal might be desired.

Here are some examples using single precision FPS.

- 3.5 = 11.1 (binary)

  = 1.11 x 2^1 sign = 0, significant = 1100..., exponent = 1 + 127 = 128 = 10000000

FPS number (3.5) = 0100000001100000...

= 0 x 40600000

- 100 = 1100100 (binary)

  = 1.100100 x 2^6 sign = 0, significant = 100100...,

  exponent = 6 + 127 = 133 = 10000101

  FPS number (100) = 010000101100100...

  = 0 x 42c80000

- What decimal number is represented in FPS as 0 x c2508000?

  Here we just reverse the steps.

  0xc2508000 = 11000010010100001000000000000000 (binary)

  sign = 1; exponent = 10000100; significant =

  10100001000000000000000

  exponent = 132 ==> 132 - 127 = 5

  -1.10100001 x 2^5 = -110100.001 = -52.125

Floating Point Arithmetic

Until fairly recently, floating point arithmetic was performed using complex algorithms with a integer arithmetic ALU. The main ALU in CPUs is still integer arithmetic ALU. However, in the mid-1980s, special hardware was developed to perform floating point arithmetic. Intel, for example, sold a chip known as the 80387 which was a math co-processor to go along with the 80386 CPU. Most people did not buy the 80387 because of the cost. A major selling point of the 80486 was that the math co-processor was integrated onto the CPU which eliminated the purchase of a separate chip to get faster floating point arithmetic.

Floating point hardware usually has a special set of registers and instructions for performing floating point arithmetic. There are also special instructions for moving data between memory or the normal registers and the floating point registers.

Addition of Floating-Point Numbers

The steps (or stages) of a floating-point addition:

1. The exponents of the two floating-point numbers to be added are compared to find the number with the smallest magnitude.
2. The significant of the number with the smaller magnitude is shifted so that the exponents of the two numbers agree.
3. The significants are added.
4. The result of the addition is normalized.
5. Checks are made to see if any floating-point exceptions occurred during the addition, such as overflow.
6. Rounding occurs.

Floating-Point Addition Example

Example: *s = x + y*

- numbers to be added are *x = 1234.00* and *y = -567.8*
- these are represented in decimal notation with a mantissa (significand) of four digits
- six stages (A - F) are required to complete the addition

Step	**A**	**B**	**C**	**D**	**E**	**F**
X	0.1234E4	0.12340E4				
Y	-0.05678E3	-0.05678E4				
S			0.066620E4	0.6662E3	0.6662E3	0.6662E3

(For this example, we are throwing out biased exponents and the assumed 1.0 before the magnitude. Also all numbers are in the decimal number system and no complements are used.)

Time for Floating-Point Addition

Consider a set of floating-point additions sequentially following one another

(as in adding the elements of two arrays)

Assume that each stage of the addition takes *t* time units;

Time:	***t***	***2t***	***3t***	***4t***	***5t***	***6t***	***7t***	***8t***
Step								
A	$x_1 + y_1$						$x_2 + y_2$	
B		$x_1 + y_1$						$x_2 + y_2$
C			$x_1 + y_1$					
D				$x_1 + y_1$				
E					$x_1 + y_1$			
F						$x_1 + y_1$		

Each floating-point addition takes *6t* time units

Pipelined Floating-Point Addition

With the proper architectural design, the floating-point addition stages can be overlapped

Time:	***t***	***2t***	***3t***	***4t***	***5t***	***6t***	***7t***	***8t***
Step								
A	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	$x_7 + y_7$	$x_8 + y_8$
B		$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	$x_7 + y_7$
C			$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$
D				$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$
E					$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$
F						$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$

This is called pipelined floating-point addition. Once the pipeline is full and has produced the first result in *6t* time units, it only takes *1t* time units to produce each succeeding sum.

8.4 Real Numbers

One sometimes hears variations on the phrase “computers can't represent real numbers exactly”. This, of course is not true. Nothing prevents us from representing (say) the square-root of two as the number two and a bit indicating that the value is the square root of the representation. Some useful operations could be very fast this way. It is true though; that we cannot represent *all* real numbers exactly. In fact, we have a similar problem as that we have with rational numbers, in that it is hard to pick a useful subset that we *can* represent exactly, other than the floating-point numbers.

For this reason, no widespread hardware contains built-in real numbers other than the usual approximations in the form of floating-point.

8.5 Summary

In this unit, we have dealt with various core operations to be performed in arithmetic viz. Addition, Subtraction, Multiplication and Division. We have discussed the various techniques for fixed point unsigned and signed numbers arithmetic. We have also discussed Booth algorithm for multiplication of binary numbers. A separate section is dedicated to the Floating point standard prevalent today that includes the IEEE standard also. We have also seen the addition operation for floating point numbers, laying stress on the time constraint and pipelining concepts.

Self Assessment Questions

1. Using 4-bit fixed length binary, the addition of 4 and 14 results in __________.
2. The two's complement of -5 is __________.
3. Floating-point numbers is to represent a number __________.
4. IEEE FPS represents numbers with __________.
5. Floating point hardware usually has a special set of __________ and __________ for performing floating point arithmetic.

8.6 Terminal Questions

1. Explain the addition of a two floating point numbers with examples.
2. Discuss the different formats of floating point numbers.
3. Compute the product of 7 by 2 using booth's algorithm.

8.7 Answers

Self Assessment Questions

1. overflow
2. (1011)
3. as mantissa and an exponent
4. a sign bit, the mantissa and the exponent
5. registers, instructions

Terminal Questions

1. Refer Section 8.2
2. Refer Section 8.3
3. Refer Section 8.2