Chapter 4 Computer Arithmetic

4.1 Integer Representation

In the binary number system, arbitrary numbers can be represented with just two digits zero and one, the minus sign, and the period, or **radix point**. E.g -1101.01012 = -13.312510. For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An eight bit word can be represented as 0=0000000, 1=0000001, 7=00000111.

In general, if an n-bit sequence of binary digits is interpreted as an unsigned integer A, its value is

$A=\sum_{i=0}^{n-1} 2^i a_i$

4.1.1 Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative. The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost bits hold the magnitude of the integer.

+ 7 = 0111; -7 = 1111 (sign magnitude)

Drawbacks

- Addition and subtraction requires a consideration of both sign of numbers and their relative magnitudes to carry out the required operation.
- Two representations for 0.

4.1.1.1 Two's complement representation

Like sign magnitude two's complement representation uses the most significant bit as sign bit making it easy to test whether the integer is negative or positive. It differs from the use of sign magnitude representation in the way the other bits are interpreted.

Method:

For negation take the Boolean complement of each bits of corresponding positive number, and then add one to the resulting bit pattern.

Consider n nit integer A in two's complement representation. If A is positive then the sign bit $a_{n-1\ zero}$. The remaining bit represents the magnitude of the number.

$A=\sum_{i=0}^{n-1} 2^i a_i$. The range of positive integer that may be represented is $0- 2^{n-1}-1$.

Now for negative numbers integer A, the sign bit a_{n-1} is 1. The range of negative integer that can be represented is from -1 to -2^{n-1}

	Sign representation	Two's complement
+7	0111	0111
-7	1111	1001

4.1.1.2 Float point representation

MR^E

The float point representation of a number has two parts. The first part represents a signed, fixed point number called mantissa. The second part gives the position of the decimal point called exponent.

	Fraction	Exponent
+6132.789	+0.6132789	+04

0.6132789×10^4

In the same it can be used for binary numbers.

	Fraction	Exponent
+1001.11	0.1001110	000100

$(0.1001110) \times 2^4$

Only the mantissa m and the exponent e are physically represented in the register (including their signs).

Negation

In sign magnitude negation is done by inverting the sign bit.

In two's complement method:

- Take Boolean complement if each bit. Then add 1

4.2 Integer Arithmetic

4.2.1 Addition and subtraction with sign twos complement data

Addition

0011
+ 0100
0111

Subtraction

M-N

1. Add M to r's complement of N.
2. Inspect the result obtained in step 1 for an end carry.
   a. If an end carry occurs discard it.
   b. If an end carry does not occur take r's complement of the number obtained in step 1 and place a negative sign in front.

M-N, M=1010100, N=1000100

1's complement of N= 0111011

2's complement of N= (0111011+1) =0111100

1010100
+ 0111100
10010000
Here end carry is generated , so discard it, hence final answer is 0010000

M=1000100, N=1010100

1's complement of N= 0101011

2's complement of N= (0101011+1) =0101100

1000100
+ 0101100
111000
Here no end carry , take 2's complement of 111000 and place a negative sign =-10000

On any addition the result may be larger than the word size being used then it is called overflow. When overflow occurs the ALU must signal so that the process is stopped.

Overflow rule

If two numbers are added and they are both positive or negative, then overflow occurs if and only if the result has opposite sign.

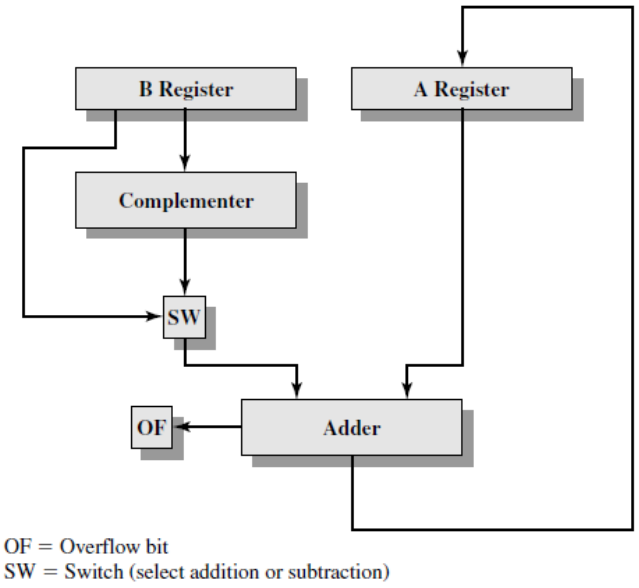

Figure 4.1: block diagram of hardware for addition and subtraction

- Figure shows the data path and hardware elements needed to accomplish addition and subtraction.
- Binary adder is the central element for addition to produce sum and overflow indication.
- For addition of two number in register A and B passed to adder.
- For subtraction the subtrahend (register B) is 2's complemented and added so that the nos. are subtracted.
- Overflow indication stored in 1 bit overflow flag, 0= no overflow, 1= overflow.

4.4 Unsigned Binary Multiplication Algorithm

4.4.1 Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software.

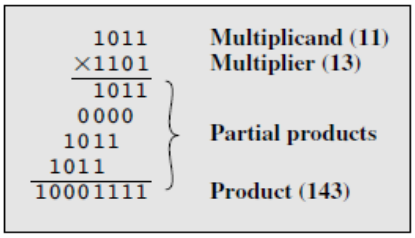

Figure 4.2 : multiplication of unsigned binary integers

This method is not efficient for computer operations so for computer operation, we follow the following algorithms for unsigned number.

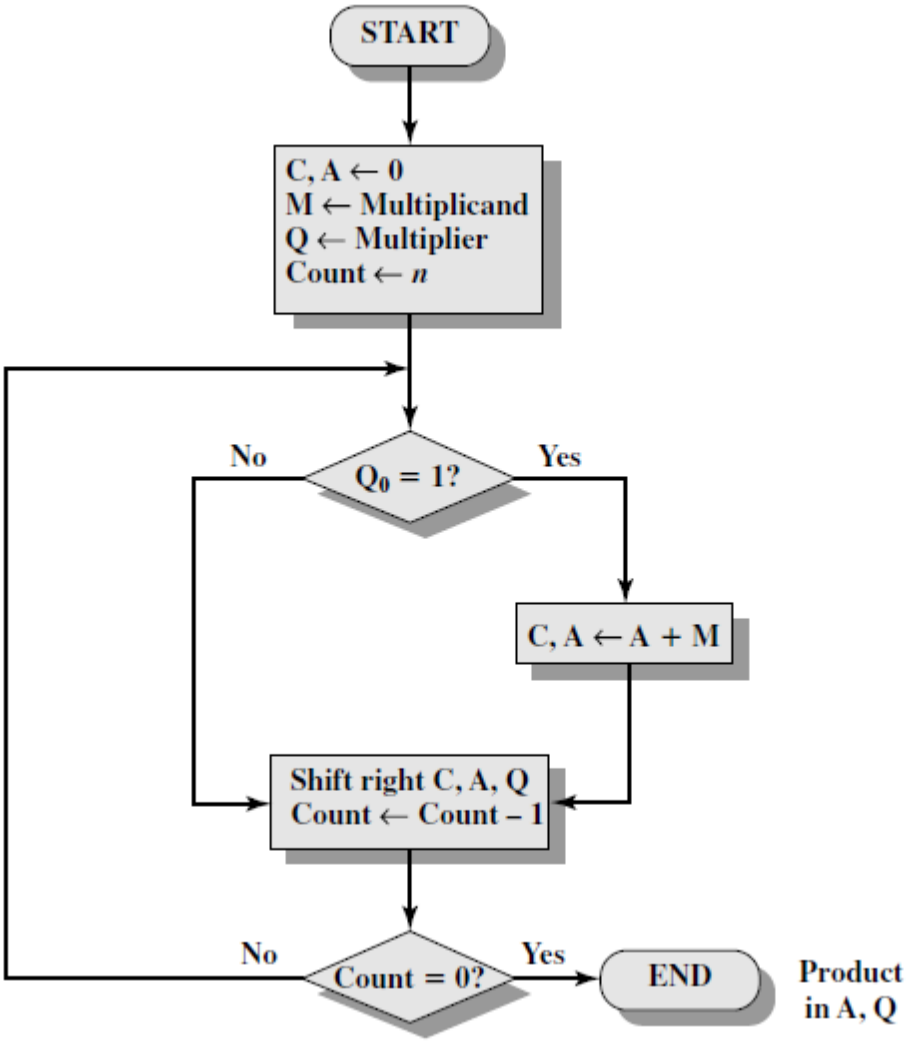

Figure 4.3: flow chart for unsigned binary multiplication

Example: 1101x1011= 1000111, 13x11=143

Count	C	A	Q	M	operations
4	0	0000	1101	1011	Initial values
	0	1011	1101	1011	Add(A+M)
3	0	0101	1110	1011	Shift(right)
2	0	0010	1111	1011	Shift(right)
	0	1101	1111	1011	Add(A+M)
1	0	0110	1111	1011	Shift(right)
	1	0001	1111	1011	Add
0	0	1000	1111	1011	Shift(right)

Final value AQ= $(10001111)_2$

Perform for 7*-3=-21

4.5 Booths algorithm

Multiplication of signed number or negative number is not possible by above method so for that we need booths algorithm

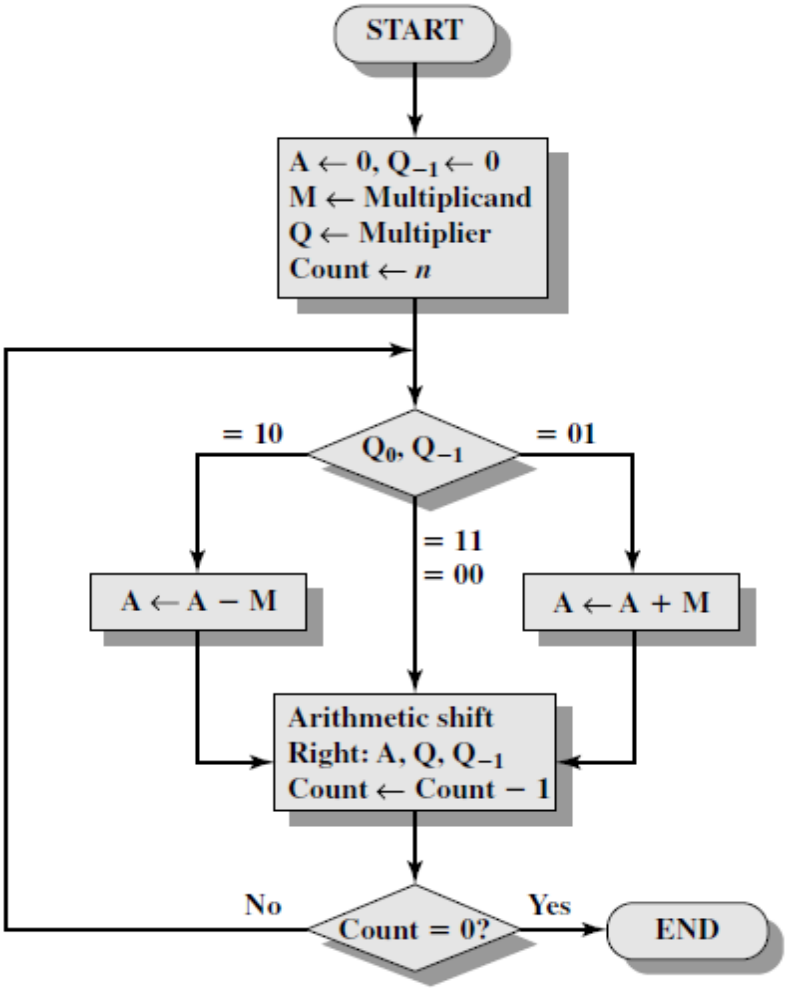

Figure 4.4: flowchart of booth's algorithm for two's complement multiplication.

Perform: 7x-3=-21= $(11101011)_2$

Count	A	Q	Q_{-1}	M	operations
4	0000	0111	0	1101	Initial values
	0011	0111	0	1101	A←A-M
3	0001	1011	1	1101	Shift
2	0000	1101	1	1101	Shift
1	0000	0110	1	1101	Shift
	1101	0110	1	1101	A←A+M
0	1110	1011	0	1101	Shift

AQ= $(11101011)_2$

4.6 Unsigned Binary Division Algorithm

Algorithm

1. Load M with divisor, AQ with dividend (using sign bit).
2. Shift AQ left 1 position (logical shift).
3. If M and A have same sign, A←A-M, else A←A+M
4. Q_0←1, if sign bit of A has not changed.

Else Q_0=0 and restore A.

5. Repeat 2 to 4 till counter is Zero.
6. Remainder in A and quotient in Q.

Example: 5/2 = 2, 1

Count	A	Q	M	-M	operation
4	0000	0101	0010	1110	Initial values
	0000	1010	0010	1110	Shift AQ
Changed	1110	1010	0010	1110	A←A-M
3	0000	1010	0010	1110	
	0001	0100	0010	1110	Shift AQ
changed	1111	0100			A←A-M
2	0001	0100	0010	1110	
	0010	1000	0010	1110	Shift AQ
Not	0000	1000			A←A-M
1	0000	1001	0010	1110	
	0001	0010			Shift AQ
Changed	1111	0010			A←A-M
0	0001	0010	0010	1110	

Remainder=A=0001, quotient =Q=0010

Do for 7/3

4.7 BCD arithemtic Unit

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a four-bit binary adder. The adder will form the sum in *binary* and produce a result that ranges from 0 through 19.

Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

Figure 4.5 Derivation table of BCD adder

These binary numbers are listed in Table 4.5 and are labeled by symbols K, $Z8$, $Z4$, $Z2$, and $Z1$. K is the carry, and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The output sum of two decimal digits must be represented in BCD and should appear in the form listed in the columns under "BCD Sum." The problem is to find a rule by which the binary sum is converted to the correct BCD digit representation of the number in the BCD sum. In examining the contents of the table, it becomes apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain an invalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required. The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 through 1111 that need a correction have a 1 in position $Z8$. To distinguish them from binary 1000 and 1001, which also have a 1 in position $Z8$, we specify further that either $Z4$ or $Z2$ must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage. A BCD adder that adds two BCD digits and produces a sum digit in BCD is shown in Fig. below.

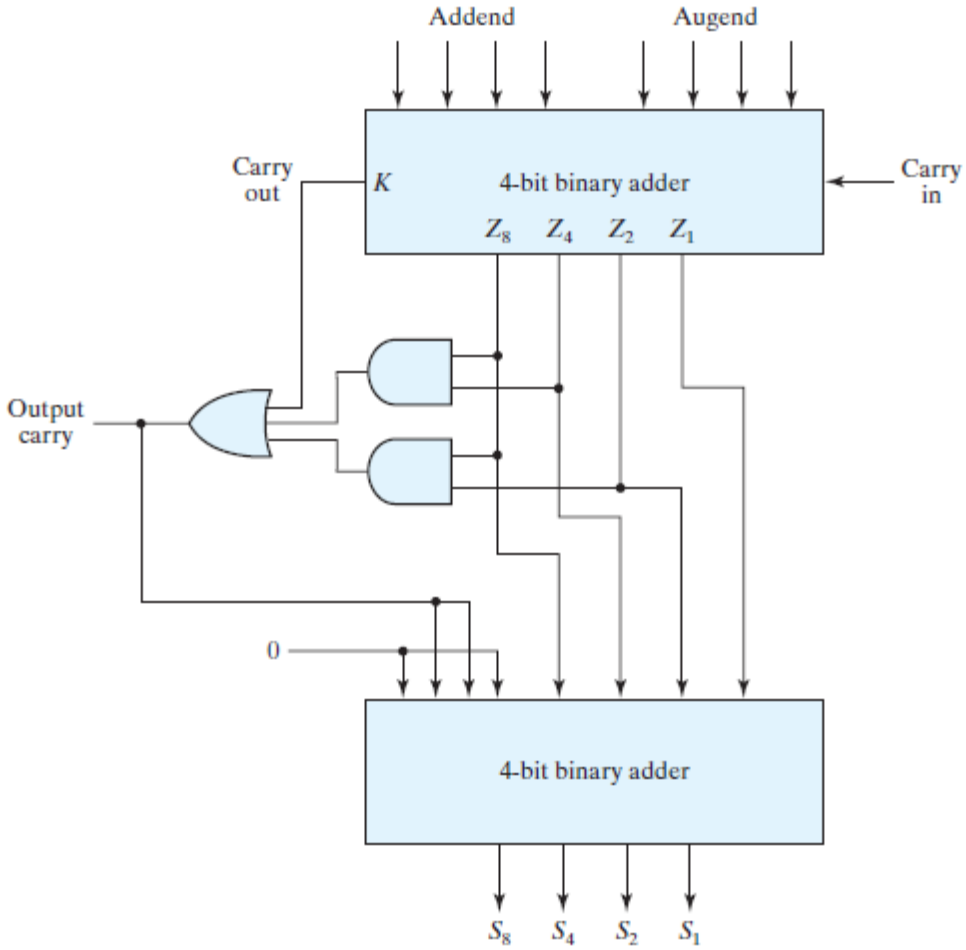

Figure: 4.5 Block diagram of BCD adder

The two decimal digits, together with the input carry, are first added in the top four-bit adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom four-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. A decimal parallel adder that adds *n* decimal digits needs *n* BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher order stage.

4.8 Arithmetic pipeline

Pipeline units are usually found in very high speed computers. They are used to implement for floating operations like multiplication. The floating point operations are easily decomposed into sub operations and then calculated. Example: a pipeline unit for floating addition and subtraction. Consider the two normalized floating point numbers

$X = 0.9504\ x\ 10^3$ & Y= $0.8200\ x\ 10^2$

Now

Segment 1: Two exponents are subtracted 3-2=1. The larger exponent is chosen as the exponent of the result.

Segment 2: Shift the mantissa of y to the right to obtain $X = 0.9504\ x\ 10^3$ & Y= $0.08200\ x\ 10^3$.. This aligns the mantissa under the same exponent.

Segment 3: the addition of two mantissa in segment 3 produces the sum $Z = 1.0324\ x\ 10^3$

Segment 4: the sum is adjusted by the normalizing the result so that it has a fraction with non zero first digit.

$Z = 0.10324\ x\ 4$

The comparator, shifter, adder-subtractor, incrementer and decrementer in the floating point pipeline are implemented with the combination circuits, suppose the time delay of the four segments are t1=60ns, t2=70ns, t3=100ns & t4=80ns, and the interface register have a delay of tr=10 ns. The clock cycle is choosen to be tp=t3+tr=110ns. An equivalent non pipeline floating adder-subtractor will have delay time of tn=t1+t2+t3+t4+t5+tr=320ns.

Speed up=320/110=2.9 over a non pipelined adder.

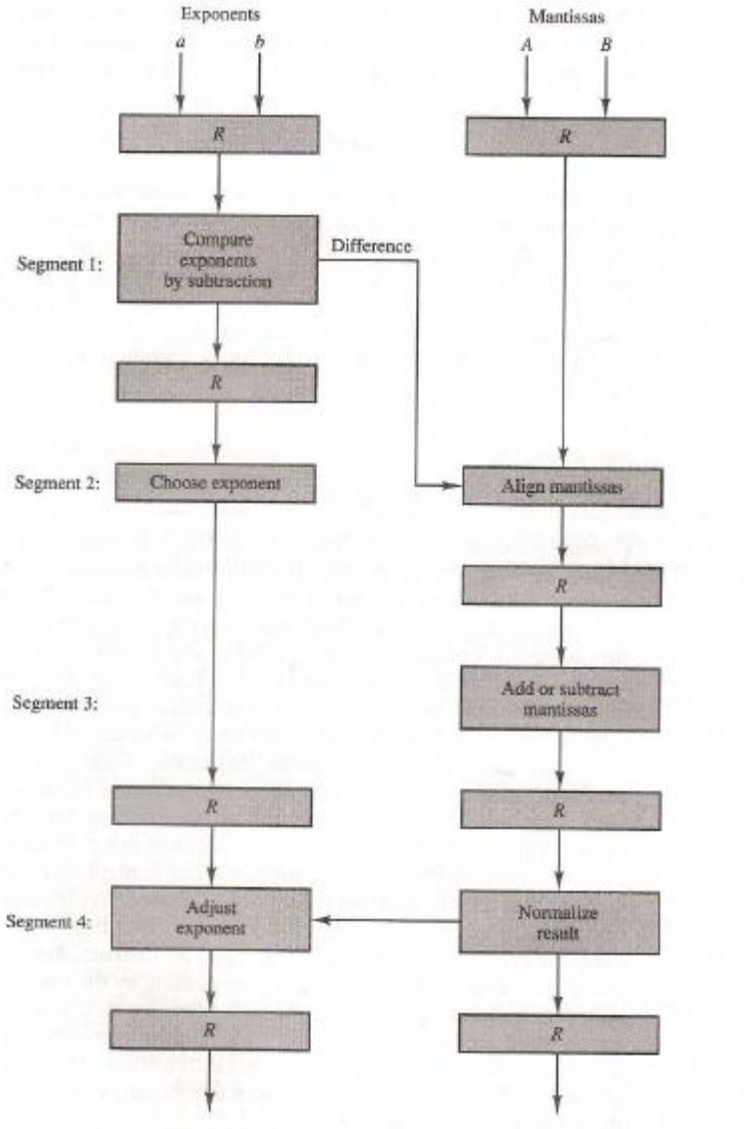

Figure 4.6 : Arithmetic pipeline for floating point addition and subtraction.