NAAC ACCREDITED "B++" (CGPA 2.89)

Yogoda Satsanga Mahavidyalaya

Jagannathpur, Dhurwa, Ranchi-834004

www.ysmranchi.net

ysmranchi4@gmail.com

Course :Computer System Architecture

Class : Sem-1

Lesson :Number System Contd..

By :Goutam Sanyal

Multiplication (decimal)

$$
\begin{array}{r}
13 \\
\times 11 \\
\hline
13 \\
+130 \\
\hline
143
\end{array}
$$

Binary Multiplication

The binary multiplication table is simple:

$$0 * 0 = 0 \;|\; 1 * 0 = 0 \;|\; 0 * 1 = 0 \;|\; 1 * 1 = 1$$

Extending multiplication to <u>multiple digits</u>:

Multiplicand	**1011**
Multiplier	**<u>x 101</u>**
Partial Products	**1011**
	0000 -
	<u>1011 - -</u>
Product	**110111**

Multiplication (binary)

$$\begin{array}{r} 1101 \\ \times 1011 \\ \hline 1101 \\ 11010 \\ +1101000 \\ \hline 10001111 \end{array}$$

Multiplication (binary)

It’s interesting to note that binary multiplication is a sequence of shifts and adds of the first term (depending on the bits in the second term.

$$
\begin{array}{r}
\boxed{1101} \\
\times 1011 \\
\hline
\boxed{1101} \\
\boxed{1101}0 \\
+\boxed{1101}000 \\
\hline
10001111
\end{array}
$$

*1101*00 is missing here because the corresponding bit in the second terms is 0.

Multiplication

- Multiplication is much like as you do it in decimal
  - Line up the numbers and multiply the multiplicand by one digit of the multiplier, aligning it to the right column, and then adding all products together
    - but in this case, all values are either going to be multiplied by 0 or 1
  - So in fact, multiplication becomes a series of shifts and adds:

```
      110011
    * 101001
    --------
      110011
     000000
    000000
   110011
  000000
 110011
 ----------
 Add these values
```

This is the same as:

110011 * 101001 =
110011 * 100000 + 110011 * 00000 +
110011 * 1000 + 110011 * 000 +
110011 * 00 + 110011 * 1
= 110011 * 100000 + 110011 * 1000 + 110011 * 1

We will use a tabular approach for simplicity (see next slides)

Multiplication Algorithm

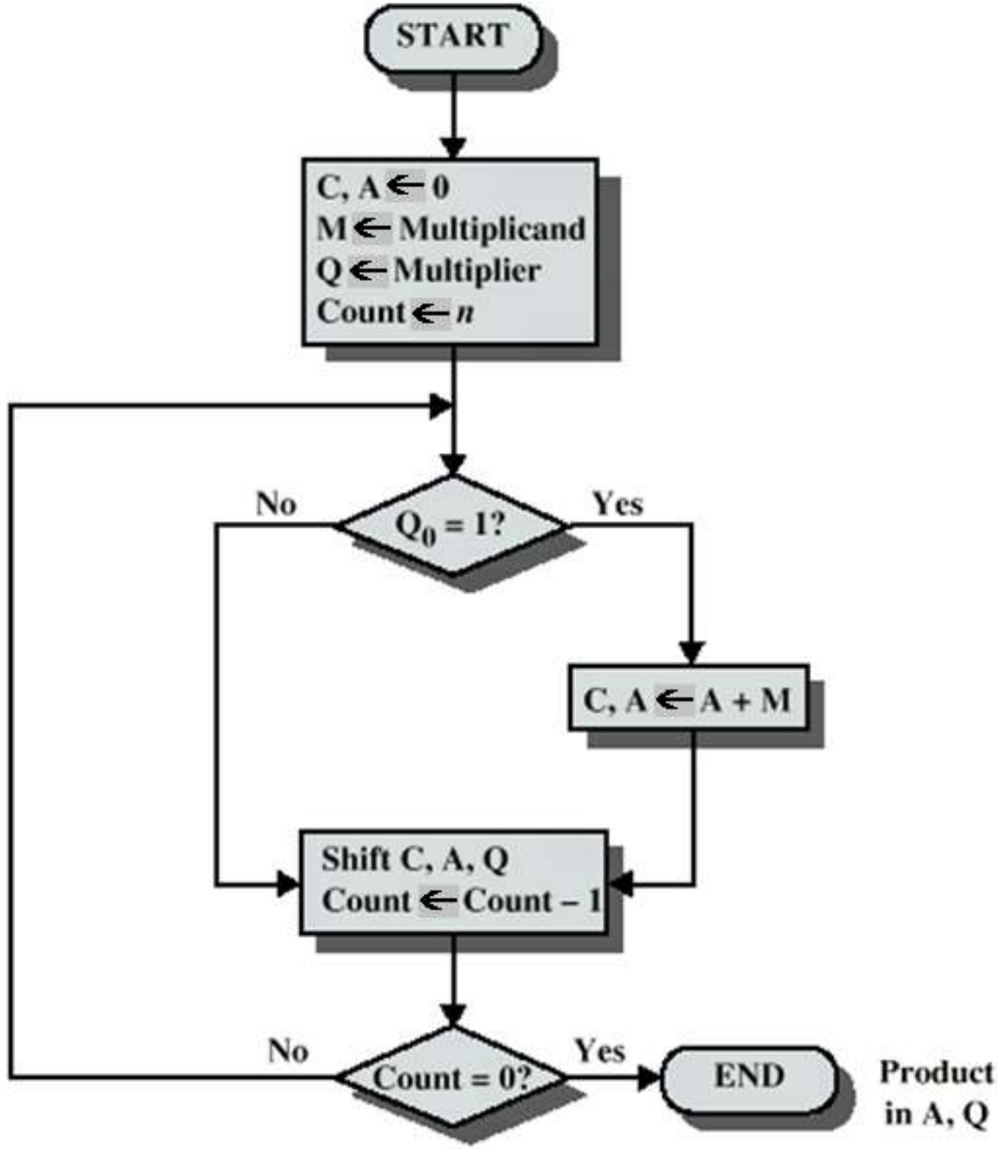
NOTE: this algorithm works only if both numbers are positive. If we have negative values in two's complement, we will use a different algorithm

A is the accumulator

M and Q are temporary registers

C is a single bit storing the carry out of the addition of A and M

The result is stored in the combination of registers A and Q (A storing the upper half of the product, Q the lower half)



Example

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Need 8 bit location to store result of two 4 bit multiplications

First, load the multiplicand in M and the multiplier in Q

A is an accumulator along with the left side of Q

As we shift C/A/Q, we begin to write over part of Q (but it's a part that we've already used in the multiplication)

For each bit in Q, if 0 then merely shift C/A/Q, otherwise add M to C/A

Notice that A/Q stores the resulting product, not just A

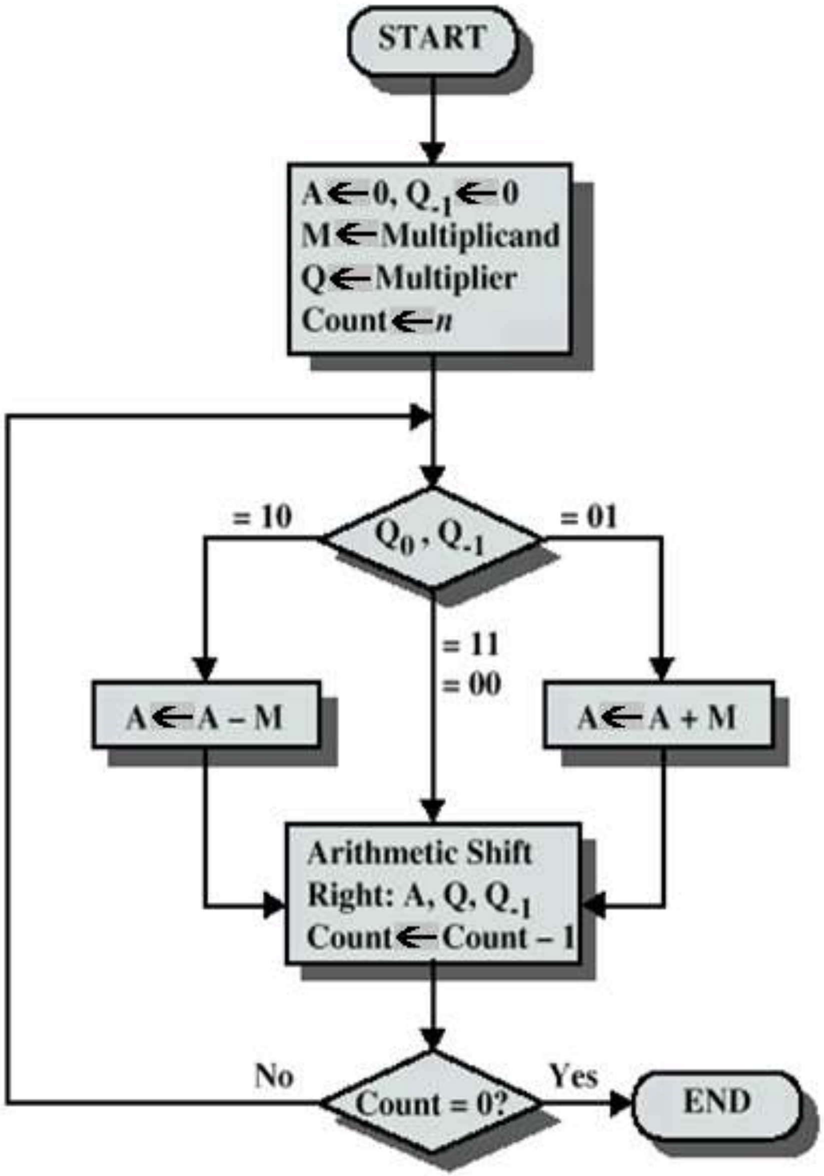
Booth's Algorithm

We will use Booth's algorithm if either or both numbers are negative

The idea is based on this observation:

0011110 = 0100000 – 0000010

So, in Booth's, we look for transitions of 01 and 10, and ignore 00 and 11 sequences in our multiplier



Compare rightmost bit of Q (that is, Q_0) with the previous rightmost bit from Q (which is stored in a single bit Q_{-1})

Q_{-1} is initialized to 0

If this sequence is 0 – 1 then add M to A

If this sequence is 1 – 0 then sub M from A

If this sequence is 0 – 0 or 1 – 1 then don't add

After each iteration, shift A >> Q >> Q_{-1}

Example of Using Booth

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A ← A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A ← A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Initialize A to 0
Initialize Q to 0011
Initialize M to 0111
Initialize Q_{-1} to 0

1) Q/Q_{-1}=10, A←A–M, Shift
2) Q/Q_{-1}=11, Shift
3) Q/Q_{-1}=01,A←A+M, Shift
4) Q/Q_{-1}=00, Shift

Done, Answer = 00010101

Division

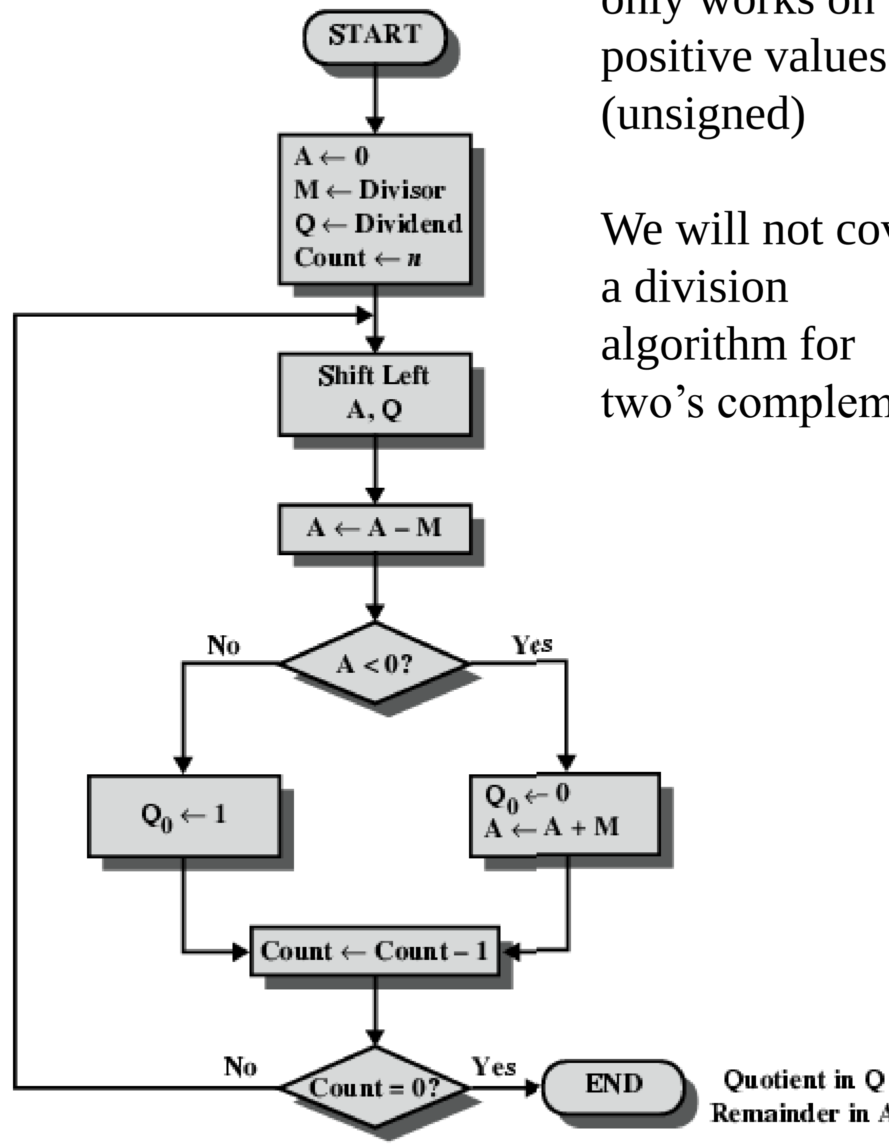
- Just as multiplication is a series of additions and shifts, division is a series of shifts and subtractions
  - The basic idea is this:
    - how many times can we subtract the denominator from the numerator?

Consider 110011 / 000111

We cannot subtract 000111 from 000001	Our divisor is 0, shift 000001
We cannot subtract 000111 from 000011	Our divisor is 00, shift 00011
We cannot subtract 000111 from 000110	Our divisor is 000, shift 000110
We can subtract 000111 from 001100 leaving 000101	Now, our divisor is 0001, shift 000101
We can subtract 000111 from 001010 leaving 000101	Now our divisor is 00011, shift 000101
We can subtract 000111 from 001010 leaving 000101	Our divisor is now 000111
Giving the answer 000111 with a remainder of 000101	We are done after 6 iterations (6 bits)

Division Algorithm

- Dividend is expressed using 2*n bits and loaded into the combined A/Q registers
  – upper half in A, lower half in Q
- Notice that we subtract M from A and then determine if the result is negative – if so, we restore A,
- An easier approach is:
  – Remove A ← A – M
  – Replace A < 0? With A< M?
  – If No, then A ← A – M, Q_n ← 1
  – If Yes, then Q_n ← 0
  – Now we don't need to worry about restoring A
- At the conclusion of the operation
  – the quotient is in Q
  – and any remainder is in A



This algorithm only works on positive values (unsigned)

We will not cover a division algorithm for two's complement

Division Example: 7 / 3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110		Shift A/Q left 1 bit
			Since A < M, insert 0 into Q_0
0001	1100		Shift A/Q left 1 bit
			Since A < M, insert 0 into Q_0
0011	1000		Shift A/Q left 1 bit
0000	1001		Since A >= M, A $\leftarrow$ A-M, insert 1 into Q_0
0001	0010		Shift A/Q left 1 bit
			Since A < M, insert 0 into Q_0

Done (4 shifts)

Result: Q = 0010, A = 0001

A = remainder (1) and Q = quotient (2) or 7 / 3 = 2 1 / 3

Bias Representation

- We can use unsigned magnitude to represent both positive and negative numbers by using a bias, or excess, representation
  - The entire numbering system is shifted up some positive amount
    - To get a value, subtract it from the excess
      - For instance, in excess-16, we subtract 16 from the number to get the real value (11001 in excess-16 is 11001 – 10000 in binary = 01001 = +9)
  - To use the representation
    - numbers have to be shifted, then stored, and then shifted back when retrieved
      - this seems like a disadvantage, so we won't use it to represent ordinary integer values
      - but we will use it to represent exponents in our floating point representation (shown next)

Excess-8 Notation

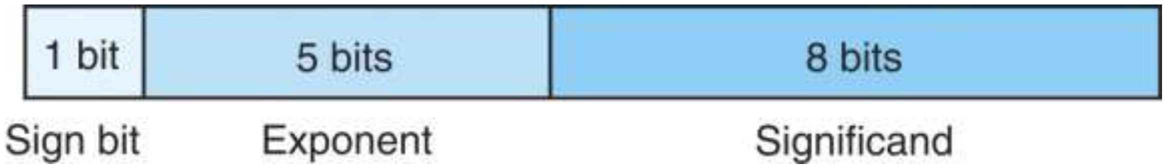
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

Floating Point Representation

- Floating point numbers have a *floating* decimal point
  - Recall the fraction notation used a fixed decimal point
  - Floating point is based on scientific notation
    - $3518.76 = .351876 * 10^4$
    - We represent the floating point number using 2 integer values called the significand and the exponent, along with a sign bit
      - The integers are 351876 and 4 for our example above
  - For a binary version of floating point, we use base 2 instead of 10 as the radix for our exponent
  - We store the 2 integer values plus the sign bit all in binary
    - We *normalize* the floating point number so that the decimal is implied to be before the first 1 bit, and in shifting the decimal point, we determine the exponent
    - The exponent is stored in a bias representation to permit both positive and negative exponents
    - The significand is stored in unsigned magnitude

Examples

- Here, we use the following 14-bit representation:



Exponents will be stored using excess-16

0	1 0 1 0 1	1 0 0 0 1 0 0 0

Sign bit = 0 (positive)
Exponent = 5 (10101 – 10000 = 5)
Significand = .10001000
We shift the decimal point 5 positions giving us 10001.0 = +17

0	0 1 1 1 0	1 0 0 0 0 0 0 0

Sign bit = 0 (positive)
Exponent = -2 (01110 – 10000 = -2)
Significand = .10000000
We shift the decimal point 2 positions to the left, giving us 0.001 = +.125

1	1 0 0 1 1	1 1 0 1 0 1 0 0

Sign bit = 1 (negative)
Exponent = 3 (10011 – 10000 = 3)
Significand = .11010100
We shift the decimal point 3 positions to the right, giving 110.101= -6.625

Floating Point Formats and Problems

- To provide a standard for all architectures, IEEE provides the following formats:
  - Single precision
    - 32-bits: 1-bit sign, 8-bit exponent using excess-127, 23-bit significand
  - Double precision
    - 64-bits: 1-bit sign, 11-bit exponent using excess-1023, 52-bit significand
  - IEEE also provides NAN for errors when a value is not a real number
    - NAN *= not a number*

- Problems
  - there are numerous ways to represent the same number (see page 58), but because we normalize the numbers, there will ultimately be a single representation for the number
  - Errors arise from
    - overflow (too great a positive number or too great a negative number) – overflowing the signficand
    - underflow (too small a fraction) – overflowing the exponent

Representing Characters

- We use a code to represent characters
  - EBCDIC – developed for the IBM 360 and used in all IBM mainframes since then
    - An 8-bit representation for 256 characters
  - ASCII – used in just about every other computer
    - A 7-bit representation plus the high-order bit used for parity
  - Unicode – newer representation to include non-Latin based alphabetic characters
    - 16 bits allow for 65000+ characters
    - It is downward compatible with ASCII, so the first 128 characters are the same as ASCII
  - See figures 2.6-2.8

Error Detection

- Errors will still arise, so we should also provide error detection and correction mechanisms
  - One method is to add a checksum to each block of data
    - Bits are appended to every block that somehow encode the information
    - One common form is the CRC
      - Cyclic redundancy check
      - see pages 81-84 for details
- Simpler approach is to use a parity bit to every byte of information
  - Add up the number of 1 bits in the byte, add a bit so that the number of total 1s is even
    - 00101011 has a parity bit of 0, 11100011 has a parity bit of 1
  - With more parity bits, we can not only detect an error, but correct it, or detect 2 errors
- Hamming Codes are a common way to provide high redundancy on error checking
  - We will skip the discussion on Hamming Codes, but if you want to read it, its on pages 84-90 of the textbook)