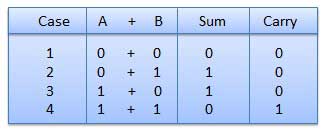
**Binary arithmetic**

**Binary arithmetic is essential part of all the digital computers and many other digital system.**

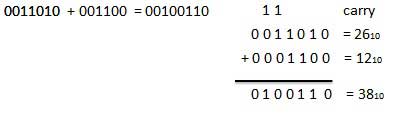
**Binary Addition**

**It is a key for binary subtraction, multiplication, division. There four rules of the binary addition.**

****

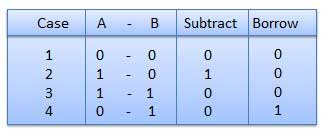
**In fourth case, a binary addition is creating a sum of (1+1=10) i.e. 0 is write in the given column and a carry of 1 over to the next column.**

**EXAMPLE - ADDITION**

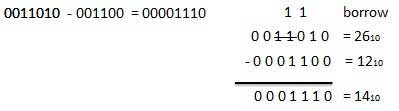
****

**Binary Subtraction**

**Subtraction and Borrow, these two words will be used very frequently for the binary subtraction.There four rules of the binary substration.There four rules of the binary Subtraction.**

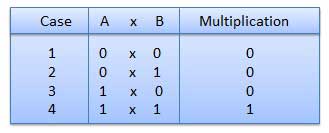
****

**EXAMPLE - SUBTRACTION**

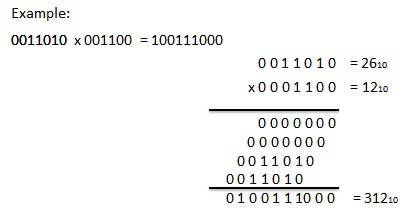
****

**Binary Multiplication**

**Binary multiplication is similar to decimal multiplication. It is simpler than decimal multiplication because only 0s and 1s are involved.There four rules of the binary multiplication.**

****

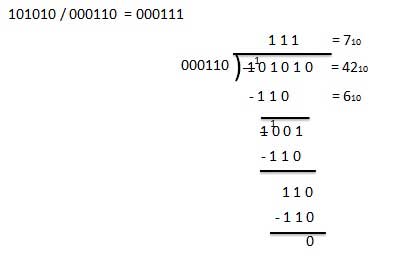
**EXAMPLE - MULTIPLICATION**

****

**Binary Division**

**Binary division is similar to decimal division. It is called as the long division procedure.**

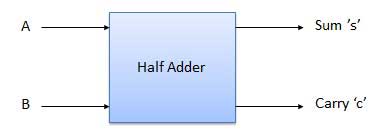
**EXAMPLE - DIVISION**

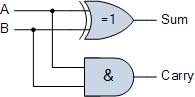
****

**Half Adder**

**Half adder is a combinational logic circuit with two input and two output. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of twosingle bit numbers. This circuit has two outputs carry and sum.**

**BLOCK DIAGRAM**

****



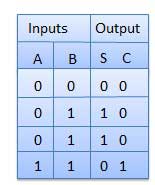
**For the SUM bit**

**SUM = A XOR B = A ⊕ B**

**For the CARRY bit**

**CARRY = A AND B = A.B**

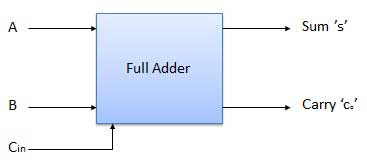
**TRUTH TABLE**

****

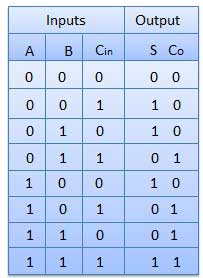
**Full Adder**

**Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry c. The full adder is a three input and two output combinational circuit.**

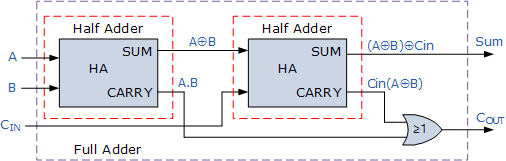
**BLOCK DIAGRAM**

****

**TRUTH TABLE: FULL ADDER**

****

Full Adder Logic Diagram



**As the full adder circuit above is basically two half adders connected together, the truth table for the full adder includes an additional column to take into account the *Carry-in*, CIN input as well as the summed output, S and the Carry-out, COUT bit.**

**Full Adder Truth Table with Carry**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Symbol | Truth Table | | | | |
| **full adder** | **C-in** | **B** | **A** | **Sum** | **C-out** |
| **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **1** | **0** |
| **0** | **1** | **0** | **1** | **0** |
| **0** | **1** | **1** | **0** | **1** |
| **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **1** | **0** | **1** |
| **1** | **1** | **0** | **0** | **1** |
| **1** | **1** | **1** | **1** | **1** |

**Then the Boolean expression for a full adder is as follows.**

**For the SUM (S) bit**

**SUM = (A XOR B) XOR Cin = (A ⊕ B) ⊕ Cin**

**For the CARRY-OUT (Cout) bit**

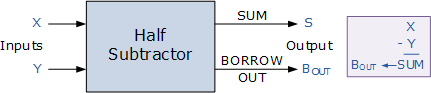
**CARRY-OUT = A AND B OR Cin(A XOR B) = A.B + Cin(A ⊕ B)**

**2-input Exclusive-OR Gate**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol** | **Truth Table** | | |
| **2-input exclusive-or gate**  **2-input Ex-OR Gate** | **Y** | **X** | **S** |
| **0** | **0** | **0** |
| **0** | **1** | **1** |
| **1** | **0** | **1** |
| **1** | **1** | **0** |
|  |  |  |  |

**HALF SUBTRACTOR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbol** | **Truth Table** | | | |
| **half subtractor circuit** | **Y** | **X** | **DIFFERENCE** | **BORROW** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **0** |
| **1** | **0** | **1** | **1** |
| **1** | **1** | **0** | **0** |



**DIFFERENCE bit**

**D = X XOR Y = X +Y**

**For the BORROW bit**

**B = not-X AND Y = X’.Y**

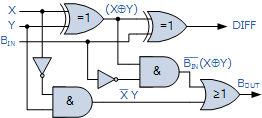
**Half Subtractors**

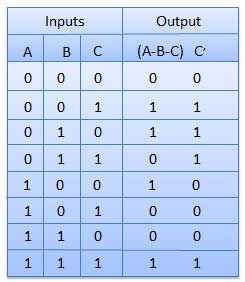
**Half subtractor is a combination circuit with two inputs and two outputs (difference and borrow). It produces the difference between the two binary bits at the input and also produces a output (Borrow) to indicate if a 1 has been borrowed. In the subtraction (A-B), A is called as Minuend bit and B is called as Subtrahend bit.**

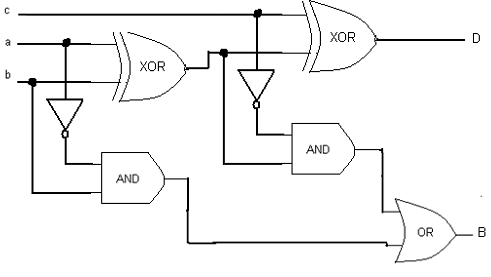
**Full Subtractors**

**The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A,B,C and two output D and C'. A is the minuend, B is subtrahend, C is the borrow produced by the previous stage, D is the difference output and C' is the borrow output.**

**TRUTH TABLE**



****

****

**FULL ADDER:**

**SUM=A’B’C+A’BC’+AB’C’+ABC**

**=A’(B’C+BC’) + A(BC+B’C’)**

**= A (B**http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/graphics/enplus.gif**C)+A (B**http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/graphics/enplus.gif**C)’**

**=A**http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/graphics/enplus.gif**B**http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/graphics/enplus.gif**C**

**CARRY= A’BC + AB’C + ABC’+ABC**

**=BC(A+A’) + AC (B+B’) + AB (C+C’) = BC+AC+AB**

**Ripple Carry Adder**

**Typical Ripple Carry Addition is a Serial process:**

**• Addition starts by adding LSBs of the augend and addend.**

**• Then next position bits of augend and addend are added along with the carry (if any) from the preceding bit.**

**• This process is repeated until the addition of MSBs is completed.**

**Carry Propagation**

**• Speed of a ripple adder is limited due to carry propagation or carry ripple.**

**• Sum of MSB depends on the carry generated by LSB.**

**For example.**

**4-bit Carry Ripple Adder**

**Assume you want to add two operands A and B where**

**A= A3 A2 A1 A0**

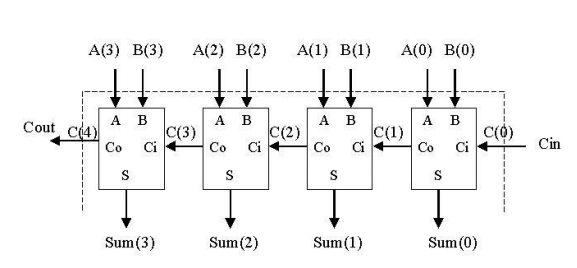
**B= B3 B2 B1 B0**

**For example: A= 1 0 1 1 + B= 1 1 0 1**

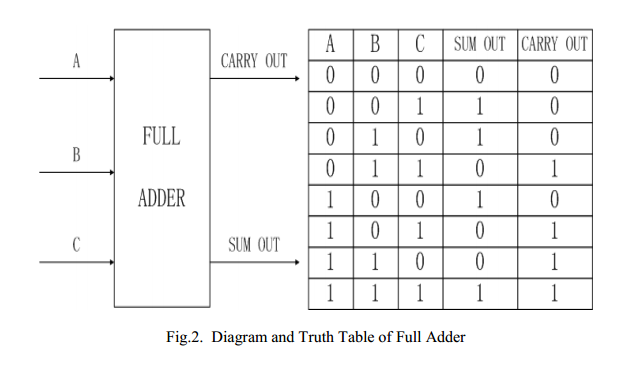
**---------------**

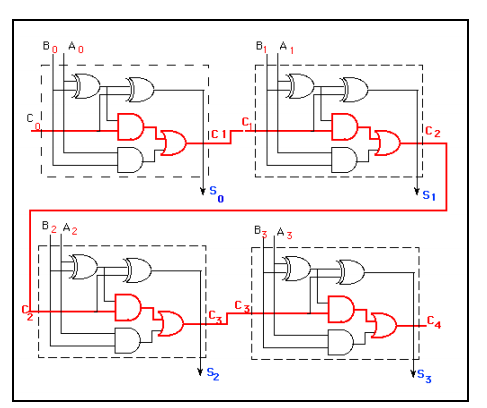
**A+B= 11 0 0 0 = C out S3 S2 S1 S0**

**From the example above it can be seen that we are adding 3 bits at a time sequentially until all bits are added. A full adder is a combinational circuit that performs the arithmetic sum of three input bits: augends Ai, addend Bi and carry in Cin from the previous adder. Its results contain the sum Si and the carry out, Cout to the next stage.**

****

**So to design a 4-bit adder circuit we start by designing the 1–bit full adder then connecting the four 1-bit full adders to get the 4-bit adder as shown in the diagram above. For the 1-bit full adder, the design begins by drawing the Truth Table for the three input and the corresponding output SUM and CARRY. The Boolean Expression describing the binary adder circuit is then deduced. The binary full adder is a three input combinational circuit which satisfies the truth table below.**

****

****

**An n-bit Binary Adder [Ripple Carry Adder]**

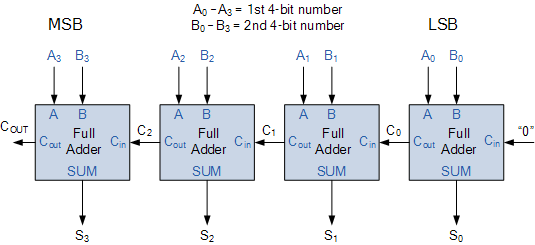
**We have seen above that single 1-bit binary adders can be constructed from basic logic gates. But what if we wanted to add together two n-bit numbers, then n number of 1-bit full adders need to be connected or “cascaded” together to produce what is known as a Ripple Carry Adder.**

**A “ripple carry adder” is simply “n”, 1-bit full adders cascaded together with each full adder representing a single weighted column in a long binary addition. It is called a ripple carry adder because the carry signals produce a “ripple” effect through the binary adder from right to left, (LSB to MSB).**

**For example, suppose we want to “add” together two 4-bit numbers, the two outputs of the first full adder will provide the first place digit sum (S) of the addition plus a carry-out bit that acts as the carry-in digit of the next binary adder.**

**The second binary adder in the chain also produces a summed output (the 2nd bit) plus another carry-out bit and we can keep adding more full adders to the combination to add larger numbers, linking the carry bit output from the first full binary adder to the next full adder, and so forth. An example of a 4-bit adder is given below.**

**A 4-bit Ripple Carry Adder**



**One main disadvantage of “cascading” together 1-bit binary adders to add large binary numbers is that if inputs A and B change, the sum at its output will not be valid until any carry-input has “rippled” through every full adder in the chain because the MSB (most significant bit) of the sum has to wait for any changes from the carry input of the LSB (less significant bit). Consequently, there will be a finite delay before the output of the adder responds to any change in its inputs resulting in a accumulated delay.**

**Carry-lookahead adders**

**A carry-lookahead adders (CLA) is a type of adder used in digital logic. A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower,*ripple carry adder* for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits (see adder for detail on ripple carry adders). The carry-lookahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits.**

**Accordingly, reducing the carry propagation delay of adders is of great importance. Different logic design approaches have been employed to overcome the carry propagation problem.**

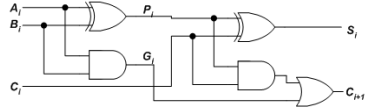
**One widely used approach employs the principle of carry look-ahead solves this problem by calculating the carry signals in advance, based on the input signals.**

**This type of adder circuit is called as carry look-ahead adder (CLA adder). It is based on the fact that a carry signal will be generated in two cases:**

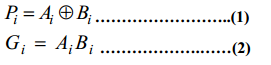
**(1) when both bits Ai and Bi are 1, or**

**(2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.**

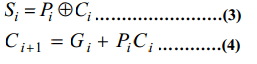
**The Figure shows the full adder circuit used to add the operand bits in the I th column; namely Ai & Bi and the carry bit coming from the previous column (Ci ).**

****

**In this circuit, the 2 internal signals Pi and Gi are given by:**

****

**The output sum and carry can be defined as :**

****

**Gi is known as the carry Generate signal since a carry (Ci+1) is generated whenever Gi =1, regardless of the input carry (Ci).**

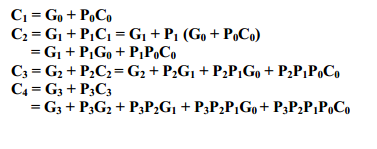
**Pi is known as the carry propagate signal since whenever Pi =1, the input carry is propagated to the output carry, i.e., Ci+1. = Ci (Note that whenever Pi =1, Gi =0).**

**Computing the values of Pi and Gi only depend on the input operand bits (Ai & Bi) as clear from the Figure and equations.**

**Thus, these signals settle to their steady-state value after the propagation through their respective gates.**

* **Computed values of all the Pi’s are valid one XOR-gate delay after the operands A and B are made valid.**
* **Computed values of all the Gi’s are valid one AND-gate delay after the operands A and B are made valid.**

**The Boolean expression of the carry outputs of various stages can be written as follows:**

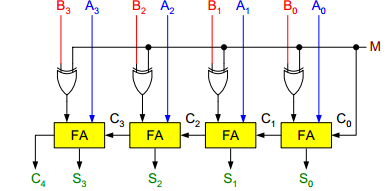
****

**Binary Parallel Adder/Subtractor:**

**The addition and subtraction operations can be done using an Adder-Subtractor circuit.**

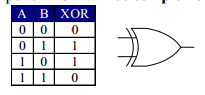
**The figure shows the logic diagram of a 4-bit**

**Adder-Subtractor circuit.**

**Adder-Subtractor circuit **

**The circuit has a mode control signal M which determines if the circuit is to operate as an adder or a subtractor.**

**Each XOR gate receives input M and one of the inputs of B, i.e., Bi. To understand the behavior of XOR gate consider its truth table given below. If one input of XOR gate is zero then the output of XOR will be same as the second input. While if one input of XOR gate is one then the output of XOR will be complement of the second input.**

****

**So when M = 0,**

**the output of XOR gate will be Bi ⊕ 0 = Bi.**

**If the full adders receive the value of B, and the input carry C0 is 0, the circuit performs A plus B.**

**When M = 1, the output of XOR gate will be Bi ⊕ 1 = Bi. If the full adders receive the value of B’s, and the input carry C0 is 1, the circuit performs A plus 1’s complement of B plus 1, which is equal to A minus B.**

**Binary Multiplier:**

**Multiplication of binary numbers is performed in the same way as with decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit.**

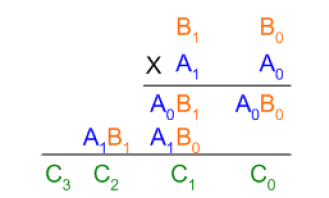
**The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left.**

**The product is obtained by adding these shifted partial products.**

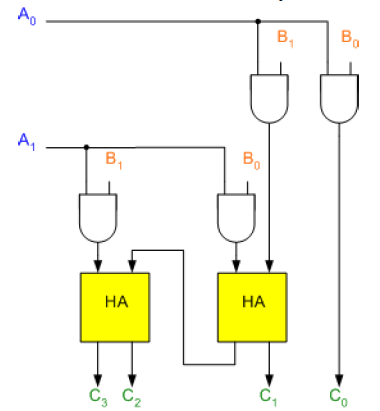
**Example 1: Consider an example of multiplication of two numbers, say A and B (2 bits each),**

**C = A x B.**

* **The first partial product is formed by multiplying the B1B0 by A0. The multiplication of two bits such as A0 and B0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.**
* **The second partial product is formed by multiplying the B1B0 by A1 and is shifted one position to the left.**

****

**The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use FAs.**

****

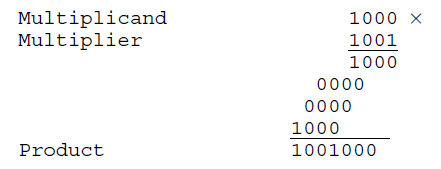
**The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure.**

**Shift – and – Add multiplier :**

**Shift-and-add multiplication is similar to the multiplication performed by paper and pencil. This method adds the multiplicand *X* to itself *Y* times, where *Y* denotes the multiplier. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.**

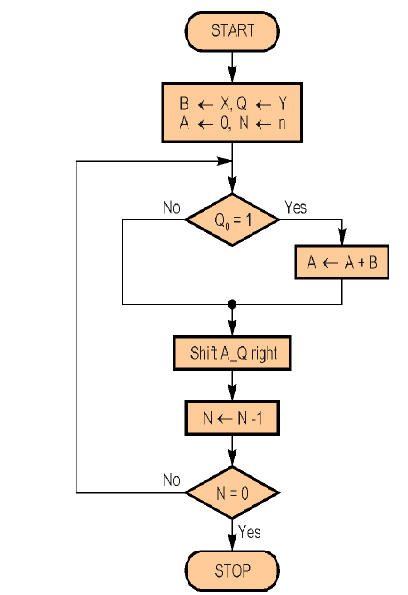
**As an example, consider the multiplication of two unsigned 4-bit numbers,**

**8 (1000) and 9 (1001).**

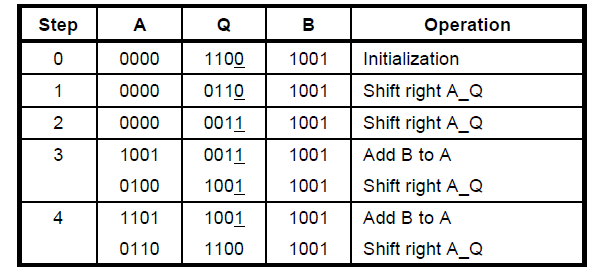
****

**In the case of binary multiplication, since the digits are 0 and 1, each step of the multiplication is simple. If the multiplier digit is 1, a copy of the multiplicand (1 ×multiplicand) is placed in the proper positions; if the multiplier digit is 0, a number of 0 digits (0 × multiplicand) are placed in the proper positions.**

**Consider the multiplication of positive numbers. The first version of the multiplier circuit, which implements the shift-and-add multiplication method for two *n*-bit numbers, is shown in Figure.**

****

**For Example, Perform the multiplication 9[multiplicand] x 12[multiplier] (1001 x 1100). Finally, both A and Q contains the result of product.**

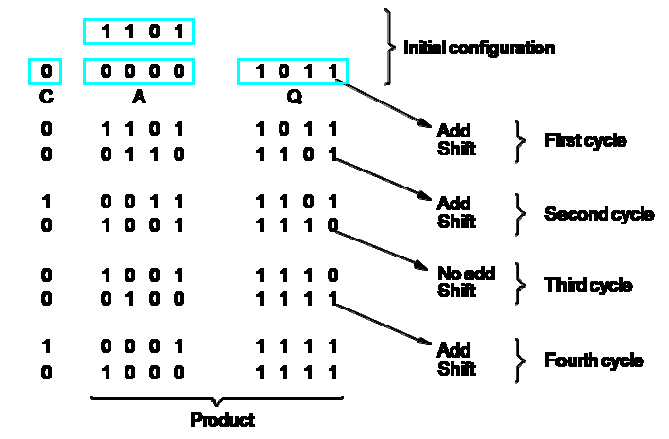
****

**Another Example.**

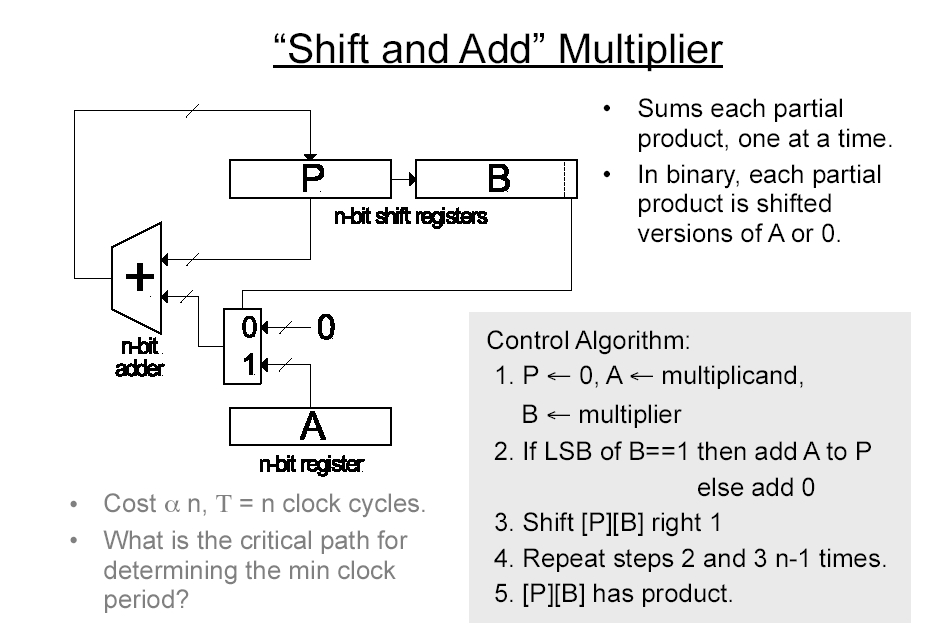
**A = 0000**

**M(Multiplicand) 🡪 13 ( 1 1 0 1)**

**Q(Multiplier) 🡪 11 (1 0 1 1)**

****

**CONCEPT OF SHIFT AND ADD MULTIPLIER**

****

**Signed Multiplication**

**Booth’s Algorithm**

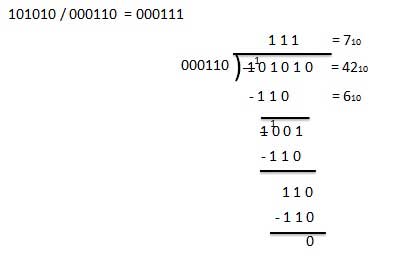
**A powerful algorithm for signed –number multiplication is a Booth’s algorithm which generates a 2n bit product and treats both positive and negative numbers uniformly. This algorithm suggests that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.**

|  |
| --- |
| **Qn Qn+1 Action** |
| **0 0 No change . Shift Left** |
| **1 1 No change . Shift Left** |
| **0 1 A + M, Shift Left** |
| **1 0 A - M, Shift Left** |

***For further information on BOOTHS algorithm refer to the PPT slides file [8 BOOTHS algorithm] in the UNIT-I Lecture Notes.***

**Integer Division**

**Manual Division**

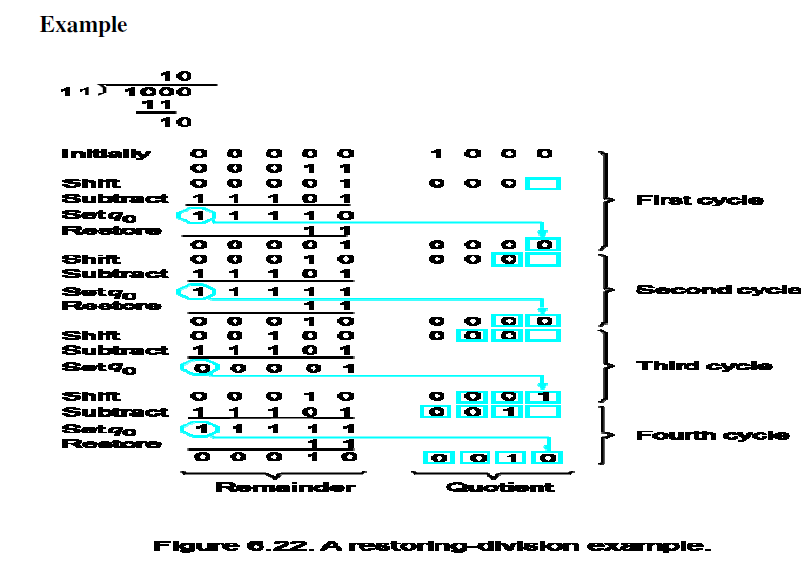
****

**Longhand Division Steps**

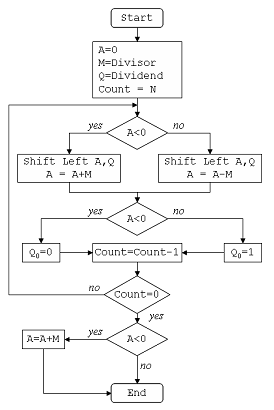
* **Position the divisor appropriately with respect to the dividend and perform a subtraction.**
* **If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.**
* **If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is reposition for another subtraction.**

**Restoring Division**

* **Similar to multiplication circuit**
* **N-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation.**
* **Register A is set to 0**
* **After the division operation is complete, the n-bit quotient is in register Q and the remainder is in register A.**
* **The required subtractions are facilitated by using 2’s complement arithmetic.**
* **The extra bit position at the left end of both A and M accommodates the sign bit during subtraction.**
* **Shift A and Q left one binary position**
* **Subtract M from A, and place the answer back in A**
* **If the sign of A is 1, set q0 to 0 and add M back to A (restore A); otherwise, set q0 to 1**
* **• Repeat these steps *n* times**

****

**Algorithm for Non-restoring division is given in below image:**

****

**In this problem, Dividend (A) = 101110, ie 46, and Divisor (B) = 010111, ie 23.**

**Initialization :**

**Set Register A = Dividend = 000000**

**Set Register Q = Dividend = 101110**

**( So AQ = 000000 101110 , Q0 = LSB of Q = 0 )**

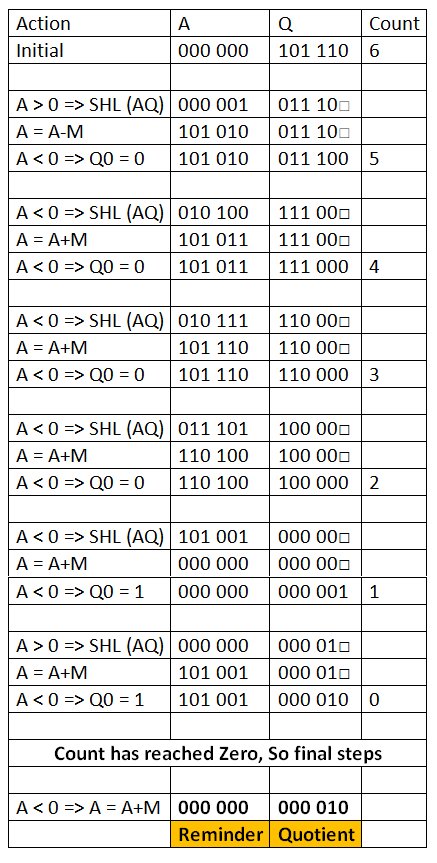
**Set M = Divisor = 010111, M' = 2's complement of M = 101001**

**Set Count = 6, since 6 digits operation is being done here.**

**After this we start the algorithm, which I have shown in a table below:**

**In table, SHL(AQ) denotes shift left AQ by one position leaving Q0 blank.**

**Similarly, a square symbol in Q0 position denote, it is to be calculated later**

****