UNIT III - COMBINATIONAL CIRCUITS:

INTRODUCTION:

The digital system consists of two types of circuits, namely

(i) Combinational circuits

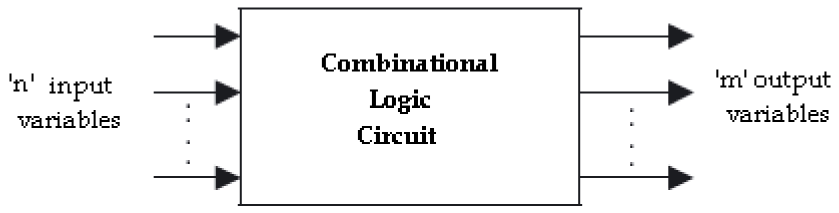
(ii) Sequential circuits

Combinational circuit consists of logic gates whose output at any time is determined from the present combination of inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions.

Sequential logic circuit comprises both logic gates and the state of storage elements such as flip-flops. As a consequence, the output of a sequential circuit depends not only on present value of inputs but also on the past state of inputs.

In the previous chapter, we have discussed binary numbers, codes, Boolean algebra and simplification of Boolean function and logic gates. In this chapter, formulation and analysis of various systematic designs of combinational circuits will be discussed.

A combinational circuit consists of input variables, logic gates, and output variables. The logic gates accept signals from inputs and output signals are generated according to the logic circuits employed in it. Binary information from the given data transforms to desired output data in this process. Both input and output are obviously the binary signals, *i.e.*, both the input and output signals are of two possible states, logic 1 and logic 0.



Block diagram of a combinational logic circuit

For *n* number of input variables to a combinational circuit, 2^n possible combinations of binary input states are possible. For each possible combination, there is one and only one possible output combination. A combinational logic circuit can be described by *m* Boolean functions and each output can be expressed in terms of *n* input variables.

DESIGN PROCEDURE:

Any combinational circuit can be designed by the following steps of design procedure.

1. The problem is stated.
2. Identify the input and output variables.
3. The input and output variables are assigned letter symbols.
4. Construction of a truth table to meet input -output requirements.
5. Writing Boolean expressions for various output variables in terms of input variables.
6. The simplified Boolean expression is obtained by any method of minimization—algebraic method, Karnaugh map method, or tabulation method.
7. A logic diagram is realized from the simplified boolean expression using logic gates.

The following guidelines should be followed while choosing the preferred form for hardware implementation:

1. The implementation should have the minimum number of gates, with the gates used having the minimum number of inputs.
2. There should be a minimum number of interconnections.
3. Limitation on the driving capability of the gates should not be ignored.

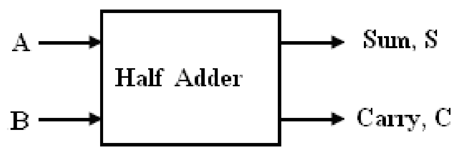
ARITHMETIC CIRCUITS – BASIC BUILDING BLOCKS:

In this section, we will discuss those combinational logic building blocks that can be used to perform addition and subtraction operations on binary numbers. Addition and subtraction are the two most commonly used arithmetic operations, as the other two, namely multiplication and division, are respectively the processes of repeated addition and repeated subtraction.

The basic building blocks that form the basis of all hardware used to perform the arithmetic operations on binary numbers are half-adder, full adder, half-subtractor, full-subtractor.

Half-Adder:

A half-adder is a combinational circuit that can be used to add two binary bits. It has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.



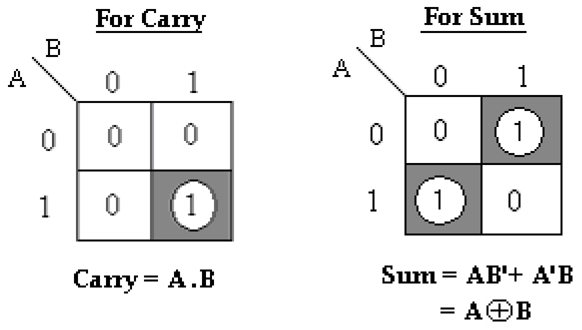
Block schematic of half-adder

The truth table of a half-adder, showing all possible input combinations and the corresponding outputs are shown below.

Inputs		Outputs	
A	**B**	**Carry (C)**	**Sum (S)**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth table of half-adder

K-map simplification for carry and sum:



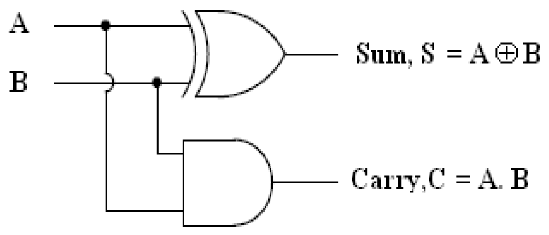
The Boolean expressions for the SUM and CARRY outputs are given by the equations,

Sum, S = A’B+ AB’= A⊕B

Carry, C = A . B

The first one representing the SUM output is that of an EX-OR gate, the second one representing the CARRY output is that of an AND gate.

The logic diagram of the half adder is,

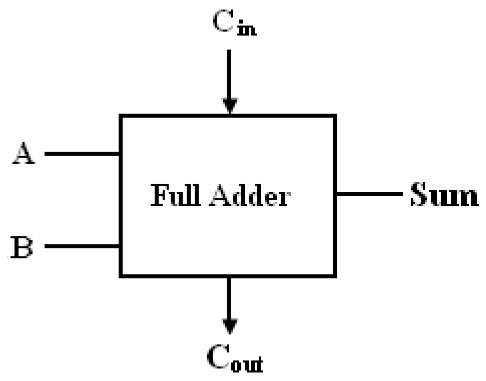


Logic Implementation of Half-adder

Full-Adder:

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of 3 inputs and 2 outputs.

Two of the input variables, represent the significant bits to be added. The third input represents the carry from previous lower significant position. The block diagram of full adder is given by,



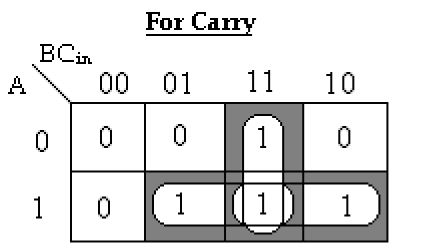
Block schematic of full-adder

The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only. As there are three input variables, eight different input combinations are possible. The truth table is shown below,

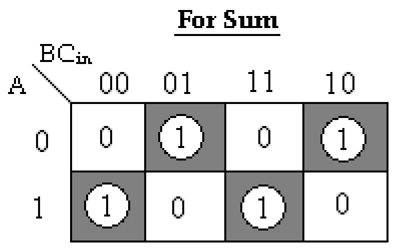
Truth Table:

Inputs			**Outputs**	
A	**B**	**C_{in}**	**Sum (S)**	**Carry (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

To derive the simplified Boolean expression from the truth table, the Karnaugh map method is adopted as,



Carry, C_{out} = $AB + AC_{in} + BC_{in}$



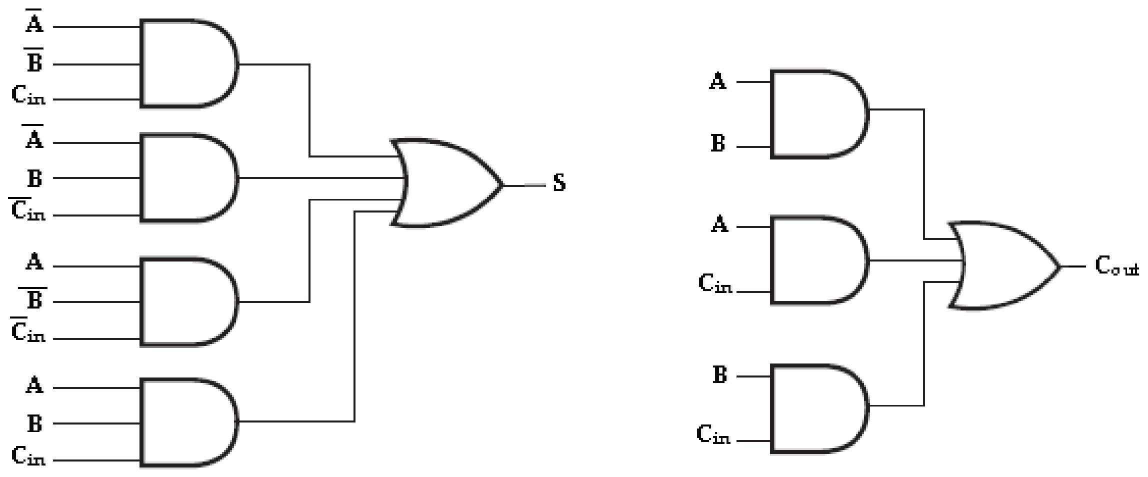
Sum, S = $A'B'C_{in} + A'BC'_{in} + AB'C'_{in} + ABC_{in}$

The Boolean expressions for the SUM and CARRY outputs are given by the equations,

Sum, S = $\mathbf{A'B'C_{in} + A'BC'_{in} + AB'C'_{in} + ABC_{in}}$

Carry, $\mathbf{C_{out}}$ = $\mathbf{AB + AC_{in} + BC_{in}}$.

The logic diagram for the above functions is shown as,



Implementation of full-adder in Sum of Products

The logic diagram of the full adder can also be implemented with two half-adders and one OR gate. The S output from the second half adder is the exclusive-OR of C_{in} and the output of the first half-adder, giving

$$\mathbf{Sum} = \mathbf{C_{in} \oplus (A \oplus B)} \qquad [x \oplus y = x'y + xy']$$

$$= C_{in} \oplus (A'B + AB')$$

$$= C'_{in}\ (A'B + AB') + C_{in}\ (A'B + AB')' \qquad [(x'y + xy')' = (xy + x'y')]$$

$$= C'_{in} (A'B + AB') + C_{in} (AB + A'B')$$

$$= A'BC'_{in} + AB'C'_{in} + ABC_{in} + A'B'C_{in}\,.$$

and the carry output is,

Carry, C_{out} = AB+ C_{in} (A'B+AB')

$= AB + A'BC_{in} + AB'C_{in}$

$= AB(C_{in}+1) + A'BC_{in} + AB'C_{in}$ [$C_{in}+1=1$]

$= ABC_{in} + AB + A'BC_{in} + AB'C_{in}$

$= AB + AC_{in}(B+B') + A'BC_{in}$

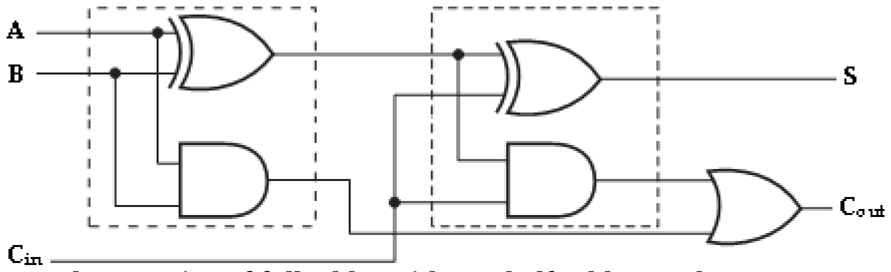
$= AB + AC_{in} + A'BC_{in}$

$= AB(C_{in}+1) + AC_{in} + A'BC_{in}$ [$C_{in}+1=1$]

$= ABC_{in} + AB + AC_{in} + A'BC_{in}$

$= AB + AC_{in} + BC_{in}(A+A')$

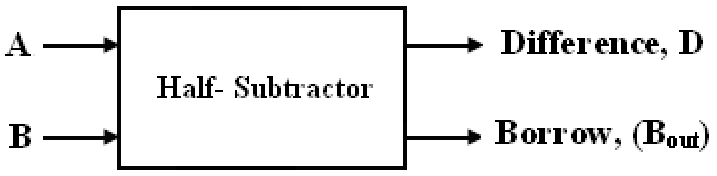
$= AB + AC_{in} + BC_{in}$.



Implementation of full adder with two half-adders and an OR gate

Half -Subtractor:

A *half-subtractor* is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a ‗1' has been borrowed to perform the subtraction.

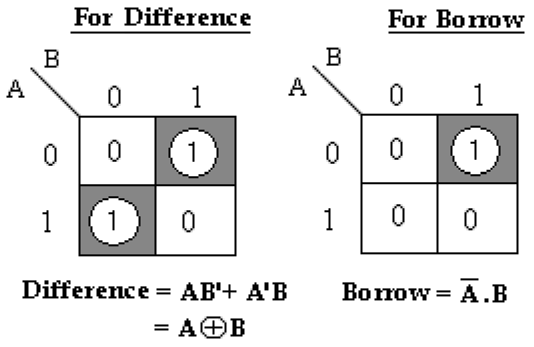


Block schematic of half-subtractor

The truth table of half-subtractor, showing all possible input combinations and the corresponding outputs are shown below.

Input		Output	
A	**B**	**Difference (D)**	**Borrow (B_{out})**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

K-map simplification for half subtractor:



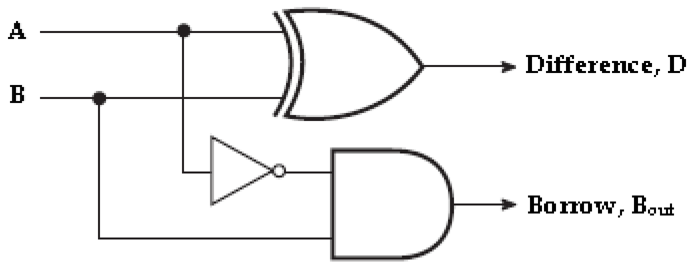
The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = A'B+ AB'= A ⊕ B

Borrow, B_{out} = A' . B

The first one representing the DIFFERENCE (**D**)output is that of an exclusive-OR gate, the expression for the BORROW output (**B_{out}**) is that of an AND gate with input A complemented before it is fed to the gate.

The logic diagram of the half adder is,



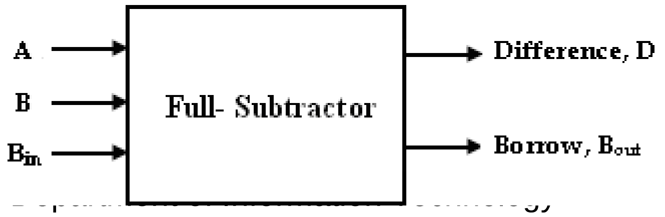
Logic Implementation of Half-Subtractor

Comparing a half-subtractor with a half-adder, we find that the expressions for the SUM and DIFFERENCE outputs are just the same. The expression for BORROW in the case of the half-subtractor is also similar to what we have for CARRY in the case of the half-adder. If the input A, ie., the minuend is complemented, an AND gate can be used to implement the BORROW output.

Full Subtractor:

A *full subtractor* performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a ‗1' has already been borrowed by the previous adjacent lower minuend bit or not.

As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in}. There are two outputs, namely the DIFFERENCE output D and the BORROW output B_o. The



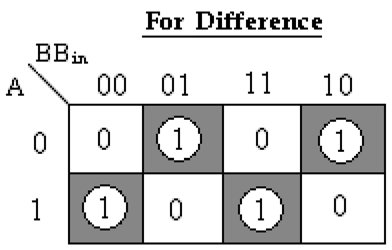
BORROW output bit tells whether the minuend bit needs to borrow a ‗1‘ from the next possible higher minuend bit.

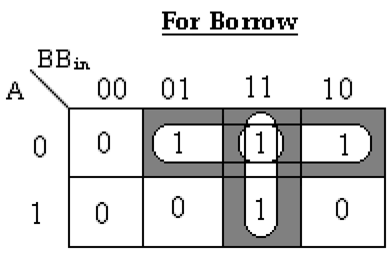
Block schematic of full-adder

The truth table for full-subtractor is,

Inputs			**Outputs**	
A	**B**	**B_{in}**	**Difference(D)**	**Borrow(B_{out})**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

K-map simplification for full-subtractor:





Difference, D = $A'B'B_{in}$+ $A'BB'_{in}$+ $AB'B'_{in}$+ ABB_{in}

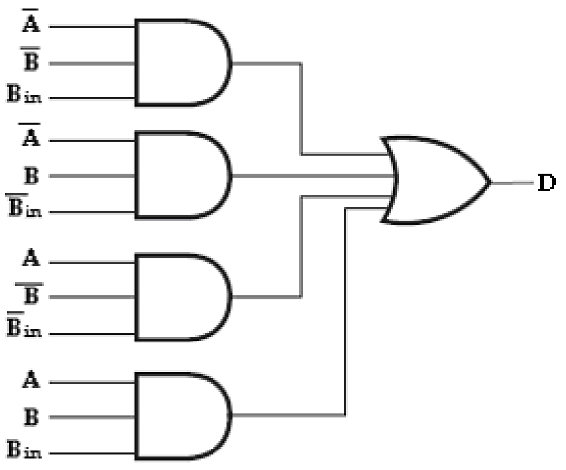
Borrow, B_{out} = $A'B$+ $A'B_{in}$ + BB_{in}

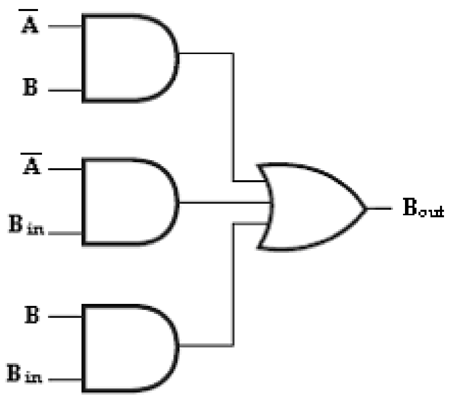
The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D $= A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

Borrow, B_{out} $= A'B + A'C_{in} + BB_{in}$.

The logic diagram for the above functions is shown as,





<u>Implementation of full-adder in Sum of Products</u>

The logic diagram of the full-subtractor can also be implemented with two half-subtractors and one OR gate. The difference,D output from the second half subtractor is the exclusive-OR of B_{in} and the output of the first half-subtractor, giving

Difference,D= $\mathbf{B_{in} \oplus (A \oplus B)}$ $[x \oplus y = x'y + xy']$

$= B_{in} \oplus (A'B+AB')$

$= B'_{in}\ (A'B+AB') + B_{in}\ (A'B+AB')'$ $[(x'y+xy')' = (xy+x'y')]$

$= B'_{in}\ (A'B+AB') + B_{in}\ (AB+A'B')$

$= A'BB'_{in} + AB'B'_{in} + ABB_{in} + A'B'B_{in}.$

and the borrow output is,

Borrow, $\mathbf{B_{out} = A'B + B_{in}\ (A'B+AB')'}$ $[(x'y+xy')' = (xy+x'y')]$

$= A'B + B_{in}\ (AB+A'B')$

$= A'B + ABB_{in} + A'B'B_{in}$

$= A'B\ (B_{in}+1) + ABB_{in} + A'B'B_{in}$ $[C_{in}+1 = 1]$

$= A'BB_{in} + A'B + ABB_{in} + A'B'B_{in}$

$= A'B + BB_{in}\ (A+A') + A'B'B_{in}$ $[A+A' = 1]$

$= A'B + BB_{in} + A'B'B_{in}$

$= A'B\ (B_{in}+1) + BB_{in} + A'B'B_{in}$ $[C_{in}+1 = 1]$

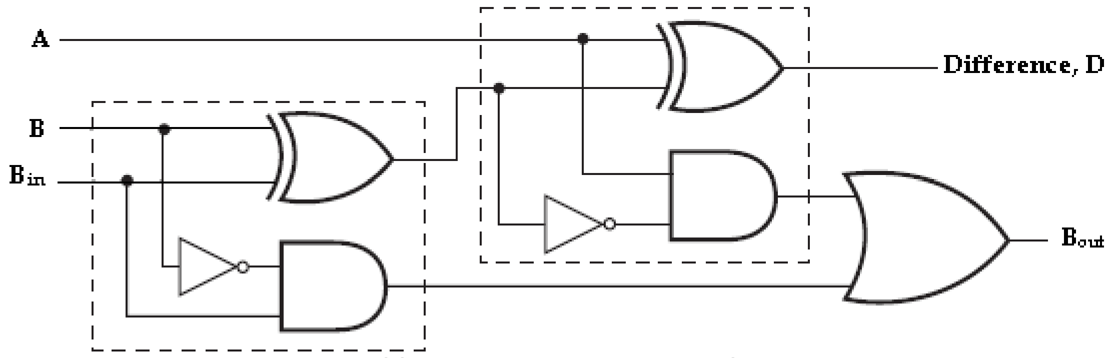
$= A'BB_{in} + A'B + BB_{in} + A'B'B_{in}$

$= A'B + BB_{in} + A'B_{in}\ (B + B')$

$= A'B + BB_{in} + A'B_{in}.$

Therefore,

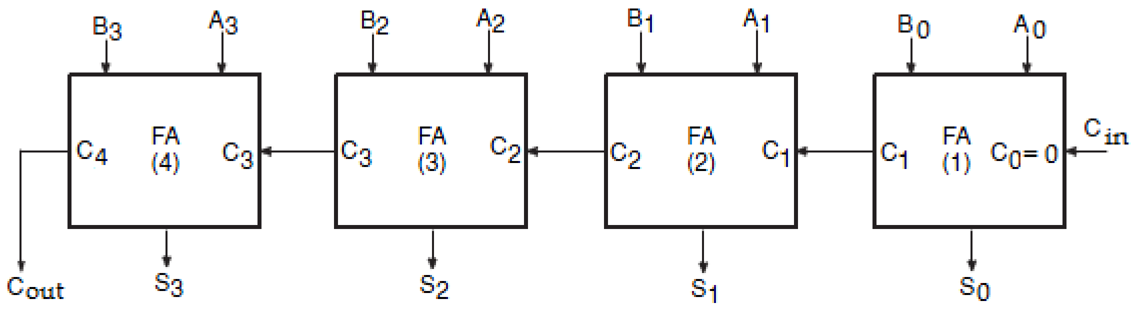
we can implement full-subtractor using two half-subtractors and OR gate as,



<u>Implementation of full-subtractor with two half-subtractors and an OR gate</u>

Binary Adder (Parallel Adder):

The 4-bit binary adder using full adder circuits is capable of adding two 4-bit numbers resulting in a 4-bit sum and a carry output as shown in figure below.



4-bit binary parallel Adder

Since all the bits of augend and addend are fed into the adder circuits simultaneously and the additions in each position are taking place at the same time, this circuit is known as parallel adder.

Let the 4-bit words to be added be represented by,

$A_3A_2A_1A_0 = 1111$ and $B_3B_2B_1B_0 = 0011$.

Significant place		4	3	2	1	
Input carry		1	1	1	0	
Augend word A :		1	1	1	1	
Addend word B :		0	0	1	1	
	1	0	0	1	0	← Sum
	↑ Output Carry					

The bits are added with full adders, starting from the least significant position, to form the sum it and carry bit. The input carry C_0 in the least significant position must be 0. The carry output of the lower order stage is connected to the carry input of the next higher order stage. Hence this type of adder is called ripple-carry adder.

In the least significant stage, A_0, B_0 and C_0 (which is 0) are added resulting in sum S_0 and carry C_1. This carry C_1 becomes the carry input to the second stage. Similarly in the second stage, A_1, B_1 and C_1 are added resulting in sum S_1 and carry C_2, in the third stage, A_2, B_2 and C_2 are added resulting in sum S_2 and carry C_3, in the third stage, A_3, B_3 and C_3 are added resulting in sum S_3 and C_4, which is the output carry. Thus the circuit results in a sum ($S_3S_2S_1S_0$) and a carry output (C_{out}).

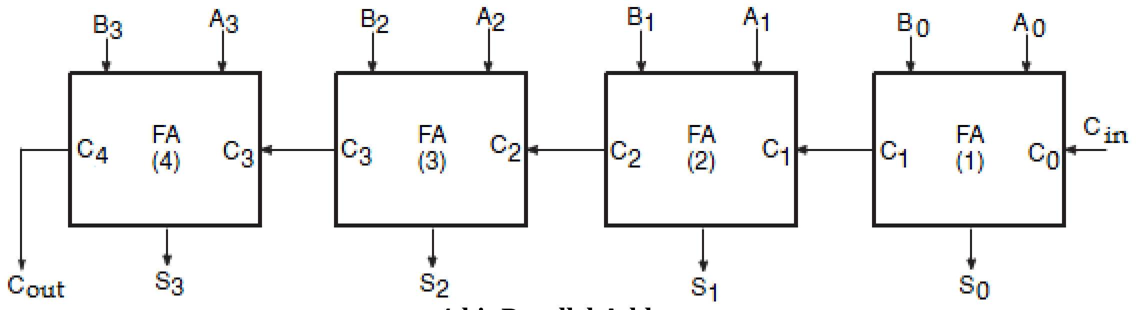
Though the parallel binary adder is said to generate its output immediately after the inputs are applied, its speed of operation is limited by the carry propagation delay through all stages. However, there are several methods to reduce this delay.

One of the methods of speeding up this process is look-ahead carry addition which eliminates the ripple-carry delay.

<u>Carry Propagation–Look-Ahead Carry Generator:</u>

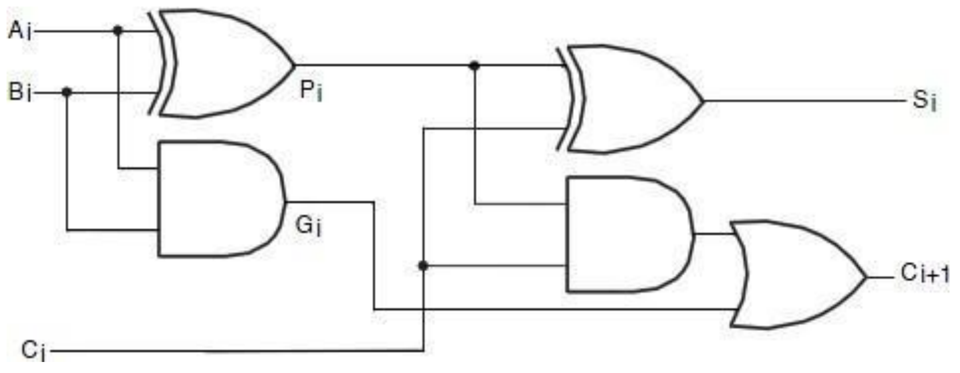
In Parallel adder, all the bits of the augend and the addend are available for computation at the same time. The carry output of each full-adder stage is connected to the carry input of the next high-order stage. Since each bit of the sum output depends on the value of the input carry, time delay occurs in the addition process. This time delay is called as **carry propagation delay.**

For example, addition of two numbers (0011+ 0101) gives the result as 1000. Addition of the LSB position produces a carry into the second position. This carry when added to the bits of the second position, produces a carry into the third position. This carry when added to bits of the third position, produces a carry into the last position. The sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous position. i.e., the adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in an each full-adder. For example, if each full adder is considered to have a propagation delay of 30nsec, then S_3 will not react its correct value until 90 nsec after LSB is generated. Therefore total time required to perform addition is 90+ 30 = 120nsec.



<u>4-bit Parallel Adder</u>

The method of speeding up this process by eliminating inter stage carry delay is called **look ahead-carry addition**. This method utilizes logic gates to look at the lower order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions: carry generate and carry propagate.



Full-Adder circuit

Consider the circuit of the full-adder shown above. Here we define two functions: carry generate (G_i) and carry propagate (P_i) as,

Carry generate, $\mathbf{G_i = A_i \oplus B_i}$

Carry propagate, $\mathbf{P_i = A_i \oplus B_i}$

the output sum and carry can be expressed as,

$$\mathbf{S_i = P_i \oplus C_i}$$

$$\mathbf{C_{i+1} = G_i \oplus P_iC_i}$$

G_i (carry generate), it produces a carry 1 when both Ai and Bi are 1, regardless of the input carry C_i.

Pi (carry propagate) because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

The Boolean functions for the carry outputs of each stage and substitute for each C_i its value from the previous equation:

C_0= input carry

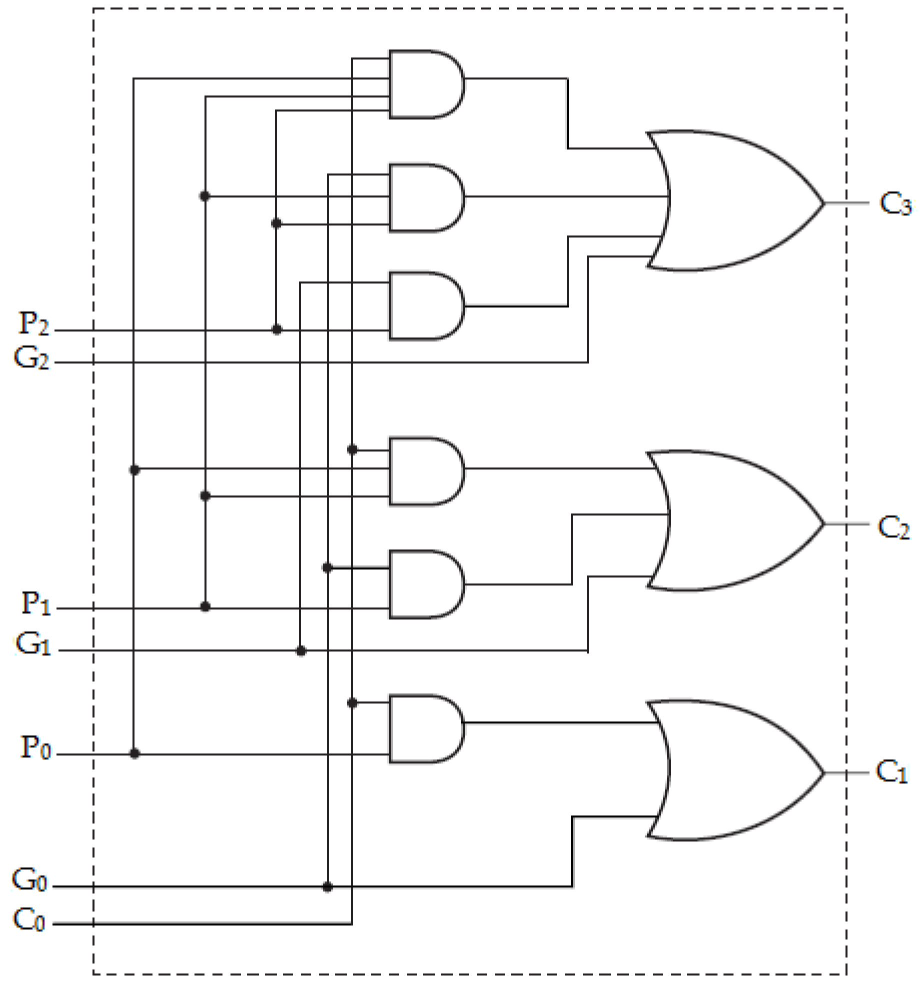
$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2 (G_1 + P_1G_0 + P_1P_0C_0)$

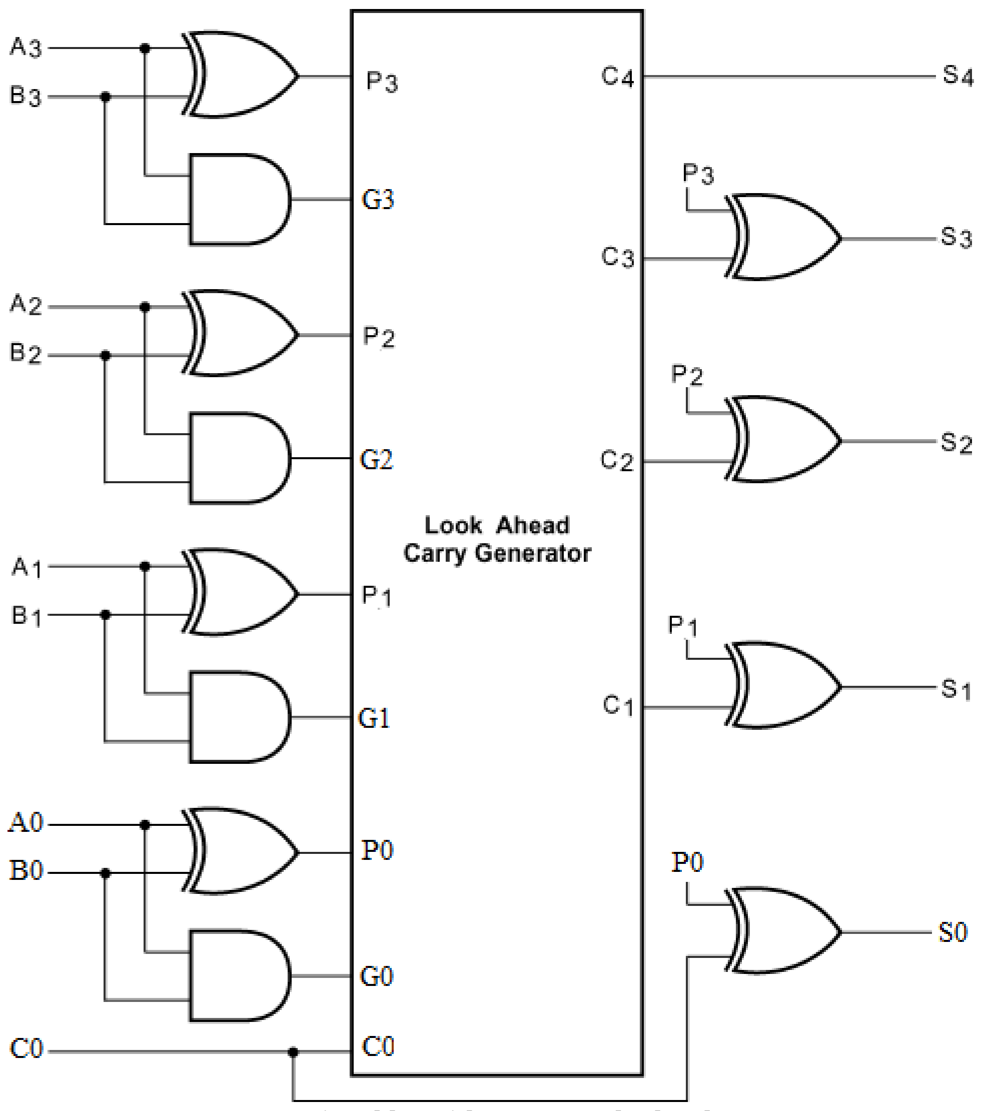
$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$



Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate. The three Boolean functions for C_1, C_2 and C_3 are implemented in the carry look-ahead generator as shown below. Note that C_3 does not have to wait for C_2 and C_1 to propagate; in fact C_3 is propagated at the same time as C_1 and C_2.

Logic diagram of Carry Look-ahead Generator

Using a Look-ahead Generator we can easily construct a 4-bit parallel adder with a Look-ahead carry scheme. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry look-ahead generator and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times.

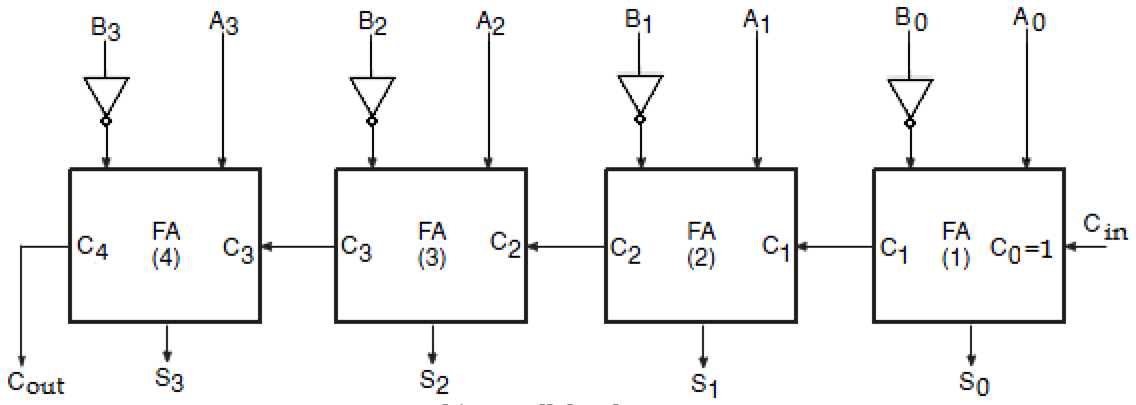


4-Bit Adder with Carry Look-ahead

Binary Subtractor (Parallel Subtractor):

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. The subtraction A-B can be done by taking the 2‘s complement of B and adding it to A. The 2‘s complement can be obtained by taking the 1‘s complement and adding 1 to the least significant pair of bits. The 1‘s complement can be implemented with inverters and a 1 can be added to the sum through the input carry.

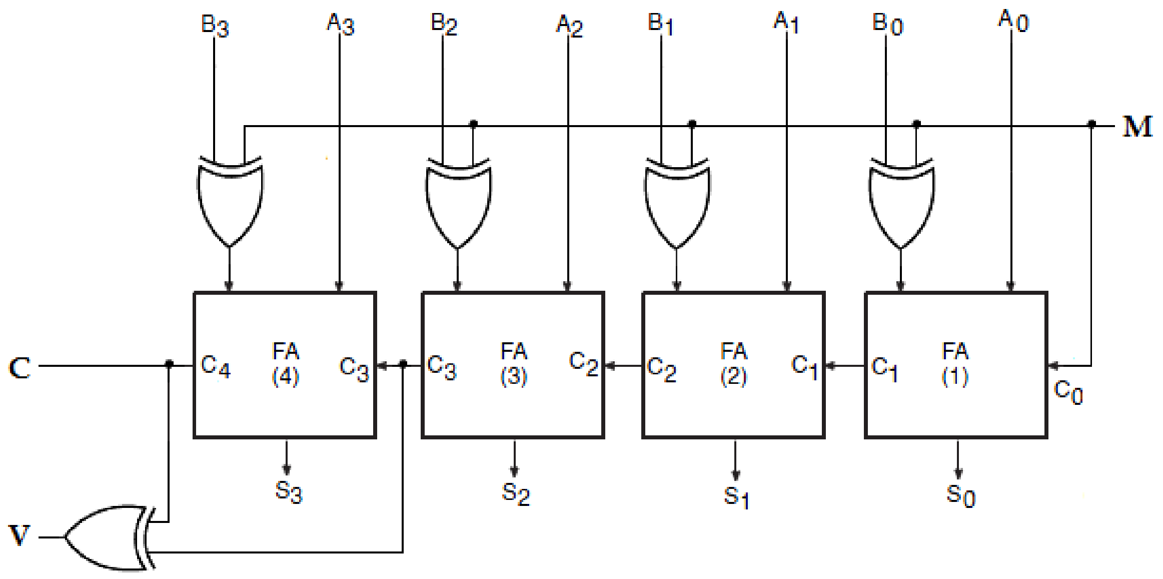
The circuit for subtracting A-B consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when performing subtraction. The operation thus performed becomes A, plus the 1‘s complement of B, plus1. This is equal to A plus the 2‘s complement of B.



4-bit Parallel Subtractor

Parallel Adder/ Subtractor:

The addition and subtraction operation can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each full adder. A 4-bit adder Subtractor circuit is shown below.



4-Bit Adder Subtractor

The mode input M controls the operation. When M= 0, the circuit is an adder and when M=1, the circuit becomes a Subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When M=0, we have B 0= B□The full adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When M=1, we have B 1= B□and C_0=1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2‘s complement of B. The exclusive-OR with output V is for detecting an overflow.

Decimal Adder (BCD Adder):

The digital system handles the decimal number in the form of binary coded decimal numbers (BCD). A BCD adder is a circuit that adds two BCD bits and produces a sum digit also in BCD.

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 is the sum being an input carry. The adder will form the sum in binary and produce a result that ranges from 0 through 19.

These binary numbers are labeled by symbols K, Z_8, Z_4, Z_2, Z_1, K is the carry. The columns under the binary sum list the binary values that appear in the outputs of the 4-bit binary adder. The output sum of the two decimal digits must be represented in BCD.

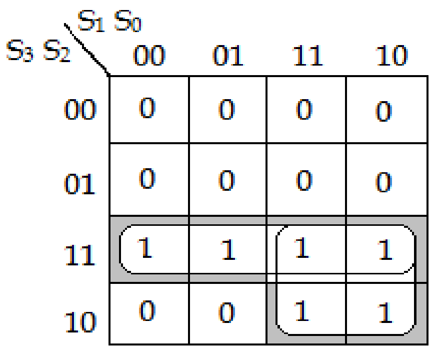
Binary Sum					**BCD Sum**					**Decimal**
K	**Z_8**	**Z_4**	**Z_2**	**Z_1**	**C**	**S_8**	**S_4**	**S_2**	**S_1**	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	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

In examining the contents of the table, it is 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 9 (1001), we obtain a non-valid 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 to detect sum greater than 9 can be determined by simplifying the boolean expression of the given truth table.

Inputs				Output
S_3	S_2	S_1	S_0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

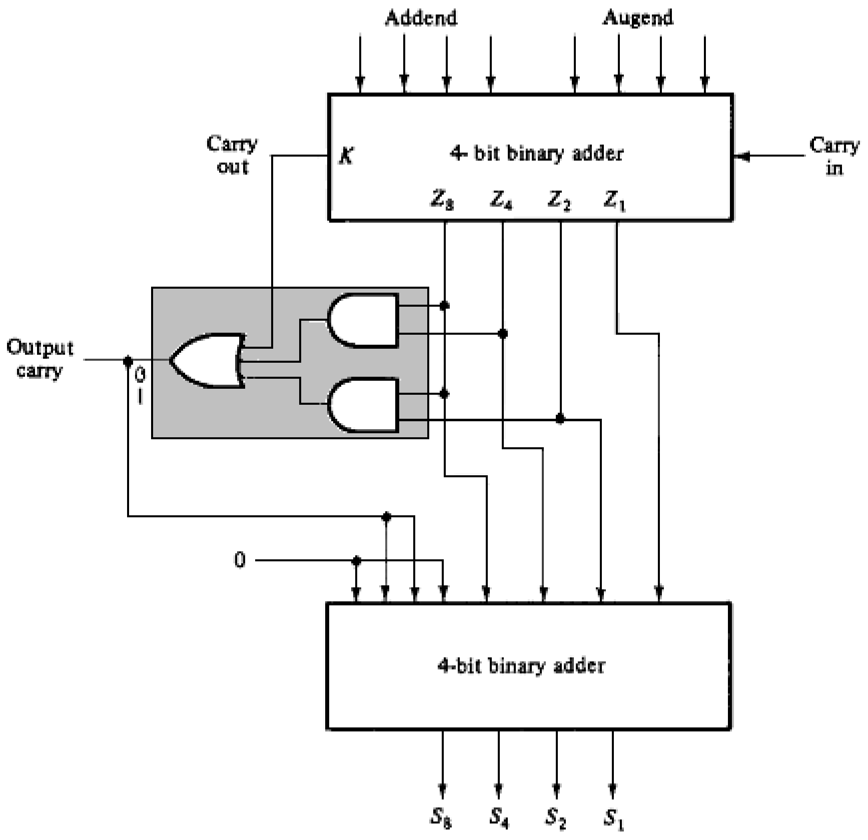


$Y = S_3S_2 + S_3S_1$

To implement BCD adder we require:

- 4-bit binary adder for initial addition
- Logic circuit to detect sum greater than 9 and
- One more 4-bit adder to add 0110_2 in the sum if the sum is greater than 9 or carry is 1.

The two decimal digits, together with the input carry, are first added in the top4-bit binary adder to provide the binary sum. When the output carry is equal to zero, nothing is added to the binary sum. When it is equal to one, binary 0110 is added to the binary sum through the bottom 4-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. The output carry from one stage must be connected to the input carry of the next higher-order stage.

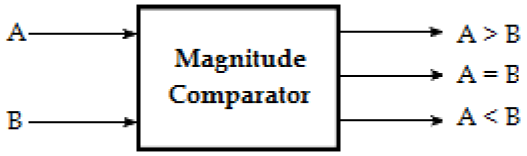


Block diagram of BCD adder

MAGNITUDE COMPARATOR:

A ***magnitude comparator*** is a combinational circuit that compares two given numbers (A and B) and determines whether one is equal to, less than or greater than the other. The output is in the form of three binary variables representing the conditions A

= B, A>B and A<B, if A and B are the two numbers being compared.



Block diagram of magnitude comparator

Truth table

A	B	A<B	A=B	A>B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

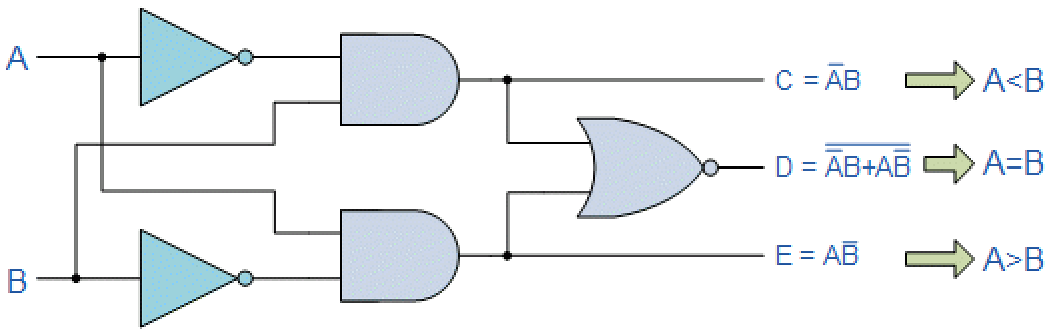
Logical expression

A<B – A’B

A>B – AB’

A=B – A’B’+AB

Logical design



For comparison of two *n*-bit numbers, the classical method to achieve the Boolean expressions requires a truth table of 2^{2n} entries and becomes too lengthy and cumbersome.

2-bit Magnitude Comparator:

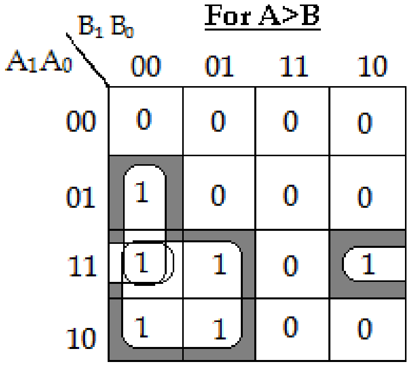
The truth table of 2-bit comparator is given in table below—

Truth table:

Inputs				**Outputs**		
A_3	**A_2**	**A_1**	**A_0**	**A>B**	**A=B**	**A<B**
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

K-map Simplification**:**

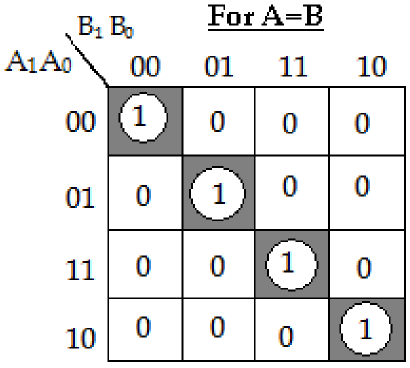
For A>B



$A_1A_0 \backslash B_1B_0$	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$A>B = A_0B_1'B_0' + A_1B_1' + A_1A_0B_0'$

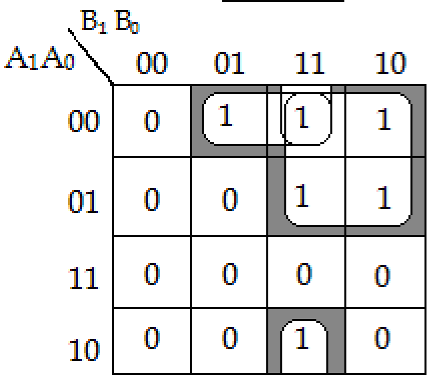
For A=B



$A_1A_0 \backslash B_1B_0$	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$$\begin{aligned} A=B &= A_1'A_0'B_1'B_0' + A_1'A_0B_1'B_0 + A_1A_0B_1B_0 + A_1A_0'B_1B_0' \\ &= A_1'B_1'(A_0'B_0' + A_0B_0) + A_1B_1(A_0B_0 + A_0'B_0') \\ &= (A_0 \odot B_0)(A_1 \odot B_1) \end{aligned}$$

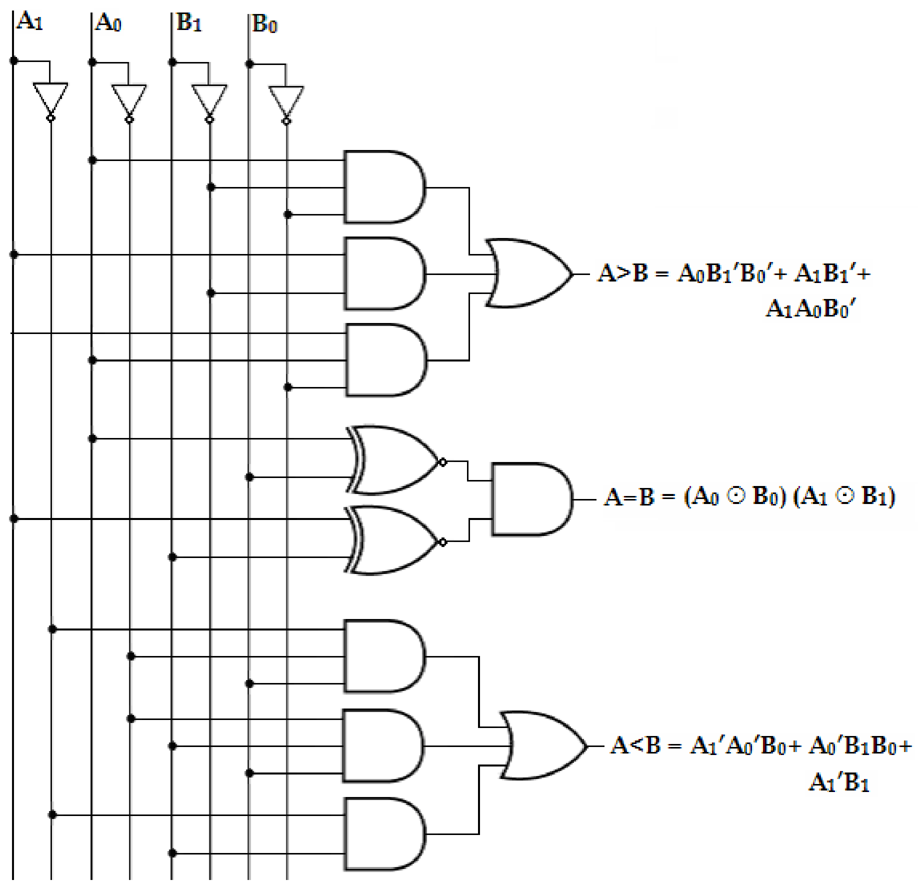
For A<B



$A_1A_0 \backslash B_1B_0$	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$A<B = A_1'A_0'B_0 + A_0'B_1B_0 + A_1'B_1$

Logic Diagram:



A_1
A_0
B_1
B_0
$A>B = A_0B_1'B_0' + A_1B_1' + A_1A_0B_0'$
$A=B = (A_0 \odot B_0)(A_1 \odot B_1)$
$A<B = A_1'A_0'B_0 + A_0'B_1B_0 + A_1'B_1$

2-bit Magnitude Comparator