UNIT-II Combinational Logic

Combination Logic (Circuit)

- In *combinational circuit, the* output at any time depends only on the present combination of inputs at that point of time irrespective to the past state of the 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.

Implementing Combinational Logic

- The different steps involved in the design of a combinational logic circuit are as follows:
  1. Statement of the problem.
  2. Identification of input and output variables.
  3. Expressing the relationship between the input and output variables.
  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. Minimization of Boolean expressions.
  7. Implementation of minimized Boolean expressions.

Half Adder

- A *half-adder* is an arithmetic circuit block that can be used to add two bits.
- Such a circuit thus has two inputs which represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.
- Figure 2.1 shows the truth table of a half-adder, showing all possible input combinations and the corresponding outputs

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

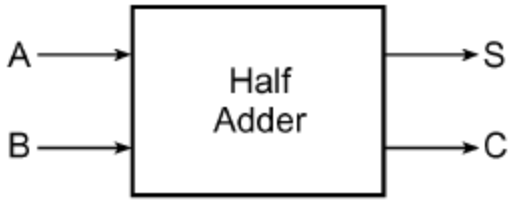


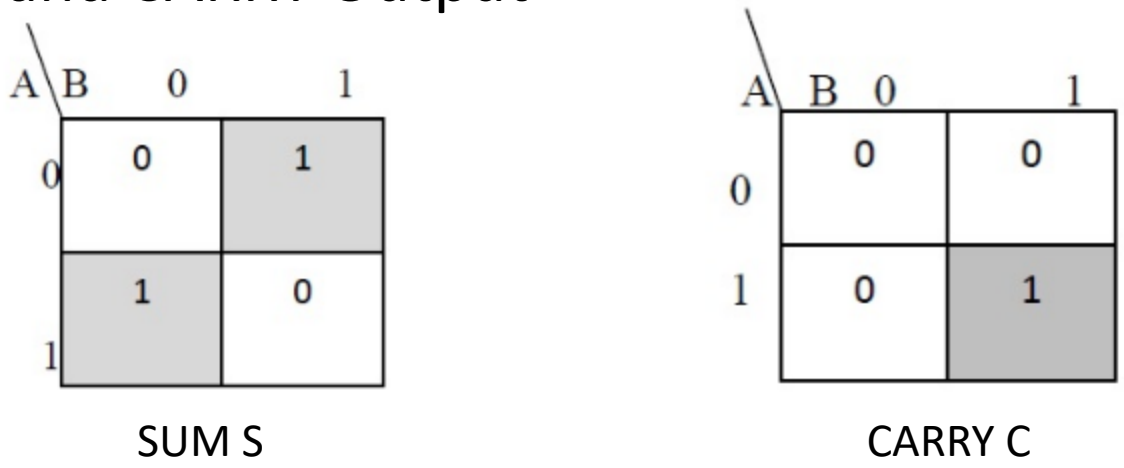
Figure 2.1 Truth Table of Half Adder

- The Boolean expressions for the SUM and CARRY outputs from the above truth table are given by the equations

$$\text{SUM } S = A.\overline{B} + \overline{A}.B \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (2.1)$$

$$\text{CARRY } C = A.B \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (2.2)$$

- Figure 2.2 shows the K-Map Simplification of the SUM output and CARRY Output



SUM S CARRY C

Figure 2.2 K-Map simplification of Half Adder

- In the above two expressions there is no scope for further simplification.
- While 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.

- Figure 2.3 shows the simplest way for implementing a half-adder, where a two-input EX-OR gate is used for the SUM output and a two-input AND gate is used for the CARRY output

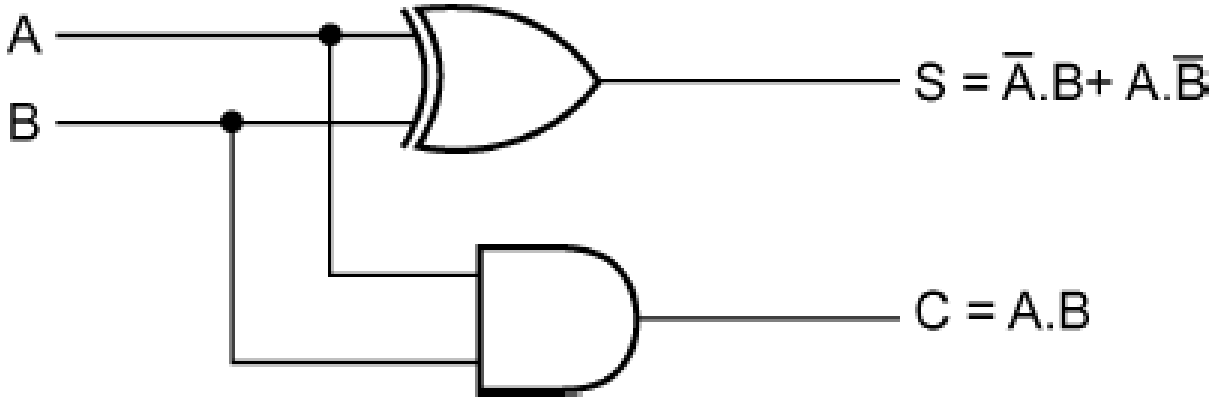


Figure 2.3 Logic Implementation of Half Adder

- The Half-Adder can be implemented by using an appropriate arrangement of either NAND or NOR gates.
- Figure 2.4 shows the implementation of a half-adder with NAND gates only. Total 5 NAND gates are required to design Half-Adder.

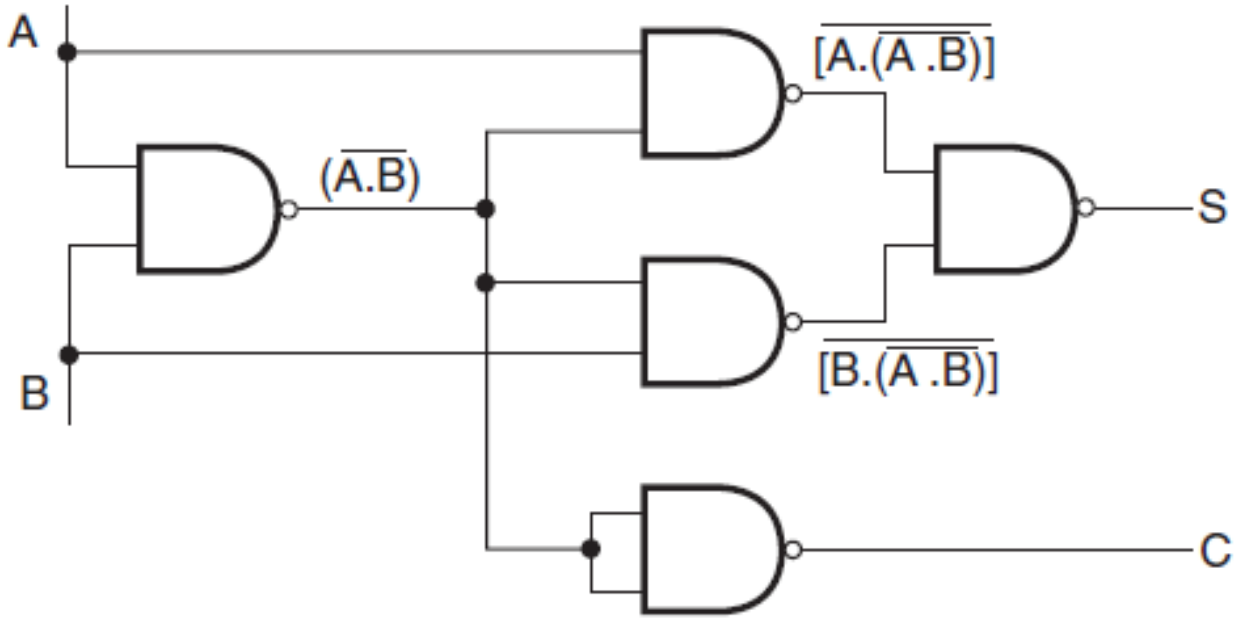


Figure 2.4 Implementation of Half Adder using NAND Gates

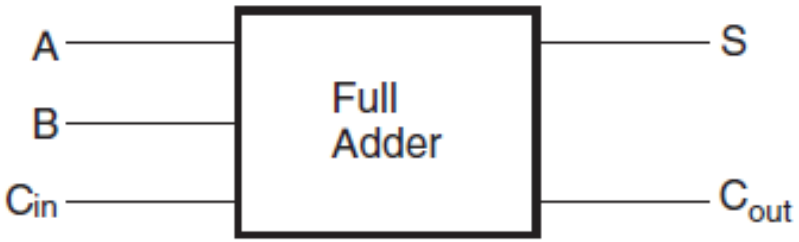
Limitations of Half Adder

- The reason these simple binary adders are called Half Adders is because there is no scope for them to add the carry bit from previous bit.
- This is a major limitation of half adders when used as binary adders especially in real time scenarios which involves addition of multiple bits.
- To overcome this limitation, full adders are developed.

Full Adder

- A *Full Adder* circuit is an arithmetic circuit block that can be used to add three bits to produce a SUM and a CARRY output.
- Such a building block becomes a necessity when it comes to adding binary numbers with a large number of bits.
- The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only.
- When adding larger binary numbers, we begin with the addition of LSBs of the two numbers. We record the sum under the LSB column and take the carry, if any, forward to the next higher column bits.
- As a result, when we add the next adjacent higher column bits, we would be required to add three bits if there were a carry from the previous addition.

- We have a similar situation for the other higher column bits also until we reach the MSB.
- A full adder is therefore essential for the hardware implementation of an adder circuit capable of adding larger binary numbers.
- Figure 2.5 shows the truth table of a Full Adder circuit showing all possible input combinations and corresponding outputs.



A	B	C_{in}	SUM (S)	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

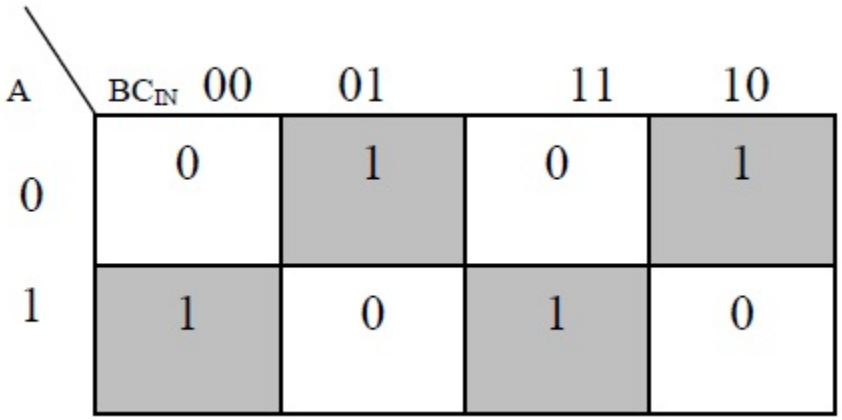
Figure 2.5 Truth Table of Full Adder

- From this truth table, the Boolean expressions for the SUM output (S) and for the CARRY output (Cout) are:

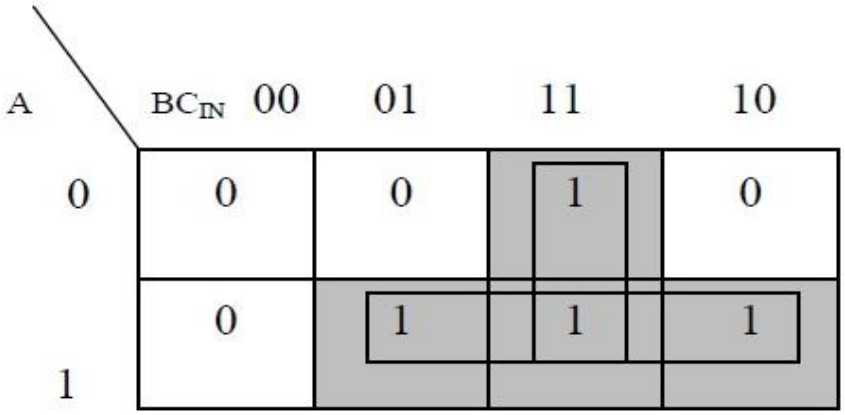
$$S = \overline{A}.\overline{B}.C_{in} + \overline{A}.B.\overline{C}_{in} + A.\overline{B}.\overline{C}_{in} + A.B.C_{in} \quad \ldots\ldots\ldots\ldots \quad (2.3)$$

$$C_{out} = \overline{A}.B.C_{in} + A.\overline{B}.C_{in} + A.B.\overline{C}_{in} + A.B.C_{in} \quad \ldots\ldots\ldots\ldots \quad (2.4)$$

- The above two expressions can be simplified using K-Map technique. K-maps for the two expressions are given in Fig. 2.6(a) for the SUM output and Fig. 2.8(b) for the



(a) SUM S



(b) CARRY Cout

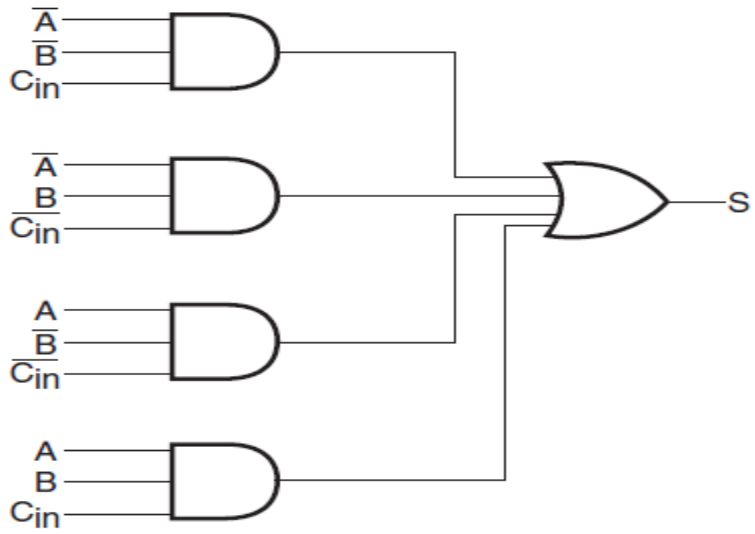
Figure 2.6 K-Map Simplification of Full Adder

- The Boolean expressions for the SUM (S) and CARRY (Cout) from the above K-Map are given by the equations

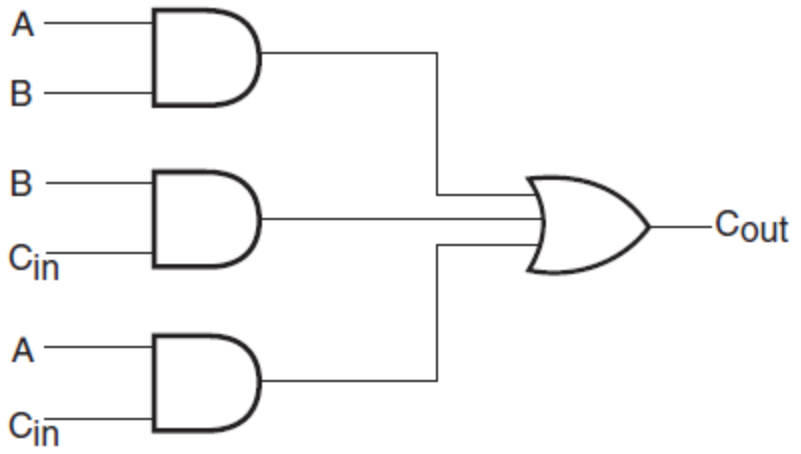
$$S = \bar{A}.\bar{B}.C_{in} + \bar{A}.B.\overline{C_{in}} + A.\bar{B}.\overline{C_{in}} + A.B.C_{in} \quad \ldots\ldots\ldots (2.5)$$

$$C_{out} = B.C_{in} + A.B + A.C_{in} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.6)$$

- In order to implement a combinational circuit for Full Adder, it is clear from the above equations, that we need 4 three input AND gates and 1 four input OR gate for Sum and 3 two input AND gates and 1 three input OR gate for Carry – out.
- The logic circuit for full adder as shown in Figure 2.7.



(a) Sum S



(b) Carry Cout

Figure 2.7 Logic Circuit Diagram of Full Adder

Implementation of Full Adder Using Half Adders

- A full adder can be implemented by logically connecting two half adders.
- Figure 2.8 shows the block diagram for implementation of a full adder using two half adders

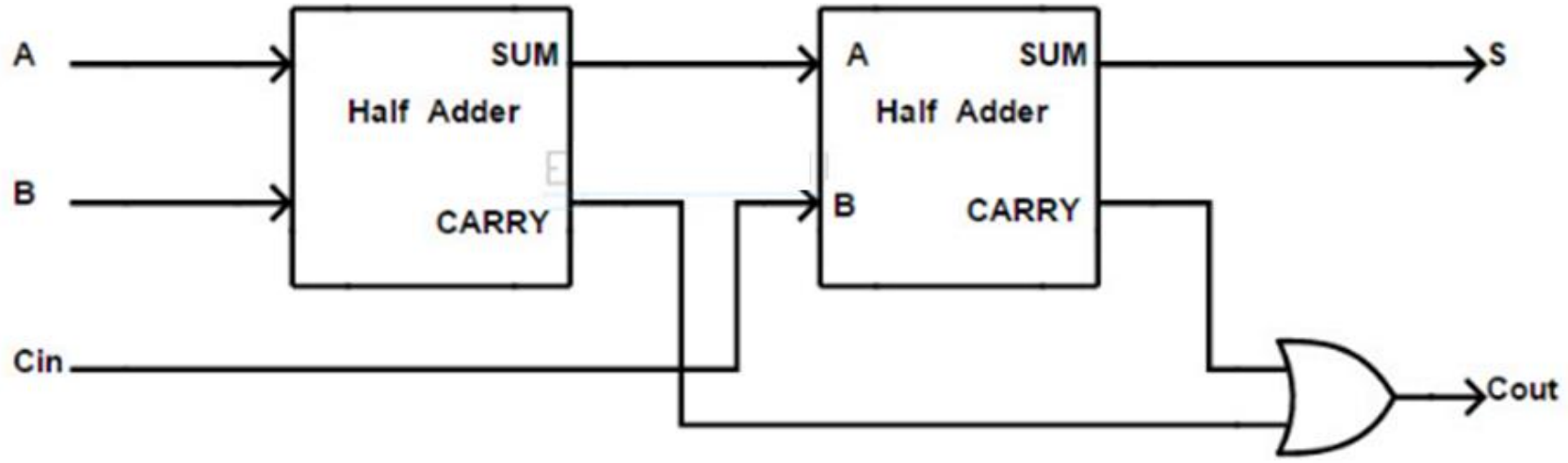


Figure 2.8 Block Diagram for Implementation of Full Adder using Half Adder

- The Boolean expressions for the SUM (S) and CARRY (Cout) can be rewritten as

$$
\begin{aligned}
S &= \bar{A}.\bar{B}.C_{in} + \bar{A}.B.\overline{C_{in}} + A.\bar{B}.\overline{C_{in}} + A.B.C_{in} \\
&= C_{in}(\bar{A}.\bar{B} + A.B) + \overline{C_{in}}(\bar{A}.B + A.\bar{B}) \\
&= C_{in}(\overline{A \oplus B}) + \overline{C_{in}}(A \oplus B) \\
&= A \oplus B \oplus C_{in} \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.7)
\end{aligned}
$$

$$
\begin{aligned}
C_{out} &= B.C_{in} + A.B + A.C_{in} \\
&= B.C_{in}(\bar{A} + A) + A.C_{in} + A.B \\
&= A.B.C_{in} + AB + A.C_{in} + \bar{A}.B.C_{in} \\
&= A.B(C_{in} + 1) + A.C_{in} + \bar{A}.B.C_{in} \\
&= AB + A.C_{in} + \bar{A}.B.C_{in} \\
&= AB + A.C_{in}(\bar{B} + B) + \bar{A}.B.C_{in} \\
&= AB + A.C_{in}B + A.C_{in}\bar{B} + \bar{A}.B.C_{in} \\
&= AB(1 + C_{in}) + C_{in}(A.\bar{B} + \bar{A}.B) \\
&= AB + C_{in}(A \oplus B) \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.8)
\end{aligned}
$$

- From the above two equations, the full adder circuit can be implemented using two half adders and an OR gate.
- The implementation of full adder using two half adders is show below Figure 2.9

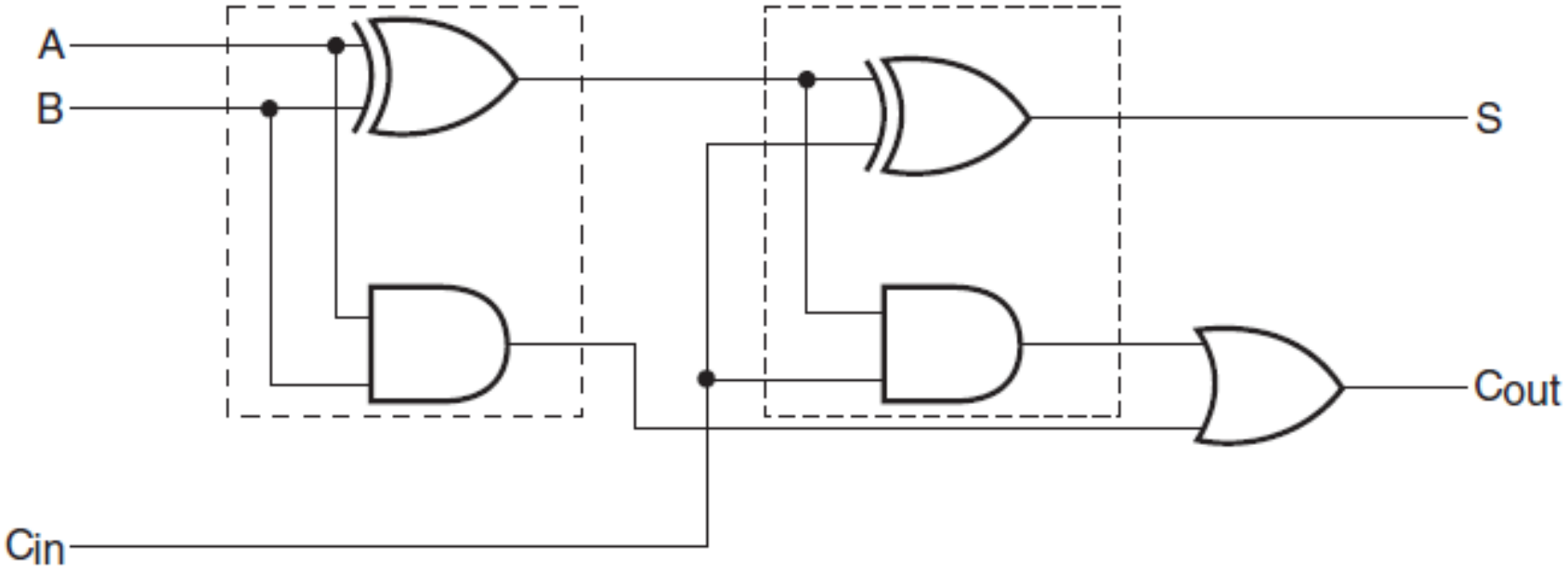


Figure 2.9 Logic Diagram for Implementation of Full Adder using Half Adder

Full Adder using NAND Gates

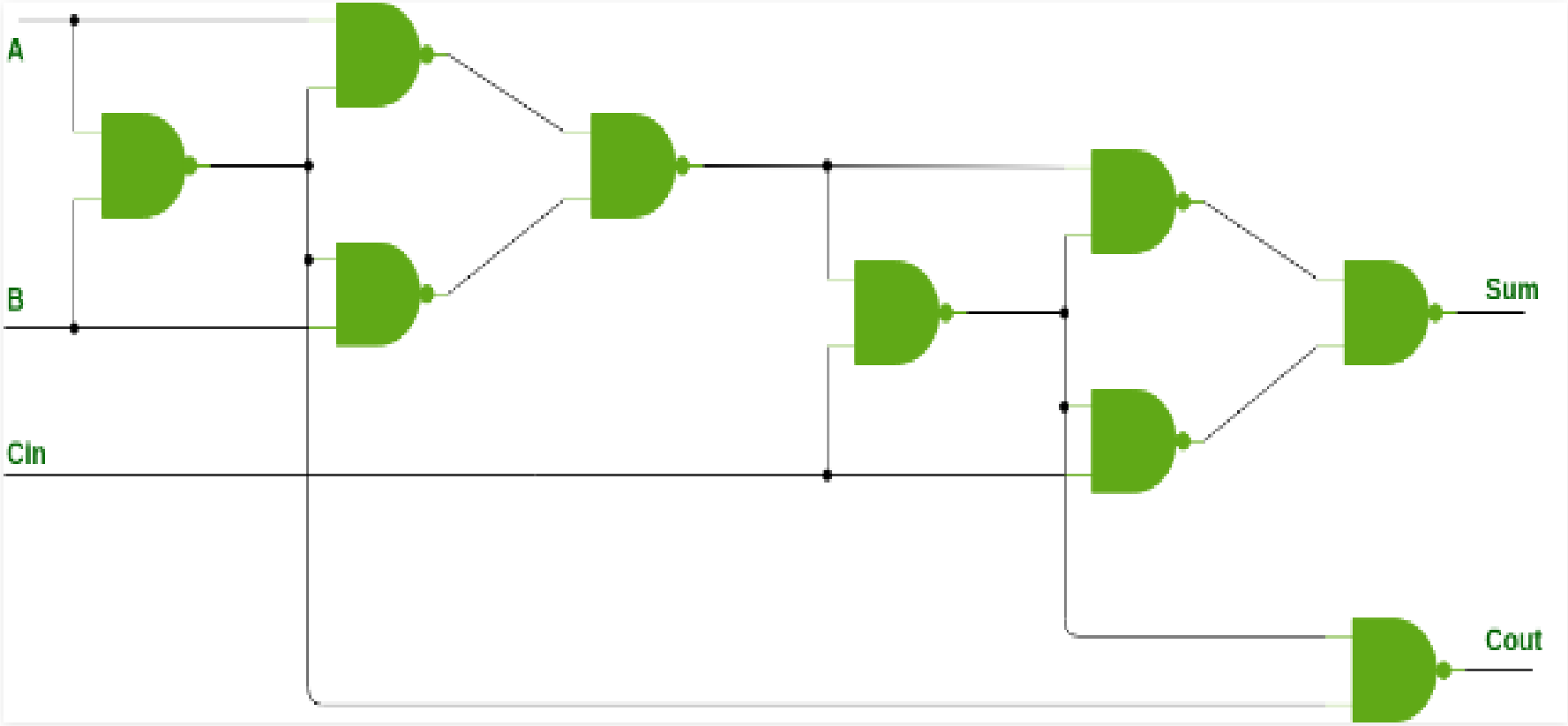


Figure 2.10 Full Adder using only NAND Gates

Full Adder using NOR Gates

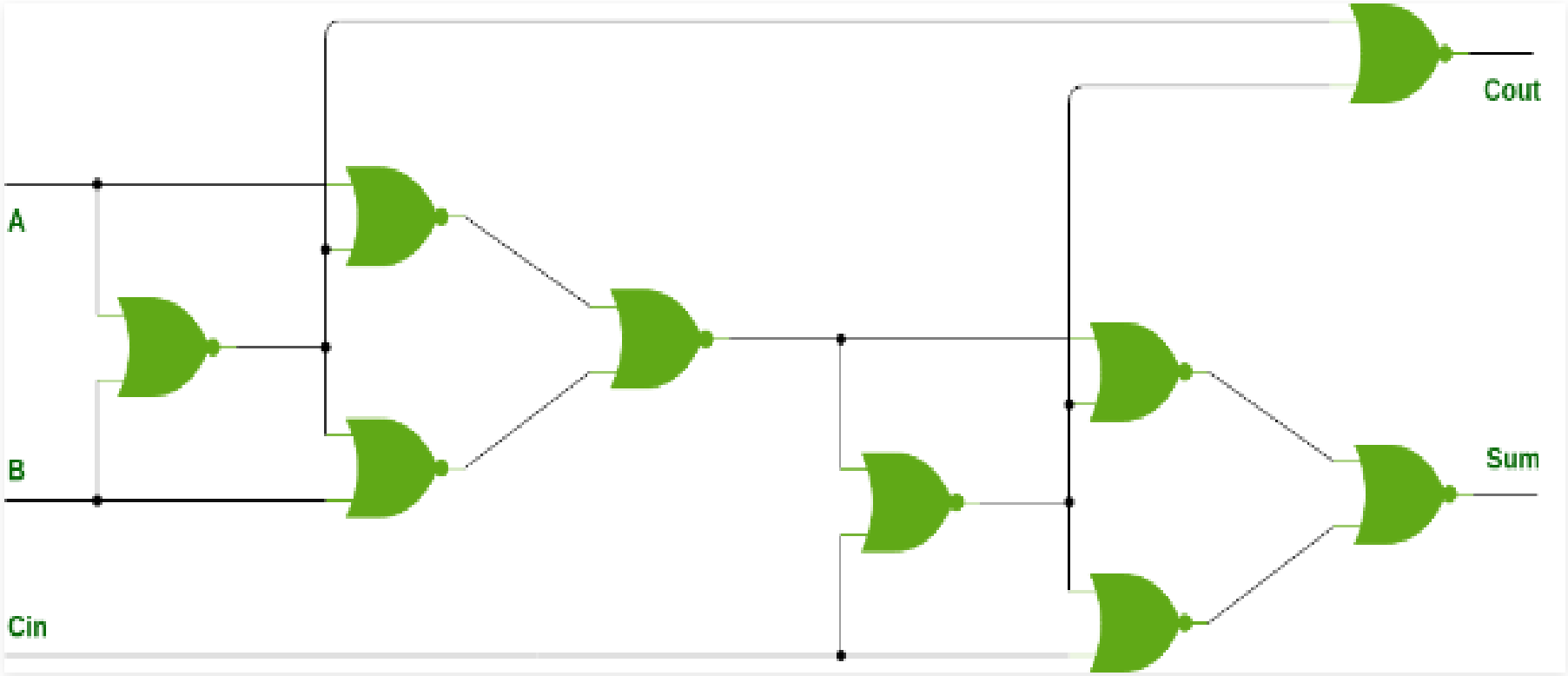
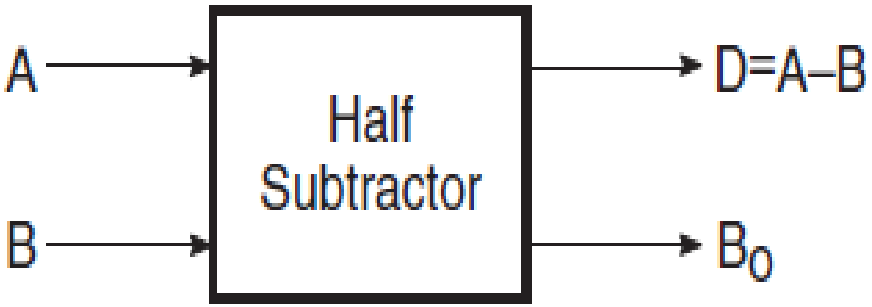


Figure 2.11 Full Adder using only NOR Gates

Half Subtractor

- A *half-subtractor* is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE (D) output and a BORROW (Bo) output.
- The BORROW output here specifies whether a '1' has been borrowed to perform the subtraction.
- The truth table of a half-subtractor, as shown in Fig. 2.12, explains this further.



A	B	D	B_0
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Figure 2.12 Truth table for Half Subtractor

- The Boolean expressions for the two outputs are given by the equations

$$D = \bar{A}.B + A\bar{B} \quad \text{..... (2.9)}$$

$$B_o = \bar{A}.B \quad \text{.....(2.10)}$$

- The expression for the DIFFERENCE (D) output is that of an EX-OR gate, whereas the expression for the BORROW output (Bo) is that of an AND gate with input A complemented before it is fed to the gate.
- Figure 2.13 shows the logic implementation of a half-subtractor.

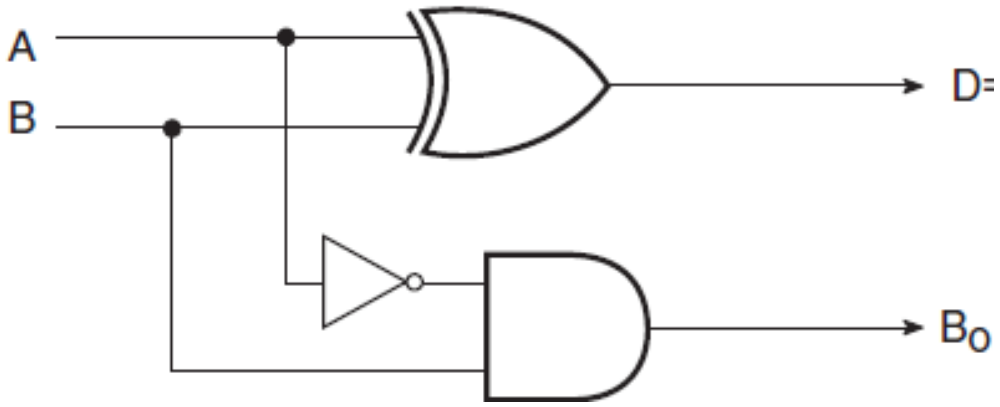
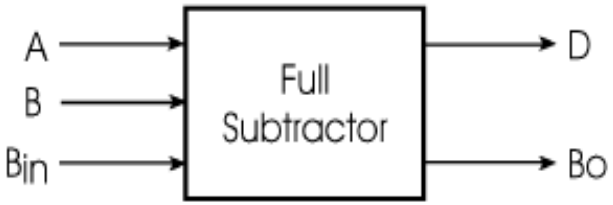


Figure 2.13 Logic Implementation for Half Subtractor

- 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.
- Therefore, by only complementing the minuend input A, that is, the minuend, a half adder can be converted into a half subtractor.
- One major disadvantage of the *Half Subtractor* circuit when used as a binary subtractor, is that there is no provision for a "Borrow-in" from the previous circuit when subtracting multiple data bits from each other. Therefore there is a need to implement a "full binary subtractor" circuit to take into account this borrow-in input from a previous circuit.

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 Bin .
- There are two outputs, namely the DIFFERENCE output D and the BORROW output Bo.
- The BORROW output bit tells whether the minuend bit needs to borrow a ‘1’ from the next possible higher minuend bit.
- Figure 2.14 shows the truth table of a full subtractor.



Minuend (A)	Subtrahend (B)	Borrow In (B_{in})	Difference (D)	Borrow Out (B_o)
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

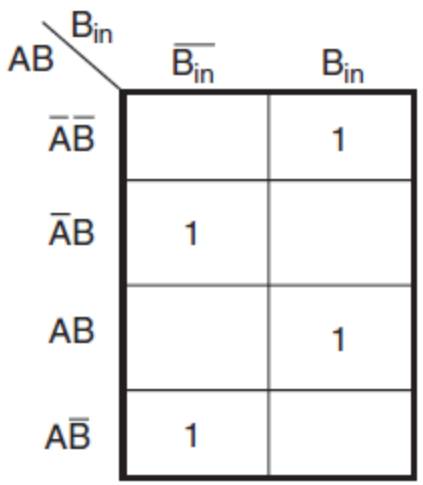
Figure 2.14 Truth Table for Full Subtractor

- The Boolean expressions for the two output variables are given by the following equations

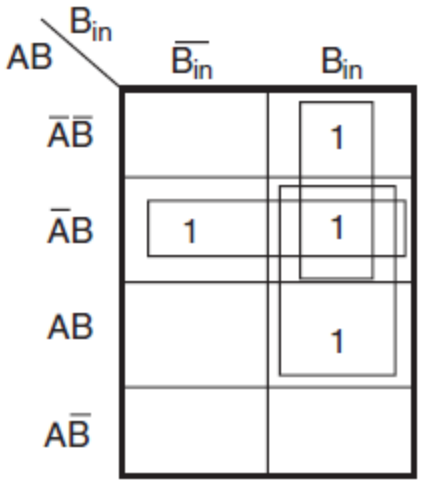
$$D = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + A.\overline{B}.\overline{B}_{in} + A.B.B_{in} \quad \text{..... (2.11)}$$

$$B_o = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + \overline{A}.B.B_{in} + A.B.B_{in} \quad \text{..... (2.12)}$$

- The K-maps for the two expressions are given in Fig. 2.15(a) for DIFFERENCE output D and in Fig. 2.15(b) for BORROW output Bo.



(a) Difference D



(b) Borrow Bo

Figure 2.15 K-Map for Full Subtractor

- From the above K-maps, it is clear that no simplification is possible for the difference output D, whereas for Borrow Bo the simplified equation is given as:

$$B_o = \overline{A}.B + \overline{A}.B_{in} + B.B_{in} \qquad \text{…….. (2.13)}$$

- The equations 2.11 and 2.13 can be further simplified and rewritten as

$$D = A \oplus B \oplus B_{in} \quad \text{..... (2.14)}$$

$$B_o = \bar{A}.B + B_{in}(\overline{A \oplus B}) \quad \text{..... (2.15)}$$

- The logic implementation of full subtractor is given in Figure 2.16.

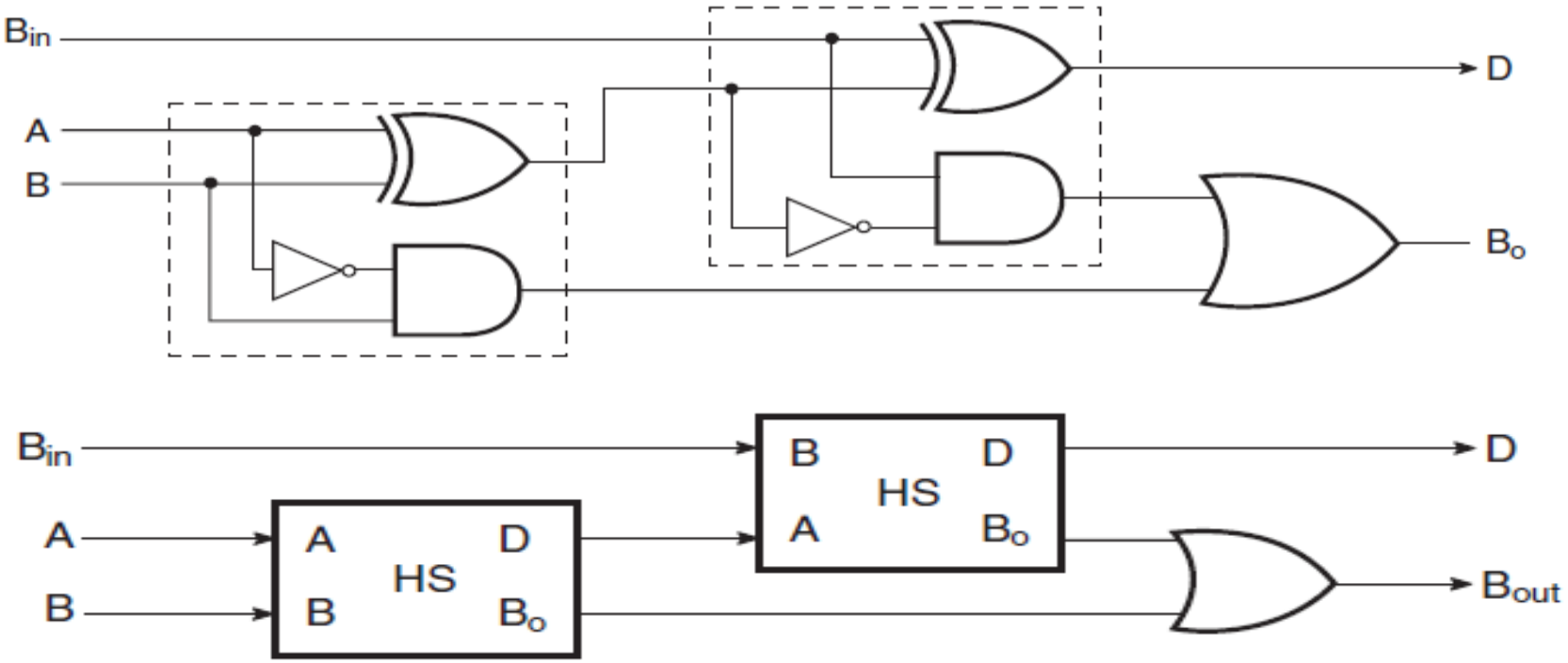


Figure 2.16 Logic Implementation of Full Subtractor

- If we compare these expressions with those derived earlier in the case of a full adder, we find that the expression for DIFFERENCE output D is the same as that for the SUM output.
- Also, the expression for BORROW output Bo is similar to the expression for CARRY-OUT Co.
- Figure 2.16 also shows that a full subtractor can be implemented with half-subtractors. It is done in the same way as a full adder was constructed using half-adders.

Binary Adder (7483)

- **Design and explain 8 bit binary adder using IC 7483**

- IC 7483 is a 4 bit parallel adder which consists of four interconnected full adders along with the look ahead carry circuit. The pin diagram of IC 7483 is as shown in Figure 2.17. It is a 16pin IC. The inputs to the IC are A, B and C_{in0} while outputs are S and C_{out3}.

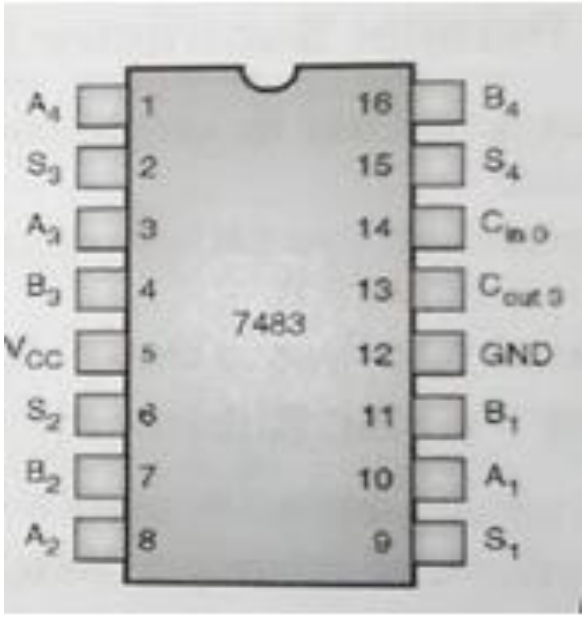


Figure 2.17 Pin Diagram of IC 7483

- $A_3A_2A_1A_0$ is a 4 bit input word 'A' and $B_3B_2B_1B_0$ is the second 4 bit input word 'B'. C_{in0} is the input carry. The IC adds the two four bit words along with input carry to produce a 4 bit sum and a one bit carry-out. C_{out3} represents the output carry. S_3,S_2,S_1,S_0 represents sum output with S_3 as the MSB.
- In order to design an 8 bit adder, we require two IC 7483s cascaded as shown in the figure 2.18.

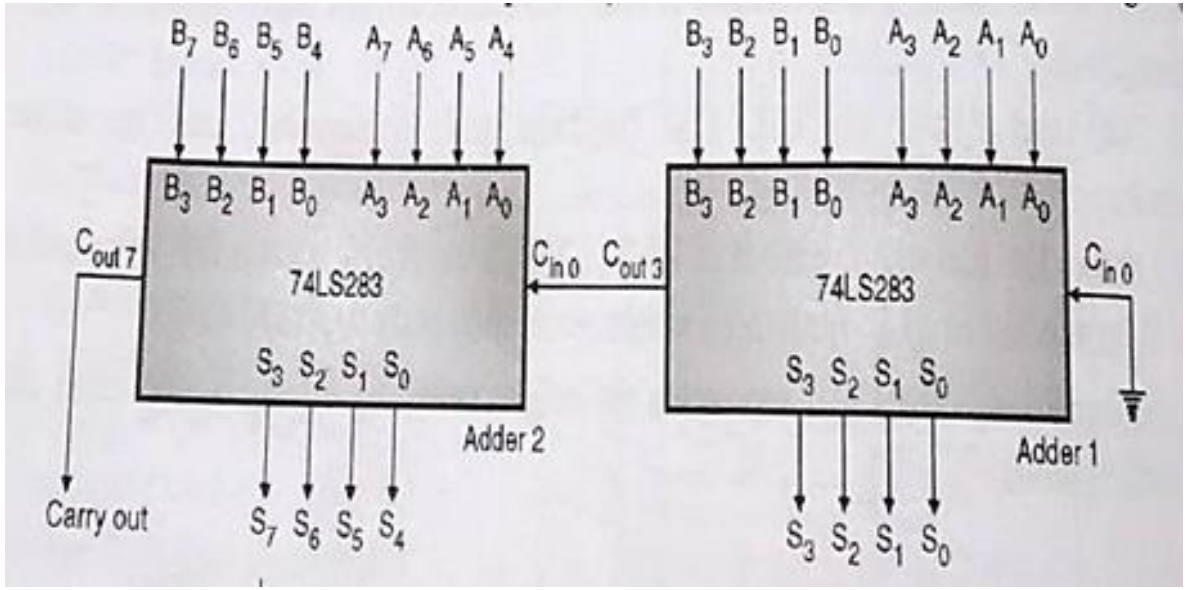


Figure 2.18 Cascading of two 7483 IC's

- Adder-1 is the LSB adder and it adds the four LSB bits of the two 8-bit input words ie A_3–A_0 and B_3–B_0.
- The carry input of first adder is supposed to be 0. Hence the carry in pin of LSB IC is connected to the ground.
- So the first IC adds the LSB bits of A and B, and produces S_3–S_0 that is, LSB of sum, along with a carry out C_{out3}.
- This C_{out3} of adder-1 is connected to C_{in0} input of Adder-2. The second adder adds this carry and the four MSB bits of numbers A_3–A_0 and B_3–B_0 to produce MSB sum S_7–S_4 along with final carry out bit C_{out7}.
- Thus adder-1 and adder-2 when cascaded as shown in the figure can add two 8-bit words. C_{out7} of adder-2 acts as the final output carry and the sum output is from S_7 though S_0.

BCD Adder

- A *BCD adder* is used to perform the addition of BCD numbers.
- A BCD digit can have any of the ten possible four-bit binary representations, that is, 0000, 0001, … , 1001, the equivalent of decimal numbers 0, 1, … , 9.
- When we set out to add two BCD digits and we assume that there is an input carry too, the highest binary number that we can get is the equivalent of decimal number 19 (9+9+1).
- This binary number is going to be (10011)2.
- On the other hand, if we do BCD addition, we would expect the answer to be (0001 1001)BCD.
- And if we restrict the output bits to the minimum required, the answer in BCD would be (1 1001)BCD.

- Table 2.1 lists the possible results in binary and the expected results in BCD when we use a four-bit binary adder (7483) to perform the addition of two BCD digits.

Table 2.1: Results in binary and the expected results in BCD using a four-bit binary adder to perform the addition of two BCD digits.

Decimal sum	Binary sum					BCD sum				
	K	Z_3	Z_2	Z_1	Z_0	C	S_3	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

- It is clear from the table that, as long as the sum of the two BCD digits remains equal to or less than 9, the four-bit adder produces the correct BCD output. The binary sum and the BCD sum in this case are the same. Therefore, in this case the BCD is a valid BCD and hence represent as 0 in column ‘C’.
- It is only when the sum is greater than 9 that the two results are different. It can also be seen from the table that, for a decimal sum greater than 9 (or the equivalent binary sum greater than 1001), if we add 0110 to the binary sum, we can get the correct BCD sum and the desired carry output too. Hence, in this case the BCD is an invalid BCD and therefore, represented as 1 in column ‘C’.
- The K-map for Z1, Z2, Z3 and Z4 as input and C as output from 0000 to 1111 is as shown in below figure 2.19

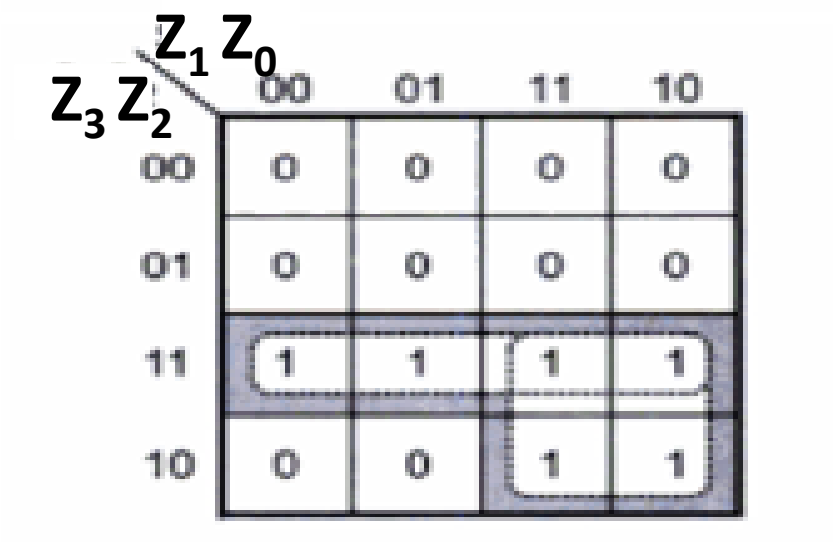


Figure 2.19 Truth Table

- The Boolean expression from the above K-Map is given as follows

$$C = Z_3.Z_2 + Z_3.Z_1 \quad \text{..... (2.16)}$$

- The equation 2.16 implies that, a correction needs to be applied whenever both Z_3 and Z_2 are '1'. This takes care of the four entries from the bottom, corresponding to a decimal sum equal to 12, 13, 14 and 15.
- For other two entries corresponding to a decimal sum equal to 10 and 11, a correction is applied for both Z_3 and Z_1, being '1'.

- A correction also needs to be applied whenever K = 1, which is applicable for the last four entries.
- The Final Boolean expression that can apply the necessary correction is written as

$$C = K + Z_3.Z_2 + Z_3.Z_1 \qquad \text{..... (2.17)}$$

- While hardware-implementing, 0110 can be added to the binary sum output with the help of a second four-bit binary adder.
- The correction logic as dictated by the Boolean expression (2.17) should ensure that (0110) gets added only when the above expression is satisfied.
- Otherwise, the sum output of the first binary adder should be passed on as such to the final output, which can be accomplished by adding (0000) in the second adder.
- Figure 2.20 shows the logic arrangement of a BCD adder capable of adding two BCD digits with the help of two four-bit binary adders and some additional combinational logic.

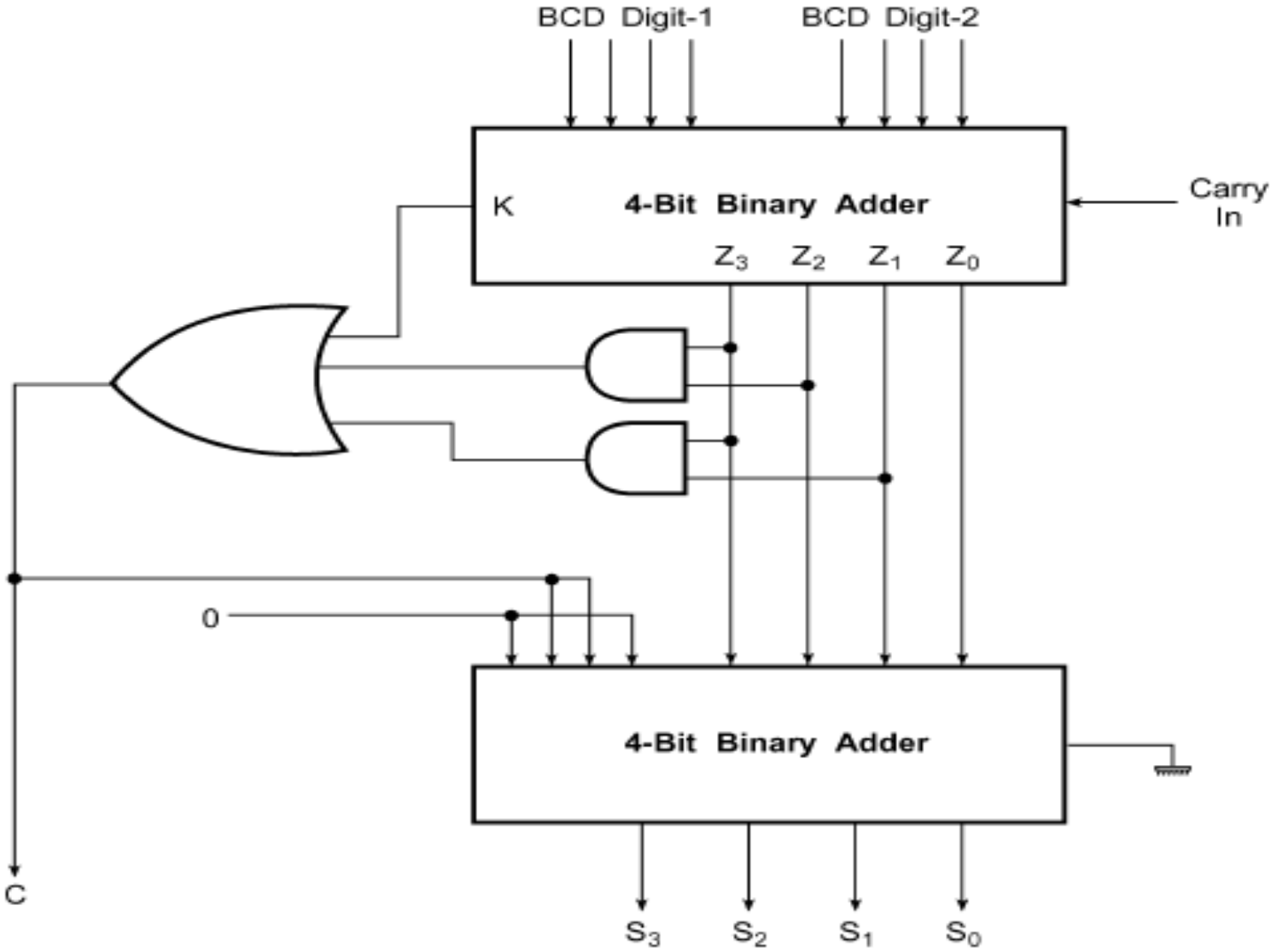


Figure 2.20 Logic Implementation of BCD Adder

Three Cases of BCD Addition

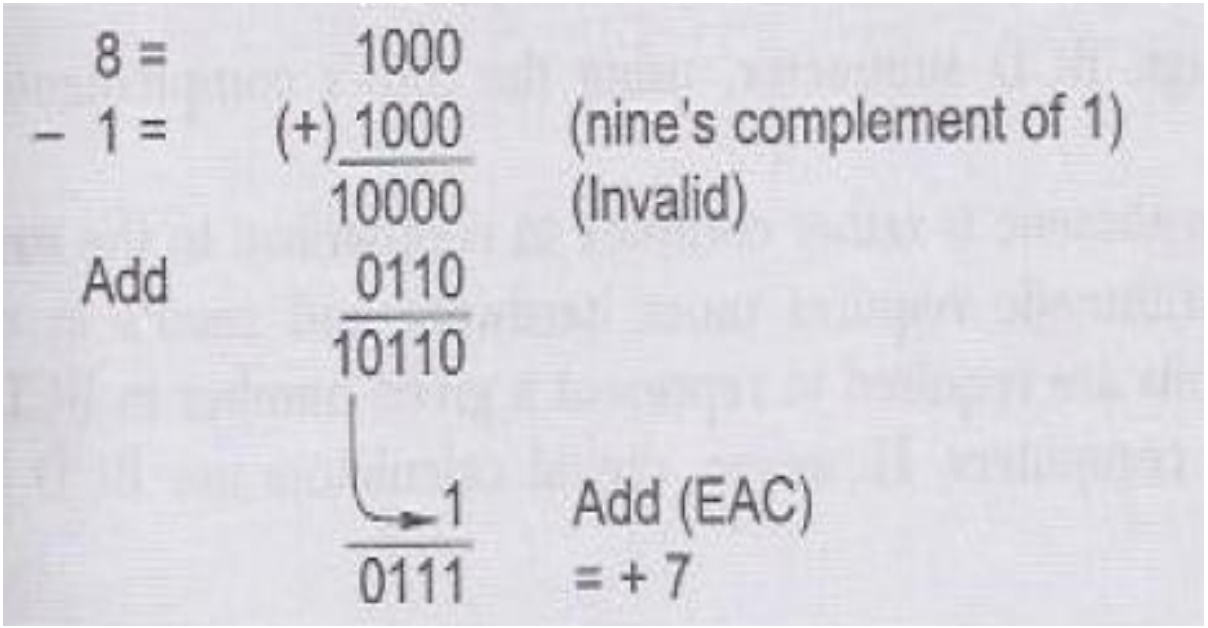
Case 1: If the sum of minuend and nine's complement of subtrahend is an invalid BCD code, add decimal 6 (Binary 0110) and end around carry to this sum. The result is positive number represented by this sum.

Soln:

9 =	1001	
– 5 =	(+) 0100	(nine's complement of 5)
	1101	(Invalid)
Add	0110	
	10011	
	→ 1	Add End Around Carry (EAC)
	0100	= + 4

Case 2: In the sum of minuend and nine's complement of subtrahend if a carry is produced from MSB bit, add decimal 6 (Binary 0110) and end around carry to this sum. The result is positive number represented by this sum.

Soln:



Case 3: If the sum of minuend and nine's complement of subtrahend is a valid BCD code. The result is negative and is in nine's complement form.

Soln:

$$\begin{array}{rrl} 4 = & 0100 & \\ -\ 8 = & (+)\ 0001 & \text{(nine's complement of 8)} \\ \hline & 0101 & \end{array}$$

Nine's complement of 0101 = 4

Therefore, the answer is –4.