CHAPTER 6. COMBINATIONAL CIRCUITS

- **Adder**
  - **Half Adder**
  - **Full Adder**
  - **Parallel Adder**
- **Comparator**
- **Decoder**
- **Encoder**

Half Adder

Half Adder (HA)

Binary addition, as previously mentioned, works as follows:

0+0 = 0 (sum=0, carry=0)

0+1 = 1 (sum=1, carry=0)

1+0 = 1 (sum=1, carry=0)

1+1 = 10 (sum=0, carry=1)

A half adder is a circuit designed to perform one-bit addition. It is a combinational circuit that takes two 1-bit numbers as input and provides two outputs: the sum and the carry.

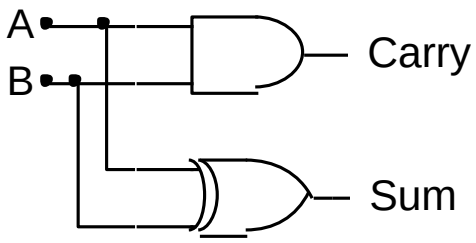
Half Adder

Truth table for a half adder is shown below.

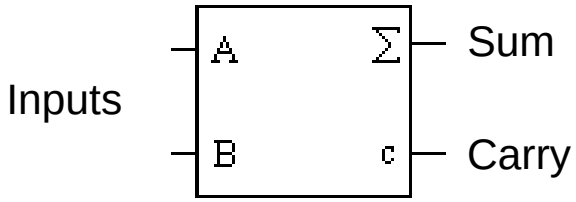
A B	Sum	Carry
0 0	0	0
0 1	1	0
1 0	1	0
1 1	0	1

$$Sum = A \oplus B$$

$$Carry = A.B$$



Logic diagram for a half adder



Block diagram for a half adder

Full Adder

Full Adder (FA)

A full adder is a combinational circuit with three inputs (two 1-bit binary numbers and a carry-in) and two outputs (sum and carry-out). Unlike a half adder, the full adder includes an additional input for the carry-in.

The truth table for a full adder is shown below.

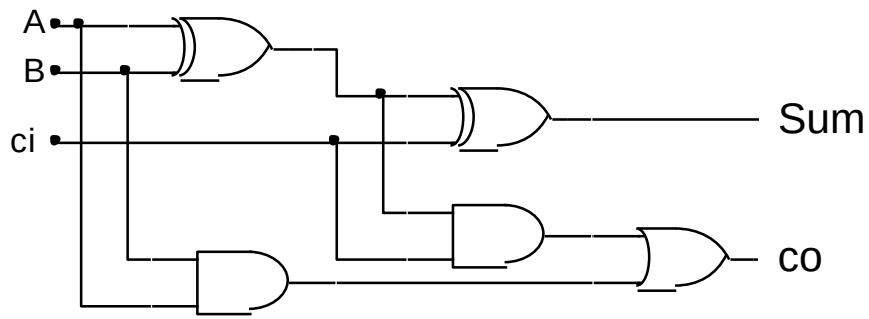
Inputs			Outputs	
A	**B**	**ci**	**Sum**	**co**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned} Sum &= A'.B'.ci + A'.B.ci' + A.B'.ci' + A.B.ci \\ &= ci.(A'.B' + A.B) + ci'.(A'.B + A.B') \\ &= ci.(A \oplus B)' + ci'.(A \oplus B) \\ &= A \oplus B \oplus ci \end{aligned}$$

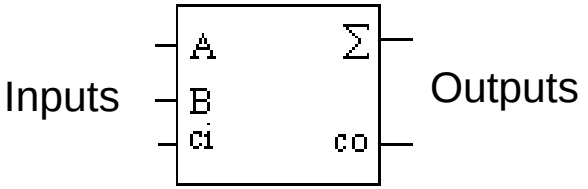
$$\begin{aligned} co &= A'.B.ci + A.B'.ci + A.B.ci' + A.B.ci \\ &= ci.(A \oplus B) + A.B \end{aligned}$$

Full Adder

$$Sum = A \oplus B \oplus ci \qquad co = ci.(A \oplus B) + A.B$$

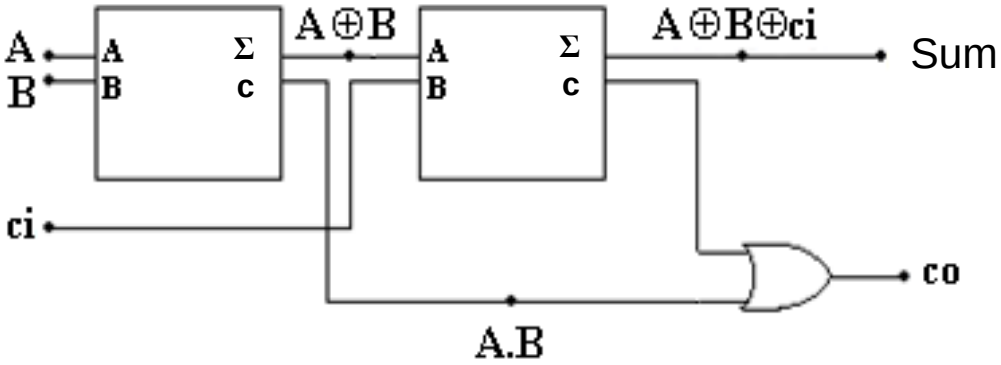


Logic diagram for a full adder



Block diagram for a full adder

A full adder can be constructed using two half adders and an OR gate, as shown below:



In a half adder

Carry = A.B

Sum = A ⊕ B

Parallel Adder

Parallel Binary Adder

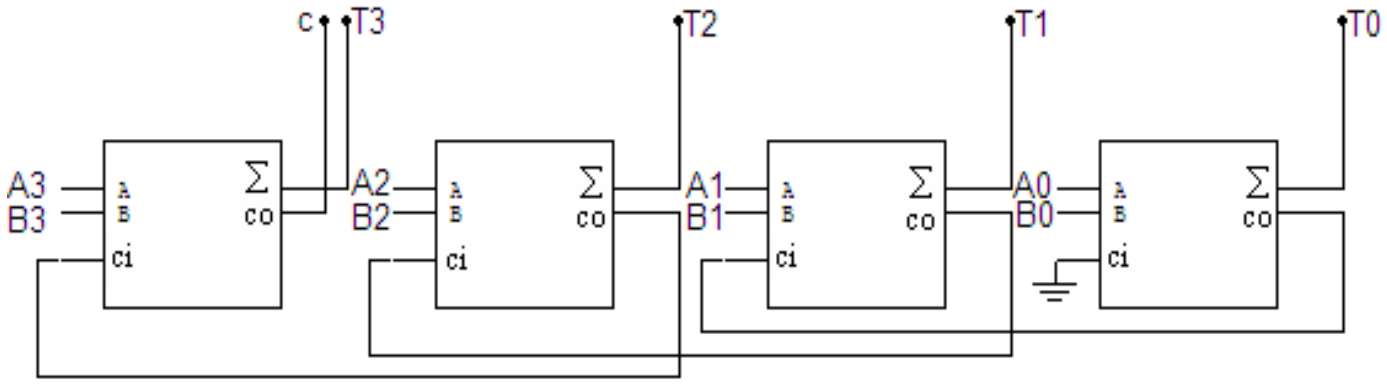
We use half adders and full adders to add 1-bit binary numbers. To add binary numbers with more than one bit, **parallel adders** are used. A parallel adder is implemented by cascading multiple full adders.

For instance, a **4-bit parallel adder** is created by connecting four full adders in sequence:

- The **carry-out** of each full adder is connected to the **carry-in** of the next full adder.
- The **carry-in** of the first full adder (handling the least significant bits, LSBs) is set to 0.
- The **carry-out** of the last full adder (handling the most significant bits, MSBs) becomes the final carry output of the parallel adder.

This arrangement allows efficient addition of multi-bit binary numbers.

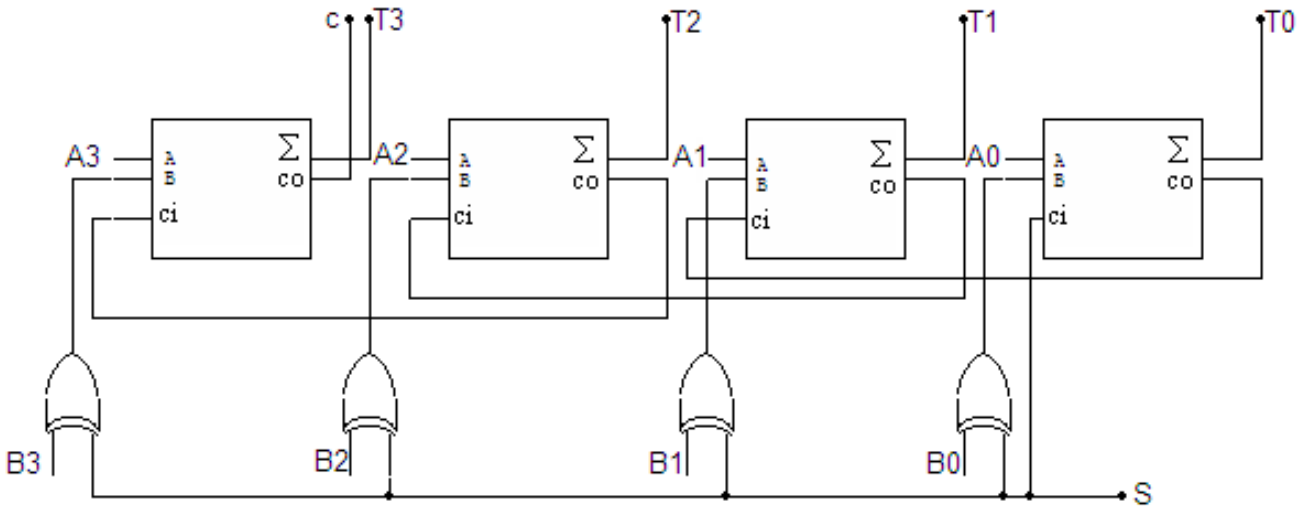
	A3	**A2**	**A1**	**A0**
+	**B3**	**B2**	**B1**	**B0**
c	**T3**	**T2**	**T1**	**T0**



Parallel Adder

Binary subtraction can be performed using two's complement. The circuit shown below handles both addition and subtraction. To subtract a binary number, we first convert the number $(B3B2B1B0)_2$ to its two's complement by inverting all its bits and adding 1 to the result. The inversion of the bits is achieved using XOR gates, and the addition of 1 is done by providing a carry-in of 1 to the first full adder. The circuit then uses this two's complement to perform subtraction alongside addition, enabling it to handle both operations.

	A3	**A2**	**A1**	**A0**
±	**B3**	**B2**	**B1**	**B0**
c	**T3**	**T2**	**T1**	**T0**

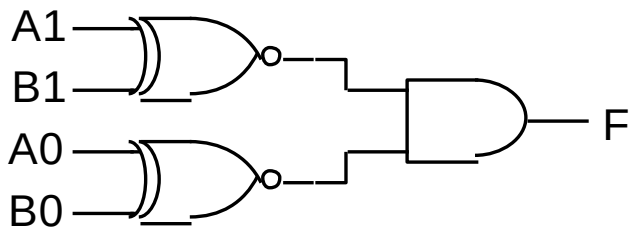


When S = 0, the circuit performs addition.

When S = 1, the circuit performs subtraction.

Comparator

Comparators are used to compare the magnitude of two binary numbers. To detect if two numbers are equal, we use XOR or XNOR gates. For example, to compare two 2-bit numbers, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, the circuit shown below can be used.



This circuit outputs 1 when the two numbers are equal and outputs 0 when they are different. The same concept can be extended to compare N-bit numbers by expanding the circuit accordingly.

Comparator

For example, if we want to compare the magnitudes of two 4-bit numbers ($A_3A_2A_1A_0$ and $B_3B_2B_1B_0$), we can write the following expressions:

- If A_3=1 and B_3=0, then A>B,
- If A_3=0 and B_3=1, then A<B,
- If A_3=B_3, we need to check the less significant bits.

The following equalities can be written:

$$\text{Output (A>B)} = A_3.B_3' + (A_3 \otimes B_3).A_2.B_2' + (A_3 \otimes B_3).(A_2 \otimes B_2).A_1.B_1' + (A_3 \otimes B_3).(A_2 \otimes B_2).(A_1 \otimes B_1).A_0.B_0'$$

$$\text{Output (A<B)} = A_3'.B_3 + (A_3 \otimes B_3).A_2'.B_2 + (A_3 \otimes B_3).(A_2 \otimes B_2).A_1'.B_1 + (A_3 \otimes B_3).(A_2 \otimes B_2).(A_1 \otimes B_1)A_0'.B_0$$

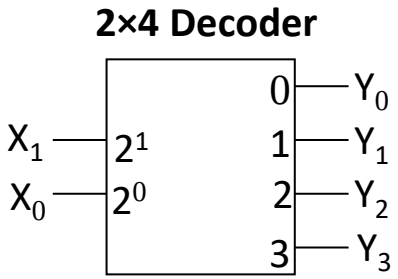
$$\text{Output (A=B)} = (A_3 \otimes B_3).(A_2 \otimes B_2).(A_1 \otimes B_1).(A_0 \otimes B_0)$$

Decoder

Decoders are combinational circuits with n inputs and at most 2^n outputs. The input is a binary number, and the active output corresponds to its decimal value.

The truth table of a 2×4 decoder is shown below.

X_1 X_0	Y_0	Y_1	Y_2	Y_3
0 0	1	0	0	0
0 1	0	1	0	0
1 0	0	0	1	0
1 1	0	0	0	1



Block diagram of a 2×4 decoder

$Y_0 = X_1'.X_0'$

$Y_1 = X_1'.X_0$

$Y_2 = X_1.X_0'$

$Y_3 = X_1.X_0$

Decoder

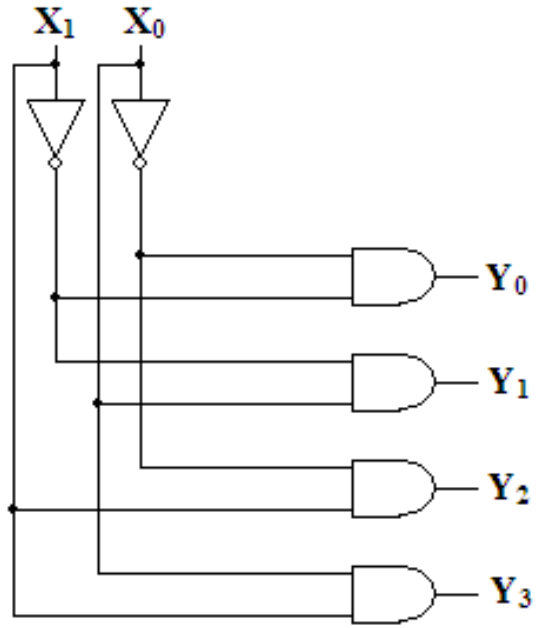
In decoders, an enable (EN) input can also be used. When EN = 1, the decoder operates normally, but when EN = 0, the outputs are inactive (all outputs are 0).

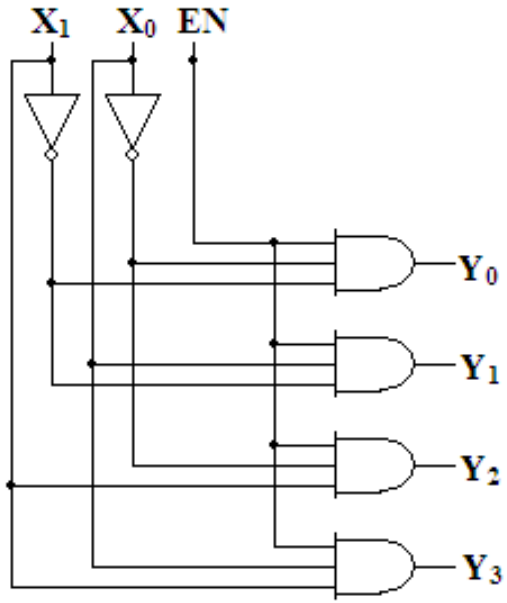
$Y_0 = X_1'.X_0'$

$Y_1 = X_1'.X_0$

$Y_2 = X_1.X_0'$

$Y_3 = X_1.X_0$



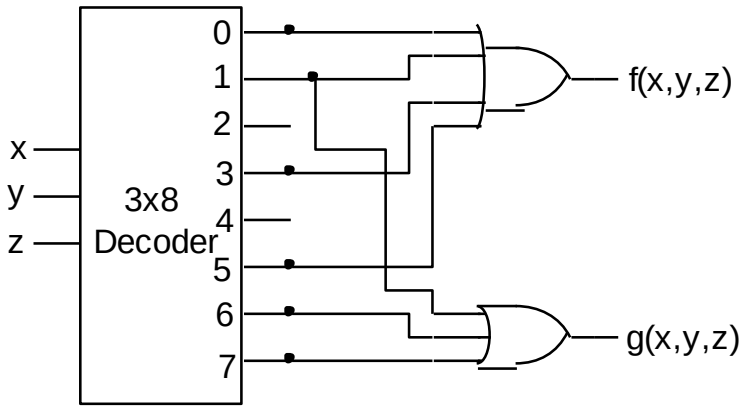


Decoder

Decoders can be used to implement logical functions. Each output of a decoder represents a minterm.

In general, a logical expression with n inputs and m outputs can be implemented using an $n \times 2^n$ decoder and m OR gates.

Example: Let's implement $f(x,y,z) = \Sigma(0,1,3,5)$ and $g(x,y,z) = \Sigma(1,6,7)$ with a decoder.



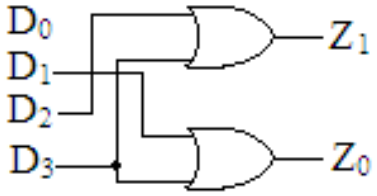
Encoder

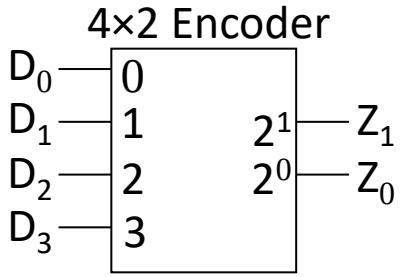
Encoders are combinational circuits that perform the reverse operation of decoders. Generally, they have n outputs and at most 2^n inputs. Under normal conditions, only one of the inputs should be 1. In this case, the circuit produces a binary code at the output, indicating which input is 1.

The truth table for a 4-input, 2-output encoder is as follows:

D_0	D_1	D_2	D_3	Z_1 Z_0
1	0	0	0	**0 0**
0	1	0	0	**0 1**
0	0	1	0	**1 0**
0	0	1	1	**1 1**

Since only one input is allowed to be 1 at the same time, we can treat cases where multiple inputs are 1 simultaneously as "don't care" conditions when calculating the Z_1 and Z_0 outputs using Karnaugh maps.





$Z_1 = D_2 + D_3$

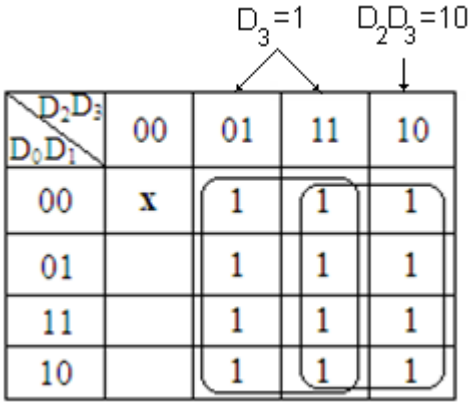
$Z_0 = D_1 + D_3$

$NI = D_0'.D_1'.D_2'.D_3'$ (No Input)

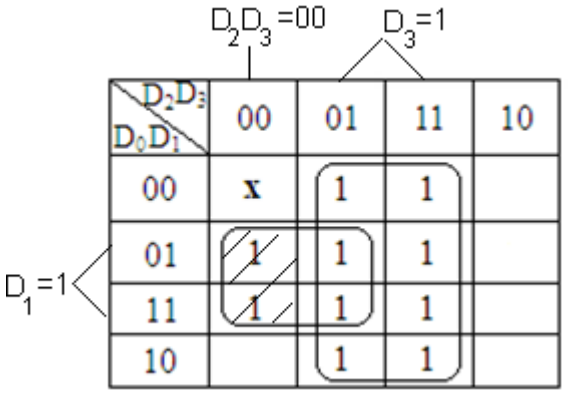
Encoder

In some cases, it may not be possible for only one input of the encoder to be 1 at a time. To handle such situations, priority encoders have been developed. An example implementation for 4 inputs and 3 outputs is shown below, where the priority order is from D_3 to D_0.

D_0	D_1	D_2	D_3	Z_1	Z_0	NI
0	0	0	0	x	x	1
1	0	0	0	0	0	0
x	1	0	0	0	1	0
x	x	1	0	1	0	0
x	x	x	1	1	1	0



$Z_1 = D_2 + D_3$



$Z_0 = D_3 + D_2'.D_1$

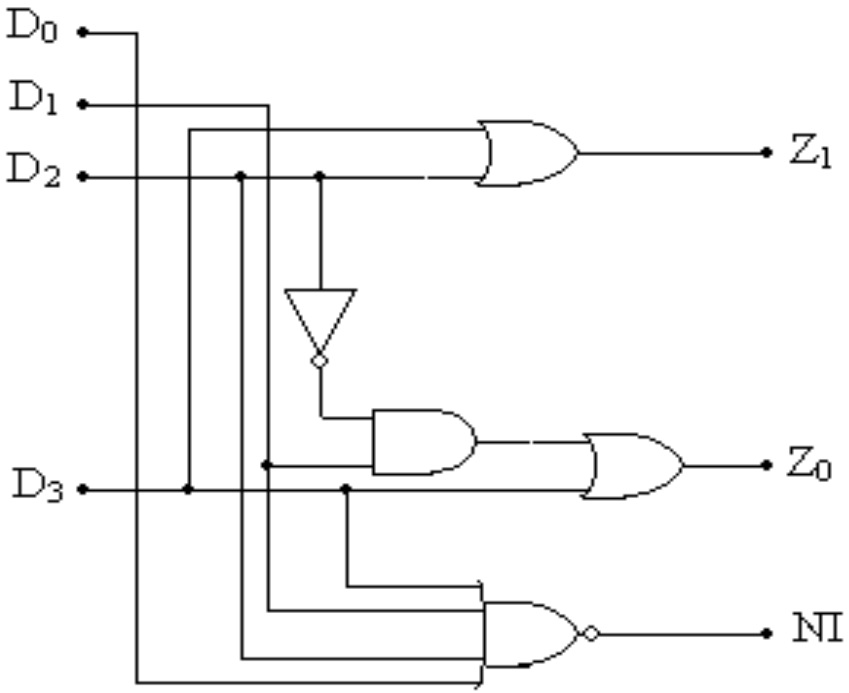
$NI = D_0'.D_1'.D_2'.D_3'$

Encoder

$Z_1 = D_2 + D_3$

$Z_0 = D_3 + D_2'.D_1$

$NI = D_0'.D_1'.D_2'.D_3' = (D_0+D_1+D_2+D_3)'$



Logic diagram for a 4×2 priority encoder