Chapter 4

Combinational Logic

Outline

- $\boxed{\text{Combinational Circuits}}$
- Analysis and Design Procedures
- Binary Adders
- Other Arithmetic Circuits
- Decoders and Encoders
- Multiplexers

Combinational v.s Sequential Circuits

- Logic circuits may be ***combinational*** or ***sequential***
- Combinational circuits:
  - Consist of ***logic gates*** only
  - Outputs are determined from the present values of inputs
  - Operations can be specified by a set of Boolean functions
- Sequential circuits:
  - Consist of ***logic gates*** and ***storage elements***
  - Outputs are a function of the inputs and the state of the storage elements
    - Depend not only on present inputs, but also on past values
  - Circuit behavior must be specified by a time sequence of inputs and internal states

Combinational Circuit (1/2)

- A combinational circuit consists of
  - Input variables
  - Logic gates
  - Output variables

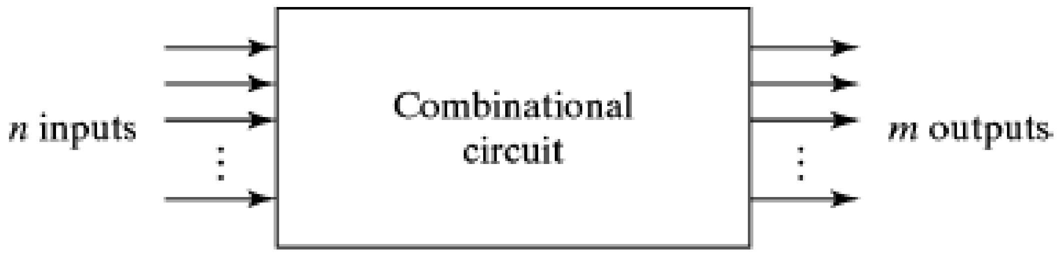


Fig. 4-1 Block Diagram of Combinational Circuit

Combinational Circuit (2/2)

- Each input and output variable is a binary signal
  - Represent logic 1 and logic 0
- There are 2^n possible binary input combinations for n input variable
- Only one possible output value for each possible input combination
- Can be specified with a truth table
- Can also be described by m Boolean functions, one for each output variable
  - Each output function is expressed in terms of n input variables

Outline

- Combinational Circuits
- $\boxed{\text{Analysis and Design Procedures}}$
- Binary Adders
- Other Arithmetic Circuits
- Decoders and Encoders
- Multiplexers

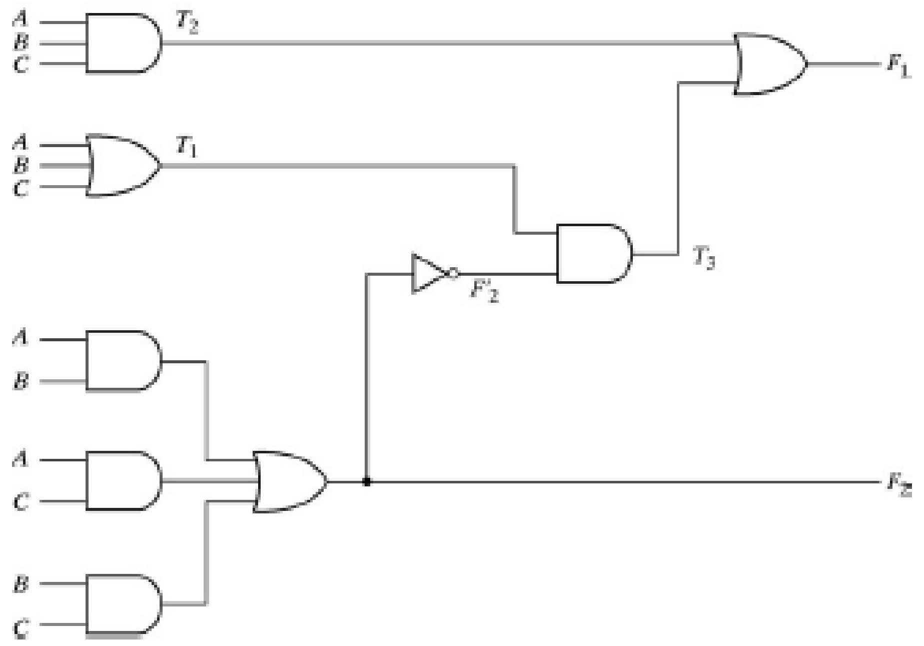
Analysis Procedure

- Analysis: determine the function that the circuit implements
  - Often start with a given logic diagram
- The analysis can be performed by
  - Manually finding Boolean functions
  - Manually finding truth table
  - Using a computer simulation program
- First step: make sure that circuit is combinational
  - Without feedback paths or memory elements
- Second step: obtain the output Boolean functions or the truth table

Output Boolean Functions (1/3)

Step 1:

- Label all gate outputs that are a function of input variables
- Determine Boolean functions for each gate output

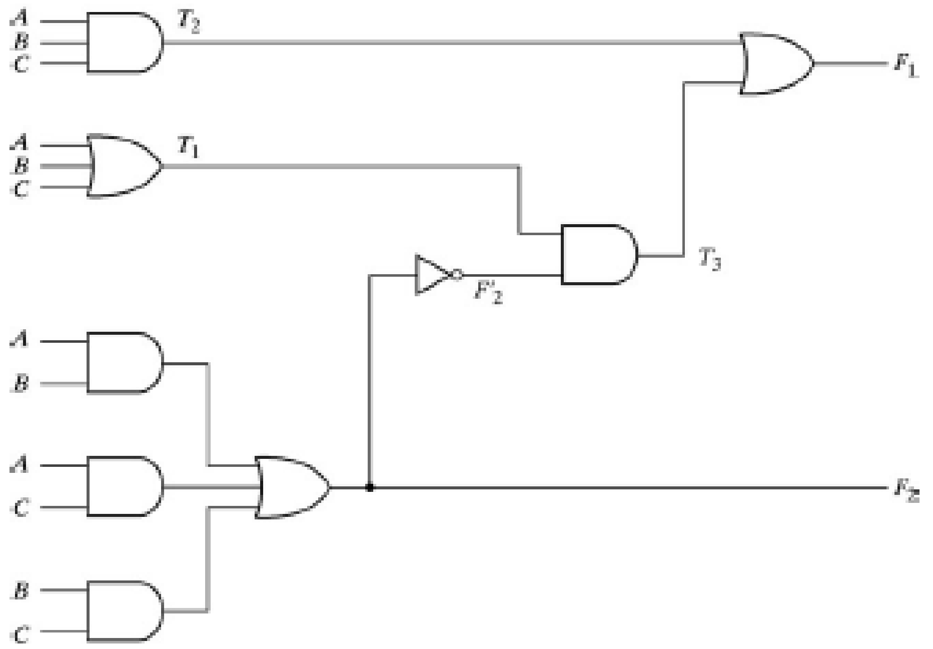


$F_2 = AB + AC + BC$
$T_1 = A + B + C$
$T_2 = ABC$

Output Boolean Functions (2/3)

Step 2:

- Label the gates that are a function of input variables and previously labeled gates
- Find the Boolean function for these gates



$T_3 = F'_2\, T_1$
$F_1 = T_3 + T_2$

Output Boolean Functions (3/3)

Step 3:

- Obtain the output Boolean function in term of input variables
  - By repeated substitution of previously defined functions

$$
\begin{aligned}
F_1 &= T_3 + T_2 = F'_2 T_1 + ABC \\
&= (AB + AC + BC)'(A + B + C) + ABC \\
&= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\
&= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\
&= A'BC' + A'B'C + AB'C' + ABC
\end{aligned}
$$

Truth Table

- To obtain the truth table from the logic diagram:
  1. Determine the number of input variables

     For n inputs:
     - *2^n* possible combinations
     - List the binary numbers from 0 to *2^n-1* in a table
  2. Label the outputs of selected gates
  3. Obtain the truth table for the outputs of those gates that are a function of the input variables only
  4. Obtain the truth table for those gates that are a function of previously defined variables at step 3
     - Repeatedly until all outputs are determined

Truth Table for Fig. 4-2

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design Procedure

- Design procedure:
  - Input: the specification of the problem
  - Output: the logic circuit diagram (or Boolean functions)
- Step 1: determine the required number of inputs and outputs from the specification
- Step 2: derive the truth table that defines the required relationship between inputs and outputs
- Step 3: obtain the simplified Boolean function for each output as a function of the input variables
- Step 4: draw the logic diagram and verify the correctness of the design

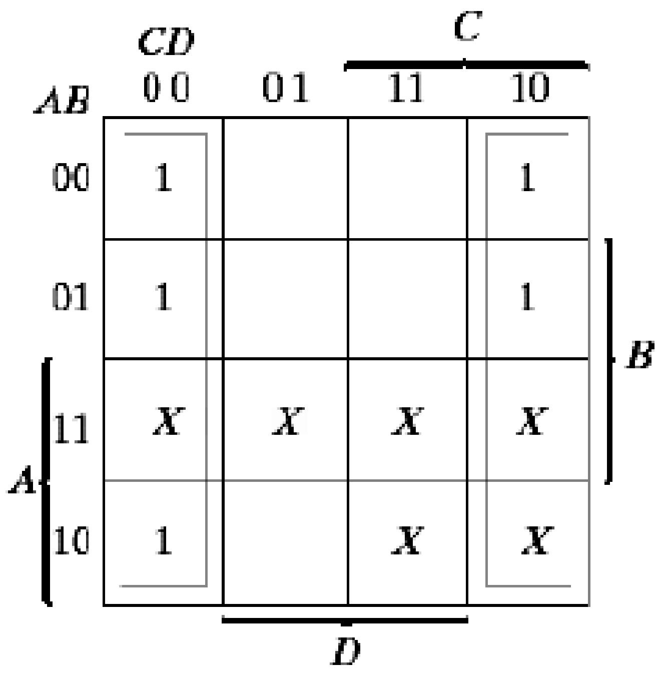
Code Conversion Example

- Convert from BCD code to Excess-3 code
- The 6 input combinations not listed are don't cares
- These values have no meaning in BCD
- We can arbitrary assign them to 1 or 0

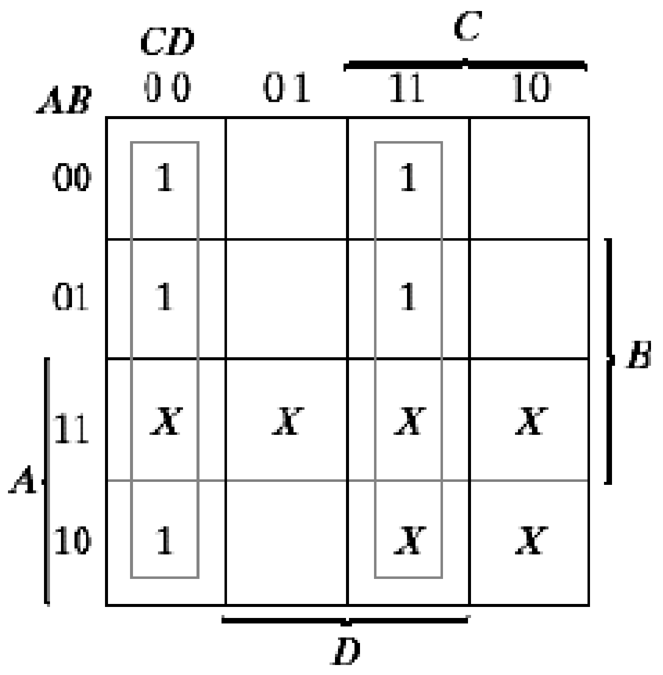
Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Maps for Code Converter (1/2)

- The six don't care minterms (10~15) are marked with X

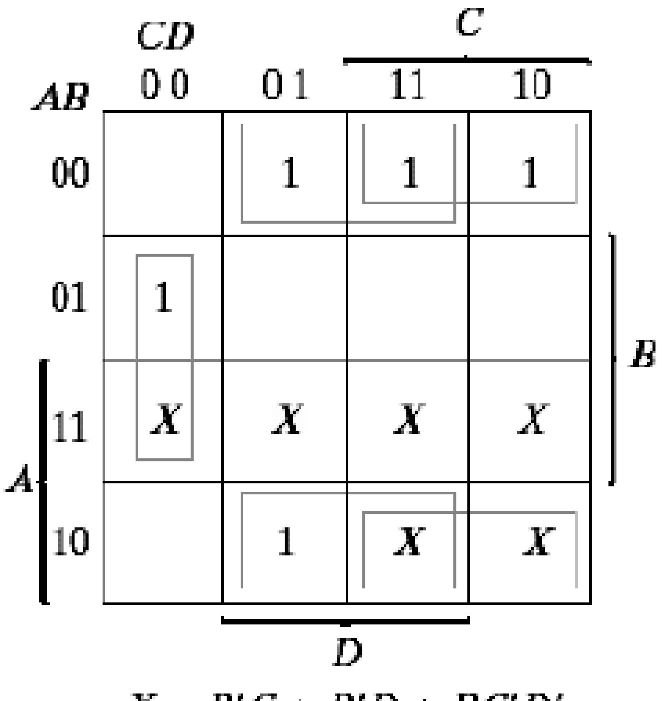


$z = D'$



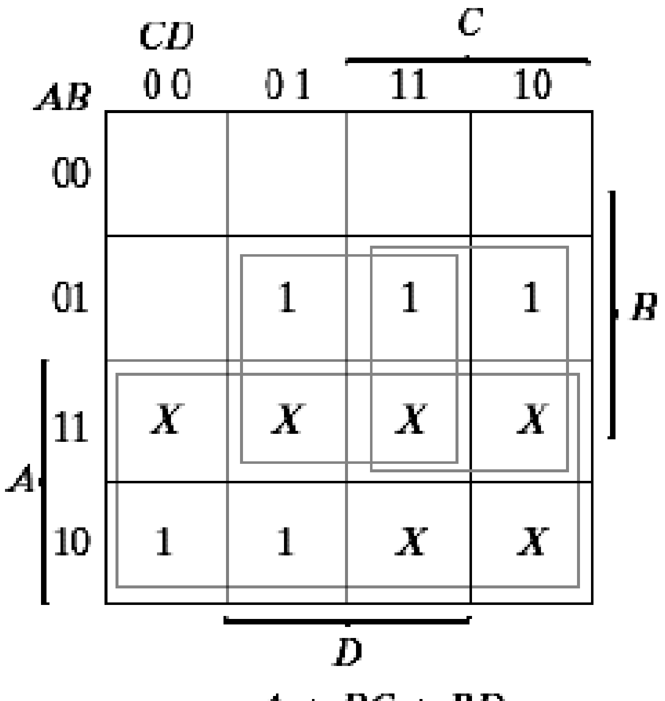
$y = CD + C'D'$

Maps for Code Converter (2/2)



AB \ CD	00	01	11	10
00		1	1	1
01	1			
11	X	X	X	X
10		1	X	X

$X = B'C + B'D + BC'D'$



AB \ CD	00	01	11	10
00				
01		1	1	1
11	X	X	X	X
10	1	1	X	X

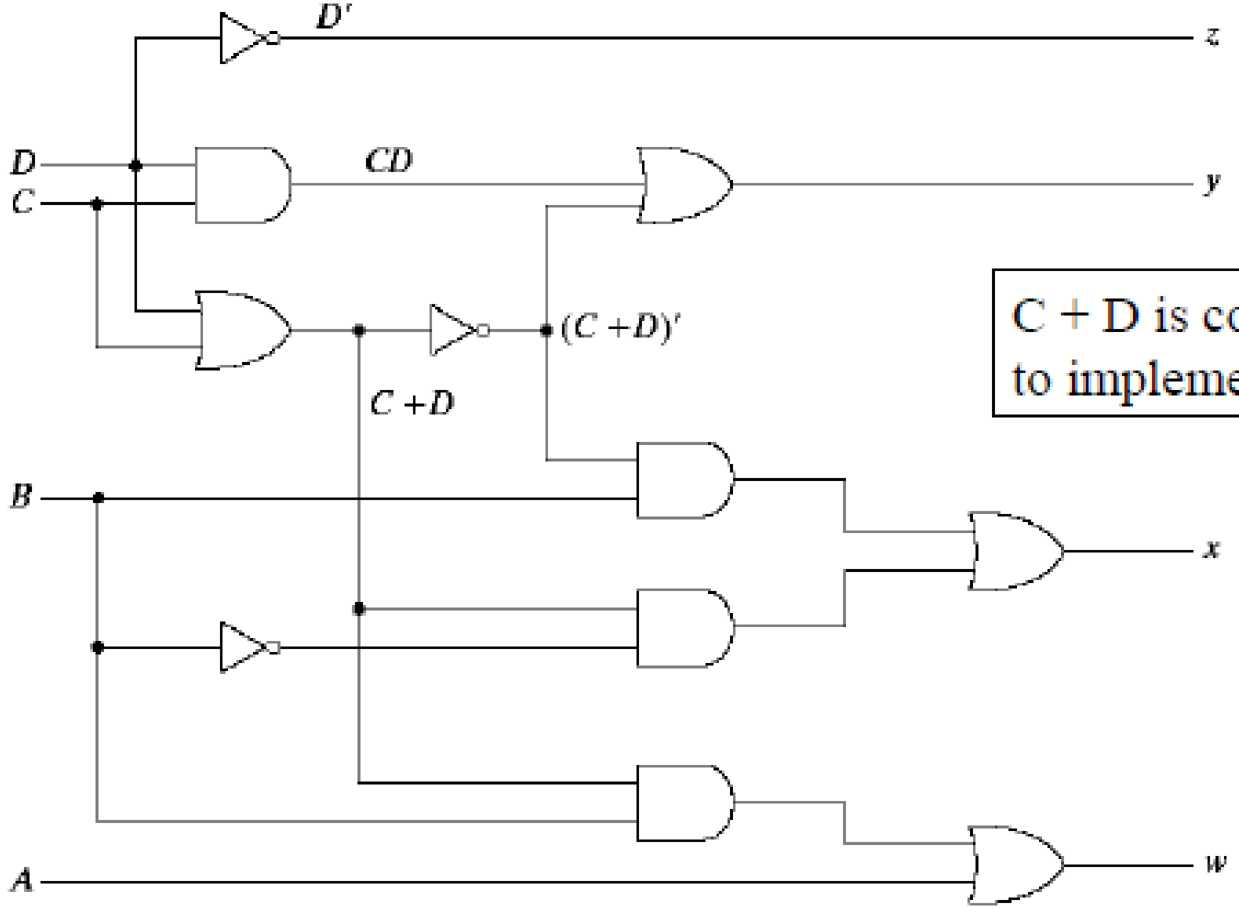
$w = A + BC + BD$

Logic Diagram for the Converter

- There are various possibilities for a logic diagram that implements a circuit
- A two-level logic diagram may be obtained directly from the Boolean expressions derived by the maps
- The expressions may be manipulated algebraically to use common gates for two or more outputs
  - Reduce the number of gates used

$$
\begin{aligned}
z &= D' \\
y &= CD + C'D' = CD + (C + D)' \\
x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\
&= B'(C + D) + B(C + D)' \\
w &= A + BC + BD = A + B(C + D)
\end{aligned}
$$

Logic Diagram for the Converter



C + D is commonly used
to implement the three outputs

Outline

- Combinational Circuits
- Analysis and Design Procedures
- $\boxed{\text{Binary Adders}}$
- Other Arithmetic Circuits
- Decoders and Encoders
- Multiplexers

Adder

- The most basic arithmetic operation is the addition of two binary digits
  - When both augend and addend bits are equal to 1, the binary sum consists of two digits (1 + 1 = 10)
  - The higher significant bit of this result is called a ***carry***
- A combination circuit that performs the addition of two bits is ***half adder***
- A adder performs the addition of 2 significant bits and a previous carry is called a ***full adder***

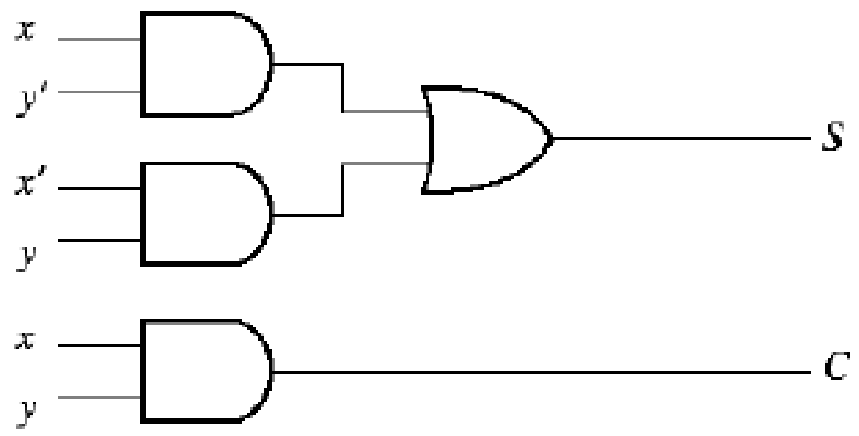
Half Adder

- Half adder
  - Inputs: x and y
  - Outputs: S (for sum) and C (for carry)

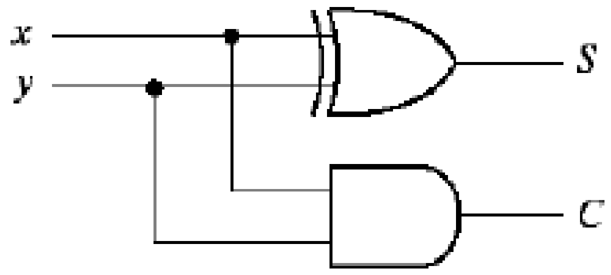
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = x'y + xy'$$
$$C = xy$$

Implementation of a Half Adder



(a) $S = xy' + x'y$
$C = xy$

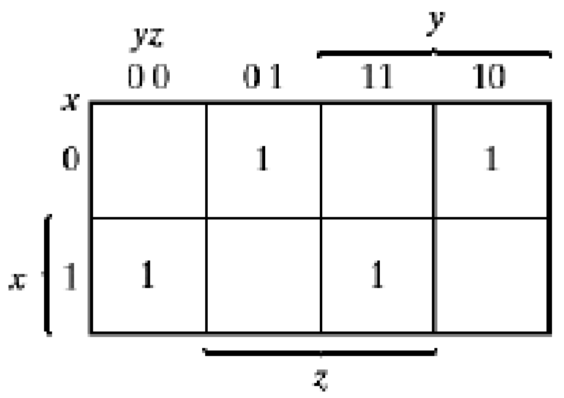


(b) $S = x \oplus y$
$C = xy$

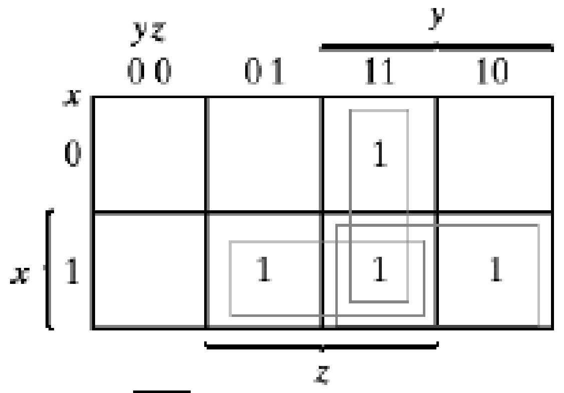
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

x, y: the two significant bits to be added
z: the carry from the previous position

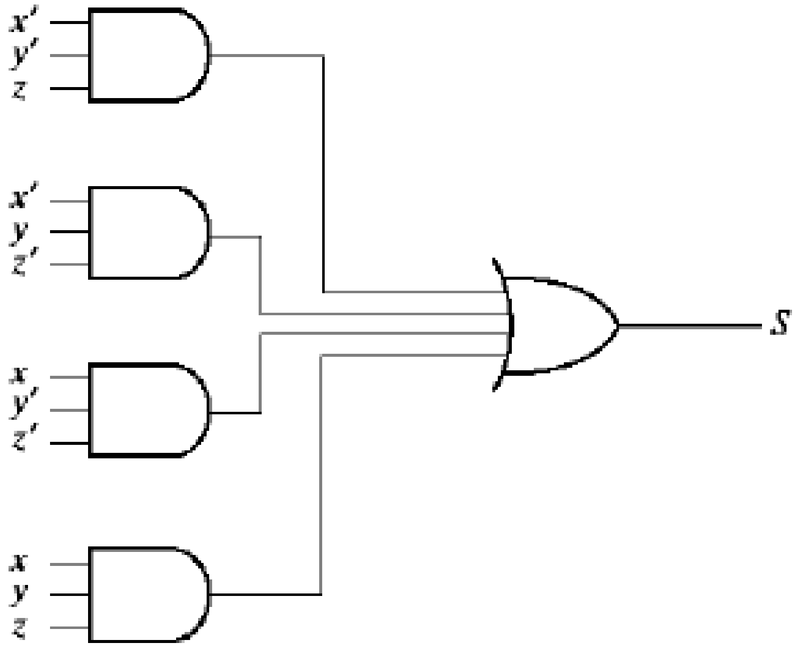


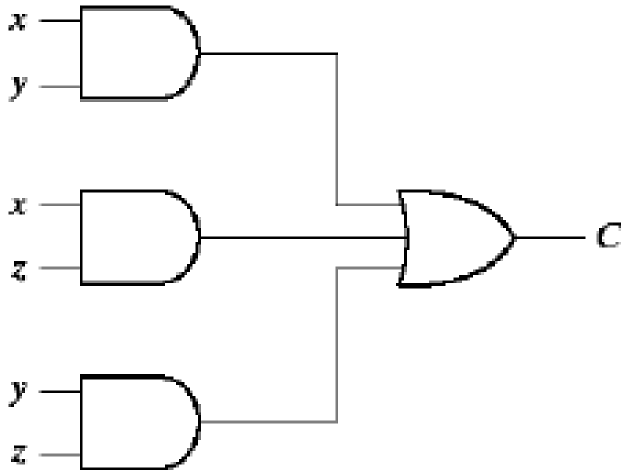
$$S = x'y'z + x'yz' + xy'z' + xyz$$



$$\boxed{C} = xy + xz + yz$$
$$= xy + xy'z + x'yz$$

Implementation of a Full Adder



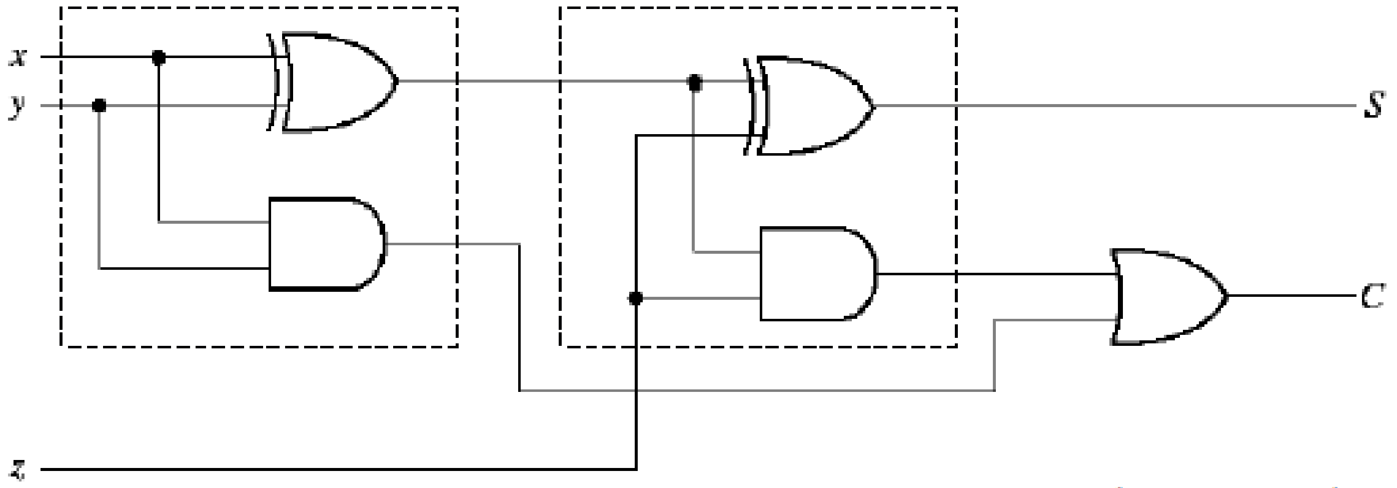


$C = xy + xz + yz$

$S = x'y'z + x'yz' + xy'z' + xyz$

Implementation of a Full Adder

- A full adder can be implemented with two half adders and an OR gate

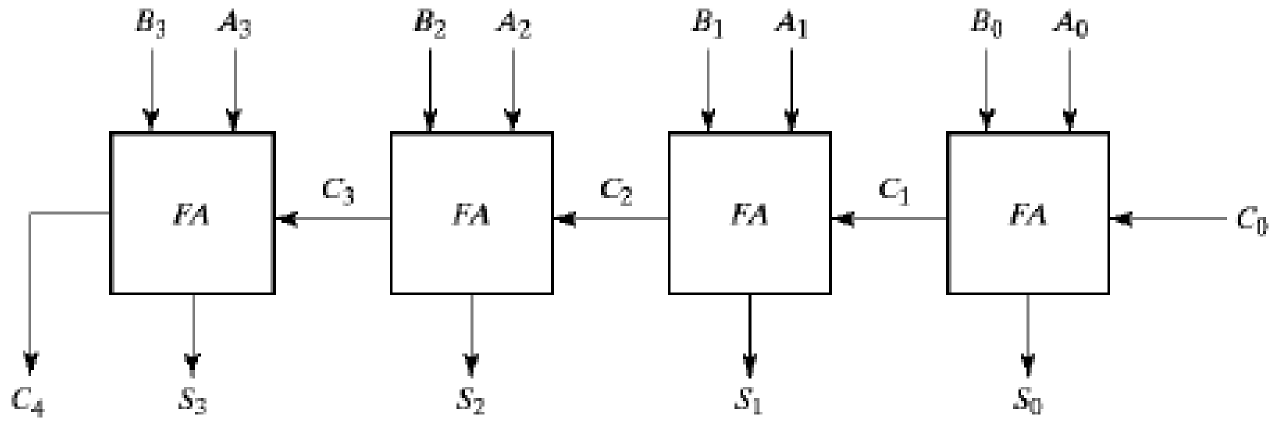


$$
\begin{aligned}
S &= z \oplus (x \oplus y) \\
&= z'(xy' + x'y) + z(xy' + x'y)' \\
&= z'(xy' + x'y) + z(xy + x'y') \\
&= xy'z' + x'yz' + xyz + x'y'z
\end{aligned}
$$

$$
\begin{aligned}
C &= z(xy' + x'y) + xy \\
&= xy'z + x'yz + xy \\
&= xy'z + x'yz + xyz + xyz' \\
&= xz + yz + xy
\end{aligned}
$$

Binary Adder

- A binary adder produces the arithmetic sum of two binary numbers
- Can be constructed with full adders connected in cascade
  - The output carry from each full adder is connected to input carry of the next full adder in the chain
  - n-bit binary ***ripple carry adder*** is connected by n FAs



4-bit Adder Example

- Consider two binary number A = 1011 and B = 0011

Subscript i :	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

RTL for full adder

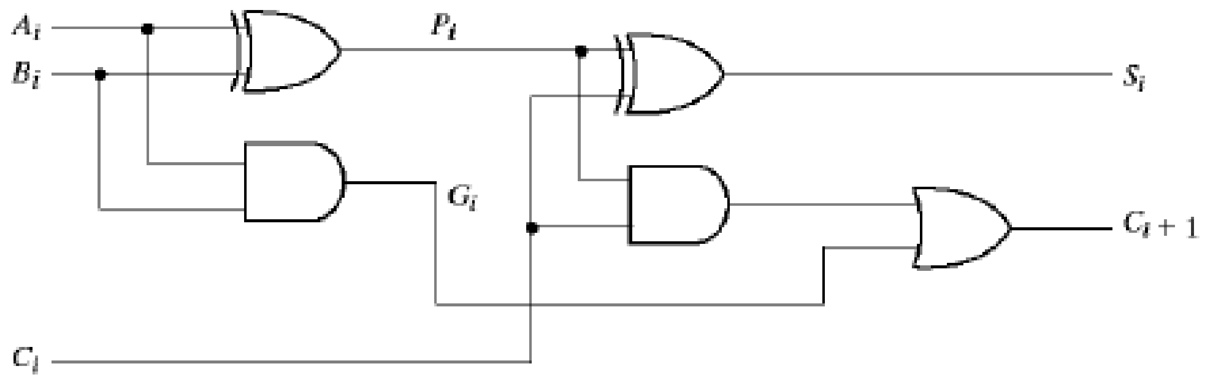
```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity andGate is
   port( A, B : in std_logic;
            F : out std_logic);
end andGate;

architecture func of andGate is
begin
   F <= A and B;
end func;
--*==============================

-- Here we define the XOR gate that we need for
-- the Half Adder
library ieee;
use ieee.std_logic_1164.all;

entity xorGate is
   port(   A, B : in std_logic;
            F : out std_logic);
end xorGate;

architecture func of xorGate is
begin
   F <= A xor B;
end func;
--*==============================

-- At this point we construct the half adder
-- using the AND and XOR gates
library ieee;
use ieee.std_logic_1164.all;

entity halfAdder is
   port( A, B : in std_logic;
    sum, Cout : out std_logic);
end halfAdder;

architecture halfAdder of halfAdder is

   component andGate is -- import AND Gate
      port( A, B : in std_logic;
               F : out std_logic);
   end component;

   component xorGate is -- import XOR Gate
      port( A, B : in std_logic;
               F : out std_logic);
   end component;

begin
   G1 : xorGate port map(A, B, sum);
   G2 : andGate port map(A, B, Cout);
end halfAdder;
--*=========================*=================== END HALF ADDER

-- Now we define the OR gate that we need for the Full Adder
library ieee;
use ieee.std_logic_1164.all;

entity orGate is
   port(   A, B : in std_logic;
            F : out std_logic);
end orGate;

architecture func of orGate is
begin
   F <= A or B;
end func;
--*================================*
--*================================*

-- We are finally ready to build the Full Adder
library ieee;
use ieee.std_logic_1164.all;

entity fullAdder is
   port( A, B, Cin : in std_logic;
         sum, Cout : out std_logic);
end fullAdder;
--
architecture fullAdder of fullAdder is

   component halfAdder is --import Half Adder entity
      port( A, B  : in std_logic;
        sum, Cout : out std_logic);
   end component;

   component orGate is --import OR Gate entity
      port( A, B : in std_logic;
            F : out std_logic);
   end component;

   signal halfTohalf, halfToOr1, halfToOr2: std_logic;

begin
   G1: halfAdder port map(A, B, halfTohalf, halfToOr1);
   G2: halfAdder port map(halfTohalf, Cin, sum, halfToOr2);
   G3: orGate port map(halfToOr1, halfToOr2, Cout);
end fullAdder;
-----------------------------------------------------------END
```

Carry Propagation

- As in a combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals
- The total propagation time is equal to the propagation delay of a typical gate times the number of levels in the circuit
- The longest propagation delay in a adder is the time that carry propagate through the full adders
- Each bit of the sum output depends on the value of the input carry
  - The value of S_i will be in final value only after the input carry C_i has been propagated

Full Adder with P and G

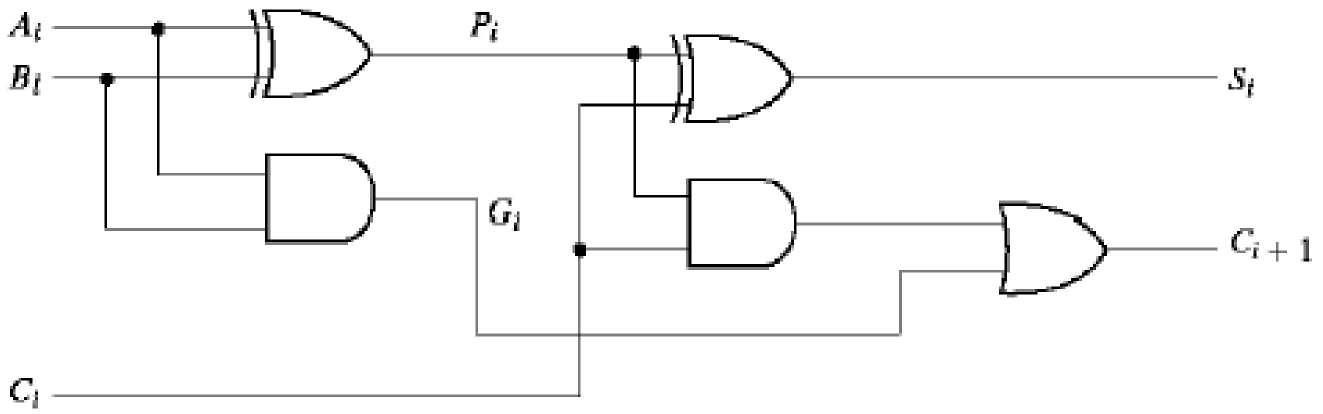
- The full adder can be redrawn with two internal signals P (propagation) and G (generation)
- The signal from input carry C_i to output carry C_{i+1} propagates through an AND and a OR gate (2 gate levels)
  - For n-bit adder, there are 2n gate levels for the carry to propagate from input to output



Carry Propagation

- The ***carry propagation time*** is a limiting factor on the speed with which two numbers are added
- All other arithmetic operations are implemented by successive additions
  - The time consumed during the addition is very critical
- To reduce the carry propagation delay
  - Employ faster gates with reduced delays
  - Increase the equipment complexity
- Several techniques for reducing the carry propagation time in a parallel adder
  - The most widely used technique employs the principle of ***carry lookahead***

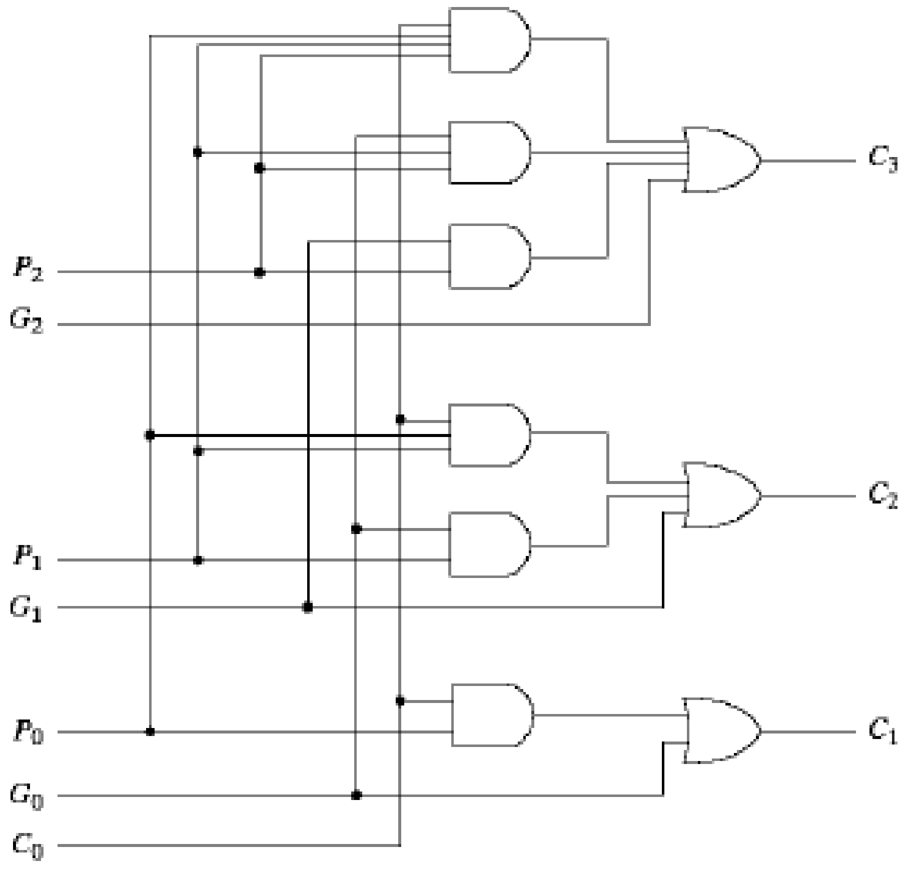
Carry Propagation & Generation



carry propagate : $P_i = A_i \oplus B_i$ $S_i = P_i \oplus C_i$

carry generate : $G_i = A_iB_i$ $C_{i+1} = G_i + P_iC_i$

Carry Lookahead Generator



C_0 = input carry

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1$

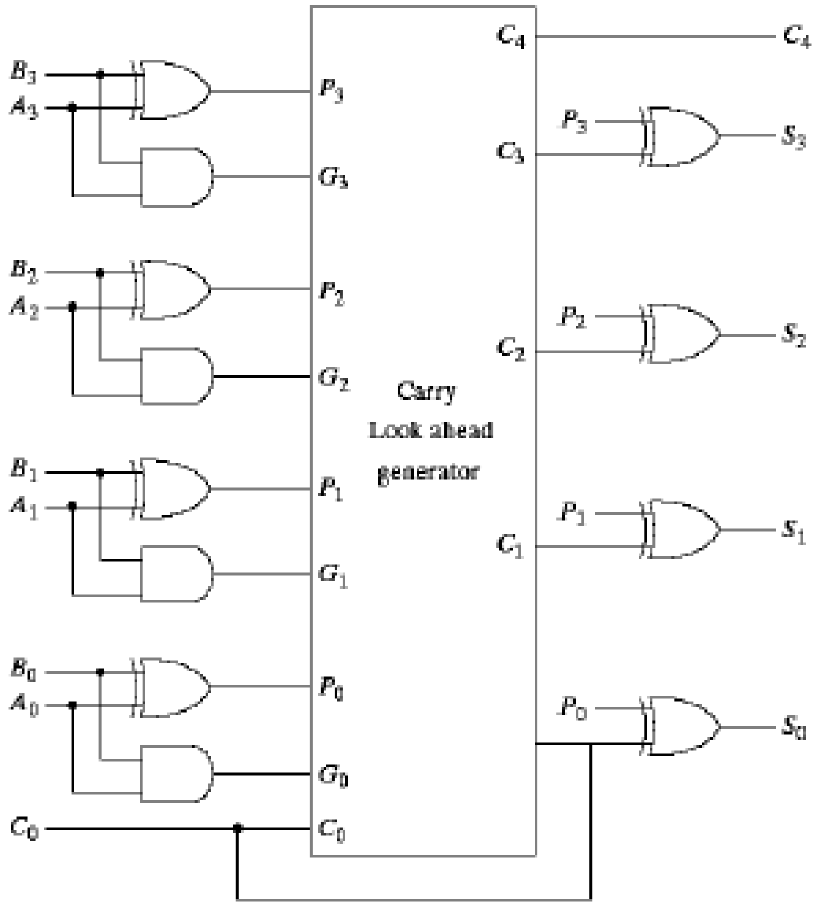
$= G_1 + P_1(G_0 + P_0C_0)$

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2$

$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

Carry Lookahead Adder



- All output carries are generated after a delay through two levels of gates
- Output S1 to S3 can have equal propagation delay times

Outline

- Combinational Circuits
- Analysis and Design Procedures
- Binary Adders
- $\boxed{\text{Other Arithmetic Circuits}}$
- Decoders and Encoders
- Multiplexers

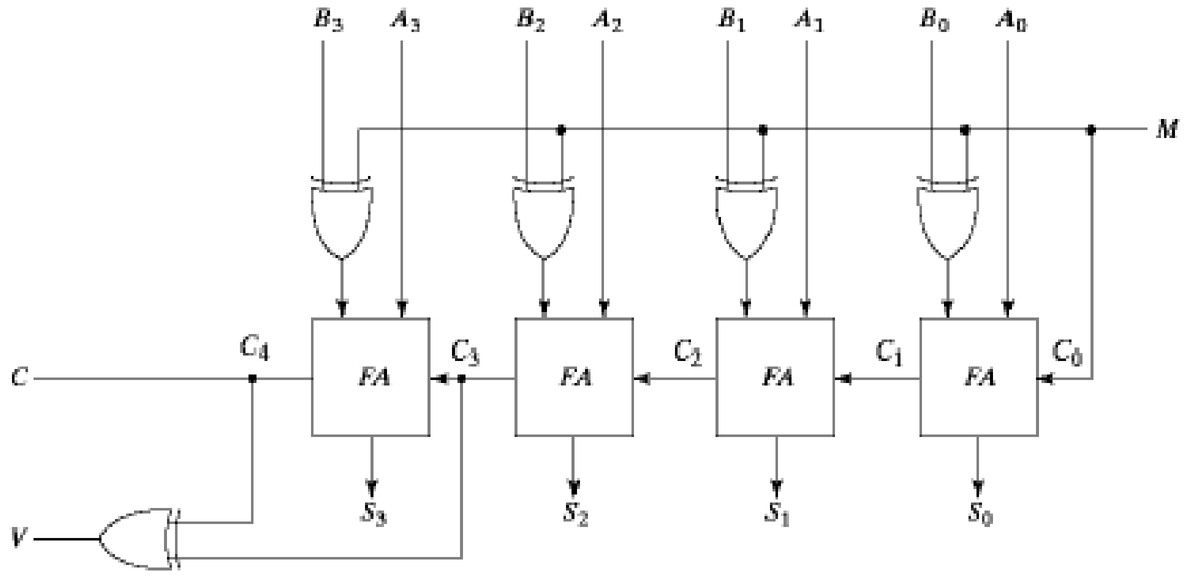
Binary Subtractor

- A – B can be done by taking the 2's complement of B and adding it to A ---> A – B = A + (-B)
  - 2'complement can be obtained by taking the 1'complement and adding on to the least significant pair of bits
  - A - B = A + (B′ + 1)
- The circuit for subtraction A – B consists of an adder with inverter placed between each data input B and the corresponding input of the full adder
- The input carry C_0 must be equal to 1

4-bit Adder-Subtractor

- M=0 (Adder)
  - Input of FA is A and B (B ⊕ 0 = B), and C_0 is 0
- M=1 (Subtractor)
  - Input of FA is A and B' (B ⊕ 1 = B'), and C_0 is 1

x	y	x⊕y	
0	0	0	y
0	1	1	
1	0	1	y'
1	1	0	



(overflow)

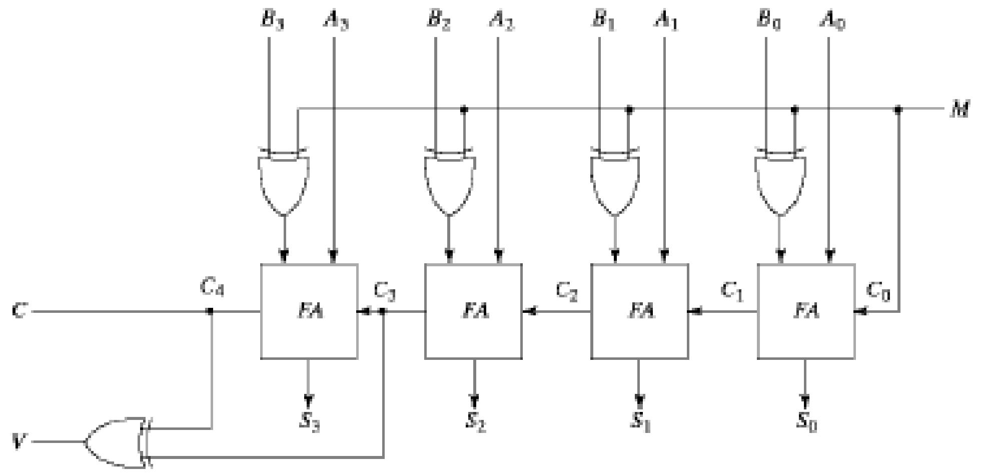
Overflow

- An overflow occurs when two number of *n* digits each are added and the sum occupies *n+1* digits
- When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position
- When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow
  - Extra overflow detection circuits are required
- An overflow can only occur when two numbers added are ***both positive*** or ***both negative***

Adder-Subtractor Circuit

- Unsigned
  - C bit detects a ***carry*** after addition or a ***borrow*** after subtraction
- Signed
  - V bit detects an overflow

    0: no overflow; 1: overflow



Decimal Adder

- A decimal adder requires a minimum of 9 inputs and 5 outputs
  - 1 digit requires 4-bit
  - Input: 2 digits + 1-bit carry
  - Output: 1 digit + 1-bit carry
- BCD adder
  - Perform the addition of two decimal digits in BCD, together with an input carry from a previous stage
  - The output sum cannot be greater than 19 (9+9+1)

Derivation of BCD Adder

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	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

BCD Adder

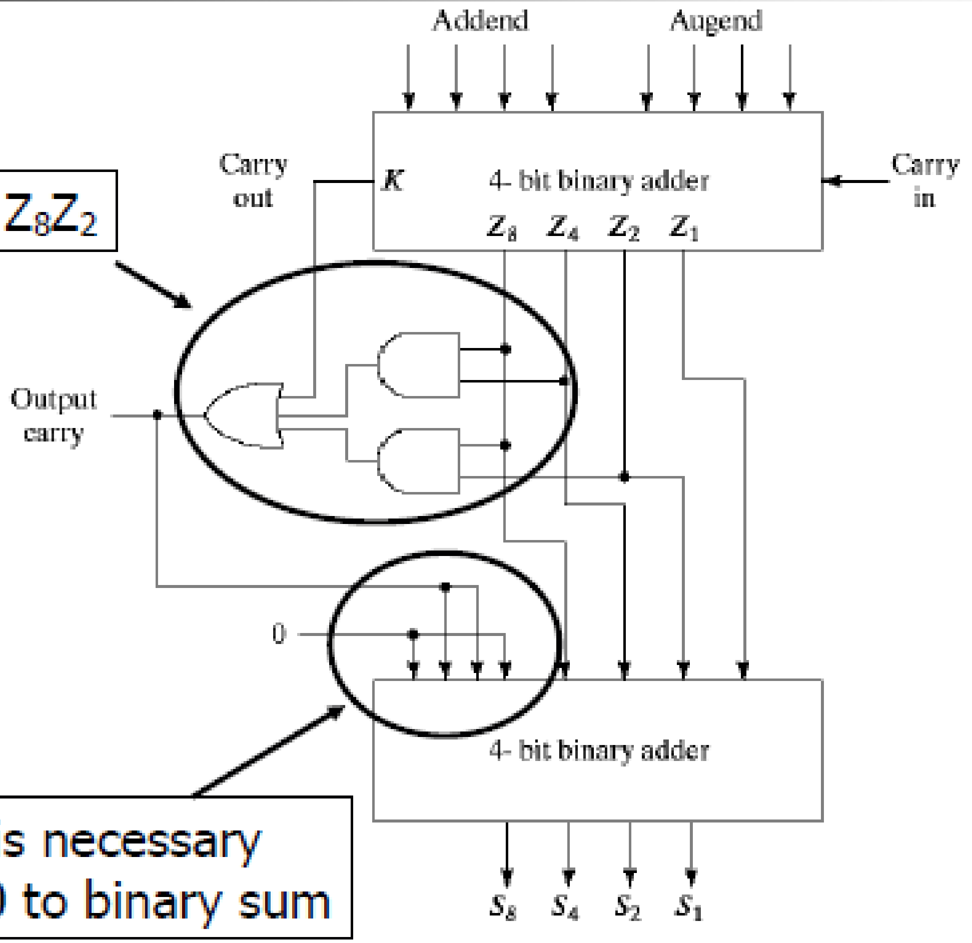
- When the binary sum is equal to or less than 1001_b
  - BCD Sum = Binary Sum
  - C = 0
- When the binary sum is greater than 1001_b
  - BCD Sum = Binary Sum + 0110_b
  - C = 1

Z_2Z_1 \ Z_8Z_4	00	01	11	10
00			1	
01			1	
11			1	1
10			1	1

$$C = K + Z_8Z_4 + Z_8Z_2$$

Block Diagram of a BCD Adder

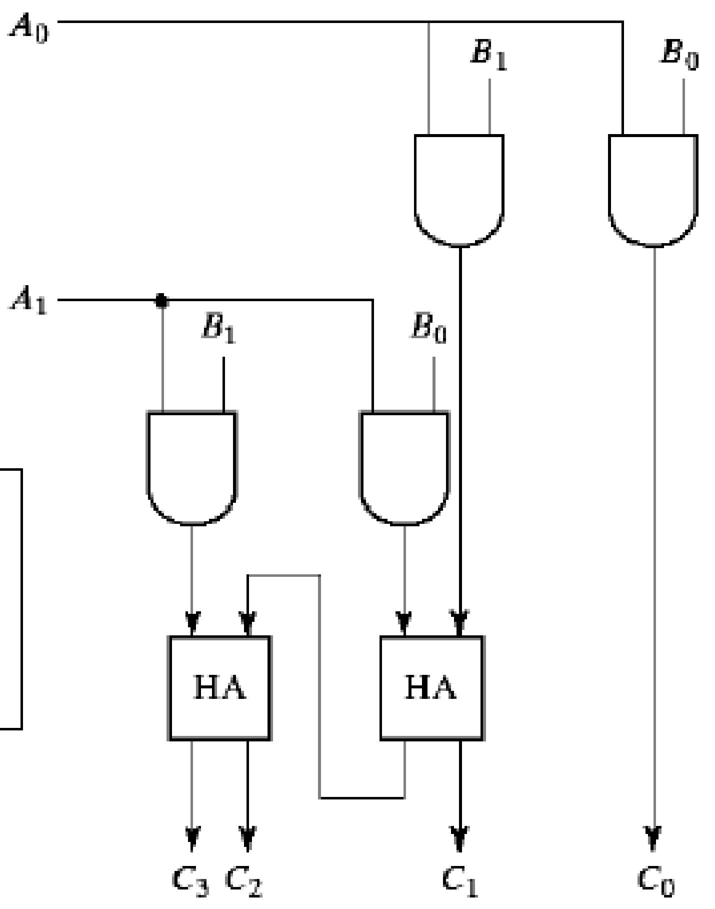
$C = K + Z_8Z_4 + Z_8Z_2$



If C =1 , it is necessary to add 0110 to binary sum

Binary Multiplier

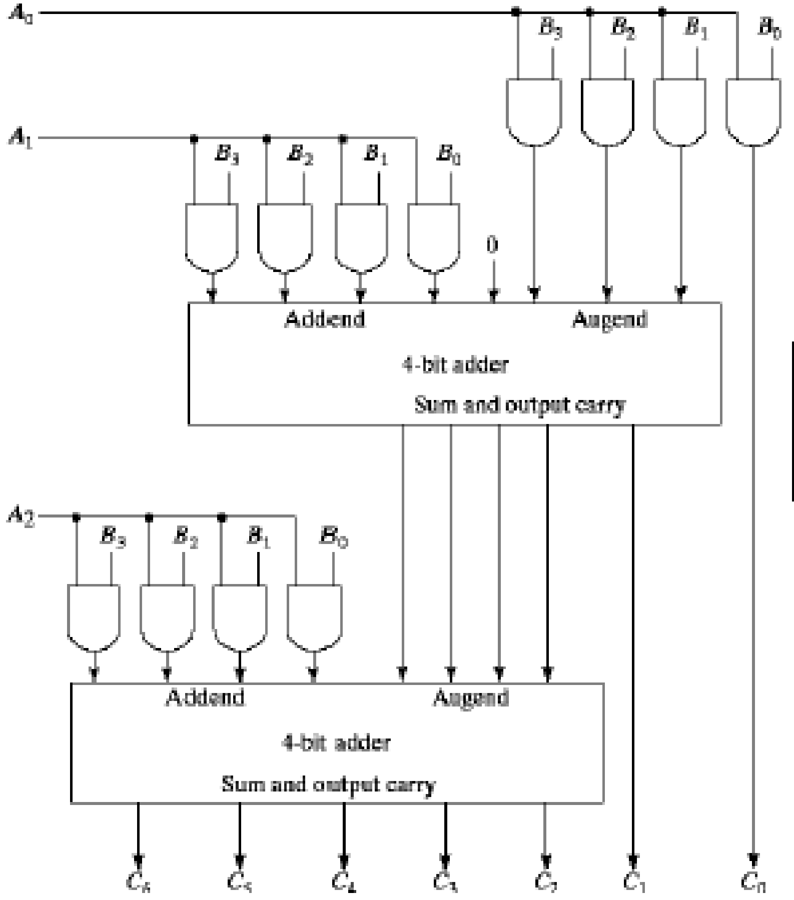
$$\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\cline{3-4}
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}$$

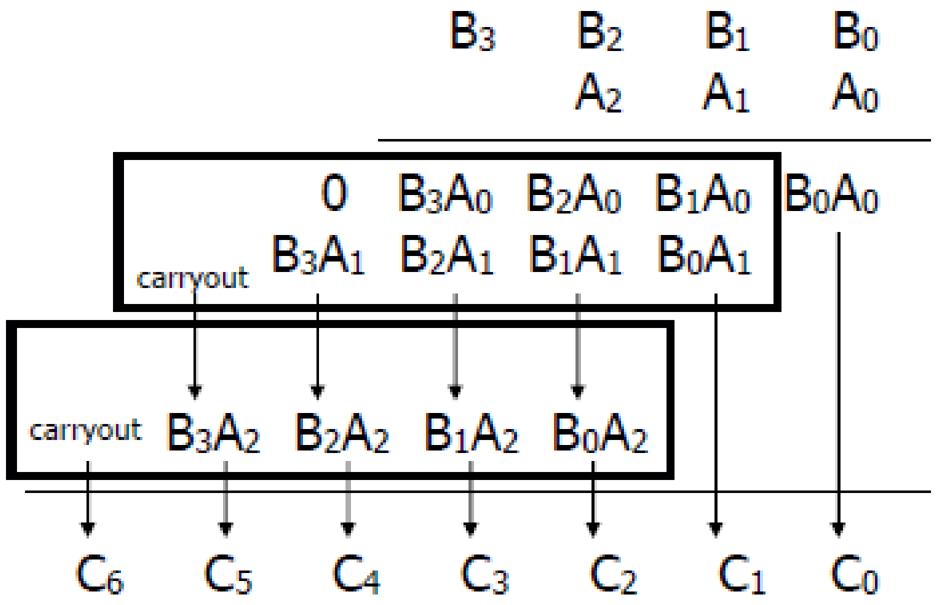


For a J*K bits multiplier , we need:
- (J * K) AND gates
- (J – 1) K-bit adders

to produce a product of J+K bits

4-Bit By 3-Bit Binary Multiplier





B_3 B_2 B_1 B_0
A_2 A_1 A_0
0 B_3A_0 B_2A_0 B_1A_0 B_0A_0
carryout B_3A_1 B_2A_1 B_1A_1 B_0A_1
carryout B_3A_2 B_2A_2 B_1A_2 B_0A_2
C_6 C_5 C_4 C_3 C_2 C_1 C_0

We need 12 AND gates and two 4-bit adders to produce a product of 7 bits

Magnitude Comparator

- Equal (A = B)
  - $A_3=B_3$ and $A_2=B_2$ and $A_1=B_1$ and $A_0=B_0$

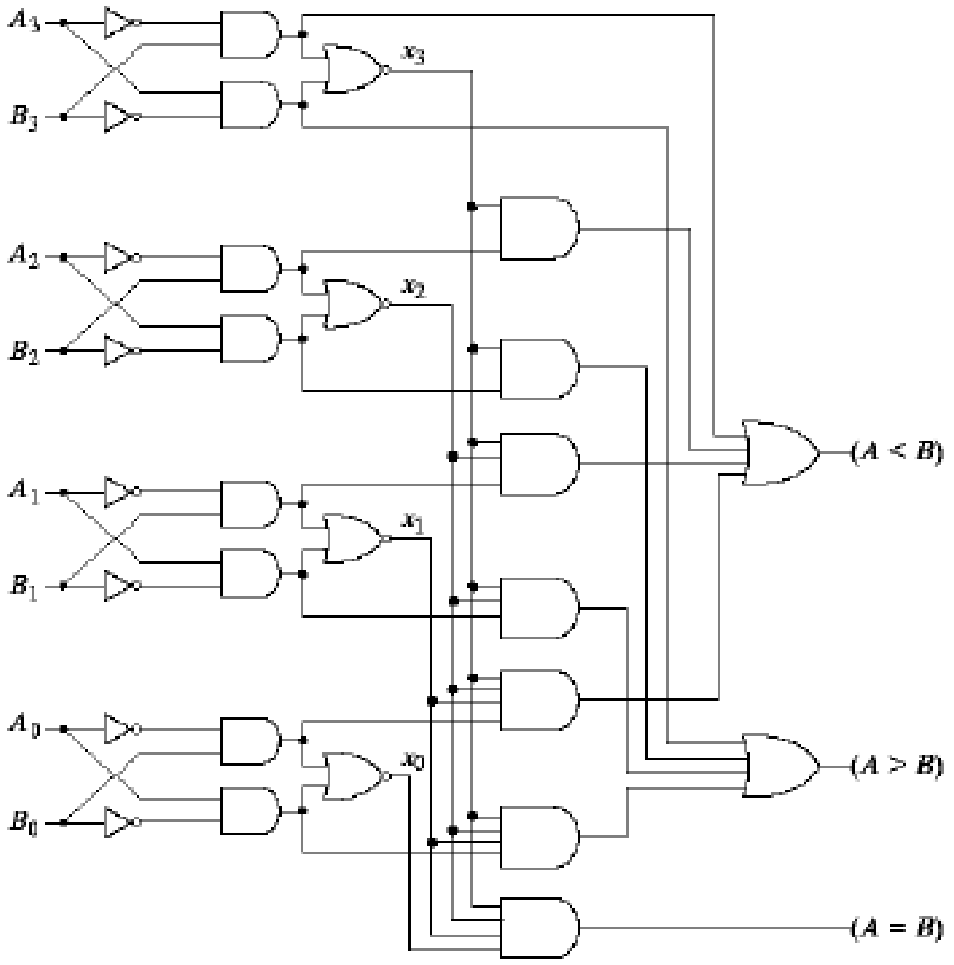
$$\boxed{X_i = A_iB_i + A_i'B_i' \quad \text{for } i = 0,1,2,3}$$

Xi = 1 means
Ai and Bi are equal !!

  - $(A=B) = X_3X_2X_1X_0$
- Greater (A > B) or Less (A < B)
  - Comparison start from the MSB
  - If the two digits are equal, compare the next lower digits
  - Continues until a pair of unequal digits is reached
    - A is 1 and B is 0 => A > B
    - A is 0 and B is 1 => A < B

$$\boxed{\begin{aligned}(A > B) &= A_3B_3' + X_3A_2B_2' + X_3X_2A_1B_1' + X_3X_2X_1A_0B_0' \\ (A < B) &= A_3'B_3 + X_3A_2'B_2 + X_3X_2A_1'B_1 + X_3X_2X_1A_0'B_0\end{aligned}}$$

4-Bit Magnitude Comparator



Outline

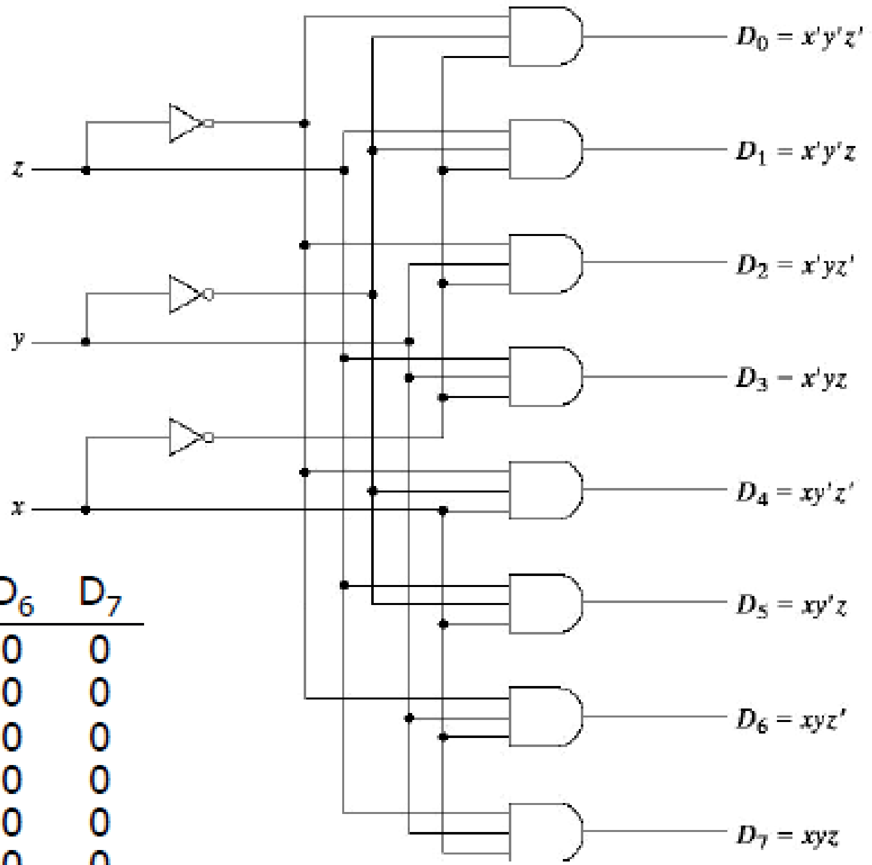
- Combinational Circuits
- Analysis and Design Procedures
- Binary Adders
- Other Arithmetic Circuits
- $\boxed{\text{Decoders and Encoders}}$
- Multiplexers

Decoder

- A circuit that coverts binary information from n input lines to a maximum of 2^n unique output lines
  - May have fewer than 2^n outputs
- A n-to-m-line decoder ($m \leq 2^n$):
  - Generate the m minterns of n input variables
- For each possible input combination, there is only one output that is equal to 1
  - The output whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines

3-to-8-Line Decoder

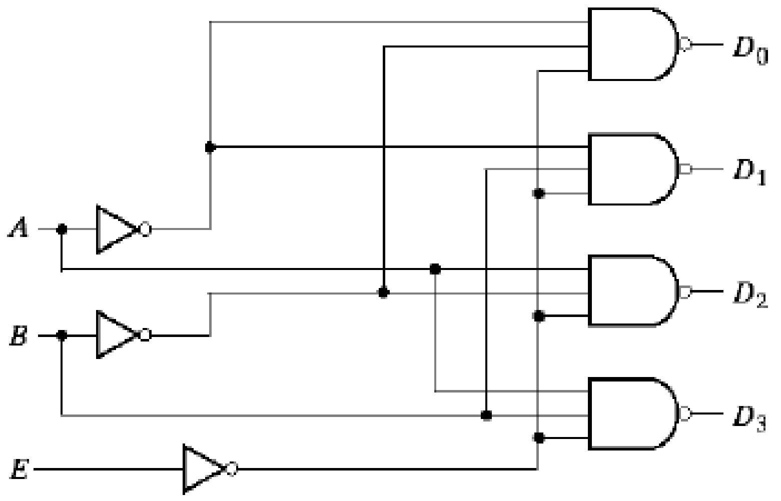
- The 3 inputs are decoded into 8 outputs
- Each represent one of the minterms of the inputs variables



Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

2-to-4-Line Decoder with Enable

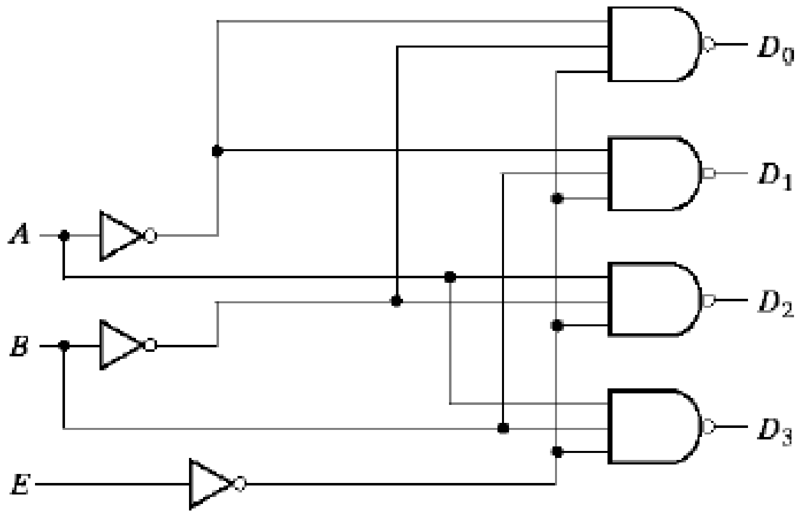
- Some decoders are constructed with NAND gates
  - Generate minterms in their complement form
- An ***enable*** input can be added to control the operation
  - E=1: disabled
  - None of the outputs are equal to 0



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Demultiplexer

- A circuit that receives information from a single line and directs it to one of 2^n possible output lines
- A decoder with enable input can function as a demultiplexer
  - Often referred to as a ***decoder/demultiplexer***



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

RTL for decoder (3:8)

```
-------------------------------------------------------------------------------
--
-- Title       : decoder3_8
-- Design      : vhdl_test

--
-------------------------------------------------------------------------------
--
-- File        : 3 : 8 Decoder using when else.vhd



library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
     port(
         din : in STD_LOGIC_VECTOR(2 downto 0);
         dout : out STD_LOGIC_VECTOR(7 downto 0)
         );
end decoder3_8;


architecture decoder3_8_arc of decoder3_8 is
begin

    dout <= ("10000000") when (din="000") else
            ("01000000") when (din="001") else
            ("00100000") when (din="010") else
            ("00010000") when (din="011") else
            ("00001000") when (din="100") else
            ("00000100") when (din="101") else
            ("00000010") when (din="110") else
            ("00000001") ;

end decoder3_8_arc;
```

Construct Larger Decoders

- Decoders with enable inputs can be connected together to form a larger decoder
- The enable input is used as the most significant bit of the selection signal
  - w=0: the top decoder is enabled
  - w=1: the bottom one is enabled
- In general, enable inputs are a convenient feature for standard components to expand their numbers of inputs and outputs

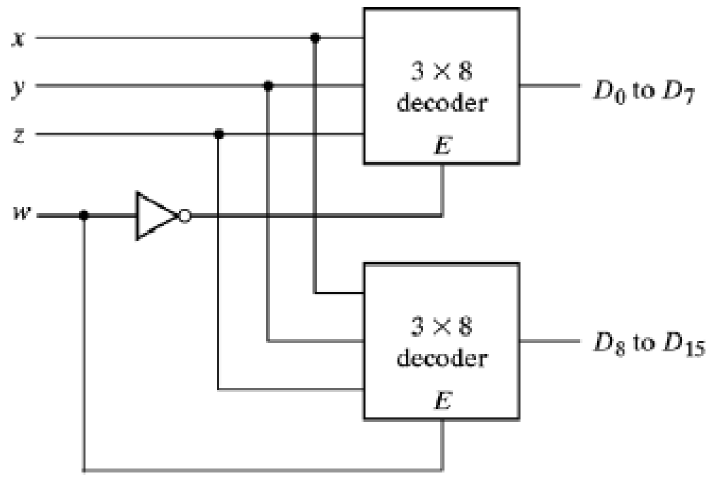


Fig. 4-20 4 × 16 Decoder Constructed with Two 3 × 8 Decoders

Encoder

- A circuit that performs the inverse operation of a decoder
  - Have 2^n (or fewer) input lines and n output lines
  - The output lines generate the binary code of the input positions
- Only one input can be active at any given time
- An extra output may be required to distinguish the cases that $D_0 = 1$ and all inputs are 0

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$$\boxed{\begin{aligned} z &= D_1 + D_3 + D_5 + D_7 \\ y &= D_2 + D_3 + D_6 + D_7 \\ x &= D_4 + D_5 + D_6 + D_7 \end{aligned}}$$

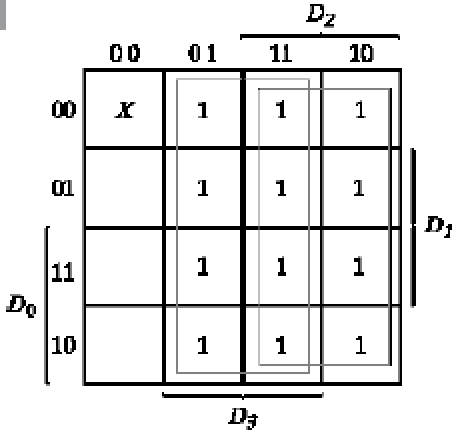
Priority Encoder

- An encoder circuit that includes the priority function
- If two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence
- In the following truth table:
  - $D_3 > D_2 > D_1 > D_0$
  - The X's in output columns represent don't-care conditions
  - The X's in input columns are useful for representing a truth table in condensed form

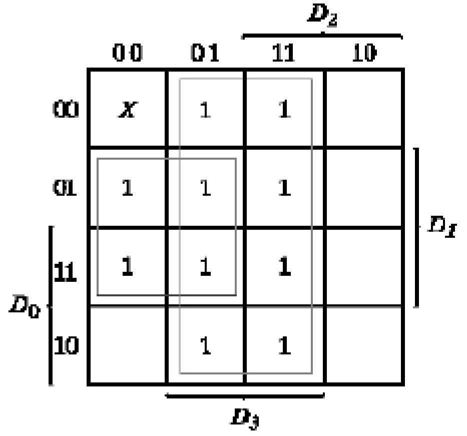
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

V = 0 :
no valid inputs

Implement a Priority Encoder



$x = D_2 + D_3$

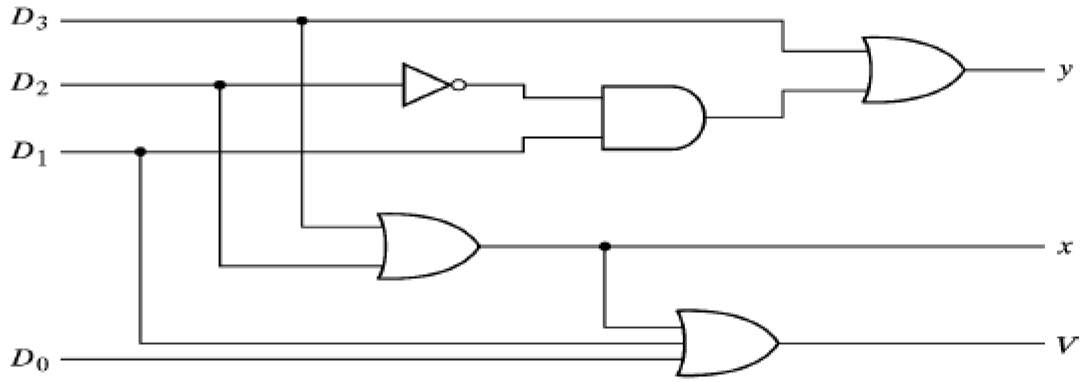


$y = D_3 + D_1D'_2$

$x = D_2 + D_3$

$y = D_3 + D_1D'_2$

$V = D_0 + D_1 + D_2 + D_3$

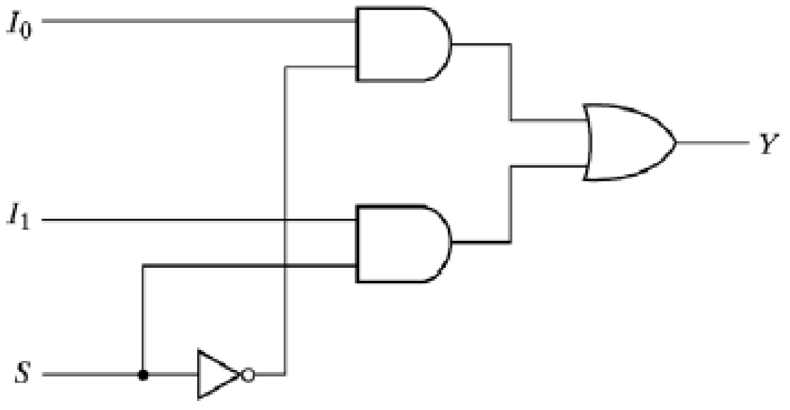


Outline

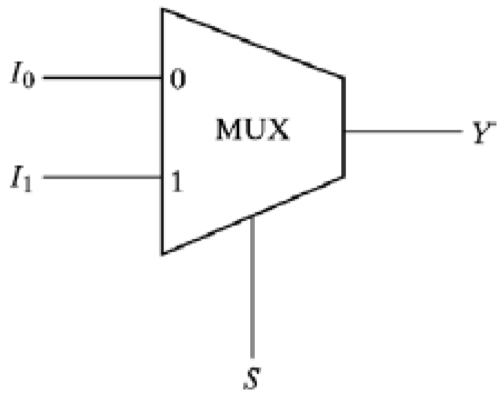
- Combinational Circuits
- Analysis and Design Procedures
- Binary Adders
- Other Arithmetic Circuits
- Decoders and Encoders
- $\boxed{\text{Multiplexers}}$

Multiplexer

- A circuit that selects binary information from one of many input lines and directs it to a single output lines
  - Have 2^n input lines and n selection lines
  - Act like an electronic switch (also called a ***data selector***)
- For the following 2-to-1-line multiplexer:
  - $S=0 \rightarrow Y = I_0$; $S=1 \rightarrow Y = I_1$



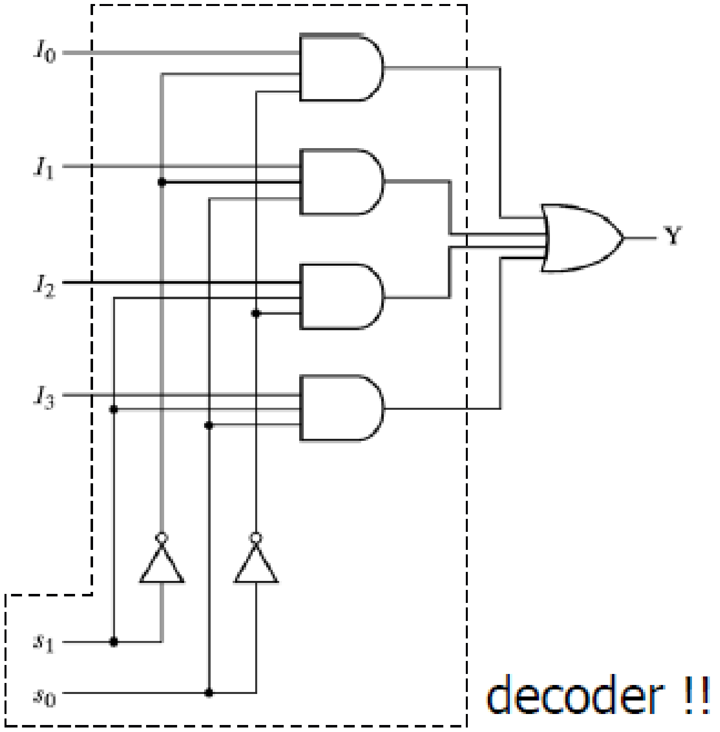
(a) Logic diagram



(b) Block diagram

4-to-1-Line Multiplexer

- The combinations of S0 and S1 control each AND gates
- Part of the multiplexer resembles a decoder
- To construct a multiplexer:
  - Start with an n-to-2^n decoder
  - Add 2^n input lines, one to each AND gate
  - The outputs of the AND gates are applied to a single OR gate



(a) Logic diagram

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

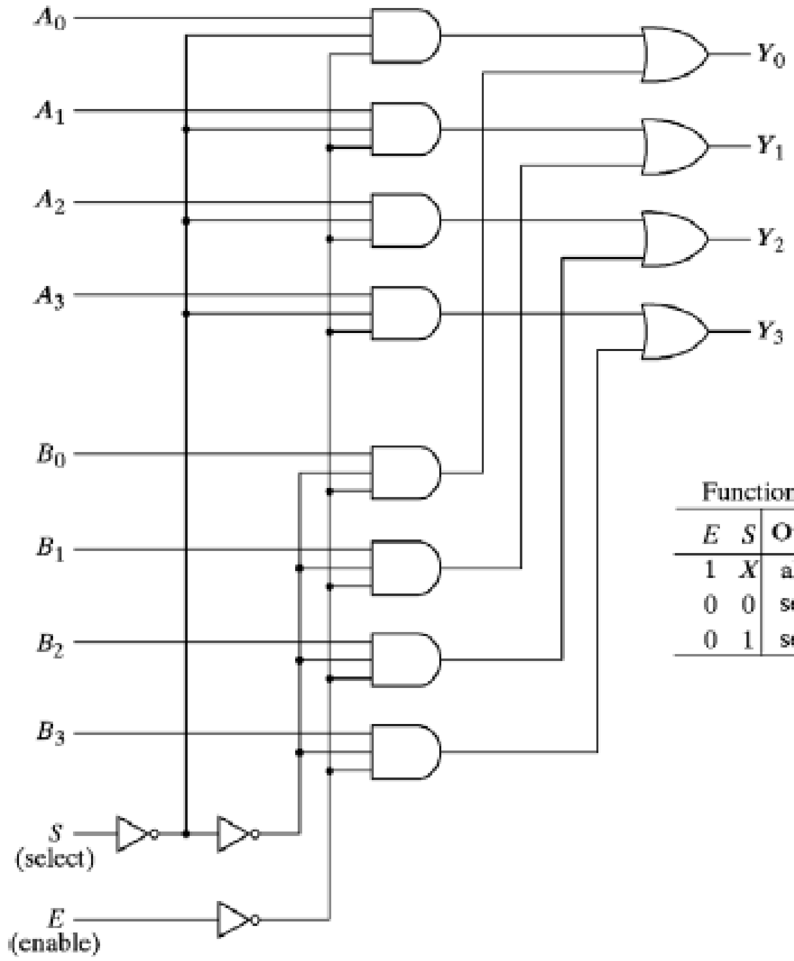
(b) Function table

Quadruple 2-to-1-Line Multiplexer

- Multiplexers can be combined with ***common selection inputs*** to provide multiple-bit selection logic
- Quadruple 2-to-1-line multiplexer:
  - Four 2-to-1-line multiplexers
  - Each capable of selecting one bit of the 2 4-bit inputs
  - E: enable input

    E=1: disable the circuit

    (all outputs are 0)



Function table

E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B