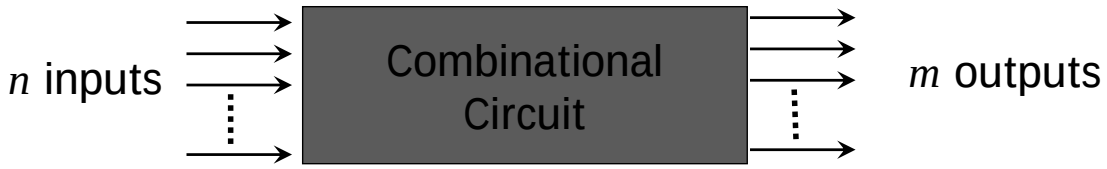
Chapter 4
Combinational Logic

Outline

- Combinational Circuits
- Analysis Procedure
- Design Procedure
- Binary Adder-Subtractor
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoders
- Multiplexers
- HDL Models of Combinational Circuits

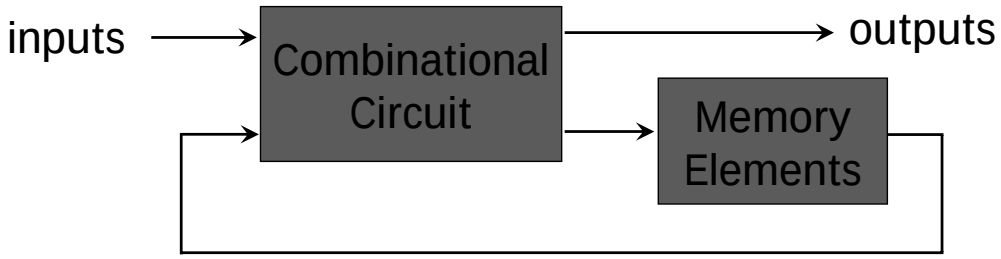
Introduction

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.



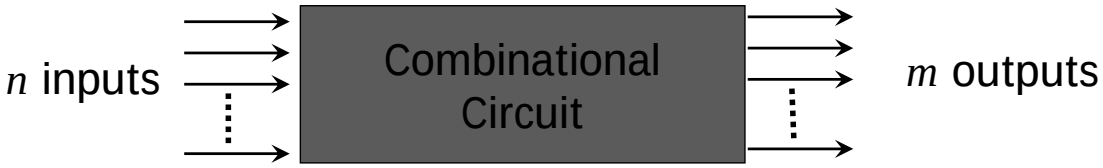
Combinational Circuits

- Logic circuits for digital system
  - Sequential circuits
    - Containing memory elements
    - The state of memory elements is a function of input.
    - The outputs are determined by the logic gates, current inputs and the state of the memory elements.
    - The outputs also depend on past inputs



Combinational Circuits

- Definition of combinational circuits
  - n input variables have 2^n possible combinations
  - m output variables describes m Boolean functions formed by combinational logic gates.



- Combinational logic can be any specific functions
  - Adders, subtractors, comparators, decoders, encoders, and multiplexers (standard cells).
  - Basic components are available (standard cell) in integrated circuit in VLSI ASIC.

Analysis Procedure

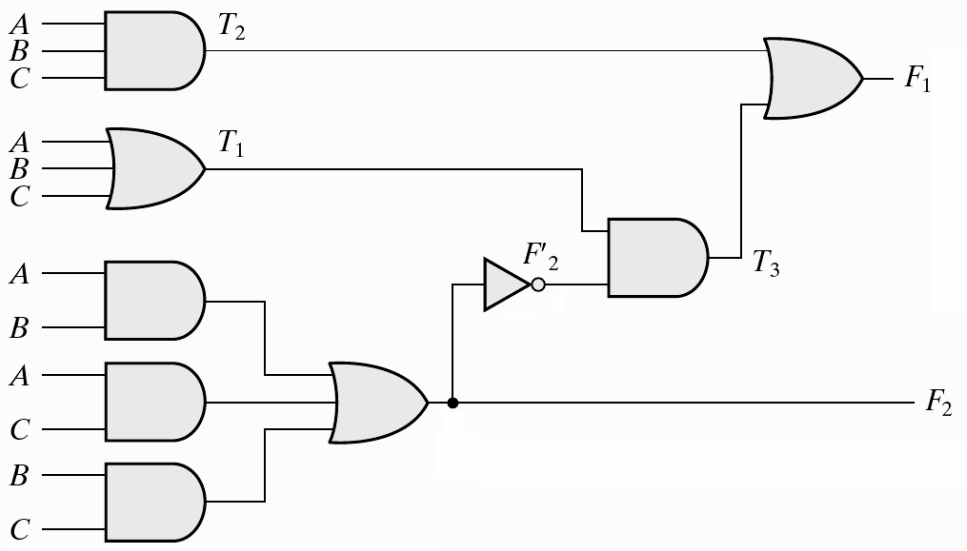
- A logic diagram is available. Analysis procedure tries to determine the function of the combination digital logic circuit for all input variables.
- Analysis procedure of combinational circuit
  - 1. To make sure that it is combinational not sequential
    - No feedback path
    - No memory element
  - 2. To derive its Boolean functions or truth table
  - 3. To verify the design
  - 4. To explain its function

Boolean Function Analysis

- To derive the Boolean function of an available circuit.
  - 1. Label all gate outputs that are a function of input variables. Determine the Boolean functions for each gate output.
  - 2. Label the gates that are a function of input variables and previously labeled gates.
  - 3. Repeat the process outlined in step 2 until all the outputs are obtained.
  - 4. Obtain the output Boolean functions in terms of input variables.

Example

- A straight-forward procedure
  - $F_2 = AB + AC + BC$
  - $T_1 = A + B + C$
  - $T_2 = ABC$
  - $T_3 = F_2'T_1$
  - $F_1 = T_3 + T_2$



Example

- $F_1 = T_3 + T_2 = F_2'T_1 + ABC$
  $= (AB + AC + BC)'(A + B + C) + ABC$
  $= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC$
  $= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC$
  $= A'BC' + A'B'C + AB'C' + ABC$
- A full-adder
  - F_1: the sum
  - F_2: the carry

Example

- The truth table

Table 4.1
Truth Table for the Logic Diagram of 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

- The design procedure of combinational circuits
  - 1. State the specification or a problem, determine the inputs and outputs, and assign a symbol to each input/output.
  - 2. Derive the truth table that defines the relationship between inputs and outputs.
  - 3. Derive the simplified Boolean functions for each output.
  - 4. Draw the logic diagram and verify the correctness.

Design Procedure

- Functional description
  - Boolean function
  - HDL (Hardware description language)
    - Verilog HDL
    - VHDL
  - Schematic entry
- Logic minimization
  - number of gates
  - number of inputs to a gate
  - propagation delay
  - number of interconnection
  - limitations of the driving capabilities

Code Conversion Example

- BCD to excess-3 code

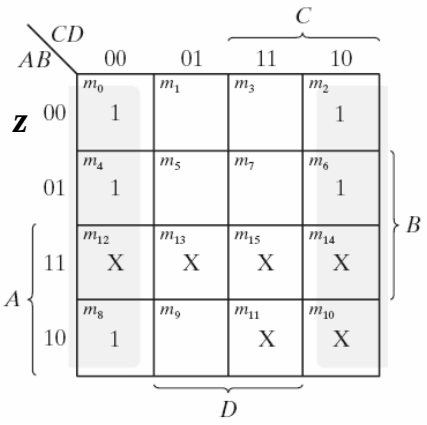
  The truth table:

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

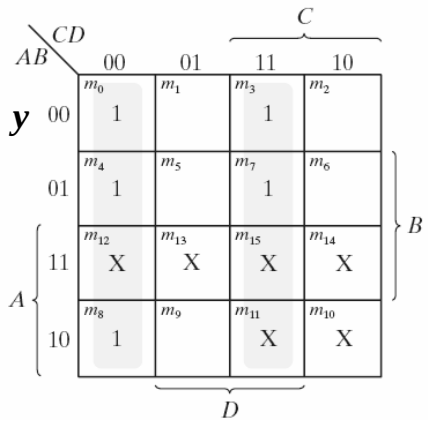
- The 4'b1010 ~ 4'b1111 never appear in the BCD code, therefore the condition can be viewed as don't-care conditions.

The K-map of Excess-3 Code

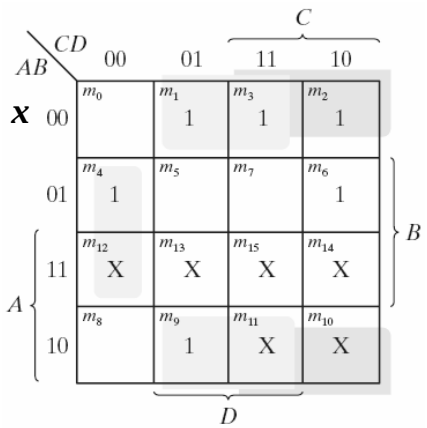
Output: z



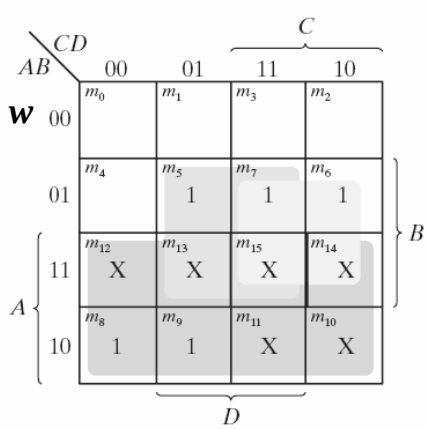
Output: y



Output: x



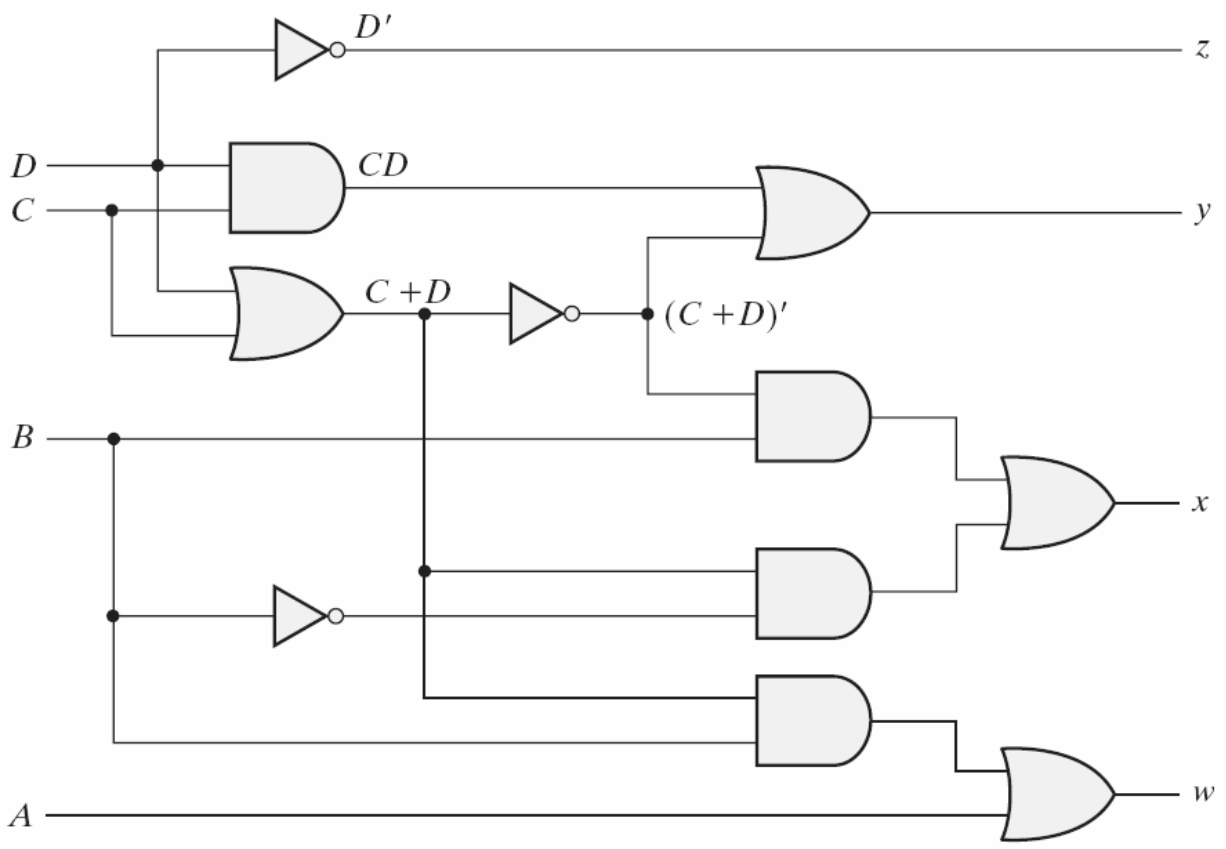
Output: w



The K-map of Excess-3 Code

- The simplified functions
  - $z = D'$
  - $y = CD + C'D'$
  - $x = B'C + B'D + BC'D'$
  - $w = A + BC + BD$
- Another implementation
  - $z = D'$
  - $y = CD + C'D' = CD + (C + D)'$
  - $x = B'C + B'D + BC'D' = B'(C + D) + B(C + D)'$
  - $w = A + BC + BD$

The Logic Diagram



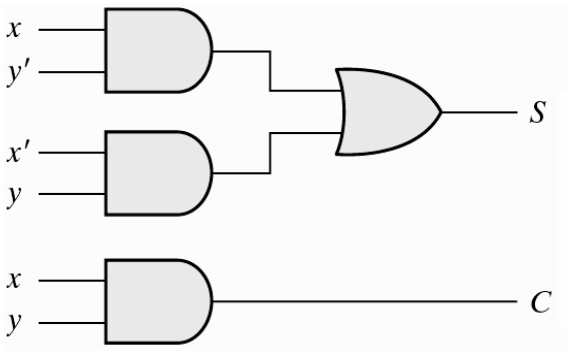
Binary Adder-Subtractor

- The adder/subtractor are the most basic arithmetic operations in digital computers.
- Half adder
  - 0 + 0 = 0 ; 0 + 1 = 1 ; 1 + 0 = 1 ; 1 + 1 = 10
  - two input variables: x, y
  - two output variables: C (carry), S (sum)
  - truth table

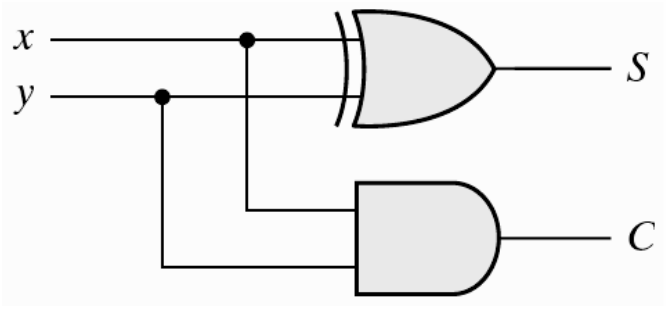
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Logic Diagram of a Half-Adder

- Boolean Expression
  - $S = x'y + xy'$
  - $C = xy$



- The flexibility for implementation
  - $S = x \oplus y$
  - $S = (x + y)(x' + y')$
  - $S' = xy + x'y'$
  - $S = (C + x'y')'$
  - $C = xy = (x' + y')'$

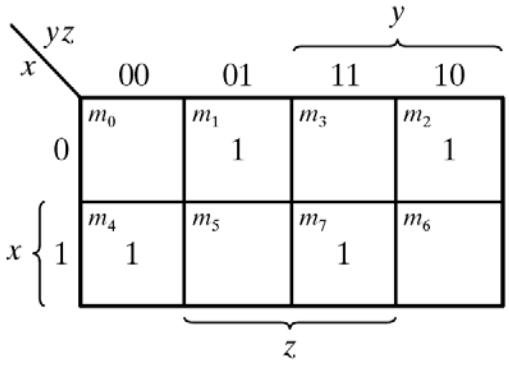


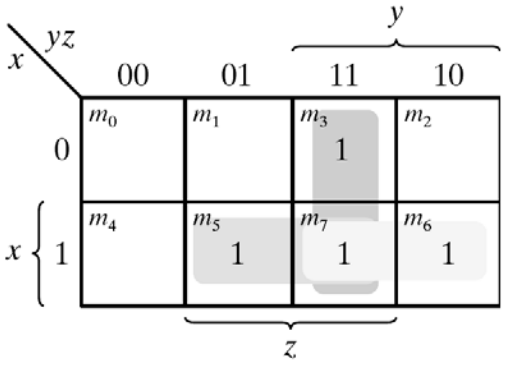
Full-Adder

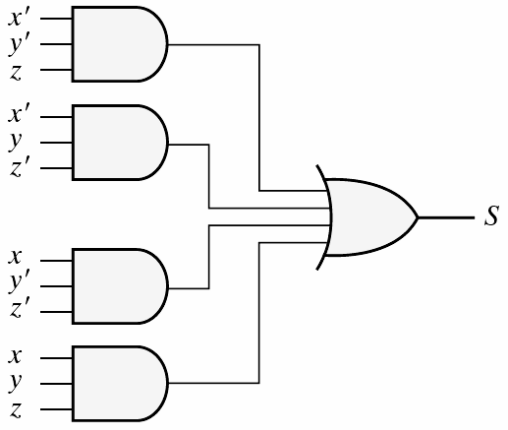
- The arithmetic sum of three input bits
- Three input bits
  - x, y: two significant bits
  - z: the carry bit from the previous lower significant bit
- Two output bits: C, S

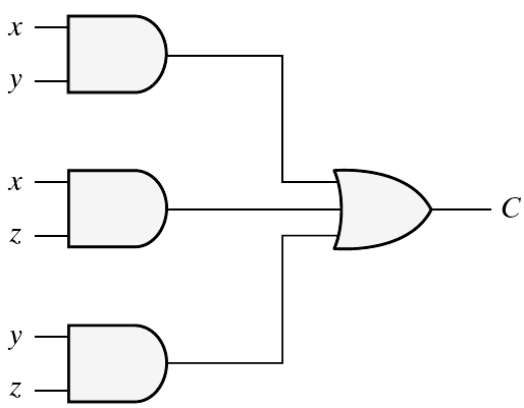
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

K-map and Logic Diagram of a Full-Adder









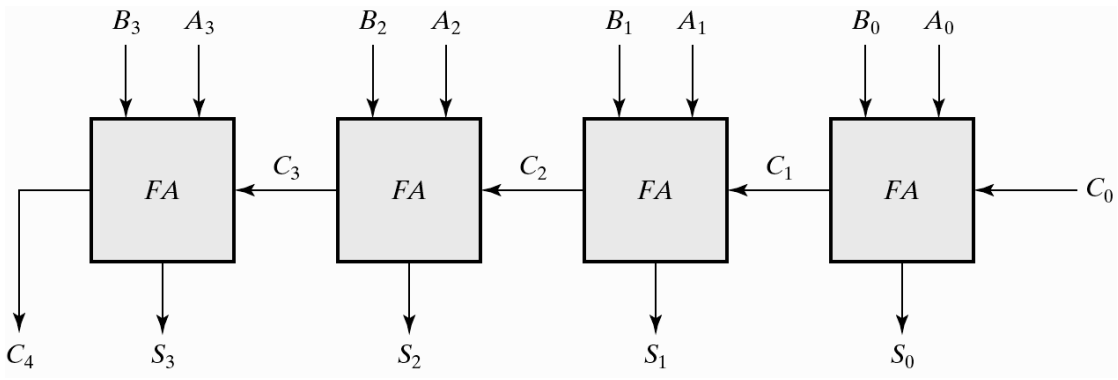
Implementation of Full Adder

- Boolean expression of full-adder
  - $S = x'y'z + x'yz' + xy'z' + xyz = z \oplus (x \oplus y)$
  - $C = xy + xz + yz$
- Implementation of full-adder using two half-adders and one OR gate.
  - $S = z \oplus (x \oplus y) = z'(xy' + x'y) + z(xy' + x'y)'$
    $= z'xy' + z'x'y + z((x' + y)(x + y'))$
    $= xy'z' + x'yz' + xyz + x'y'z$
  - $C = \underline{z(xy' + x'y) + xy}$
    $= z(x \oplus y) + xy$
    $= xy'z + x'yz + xy$

N-bit Binary Adder (Ripple Carry Adder)

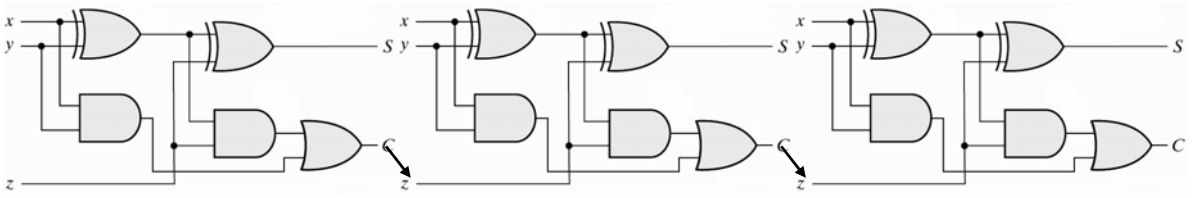
- By cascading Full-Adders, a 4-bit Binary adder can be built up through the carry-in/carry-out interconnections.

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}



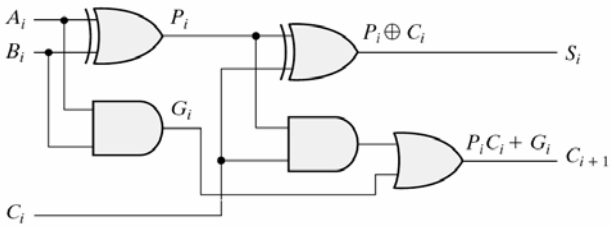
Carry-Propagation in Ripple-Carry Adder

- Carry propagation time is a limiting factor on the speed with two numbers added.
  - C_{i+1} must wait the steady-state C_i to get its correct value.
  - Critical path : $(A_0, B_0, C_0) \Rightarrow C_1 \Rightarrow C_2 \Rightarrow C_3 \Rightarrow (C_4, S_3)$
  - Carry signal must go through 8 gate levels to get its final value. Therefore, the delay time of n-bit is proportional to its bit-width n.
  - Reducing the gate-delay time of gates can improve the speed of the Adder, but it is usually not sufficient.



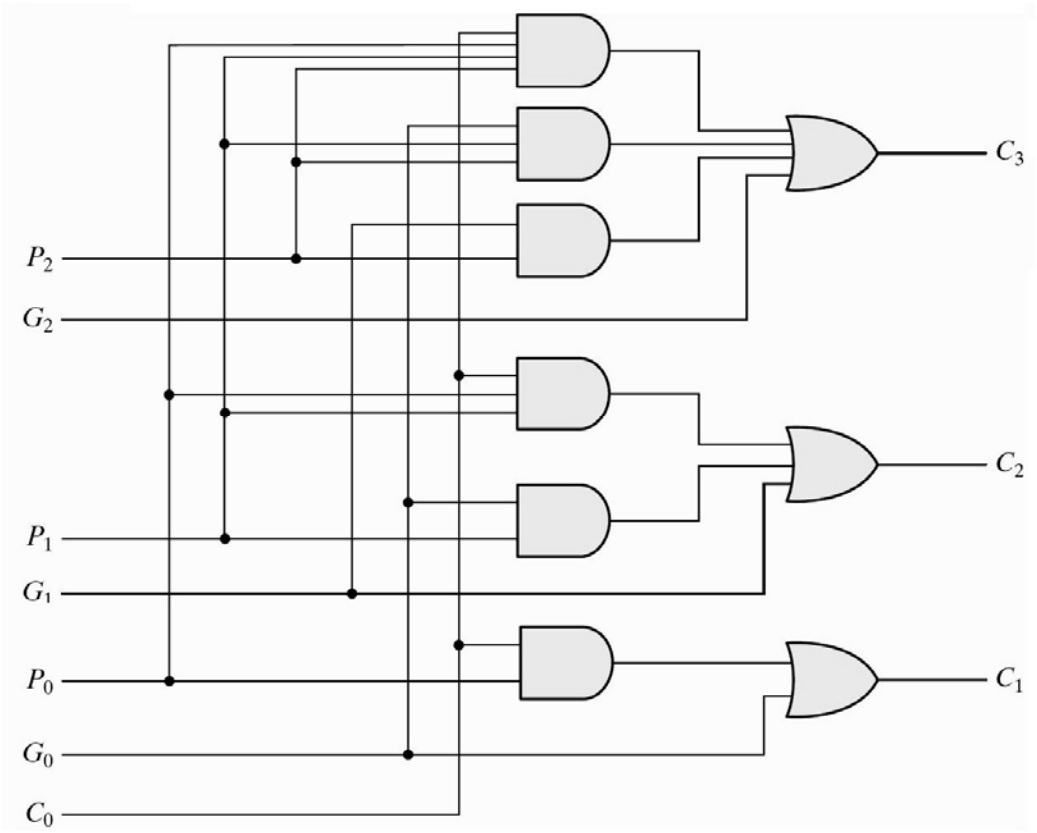
Carry Lookahead Adder

- A faster look-ahead carry mechanism can be employed to improve the speed of an n-bit adder.
- Definition of signals in carry-lookahead adder
  - carry propagate: $P_i = A_i \oplus B_i$
  - carry generate: $G_i = A_i B_i$
  - sum: $S_i = P_i \oplus C_i$
  - carry: $C_{i+1} = G_i + P_i C_i$



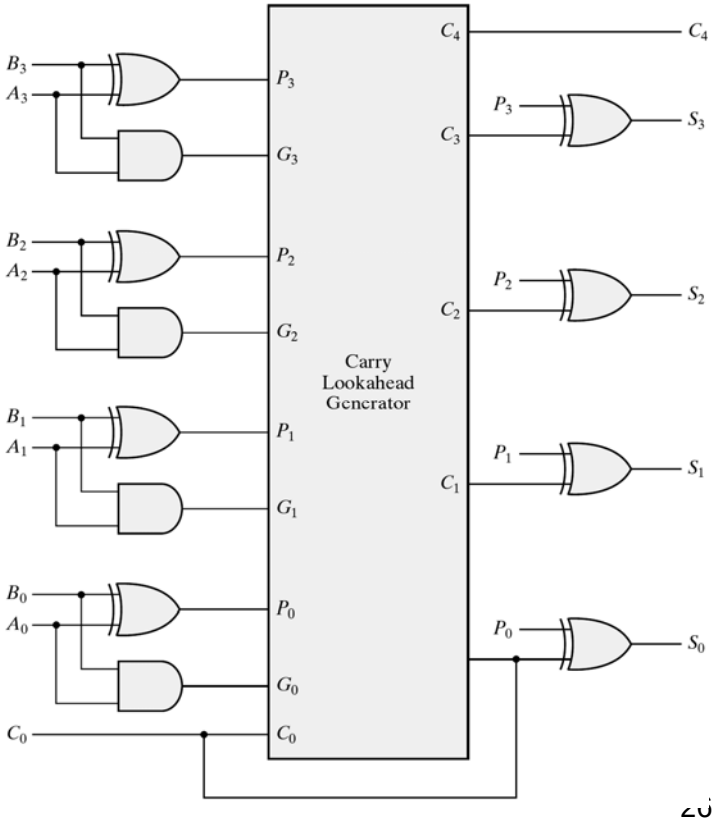
- Carry outputs of each stage can be substituted with the carry of previous equations. This will form the logic diagram of Carry Lookahead Generator in the next page.
  - $C_1 = G_0 + P_0 C_0$
  - $C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$
    $= G_1 + P_1 G_0 + P_1 P_0 C_0$
  - $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

Logic Diagram of a Carry Lookahead Generator



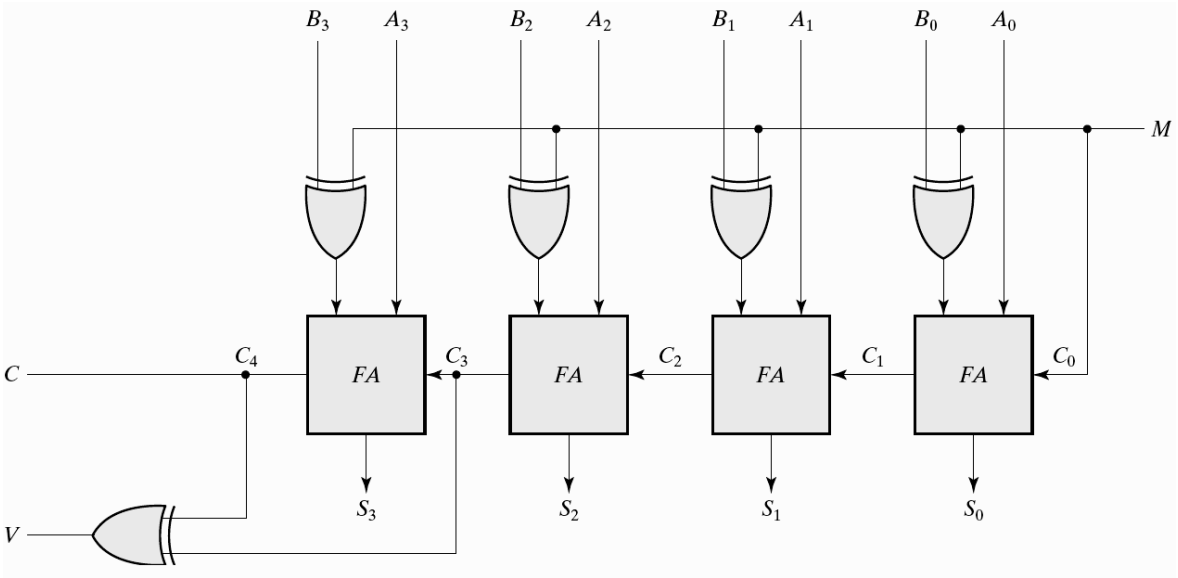
4-bit Carry Lookahead Adder

- Since the Carry Lookahead Generator has fixed 2-level gate delay, the delay of *n*-bit adder has a fixed delay with no relation to the carry-propagation time or bitwidth.



Binary Subtractor

- $A - B = A$ + (2's complement of B)
- 4-bit Adder-Subtractor
  - M = 0, A + B (Addition)
  - M = 1, A + B' + 1 (Subtraction)



Overflow in Digital Number

- Overflow
  - The storage is limited in digitalized hardware.
- Unsigned Number
  - Overflow can be detected by the end carry out of the most significant bit (MSB).
- Signed Number
  - Add two positive numbers and obtain a negative number
  - Add two negative numbers and obtain a positive number
  - V = 0, no overflow; V = 1, overflow

```
+70  0 1000110          -70  1 0111010
+80  0 1010000          -80  1 0110000
-----------------      -----------------
+150 1 0010110          -150 0 1101010
     ↑                       ↑
     └──→ V signal ←─────────┘
```

Decimal Adder

- Addition of two decimal digits in BCD.
  - Each 4-bit signal input does not exceed 9.
  - Decimal Adder has 9 inputs, 5 outputs.
  - Summation of two BCD digits can not be greater than 9 + 9 + 1 = 19.
  - Carry out modification: The carry_out = 1 when number in binary exceeds 15, but carry_out = 1 when number exceeds 9 in BCD.
  - When 4-bit sum > 1001, we need to add 0110 to the sum to modify the output.
  - BCD modification: The adder will form the sum in binary ranging from 0_0000 to 1_0011, which must be converted to BCD representation again.

Truth Table of BCD Adder

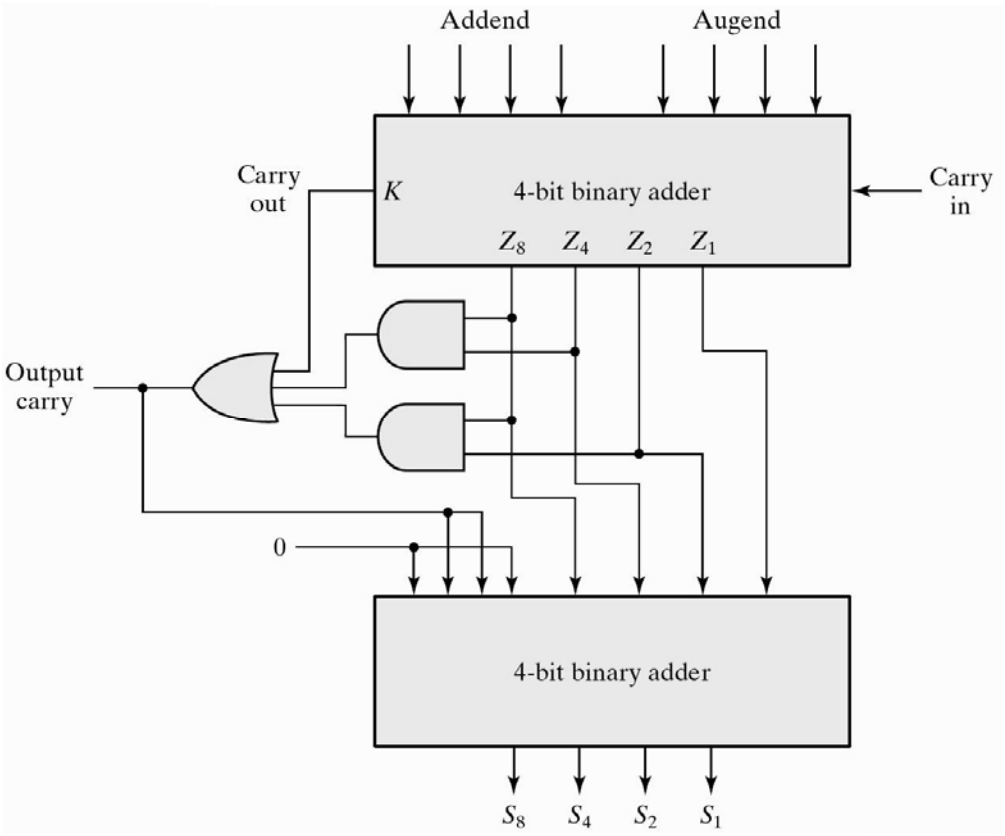
Table 4.5
Derivation of BCD Adder

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	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

Decimal Adder

- Carry out modifications are needed if the sum is greater than 9
  - BCD carry out $C = K + Z_8Z_4 + Z_8Z_2$
- BCD modification is needed if the sum is greater than 9, i.e., $C = 1$.
  - Addition of $-(10)_d$ or +6 is applied to binary number.

BCD Adder



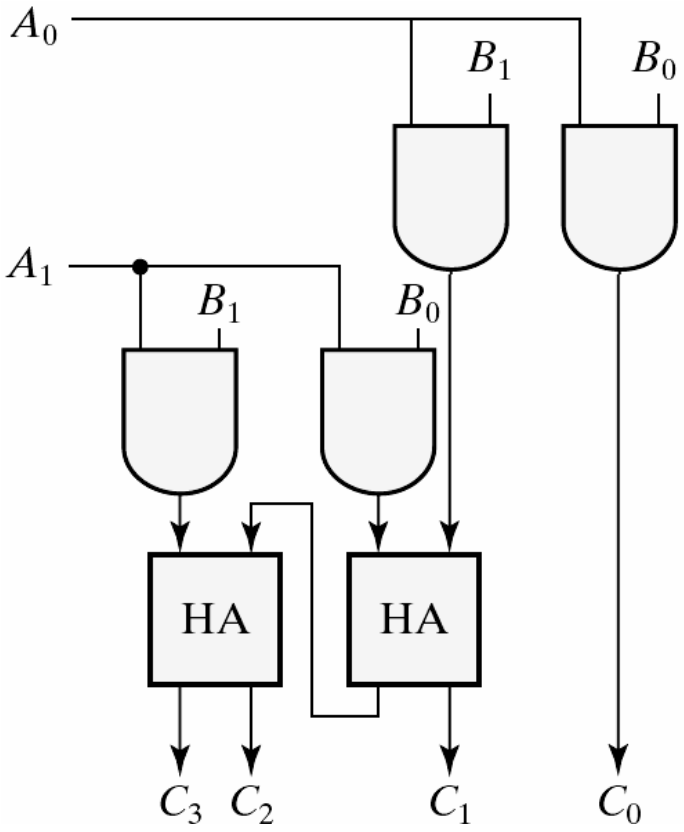
Binary Multiplier

- Partial products
  - AND operations

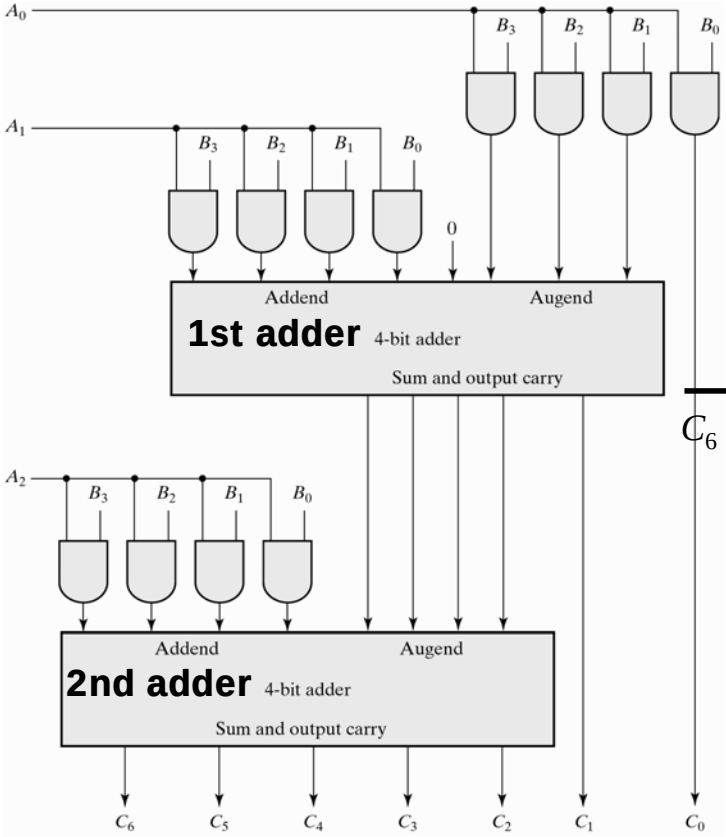
2-bit by 2-bit Multiplier

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

Half Adder



4-bit by 3-bit Binary Multiplier



$$\begin{array}{ccccccc} B_3 & B_2 & B_1 & B_0 & & & \\ & A_2 & A_1 & A_0 & & & \\ \hline & A_0B_3 & A_0B_2 & A_0B_1 & A_0B_0 & & \\ A_1B_3 & A_1B_2 & A_1B_1 & A_1B_0 & & & \\ A_2B_3 & A_2B_2 & A_2B_1 & A_2B_0 & & & \\ \hline C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 \end{array}$$

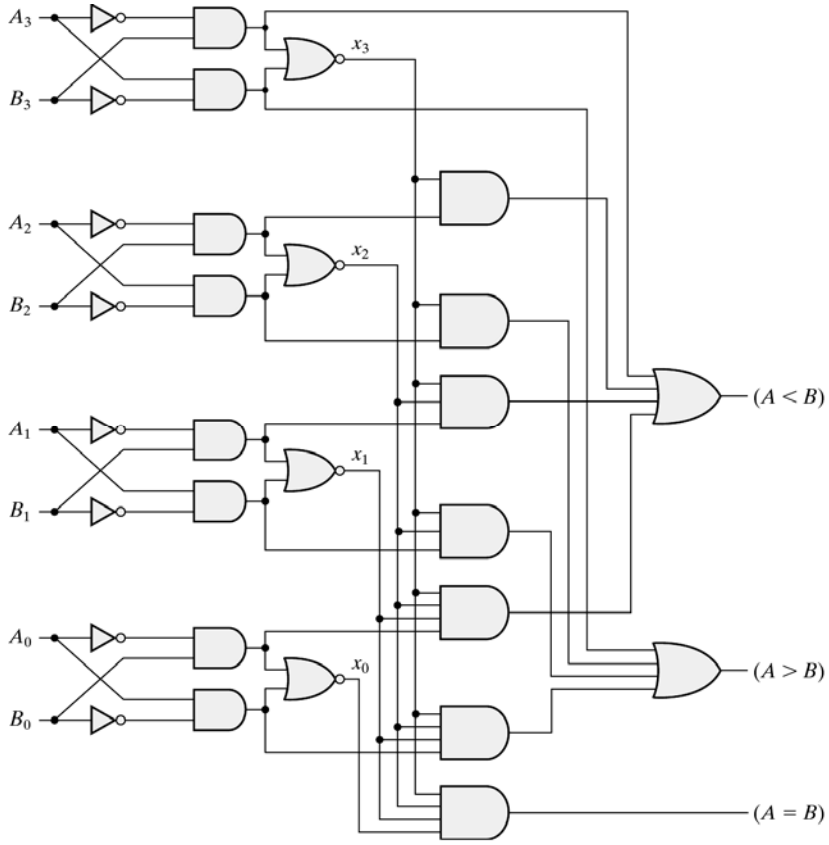
Magnitude Comparator

- The comparison of two numbers
  - outputs: $A > B$, $A = B$, $A < B$
- Design Approaches
  - the truth table for 2 n-bit numbers:
    - 2^{2n} entries - too cumbersome for large n
  - use inherent regularity of the problem
    - reduce design efforts
    - reduce human errors
    - reduce circuit complexity

Magnitude Comparator

- Algorithm procedure is used to compare the two numbers in the logic level. Compare the two numbers from MSB digit to LSB digit.
  - $A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$
  - $A = B$ if $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_1 = B_1$
    - equality: $x_i = A_iB_i + A_i'B_i'$
    - $(A = B) = x_3x_2x_1x_0$
  - $(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$
  - $(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$
- Implementation
  - $x_i = (A_iB_i' + A_i'B_i)'$

Four-Bit Magnitude Comparator



Decoder

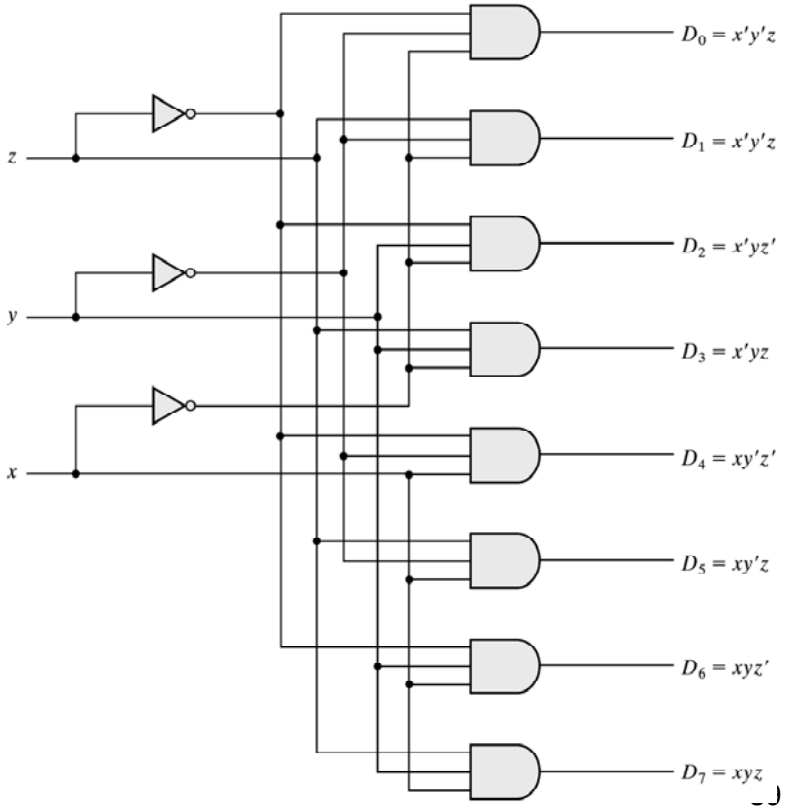
- An n-to-m decoder
  - a binary code of n bits = 2^n distinct information
  - n input variables; up to 2^n output lines
  - only one output can be active (high) at any time

Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

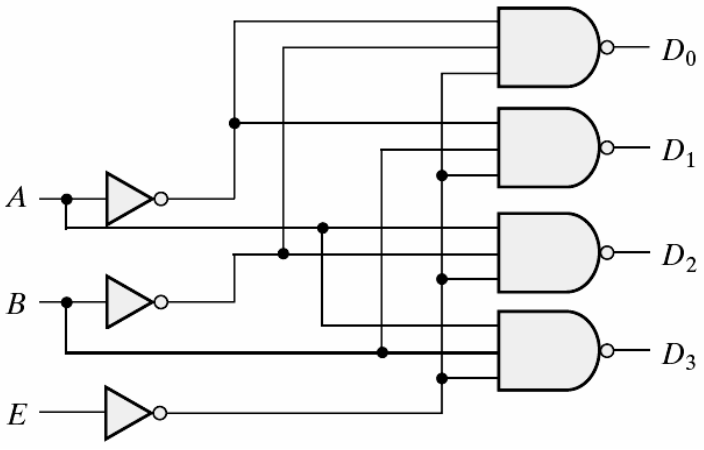
Inputs			Outputs							
x	*y*	*z*	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Combinational Logic of n-to-m Decoder

Logic diagram of n-bit decoder can be derived from 8 minterms of the three input signals.



Decoder with Enable Input

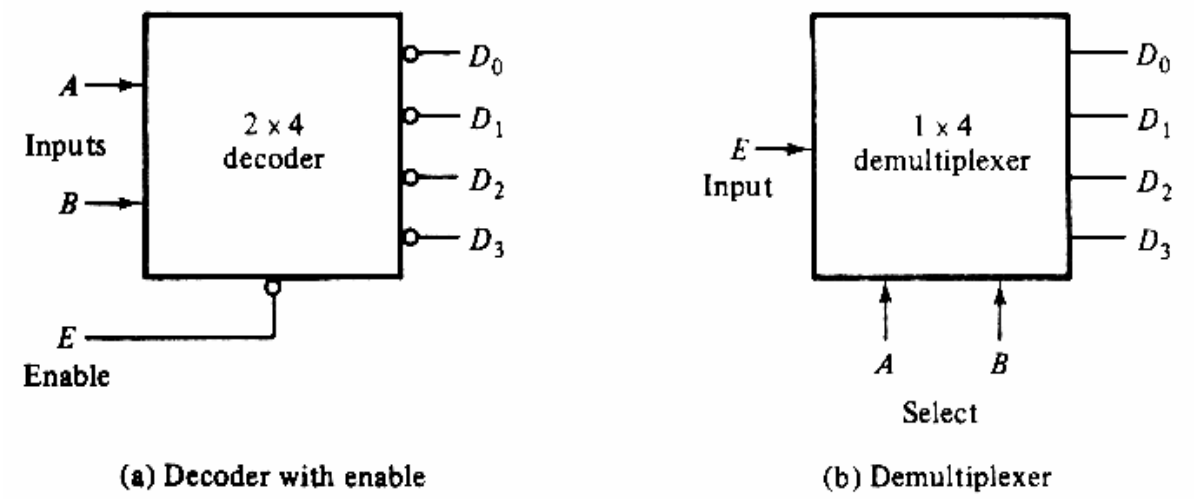


E = 1, disable
E = 0, enable

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

- A decoder with enable input can function as a demultiplexer.
  - Received Signal: Enable Input
  - Selection Signal: A, B
  - Received signal is directed to one of the output according to selection signal A and B.

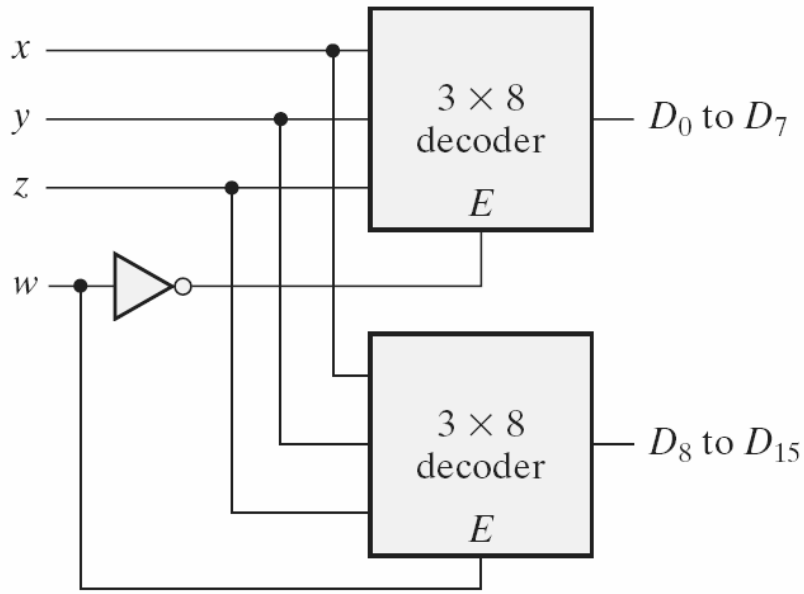
Decoder / Demultiplexers



Function of a demultiplexer: A circuit that receives information from a single line and directs it to one of 2^n possible output lines.

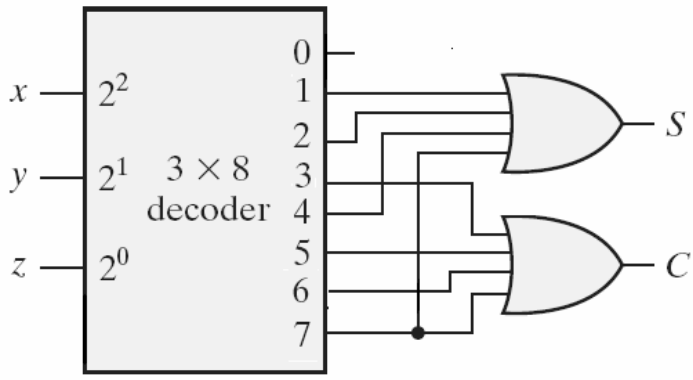
Decoder Expansion

- Decoder with enable input can be expanded to higher-level decoder
  - Two 3-to-8 decoders ⇒ a 4-to-16 decoder



Combinational Logic Implementation

- General Boolean algebra
  - Sum of all possible minterms
- each output = a minterm
- A decoder and an external OR gate can be used to implement any Boolean function of n input variables
  - A full-adder
  - $S(x, y, z) = \Sigma(1, 2, 4, 7)$
    $C(x, y, z) = \Sigma(3, 5, 6, 7)$



Combinational Logic Implementation

- Two possible approaches of using a decoder to implement a Boolean function
  - OR (minterms of F): k inputs
  - NOR (minterms of F'): $2^n - k$ inputs

Depends on how many inputs to the gate.

- In general, it is not a practical implementation because the cost is maximum.
- If the decoder is implemented by NAND gate, then the external gates must be NAND gates instead of OR gates. Why?

Encoder

- The inverse function of a decoder
  - Only one input has a value of 1 at any given time.
- Truth table method
  - $x = D_4 + D_5 + D_6 + D_7$
  - $y = D_2 + D_3 + D_6 + D_7$
  - $z = D_1 + D_3 + D_5 + D_7$

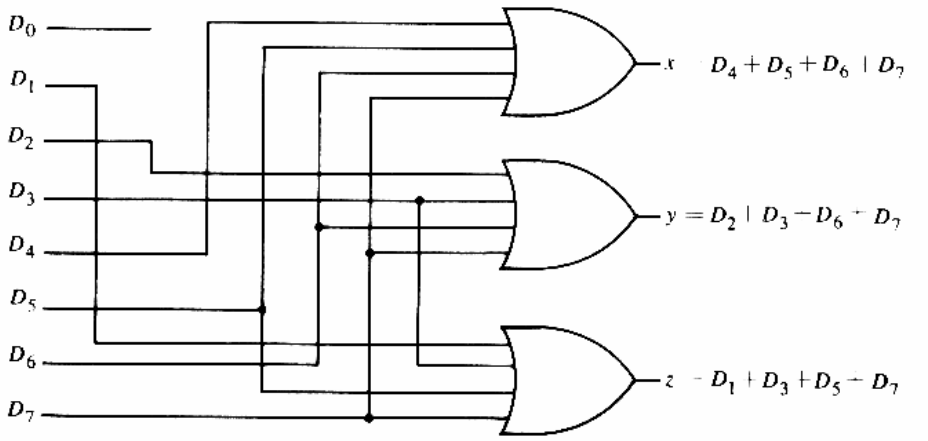
The encoder can be implemented with three OR gates.

Table 4.7
Truth Table of an Octal-to-Binary Encoder

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

Encoder

- Combinational logic implementation



- limitations of input signals: only one input can be active at any given time.
  - Example of illegal signals : $D_3 = D_6 = 1$
  - the output = 111 (13 and 16)

Priority Encoder

- Priority encoder is used to resolve the ambiguity of illegal inputs.
- Only one of the inputs is encoded

Table 4.8
Truth Table of a Priority Encoder

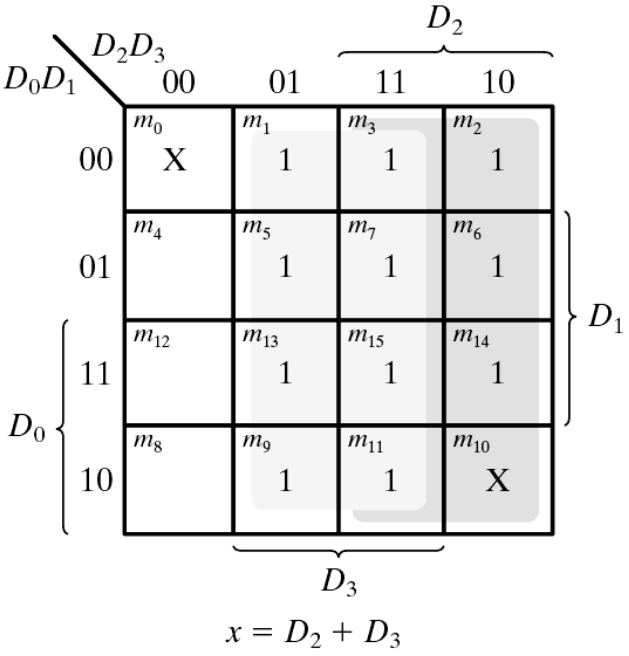
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Truth Table Method
- $x = D_2+D_3$
- $y = D_3+D_1D_2'$
- $V = D_0+D_1+D_2+D_3$

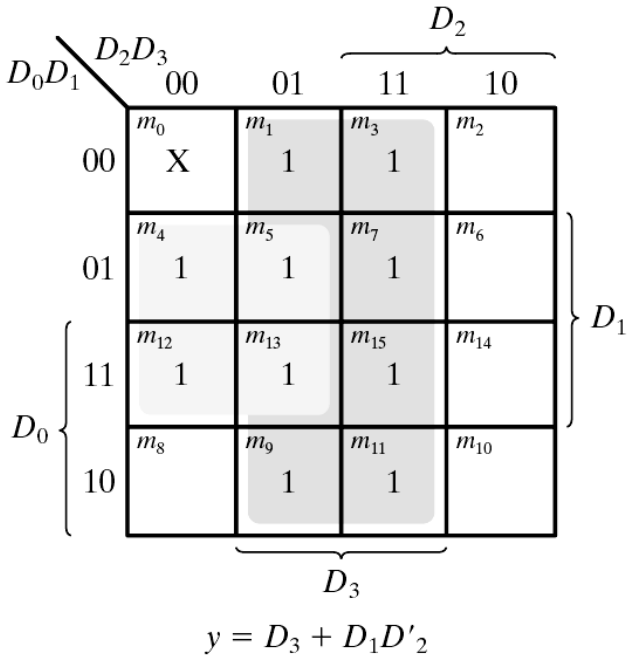
D_3 has the highest priority
D_0 has the lowest priority

X: don't-care conditions
V: valid output indicator

The Maps for Simplifying Outputs x and y

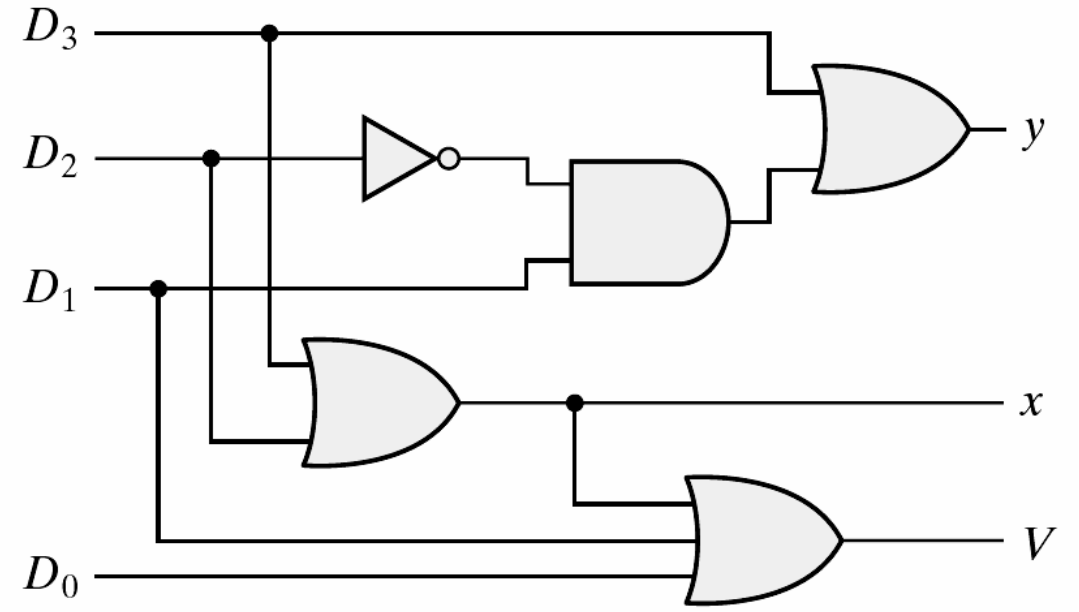


$x = D_2 + D_3$



$y = D_3 + D_1D'_2$

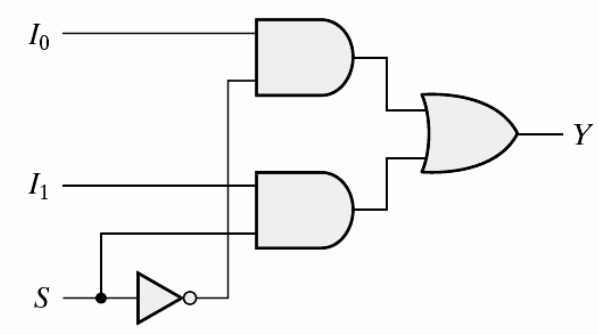
Implementation of Priority



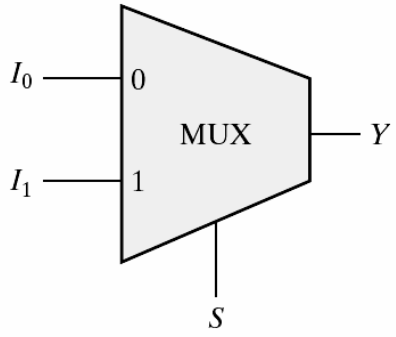
$$x = D_2 + D_3$$
$$y = D_3 + D_1 D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$

Multiplexers (Data Selector)

- Multiplexer selects binary information from one of many inputs lines and direct it to a single output line.
  - 2^n input lines, n selection lines and one output line
  - e.g.: 2-to-1-line multiplexer

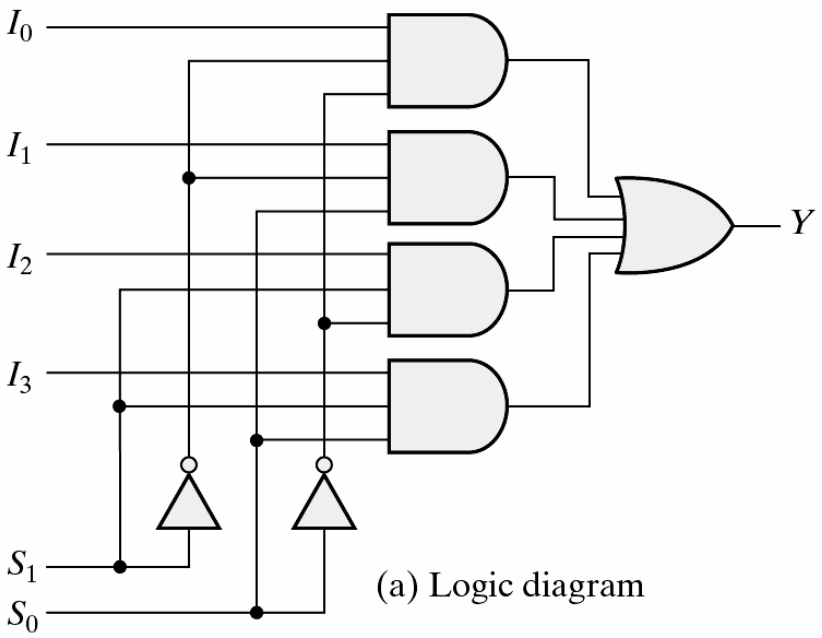


(a) Logic diagram



(b) Block diagram

4-to-1-line Multiplexer



(a) Logic diagram

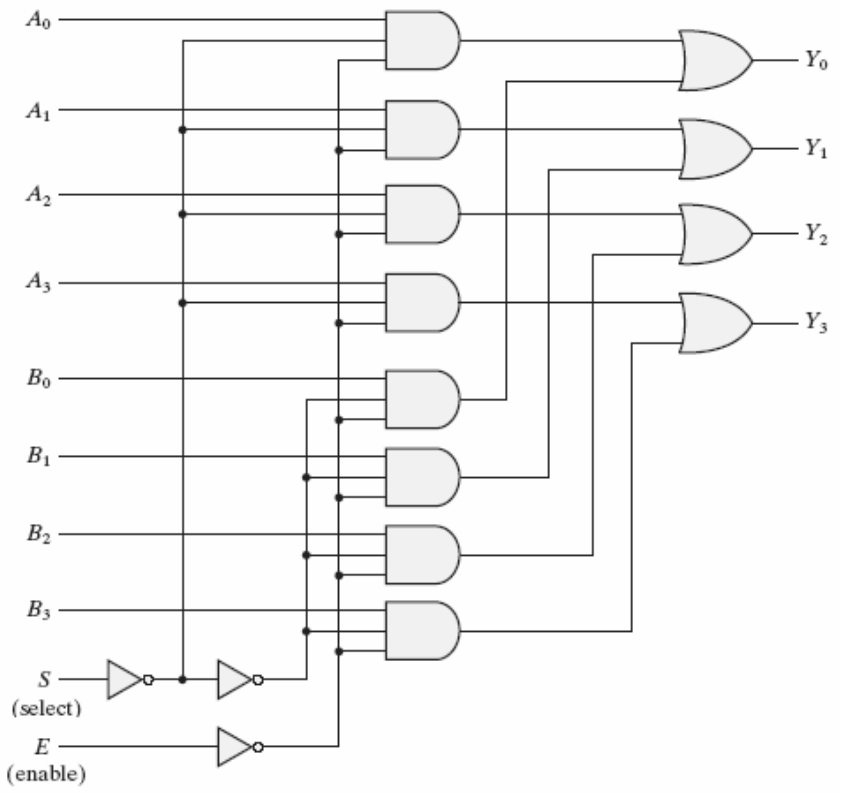
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Multiplexer as a Decoder

- Decoder explanation of a multiplexer
  - A multiplexer can be used to decode the selections.
  - The multiplexer decodes the selection input lines as an n-to-2^n decoder
  - Add the 2^n input lines to each AND gate and ORs all AND gates.
  - An enable input (an option).

Quadruple Two-to-One-Line Multiplexer



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

Function table

Boolean Function Implementation

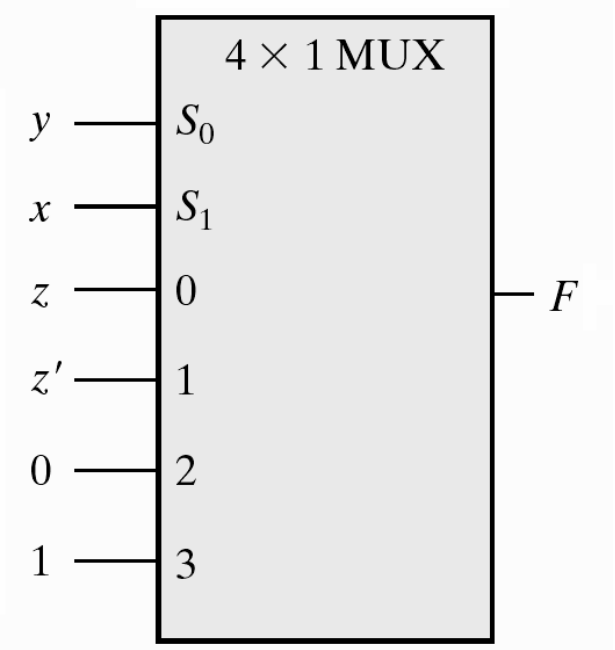
- Multiplexer is a decoder plus an OR gate.
  - 2^n-to-1 MUX can implement any Boolean function of n input variable
- A better solution: implement any Boolean function of n input variables
  - $(n - 1)$ of these variables: the selection lines
  - The remaining variable: the input

Boolean Function with a Multiplexer

- $F(x, y, z) = \Sigma(1, 2, 6, 7)$

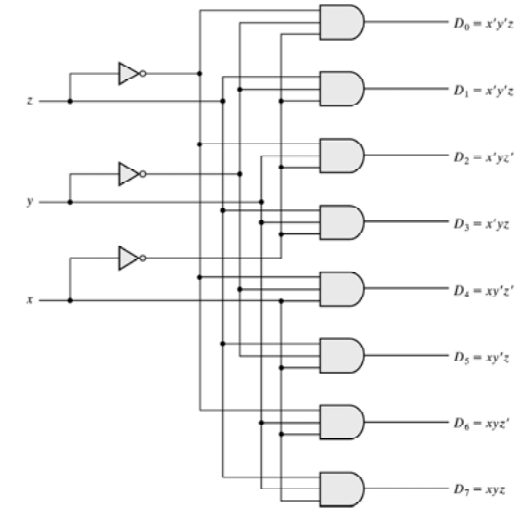
x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

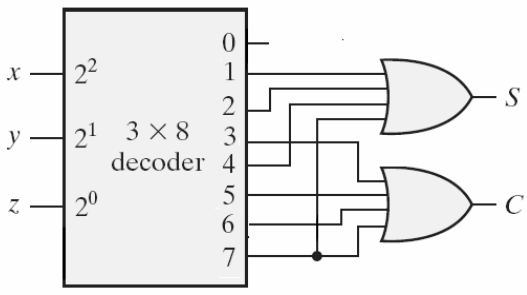
(a) Truth table

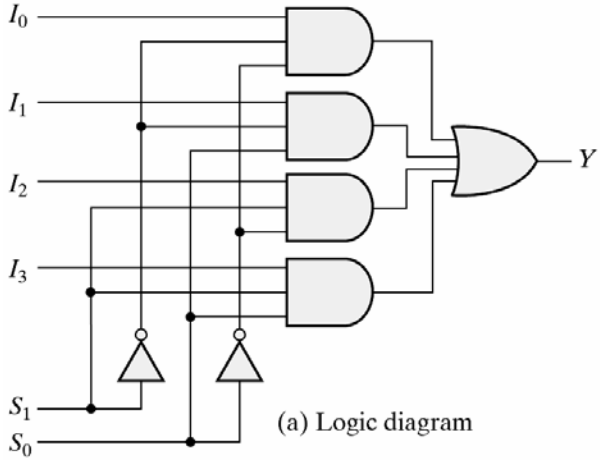


(b) Multiplexer implementation

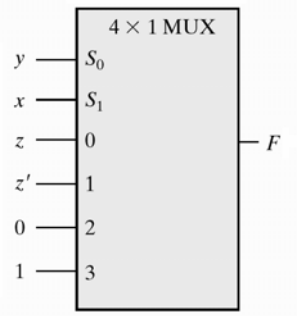
Comparison b/w Decoder and Encoder as a Boolean Function







(a) Logic diagram



(b) Multiplexer implementation

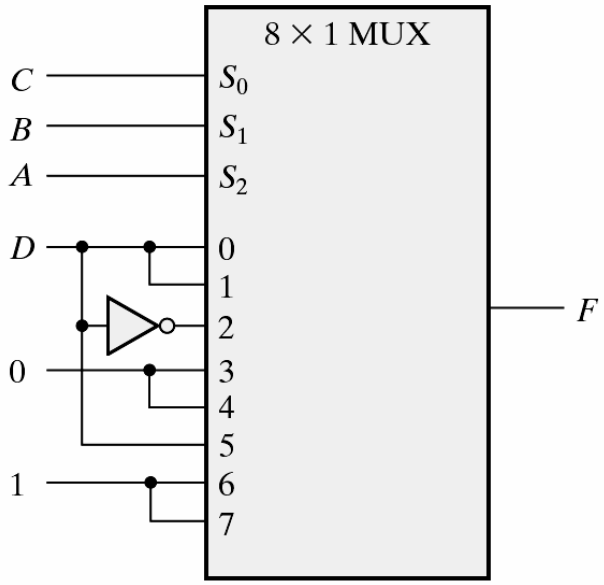
Design Procedure

1. Assign an ordering sequence of the $(n - 1)$ input variables (xy in the example) to the selection input of multiplexer.
2. The last variable (z in the example) will be used for the input lines.
3. Construct the truth table.
4. Consider a pair of consecutive minterms starting from m_0.
5. Determine the input lines according to last variable (z) and output signals (F) in the truth table.

4-input Function with a Multiplexer

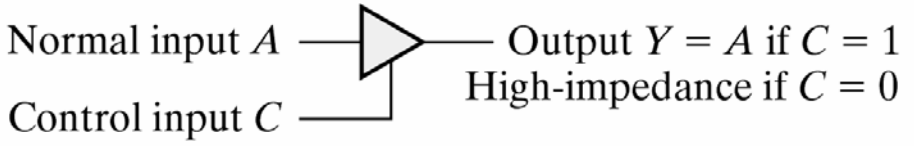
A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$



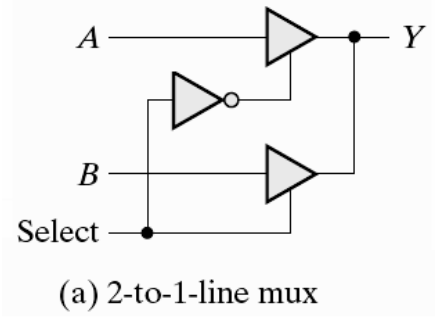
Three-State Gates

- A three-state gate is a digital circuit that exhibits three states: 0, 1, and high-impedance.
- The high-impedance state:
  - 1. behaves like an open circuit, which means the disconnected output.
  - 2. has no logic significance:
    - high-impedance connects with 0 equals 0
    - high-impedance connects with 1 equals 1
  - 3. is independent of the input states.
- The three-state gates help the shared bus structure of the digital systems.

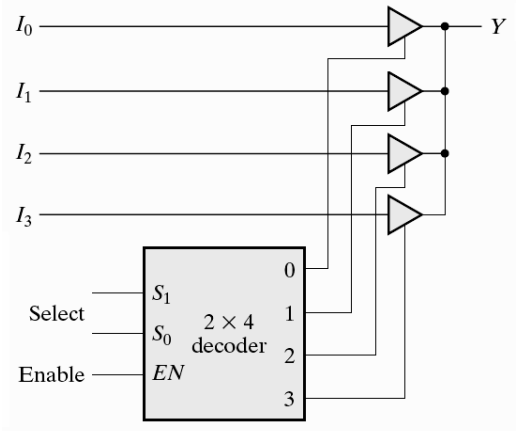


Multiplexers with Three-State Gates

- The three state gate can be any conventional logic, such as AND or NAND.
- The outputs of three-state gates are connected together to form a single output line.
- No more than one buffer may be in the active state at any given time.



(a) 2-to-1-line mux



(b) 4-to-1-line mux

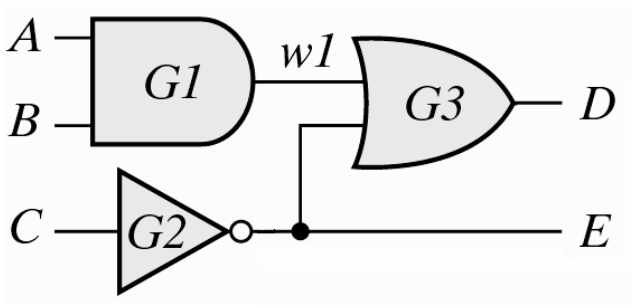
HDL for Combinational Circuit

- A combinational modules can be described in any one of the following modeling techniques.
  - Gate-level modeling using instantiation of primitive gates and user-defined modules.
  - Dataflow modeling using continuous assignment statements with keyword **assign**.
  - Behavioral modeling using procedural statements with keyword **always.**

Gate Level Modeling – Primitive Gates

- **and / nand / or / nor / xor / xnor / not / buf** are the **logic primitive gate** defined by verilog HDL.
- Basic form : **and** n1(output,…,input,…)

```
// Description of simple circuit Fig. 3-37
module Simple_Circuit(A, B, C, D, E);
input A, B, C;
output D, E;
wire w1;
and G1(w1, A, B);
not G2(E, C);
or G3(D, w1, E);
endmodule
```



Gate Level Modeling – Primitive Gates

- Truth table for 4 predefined logic primitive gates (**and**, **or**, **xor**, and **not**).

x: assigned when inputs or outputs are ambiguous.
z: high impedance, output of the three-state gates that are not enabled.

Table 4.9
Truth Table for Predefined Primitive Gates

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

not	input	output
	0	1
	1	0
	x	x
	z	x

Gate-level Modeling

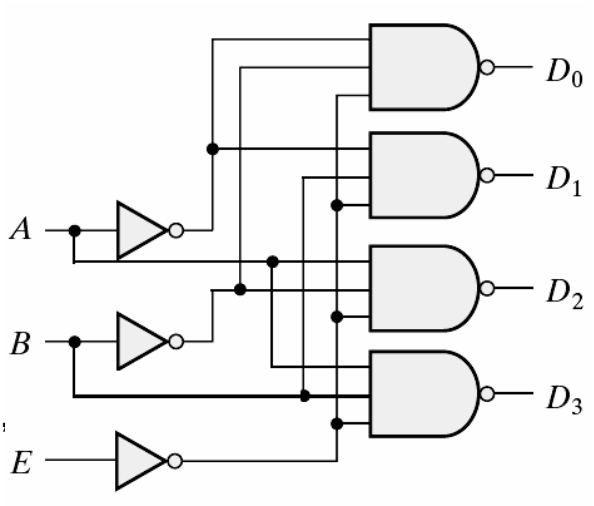
Example:

```
output  [0: 3]  D;
wire    [7: 0]  SUM;
```

1. The first statement declares an output vector D with four bits, 0 through 3.
2. The second declares a wire vector SUM with eight bits numbered 7 through 0.

Gate-Level Modeling – Example

```
// Gate-level description of 2-to-4 decoder
module decoder_g1(D,A,B,E);
input A, B, E;
output [0: 3] D;
wire A_not, B_not, E_not;
not
        G1(A_not, A),
        G2(B_not, B),
        G3(E_not, E);
nand
        G4(D[0], A_not, B_not, E_not),
        G5(D[1], A_not, B, E_not),
        G6(D[2], A, B_not, E_not),
        G7(D[3], A, B, E_not);
endmodule
```



Gate-Level Modeling – Example

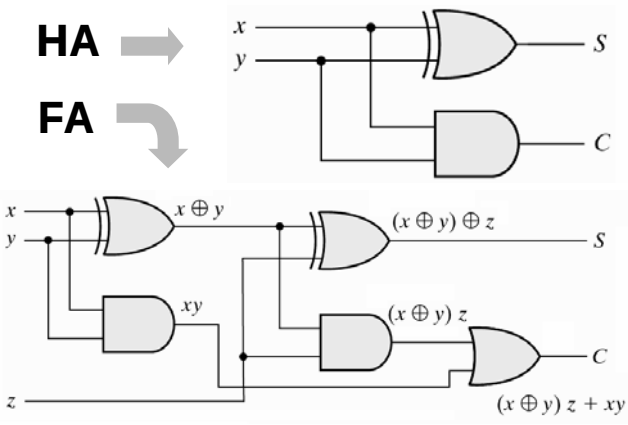
- Hierarchical description of coding of a 4-bit adder

```
/* Gate-level description of 4-bit ripple-carry
adder */

// Description of half adder
// module half_adder (S, C, x, y);
//  output          S, C;
//  input           x, y;

// Alternative Verilog 2005 syntax:
module half_adder (output S, C, input x,
y);
 // Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
endmodule

// Description of full adder
// module full_adder (S, C, x, y, z);
//  output          S, C;
//  input           x, y, z;
// alternative Verilog 2005 syntax:
```

```
module full_adder (output S, C, input x, y,
z);
  wire     S1, C1, C2;

// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or  G1 (C, C2, C1);
endmodule
```

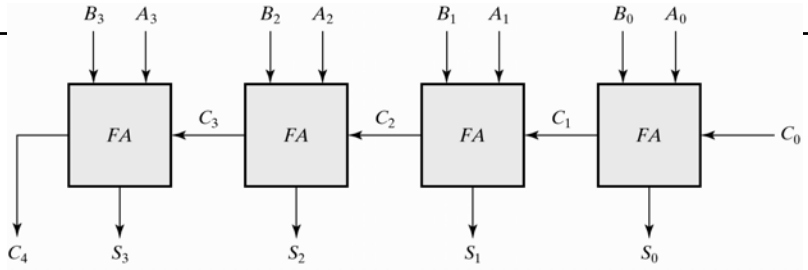
HA

FA



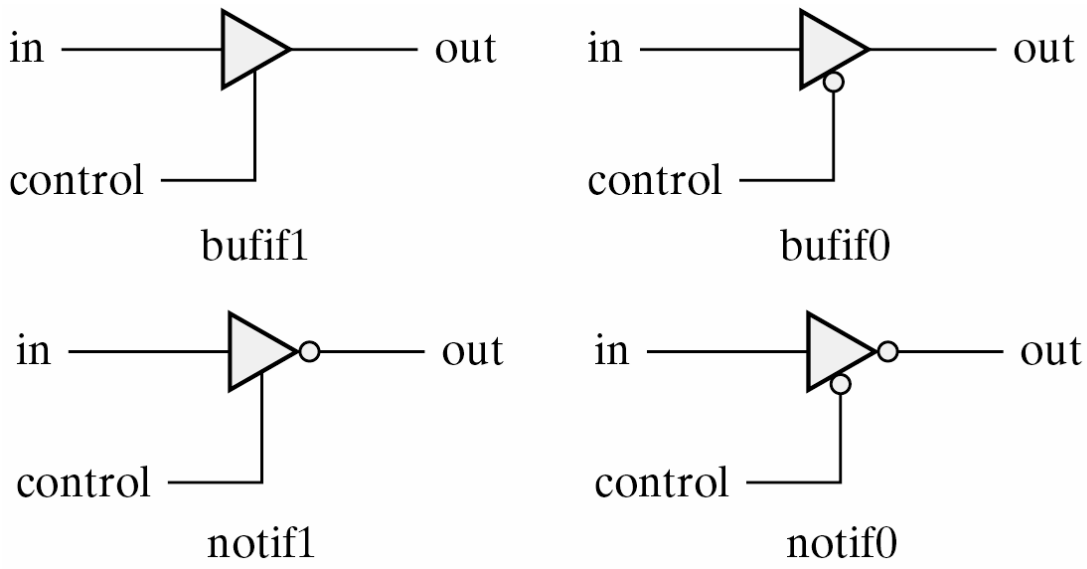
Gate-Level Modeling – Example

```
// Description of 4-bit adder (see Fig 4-9)
// module ripple_carry_4_bit_adder ( Sum, C4, A, B, C0);
//  output   [3: 0]  Sum;
//  output           C4;
//  input    [3: 0]  A, B;
//  input            C0;

// Alternative Verilog 2005 syntax:
module ripple_carry_4_bit_adder (output [3: 0] Sum, output C4, input [3:0]  A, B,
input C0);
  wire               C1, C2, C3;          // Intermediate carries
// Instantiate chain of full adders
  full_adder         FA0 (Sum[0], C1, A[0], B[0], C0),
                     FA1 (Sum[1], C2, A[1], B[1], C1),
                     FA2 (Sum[2], C3, A[2], B[2], C2),
                     FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```



Gate Level Modeling – Three State Gates

- **bufif1 / bufif0 / notif1 / notif0** are four basic **three-state gates** defined by Verilog HDL.
- Basic form : **bufxxx** n1(output, input, control);



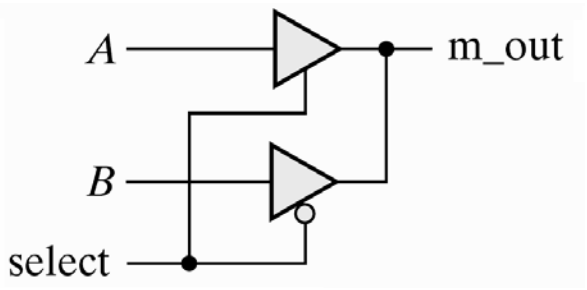
Three-State Gates

- Examples of gate instantiation

```
bufif1 (OUT, A, control);
notif0 (Y, B, enable);
```

- The HDL description of a **tri** data type for the output:

```
// Mux with three-state output
module mux_tri (m_out, A, B, select);
output          m_out;
input           A, B, select;
tri             m_out;
bufif1 n1(m_out, A, select);
bufif0 n2(m_out, B, select);
endmodule
```



Dataflow Modeling

- Data flow modeling using symbols to design the logic function:

Example:

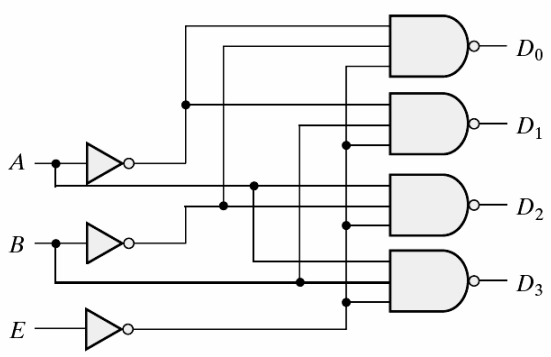
assign Y = (A & S) | (B & ~S)

Operator Type	Symbol	Operation Performed
Arithmetic	+	addition
	-	subtraction
	*	multiplication
	/	division
	%	modulus
Logic (Bit-wise)	~	negation (complement)
	&	AND
	|	OR
	^	Exclusive-OR
Logical	!	Negation
	&&	AND
	||	OR
Shift	>>	
	<<	
	{,}	concatenation
Relational	>	greater than
	<	less than
	==	equality
	!=	inequality
	>=	greater than or equal
	<=	less than or equal

HDL Example 4.3

- Dataflow description of a 2-to-4-line decoder

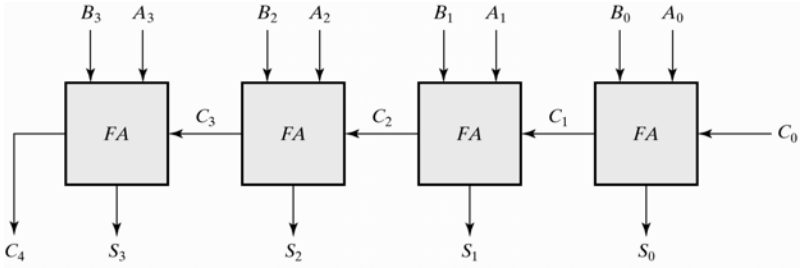
```
// Dataflow description of 2-to-4-line decoder
// See Fig. 4-19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (D, A, B, enable);
  output   [0: 3]          D;
  input                    A, B, enable;

  assign   D[0] = ~(~A & ~B & ~enable),
           D[1] = ~(~A & B & ~enable),
           D[2] = ~(A & ~B & ~enable),
           D[3] = ~(A & B & ~enable);
endmodule
```



HDL Example 4-4

- Dataflow description of 4-bit adder

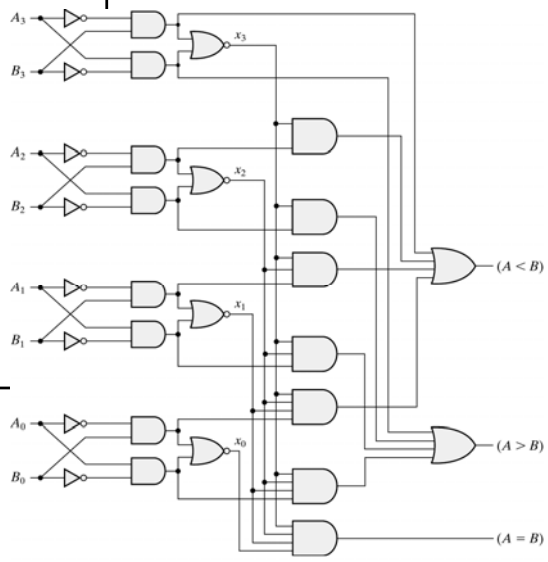
```
// Dataflow description of 4-bit adder
// Verilog 2005 module port syntax
module adder_4_bit_df (
  output [3: 0] Sum,
  output        C_out,
  input  [3: 0] A, B,
  input         C_in
);

  assign {C_out, Sum} = A + B + C_in;
endmodule
```



HDL Example 4-5

- Dataflow description of 4-bit magnitude comparator

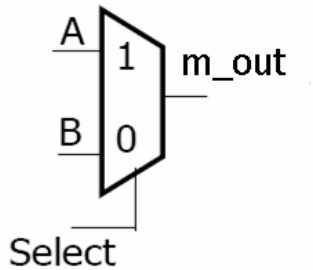
```
// Dataflow description of a 4-bit comparator

module mag_compare
( output        A_lt_B, A_eq_B, A_gt_B,
 input    [3: 0]  A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```



HDL Example 4-6

- The conditional operator functions (? :) as single-net or bus multiplexers.
- Basic form: **assign** OUT = (select) ? A : B;
- Dataflow description of a 2-to-1-line multiplexer

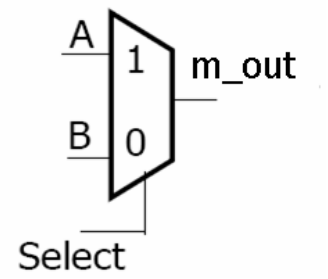
```
// Dataflow description of 2-to-1 line multiplexer
module mux_2x1_df(m_out, A, B, select);
  output        m_out;
  input         A, B;
  input         select;

  assign        m_out = (select)? A : B;
endmodule
```



Behavioral Modeling – always if

- Procedural assignment: **always@(**input event**)**
- Declaration **reg** of output
- Conditional description by **if**/**else**

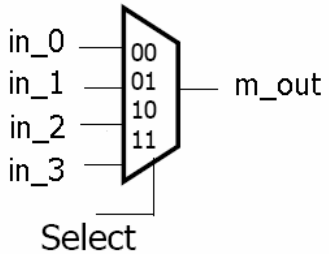
```
// Behavior description of 2-to-1 multiplexer

module MUX_2x1_beh(m_out, A, B, select);
  input A, B, select;
  output m_out;
  reg m_out;

  always @ (select or A or B)
    if (select == 1) m_out = A;
    else m_out = B;
endmodule
```



Behavior Modeling – always case

- Procedural assignment : **always@(**input event**)**
- Declaration **reg** of output
- Conditional description by **case**

```
// Behavioral description of 4-to-1 line multiplexer
// Verilog 2005 port syntax
module mux_4x1_beh
( output reg      m_out,
  input  in_0, in_1, in_2, in_3,
  input [1: 0]    select
);
  always @ (in_0, in_1, in_2, in_3, select)    // Verilog 2005 syntax
    case (select)
      2'b00:        m_out = in_0;
      2'b01:        m_out = in_1;
      2'b10:        m_out = in_2;
      2'b11:        m_out = in_3;
    endcase
  endmodule
```



Writing a Simple Test Bench

- **initial** block

```
initial
  begin
          A = 0; B = 0;
  #10     A = 1;
  #20     A = 0; B = 1;
  end
```

```
initial
  begin
    D = 3'b000;
    repeat (7)
      #10    D = D+ 3'b001;
  end
```

⟹ Three-bit truth table

Writing a Simple Test Bench

- Interaction between stimulus and design modules

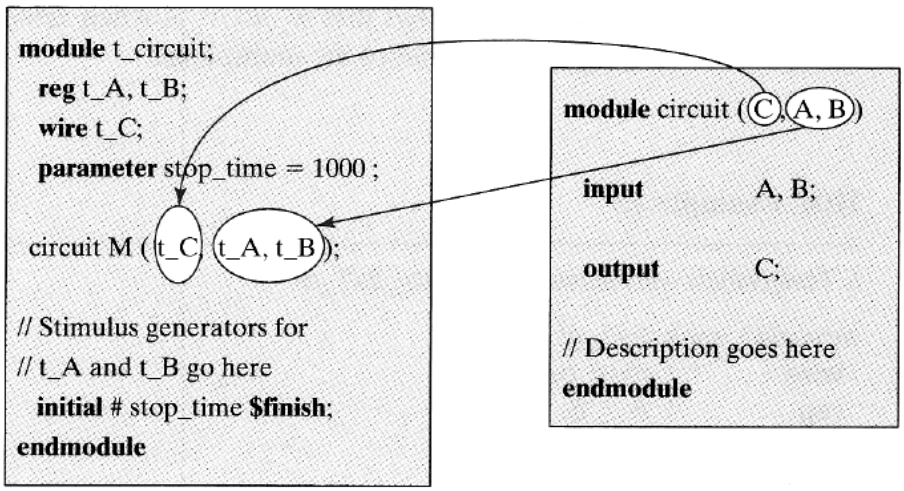


FIGURE 4.33
Interaction between stimulus and design modules

Writing a Simple Test Bench

- Stimulus module

```
module test_module_name;
  // Declare local reg and wire identifiers.
  // Instantiate the design module under test.
  // Specify a stopwatch, using $finish to terminate the simulation.
  // Generate stimulus, using initial and always statements.
  // Display the output response (text or graphics (or both)).
endmodule
```

- System tasks for display

$display – display a one-time value of variables or strings with an end-of line return.
$write – same as **$display**, but without going to next line.
$monitor – display variables whenever a value changes during a simulation run.
$finish – terminate the simulation.

Writing a Simple Test Bench

- Syntax for **$dispaly**, **$write**, and **$monitor**:

```
Task-name (format specification, argument list);
```

- Example:

```
$display ("%d %b %b", C, A, B);
```

- Example:

```
$display ("Time = %0d A = %b B = %b", $time, A, B);
```

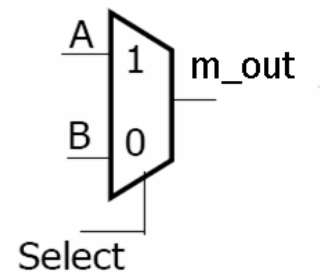
⟹

```
Time = 3  A= 10  B = 1
```

HDL Example 4-9

- Stimulus module

```
// Test bench with stimulus for mux_2x1_df
module t_mux_2x1_df;
  wire          t_m_out;
  reg           t_A, t_B;
  reg           t_select;
  parameter     stop_time = 50;

  mux_2x1_df M1 (t_m_out, t_A, t_B, t_select);  // Instantiation of circuit to be tested

  initial # stop_time $finish;

  initial begin                                  // Stimulus generator
          t_select = 1; t_A = 0; t_B = 1;
    #10   t_A = 1; t_B = 0;
    #10   t_select = 0;
    #10   t_A = 0; t_B = 1;
  end
```



HDL Example 4-9

```
initial begin                                  // Response Monitor
    // $display ("    time   select  A     B     m_out");
    // $monitor ($time, "   %b  %b  %b  %b", t_select, t_A, t_B, t_m_out);
    $monitor ("time = %0d", $time, "select = %b A = %b B = %b m_out =
     %b", t_select, t_A, t_B, t_m_out);
  end
endmodule

// Dataflow description of 2-to-1 line multiplexer  // from Example 4-6
module mux_2x1_df (m_out, A, B, select);
  output          m_out;
  input           A, B;
  input           select;
  assign  m_out = (select)? A : B;
 endmodule
```

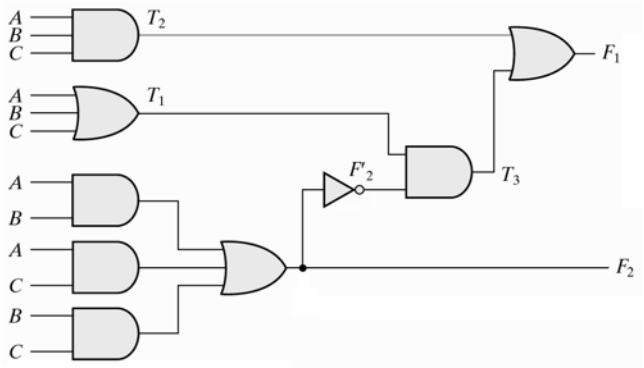
Simulation log:

```
time = 0 select = 1 A = 0 B = 1 m_out = 0
time = 10 select = 1 A = 1 B = 0 m_out = 1
time = 20 select = 0 A = 1 B = 0 m_out = 0
time = 30 select = 0 A = 0 B = 1 m_out = 1
```

HDL Example 4-10

- Gate-level description of a full adder

```
// Gate-level description of circuit in Fig. 4-2
module Circuit_of_Fig_4_2 (A, B, C, F1, F2);
  output          F1, F2;
  input           A, B, C;
  wire        T1, T2, T3, F2_not, E1, E2, E3;
  or          G1  (T1, A, B, C);
  and         G2  (T2, A, B, C);
  and         G3  (E1, A, B);
  and         G4  (E2, A, C);
  and         G5  (E3, B, C);
  or          G6  (F2, E1, E2, E3);
  not         G7  (F2_not, F2);
  and         G8  (T3, T1, F2_not);
  or          G9  (F1, T2, T3);
endmodule
//Stimulus to analyze the circuit
module test_circuit;
  reg         [2: 0] D;
  wire        F1, F2;
```



HDL Example 4-10

```
  Circuit_of_Fig_4_2 M_F4_32 (D[2], D[1], D[0], F1, F2);
  initial
    begin                                  // Stimulus generator
      D = 3'b000;
      repeat (7) #10 D = D + 1'b1;
    end
  initial
  $monitor ("ABC = %b  F1 = %b  F2 = %b", D, F1, F2);
endmodule

Simulation log:
/*        A     B     C     F1    F2
          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          */
```