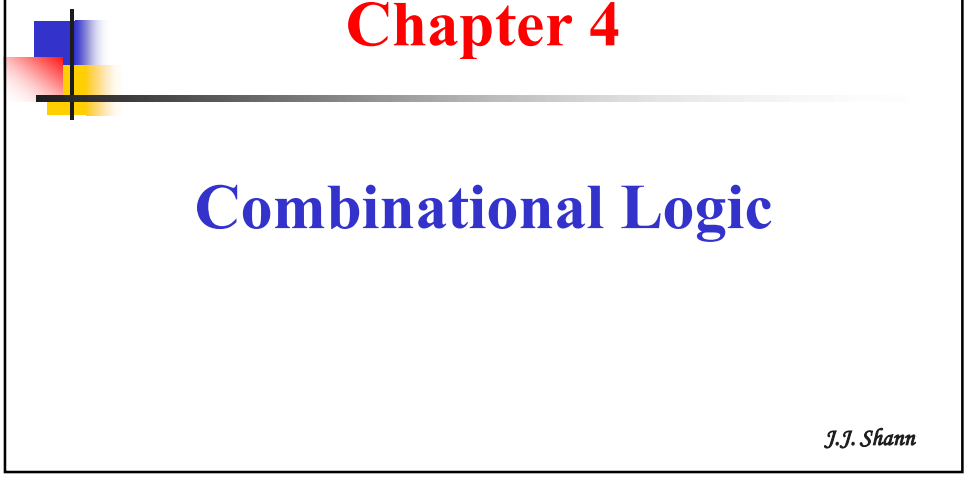


Chapter 4

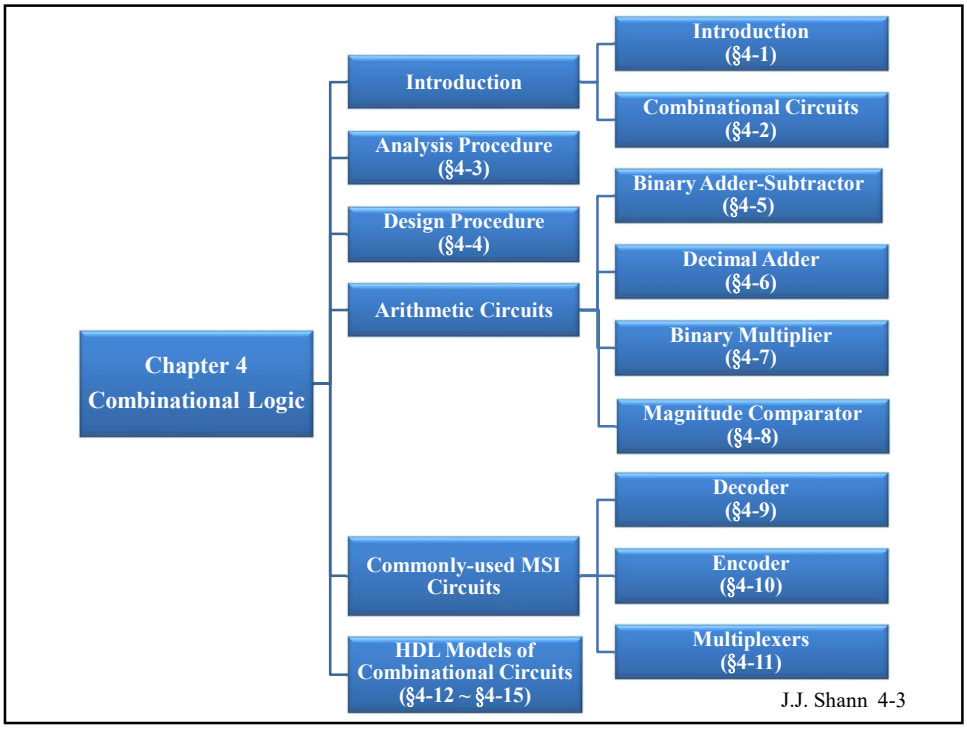
Combinational Logic

J.J. Shann

1

Contents

2



3

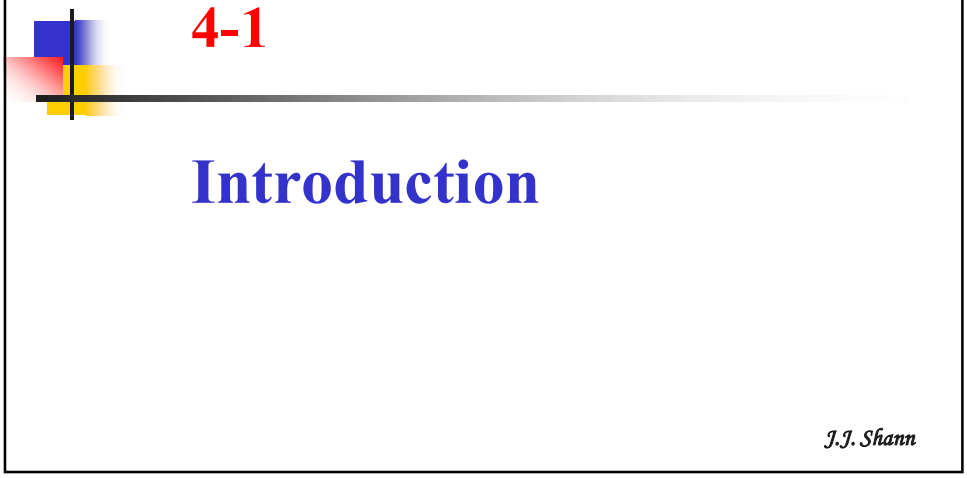
Exercises in Textbook (6th ed)

Sections	Exercises	Typical Ones
§4-3	4.1~4.3	4.1(a)*(b)
§4-4	4.4~4.10	4.5
§4-5	3.29, 4.11~ 4.17, 4.21	4.12, 4.15
§4-6	4.18, 4.19	4.18
§4-7	4.20	4.20(a)
§4-9	4.22~4.28	4.25, 4.28 (a)
§4-10	4.29, 4.30	4.29*
§4-11	4.31~4.35	4.32(a), 4.35(a)*
HDL	4.6(b), 4.36~4.65	

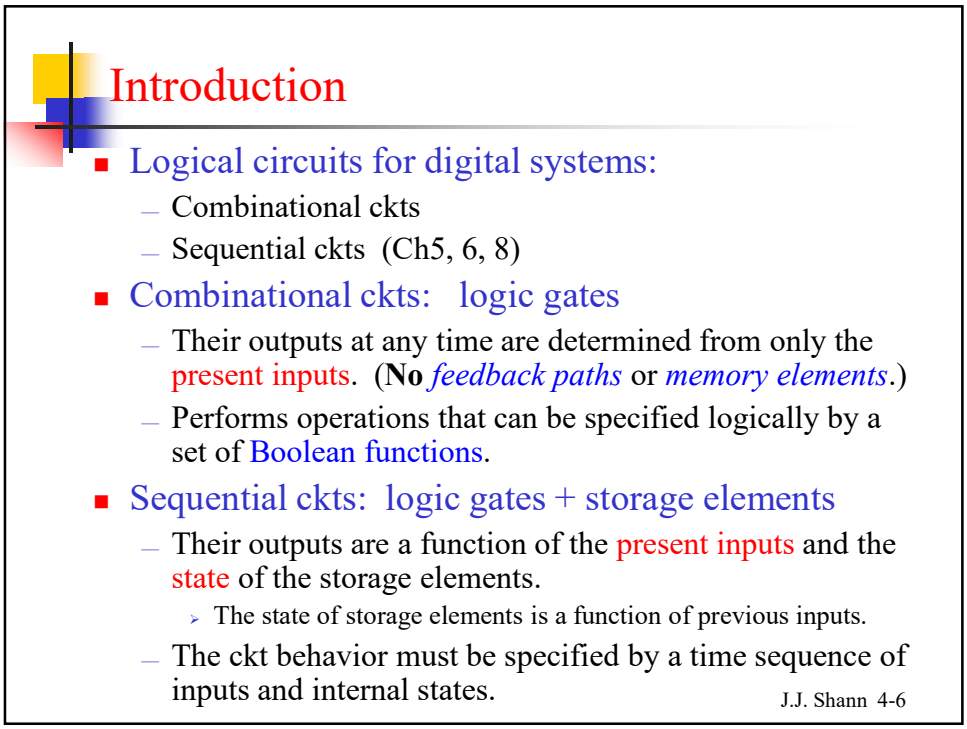
* : Answers to problems appear at the end of the text.

J.J. Shann 4-4

4


4-1
Introduction
J.J. Shann

5


Introduction
Logical circuits for digital systems:
Combinational ckts
Sequential ckts (Ch5, 6, 8)
Combinational ckts: logic gates
Their outputs at any time are determined from only the present inputs. (No feedback paths or memory elements.)
Performs operations that can be specified logically by a set of Boolean functions.
Sequential ckts: logic gates + storage elements
Their outputs are a function of the present inputs and the state of the storage elements.
The state of storage elements is a function of previous inputs.
The ckt behavior must be specified by a time sequence of inputs and internal states.
J.J. Shann 4-6

6

4-2

Combinational Circuits

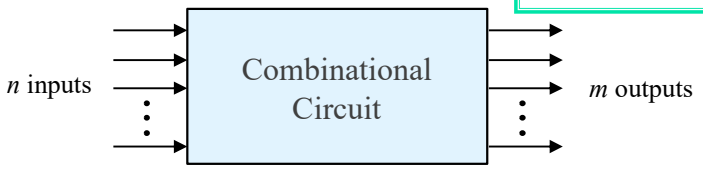
J.J. Shann

7

Combinational Circuits

- Combinational ckt: logic gates

n input variables
→ 2^n possible binary input combinations



- It outputs at any time are determined from the present inputs. (no feedback paths or memory elements)
- can be specified with
  i. a truth table:
  - lists the output values for each combination of input variables

  ii. *m* Boolean functions, one for each output variable
  - Each output function is expressed in terms of the input variables.

J.J. Shann 4-8

8

4-3

Analysis Procedure

J.J. Shann

9

Analysis Procedure

- Analysis:

  Logic diagram → Output Boolean functions,
  a truth table, or
  a verbal explanation of the ckt operation

  * Make sure that the given ckt is combinational. (not seq.)

J.J. Shann 4-10

10

Logic diagram → Output Boolean functions

- Procedure:
  1. Label all gate outputs that are a function of input variables w/ arbitrary symbols.

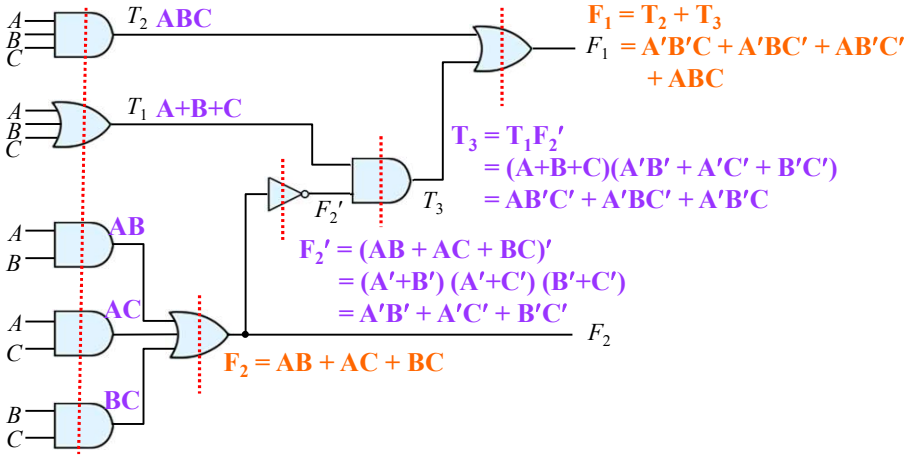
     Determine the Boolean function for each gate output.
  2. Label the gates that are a function of input variables and previously labeled gates w/ other arbitrary symbols.

     Find the Boolean functions for these gates.
  3. Repeat step 2 until the outputs of the ckt are obtained in terms of input variables.
  4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example

- Analyze the following logic diagram:

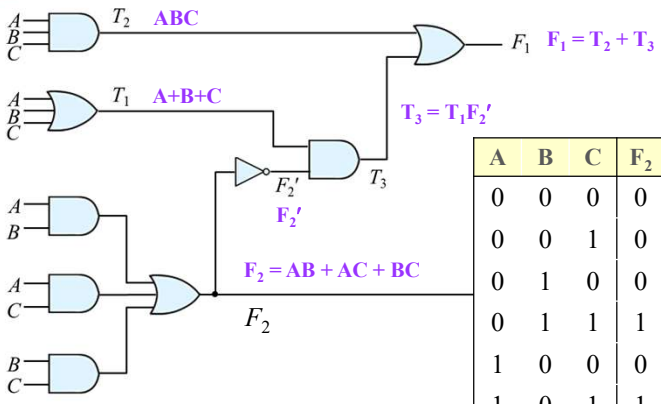
3 inputs: A, B, C; 2 outputs: F_1, F_2



Logic diagram → Truth table

- Procedure:
  - Logic diagram → Output Boolean functions
    → Truth table
  - Logic diagram → Truth table
    1. Determine the # of input variables in the ckt.
       For n inputs, form the 2^n possible input combinations and list the binary numbers from 0 to $2^n - 1$ in a table.
    2. Label the outputs of selected gates w/ arbitrary symbols.
    3. Obtain the truth table for the outputs of those gates that are a function of the input variables only.
    4. Proceed to obtain the truth table for the outputs of those gates that are a function of previously defined values until the columns for all outputs are determined.

Example



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

4-4

Design Procedure

J.J. Shann

15

Design Procedure

- Design:

  Specification of the problem

  → Logic ckt diagram or

  a set of Boolean functions from which the logic diagram can be obtained

- Procedure:

  1. From the specifications of the ckt, determine the required # of inputs and outputs and assign a symbol to each. (Specification)
  2. Derive the truth table or initial Boolean equations that defines the required relationship b/t inputs and outputs. (Formulation)
  3. Obtain the simplified Boolean functions for each output as a function of the input variables. (Optimization: *2-level* or *multi-level*)
  4. Draw the logic diagram. (Technology mapping)
  5. Verify the correctness of the design. (Verification = Analysis)

16

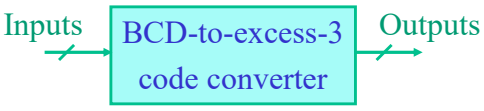
- Constraints considered for a practical design:
  - # of gates
  - # of inputs to a gate (fan in)
  - propagation time of the signal through the gates
  - # of interconnections
  - criteria of ICs :
    - E.g.: limitations of the driving capability of each gate (fan out)
- Elementary objective:
  - produce the simplified Boolean functions in a standard form (SoP/PoS).
  - proceed w/ further steps to meet other performance criteria.

J.J. Shann 4-17

17

Example: Code Conversion

- Convert the ***binary coded decimal*** (***BCD***) to the ***excess-3 code*** for the decimal digitals.

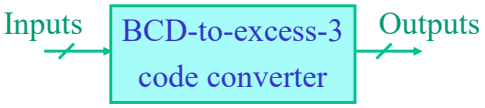


Decimal digit	BCD	Excess-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100
	1010	0000
	1011	0001
Unused bit	1100	0010
Combinations	1101	1101
	1110	1110
	1111	1111

J.J. Shann 4-18

18

<Ans.>



Step1: Specification

input variables — 4; A, B, C, D

output variables — 4; *w*, *x*, *y*, *z*

Step2: Formulation

Truth table

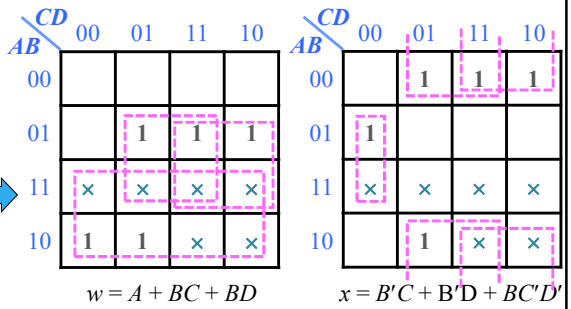
Input BCD				Output Excess-3 Code			
A	***B***	***C***	***D***	***w***	***x***	***y***	***z***
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

4-19

19

Step3: Simplification

BCD				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



* Unused input combinations: 1010 ~ 1111
* Assumption: treat as "×".

J.J. Shann 4-20

20

Step4: Draw the logic diagram

$w = A + BC + BD \Rightarrow A + B(C + D)$

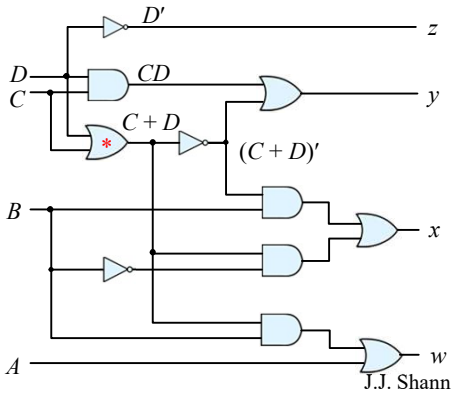
$x = B'C + B'D + BC'D' \Rightarrow B'(C + D) + BC'D' = B'(C + D) + B(C + D)'$

$y = CD + C'D' \Rightarrow CD + (C + D)'$

$z = D' \Rightarrow D'$

multiple-level ckt
GIC = 17 + 2

2-level ckt
GIC = 23 + 3 = 26

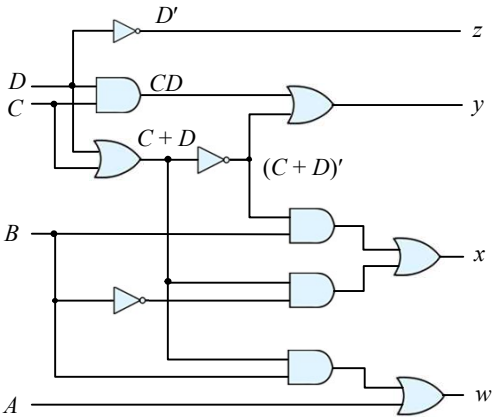


J.J. Shann 4-21

21

Step5: Verify

Logic diagram → Truth table



Analysis

Truth Table

	BCD ABCD	Excess-3 wxyz
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100
	1010	1101
	1011	1110
	1100	1111
	1101	1000
	1110	1001
	1111	1010

Unused input combinations

* Determine whether or not a given ckt implements its specified function.

J.J. Shann 4-22

22

Contents

23

4-5

Binary Adder-Subtractor

J.J. Shann

24

Binary Adder-Subtractor

- Half adder: add 2 input bits
  - augend bit, addend bit
- Full adder: add 3 input bits
  - augend bit, addend bit, carry-in bit
- Binary ***ripple carry*** adder: add two *n*-bit binary numbers
  - *n*-bit augend, *n*-bit addend, carry-in bit
- ***Carry-lookahead*** adder
- Binary subtractor
- Binary adder-subtractor

A. Half Adder

- Half adder (HA): adds 2 bits

$$\begin{array}{rr} & x \\ + & y \\ \hline C & S \end{array}$$

<Design Procedure>

Step 1: Specification

The basic rule for binary addition:

0 + 0 = 0 0
0 + 1 = 0 1
1 + 0 = 0 1
1 + 1 = 1 0

x y C S

Input variables: 2; augend and addend bits; *x*, *y*

Output variables: 2; sum and carry bits; *S*, *C*

Step2: Truth table

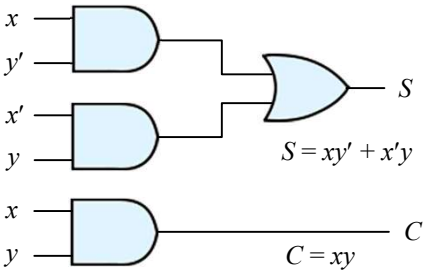
Inputs		Outputs	
x	*y*	*C*	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Step 3: Simplification

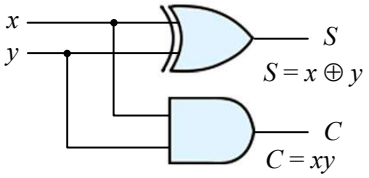
$S = x'y + xy' = x \oplus y$

$C = xy$

Step 4: Logic diagram



or



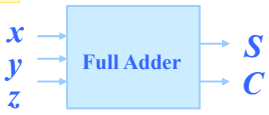
J.J. Shann 4-27

27

B. Full Adder

$$\begin{array}{cc} & z \\ & x \\ + & y \\ \hline C & S \end{array}$$

- Full adder (FA): add 3 bits



<Design Procedure>

Step 1: specification

Input variables: 3

2 significant bits *x*, *y* & a carry-in bit *z*

Output variables: 2

sum and carry bits *S*, *C*

Step 2: formulation

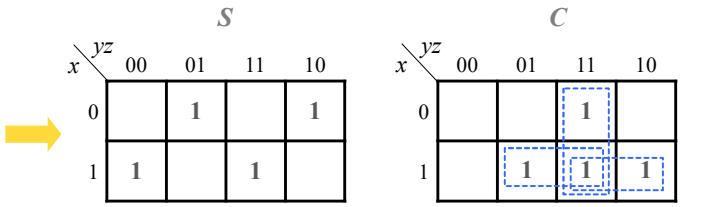
Inputs			Outputs	
x	*y*	*z*	*C*	*S*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

J.J. Shann 4-28

28

Step3: simplification

Inputs			Outputs	
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

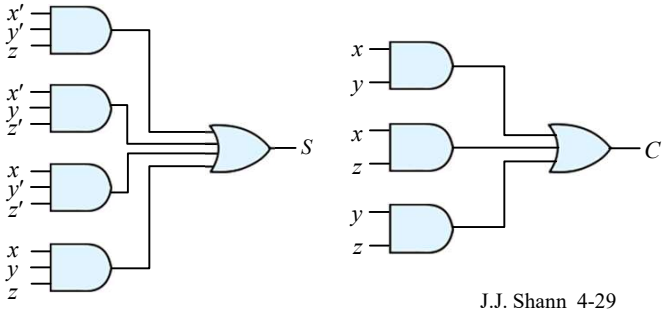


$S = x'y'z + x'yz' + xy'z' + xyz$

$C = xy + xz + yz$

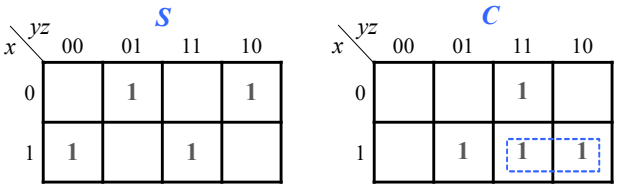
Step 4

2-level ckt



J.J. Shann 4-29

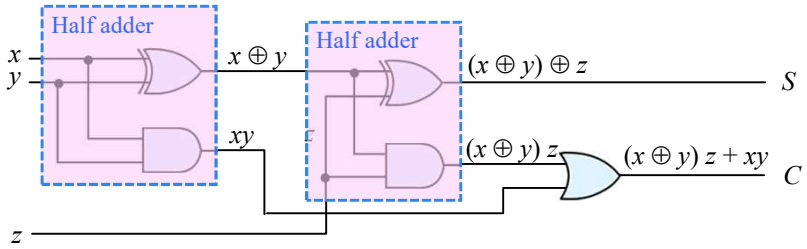
29



Alternatives of Step 3 & 4

S = x′y′z + x′yz′ + xy′z′ + xyz
= z′ (x′y + xy′) + z (xy + x′y′)
= z ⊕ (x′y + xy′)
= z ⊕ (x ⊕ y)

C = xy′z + x′yz + xyz′ + xyz
= z (xy′ + x′y) + xy (z′ + z)
= z (x ⊕ y) + xy

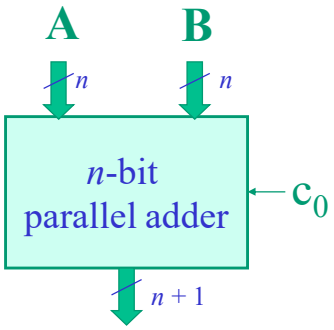


J.J. Shann 4-30

30

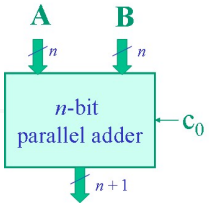
C. Binary Ripple Carry Adder

- Parallel adder:
  - a digital ckt that produces the arithmetic sum of two binary numbers using only combinational logic
  - Apply all input bits simultaneously to produce the sum
- Design method:
  - hierarchical design:
    - Top-down or Bottom-up
  - iterative design



31

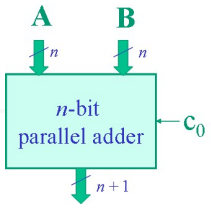
- Binary addition:
  - E.g.: 1011 + 0011



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}

- Truth table:
  9 inputs (2 4-bit numbers & a carry-in bit); 5 outputs
  ⇒ 512 entries (impractical)

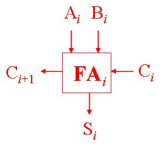
32

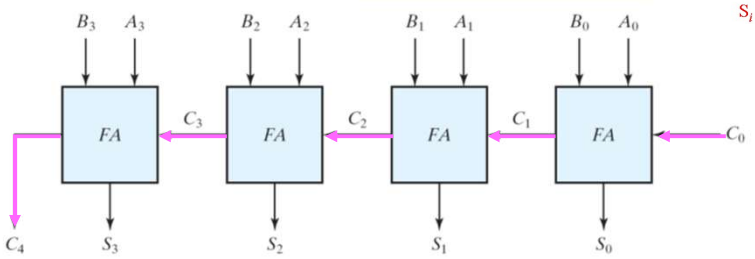


Implementation of binary *ripple carry* adder:

- uses n full adders (FAs) in parallel
  - The FAs are connected in cascade, w/ the ***carry output*** from one FA connected to the ***carry input*** of the next FA.
- E.g.: 4-bit adder

$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i)$$





* Hierarchy & Iterative design

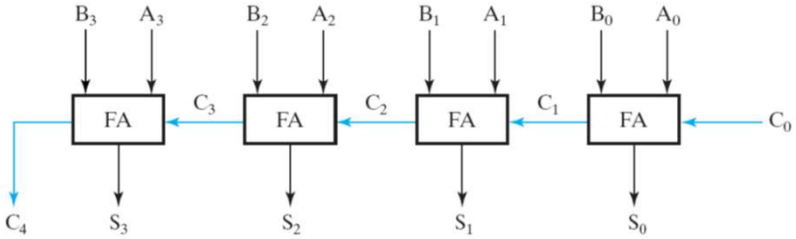
Carry Propagation

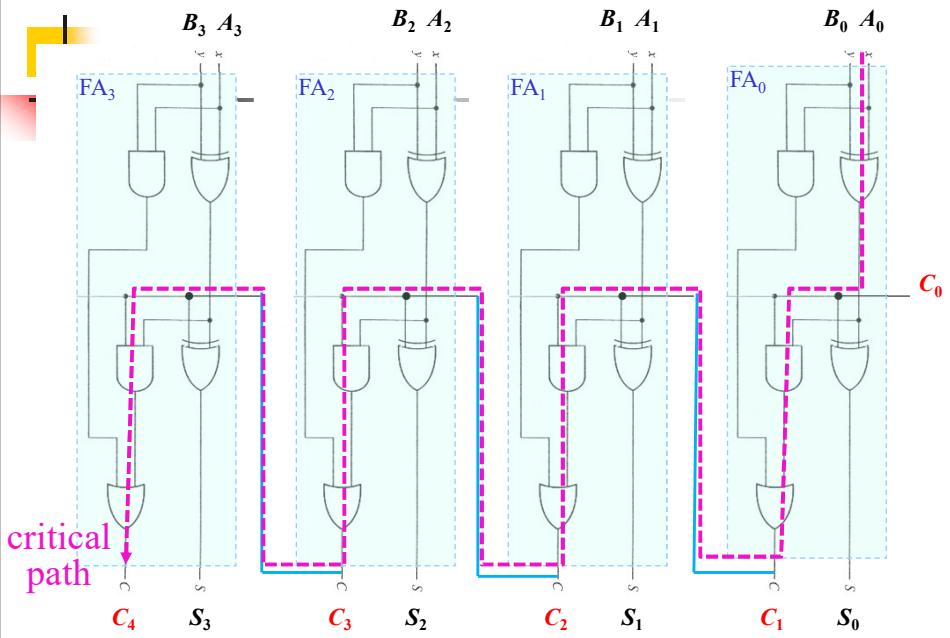
$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i)$$

- Propagation time of a combinational ckt:

  (the propagation delay of a typical gate) × (the # of gate levels in the ckt)

- Propagation time of the binary ripple carry adder:
  - the time it takes the carry to propagate through the full adders: **$2n + 2$** gate delays, n: # of bits of the input numbers
    - Assumption: delay of XOR gate = 2

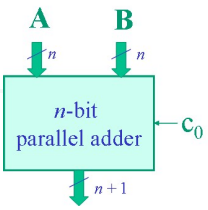




J.J. Shann 4-35

35

Methods of Speeding Addition



- Method 1:
  - employ faster gates w/ reduced delays
- Method 2:
  - increase the equipment complexity to reduce the carry delay time
  - Extreme case: transform the Boolean functions of the sum bits and carry bit into standard form

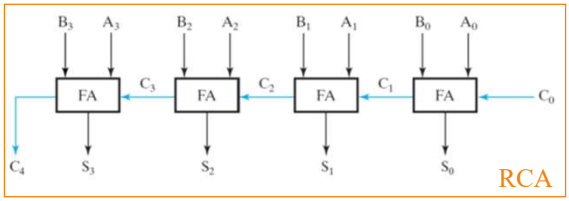
    (SoP or PoS → 2-level circuit theoretically)
  - most widely used technique: carry lookahead

J.J. Shann 4-36

36

D. Carry Lookahead Adder

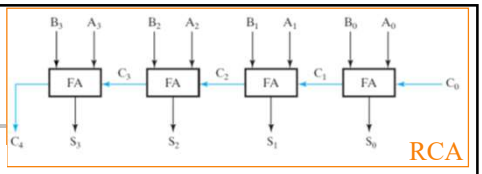
- Ripple carry adder (RCA):
  - simple
  - has a long ckt delay in the carry path from LSB to MSB



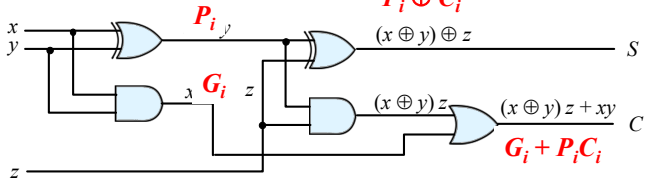
- Carry lookahead adder (CLA):
  - reduce delay at the price of more complex hardware

37

Carry Lookahead

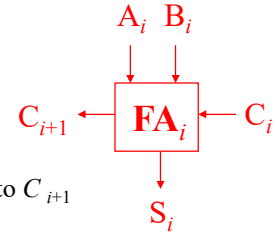


- Full adder:



$S_i = A_i \oplus B_i \oplus C_i$

$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i)$



- Carry lookahead:
  - Define 2 new binary variables:

    $P_i = A_i \oplus B_i$ … carry propagate function

    propagate the carry from C_i to C_{i+1}

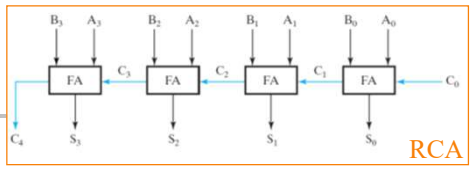
    $G_i = A_i B_i$ … carry generate function

    produce an output carry regardless of the input carry

    $\Rightarrow S_i = P_i \oplus C_i$

    $C_{i+1} = G_i + P_i C_i$

38



$C_{i+1} = G_i + P_i C_i$, C_0 = input carry

$\Downarrow$

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 C_1$

$= G_1 + P_1(G_0 + P_0 C_0)$

$= G_1 + P_1 G_0 + P_1 P_0 C_0$

$C_3 = G_2 + P_2 C_2$

$= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

$C_4 = G_3 + P_3 C_3$

$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$

$P_i = A_i \oplus B_i$

$G_i = A_i B_i$

- E.g.: 3-bit carry lookahead generator

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$

$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

$P_2 = A_2 \oplus B_2$

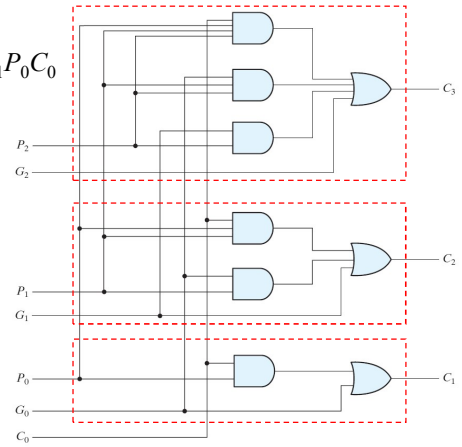
$G_2 = A_2 B_2$

$P_1 = A_1 \oplus B_1$

$G_1 = A_1 B_1$

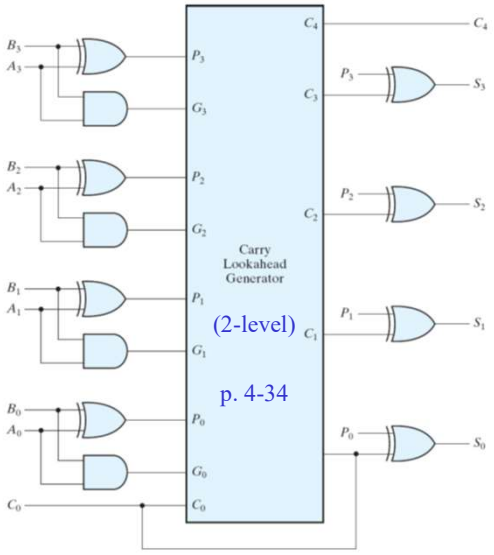
$P_0 = A_0 \oplus B_0$

$G_0 = A_0 B_0$



$P_i = A_i \oplus B_i$
$G_i = A_i B_i$

$C_1 = G_0 + P_0 C_0$
$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$
$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$



$C_4 = G_3 + P_3 C_3$

$S_i = P_i \oplus C_i$

* Propagation delay = 6

$A_i, B_i \xrightarrow{2} P_i, G_i \xrightarrow{2} C_i \xrightarrow{2} S_i$

41

E. Binary Subtractor

- Half subtractor: subtract 2 bits
- Full subtractor: subtract 2 input bits and a borrow-in bit
- Binary ripple borrow subtractor: subtract two *n*-bit binary numbers
- Borrow-lookahead subtractor

- Binary subtractor based on a parallel adder

42

Adder-subtractor

- Subtraction: (§1-5, §1-6)

  A – B = A + (the 2's-complement of B)

  = A + (the 1's-complement of B) + 1

- Adder-subtractor: based on a parallel adder

  Mode input M:

  M = 0 ⇒ A + B

  M = 1 ⇒ A + B′ + 1

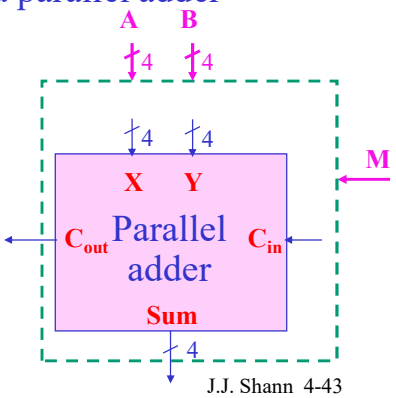
M	X	Y	C_{in}
0	A	B	0
1	A	B′	1

⇒ X = A

Y = M′B + MB′

= M⊕B

C_{in} = M



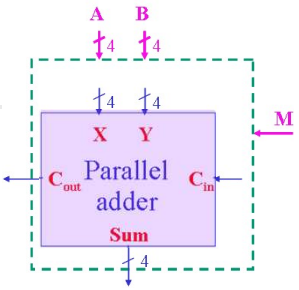
43

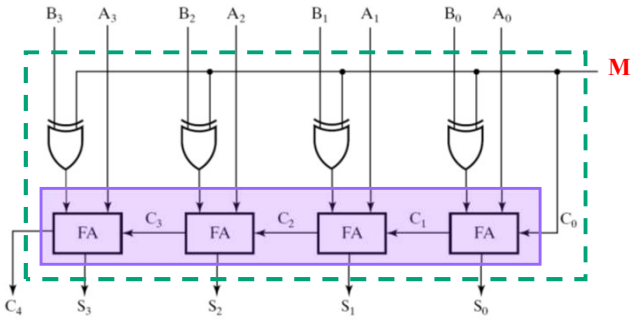
M	X	Y	C_{in}
0	A	B	0
1	A	B′	1

⇒ X = A

Y = M⊕B

C_{in} = M





44

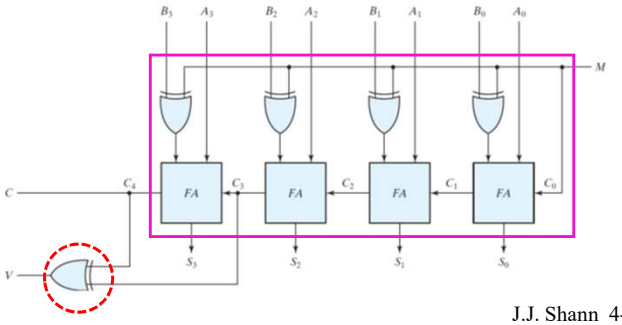
F. Overflow

- Overflow: signed or unsigned
  - When 2 numbers of n digits each are added & the sum occupies $n + 1$ digits
- Detection of an overflow:
  - For unsigned numbers: end carry out of the MSB
    - An overflow is detected from the end carry out of the MSB.
  - For signed numbers: carry into the sign bit ⊕ carry out of the sign bit
    - An overflow may occur if the 2 numbers added are both positive or both negative.
    - Examples: 8-bit numbers (+127 ~ −128)

carries:	0	1		carries:	1	0	
+70		0	1000110	−70		1	0111010
+80		0	1010000	−80		1	0110000
+150		1	0010110	−150		0	1101010

Adder-subtractor ckt w/ overflow detection:

- For unsigned numbers: $C = C_4$
  - detects a carry after addition or a borrow after subtraction
- For signed numbers: $V = C_3 \oplus C_4$
  - detects an overflow: the $(n+1)$th bit is the actual sign, but it cannot occupy the sign bit position in the result.



4-6

Decimal Adder

J.J. Shann

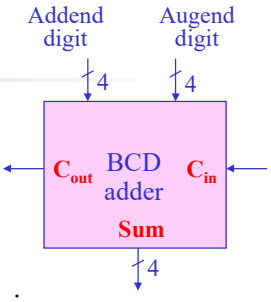
47

Decimal Adder

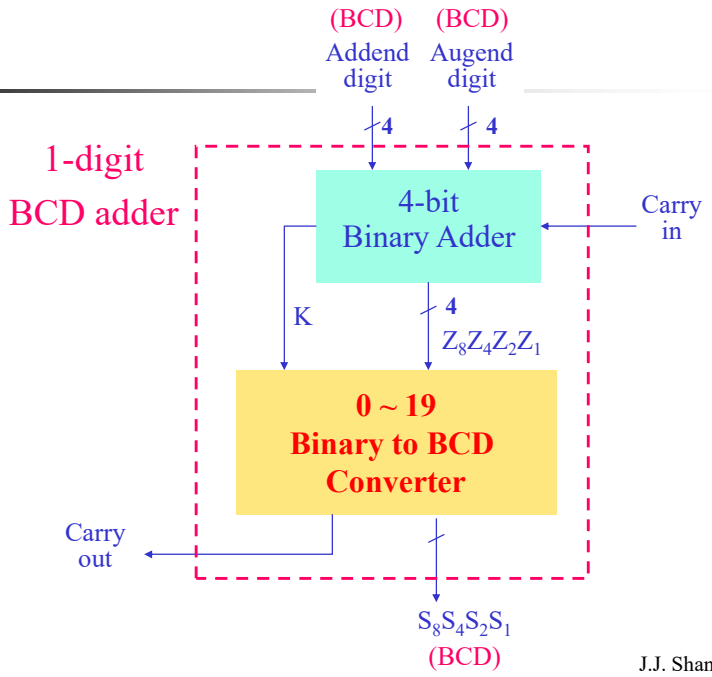
- Decimal adder:
  - Employ arithmetic ckts that accept coded decimal numbers and present results in the same code.
- Design methods:
  - Classic method
  - Add the numbers w/ binary adders

48

BCD Adder

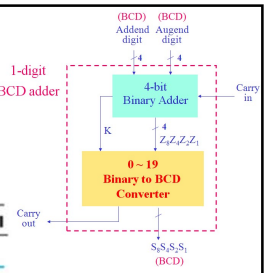


- BCD adder:
  - Add two BCD digits in parallel & produces a sum digits also in BCD and a carry-out bit
  - 9 inputs: two BCD digits and one carry-in
  - 5 outputs: one BCD digit and one carry-out
- Design approaches
  - i. Classic method: a truth table with 2^9 entries
  - ii. Add the numbers w/ binary adders
    - the sum $\leq 9 + 9 + 1 = 19$
    - Convert binary results to BCD



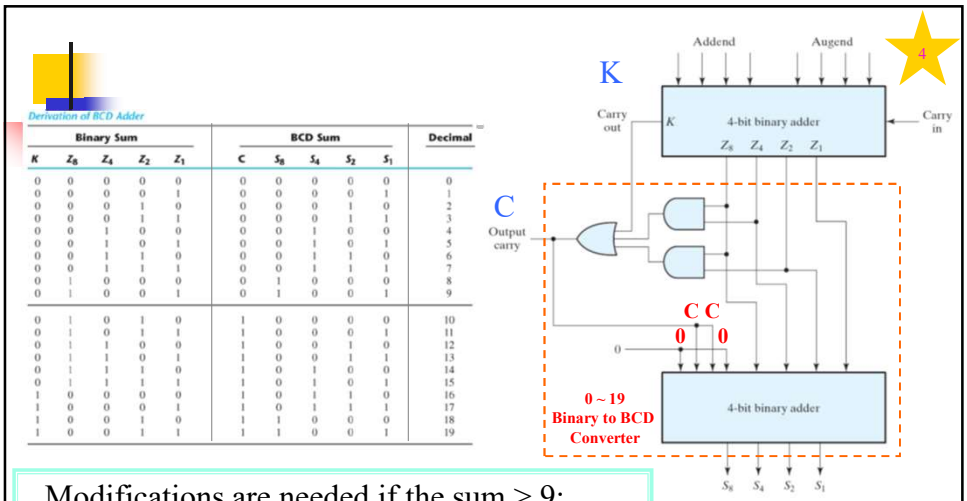
BCD Adder w/ Binary Adders

- Binary results to BCD:



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



Modifications are needed if the sum > 9:

$C = K + Z_8Z_4 + Z_8Z_2$

$S_8S_4S_2S_1 \leftarrow Z_8Z_4Z_2Z_1 + 0000$ when C = 0

$\leftarrow Z_8Z_4Z_2Z_1 + 0110$ when C = 1

$S_8S_4S_2S_1 \leftarrow Z_8Z_4Z_2Z_1 + 0CC0$

4-7

Binary Multiplication

J.J. Shann

53

Binary Multiplication

- Binary multiplication:
  - The multiplicand is multiplied by each bit of the multiplier, starting from the LSB.

    Each such multiplication forms a partial product.

    Successive partial products are shifted one bit to the left.

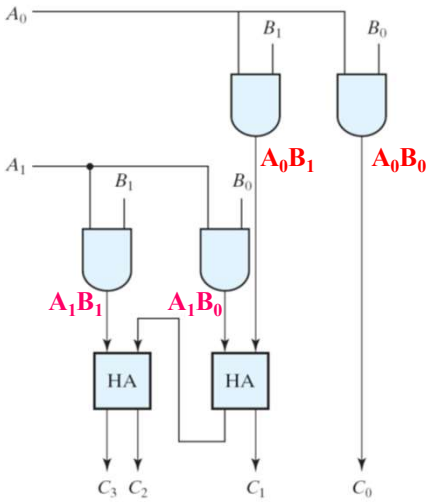
    The final product is obtained from the sum of the partial products.

  - E.g.: multiplication of two 2-bit numbers $B_1B_0 \times A_1A_0$

54

- Example: A 2-bit by 2-bit binary multiplier

		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0

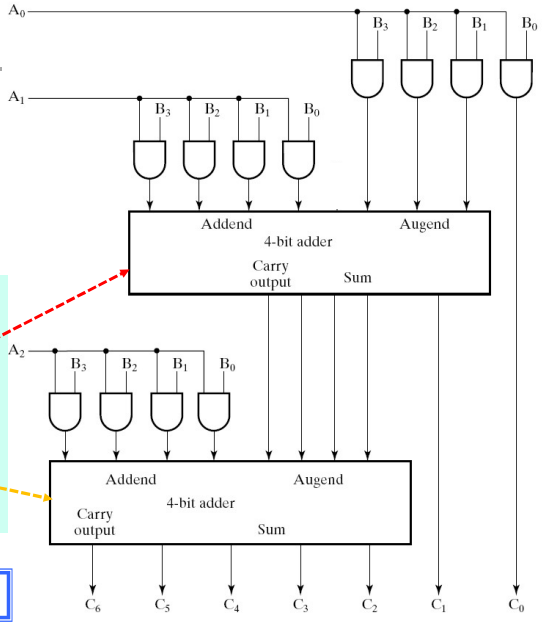


J.J. Shann 4-55

55

- Example:
  A 4-bit by 3-bit binary multiplier

	B_3	B_2	B_1	B_0
		A_2	A_1	A_0
0	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	
C_6	C_5	C_4	C_3	C_2 C_1 C_0



* 12 AND gates & two 4-bit adders

J.J. Shann 4-56

56

Construction of Comb. Binary Multiplier

- General rule:
  - A bit of the multiplier is ANDed w/ each bit of the multiplicand in as many levels as there are bits in the multiplier.
  - The binary output in each level of AND gates is added w/ the product.

  ⇒ For J multiplier bits & K multiplicand bits:

  Need $(J \times K)$ AND gates & $(J - 1)$ K-bit adders

  to produce a product of $J + K$ bits.

  - E.g.: A 4-bit by 3-bit binary multiplier (p.4-56)
    - K = 4 & J = 3 ⇒ 12 AND gates & two 4-bit adders

57

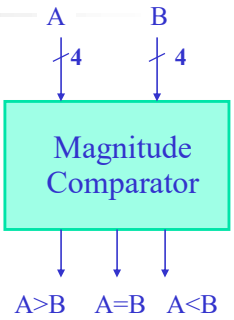
4-8

Magnitude Comparator

58

Magnitude Comparator

- Magnitude comparator:
  - Compares two numbers, A and B, and determines their relative magnitude:
    A > B, A = B, A < B

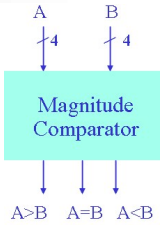


- Design Approaches
  - Classic method: truth table
    - Two n-bit numbers $\Rightarrow 2n$ input variables
      $\Rightarrow 2^{2n}$ entries … too cumbersome for large n
  - Use inherent regularity of the problem: algorithm
    - reduce design efforts
    - reduce human errors

Example: 4-bit Magnitude Comparator

- Algorithm → logic:

$A_3\,A_2\,A_1\,A_0$
$B_3\;B_2\,B_1\,B_0$



$A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$

$A = B$

- equality: $x_i = A_iB_i + A_i'B_i' = (A_i'B_i + A_iB_i')'$ for $i = 0, 1, 2, 3$

$\Rightarrow$ (A=B) $= x_3\,x_2\,x_1\,x_0$

(A>B) $= A_3\,B_3' + x_3\,A_2\,B_2' + x_3\,x_2\,A_1\,B_1' + x_3\,x_2\,x_1\,A_0\,B_0'$

(A<B) $= A_3'\,B_3 + x_3\,A_2'\,B_2 + x_3\,x_2\,A_1'\,B_1 + x_3\,x_2\,x_1\,A_0'\,B_0$

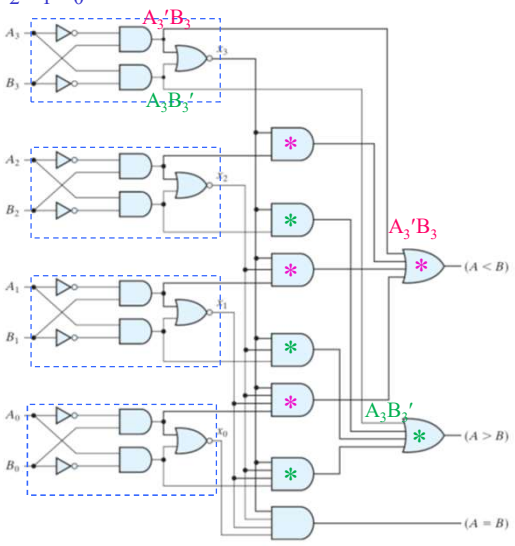
$A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$

$x_i = (A_i'B_i + A_iB_i')'$

$(A<B) = A_3' B_3$
$+ x_3A_2' B_2$
$+ x_3x_2A_1' B_1$
$+ x_3x_2x_1 A_0' B_0$

$(A>B) = A_3B_3'$
$+ x_3A_2B_2'$
$+ x_3x_2A_1B_1'$
$+ x_3x_2x_1 A_0B_0'$

$(A=B) = x_3x_2x_1x_0$



61

Contents

62

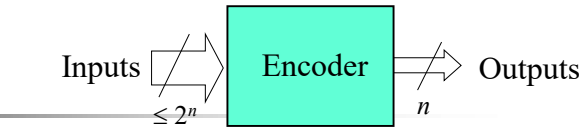
4-10

Encoders

J.J. Shann

63

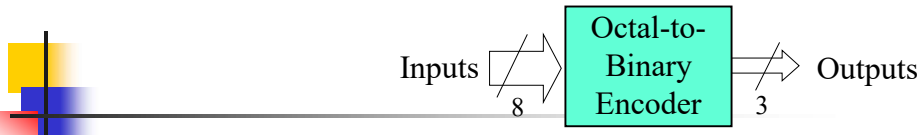
Encoders



- n-bit binary code:
  - is capable of representing up to 2^n distinct elements of coded information.
- Encoder: the inverse function of a decoder
  - Has 2^n (or fewer) input lines & n output lines.
  - The output lines generate the binary code corresponding to the input value.
- Example: an octal-to-binary encoder

J.J. Shann 4-64

64



- Example: an octal-to-binary encoder

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

$$x = D_4 + D_5 + D_6 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$z = D_1 + D_3 + D_5 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$z = D_1 + D_3 + D_5 + D_7$$

- limitation of the encoder:

  One and only one input can be active at any given time

  - When more than one input are active simultaneously

    E.g.: illegal input $D_3 = D_6 = 1$

    the output = 111 (≡ the output when D_7 is equal to 1)

    ⇒ Solution: Priority encoder
  - When all the inputs are 0:

    the output is 000 (≡ the output when D_0 is equal to 1)

    ⇒ Solution: Valid-output indicator

    Provide one more output to indicate that at least one input is equal to 1

Priority Encoder

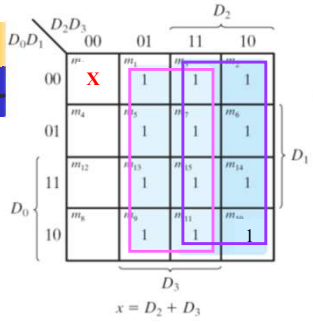
- Priority Encoder:
  - An encoder ckt that includes the priority function,
    i.e., if two or more inputs are equal to 1 at the same time,
    the input having the highest priority will take precedence.
- Example: 4-input priority encoder with valid-output indicator

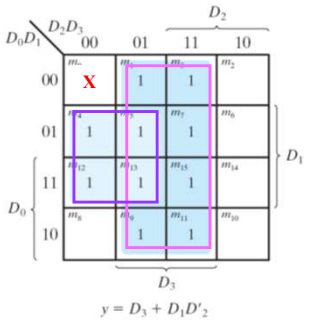
  Priority: $D_3 > D_2 > D_1 > D_0$, V: valid-output indicator

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

×: don't-care condition





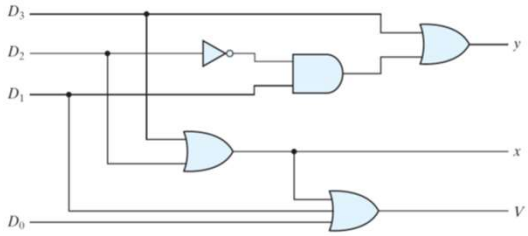
Truth Table of a Priority Encoder

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

$$x = D_2 + D_3$$

$$y = D_3 + D_1\overline{D}_2$$

$$V = D_0 + D_1 + D_2 + D_3$$



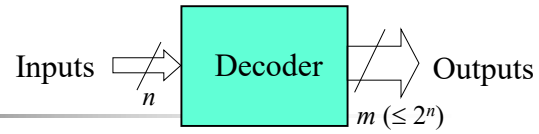
4-9

Decoders

J.J. Shann

69

Decoders



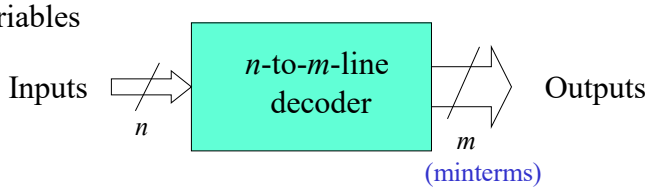
- *n*-bit binary code:
  - is capable of representing up to 2^n distinct elements of coded information.
- Decoding:
  - the conversion of an *n*-bit input code to an *m*-bit output code w/ $n \le m \le 2^n$ s.t. each valid input code word produces a unique output code.
- Decoder: the inverse function of a encoder
  - a combinational ckt w/ an *n*-bit binary code applied to its inputs and an *m*-bit binary code appearing at the output, i.e., converts binary information from *n* input lines to a maximum of 2^n unique output lines: $m \le 2^n$
  - may have unused bit combinations on it inputs for which no corresponding *m*-bit code appears at the output.

J.J. Shann 4-70

70

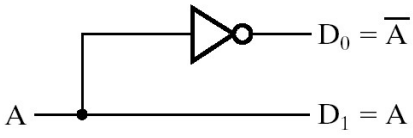
Line Decoder

- *n*-to-*m*-line decoder: $m \le 2^n$
  - generate the 2^n (or fewer) minterms/maxterms of *n* input variables



- E.g.: a 1-to-2-line decoder (outputs: active HIGH → minterms)

A	D_0	D_1
0	1	0
1	0	1



J.J. Shann 4-71

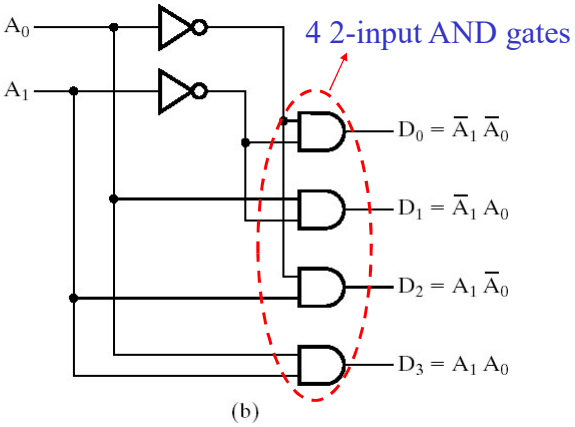
71

- E.g.: a 2-to-4-line decoder (outputs: active HIGH → minterms)

A_1	A_0	D_0	D_1	D_2	D_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

(a)

$D_0 = A_1'A_0' \rightarrow m_0$
$D_1 = A_1'A_0 \rightarrow m_1$
$D_2 = A_1A_0' \rightarrow m_2$
$D_3 = A_1A_0 \rightarrow m_3$

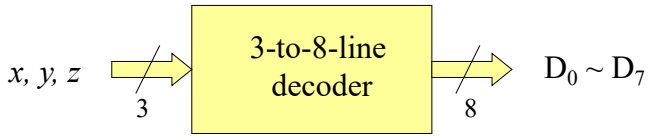


GIC = 2×4 + 2

J.J. Shann 4-72

72

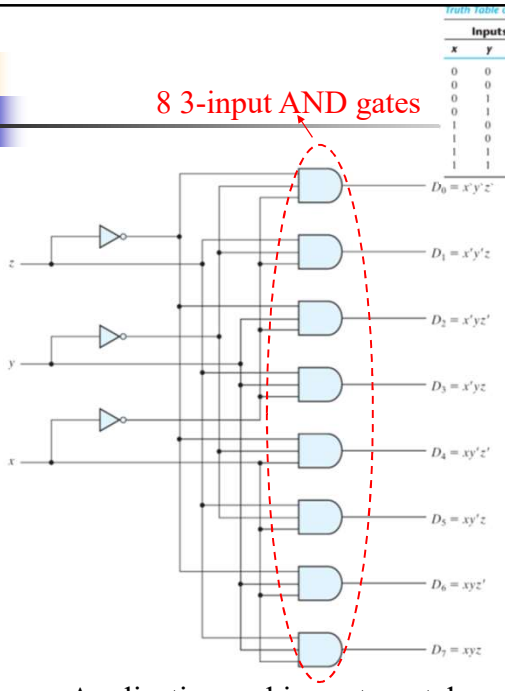
- E.g.: a 3-to-8-line decoder (output: active HIGH)



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

8 3-input AND gates

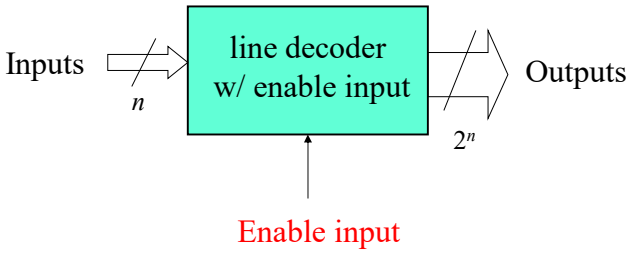


GIC = 3 × 8 + 3
= 27

Application: a binary-to-octal conversion

Line Decoder w/ Enable Input

- Line decoder w/ enable input



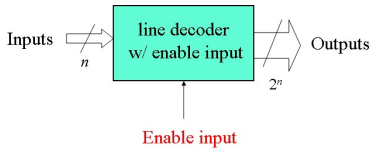
J.J. Shann 4-75

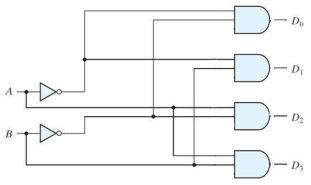
75

- E.g.: 2-to-4-line decoder w/ enable input

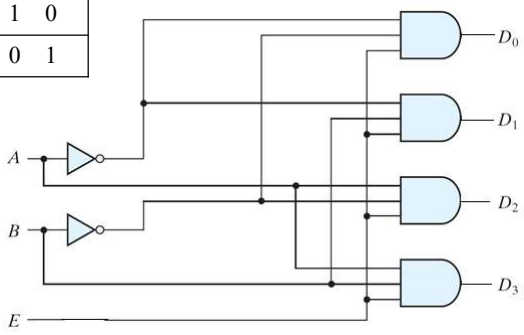
E & Outputs: active High

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





2-to-4 line decoder
(output active HIGH)



J.J. Shann 4-76

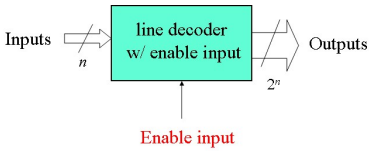
76

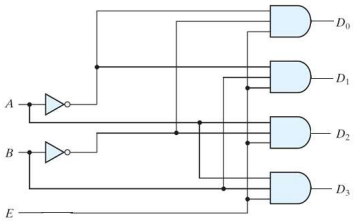
* Outputs: active LOW → maxterms

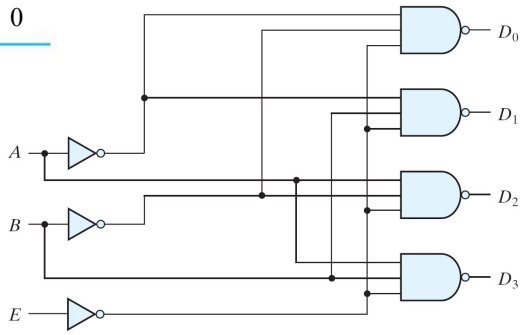
- E.g.: 2-to-4-line decoder w/ enable input

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

E & Outputs: active Low





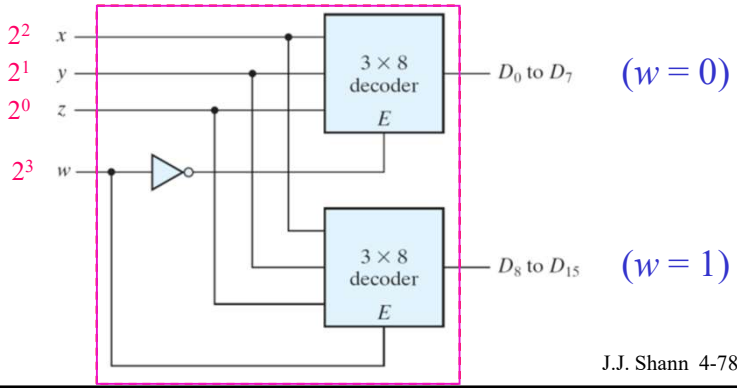


2-to-4 line decoder w/ enable input
(E & output active HIGH)

77

2^3 w	2^2 x	2^1 y	2^0 z	D_0	D_1	.....	D_{15}
0	0	0	0	1	0	.....	0
0	0	0	1	0	1	.....	0
	...					...	
1	1	1	1	0	0	.....	1

- Construction of larger decoder:
  - Decoders w/ enable inputs can be connected together to form a larger decoder ckt.
- E.g.: 2 3-to-8-line decoders ⇒ a 4-to-16-line decoder (w/ enable input)



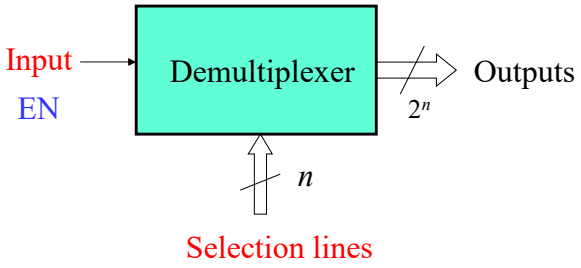
78

Demultiplexer

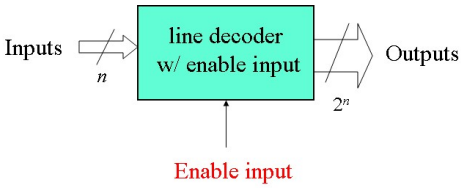
- Demultiplexer:
  - a ckt that receives information from a single input line and directs it to one of 2^n possible output lines according to the bit combination of n selection lines.

1-to-4 demux

A_1	A_0	D_0	D_1	D_2	D_3
0	0	EN	0	0	0
0	1	0	EN	0	0
1	0	0	0	EN	0
1	1	0	0	0	EN



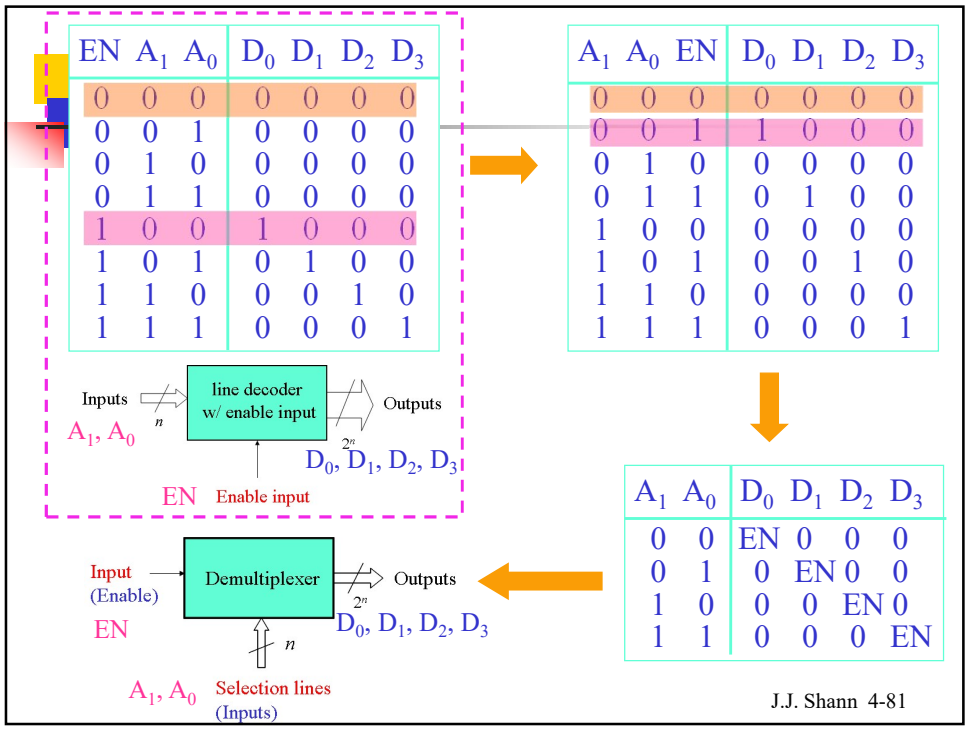
- Demultiplexer: ⇔ a line decoder w/ enable input
  - a line decoder w/ enable input:



* EN & Outputs are active HIGH.

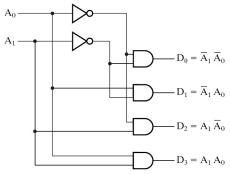
2-to-4 line decoder w/ enable line
(EN & outputs are active HIGH)

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



81

Comb. Logic Implementation



- Line-decoder w/ active-HIGH outputs:
  - provides the 2^n minterms of n input variables
    - each output = a minterm (active-HIGH outputs)
- Any Boolean function of n input variables
  - → Sum of minterms expression
  - → Use an n-to-2^n-line decoder to generate the minterms & an external OR gate to form the logical sum
- Any combinational ckt w/ n inputs and m outputs
  - → Use an n-to-2^n-line decoder & m OR gates

J.J. Shann 4-82

82

Example

E.g.: Decoder and OR gate implementation of a binary adder bit (Full adder)

<Ans.>

3 inputs:

X, Y (the two bits being added) ,

Z (the incoming carry from the right)

2 outputs:

S (the sum bit)

C (the carry bit)

$\Rightarrow$ Need a 3-to-8-line decoder & 2 OR gates.

Truth Table for 1-bit Binary 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

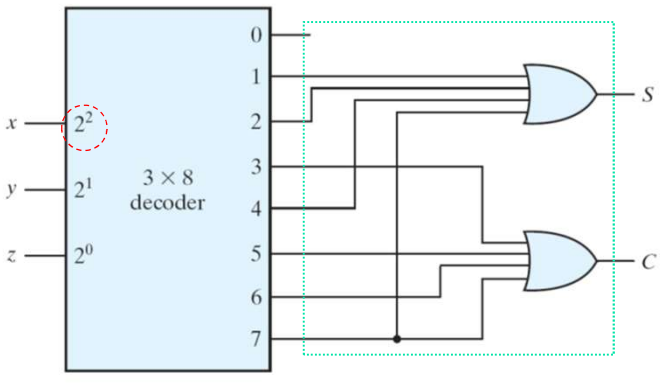
$$S(X,Y,Z) = \sum m(1,2,4,7)$$
$$C(X,Y,Z) = \sum m(3,5,6,7)$$

J.J. Shann 4-83

83

MSB

$$S(X,Y,Z) = \sum m(1,2,4,7)$$
$$C(X,Y,Z) = \sum m(3,5,6,7)$$



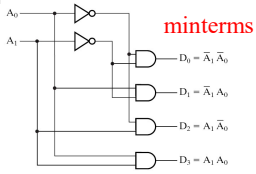
* Another approach: S′(X, Y, Z) = Σm(0,3,5,6) $\Rightarrow$?

C′(X, Y, Z) = Σm(0,1,2,4) $\Rightarrow$?

J.J. Shann 4-84

84

Possible Approaches Using Decoder



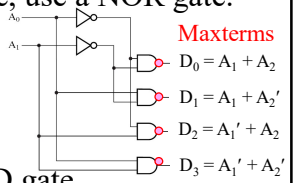
- For decoder w/ active HIGH outputs:

  (Assumption: function F has k minterms)

  – ORed the minterms of F : k-input OR gate

  – NORed the minterms of F' : $(2^n - k)$-input NOR gate

  * If $k \le 2^n/2$, then use an OR gate; otherwise, use a NOR gate.



- For decoder w/ active LOW outputs:

  (Assumption: function F has k maxterms)

  – ANDed the maxterms of F : k-input AND gate

  – NANDed the maxterms of F' : $(2^n - k)$-input NAND gate

  * If $k \le 2^n/2$, then use an AND gate; otherwise, use a NAND gate.

Summary

- Summary:

  – The decoder method can be used to implement any combinational ckt.

  – When the decoder method may provide the best solution:

    ➢ If the combinational ckt has many outputs and if each output function (or its complement) is expressed w/ a small # of minterms or maxterms.

4-11

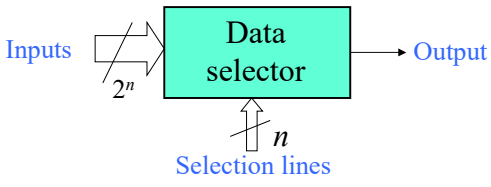
Multiplexers

J.J. Shann

87

Multiplexers

- Selection ckts: the inverse function of a demux
  - typically have a set of inputs from which selections are made, a single output, and a set of control lines for making the selection.



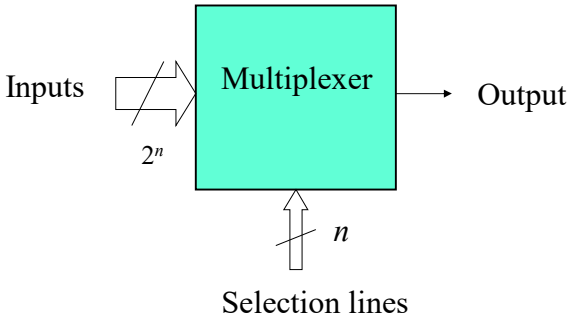
- Implementation of selection ckts:
  - Boolean function implementation
  - using three-state gates

J.J. Shann 4-88

88

A. Multiplexers

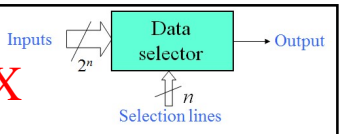
- Multiplexer (MUX): Data selector
  - a combinational ckt that selects binary information from one of many input lines & directs it to a single output line.
  - 2^n input lines, n selection lines, and one output line



J.J. Shann 4-89

89

Example: 2-to-1-line MUX



- E.g.: 2-to-1-line MUX

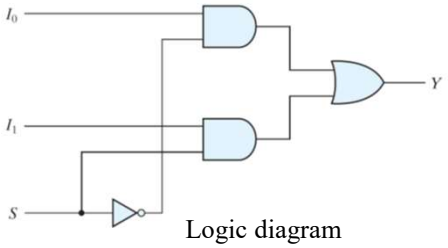
Truth table

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

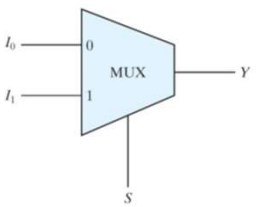
Condensed truth table

S	Y
0	I_0
1	I_1

$$Y = \overline{S} I_0 + S I_1$$



Logic diagram

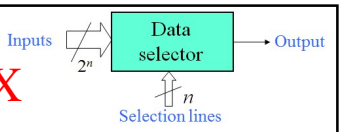


Block diagram

4-90

90

Example: 4-to-1-line MUX

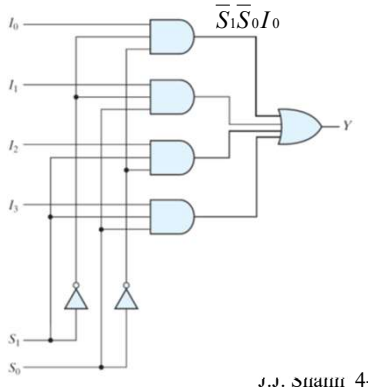


- E.g.: 4-to-1-line MUX

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

$$Y = \overline{S}_1\overline{S}_0 I_0 + \overline{S}_1 S_0 I_1 + S_1\overline{S}_0 I_2 + S_1 S_0 I_3$$

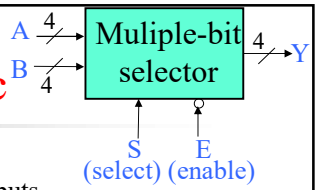
GIC = 18 (including inverter inputs)



J.J. Shann 4-91

91

Multiple-bit Selection Logic



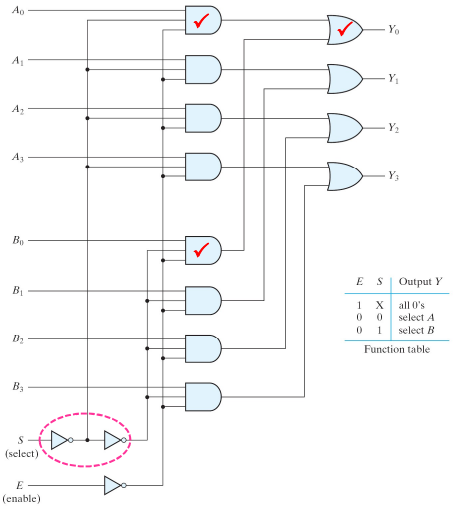
- Multiple-bit selection logic:
  - combine MUX ckts w/ common selection inputs
- Example:

Quadruple 2-to-1-line MUX
(w/ active LOW enable input)

$A = A_3 A_2 A_1 A_0$
$B = B_3\ B_2 B_1\ B_0$

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

Function table

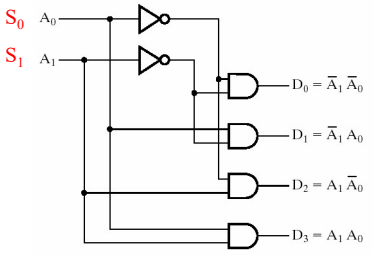


92

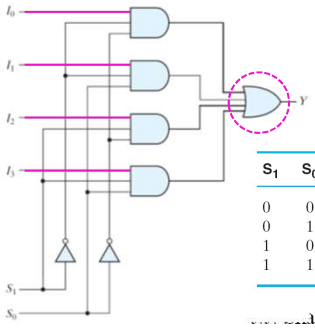
MUX vs. Decoder

- 2^n-to-1-line MUX:
  - a n-to-2^n decoder (2^n AND gates)
  - add the 2^n input lines to each AND gate
  - a OR gate: ORed all the AND gates
  - an enable input: optional (for expansion)

2-to-4 line decoder



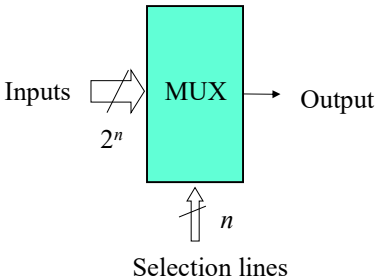
4-to-1 line Mux

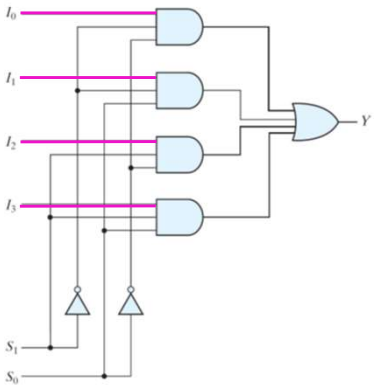


S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Boolean Function Implementation

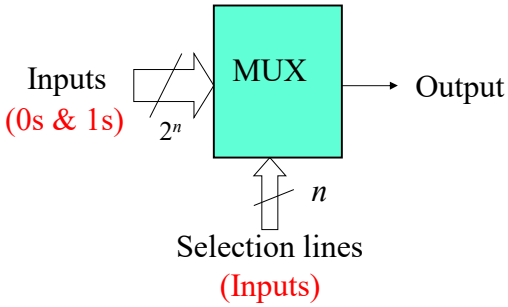
- MUX: a decoder + an OR gate
  - Decoder: generates the minterms of the selection inputs.
  - OR gate: sum the minterms





Implementing n-variable Boolean function with 2^n-to-1 MUX

- Implement a Boolean function of n input variables w/ 2^n-to-1 MUX (a MUX that has n selection lines):
  - The minterms to be included w/ the function are chosen by making their corresponding input lines equal to 1, those minterms not included in the function are disabled by making their input lines equal to 0.



Example

- E.g.: MUX implementation of a binary adder bit

Truth Table for 1-bit Binary Adder

Full adder

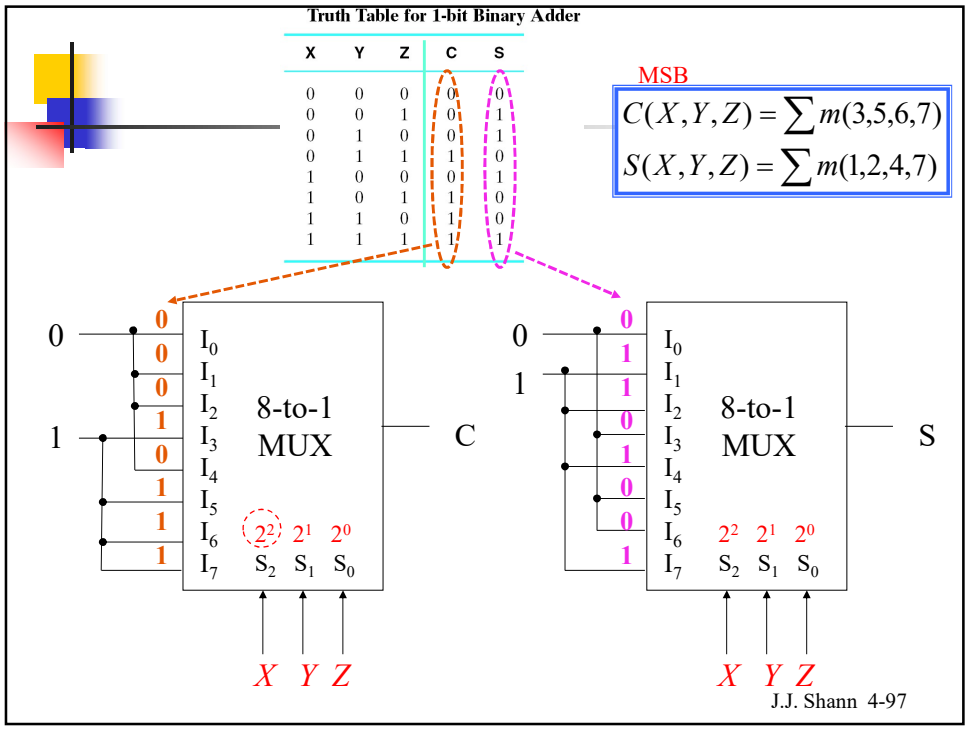
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$C(X,Y,Z) = \sum m(3,5,6,7)$$

$$S(X,Y,Z) = \sum m(1,2,4,7)$$

<Ans.>

Need two 8-to-1-line MUXs



97

Implementing *n*-variable Boolean function with 2^{n-1}-to-1 MUX

- Implement a Boolean function of *n* input variables w/ a 2^{n-1}-to-1 MUX which has $(n-1)$ selection lines:
  - The first $n-1$ variables of the function are connected to the selection inputs of the MUX.
  - The remaining single variable (Z) of the function is used for the data inputs: Z, Z', 1, or 0.
  - Procedure:
    - The Boolean function is first listed in a truth table.
    - The first $n-1$ variables in the table are applied to the selection inputs of the MUX.
    - For each combination of the selection variables, we evaluate the output as a function of the last variable Z: Z, Z', 1, or 0
    - These values are then applied to the data inputs in the proper order.

J.J. Shann 4-98

98

Example

E.g.: Alternative MUX implementation of a binary adder bit

Full adder

Truth Table for 1-bit Binary 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

Use two 4-to-1-line MUXs to implement the ckt. Choose X and Y as the selection inputs.

* It is possible to use any other variables of the function for the MUX selection inputs.

J.J. Shann 4-99

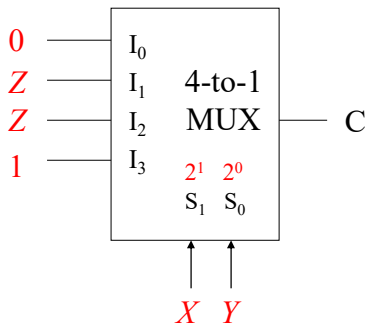
99

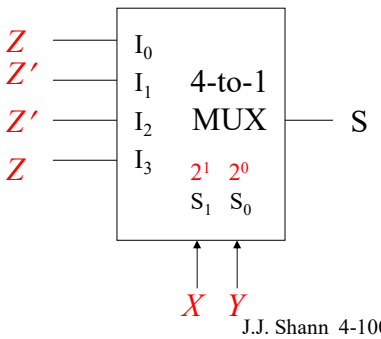
X	Y	Z	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

0
Z
Z
1

X	Y	Z	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Z
Z′
Z′
Z





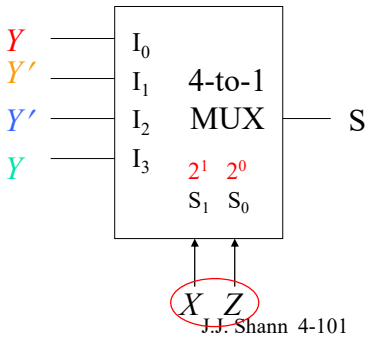
J.J. Shann 4-100

100

- It is possible to use any other variables of the function for the MUX selection inputs.
  - E.g.: Realize *S* by using *X* and *Z* as the selection lines.

X	Y	Z	S	
0	0	0	0	Y
0	0	1	1	
0	1	0	1	Y′
0	1	1	0	
1	0	0	1	Y′
1	0	1	0	
1	1	0	0	Y
1	1	1	1	

X	Z	Y	S



Example

- E.g.: MUX implementation of 4-variable function

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

  i. Use an 8-to-1 MUX to realize the function

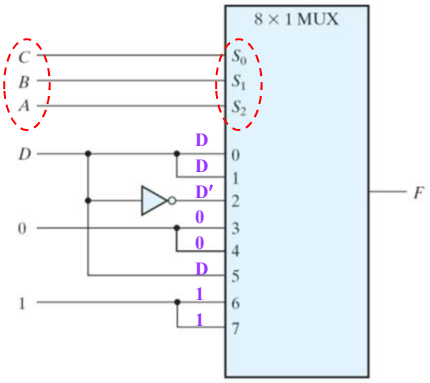
  ii. Use a 4-to-1 MUX to realize the function

F(A,B,C,D) = Σ(1,3,4,11,12,13,14,15)

<Ans.>

i. Use an 8-to-1 MUX

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	



J.J. Shann 4-103

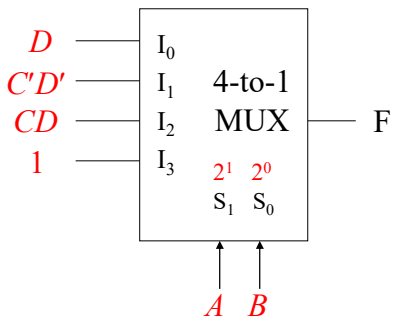
103

F(A,B,C,D) = Σ(1,3,4,11,12,13,14,15)

<Ans.>

ii. Use a 4-to-1 MUX

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



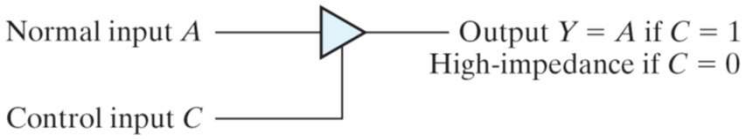
J.J. Shann 4-104

104

B. Three-State Gates

- Three-state gate:
  - A digital ckt that exhibits three states:
    logic 1, logic 0, high-impedance state
    - High-impedance state: behaves like an open ckt
      The output appears to be disconnected and the ckt has no logic significance.
  - Special feature:
    - A large # of three-state gate outputs can be connected w/ wires to form a common line w/o endangering loading effects.

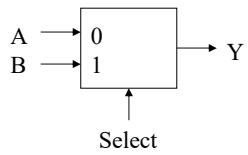
- Three-state buffer:

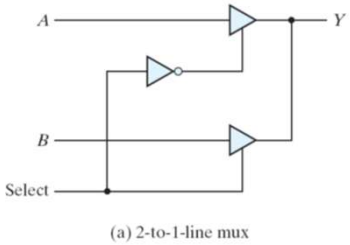


105

- MUXs w/ three-state gates:

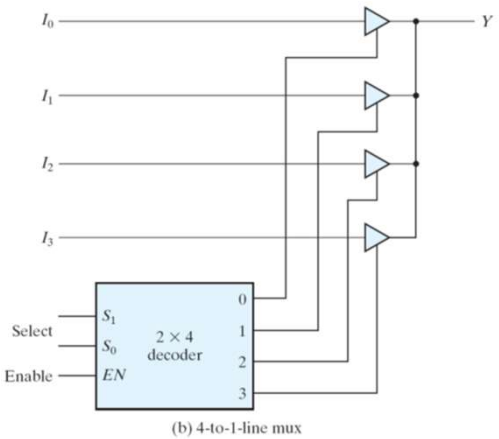
2-to-1 Mux





(a) 2-to-1-line mux

4-to-1 Mux



(b) 4-to-1-line mux

106

4-12

HDL Models of Comb. Circuits

J.J. Shann

107

Chapter Summary

- Analysis procedures of combinational circuits
- Design procedures of combinational circuits
- Function blocks used frequently to design larger ckts:
  - Arithmetic Circuits:
    - Binary Adder-Subtractor, Decimal Adder, Binary Multiplier, Magnitude Comparator
  - Encoders
  - Decoders (Demultiplexers)
  - Multiplexers
- Design of combinational logic ckts using
  - Decoders
  - Multiplexers

J.J. Shann 4-108

108