EE 200: Digital Logic Circuit Design

Dr Radwan E Abdel-Aal, COE

Unit 3
Combinational Logic

- Introduction to Analysis & Design with Examples
- Arithmetic Functions and Circuits
- MSI Functional Blocks: Decoders, Encoders, etc.

Unit 3: Combinational Logic (CL) Design Contents

Combinational Logic Circuits

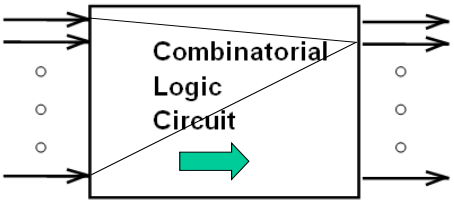
- A combinational logic circuit that has:
  - A set of *m* Boolean inputs,
  - A set of *n* Boolean outputs

  Performs n logic functions, each mapping the 2^m input combinations to an output
- Outputs are determined only by the **present** inputs

(appearing after some delay)
- No feedback paths
- No memory elements

Each Output = F (the m inputs)

Effect of Previous inputs is **NOT** "remembered"



3

CL Circuit Analysis Procedure:

Analysis:

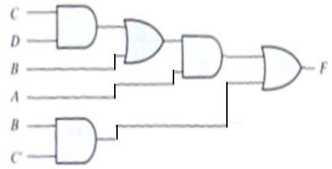
Given:
a CL Circuit* (logic diagram),
Determine:
the logic function implemented by the circuit

We can describe such a logic function by:
- **A set of Boolean Equations, or**
- **A truth table, or**
- **A word description**

***Ensure circuit is combinational: It should not have O/P to I/P feedback through storage elements**

4-Step Procedure to get O/P Function:

1. Label the outputs of gates that are functions of only the circuit inputs and obtain the Boolean function of each output

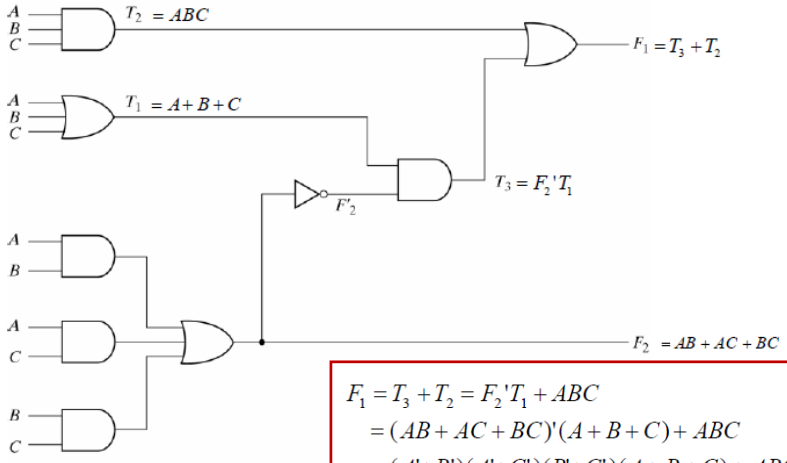


2. Label the outputs of gates that are functions of the inputs and the outputs of gates in step 1, and obtain the Boolean function of each output

3. Repeat 2 until you obtain the final output of the circuit

4. Use direct substitution to determine each output as a function of the external circuit inputs, e.g. as a SOP

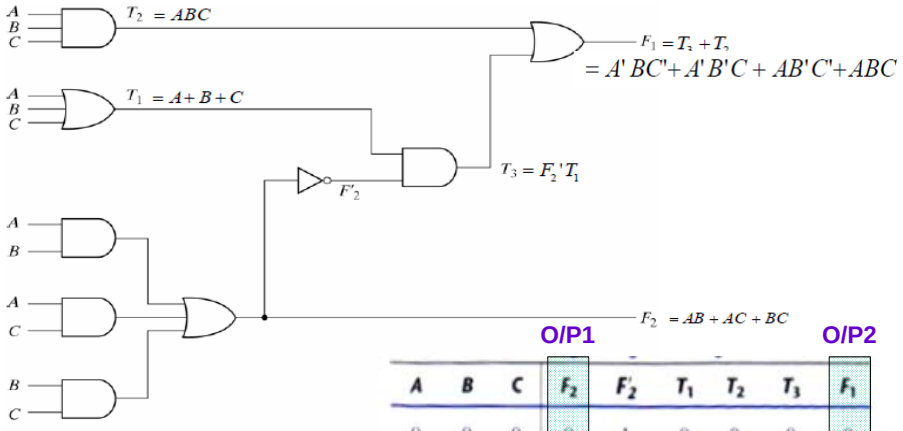
O/P Function: Example



$$F_1 = T_3 + T_2 = F_2' T_1 + ABC$$
$$= (AB + AC + BC)'(A + B + C) + ABC$$
$$= (A'+B')(A'+C')(B'+C')(A + B + C) + ABC$$
$$= (A'+B'C')(AB'+AC'+BC'+B'C) + ABC$$
$$= A'BC' + A'B'C + AB'C' + ABC$$

$$F_2 = AB + AC + BC$$

Truth Table: Example



O/P1 O/P2

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

Truth Table: 2 Ways:

1. Step by step- T1, T2 etc.
2. Get eqn. of the final output and plot it on the table

CL Design Procedure: 5 Steps

Given:

A specification of required functionality (as a Word description, Truth table, Boolean Equations),

Determine:

The logic diagram for an optimal circuit that provides the functionality

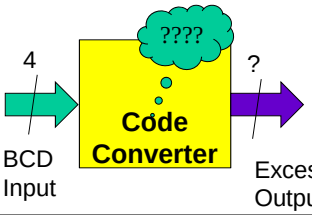
1. **From the Specification given, determine the number of inputs, number of outputs and label them**
2. **Work out the truth table specification for each output (if not given)**
3. **Obtain an optimized* logic expression for each outputs (using K-maps etc.). Global optimization if multiple outputs**
4. **Get logic diagram and truth table of circuit used and verify that it fulfils the required specification – manually or using a simulator**
4. **Implement with a universal gate if required**

***Note: In practice, other physical parameters need also to be optimized and verified, e.g. propagation delay, area on the chip, power consumption, etc.**

Design Example: BCD to Excess 3 Code Converter

Specifications

- The circuit should convert a BCD input code (decimal digits 0-9) to the corresponding Excess-3 code
  - Inputs: BCD code words for digits 0 through 9: 4-bit patterns 0000 to 1001, respectively
  - Outputs: Excess-3 code words for digits 0 through 9: 4-bit patterns obtained by adding 3 (binary 0011) to each BCD code input
  - Utilize Don't cares!



9

Design Example: BCD to Excess 3 Code Converter

1, 2: Formulation

- **How many outputs we need? Name them**

→ **The Truth Table**

- **I/P Variables**
  - **BCD: A,B,C,D**
- **O/P Variables**
  - **Excess-3 W,X,Y,Z**
- **Don't Cares**
  - **BCD codes 1010 to 1111**

Index	Input BCD A B C D	Output Excess-3 W X Y Z
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0
10	1 0 1 0	X X X X
..	.....	X X X X
15	1 1 1 1	X X X X

Design Example 1: BCD to Excess 3 Code Converter

3. Logic Optimization

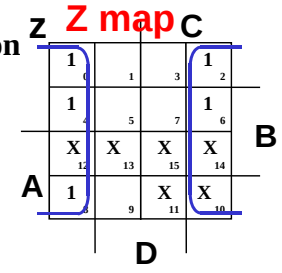
2-level using K-maps

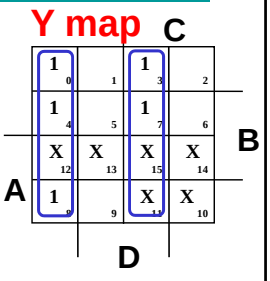
$W = A + BC + B\overline{C}\overline{D}$

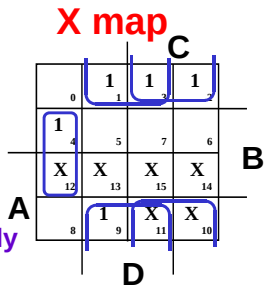
$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D}$

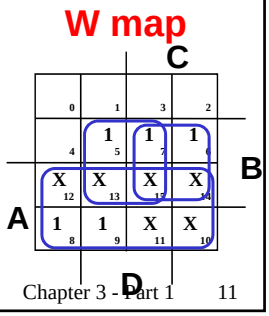
$Y = C\overline{D} + \overline{C}D$

$Z = \overline{D}$









Standard SOP 2-level Form (Directly from the K-maps)

Optimized each O/P separately

Design Example 2: BCD to Excess 3 Code Converter

3. Contd. Global Logic Optimization

b. Further optimization through multi-level, from:

$W = A + BC + B\overline{C}\overline{D}$

$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D}$

$Y = C\overline{D} + \overline{C}D$

$Z = \overline{D}$

For circuits having Multiple Outputs (here 4)

- **By taking common factors:**

Let $T_1 = C + D$

$W = A + BT_1$

$X = \overline{B}T_1 + B\overline{C}\overline{D}$

$Y = C\overline{D} + \overline{C}D$

$Z = \overline{D}$

Simpler but non-standard Form (no longer SOP, i.e. > 2 logic levels- (multi-level logic)
→ Now (C+D) is generated only once and used by the 2 outputs W, X!

Design Example 2:
BCD to Excess 3 Code Converter

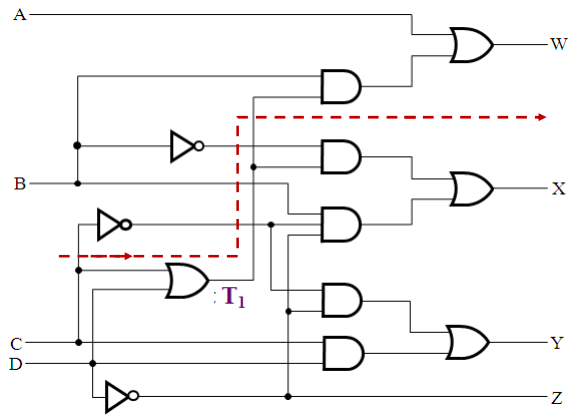
Let $T_1 = C + D$

$W = A + BT_1$

$X = \overline{B}T_1 + B\overline{C}\,\overline{D}$

$Y = CD + \overline{C}\,\overline{D}$

$Z = \overline{D}$



Multilevel (non-standard) implementation
Optimizes the logic but can increase propagation delay

Design Example 1:
BCD to Excess 3 Code Converter

4. Verification- Get the truth table of the actual circuit implemented (analysis) and show it satisfies the specified truth table:

Input BCD A B C D	Output Excess-3 WXYZ
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

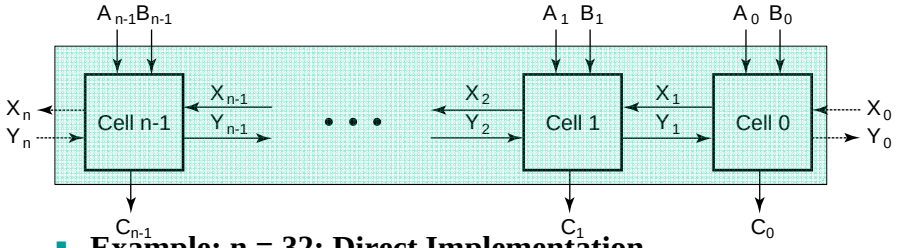
Truth table matched that for the specifications

Arithmetic Combinational Circuits: Iterative Cells (Repeated - in space or time)

- Practical Arithmetic Functions:
  - Operate on binary **bit vectors** (e.g. a 32-bit adder adds two 32-bit numbers and produces a 32-bit sum)
  - Use same basic sub-function for each **bit** position
- Designing circuits that handle the I/P vectors directly can be very difficult (large # of inputs & outputs! → large K-maps, Huge truth tables!)
- Solution: Design functional block for sub-function (e.g. for a bit) and repeat it (iterate it, reuse it) to obtain a functional block for the overall operation
- *Cell* = sub-function block
- *Iterative array* = An array of interconnected such cells

Block Diagram of a 1-D Iterative Array

Adder/Subtractor for two n-bit integers: C=A+/-B



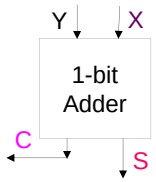
- **Example: n = 32: Direct Implementation**
  - **Number of inputs = ? (K-map size)**
  - **Truth table rows = ?**
  - **Equations in up to ? input variables**
  - **Equations with a huge number of literals**
  - **Flat Design: impractical!**
- **Iterative array takes advantage of regularity to make designs more feasible**

Functional Blocks: Addition

- Binary addition is used frequently in computers
- "Adder" Design:
- 1-bit adder cell (i.e. does the addition for 1 digit)
  - *Half-Adder* (HA), a 2-input bit-wise addition functional block
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block
- For an n-bit iterative adder: Combine n 1-bit FA adder cells together- Two ways:
  - *Ripple Carry Adder* (RCA): Carry ripples through the adder from LSB to MSB → slows down the addition operation
  - *Carry-Look-Ahead Adder* (CLA), Speeds up addition by letting each bit stage generate its carry input from scratch (i.e. from the input numbers **directly)** to avoid waiting for the carry to ripple through all previous stages

The Functional Blocks: Half-Adder (HA)

- **A 2-input (no carry input), 1-bit wide binary adder that performs the following computations:**



X	**0**	**0**	**1**	**1**
+ Y	**+ 0**	**+ 1**	**+ 0**	**+ 1**
C S	**0 0**	**0 1**	**0 1**	**1 0**

- **A half adder adds two bits, giving two outputs: S & C**
- **The result is expressed as a <u>sum bit</u> S and a <u>carry bit</u> C**
- **The half adder can be specified by the truth table ⇒**

I/Ps		O/Ps	
X	**Y**	**C**	**S**
0	**0**	**0**	**0**
0	**1**	**0**	**1**
1	**0**	**0**	**1**
1	**1**	**1**	**0**

- Applications:
- Use as the 1st stage of n-bit adder (no carry I/P)
- Use two HAs to make a full adder (FA)- **see later**

Logic Simplification: Half-Adder (HA)

- **The K-maps for S, C are:**

S

X \ Y	0	1
0	0	1 (1)
1	1 (2)	3

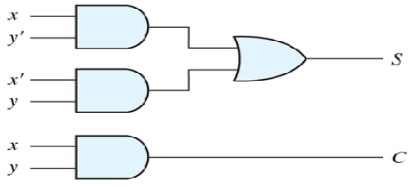
C

X \ Y	0	1
0	0	1
1	2	1 (3)

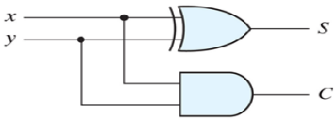
X	Y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- **By inspection:**

$$S = X\,\overline{Y} + \overline{X}\,Y = X \oplus Y \qquad C = X\,Y$$



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



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

19

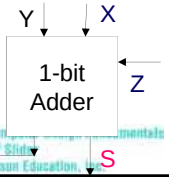
Functional Block: Full-Adder

- **A full adder is similar to a half adder, but includes a carry-in bit from the lower stage. Like the half-adder, it computes a sum bit, S and a carry bit, C.**
  - **For a carry-in (Z) of 0, it is the same as the half-adder:**

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

  - **For a carry- in (Z) of 1:**

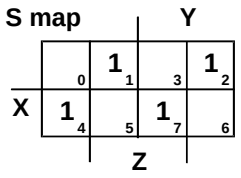
Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

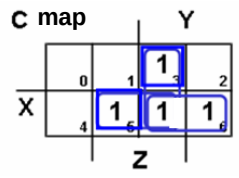


Logic Optimization: Full-Adder

- **Full-Adder Truth Table:**
- **Full-Adder K-Maps:**

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





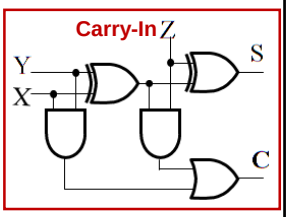
$$S = X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + \bar{X}\bar{Y}Z + XYZ$$

$$S = X \oplus Y \oplus Z$$

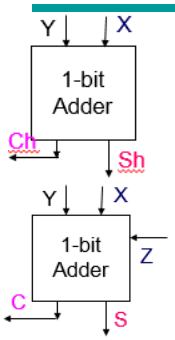
The Odd function

$$C = XY + X\bar{Y}Z + \bar{X}YZ$$
$$= XY + Z(X\bar{Y} + \bar{X}Y)$$
$$C = XY + Z(X \oplus Y)$$

Generated Carry — Propagated Carry

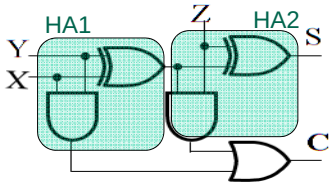


Full-Adder (FA): Implementation Using two Half-Adders (HAs) + OR Gate



Half-Adder

$$Sh = X \oplus Y$$
$$Ch = X\,Y$$

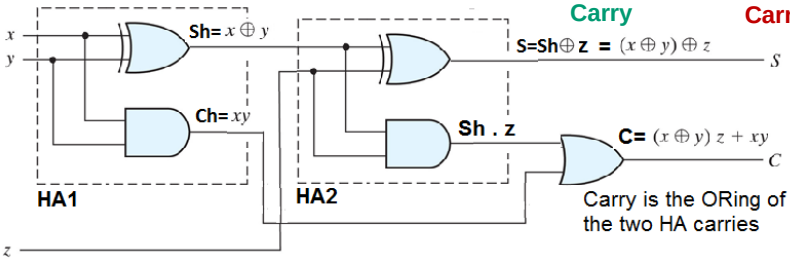


Full-Adder

$$S = X \oplus Y \oplus Z$$
$$= Sh \oplus Z$$

$$C = X\,Y + (X \oplus Y).\,Z$$
$$= Ch + Sh.\,Z$$

Generated Carry — Propagated Carry

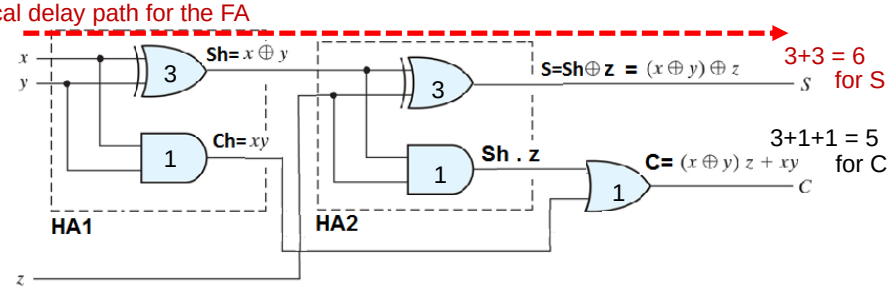


Carry is the ORing of the two HA carries

Worst Case (Critical Path) Propagation Delay for a FA Stage

Assume:
XOR: 3 standard gate delays
OR: 1 standard gate delay, AND: 1 standard gate delay

critical delay path for the FA



Path from input (X,Y,Z) to S is the critical path (largest delay to output)
Hence propagation delay is 6 standard gate delays
If this gate delay is 5 ns → FA propagation delay = 6 x 5 = 30 ns.
- How many additions per sec??

4-bit binary adder block using 4 FA stages
1. The ripple carry approach (simple but slow)

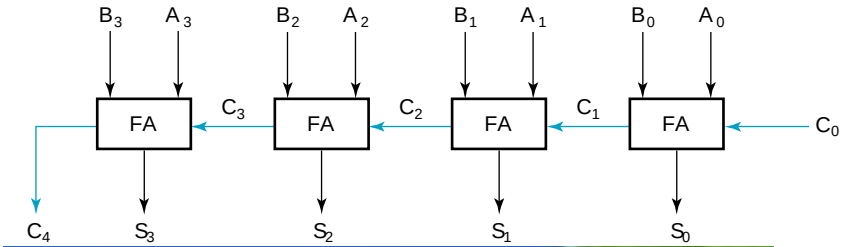
- Mimics what we do when adding multiple bits with paper and pencil
- Carry output from FA stage i is fed as carry input to FA stage (i+1)

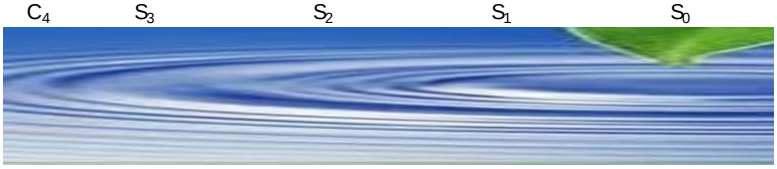
```
  ← ←
  1011
+ 0011
------
  1110
```

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}

4-bit Ripple-Carry Binary Adder

- **Four-bit Ripple Carry Adder made from four 1-bit Full Adder cells:**



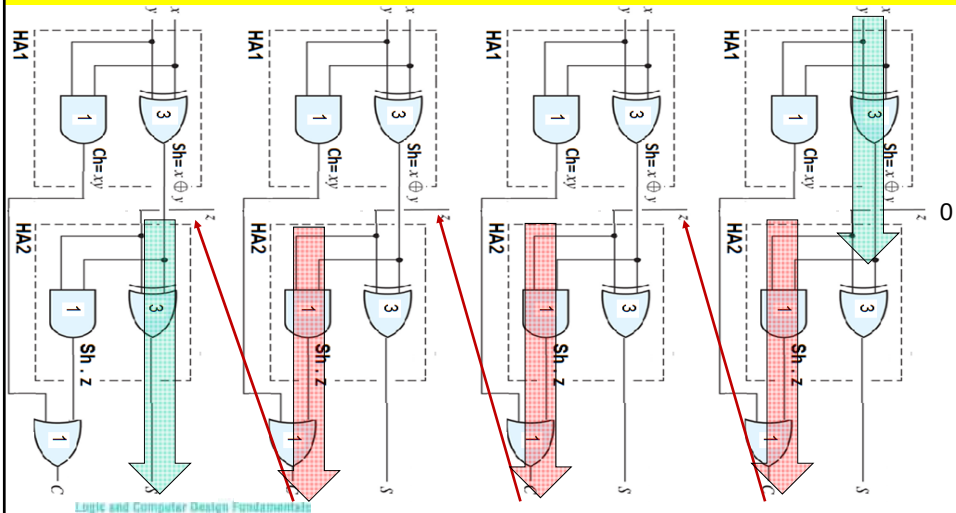


Problem: Carry has to ripple through till the end stage for the final result to appear
→ Slow addition speed for large number of stages (bits) n

25

4-bit Ripple-Carry Binary Adder
Propagation Delay Analysis

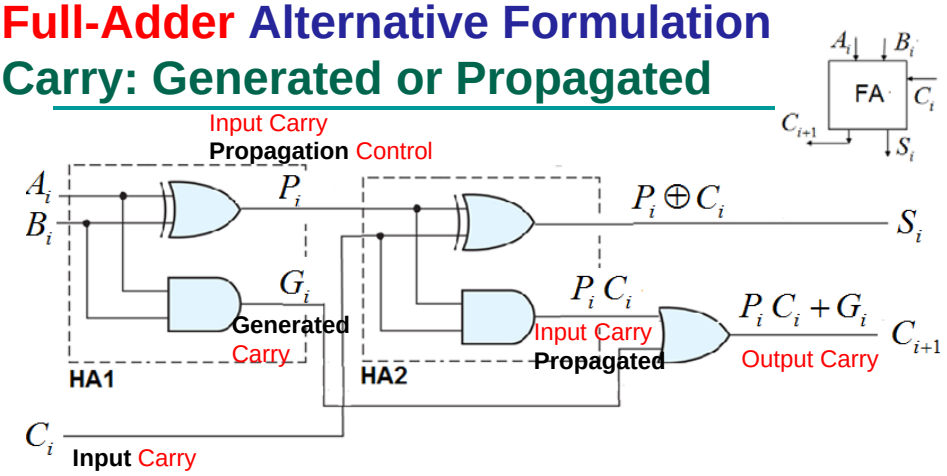
Total Worst Case Delay = 3 + 2 + 2 + 2 + 3 = 12 gate delays = 6 + (n-1) 2, n = 4 stages



26

Full-Adder Alternative Formulation

Carry: Generated or Propagated



- **Output carry occurs if either:**
  - **Carry is generated on the stage (if Ai=Bi=1). or**
  - **Input carry to the stage (Ci=1) is propagated through it if Pi=1**
- **Note: Ai, Bi (hence Pi and Gi) and C0 are not affected by ripple delay**
- **We can avoid carry ripple delay problem if we generate Ci+1 from Pis, Gis and C0**

27

Carry Lookahead Binary Adder

Speed up at the expense of more complex hardware

- **C_{i+1} outputs by all stages are derived in parallel with a set of equations using A, B, C0 inputs only**
- **Beginning at cell 0 with carry in C_0:**

Note: All P_i, G_i are functions of (A_i, B_i) Only!

Carry does not ripple anymore!
All 4 carry outputs can be generated **in one go** from: C0 and the two input numbers to be added (A,B)

✓ $C_1 = G_0 + P_0 C_0$ (No change)

$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$

✓ $C_2 = G_1 + P_1G_0 + P_1P_0 C_0$

2-Level logic
(Only 2 gate delays)

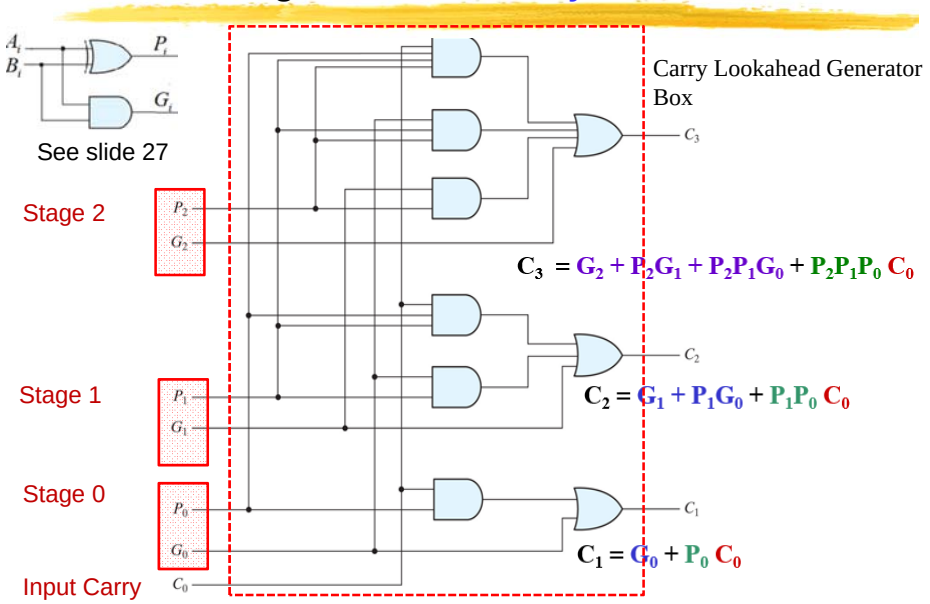
$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0 C_0)$

✓ $C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0 C_0$

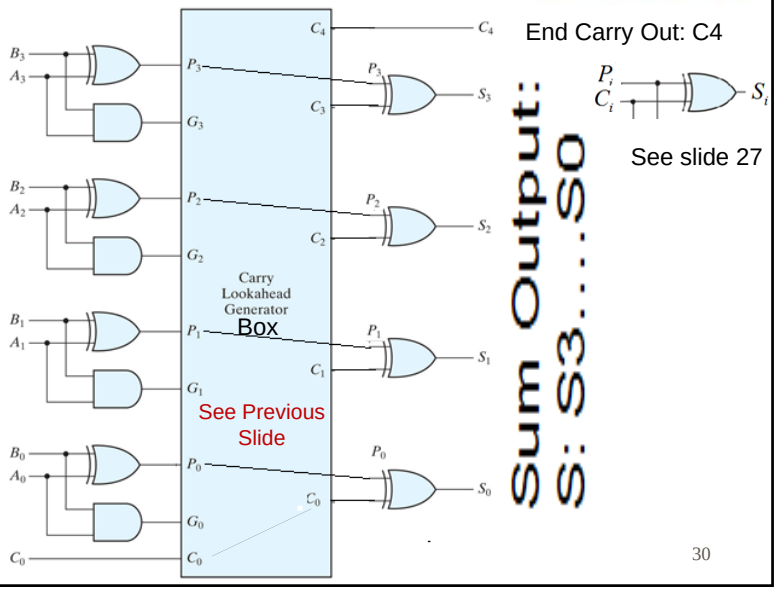
$C_4 = G_3 + P_3 C_3 = \underbrace{G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0}_{\text{Carry generated within the 4-bit block}} + \underbrace{P_3P_2P_1P_0 C_0}_{\text{Carry Propagated through the 4-bit block}}$

✓ No Rippling

Carry Lookahead Generator Box (for 4-bit adder)
C1 to C4 are generated - Only C1-C3 shown

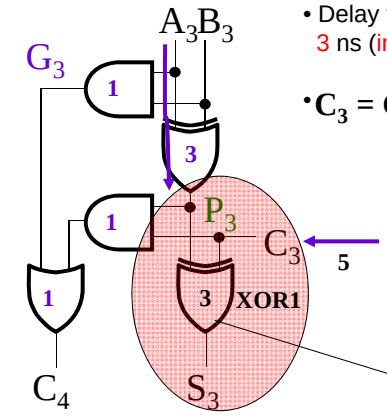


4-bit Carry Lookahead Adder: A+B = C4 S



30

Delay reduction for the 4-bit Carry Lookahead adder: Consider last stage



Last stage of the CLA 4-bit adder

- Delay from {A,B, C0} to S3 (sum from last bit) is 3 ns (in XOR1) + max (delay of P3, Delay of C3)
- $C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0\ C_0$

Ps are the **slowest** Inputs to the SOP (3 ns)

All Ps are calculated simultaneously

3 ns for **inputs** to SOP
+ 1 ns for the AND (Product)
+ 1 ns the OR (Sum)

5 Gate delays for C3

= 3 + max (3,5) = 3 + 5 = **8** ns
(Regardless of the # of stages!!)

Vs **12** ns for the 4-bit carry ripple adder

Adder/Subtractor Combined Hardware In Signed 2's Complement Notation

- **Only one adder computes A + B or A– B, as specified by Sub/$\overline{\text{Add}}$ I/P**
- **For Control input = 0 (add):**

  B is passed through to the adder as is, $C_0 = 0$

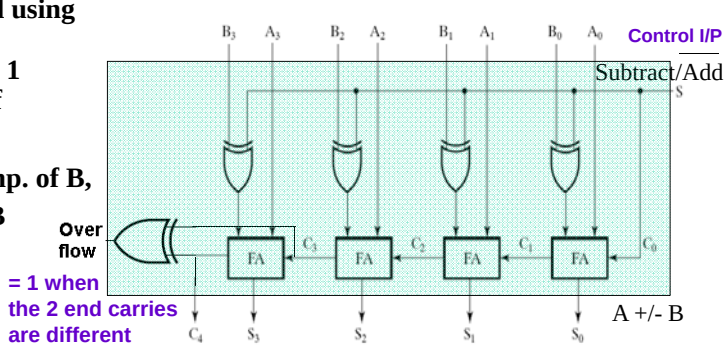
  i.e. result = A+B
- **For Control input = 1 (subtract):**

  → 2's complement of B is obtained using XORs to form the 1's comp, + 1 applied to C_0 of 1st stage

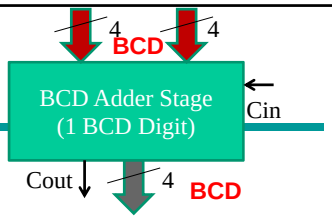
  → A + 2's comp. of B, i.e. result = A-B

All inputs and result are represented in the 2's Complement Notation

Where is the sign bit?



BCD Decimal Adder



A circuit that adds two decimal digits (0-9) (in BCD) with a possible carry input and gives the sum as a BCD + a carry out

9 inputs, 5 outputs

	0	1	0
+	6	7	9
	2	4	9
	= 8 (<9)	= 12 (>9)	= 18 (>9)
	0 8	1 2	1 8
	carry	carry	carry

Using a standard 4-bit binary adder - What changes are needed?

- For sums ≤ 1001 binary and BCD are identical- No correction
- For sums ≥ 1010 , we should subtract $10)_d$ and **send a carry to next BCD stage**
- Instead of subtracting $10)_d$, = 1010, we add its 2's complement which is 0110
- Use the binary adder o/ps to determine if correction is needed

Binary Adder Performance → **Required BCD Performance**

Binary adder						BCD adder					Decimal
K	Z_8	Z_4	Z_2	Z_1		C	S_8	S_4	S_2	S_1	sum
0	0	0	0	0	Identical- No Change	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	Corrections Needed	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

Binary Carry

BCD Carry

For sum ≥ 10 Subtract 10 (add 6) and set Carry

BCD decimal adder

Numbers that need correction (add 6) are:

K	Z8	Z4	Z2	Z1	
0	1	0	1	0	(10)
0	1	0	1	1	(11)
0	1	1	0	0	(12)
0	1	1	0	1	(13)
0	1	1	1	0	(14)
0	1	1	1	1	(15)
1	0	0	0	0	(16)
1	0	0	0	1	(17)
1	0	0	1	0	(18)
1	0	0	1	1	(19)

C = K + Z8Z4 +Z8Z2

BCD Carry = 0 → Add 0
= 1 → Add 6

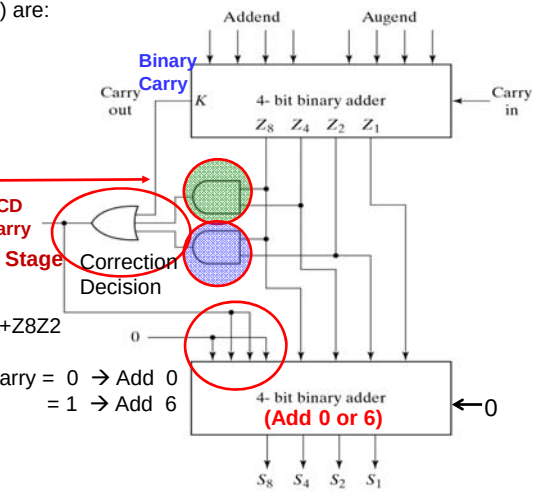


Fig. 4-14 Block Diagram of a BCD Adder

35

Binary Multiplier

- **Binary multination is done in the same way as decimal multiplication.**
- **Multiplicand is multiplied by each bit of the multiplier. Shift results 1-bit for each bit of multiplier. Add.**

Multiplicand	1011	11
Multiplier	x 101	5
Partial Products	1011	
	0000 -	
	1011 - -	
Product	110111	55

36

Binary Multiplier: 2-bit x 2-bit

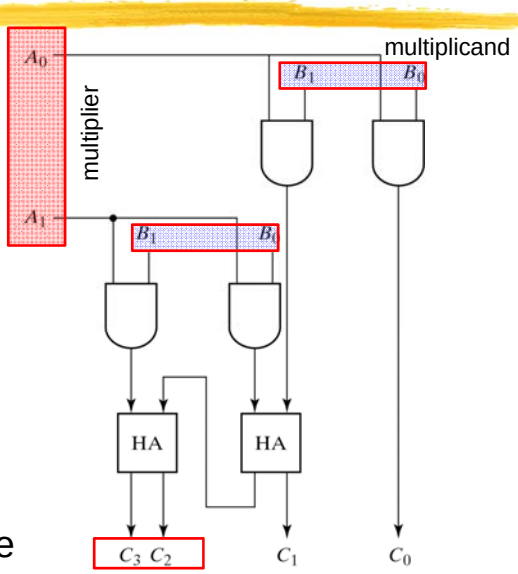
multiplicand B_1 B_0

multiplier A_1 A_0

A_0B_1 A_0B_0

A_1B_1 A_1B_0

C_3 C_2 C_1 C_0



C = A * B

C does not
stand for Carry here

37

Fig. 4-15 2-Bit by 2-Bit Binary Multiplier

4-bit by 3-bit Binary Multiplier

B3 B2 B1 B0
A2 A1 A0

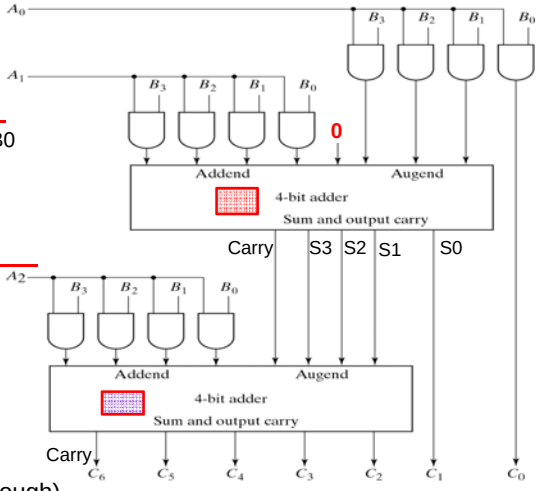
0 A0B3 A0B2 A0B1 A0B0

A1B3 A1B2 A1B1 A1B0

Carry S3 S2 S1 S0

A2B3 A2B2 A2B1 A2B0

C6 C5 C4 C3 C2

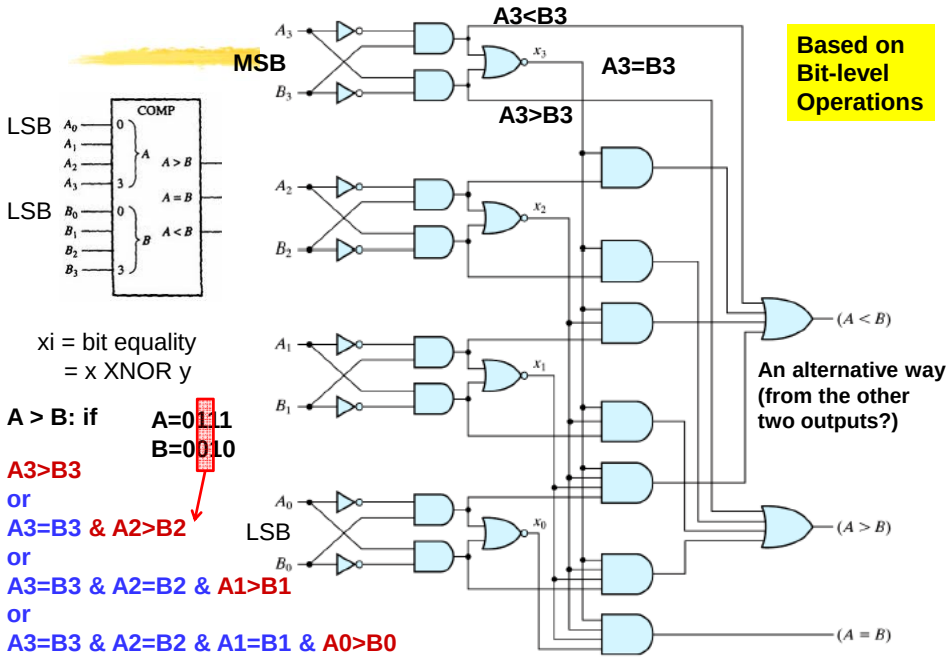


Largest Results:
15 x 7 = 105
(7 bits are enough)

38

Fig. 4-16 4-Bit by 3-Bit Binary Multiplier

4-bit Magnitude Comparator



4-bit Magnitude Comparator

- The comparison of two numbers
  - outputs: A>B, A=B, A<B
- Design Approaches
  - the truth table **for an n-bit comparator**
    - 2^{2n} entries - too cumbersome for large n
  - use inherent regularity of the problem
    - reduce design efforts **(Bit-level Operations)**
    - reduce human errors
- Algorithm -> logic
  - $A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$
  - A=B if $A_3=B_3$, $A_2=B_2$, $A_1=B_1$and $A_0=B_0$
    - equality: $x_i = A_iB_i+A_i'B_i'$
    - $(A=B) = x_3x_2x_1x_0$
  - $(A>B) = A_3B_3'+x_3A_2B_2'+x_3x_2A_1B_1'+x_3x_2x_1A_0B_0'$
  - $(A<B) = A_3'B_3+x_3A_2'B_2+x_3x_2A_1'B_1+x_3x_2x_1A_0'B_0$

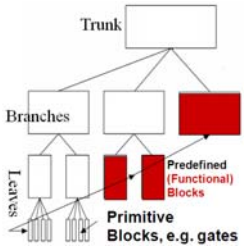
40

Other Combinational Logic Functions

- **Functions & Functional Blocks that implement them**
  - **Enabling**
  - **Encoding, Encoding with Priority in mind**
  - **Decoding, Demultiplexing (Data routing)**
  - **Multiplexing (Data selecting)**
- **Implementing any combinational function using:**
  - **Decoders & OR gates**
  - **Multiplexers (with inverters if needed)**
- **MSI Functional Block Applications**

Functions and Functional Blocks

- We consider here functions that are useful in designing and building other (higher-level) combinational and sequential circuits

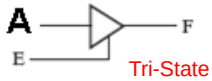


- Such functions may exist as *functional blocks*
  - In the past, many such blocks were implemented as discrete integrated circuits (ICs): SSI (small scale integration), MSI, and LSI **e.g. the 7483 is a 4-bit CLA adder, 74157 is a 4-bit Multiplexer**
  - Today, they are often available as components in a design library for use within larger VLSI circuits

2

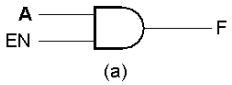
1. Enabling Function

- **Enable: Allow an input signal to pass through to an output**
- **Disable: block an input signal from passing through to an output, replacing it with a fixed state. This could be HiZ, 1, 0, depending on the gate used)**



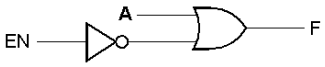
Tri-State Buffer

- **Later we use the enable function to implement decoders and multiplexers**
- **Two Examples: EN = 1 → Enable, EN = 0 → Disable**
  - When I/P disabled, output = 0



(a)

  - When I/P disabled, output = 1

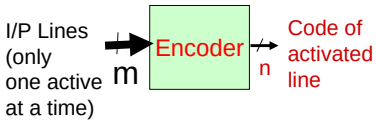


(b)

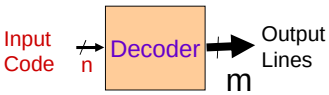
Encoding & Decoding

$m = 2^n$
For full Decoders/Encoders

Encoder: For each unique activated input line, Generate the corresponding code



Decoder: For each input code, Activate the unique corresponding output line



n-to-m decoder

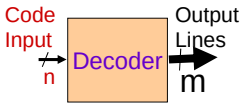
Only the line Corr. to the I/P code is activated

Smallest decoder is?

Encoder is the opposite of Decoder

Will start with Decoding

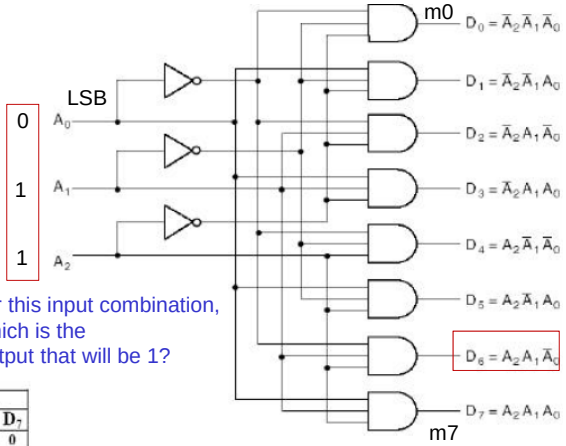
2. Decoding



- A decoder converts an *n*-bit **input code** to a unique state on **m outputs** where
  $2 \le m \le 2^n$ (with $m = 2^n$ we call it a Full Decoder)
- For each valid input code, only one unique output line is activated
- Decoder functional blocks:
  - Are called *n*-to-*m* line decoders, where $m \le 2^n$, and
  - Generate 2^n (or fewer) minterms from the *n* input variables

Decoder Design: 3-to-8 Example

Direct Approach: Generate All Minterms of the code I/Ps



Function Table

Notice that each output line is the minterm corresponding to the input code, e.g. D_5 is m_5

For this input combination, Which is the Output that will be 1?

Dec. Code	Inputs A_2	A_1	A_0	Outputs D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0	0	0	0
3	0	1	1	0	0	0	1	0	0	0	0
4	1	0	0	0	0	0	0	1	0	0	0
5	1	0	1	0	0	0	0	0	1	0	0
6	1	1	0	0	0	0	0	0	0	1	0
7	1	1	1	0	0	0	0	0	0	0	1

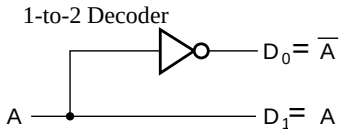
3-to-8 Line decoder

Hierarchical Decoder Design (Decoder Expansion)

Using simpler decoders to build more complex ones

- **1-to-2-Line Decoder**
  1 to 2^1 - The simplest Decoder
- **2-to-4-Line Decoder**

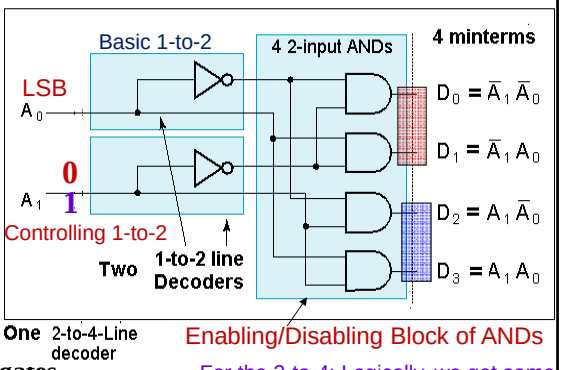
A	D_0	D_1
0	1	0
1	0	1



MSB ←

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

Decoder Expansion



- **Note that the 2-4-line made up of 2 1-to-2-line decoders and 4 AND gates.**

For the 2-to-4: Logically, we get same Circuit of the formal design (the 4 minterms!)

We better design each decoder to have its own enable input!

n-to-m Decoder with Enable (EN)

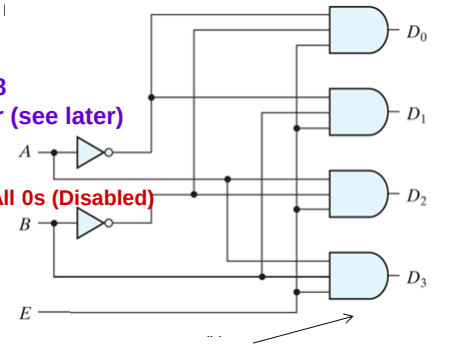
- Use the EN input to open/close the minterm-forming gates
- See truth table below for function
  - Note use of X's to denote both 0 and 1 at the inputs
  - Combination containing two X's represent four input binary combinations

2-to-4 with Enable:
Disabled decoder has all outputs = 0
→ Allows simple expansion e.g. to 3-to-8
→ Also, will use to make a demultiplexer (see later)

	EN	A_1	A_0	D_0	D_1	D_2	D_3
Decoder is disabled	0	X	X	0	0	0	0
Decoder Enabled: Normal Operation	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

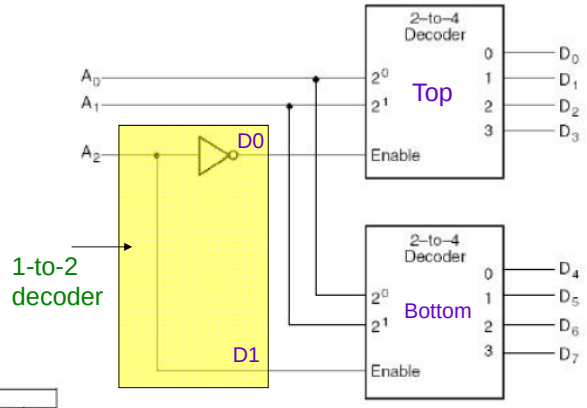
(a)

2-to-4 with Enable



Decoder Expansion
Example: 3-to-8 from two (2-to-4 with EN)

• Using two 2-to-4 decoders
& one 1-to-2 decoder



Top 2-to-4 decoder

Dec. Code	Inputs A_2	A_1	A_0	Outputs D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0	0	0	0
3	0	1	1	0	0	0	1	0	0	0	0
4	1	0	0	0	0	0	0	1	0	0	0
5	1	0	1	0	0	0	0	0	1	0	0
6	1	1	0	0	0	0	0	0	0	1	0
7	1	1	1	0	0	0	0	0	0	0	1

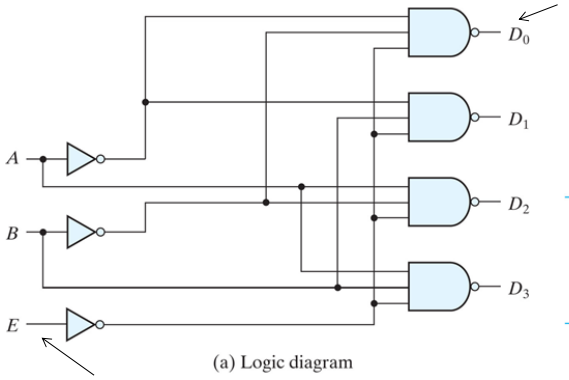
Bottom 2-to-4 decoder

3-to-8 Line decoder

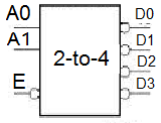
2-to-4 Decoder With Enable
Polarities for E and O/Ps can be reversed

Note: Here polarity of the Enable I/P and the decoder outputs is reversed….

NAND Gates (**not ANDs**):
→ Selected Decoder O/P = 0 (not 1)



(a) Logic diagram



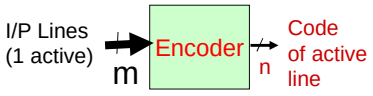
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

Correction

Decoder is activated with E = 0 - Not 1

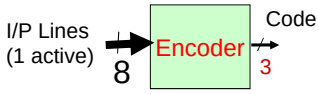
4. Encoding



- Encoding - the opposite of decoding
- An encoder converts *m* input lines to an *n*-bit output code where $2 \leq m \leq 2^n$ - such that each activated input line produces the corresponding unique output code
- If the input lines have exactly only one active line (e.g. at logic 1) → output is the binary code corresponds to the position of that input (exact opposite of decoder)
  - If not, we need to consider priority

What is the smallest Encoder?

Example: 8-to-3 Encoder

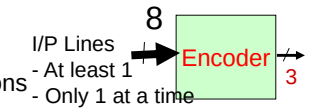


- Inputs (D_0, …, D_7): 8 lines corresponding to digits 0 through 7
- Outputs (A_2, A_1, A_0) : 3 bits of the binary code
- Function: If the ith input line D_i is a 1, the output (A_2, A_1, A_0) = the binary code for i
- Initially for simplicity:

  Assume that at least one and only one of the 8 inputs is active at any given time. So we have only 8 valid input combinations out of the $2^8 = 256$ possible combinations. Remaining rows are don't care → simplifies the design considerably

Example: 8-to-3 Encoder

Octal-to-binary encoder restrictions



Assuming only 1 (and at least 1) Input line being active at a time

The 256-row truth table is thus reduced to **only these 8 valid rows**

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

- **Equations:**

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

Ambiguities arise if conditions above are not met:

1. O/P = 000 for: D0 active, **also** for no active line
2. If two lines become active simultaneously, O/P code is wrong (represents neither of them!) e.g. if both D3 and D6 are active, output=111 (wrong) → So, We need priority encoding.!

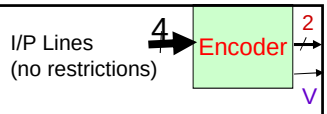
53

Priority Encoder

- If none or more than one input line is active (at logic 1), then the encoder just described does not work properly
- An encoder that can accept all possible combinations of input values and still produces meaningful output is called a *priority encoder*
- Among all the 1s that appear at the I/Ps simultaneously, it selects the "1" I/P having the highest designated priority and produces its corresponding binary code- ignoring all other lower-priority "1"s that may exist with it.
- So the code generated is for the input of the highest priority that is active (=1) (all higher priority inputs = 0), regardless of the state of all lower priority inputs

4-to-2 Priority Encoder

Example: 3 ← Higher Priority ← 0



K-map cells Included

0
1
2,3
4,5,6,7
8,9,10,11,12,13,14,15

Should be 2^4 = 16 rows/cells in total!

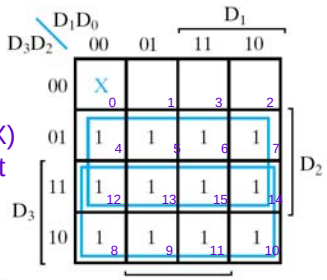
Note the difference Between a don't care (X) in the inputs and a don't care (X) in the outputs!

Higher Priority ← LSB

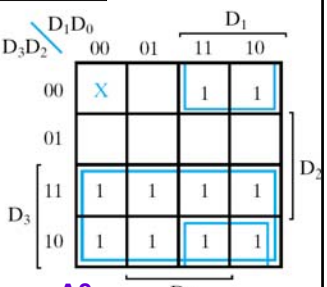
Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Higher: 0s ← 1 → Lower: Xs

"Valid" O/P (= OR of all I/Ps)



A1 map $A_1 = D_2 + D_3$



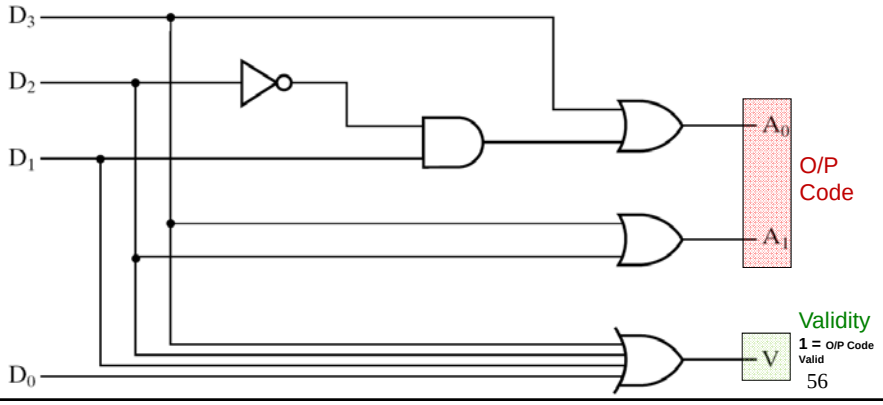
A0 map $A_0 = D_3 + D_1\overline{D}_2$

4-to-2 Priority Encoder Example

$A_1 = D_2 + D_3$

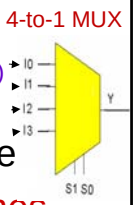
$A_0 = D_3 + D_1\overline{D}_2$

$V = D_0 + D_1 + D_2 + D_3$

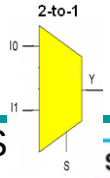


5. Selecting (Multiplexing): 2^n-to-1

- A multiplexer (**MUX**) selects one of 2^n data input lines based on an n-bit address, directing it to one output line
- A typical multiplexer has:
  - 2^n Information inputs ($I_{(2^n-1)}$, … I_0) (to select **from**)
  - *n* **S**elect (control or address) inputs (S_{n-1}, … S_0) (to select **with**))
  - 1 Information output Y (to select **to**)
- Will implement it using a decoder, see next slide
- MUX selection circuits can be **duplicated** m times (with the same selection controls) to provide m-wide data widths, e.g. select one of four input bytes using 8-wide 4-to-1 MUX



The Simplest Multiplexer
n = 1 → 2^1-to-1 MUX



- The single selection variable S has two values:
  - S = 0 selects input I_0
  - S = 1 selects input I_1

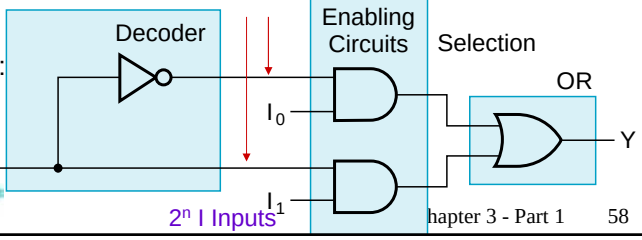
Truth Table

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

- 3-input K-map optimization gives the output equation:

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

- The circuit:
- Can also be seen As:
  - **1-to-2 decoder**
  - + Enabling
  - + Selection



Example: 4-to-1-line Multiplexer, Using

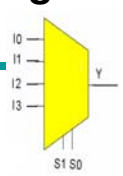
Size of the Select Inputs = Log_2 (4)

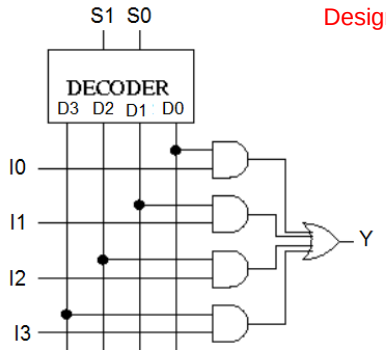
- 2-to-4 decoder +

  4 × 2-I/P AND-OR for Enabling/Selection

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

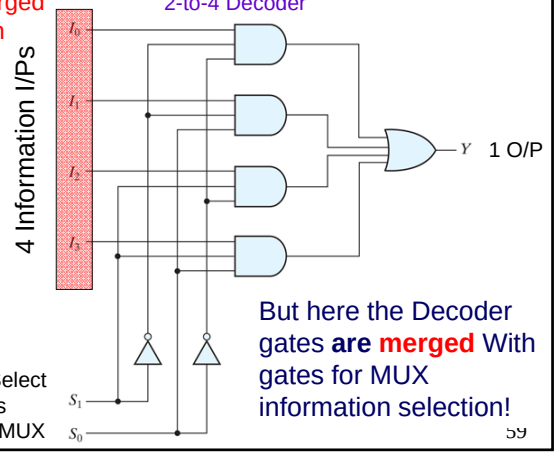
Function table





Or merged Design

2-to-4 Decoder



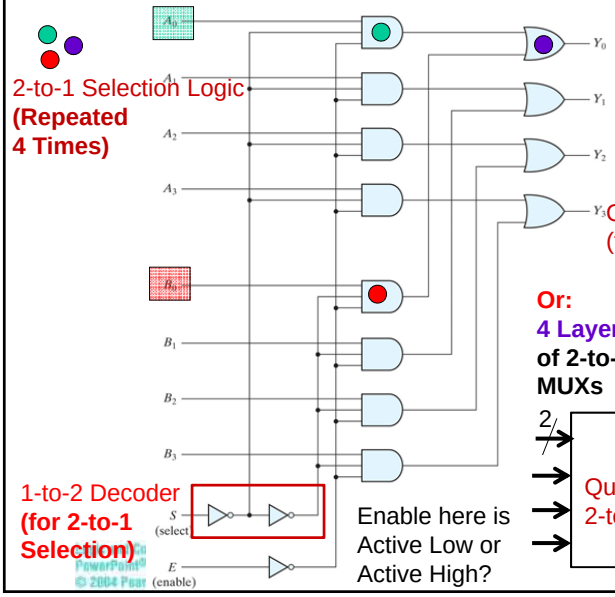
But here the Decoder gates **are merged** With gates for MUX information selection!

59

1. Duplicating the MUX in Width: m-wide 2^n-to-1 MUX

Example: Quad 2-to-1 MUX with E

Selection Capability Is the same Only widths extended

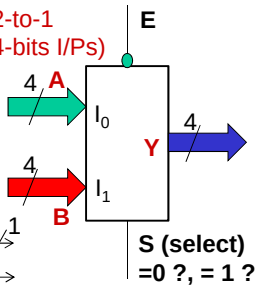


2-to-1 Selection Logic **(Repeated 4 Times)**

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

Function table

One 2-to-1 (two 4-bits I/Ps)



Or: 4 Layers of 2-to-1 MUXs

1-to-2 Decoder **(for 2-to-1 Selection)**

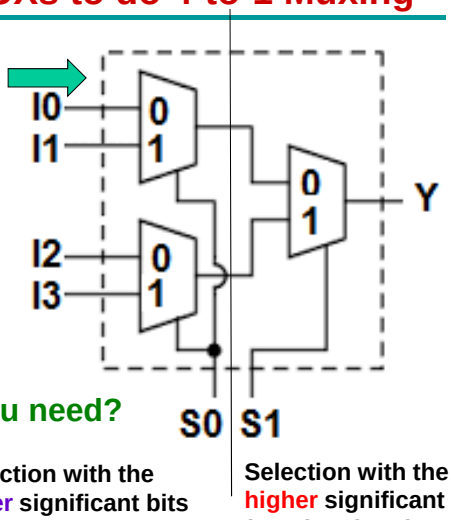
Enable here is Active Low or Active High?

Importance of labeling inputs and outputs

2. MUX Expansion: Expanding the selection capability

Example: Using 2-to-1 MUXs to do 4-to-1 Muxing

S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

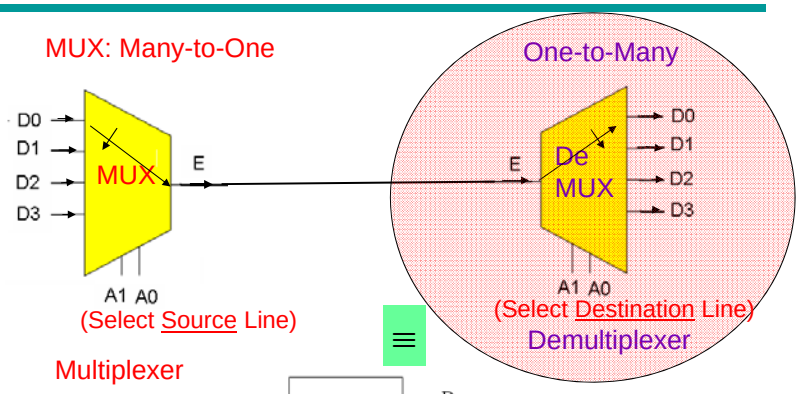


How many 2-to-1 MUXs do you need?

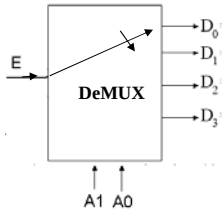
Selection with the lower significant bits (I0 or I1)/(i2 or I3)

Selection with the higher significant bits (I0, I1) or (I2,I3)

3. Demultiplexer- Opposite of Multiplexer



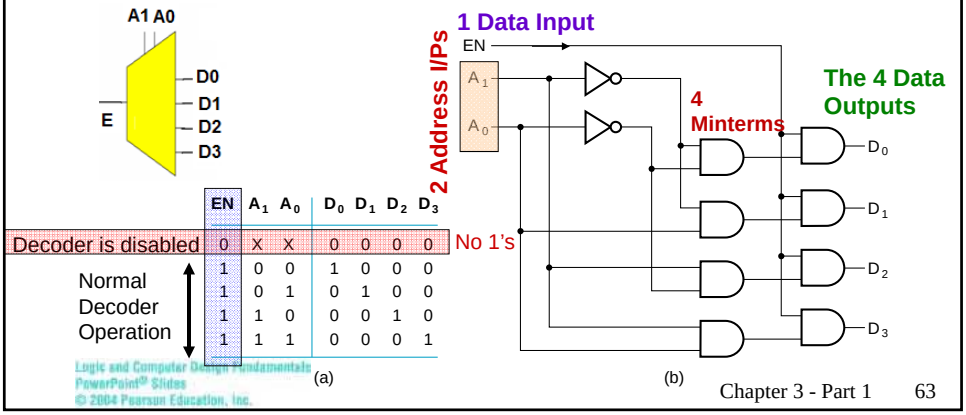
A decoder with **Enable** is Called a: Decoder/Demultiplexer



A device that moves data arriving on a single input (E) to one of m outputs (Ds) determined by the value of ($\log_2$ m) select inputs (As)

2-to-4 Decoder with Enable = 1-to-4 Demultiplexer!

- From Truth Table, decoder can be viewed as **distributing** the value of the EN input to 1 of 4 outputs
- From this perspective, it is a Demultiplexer **!**



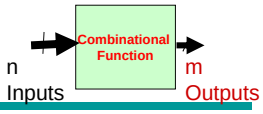
Implementing Combinational Functions Using Functional Blocks

- Two implementation techniques from the SOm canonical form (no simplification):
  - Using a Decoder + OR gates
  - Using a Multiplexer + Inverters (if needed)

  We always said **Canonical forms give complex implementations!**

  But now we have most of the complexity "hidden" inside the "ready-made" function block !! (e.g. decoder or MUX) → Utilize it!

1. Using a Decoder + OR Gates: From Canonical Form: Truth Table (or Σ m Form)



- Functions of *n inputs and m* outputs:
  - Specification:
    → As a Truth Table (has n input columns and m output columns)
    → or m SOm expressions [Σm(…)]
  - Implementation requires:
    → One *n*-to-2^n-line decoder
    → *m* "OR" gates: one gate for each output
  - Procedure:
    → From the truth table:
    For each output: For a '1' in truth table row (i), connect the corresponding D_i output of the decoder to the OR of that output
    Or → From the m minterm expression [Σm(…)]:
    Connect the decoder D_i outputs corresponding to the minterms of each output to the OR of that output

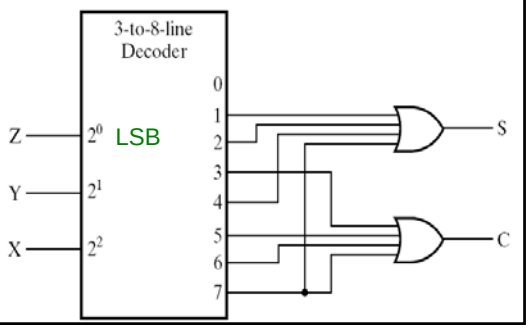
Decoder + OR Gates: Example 1-bit adder (with carries at I/P and O/P)

Truth Table for 1-bit Binary Adder

3 Inputs			LSB	2 Outputs	
X	Y	Z		C	S
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	2	0	1
0	1	1	3	1	0
1	0	0	4	0	1
1	0	1	5	1	0
1	1	0	6	1	0
1	1	1	7	1	1

3 I/Ps and 2 O/Ps: 2 SOm expressions
We need:
→ 3-to-2^3 Decoder
→ 2 "OR" gates of appropriate # of inputs

$$S(x, y, z) = \Sigma_m(1,2,4,7)$$
$$C(x, y, z) = \Sigma_m(3,5,6,7)$$



Larger # of 1's require larger ORs. If so, Consider expressing F and using a NOR instead of the OR!

Using Multiplexers: from Truth Table, or canonical Form

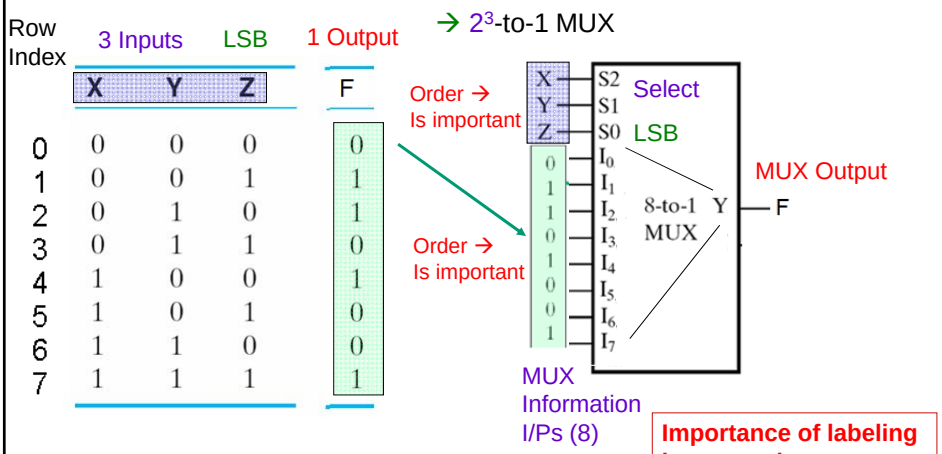
- Implementing a logic circuit of *n inputs and m* outputs requires:
  - Specification: Truth table, or Som or PoM forms
  - **Implementation: Use m x 2^n-to-1 multiplexer**

Example: 5 input, 3 output circuit: Need 3 x 2^5-to-1 MUX

- Design:
  - In the same order they appear in the truth table:
    - Apply the n **input variables** to the MUX **select** inputs S_{n-1}, … , S_0 (i.e. Observe bit significance, i.e. LS variable goes to S0)
    - Label the outputs of the multiplexer with the output variables
  - Value-fix the I **inputs** to the multiplexer using the values from the truth table. For don't cares, use either 0 or 1.

Using Multiplexers: Example:

1. Conventional approach, n inputs → Use 2^n-to-1 MUX



Importance of labeling inputs and outputs

What if we have 2 outputs?
We use a dual (i.e. x 2) 8-to-1 MUX

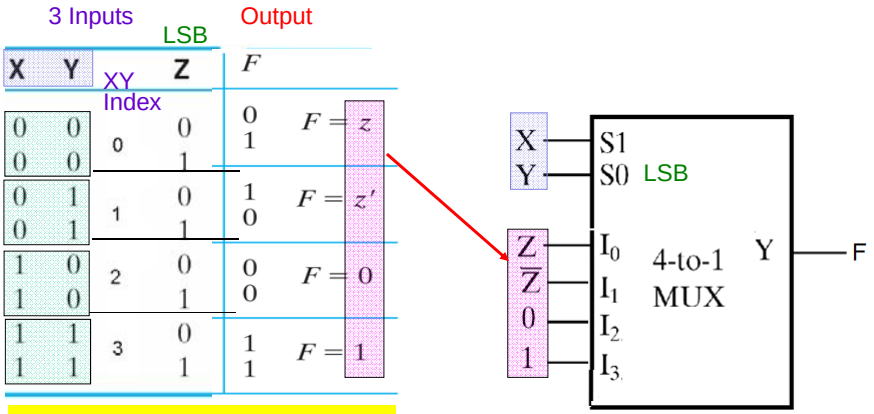
Using Multiplexers: Example: Full adder-

2. Smarter approach uses a smaller MUX:

n input variables → Use 2^{n-1}-to-1 MUX (1/2 previous size)

→Use ½ the MUX size needed earlier: 2^2-to-1 MUX

→Connect MS 2 input variables to select, and express F as 1, 0, Z, or $\overline{Z}$ for each value of XY and apply to the I inputs of the MUX.



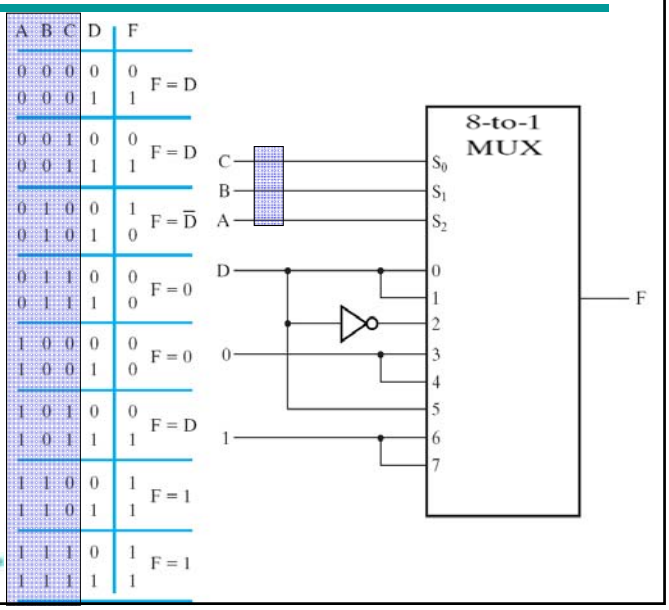
What if we have 2 outputs?
We use a dual 4-to-1 MUX

Using Multiplexers: Example

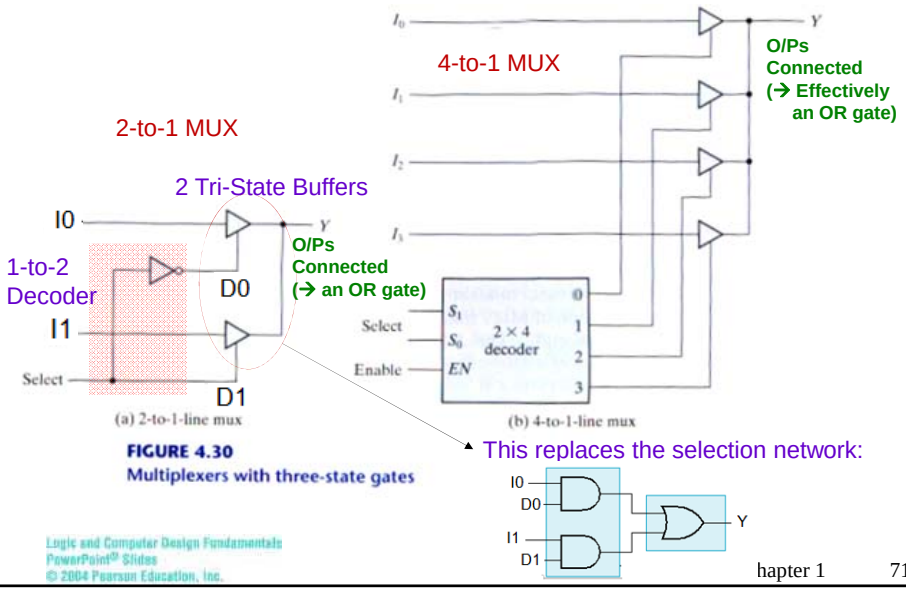
F (A,B,C,D) = Σm(1,3,4,11,12,13,14,15)

→16 rows in truth table → 16-to-1 MUX (conventional approach)

→But using the more efficient approach … will use only an 8-to-1 MUX + 1 inverter



Designing MUXes
Using 3-State Buffers for Selection



Design Examples Using MSI Combinational Functional Blocks

1. Adding three 4-bit numbers
2. Adding two 16-bit numbers using 4-bit adders
3. Building a 4-to-16 Decoders using several 2-to-4 Decoders (with Enable)
4. Selecting the larger of two 4-bit numbers
5. BCD to Excess-3 Code Converter using binary decoders and encoders only
6. Building multi-function combinational circuit (e.g. a cct that adds, subtracts, doubles, etc. according to a set of function select I/Ps)

Important in these problems: Must Label Clearly all inputs/Outputs of all function blocks

Example 1: Adding three unsigned 4-bit digits

Problem: Add three 4-bit numbers (X, Y, Z) using 2 standard MSI 4-bit adders

Solution:

Let the numbers be $X_3X_2X_1X_0$, $Y_3Y_2Y_1Y_0$, $Z_3Z_2Z_1Z_0$,

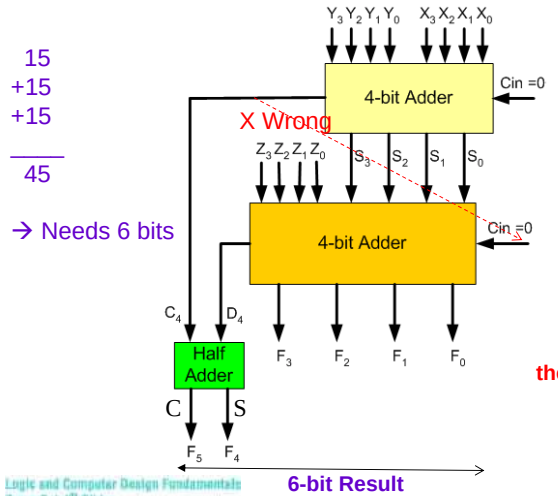
$$
\begin{array}{rl}
 & X_3X_2X_1X_0 \\
+ & Y_3Y_2Y_1Y_0 \\
\hline
C_4 & S_3S_2S_1S_0
\end{array}
\qquad
\begin{array}{l}
\\
\\
\hline
D_4\ \ F_3F_2F_1F_0
\end{array}
\qquad
\begin{array}{l}
\\
+\ Z_3Z_2Z_1Z_0 \\
\hline
\\
\end{array}
$$

Note: C_4 and D_4 are generated in digit position 4. They must be added in the same position to generate the most significant bits of the result

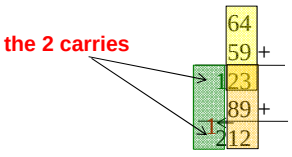
Adding three unsigned 4-bit digits

15
+15
+15
____
45

→ Needs 6 bits



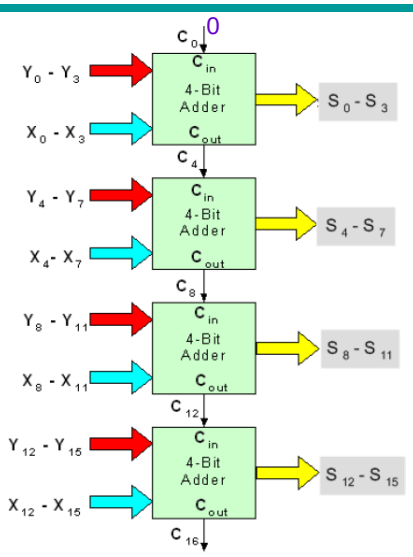
Decimal Example:
64 + 59 + 89



Example 2: Adding two 16-bit numbers using a number of 4-bit adders

Solution:

Four 4-bit adder blocks are connected in cascade, with carries rippling in between



Example 3:
Design a 4-to-16 Decoder
Using a number of 2-to-4 Decoders (each with Enable)

Problem: Design a 4-to-16 Decoder using a number of 2-to-4 Decoders with Enable

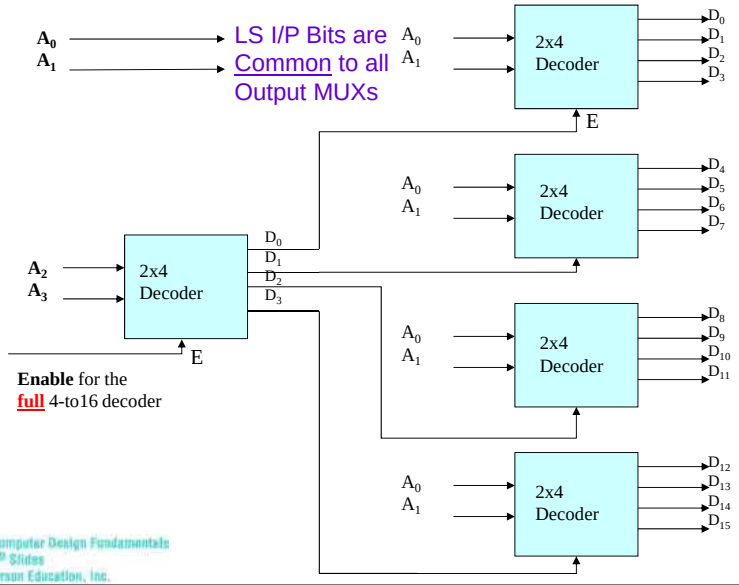
Solution:

- Four 2-to-4 decoders are fed with A1 A0 (in parallel) to generate the 16 output lines
- The remaining 2 input lines A3 A2 drive a 5th 2-to-4 decoder to select (Enable) one of the 4 decoders to perform decoding for its group of 4 lines

Select 1 of the 4 2-to-4 decoders | Common to all 4 2-to-4 decoders

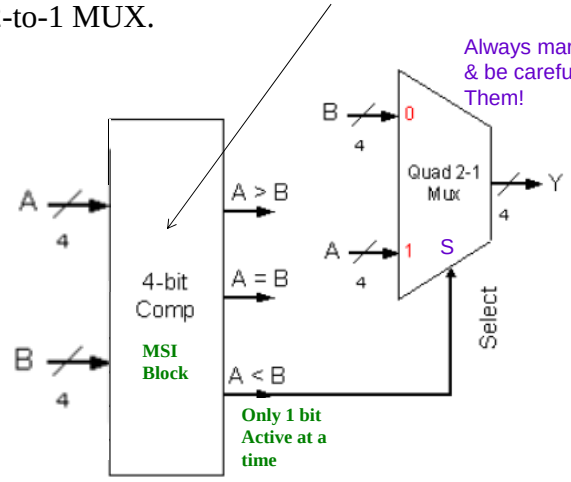
	A_3	A_2	A_1	A_0	Active Output
$A_3A_2 = 00$	0	0	0	0	D_0
	0	0	0	1	D_1
	0	0	1	0	D_2
	0	0	1	1	D_3
$A_3A_2 = 01$	0	1	0	0	D_4
	0	1	0	1	D_5
	0	1	1	0	D_6
	0	1	1	1	D_7
$A_3A_2 = 10$	1	0	0	0	D_8
	1	0	0	1	D_9
	1	0	1	0	D_{10}
	1	0	1	1	D_{11}
$A_3A_2 = 11$	1	1	0	0	D_{12}
	1	1	0	1	D_{13}
	1	1	1	0	D_{14}
	1	1	1	1	D_{15}

4-to-16 Decoder



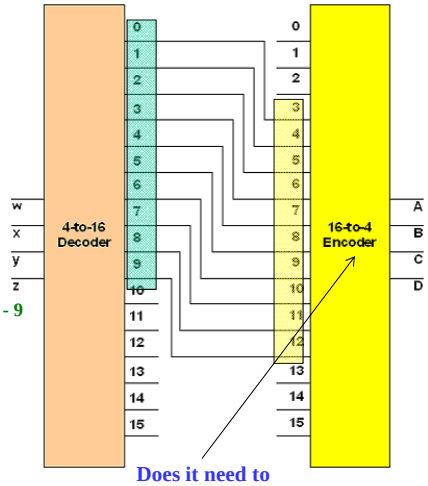
Example 4:
Hardware that compares two unsigned 4-bit numbers and selects (passes) the smaller of the two to the O/P

Solution: We will use a magnitude comparator and a Quad 2-to-1 MUX.



Example 5: BCD to Excess-3 Code Converter using a decoder and straight binary encoder

Index	BCD w	x	y	z	Excess-3 A	B	C	D
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	0
9	1	0	0	1	1	1	0	0

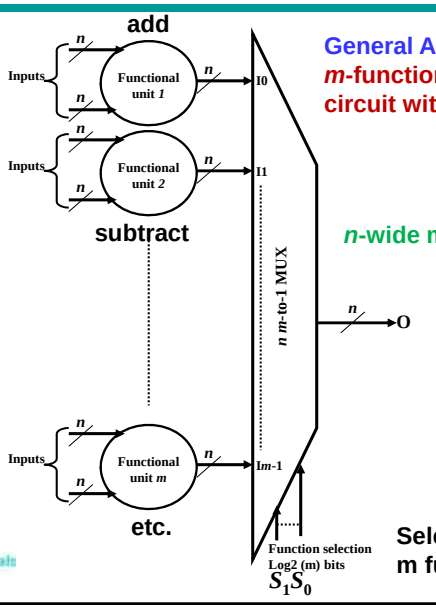


BCD: 0 - 9

Does it need to be a Priority Encoder?

Excess-3: 3 - 12

Example 6: Building multiple-function combinational circuit (e.g. add, subtract, Max, ..)



General Architecture of an *m*-function combinational circuit with *n*-bit data width

n-wide m-to-1 MUX

Select 1 of the m functions

Example Function Table

S_1S_0	O is equal to
00	Max(A, B)
01	Min(A, B)
10	$2 \times A$
11	$A - B$