

UNIT 3

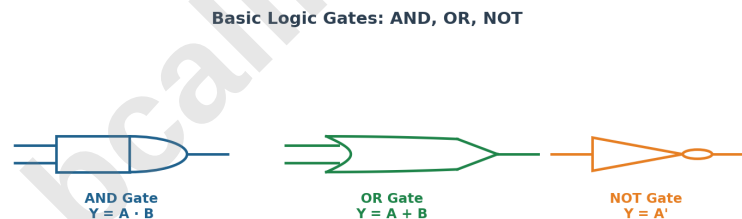
Combinational Logic Design

(16 Hours) — Digital Logic (CACSI05)

A combinational logic circuit is one whose output at any instant depends only on the present combination of inputs, with no memory of past inputs. Such circuits are built entirely from logic gates connected without any feedback path. This unit builds up from the basic logic gates, through Boolean algebra and simplification techniques (algebraic and Karnaugh map methods), to standard combinational building blocks — encoders, decoders, adders, multiplexers, demultiplexers — and finally to programmable logic devices that let engineers implement custom logic functions in a single reconfigurable chip.

3.1 Basic Logic Gates: NOT, OR, and AND

Logic gates are the elementary building blocks of digital circuits. Each gate performs a specific Boolean operation on one or more binary inputs to produce a single binary output. The three basic gates — NOT, OR, and AND — form the foundation from which every other logic function can be derived.



(A) NOT Gate (Inverter)

The NOT gate has a single input and produces the logical complement (inverse) of that input: $Y = A'$. It is also called an inverter.

A	$Y = A'$
0	1
1	0

(B) OR Gate

The OR gate produces a HIGH output if any one or more of its inputs are HIGH: $Y = A + B$. It performs Boolean addition.

A	B	$Y = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

(C) AND Gate

The AND gate produces a HIGH output only when all of its inputs are HIGH: $Y = A \cdot B$. It performs Boolean multiplication.

A	B	$Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Example 3.1 — Evaluate $Y = (A \cdot B) + C'$ for $A=1, B=0, C=1$

$$A \cdot B = 1 \cdot 0 = 0$$

$$C' = 1' = 0$$

Solution

$$Y = 0 + 0 = 0$$

Key Takeaways — Section 3.1

- ✓ NOT inverts a single input; OR is HIGH when any input is HIGH; AND is HIGH only when all inputs are HIGH.
- ✓ These three gates are functionally complete — any Boolean function can be expressed using only AND, OR, and NOT.

3.2 Universal Logic Gates: NOR and NAND

NAND (NOT-AND) and NOR (NOT-OR) gates are formed by inverting the output of an AND or OR gate respectively. They are called universal gates because either gate type, used alone and wired appropriately, can implement all three basic operations (NOT, AND, OR) — and therefore any Boolean function whatsoever. This property makes them extremely valuable in IC fabrication, since a chip needs only one type of gate.

Universal Logic Gates: NAND and NOR



Both NAND and NOR are 'universal' — any logic function (AND, OR, NOT) can be built using only one gate type.

Figure 3.2 — NAND and NOR gate symbols.

A	B	NAND: $Y = (A \cdot B)'$	NOR: $Y = (A + B)'$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

(A) Realizing Basic Gates Using Only NAND Gates

Function	NAND-Only Implementation
NOT A	Tie both inputs of a NAND gate together: $(A \cdot A)' = A'$
A AND B	NAND followed by another NAND used as inverter: $((A \cdot B)')' = A \cdot B$
A OR B	Invert A and B individually first, then NAND them (by De Morgan's law)

(B) Realizing Basic Gates Using Only NOR Gates

Function	NOR-Only Implementation
NOT A	Tie both inputs of a NOR gate together: $(A + A)' = A'$

Function	NOR-Only Implementation
A OR B	NOR followed by another NOR used as inverter: $((A+B)')' = A+B$
A AND B	Invert A and B individually first, then NOR them (by De Morgan's law)

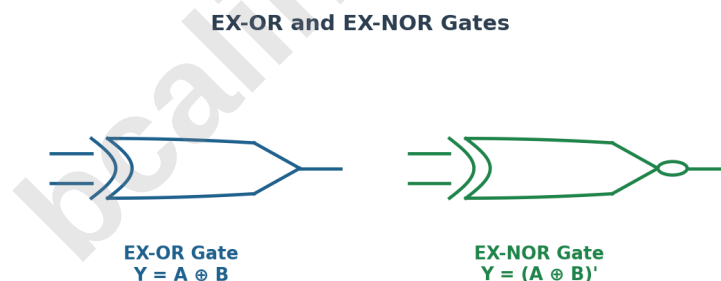
Note: Because fabricating many identical gates on a chip is simpler and cheaper than mixing gate types, real-world ICs are frequently built entirely from NAND or entirely from NOR gates.

Key Takeaways — Section 3.2

- ✓ $\text{NAND} = (A \cdot B)'$; $\text{NOR} = (A+B)'$ — both are formed by inverting AND/OR outputs.
- ✓ NAND and NOR are each functionally complete (universal) — any logic circuit can be built using only one gate type.
- ✓ This universality simplifies IC manufacturing since only a single gate type needs to be fabricated.

3.3 EX-OR and EX-NOR Gates

The Exclusive-OR (EX-OR) and Exclusive-NOR (EX-NOR) gates are derived gates widely used in arithmetic circuits (adders), comparators, and parity generation.



EX-OR output is HIGH only when inputs differ; EX-NOR output is HIGH only when inputs match.

Figure 3.3 — EX-OR and EX-NOR gate symbols.

(A) EX-OR Gate

$Y = A \oplus B = A \cdot B' + A' \cdot B$. The output is HIGH only when the inputs are different.

A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1

A	B	$Y = A \oplus B$
1	1	0

(B) EX-NOR Gate

$Y = (A \oplus B)' = A \cdot B + A' \cdot B'$. The output is HIGH only when the inputs are the same — hence it is also known as an equivalence or coincidence gate.

A	B	$Y = (A \oplus B)'$
0	0	1
0	1	0
1	0	0
1	1	1

Example 3.2 — Evaluate $Y = A \oplus B \oplus C$ for $A=1, B=1, C=0$

$$A \oplus B = 1 \oplus 1 = 0$$

$$\text{Result} \oplus C = 0 \oplus 0 = 0$$

Solution

$$Y = 0$$

Key Takeaways — Section 3.3

- ✓ EX-OR output is HIGH when inputs differ; EX-NOR output is HIGH when inputs match.
- ✓ EX-OR is the core building block of binary adders (produces the SUM bit).
- ✓ EX-NOR is used in equality comparators and Gray-code related logic.

3.4 Boolean Algebra

Boolean algebra is a mathematical system, introduced by George Boole, that deals with binary variables and logical operations. It provides the algebraic rules used to analyze and simplify digital logic circuits without resorting to truth tables.

3.4.1 Postulates and Theorems

Law / Theorem	OR Form	AND Form
Identity	$A + 0 = A$	$A \cdot 1 = A$
Null (Dominance)	$A + 1 = 1$	$A \cdot 0 = 0$
Idempotent	$A + A = A$	$A \cdot A = A$
Complement	$A + A' = 1$	$A \cdot A' = 0$
Commutative	$A + B = B + A$	$A \cdot B = B \cdot A$
Associative	$(A+B)+C = A+(B+C)$	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
Distributive	$A+(B \cdot C) = (A+B) \cdot (A+C)$	$A \cdot (B+C) = A \cdot B + A \cdot C$
Absorption	$A + A \cdot B = A$	$A \cdot (A+B) = A$
Involution	$(A')' = A$	—

De Morgan's Theorems

Theorem	Statement
De Morgan's 1st	$(A + B)' = A' \cdot B'$ — complement of a sum equals the product of complements
De Morgan's 2nd	$(A \cdot B)' = A' + B'$ — complement of a product equals the sum of complements

Example 3.3 — Simplify $Y = A \cdot B + A \cdot B'$

$Y = A \cdot (B + B')$ (Distributive law)

Solution

$B + B' = 1$ (Complement law) $\Rightarrow Y = A \cdot 1 = A$

Example 3.4 — Simplify $Y = (A + B) \cdot (A + B')$

$Y = A + (B \cdot B')$ (Distributive law, dual form)

Solution

$$B \cdot B' = 0 \text{ (Complement law)} \Rightarrow Y = A + 0 = A$$

Example 3.5 — Simplify $Y = A + A'B$ using absorption

$$Y = (A + A') \cdot (A + B) \text{ (Distributive law)}$$

Solution

$$A + A' = 1 \Rightarrow Y = 1 \cdot (A+B) = A + B$$

3.4.2 Canonical Forms — Simplification of Logic Functions

Any Boolean function can be expressed in a standard (canonical) form as a sum of minterms (Sum-of-Products, SOP) or a product of maxterms (Product-of-Sums, POS).

Term	Definition
Minterm	A product term in which every variable of the function appears exactly once, either true or complemented (e.g., $A \cdot B' \cdot C$ for a 3-variable function)
Maxterm	A sum term in which every variable appears exactly once, either true or complemented (e.g., $A+B'+C$)
SOP (Sum of Products)	A Boolean expression written as an OR of AND terms, e.g., $Y = A'B + AB' + ABC$
POS (Product of Sums)	A Boolean expression written as an AND of OR terms, e.g., $Y = (A+B) \cdot (A'+C)$

Example 3.6 — Express $Y(A,B,C) = \Sigma m(1,3,6,7)$ in SOP canonical form

$$m1 = A'B'C, m3 = A'BC, m6 = AB C', m7 = ABC$$

Solution

$$Y = A'B'C + A'BC + ABC' + ABC$$

Example 3.7 — Simplify $Y = AB + AB' + A'B$

$$\text{Group: } AB + AB' = A(B+B') = A \cdot 1 = A$$

Solution

$$Y = A + A'B = A + B \text{ (using absorption, Example 3.5 result)} \Rightarrow Y = A + B$$

Key Takeaways — Section 3.4

- ✓ Boolean postulates (identity, complement, commutative, associative, distributive, absorption) allow algebraic simplification of logic expressions.
- ✓ De Morgan's theorems convert between AND/OR forms and are essential for NAND/NOR-only implementations.
- ✓ Any function can be written in canonical SOP (sum of minterms) or POS (product of maxterms) form.
- ✓ Algebraic simplification reduces gate count but becomes cumbersome for functions with many variables — this motivates the Karnaugh map method (Section 3.5).

bcalink.com

3.5 Simplification of Logic Functions Using Karnaugh Map

A Karnaugh map (K-map) is a graphical method for simplifying Boolean expressions without lengthy algebraic manipulation. Cells are arranged so that any two physically adjacent cells (including wrap-around edges) differ in exactly one variable — achieved by ordering row/column labels in Gray code sequence.

4-Variable Karnaugh Map (Gray-Code Ordered)

AB	CD			
	00	01	11	10
00	m0	m1	m3	m2
01	m4	m5	m7	m6
11	m12	m13	m15	m14
10	m8	m9	m11	m10

Figure 3.4 — Blank 4-variable K-map with minterm numbering (Gray-code ordered).

K-Map Grouping Rules

Rule	Description
Group sizes	Groups must contain 1, 2, 4, 8, 16 ... (powers of 2) adjacent 1-cells
Maximize size	Groups should be made as large as possible to eliminate the most variables
Overlap allowed	The same cell may be used in more than one group if it helps form larger groups
Wrap-around	Cells on opposite edges/corners of the map are considered adjacent
Minimum groups	Cover every 1-cell using the fewest possible number of groups

Example 3.8 — Simplify $Y(A,B,C) = \Sigma m(0,1,2,3,6)$ using a 3-variable K-map

Plot 1's at minterms 0, 1, 2, 3, 6 on a 3-variable map (rows=A, cols=BC).

Group 1: m_0, m_1, m_2, m_3 (entire row $A=0$) $\rightarrow$ simplifies to A'

Group 2: m_2, m_6 (adjacent pair) $\rightarrow$ simplifies to BC'

Solution

$$Y = A' + BC'$$

Example 3.9 — Simplify $Y(A,B,C,D) = \Sigma m(0,2,8,10)$ using a 4-variable K-map

All four minterms have $B=0$ and $D=0$ in common (check each: 0000,0010,1000,1010)

These four cells form a valid 4-cell wrap-around group (corners of the map)

Solution

$$Y = B'D'$$

3.5.1 Analysis of SOP and POS Expressions

K-maps can simplify a function into either Sum-of-Products (grouping 1-cells) or Product-of-Sums form (grouping 0-cells). Grouping the 1's on the map and reading each group gives the minimal SOP expression. Grouping the 0's instead and then complementing gives the minimal POS expression — useful when a NOR-based implementation is preferred.

Example 3.10 — Find the POS form of $Y(A,B,C) = \Sigma m(0,1,2,3,6)$

The 0-cells are at minterms 4, 5, 7.

Group $m_4, m_5 \rightarrow AB'$; Group $m_5, m_7 \rightarrow AC$

Complement of the 0-groups' SOP gives the POS form of Y .

Solution

$$Y' = AB' + AC \Rightarrow Y = (A' + B) \cdot (A' + C') \text{ (POS form)}$$

Note: SOP and POS forms describe the same function — the choice depends on which gate type (AND-OR vs. OR-AND, or NAND-NAND vs. NOR-NOR) is more convenient for the target hardware.

Key Takeaways — Section 3.5

- ✓ K-maps exploit Gray-code adjacency to visually identify groups of minterms that can be combined.
- ✓ Larger groups (powers of 2: 1, 2, 4, 8...) eliminate more variables and give simpler expressions.
- ✓ Grouping 1-cells yields the minimal SOP form; grouping 0-cells (then complementing) yields the minimal POS form.
- ✓ K-maps are practical for up to about 4–5 variables; beyond that, computer-based methods (e.g., Quine–McCluskey) are used.

bcalink.com

3.6 Implementation of Combinational Logic Functions

3.6.1 Encoders and Decoders

Decoders and encoders are complementary combinational circuits that convert between a binary code and a one-hot (single active line) representation.

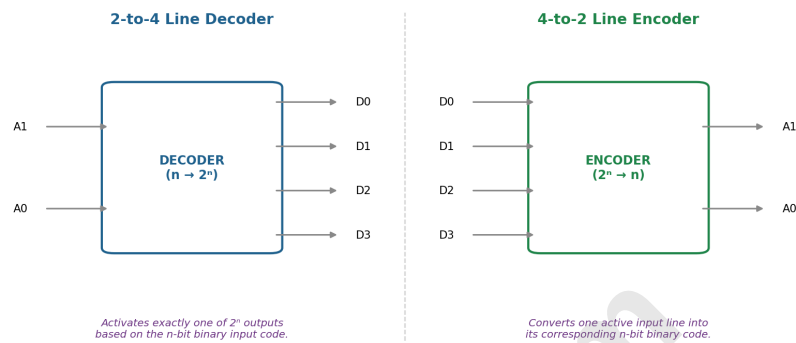


Figure 3.5 — Block diagrams of a 2-to-4 decoder and a 4-to-2 encoder.

(A) Decoder

A decoder converts an n -bit binary input code into one of 2^n mutually exclusive output lines — exactly one output goes HIGH for each unique input combination. Decoders are used for memory address selection, instruction decoding, and demultiplexing.

A1	A0	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Each output is a minterm of the inputs: $D0=A1'A0'$, $D1=A1'A0$, $D2=A1A0'$, $D3=A1A0$.

(B) Encoder

An encoder performs the reverse operation — it converts an active input line (one of 2^n) into its corresponding n -bit binary code. A basic encoder assumes only one input is active at a time; a priority encoder resolves the case where multiple inputs are active simultaneously by encoding only the highest-priority active input.

D0	D1	D2	D3	A1	A0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Note: Priority encoders (e.g., the 74148 IC) are widely used in interrupt-handling systems, where the highest-priority pending interrupt line must be encoded even if several lines are active.

3.6.2 Half Adder and Full Adder

Adders are combinational circuits that perform binary addition — the fundamental arithmetic operation inside every ALU (Arithmetic Logic Unit).

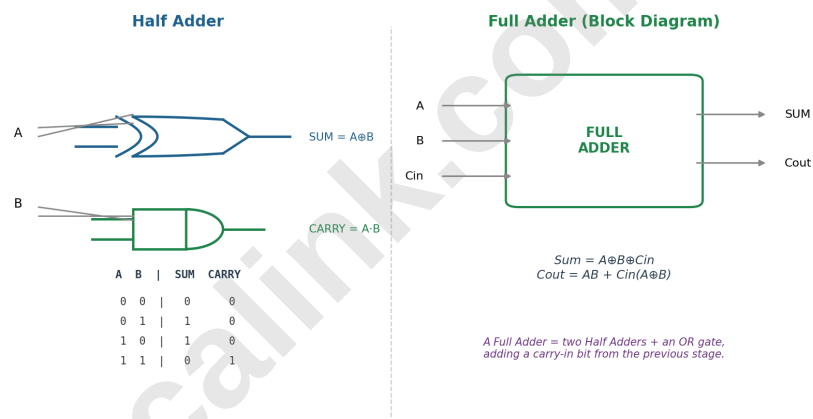


Figure 3.6 — Half adder logic and full adder block diagram.

(A) Half Adder

A half adder adds two single bits (A, B) and produces a SUM and a CARRY, but has no input for a carry from a previous stage.

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$SUM = A \oplus B, CARRY = A \cdot B$$

(B) Full Adder

A full adder extends the half adder by accepting a carry-in (C_{in}) from the previous stage, making it suitable for cascading into multi-bit adders.

A	B	C_{in}	SUM	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$SUM = A \oplus B \oplus C_{in}, C_{out} = A \cdot B + C_{in} \cdot (A \oplus B)$$

Example 3.11 — Full adder output for $A=1$, $B=1$, $C_{in}=1$

$$SUM = 1 \oplus 1 \oplus 1 = 1$$

$$C_{out} = (1 \cdot 1) + 1 \cdot (1 \oplus 1) = 1 + 1 \cdot 0 = 1$$

Solution

$$SUM = 1, C_{out} = 1 \quad (1+1+1 = 11 \text{ in binary } \checkmark)$$

Key Takeaways — Section 3.6

- ✓ A decoder activates one of 2^n outputs based on an n -bit binary code; an encoder does the reverse.
- ✓ Priority encoders resolve simultaneous active inputs by encoding the highest-priority line.
- ✓ A half adder adds 2 bits with no carry-in; a full adder adds 2 bits plus a carry-in, enabling cascading.
- ✓ Full adders are the basic building block of multi-bit parallel binary adders (Section 3.7).

3.7 Implementation of Data Processing Circuits

3.7.1 Multiplexers and Demultiplexers

Multiplexers and demultiplexers are data-routing circuits widely used to reduce the number of physical lines needed to transmit multiple signals.

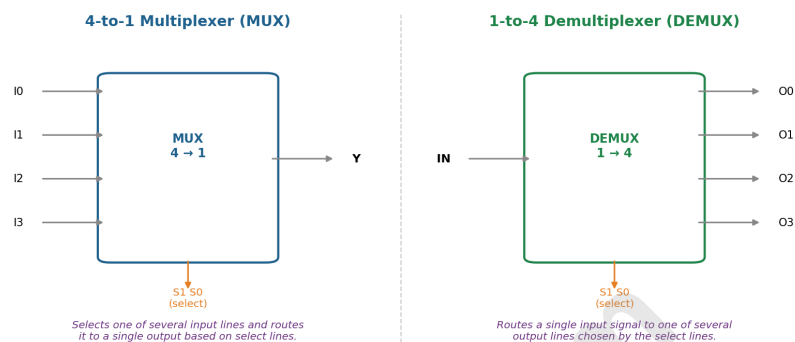


Figure 3.7 — 4-to-1 multiplexer and 1-to-4 demultiplexer block diagrams.

(A) Multiplexer (MUX)

A multiplexer (data selector) selects one of several input lines and routes it to a single output line, based on the binary value applied to its select lines. A 4-to-1 MUX needs 2 select lines ($2^2 = 4$ inputs).

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

$$Y = S1'S0'I0 + S1'S0 I1 + S1 S0' I2 + S1 S0 I3$$

(B) Demultiplexer (DEMUX)

A demultiplexer performs the reverse function of a MUX — it routes a single input signal to one of several output lines, chosen by the select lines.

S1	S0	Active Output
0	0	O0
0	1	O1

S1	S0	Active Output
1	0	O2
1	1	O3

Note: A decoder with an added enable input effectively functions as a demultiplexer — the enable line becomes the data input, and select lines choose the active output.

3.7.2 Parallel Adder, Parity Generator/Checker, and MUX-Based Logic

(A) Parallel Binary Adder

To add multi-bit binary numbers, full adders are cascaded so that the carry-out of each stage feeds the carry-in of the next — called a ripple-carry adder. An n-bit parallel adder requires n full adders (or n–1 full adders plus one half adder for the least significant bit, if no carry-in is needed).

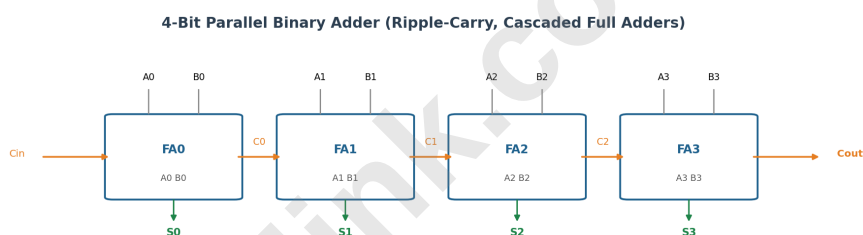


Figure 3.8 — 4-bit parallel binary (ripple-carry) adder built from cascaded full adders.

Example 3.12 — Add $(0110)_2 + (0011)_2$ using a 4-bit parallel adder

A = 0110 (6), B = 0011 (3), Cin(bit0) = 0

Bit0: A0=0, B0=1, Cin=0 → S0=1, C0=0

Bit1: A1=1, B1=1, Cin=0 → S1=0, C1=1

Bit2: A2=1, B2=0, Cin=1 → S2=0, C2=1

Bit3: A3=0, B3=0, Cin=1 → S3=1, C3=0

Solution

Sum = 1001 = $(9)_{10}$ [6+3=9 ✓], final Cout = 0

(B) Parity Generator and Parity Checker

A parity generator uses a tree of EX-OR gates to compute a parity bit from a block of data bits before transmission. A parity checker at the receiver recomputes the parity over the received data plus the parity bit; a non-zero result flags a transmission error.

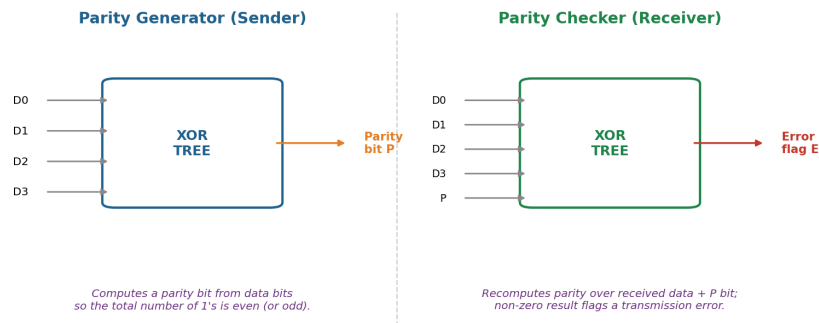


Figure 3.9 — Parity generator (sender) and parity checker (receiver) using EX-OR trees.

Example 3.13 — Generate even parity for 4-bit data 1011

$$P = D0 \oplus D1 \oplus D2 \oplus D3 = 1 \oplus 0 \oplus 1 \oplus 1$$

Solution

$P = 1$ (total 1's including P becomes 4 → even ✓)

(C) Implementing Logic Functions Using Multiplexers

Any Boolean function of n variables can be implemented using a 2^{n-1} -to-1 MUX: apply $n-1$ variables to the select lines, and connect each data input to 0, 1, the remaining variable, or its complement, according to the function's truth table.

Example 3.14 — Implement $Y(A,B,C) = \sum m(1,3,5,6)$ using a 4-to-1 MUX (select=A,B)

For each combination of A,B, examine what Y equals as C varies:

A=0,B=0: minterms 0,1 → Y=0 when C=0, Y=1 when C=1 → input = C

A=0,B=1: minterms 2,3 → Y=0 when C=0, Y=1 when C=1 → input = C

A=1,B=0: minterms 4,5 → Y=0 when C=0, Y=1 when C=1 → input = C

A=1,B=1: minterms 6,7 → Y=1 when C=0, Y=0 when C=1 → input = C'

Solution

MUX data inputs: I0=C, I1=C, I2=C, I3=C' (select lines = A,B)

Key Takeaways — Section 3.7

- ✓ A MUX selects one of many inputs to a single output; a DEMUX routes one input to one of many outputs.
- ✓ Cascaded full adders (ripple-carry) form multi-bit parallel binary adders; carry propagation limits their speed.
- ✓ Parity generators/checkers use EX-OR trees to detect single-bit transmission errors.
- ✓ A 2^{n-1} -to-1 MUX can implement any n-variable Boolean function without additional gates.

bcalink.com

3.8 Basic Concepts of Programmable Logic

Programmable Logic Devices (PLDs) are ICs containing arrays of AND and OR gates whose interconnections can be configured (programmed) by the user to implement custom logic functions, rather than being fixed at the time of manufacture. This drastically reduces chip count and design turnaround time compared to using discrete SSI/MSI gates.

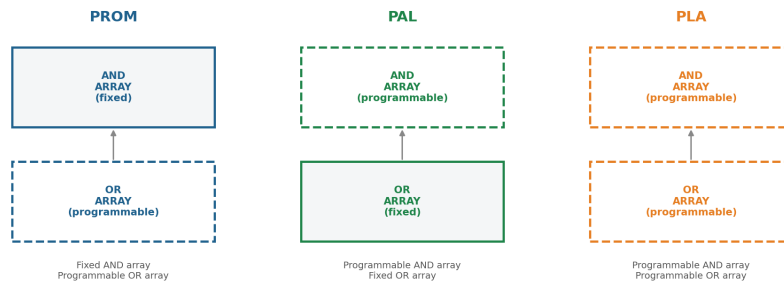


Figure 3.10 — Structural comparison of PROM, PAL, and PLA: which array (AND/OR) is fixed vs. programmable.

3.8.1 PROM (Programmable Read-Only Memory)

A PROM has a fixed AND array (decoding all possible input minterms) and a programmable OR array. Because every minterm is generated internally, PROMs are well suited to functions requiring many minterms, but the fixed AND array wastes silicon area for functions that use only a few product terms. Once programmed (by blowing internal fuses), a standard PROM cannot be reprogrammed.

3.8.2 EPROM (Erasable Programmable Read-Only Memory)

An EPROM functions like a PROM but its stored configuration can be erased — typically by exposing the chip's quartz window to ultraviolet (UV) light for several minutes — and then reprogrammed electrically. This reusability made EPROMs popular for prototyping and low-volume production before EEPROM/Flash memory became widespread.

3.8.3 PAL (Programmable Array Logic)

A PAL has a programmable AND array but a fixed OR array. This structure is simpler and faster to manufacture and program than a PLA, and is efficient for functions where each output needs only a limited number of product terms. PALs are typically one-time programmable (fuse-based).

3.8.4 PLA (Programmable Logic Array)

A PLA offers the most flexibility: both the AND array and the OR array are programmable, allowing product terms to be shared across multiple outputs. This generally uses silicon more

efficiently than a PAL for complex multi-output functions, at the cost of being slower and more complex to program.

Device	AND Array	OR Array	Key Characteristic
PROM	Fixed	Programmable	Generates all minterms; simple but silicon-inefficient
EPROM	Fixed	Programmable	UV-erasable and reprogrammable version of PROM
PAL	Programmable	Fixed	Faster, simpler; limited product-term sharing
PLA	Programmable	Programmable	Most flexible; supports product-term sharing across outputs

Note: Modern digital design has largely moved on to CPLDs and FPGAs, which extend the PLD concept to thousands of programmable logic blocks, but PROM/PAL/PLA remain the foundational concepts underlying all programmable logic.

Key Takeaways — Section 3.8

- ✓ PROM: fixed AND array, programmable OR array — simple, but wasteful for sparse functions.
- ✓ EPROM: like PROM but UV-erasable and reprogrammable.
- ✓ PAL: programmable AND array, fixed OR array — fast and efficient for functions with few product terms.
- ✓ PLA: both AND and OR arrays programmable — most flexible, supports shared product terms across outputs.

— End of Unit 3 —