ELEC205

Digital Systems Design

COURSE NOTES

Aykhan Ahmadzada
Koç University

This work is a personal academic compilation created for educational purposes as part of the **ELEC205 (Digital System Design)** course at **Koç University**.

Compiled in **Istanbul, Turkey**.

ELEC205

- 13. Sequential Logic and Memory
- 14. Flip-Flops & Sequential Circuit Analysis
- 15. Finite State Machines: State Diagrams, Models, and Representations
- 16. FSM Design & Sequence Detection
- 17. State Assignment & Minimization
- 18. Sequential-Circuit Fundamentals & Flip-Flops
- 19. Registers & Bus-Based Transfer Structures
- 20. Registers and Register Transfer Operations
- 21. Counters, Shift Registers, and Serial Transfer
- 22. Programmable Computer and Control Unit
- 23. Algorithmic State Machines and ASM Design
- 24. Design Examples
- 25. Final Exam Review

1. Introduction to Digital Devices

Objective: This note covers the topics from **ELEC 205** Week 1 (Slides 9–36), focusing on the transition from analog to digital systems, fundamental digital devices, and an introductory multiplexer (MUX) design example.

What is Digital? (Analog vs. Digital)

Digital systems operate on discrete (individually separate and distinct) signals—typically 0s and 1s—in contrast to the continuous range of analog signals. Understanding the interplay between analog and digital is crucial for modern electronic design.

Analog vs. Digital Signals

Real-world signals (such as sound waves) are analog and vary continuously. Digital systems sample these signals at discrete intervals.

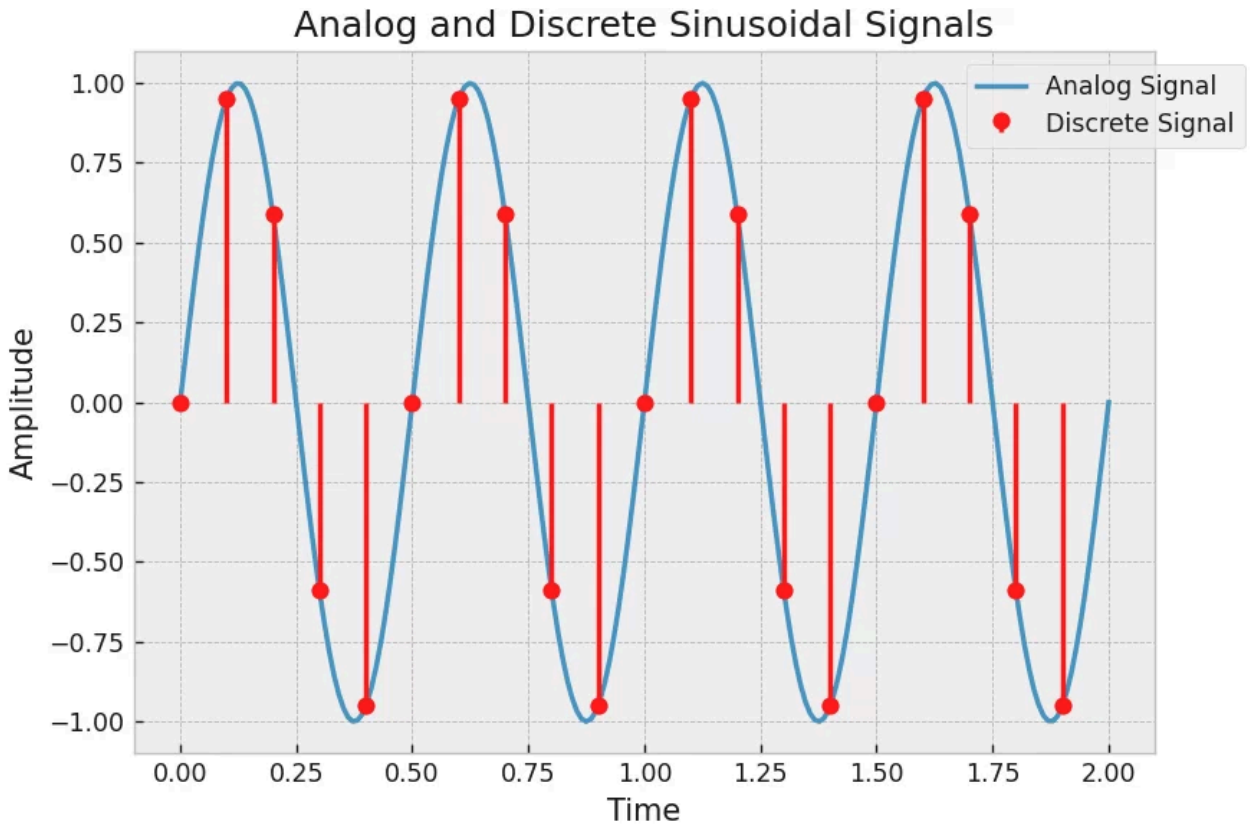
> **(NYQUIST-SHANNON) SAMPLING THEOREM:** If you sample an analog signal at a rate at least twice its highest frequency component, you can fully reconstruct it from these samples. This minimum sampling rate is called the **Nyquist rate.**

Frequency: the number of waves that pass by each second, and is measured in Hertz (Hz).

Physical Storage of 0s and 1s

- Bumps on a CD
- Magnetic domains on a hard disk
- Charge in flash memory transistors
- Flip-flops in integrated circuits

Although hardware at the transistor level is analog, digital abstraction ensures signals are treated as purely “HIGH” (1) or “LOW” (0).



Everyday Examples

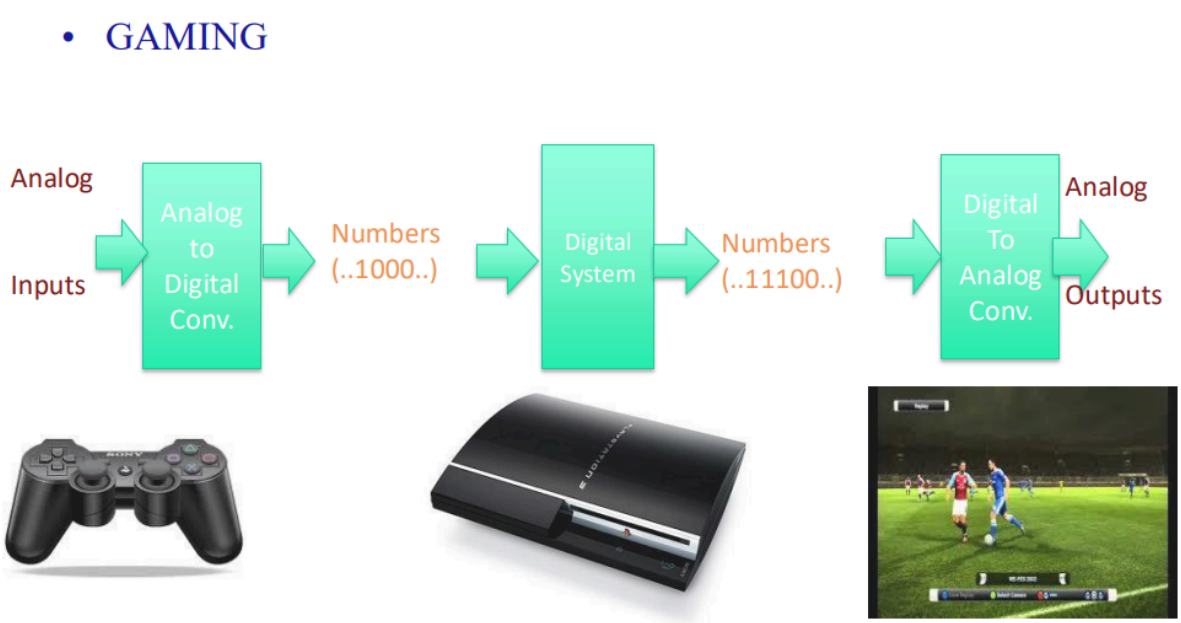
- **Analog TV vs. Digital TV**: Digital TV encodes images as binary data frames, while analog TV uses continuously modulated waves.
- **CD Audio**: Music is stored as samples at 44.1 kHz, each sample represented by bits.
- **Cell Phones**: Convert voice to digital signals, process internally, then convert back to analog for playback.

Digital Systems and Applications

A **digital system** takes binary data as input, performs logic or arithmetic on it, and outputs new binary data. In practice, an ADC (Analog-to-Digital Converter) fronts the system for input, and a DAC (Digital-to-Analog Converter) follows it for output.

Examples of Applications

1. **Gaming Consoles**: Controller inputs are converted to binary, processed by a CPU/GPU, then sent to a display or speakers (digital or analog output).
2. **Personal Computers**: Keyboard and mouse signals are interpreted digitally, processed, and output to a monitor or speaker.
3. **Cell Phones**: Convert audio to packets of digital data, process and store them, and finally reproduce audio signals.



Benefits of Digital Systems

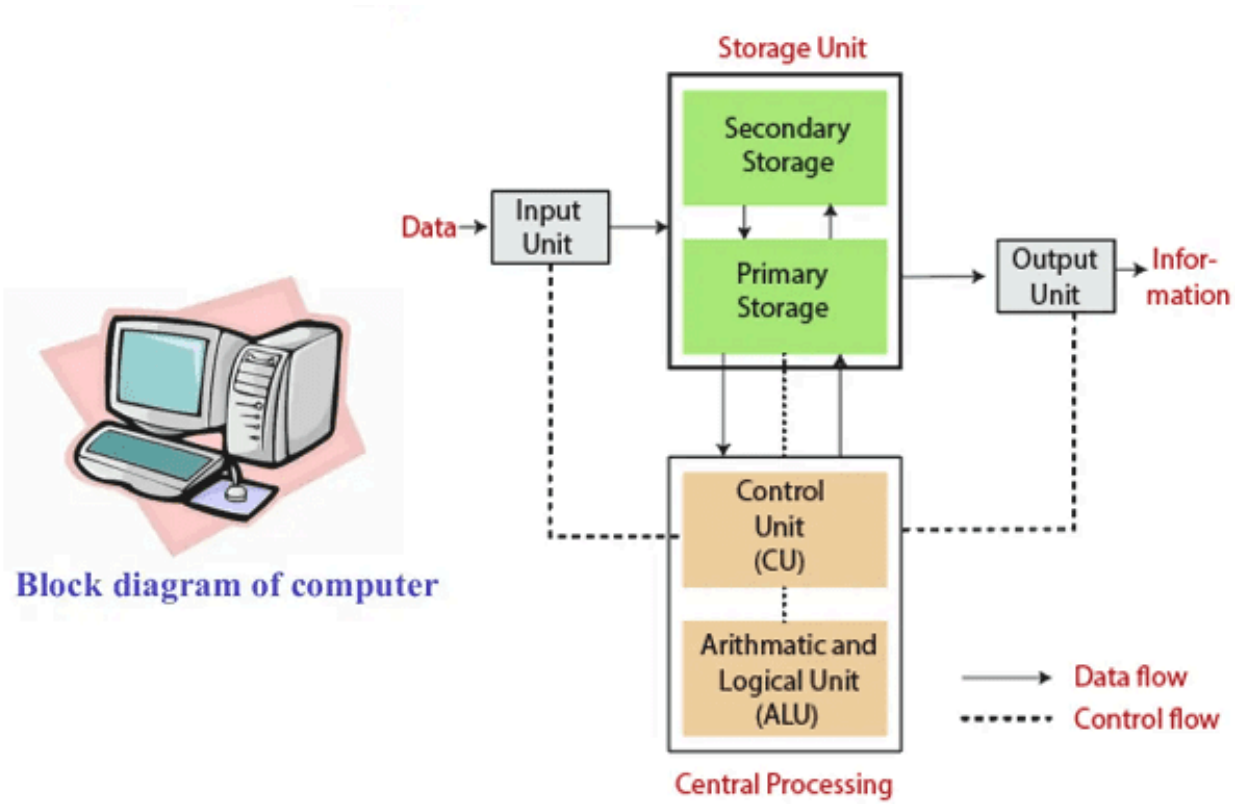
> **BENEFITS:**
>
> - **Reproducibility**: Digital copying does not degrade quality.
> - **Ease of Design**: Logical operations (AND, OR, NOT) are simpler conceptually than continuously variable signals.

> - **Flexibility & Programmability**: Can be updated/reconfigured using firmware or hardware description languages (HDLs).
> - **Speed**: Modern transistors switch in picoseconds, enabling rapid processing.
> - **Economy**: Highly complex functionality on tiny chips.
> - **Advancing Technology**: Each new generation of semiconductor technology brings higher performance at lower cost.

Digital Computer Architecture

A basic digital computer typically includes:

- **Memory**: Stores both instructions and data.
- **Datapath**: Executes arithmetic and logical operations.
- **Control Unit**: Directs the flow of data and orchestrates operations.
- **CPU**: Combines control and datapath, often featuring a **Floating Point Unit (FPU)** for specialized arithmetic and a **Memory Management Unit (MMU)** for handling caches and memory addressing.

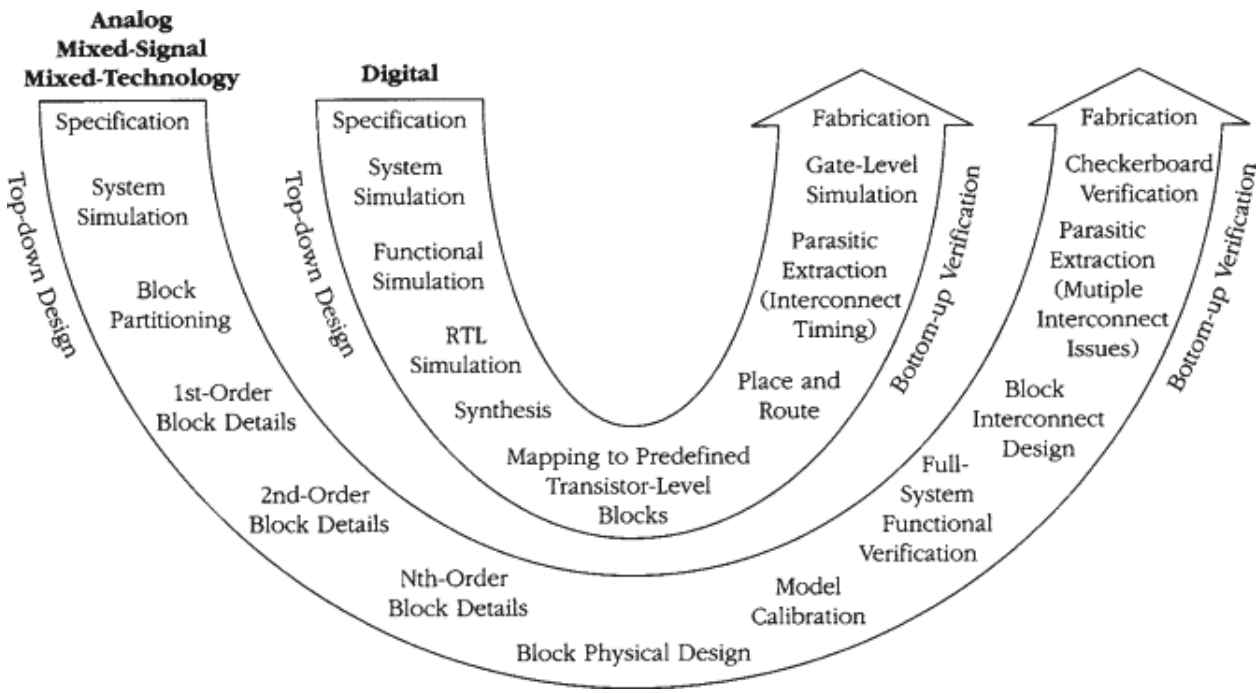


Block diagram of computer

Constructing Digital Systems

> **DIGITAL SYSTEM:** Inputs (binary) → Processing (logic or arithmetic) → Outputs (binary)

The high-level design flow involves specifying the required behavior, then transforming it into logical components. Modern workflows use simulators and HDLs to validate designs before hardware fabrication.



Digital Devices (Gates and Memory)

Basic Gates

AND, OR, and NOT form the foundational building blocks of any digital logic design. Combinational circuits are built exclusively from these gates or their derivatives.

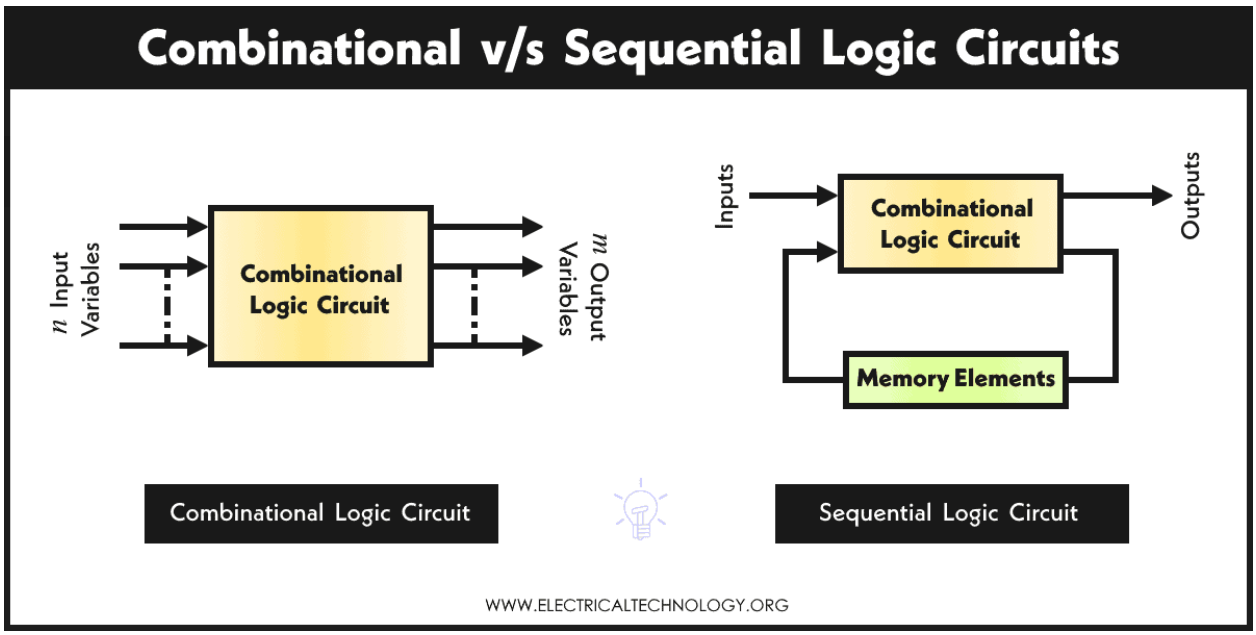
> **COMBINATIONAL CIRCUIT:** A circuit whose output depends solely on its current inputs, with no internal storage. **(no memory!)**

Memory Elements

> **FLIP-FLOP:** A 1-bit storage device that latches data on a clock edge.

> **SEQUENTIAL CIRCUIT:** Combines gates and flip-flops so that outputs depend on current inputs and previously stored states (past inputs). **(with memory!)**

Sequential circuits can implement counters, shift registers, and entire finite state machines.



Electronic and Software Aspects

All gates and flip-flops are physically analog (transistors, resistors, capacitors), but operate within defined voltage levels to represent 0 or 1. Designers rely on:

- **CAD Tools & Simulators**: For schematic-based or HDL-based design entry.
- **HDLs (VHDL, Verilog, SystemVerilog)**: For specifying either the behavior (dataflow/behavioral style) or structure (gate-level) of a design.

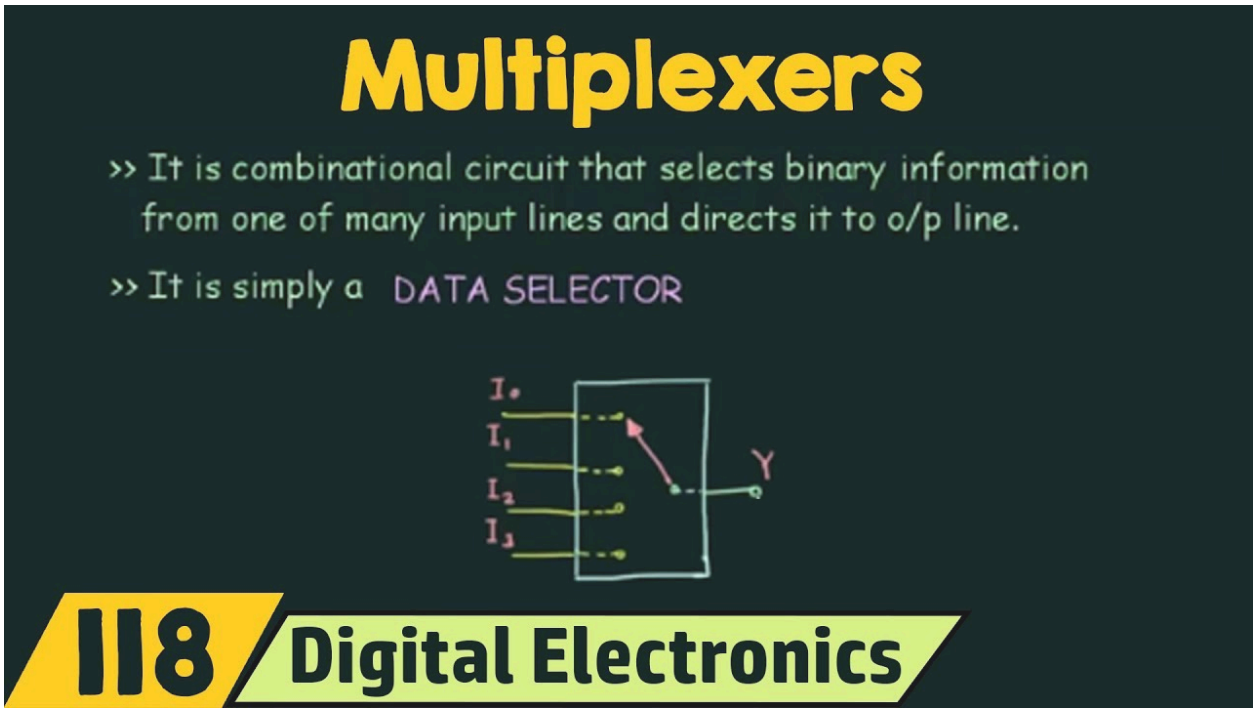
HDL: VHDL is a hardware description language (HDL) that is used to describe the structure and behavior of digital systems and circuits.

Digital Design Levels (MUX Example)

Multiplexer (2-Input MUX)

A multiplexer (MUX) selects one of its inputs to pass through to the output.

> **MULTIPLEXER:** A device that outputs one of several data inputs, controlled by a select signal.



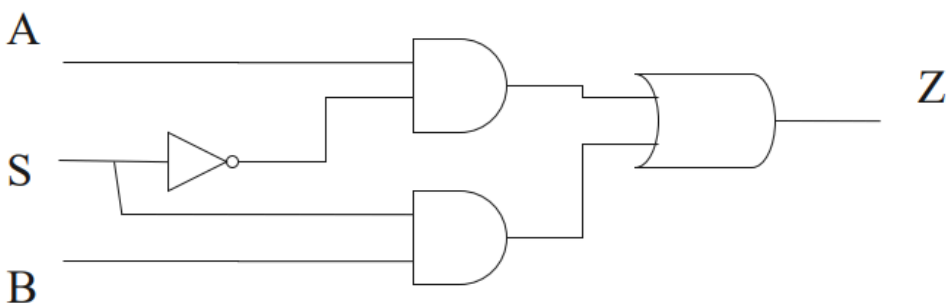
Truth Table

S	A	B	Z
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

When **S = 0**, **Z = A**. When **S = 1**, **Z = B**.

Gate-Level Diagram

- Gate level logic diagram (schematic):



HDL (VHDL) Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V1mux is
   port(
      A, B, S : in  STD_LOGIC;
      Z       : out STD_LOGIC
   );
end V1mux;

architecture V1mux_arch of V1mux is
begin
   -- Dataflow style
   Z <= A when S = '0' else B;
end V1mux_arch;
```

This example demonstrates how designers can describe hardware behavior at a higher level. A gate-level variant would instantiate specific AND, OR, and NOT components

directly.

Why Digital Wins: Precision, Stability, and Efficiency

Even though the physical world operates in an **analog** way—like sound waves, temperature changes, and light brightness—modern technology increasingly relies on **digital systems** because they offer **precision, reliability, and efficiency** that analog cannot always provide.

Analog signals are **continuous** and can take infinite values, but they are also **vulnerable to noise and degradation**. That's why, when you listen to an **analog radio**, you sometimes hear that **"dzzzzzz"** noise—it's interference corrupting the signal. Over time, copying an analog signal (like a cassette tape) causes it to lose quality, whereas digital signals remain **unchanged** no matter how many times they are copied.

However, some things still feel analog, even in digital systems. For example, when you adjust the **brightness of your iPhone's flashlight**, it seems like a smooth change, but the LED is actually flickering on and off rapidly using **Pulse Width Modulation (PWM)**. This is how digital systems **simulate** the behavior of analog while still keeping the benefits of binary operation.

One key feature of digital devices is that they **either work perfectly or not at all**. Since digital signals are based on **1s and 0s**, there's no gradual loss of quality. A radio station using digital transmission either delivers a **clear signal** or nothing at all—there's no static like in analog radio. The same applies to digital files, which either open **perfectly** or become completely unreadable if corrupted.

In the end, **digital doesn't replace analog completely—it refines and optimizes it.** We still live in an analog world, but digital technology helps us process, store, and transmit information more efficiently, making it clearer, more reliable, and easier to manipulate.

Self Test

Self-Test: Lecture 1

2. Binary numbers, Unsigned addition/subtraction, Two's complement system

Objective: This note covers the topics from **ELEC 205** Week 1 (Slides 37–57), focusing on number systems (binary, octal, hexadecimal), two's complement, and various binary arithmetic operations.

Number Systems

A number system defines how numeric values are represented. In digital electronics, we commonly use **binary** (base 2), but other bases such as **octal** (base 8) and **hexadecimal** (base 16) are also helpful.

Numbering System		
System	Base	Digits
Binary	2	0, 1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Octal and Hexadecimal Numbers

Octal (base 8) digits cover 0 to 7, each corresponding to three bits in binary. Hexadecimal (base 16) digits span 0 to 9 and A to F, each corresponding to four bits in binary. Converting between binary and hex is straightforward by grouping bits in fours; octal uses groupings of three bits.

> **OCTAL DIGIT:** Uses 3 bits (e.g., 101_2 = 5_8).
>
> **HEXADECIMAL DIGIT**: Uses 4 bits (e.g., 1010_2 = A_{16}).

These systems give concise shorthand for large binary strings, useful in debugging, memory addresses, and processor instructions.

Positional Number System Conversions

Numbers in any base r are interpreted by positional notation. For digits to the left of the radix point, powers of r increase from right to left; for digits to the right, powers of r are negative.

1. **Base-r to Decimal**: Multiply each digit by r^{position} and sum the results.
2. **Decimal to Base-r**: Repeatedly divide by r, keep track of remainders, and reverse them at the end.

EXAMPLE: Converting 179_{10} to binary:

1. $179 \div 2 = 89$ remainder 1
2. $89 \div 2 = 44$ remainder 1

3. $44 \div 2 = 22$ remainder 0
4. $22 \div 2 = 11$ remainder 0
5. $11 \div 2 = 5$ remainder 1
6. $5 \div 2 = 2$ remainder 1
7. $2 \div 2 = 1$ remainder 0
8. $1 \div 2 = 0$ remainder 1 (MSB)

Reverse remainders → 10110011_2.

Additional Example

- Convert 300_{10} to octal:
  - $300 \div 8 = 37$ remainder 4
  - $37 \div 8 = 4$ remainder 5
  - $4 \div 8 = 0$ remainder 4 (MSB)
  - Result: 454_8.

Addition and Subtraction of Binary Numbers

Binary addition and subtraction are analogous to decimal, except the base is 2.

> **BINARY ADDITION:** 1 + 1 = 10_2. If both bits are 1, produce a sum bit of 0 and carry out 1.
>
> **BINARY SUBTRACTION**: 1 – 1 = 0, but 0 – 1 requires borrowing from a more significant bit.

Additional Examples

- **Addition**: 1011_2 (11 in decimal) + 1000_2 (8 in decimal) = 10011_2 (19 in decimal)

- Addition

```
          101111000
X    190  10111110
Y    141  10001101
___+_______________
X+Y  331 101001011
```

C_{in}/b_{in}	X	Y	C_{out}	S	B_{out}	D
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

- **Subtraction**: 1011_2 (11 in decimal) – 1000_2 (8 in decimal) = 0011_2 (3 in decimal)
  - Binary subtraction: (borrow, difference bits)

```
B_out                  001111100
Minuend     X    229    11100101
Subtrahend  Y     46    00101110
_____________-_________________
Difference  X-Y  183    10110111
```

C_{in}/b_{in}	X	Y	C_{out}	S	B_{out}	D
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

    - Use binary subtraction to compare numbers.
    - If X-Y produces a borrow out at the most significant bit, then X is less than Y.

Representation of Negative Numbers

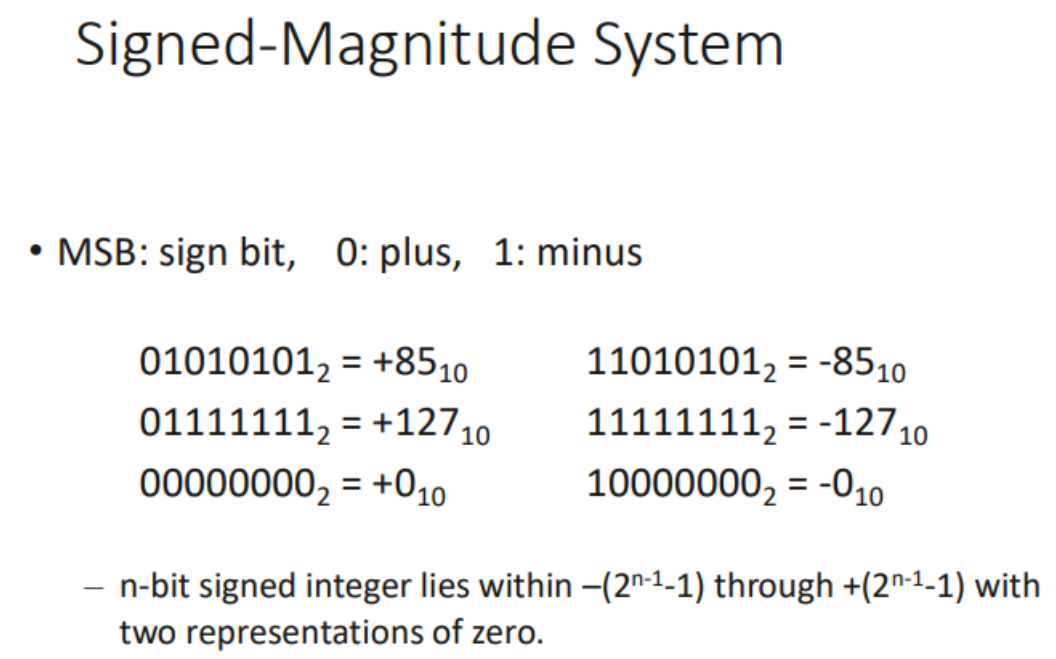
Various methods can represent signed integers in binary:

> **SIGNED-MAGNITUDE:** A sign bit plus magnitude bits.
>
> **ONE'S COMPLEMENT:** Flip (invert) all bits to represent negative.
>
> **TWO'S COMPLEMENT:** Invert bits and then add 1 for negative.

Two's complement is most common because it simplifies hardware for addition/subtraction.

Signed-Magnitude System

The most significant bit (MSB) is the sign: 0 for positive, 1 for negative. The rest of the bits store the magnitude. Though intuitive, arithmetic operations are more complex compared to two's complement.



Diminished Radix Complement System

> $(r-1)$**'S COMPLEMENT:** For an n-digit number N, $(r^n-1)-N$. In binary ($r=2$), this is one's complement, created by flipping all bits.

The **diminished radix complement system** is a way of representing negative numbers by subtracting a number from the **largest possible value** in a given number system.

- Given a number N in radix r having n digits
- The (r-1)'s complement of N is defined as: $(r^n - 1) - N$

- For binary, **one's complement** of N is: $(-N) = (2^n - 1) - N$

  For n = 4
  -0001 = 1111 - 0001 = 1110
  -0101 = 1111 - 0101 = 1010

Complement Number Systems

Complement systems allow negative numbers to be handled using the same addition logic as positive numbers.

> **RADIX COMPLEMENT:** For base r, the complement is $r^n - N$. In binary, $2^n - N$ (two's complement).

- Taking the complement is more difficult than changing the sign, but in complement system add/subt are easier.
- Radix complement of N: $(-N) + N = r^n \rightarrow (-N) = r^n - N$ where N is an n-digit number
  - If N is between 1 and r^n-1 then (-N) is between r^n-1 and r^n- (r^n-1) = 1
  - When N=0, (-N) is r^n, which is (n+1) digits, hence (-N) is also 0.

Two's Complement System

1. Invert (one's complement).
2. Add 1.

> **TWO'S COMPLEMENT:** $-N$ is $2^n - N$. This avoids having two representations for zero and simplifies arithmetic.

- Range for n bits: -2^{n-1} through $+2^{n-1} - 1$.

- Radix complement for binary numbers
  N: n bit binary number

$$(-N) = 2^n - N = (2^n - 1) - N + 1$$

$$17_{10} = 00010001_2 \qquad 0000_2$$
$$11101110 \qquad 1111$$
$$+\_\_\_\_\_1 \qquad +\_\_\_1$$
$$11101111_2 = -1710 \qquad 10000_2$$

Additional Example

- **Convert −6 to 8-bit two's complement**:
  1. $+6 = 0000\ 0110$.
  2. Flip bits → $1111\ 1001$.
  3. Add 1 → $1111\ 1010$ (final representation of −6).

Signed Number Systems

> **SIGNED NUMBER SYSTEMS:**
>
> - Signed-Magnitude (clear sign bit, complex arithmetic)
> - One's Complement (invert bits for negative)
> - Two's Complement (dominant standard, single zero, consistent addition/subtraction)

Let's consider n-bit binary number: A = $a_{n-1}\ a_{n-2}\ \ldots a_1\ a_0$

- Sign-Magnitude System ($-(2^{n-1}-1)$ to $(2^{n-1}-1)$)
  - Sign: a_{n-1} (MSB); a_{n-1}= 0 positive, a_{n-1}= 1 negative
  - Magnitude: $a_{n-2}\ldots a_1\ a_0$
- One's Complement System ($-(2^{n-1}-1)$ to $(2^{n-1}-1)$)
  - Negation is represented with bitwise NOT
    - $(-A) = (2^n-1) - A = a'_{n-1}\ a'_{n-2}\ldots a'_1\ a'_0$; (+1=0001, -1=1110)
- Two's Complement System ($-(2^{n-1})$ to $(2^{n-1}-1)$)
  - Decimal value of A is represented as:
    - $A = -2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + \ldots 2^1a_1 + 2^0a_0$
  - Satisfies $(-A) + A = 2^n \rightarrow (-A) = 2^n - A = (2^n - 1) - A + 1$

3-Bit Number Examples

When $n = 3$, two's complement ranges from $-4\ (100_2)$ to $+3\ (011_2)$.

Bits	Unsigned	Signed Magnitude	One's Complement	Two's Complement
0 0 0	0	0	0	0
0 0 1	1	1	1	1
0 1 0	2	2	2	2
0 1 1	3	3	3	3
1 0 0	4	-0	-3	-4
1 0 1	5	-1	-2	-3
1 1 0	6	-2	-1	-2
1 1 1	7	-3	-0	-1

Additional Example

Binary	Unsigned	Signed-Magnitude	One's Complement	Two's Complement
100	4	−0	−3	−4
101	5	−1	−2	−3
110	6	−2	−1	−2
111	7	−3 (or −0)	−0 (or −0)	−1

Addition in Two's Complement

Perform regular binary addition. A negative operand is already stored in two's complement form.

EXAMPLE: $-3 + 1$ in 4-bit two's complement

- $-3 \rightarrow 1101_2$
- $+1 \rightarrow 0001_2$
- Sum = 1110_2 (−2 in decimal)

Check for **overflow** when results exceed $[-2^{n-1}, 2^{n-1} - 1]$.

Additional Example

- $-4 + 2$ in 4-bit:
  - $-4 = 1100_2$
  - $2 = 0010_2$
  - Sum = $1110_2 \rightarrow (-2)$

Subtraction in Two's Complement

To subtract Y from X, compute $X + (-Y)$. Negative numbers are formed via two's complement.

Additional Example

- $5 - 6$ in 4-bit:
  - $5 = 0101_2$

- $6 = 0110_2$
- $-6 = 1001_2 + 1 = 1010_2$
- $0101 + 1010 = 1111_2$ = (−1)

1. Unsigned System

- Represents **only non-negative** numbers.
- All bits contribute to the numerical value.
- The range is from **0 to** $2^n - 1$ for an n-bit system.
- Example (4-bit system):
  - **Binary:** `0000` to `1111`
  - **Decimal equivalents:** `0` to `15`

There is **no concept of negative numbers** in an **unsigned** system.

2. Signed Systems (Different Complement Methods)

A. Signed Magnitude

- The **leftmost bit (MSB)** is the **sign bit**.
  - `0` = **positive**
  - `1` = **negative**
- The remaining bits represent the magnitude.
- The range for an n-bit system is $-(2^{n-1} - 1)$ **to** $+(2^{n-1} - 1)$.
- Example (4-bit system):
  - `0111` = `+7`
  - `1111` = `-7`
  - **Problem:** Two representations of zero (`0000` and `1000`), which makes arithmetic operations complicated.

B. One's Complement (Diminished Radix Complement)

- Negative numbers are represented by **flipping all bits**.
- The range for an n-bit system is $-(2^{n-1}-1)$ **to** $+(2^{n-1}-1)$.
- Example (4-bit system):
  - `0111` = `+7`
  - `1000` = `-7`
  - `0000` = `+0`
  - `1111` = `-0` (**Problem:** Two zeros, which complicates arithmetic)

C. Two's Complement (Radix Complement)

- Negative numbers are found by **flipping all bits and adding 1**.
  - The range for an n-bit system is -2^{n-1} **to** $+(2^{n-1}-1)$.
- Example (4-bit system):
  - `0111` = `+7`
  - `1001` = `-7`
  - **Only one zero (`0000`), solving the double-zero issue.**
  - Arithmetic works smoothly, making it the **standard for modern computers**.

Key Differences Between Signed and Unsigned Systems

Feature	Unsigned	Signed Magnitude	One's Complement	Two's Complement
MSB Role	Part of the magnitude	Sign bit (0 = +, 1 = -)	Sign bit (0 = +, 1 = -)	Sign bit (0 = +, 1 = -)
Range (4-bit)	`0 to 15`	`-7 to +7`	`-7 to +7`	`-8 to +7`
Negative Representation	Not supported	Flip MSB	Flip all bits	Flip all bits and add 1
Zero Representation	`0000` (0 only)	`0000` (+0), `1000` (-0)	`0000` (+0), `1111` (-0)	`0000` (only one zero)
Arithmetic Simplicity	Simple but no negatives	Complex	Complex (double zero)	**Efficient (modern**

				standard)

- **Unsigned numbers** are simple but cannot represent negatives.
- **Signed numbers** use different methods, with **two's complement being the most practical and widely used** because it avoids double-zero issues and simplifies arithmetic.
- **Diminished Radix Complement (One's, Nine's, Three's, etc.)**
  - Found by **subtracting from the highest possible value minus 1**.
  - Has **two representations of zero**.
- **Radix Complement (Two's, Ten's, Four's, etc.)**
  - Found by **subtracting from the full radix power**.
  - Has **only one zero representation**, making it more practical for arithmetic.

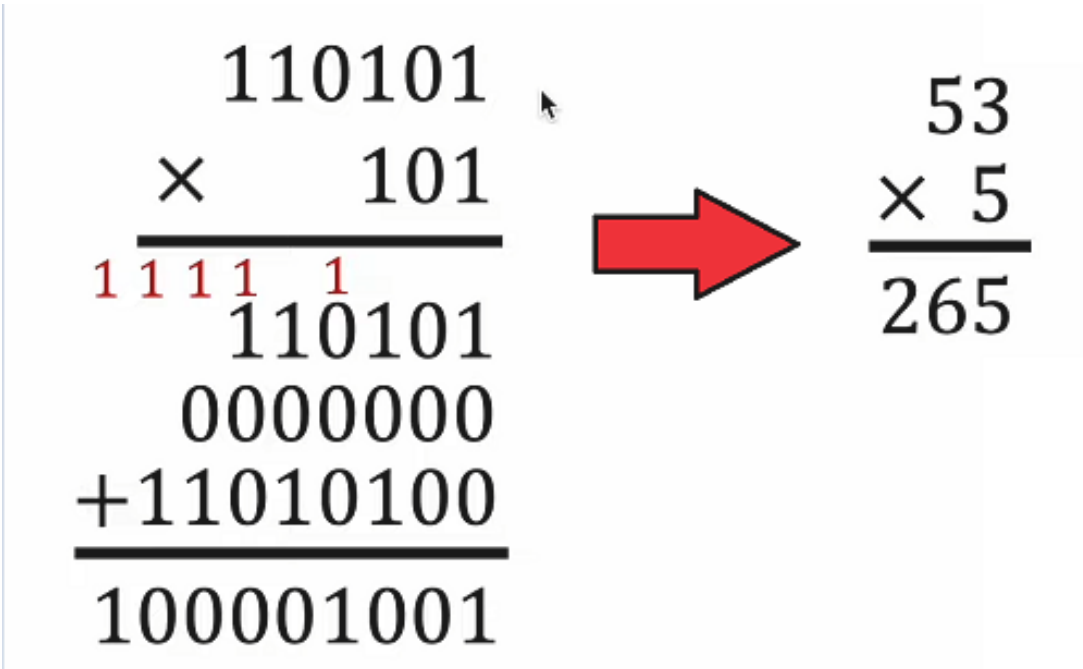
Complement Comparison Table

Base	Diminished Radix Complement	Formula	Radix Complement	Formula
Binary (2)	One's Complement	$(2^n - 1) - N$	Two's Complement	$2^n - N$
Decimal (10)	Nine's Complement	$(10^n - 1) - N$	Ten's Complement	$10^n - N$
Quaternary (4)	Three's Complement	$(4^n - 1) - N$	Four's Complement	$4^n - N$

Binary Multiplication

Unsigned multiplication uses **shift-and-add**. Two's complement multiplication extends this approach while managing sign bits.

> **BINARY MULTIPLICATION:** Multiply partial products, each shifted by the position of the bit in the multiplier.

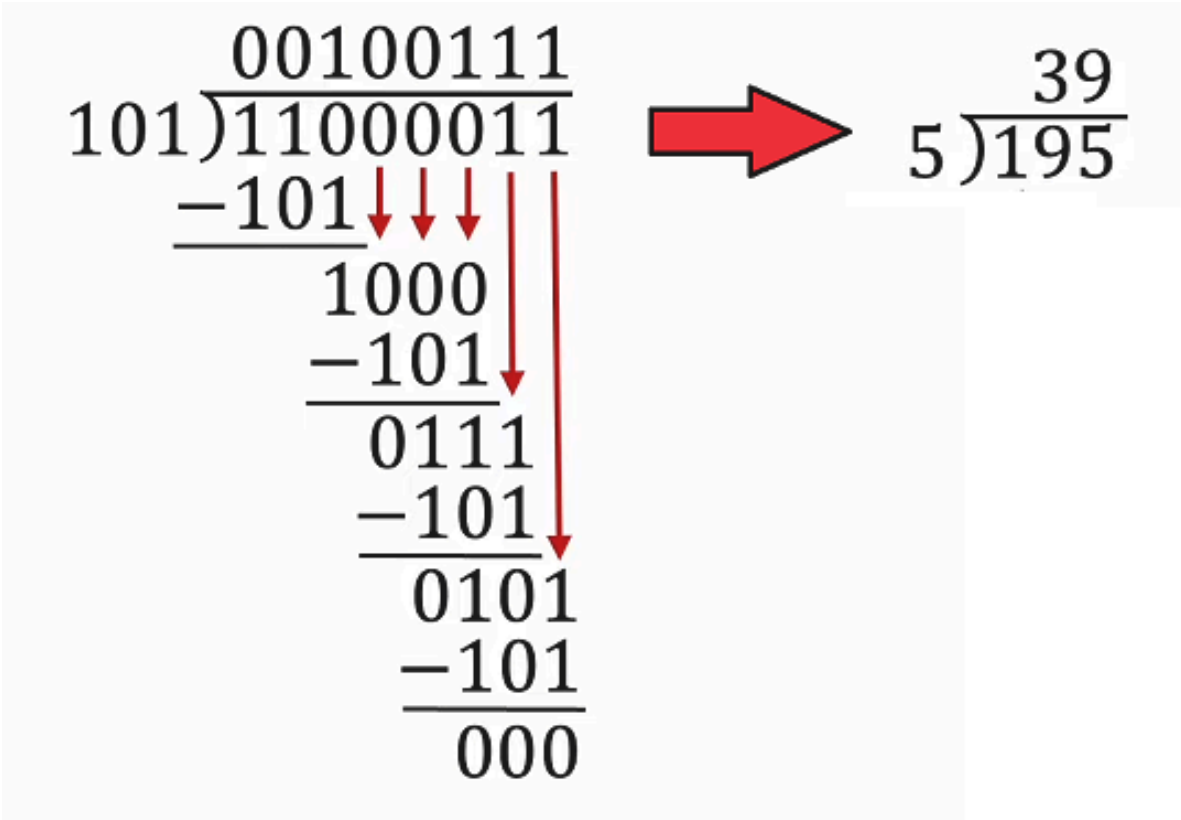


Additional Example

- 3×2 in binary (unsigned):
  - $3 = 0011_2$
  - $2 = 0010_2$
  - Multiply partial products:
    - 1 (LSB of 2) → 0011
    - Next bit is 0 → 0000, shifted
    - Sum = 0110_2 (6 in decimal)

Binary Division

Binary division often uses **shift-and-subtract**. For signed numbers, adjust for sign before or after the division.



Additional Example

- $14 \div 2$ in binary:
  - $14 = 1110_2$, $2 = 0010_2$.
  - Perform repeated shifting and subtracting until the final quotient is found (0111_2) = 7, with remainder 0.

Binary-Coded Decimal (BCD)

BCD encodes each decimal digit (0–9) into a 4-bit binary code (0000 to 1001). Any 4-bit pattern above 1001 is invalid in standard BCD.

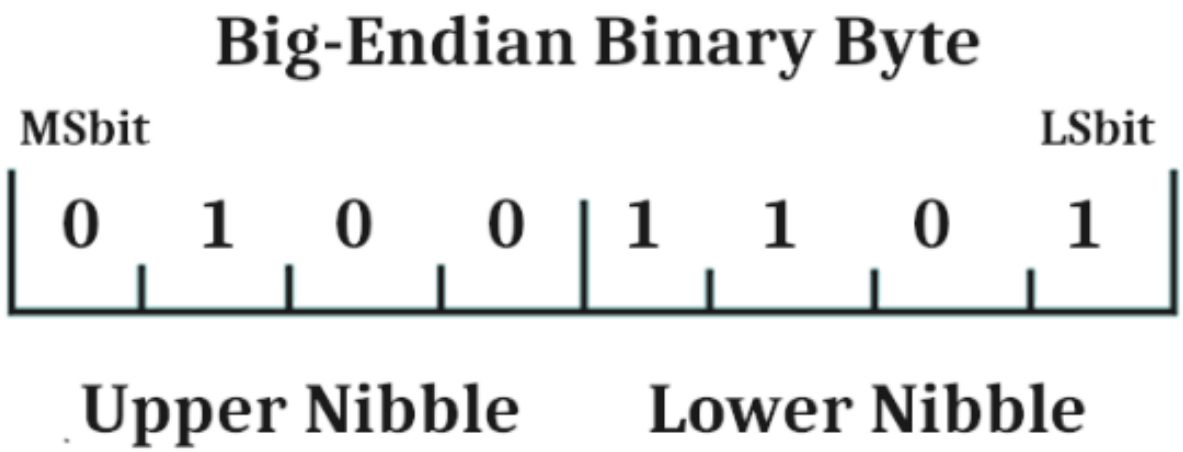
> BCD: 0111 (7) is valid, 1010 (10) is not valid for a single decimal digit.

BCD Addition

If a 4-bit sum exceeds 9 (1001_2), add 6 (0110_2) to adjust the digit and manage any carry.

Additional Example

- Add **44** and **89** in BCD:
  - 44 → $0100\ 0100$
  - 89 → $1000\ 1001$
  - Initial sum → $1\ 0011\ 1111$ (not corrected)
  - Adjust each nibble > 1001 by adding 0110
  - Final correct BCD result = 133 (1 3 3 = $0001\ 0011\ 0011$ in BCD)



Why is BCD (Binary-Coded Decimal) Used?

BCD (**Binary-Coded Decimal**) is used primarily in **applications where decimal precision is important**. Instead of storing numbers in pure binary, BCD represents each **decimal digit separately** using a **4-bit binary equivalent**. **BCD is not memory-efficient.** It **wastes storage** compared to pure binary. **BCD is easy to encode and decode.**

Self Test

Self-Test: Lecture 2

3. Multiplication, Division, and Binary-Coded Decimal in Digital Systems

This note covers the full range of arithmetic operations in digital systems, including both multiplication and division for signed and unsigned numbers, as well as an in-depth look at Binary-Coded Decimal (BCD).

Multiplication in Digital Systems

Multiplication in digital systems can be performed on both unsigned and signed numbers. The underlying mechanism is typically based on the shift-and-add algorithm.

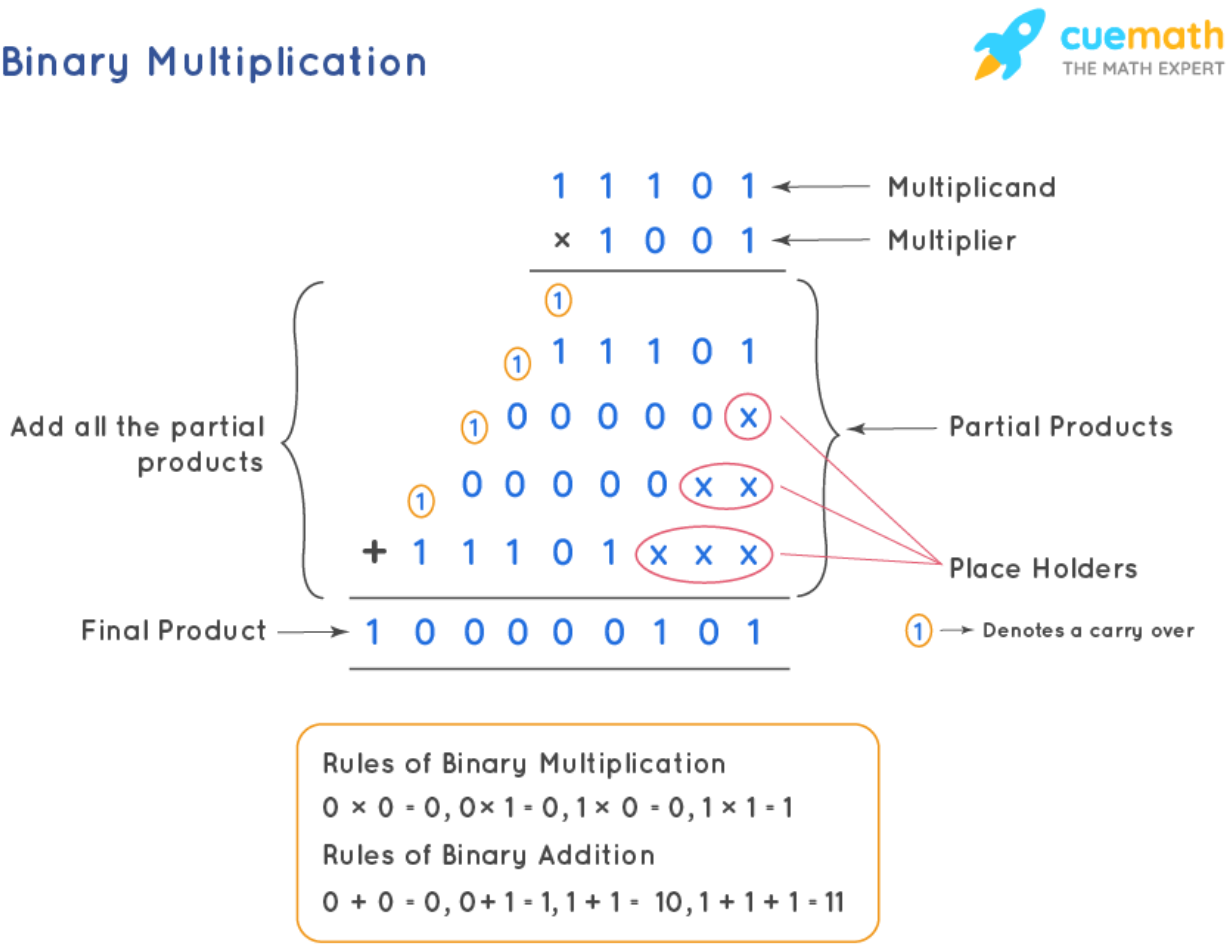
Unsigned Multiplication

For unsigned numbers, multiplication is conceptually similar to decimal multiplication but performed in binary. The basic idea is:

- **Shift:** For each bit in the multiplier, shift the multiplicand by the appropriate number of positions.
- **Add:** Sum the shifted multiplicands where the corresponding bit of the multiplier is 1.

Example: Multiply 1011_2 (11 in decimal) by 0101_2 (5 in decimal).

- Multiply each bit of the multiplier by the multiplicand and shift accordingly.
- Add the resulting partial products to obtain the final product.

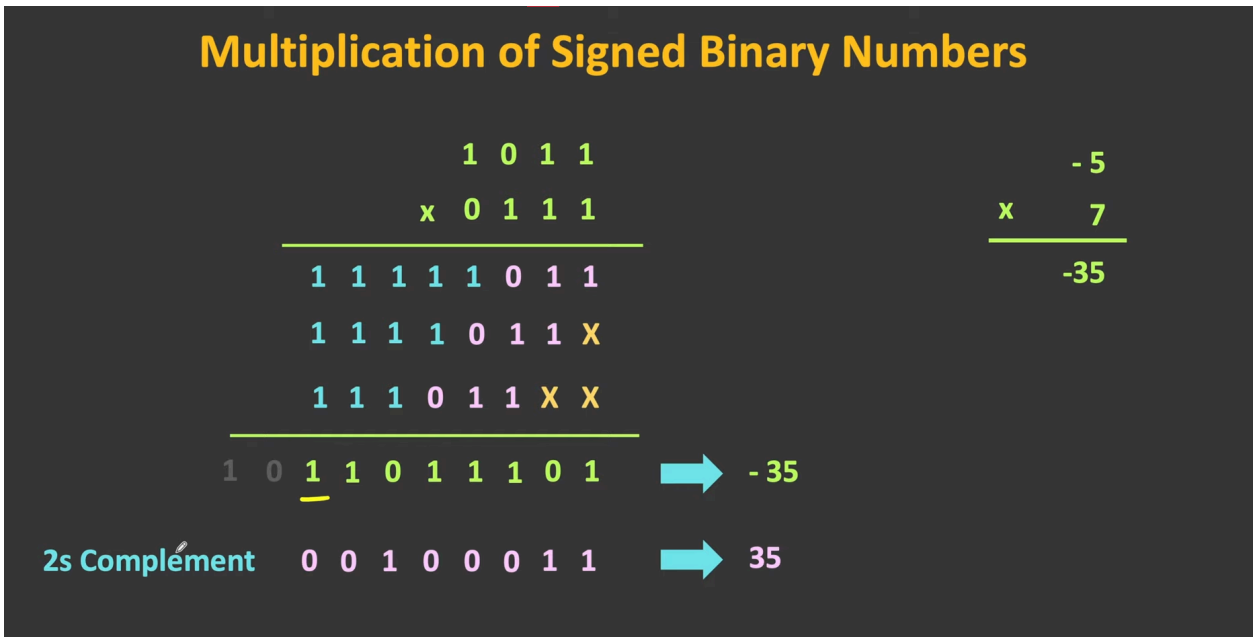


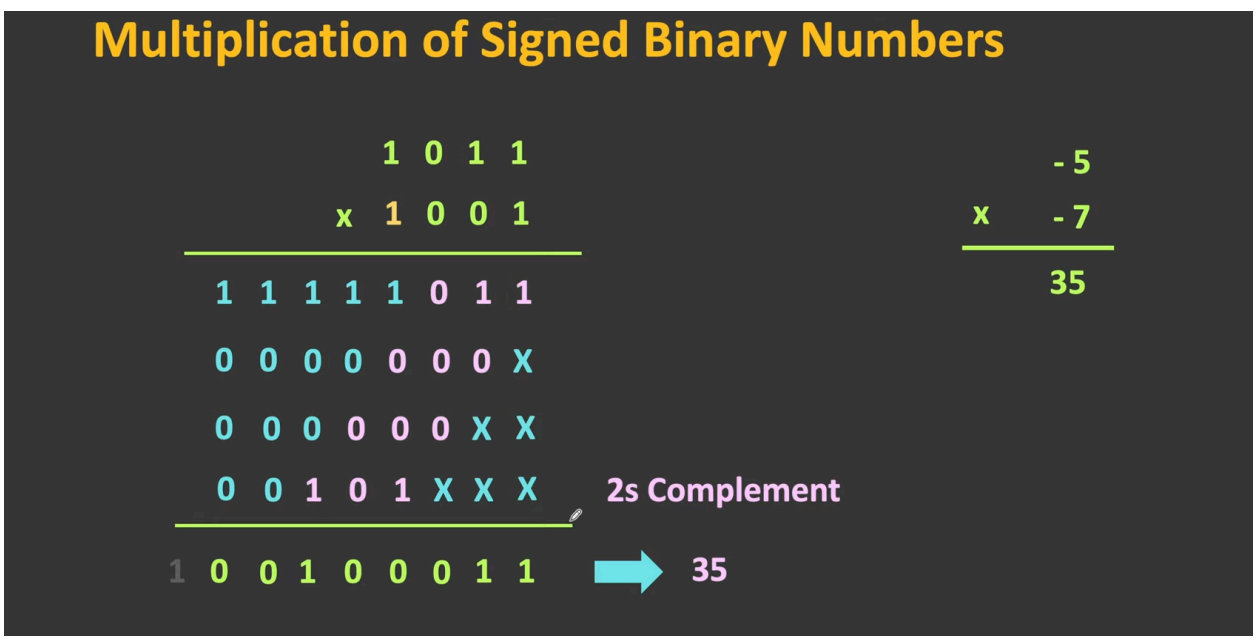
Signed Multiplication

For signed numbers, the typical method is to use two's complement representation. The same shift-and-add procedure is applied, but extra care is needed to:

- **Sign Extend:** Ensure that when shifting, the sign bit is correctly extended.
- **Adjust:** Interpret the final result as a two's complement number.
- **Two's Complement Correction:** If the multiplicand is negative, the last partial product must be converted to its two's complement form to maintain correctness.

Key Point: The algorithm for signed multiplication is similar to unsigned multiplication, but the hardware or software must handle sign bits appropriately, ensuring that negative partial products are correctly processed using two's complement conversion when necessary.





2. Division in Digital Systems

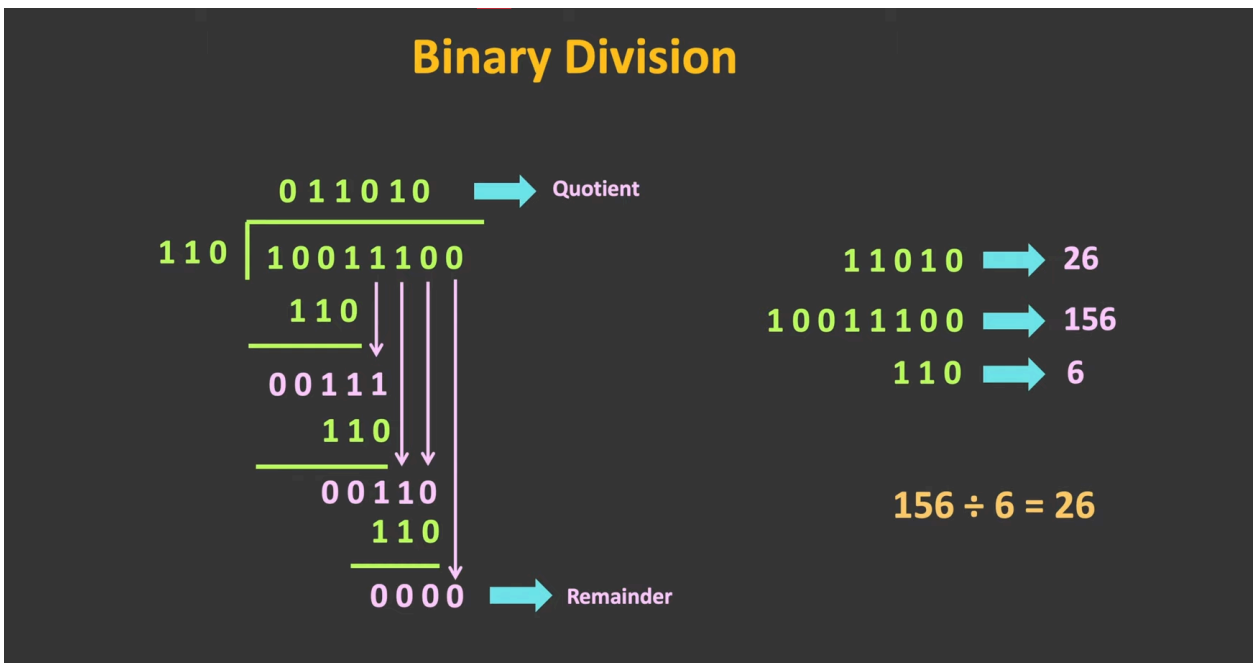
Division is the inverse of multiplication and, like multiplication, is performed differently for unsigned and signed numbers.

Unsigned Division

Unsigned division is generally carried out by a **shift-and-subtract** method:

- **Shift:** Align the divisor with the dividend's most significant bit.
- **Subtract:** Subtract the divisor (or its shifted version) from the dividend if it fits; record a 1 in the quotient.
- **Repeat:** Continue the process by shifting and subtracting until all bits are processed.

Example: Dividing a binary number by another using repeated subtraction and shifts.



Signed Division

For signed division using two's complement representation, the algorithm typically involves:

- **Determining the Sign:** The sign of the result is the product of the signs of the dividend and divisor.
- **Converting to Unsigned:** Temporarily convert both numbers to their absolute (unsigned) values.

- **Performing Division:** Use the unsigned division algorithm.
- **Restoring the Sign:** Apply the appropriate sign to the quotient.

Considerations: Care must be taken to handle cases like division by zero and the edge case where the dividend is the minimum representable value.

3. Binary-Coded Decimal (BCD)

Binary-Coded Decimal (BCD) is a method of representing decimal numbers in which each digit is stored as its own 4-bit binary number. This is especially useful in applications where decimal precision is critical (e.g., financial calculations).

Representation

- **Standard BCD:** Each decimal digit (0 through 9) is represented by a 4-bit binary code.
  - For example, the decimal number 93 is represented as:

$$9 \to 1001, \quad 3 \to 0011, \quad \text{so } 93 \text{ is } 1001\ 0011 \text{ in BCD.}$$

Operations in BCD

Arithmetic operations in BCD (addition, subtraction) are performed digit by digit. However, if the result of a digit addition exceeds 9 (1001 in binary), a correction must be applied:

- **Correction Rule:** If the sum of a digit exceeds 9, add 6 (0110 in binary) to that digit and propagate the carry to the next higher digit.

Example of BCD Addition:

Add 44 and 89 in BCD:

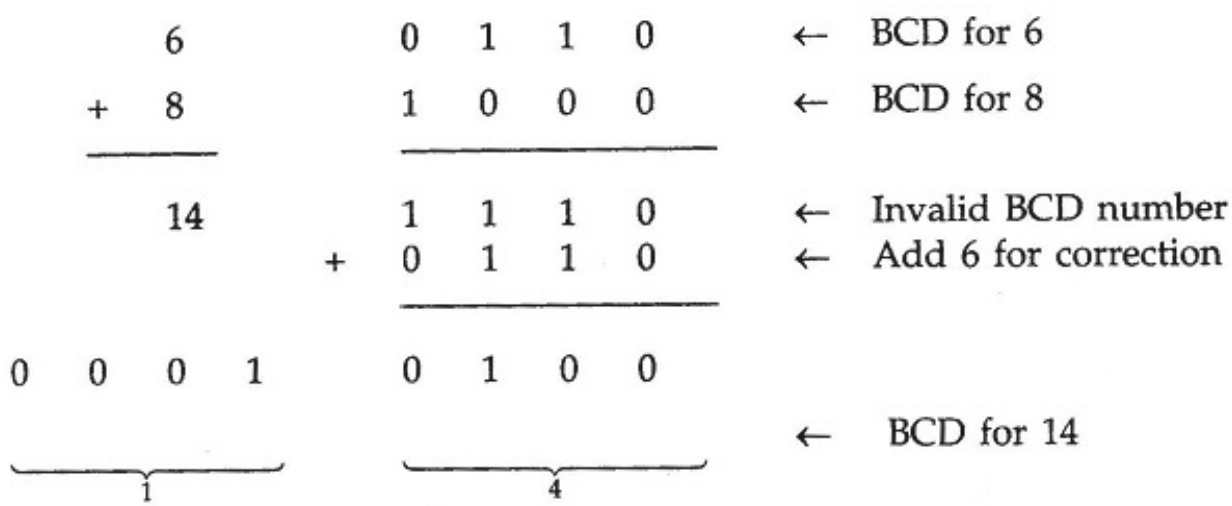
- Represent 44 as: $0100\ 0100$
- Represent 89 as: $1000\ 1001$
- Add corresponding digits:
  - Right nibble: $0100 + 1001 = 1101$. Since 1101 (13) is greater than 9, add 0110 to get $1101 + 0110 = 1\,0011$ (carry 1, result digit 0011).

- Left nibble: $0100 + 1000 + \text{carry } 1 = 0100 + 1000 + 0001 = 1101$. Again, 1101 (13) is greater than 9, so add 0110 to get $1101 + 0110 = 1\,0011$ (carry 1, result digit 0011).

- The final BCD result must be adjusted to reflect the carried digits appropriately. (The exact BCD representation would depend on the method used, ensuring the final answer has the correct number of significant digits based on the lowest precision input.)

Importance of BCD

BCD is crucial in systems where decimal accuracy matters because it avoids rounding errors that can occur when converting between binary and decimal. Although it is less space-efficient than pure binary, its simplicity in representing decimal digits makes it valuable in financial and commercial applications.



Summary

This note has covered:

- **Multiplication:** Both unsigned and signed (using two's complement), focusing on the shift-and-add method and the handling of sign bits.
- **Division:** Both unsigned (via shift-and-subtract) and signed division (including conversion to absolute values and sign correction).
- **Binary-Coded Decimal (BCD):** Representation of decimal digits in 4-bit groups, the rules for BCD arithmetic, and the importance of maintaining decimal precision.

Understanding these operations is essential for designing efficient digital systems, especially in contexts where arithmetic accuracy and data representation are critical.

Self Test

Self-Test: Lecture 3

4. Combinational Digital Systems and Boolean Algebra

This note covers topics from combinational digital systems to Boolean algebra, as presented in the lecture slides (23 to 57). It includes detailed explanations of digital logic circuits, basic gate functions, multi-input systems, and key Boolean algebra theorems and properties.

Combinational Digital Systems

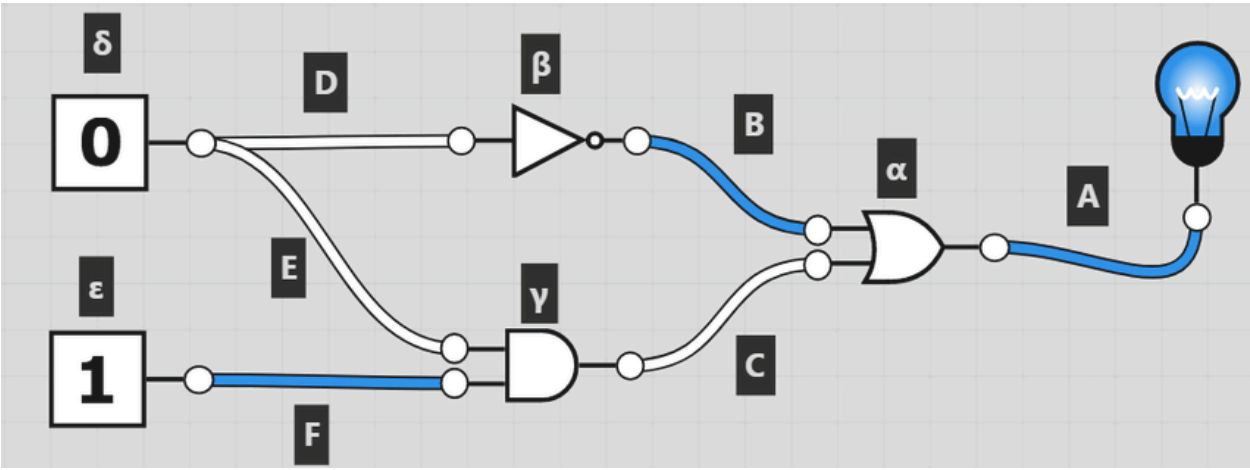
Combinational digital systems are circuits where the output depends solely on the current inputs, with no memory elements. They contrast with sequential systems, where past inputs affect the current output.

> **COMBINATIONAL DIGITAL SYSTEM:** A system in which the output is determined only by the current combination of inputs.

Key Points:

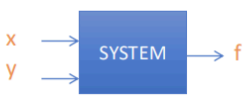
- No storage or memory; all operations are instantaneous.
- Used for arithmetic operations, data routing, and signal processing.

- Fundamental building blocks for more complex circuits like adders, multiplexers, and decoders.



Combinational Digital Systems

Let's complicate things a little bit:
Two inputs, one output



x y	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
0 0	0	0	0	1	1	0	1	1
0 1	0	0	1	1	0	1	0	1
1 0	0	0	1	1	0	1	0	1
1 1	0	1	1	0	0	0	1	1

$f_1=0$ Value Fixing | $f_2=xy$ AND | $f_3=x+y$ OR | $f_4=(xy)'$ NAND | $f_5=(x+y)'$ NOR | $f_6=x \oplus y$ XOR | $f_7=(x \oplus y)'$ XNOR | $f_8=1$ Value Fixing

25

Logic Gates

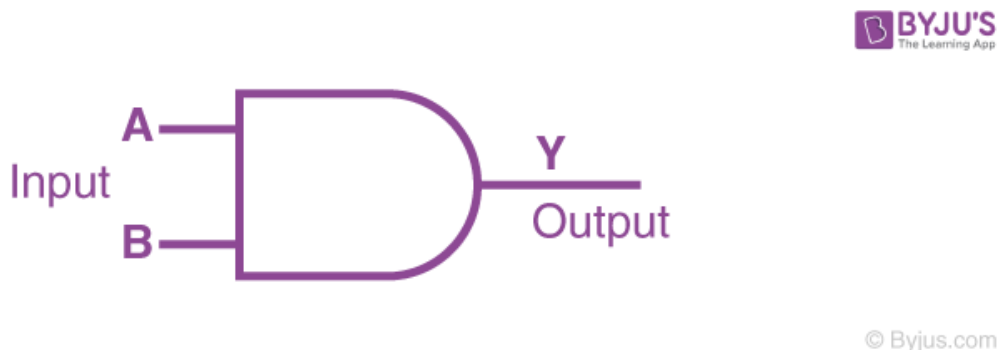
AND Gate

- **Function:** Outputs 1 only if both inputs are 1.
- **Algebraic Expression:** $f = x \cdot y$

- **Truth Table:**

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

> **AND GATE:** A basic digital logic gate that produces an output of 1 if and only if all its inputs are 1.

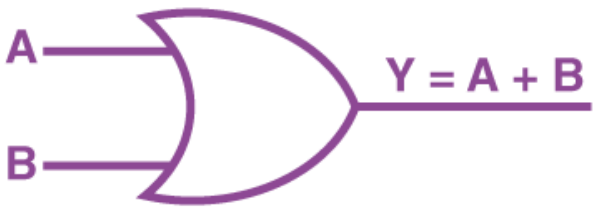


OR Gate

- **Function:** Outputs 1 if at least one input is 1.
- **Algebraic Expression:** $f = x + y$
- **Truth Table:**

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

> **OR GATE:** A logic gate that outputs 1 if one or more of its inputs are 1.

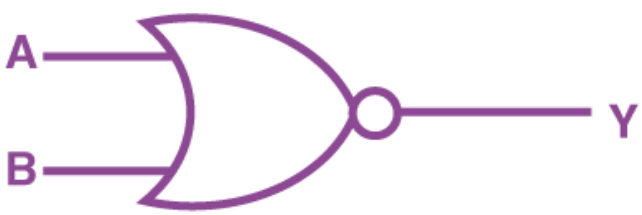


NOR Gate

- **Function:** Outputs 1 only if all inputs are 0 (i.e., the complement of OR).
- **Algebraic Expression:** $f = (x + y)'$
- **Truth Table:**

x	y	f
0	0	1
0	1	0
1	0	0
1	1	0

> **NOR GATE:** A logic gate whose output is the complement of the OR gate's output.



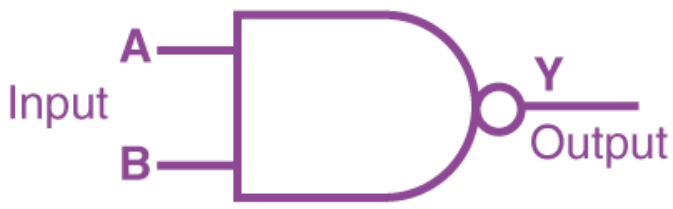
NAND Gate

- **Function:** Outputs 0 only if all inputs are 1 (i.e., the complement of AND).
- **Algebraic Expression:** $f = (x \cdot y)'$

- **Truth Table:**

x	y	f
0	0	1
0	1	1
1	0	1
1	1	0

> **NAND GATE:** A gate that outputs the inverse of the AND gate's result.

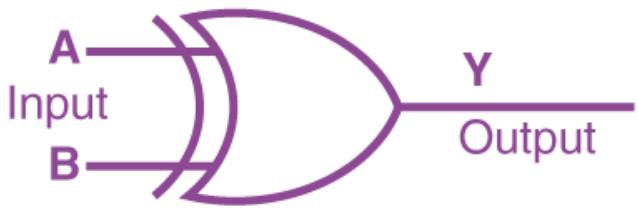


XOR Gate

- **Function:** Outputs 1 if the inputs are different.
- **Algebraic Expression:** $f = x \oplus y = (x \cdot \overline{y}) + (\overline{x} \cdot y)$
- **Truth Table:**

x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

> **XOR GATE:** An exclusive OR gate that outputs 1 when the number of 1's in the inputs is odd.

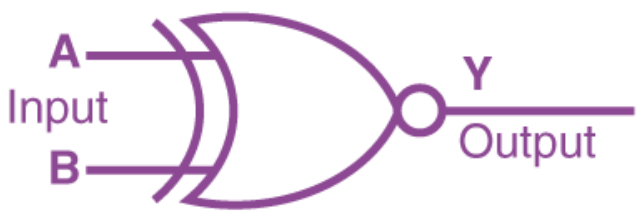


XNOR Gate

- **Function:** Outputs 1 if the inputs are the same.
- **Algebraic Expression:** $f = (x \oplus y)'$
- **Truth Table:**

x	y	f
0	0	1
0	1	0
1	0	0
1	1	1

> **XNOR GATE:** A gate that produces an output of 1 when both inputs are equal; it is the complement of the XOR gate.

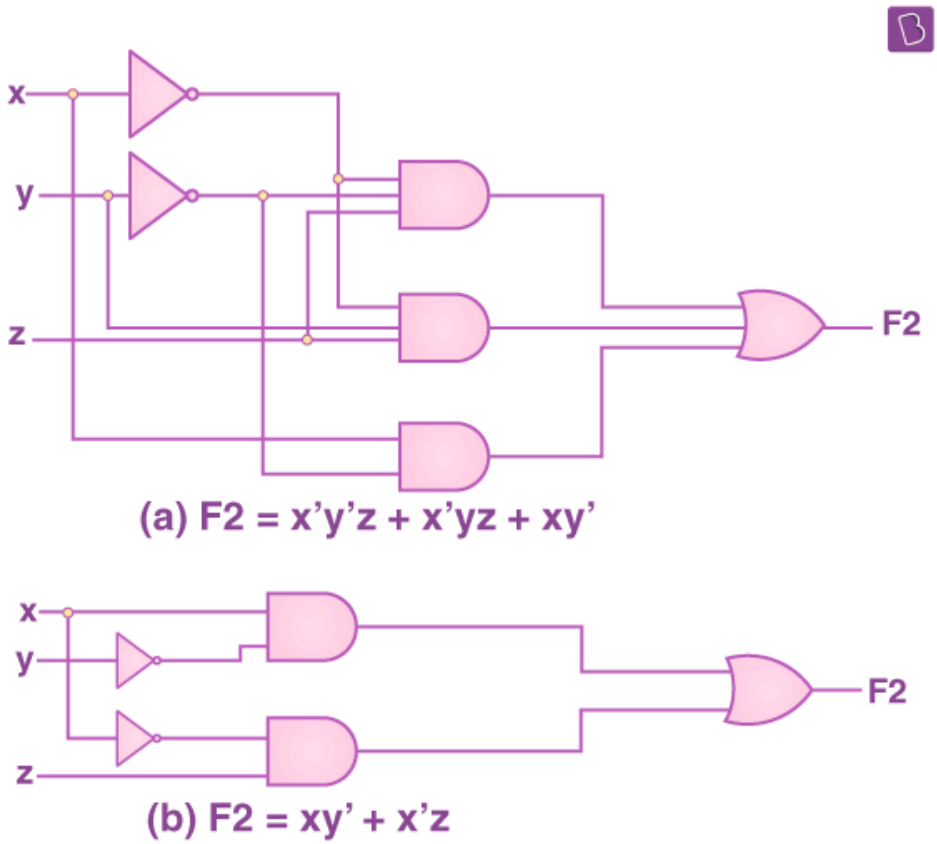


Combinational Digital Systems

These systems are built using logic gates. They produce an output solely based on the current inputs.

- **Simple Two-Input Systems:** Basic circuits that perform operations like AND, OR, and XOR.
- **Multi-Input Systems:** Systems with more than two inputs can be built by combining two-input gates.
- **Design Strategies:**
  - **Sum of Products (SoP):** Expresses the function as an OR of minterms.
  - **Product of Sums (PoS):** Expresses the function as an AND of maxterms.

> **COMBINATIONAL DIGITAL SYSTEM:** A circuit where the output is a function solely of the current input values, with no memory element.



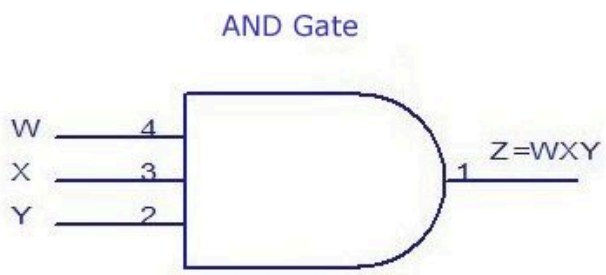
Three-Input Systems

When designing systems with three or more inputs, the complexity increases, but the principles remain the same.

- **Example:** A three-input AND gate outputs 1 only when all three inputs are 1.
- **Design Consideration:** Ensure that the circuit is scalable by breaking down the logic into simpler two-input operations if necessary.

> **THREE-INPUT SYSTEM:** A digital logic circuit that accepts three binary inputs and produces an output based on a specified Boolean function.

3 Input AND Gate



TRUTH TABLE

INPUTS			OUTPUT
W	X	Y	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

General Approach for Building Combinational Digital Systems

The design of combinational circuits generally follows these steps:

- Define the Boolean function using truth tables.
- Express the function in standard forms, such as SoP or PoS.
- Use Boolean algebra to simplify the expression.
- Implement the simplified expression using basic logic gates.

> **DESIGN APPROACH:** Use Boolean algebra and standard forms (SoP or PoS) to design efficient combinational circuits from basic gates.

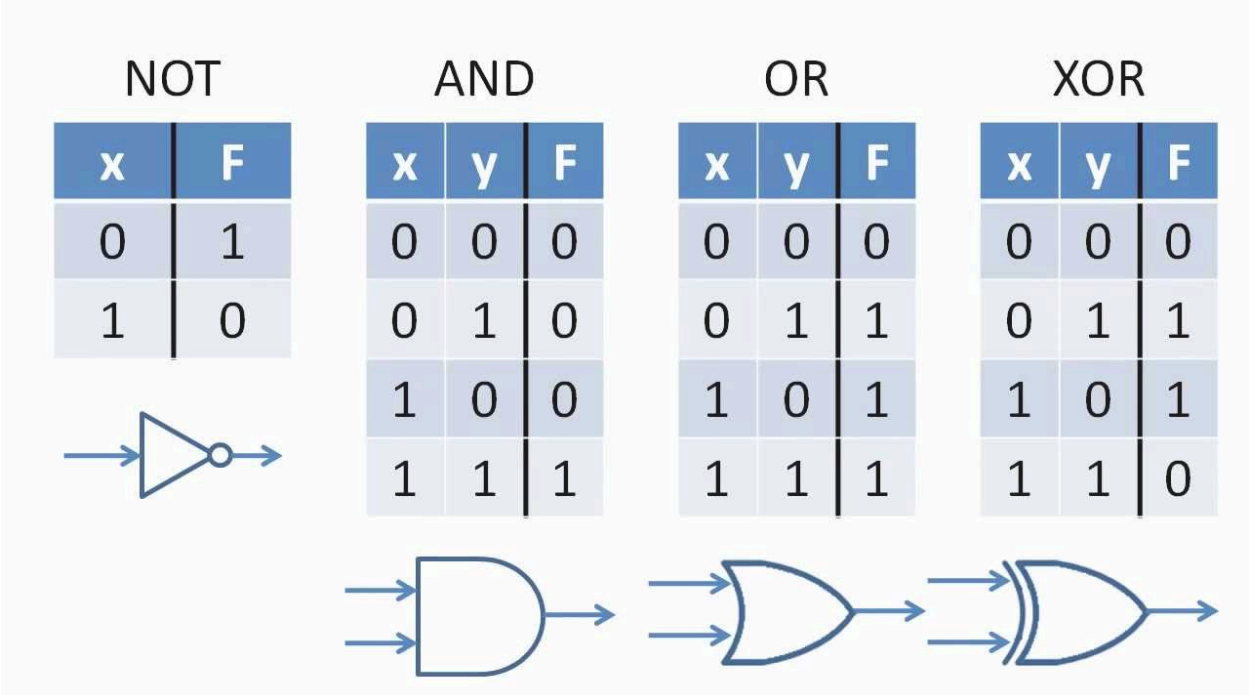
Boolean Algebra

Boolean algebra is the mathematical foundation of digital logic. It uses a set of variables that take on values from $\{0, 1\}$ and is governed by specific operations and laws.

Basic Operations

- **AND ($\cdot$)**
- **OR ($+$)**
- **NOT ($'$)**

> **BOOLEAN ALGEBRA:** A branch of algebra dealing with binary variables and logical operations.



Boolean Algebra Theorems

Identities and Null Elements

- **Identity Laws:**
  $x + 0 = x$ and x $\cdot 1 = x$
- **Null Laws:**
  $x + 1 = 1$ and $x \cdot 0 = 0$

> **IDENTITY AND NULL LAWS:** Fundamental rules that simplify expressions by defining the effect of adding 0 or multiplying by 1, and their opposites.

Idempotency and Complements

- **Idempotent Laws:**
  $x + x = x$ and $x \cdot x = x$
- **Complement Laws:**
  $x + x' = 1$ and $x \cdot x' = 0$

> **IDEMPOTENCY:** The property that combining a variable with itself does not change its value.
>
> **COMPLEMENT:** A variable's complement is the opposite value (if $x = 0$, then $x' = 1$; if $x = 1$, then $x' = 0$).

Involution and Commutativity

- **Involution:**
  $(x')' = x$
- **Commutativity:**
  $x + y = y + x$ and $x \cdot y = y \cdot x$
- **Associativity:**
  $x + (y + z) = (x + y) + z$ and $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

> **INVOLUTION:** The principle that taking the complement twice returns the original value.

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

More Theorems and Distributive Properties

- **Distributive Laws:**
  $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
  $x + (y \cdot z) = (x + y) \cdot (x + z)$
- **Absorption Laws:**
  $x + (x \cdot y) = x$
  $x \cdot (x + y) = x$

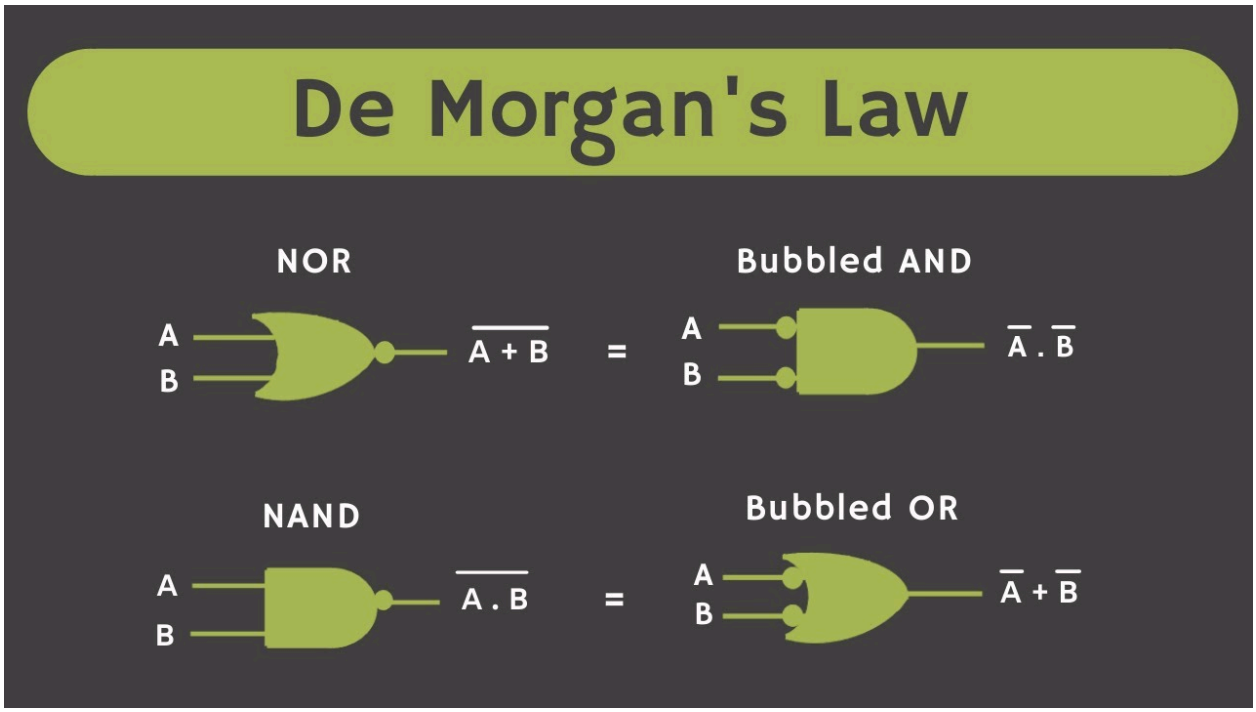
> **DISTRIBUTIVE AND ABSORPTION LAWS:** Rules that allow the reorganization and simplification of Boolean expressions.

DeMorgan's Equivalences and Duality

DeMorgan's Equivalences

- $(x \cdot y)' = x' + y$'
- $(x + y)' = x' \cdot y$'

> **DE-MORGAN'S LAWS:** Fundamental transformations that allow the complement of a conjunction to be expressed as the disjunction of the complements, and vice versa.



Duality Principle

The duality principle states that every Boolean expression remains valid if you swap AND with OR and 0 with 1 throughout the expression.

> **DUALITY:** The principle that the dual of any Boolean expression (by interchanging + and ·, and swapping 0 and 1) is also valid.

Extension to N-Variable Theorems

Boolean algebra extends naturally to functions of n variables:

- Theorems such as:

$$(x_1 + x_2 + \cdots + x_n)' = x_1' \cdot x_2' \cdots x_n'$$

$$(x_1 \cdot x_2 \cdots x_n)' = x_1' + x_2' + \cdots + x_n'$$

- These generalize the two-variable cases to functions with many inputs.

> **NOTE:** N-variable theorems are critical when designing complex digital circuits that involve multiple inputs.

Back to Our Earlier Example

In earlier slides, a Boolean function was presented and simplified using the Sum of Products (SoP) approach. Although we are excluding the detailed minterm/maxterm representations beyond slide 57, it is important to understand that:

- **SoP (Sum of Products)** expresses a Boolean function as an OR of multiple AND terms.
- This method is widely used to implement digital circuits.
- The simplified expressions allow for efficient hardware implementations using basic gates.

> **SUMMARY:** Using Boolean algebra, any Boolean function can be systematically simplified and implemented using a combination of logic gates, ensuring efficient digital circuit design.

Summary

This study material has covered:

- **Combinational Digital Systems:** The basis of digital circuits with outputs depending solely on current inputs.
- **Logic Gates:** Detailed discussion of AND, OR, NOR, NAND, XOR, and XNOR gates, including their functions and truth tables.
- **Multi-Input Systems:** How combinational systems extend to three or more inputs.
- **Design Approaches:** Using Boolean algebra to express functions in standard forms such as Sum of Products (SoP) and Product of Sums (PoS).
- **Boolean Algebra:** Fundamental theorems, identities, and properties (including idempotency, complements, DeMorgan's Laws, duality, and distributivity).
- **Extension to N-Variable Functions:** Generalizing Boolean expressions for more complex systems.

Understanding these topics is crucial for designing and optimizing digital systems, as well as for simplifying and implementing logical functions in hardware.

Self Test

Self-Test: Lecture 4

5. Boolean Function Representations and Circuit Optimization

Objective & Scope

This note covers key topics from slides 14 to 49, focusing on methods to represent and optimize Boolean functions and digital circuits. Topics include:

- Sum of Products (SOP) and Product of Sums (POS) approaches
- Comparison between SOP and POS representations
- Standard forms using minterms and maxterms
- Alternative representations and algebraic simplifications
- Circuit optimization criteria such as literal cost and gate input cost

This note is designed to provide a clear, comprehensive understanding of these fundamental digital design concepts.

Sum of Products (SOP) Approach

> **SOP Approach:** A method of representing Boolean functions where the function is expressed as a sum (OR) of product (AND) terms, each called a minterm.

- **Minterms:**

  > **MINTERM:** A product term in which all variables appear exactly once (either complemented or uncomplemented). For a two-variable function, the minterms are:
  >
  > $$X'Y',\ X'Y,\ XY',\ XY$$

- **Example (Two-Input System):**

  Consider a function f defined by the truth table:

x	y	f
0	0	1
0	1	1
1	0	0
1	1	1

  This function can be represented as:

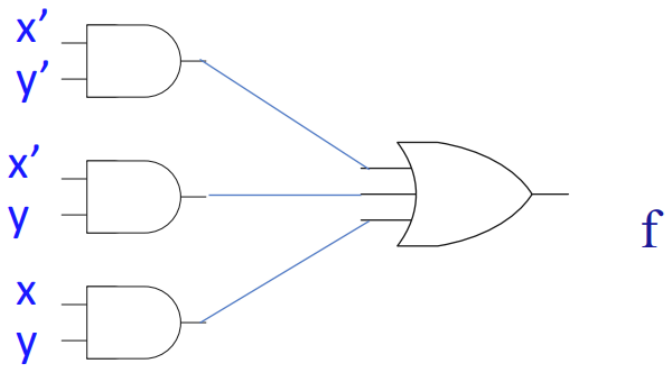
$$f = m_0 + m_1 + m_3$$

  where each m_i is a minterm corresponding to the input combination.

Example

$$f = m_0 + m_1 + m_3$$

$$f = x'y' + x'y + xy$$



x	y	f
0	0	1
0	1	1
1	0	0
1	1	1

Product of Sums (POS) Approach

> **POS Approach:** An alternative representation where the Boolean function is expressed as a product (AND) of sum (OR) terms. Each sum term is called a maxterm.

- **Maxterms:**

  > **MAXTERM:** A sum term that contains every variable exactly once (in complemented or uncomplemented form). For a two-variable function, the maxterms are:
  >
  > $$X + Y,\ X + Y',\ X' + Y,\ X' + Y'$$

- **Example (Two-Input System):**

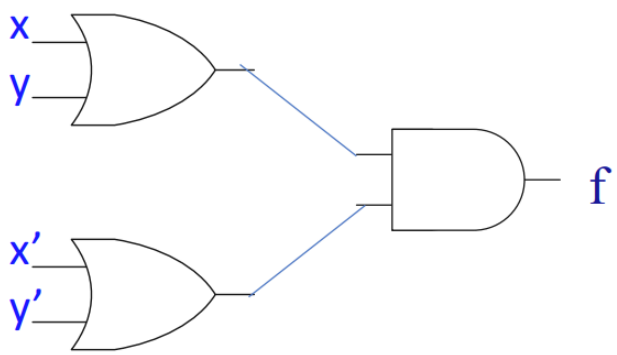
  For a function defined by:

x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

The POS representation may be obtained by first determining the maxterms for the outputs that are 0 and then forming the product.

- Going back to the XOR function
- Apply the bubble trick!

$f = (x+y) \cdot (x'+y')$



x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

SOP vs. POS Representations

- **Comparison:**
  - **SOP (Sum of Products):** Uses minterms; generally leads to an OR of AND terms.
  - **POS (Product of Sums):** Uses maxterms; results in an AND of OR terms.

> **Key Observation:** For any Boolean function, the SOP and POS representations are duals of each other. The duality can be obtained by swapping ANDs with ORs, 0's with 1's, and variables with their complements.

- **Example (XOR Function):**

  The XOR function can be expressed in SOP form as:

$$f = x'y + xy'$$

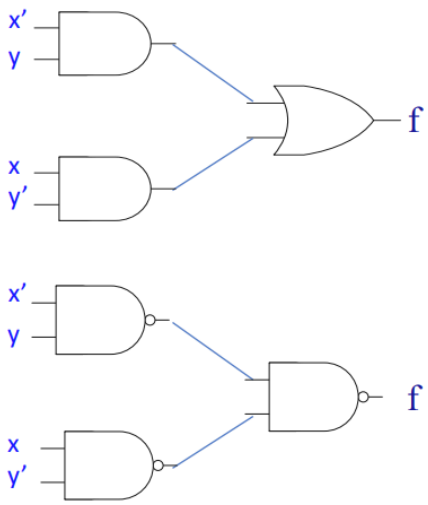
  Its equivalent POS form can be derived as:

$$f = (x + y) \cdot (x' + y')$$

This dual representation is useful when designing circuits using only NAND or NOR gates.

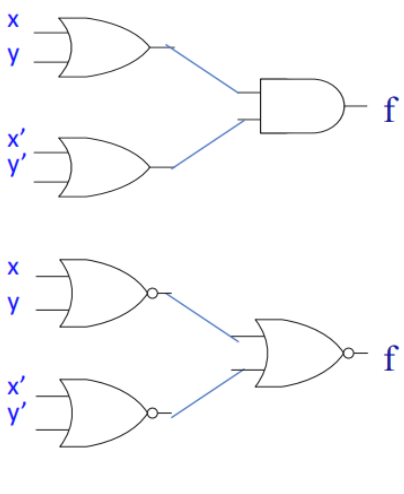
SOP

$f = x'y + xy'$



POS

$f = (x'+y')\,(x+y)$



x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

Standard Forms: Minterms and Maxterms

- **Minterms:**
  - There are 2^n minterms for an n-variable function.
  - Every Boolean function can be expressed as the sum of its minterms.
  - Missing minterms correspond to the complement function.
- **Maxterms:**
  - There are 2^n maxterms for an n-variable function.
  - Every Boolean function can also be represented as the product of its maxterms.
  - A function that includes all maxterms equals 0.

> **Standard Form Representations:** A canonical SOP expression is written as $f = \Sigma m(i)$, while a canonical POS expression is written as $f = \Pi M(i)$.

- **Example (Three-Variable Function):**
  For a function $F(x, y, z)$, one might have:

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

  and its complement can be expressed using maxterms.

Alternative Representations and Algebraic Simplification

- **Algebraic Simplification:**

  Boolean algebra theorems (e.g., absorption, DeMorgan's laws) are used to reduce expressions, which directly leads to simplified circuits.

  - **Absorption Law Examples:**

$$X + XY = X \quad \text{and} \quad X(X + Y) = X$$

  - **DeMorgan's Equivalences:**

$$(X \cdot Y)' = X' + Y' \quad \text{and} \quad (X + Y)' = X' \cdot Y'$$

- **Applying the Bubble Trick:**

  A technique to derive the POS form from a given SOP form by complementing and then re-complementing the function.

- **Canonical to Simplified Form:**

  Starting with a canonical sum (or product) and then applying algebraic methods to minimize the literal count, which directly impacts circuit cost.

Circuit Optimization: Literal and Gate Input Cost

- **Optimization Goals:**

  Minimize the hardware cost (number of gates and inputs) while ensuring correct logical functionality.

- **Literal Cost (L):**

> **LITERAL COST:** The total number of literal appearances (variables and their complements) in a Boolean expression.

- **Gate Input Cost (G):**

> **GATE INPUT COST:** The total number of inputs to the gates used in the circuit implementation. Sometimes, the cost with NOT gates is also considered (GN).

- **Example:**

  For a circuit implementing

$$F = BD + AB'C + AC'D'$$

  the literal cost might be 8 if each variable appearance is counted. Gate input cost is calculated by summing the inputs for each gate used.

- **Choosing the Best Implementation:**

  A lower literal and gate input cost often means a simpler, more efficient circuit. Designers may choose between alternative representations (SOP vs. POS) based on these cost criteria.

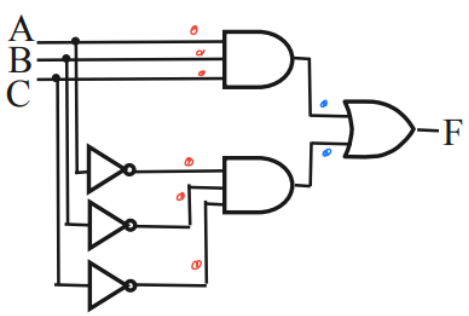
Cost Criteria - Example

- F = A B C + A'B'C'

  L = 6

  G = 6+2 = 8

  GN = 8+3 = 11

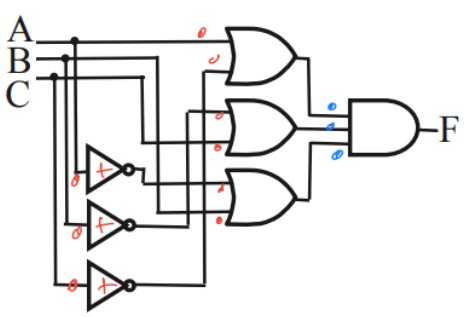


- F = (A + C')(B' + C)(A' + B)

  L = 6

  G = 6+3 = 9

  GN = 9+3 = 12



Final Summary & Key Takeaways

- **Representation Methods:**
  - **SOP** uses minterms and provides an OR of AND terms.
  - **POS** uses maxterms and provides an AND of OR terms.
- **Duality and Equivalence:**

  Understanding the duality between SOP and POS forms helps in converting and optimizing Boolean expressions.
- **Standard Forms:**

  Canonical forms using minterms and maxterms provide a systematic way to represent any Boolean function, serving as a starting point for simplification.
- **Circuit Optimization:**

  Techniques such as algebraic simplification, the bubble trick, and cost analysis (literal and gate input cost) are crucial for designing efficient digital circuits.

This comprehensive note consolidates the key points from slides 14 to 49, equipping you with the foundational knowledge to represent and optimize Boolean functions for digital circuit design.

6. Karnaugh Maps and Boolean Function Optimization

Objective & Scope

This note covers the use of Karnaugh Maps (K-maps) and related techniques for optimizing Boolean functions and digital circuits. It focuses on:

- Understanding the basic structure and purpose of K-maps
- Using two-variable and three-variable K-maps for function representation
- Alternative map labeling for improved pattern recognition
- Combining squares (grouping) to simplify Boolean expressions
- Practical examples of K-map simplification and circuit optimization

This comprehensive note is intended to provide both the theoretical foundations and practical applications needed for effective digital circuit design.

Karnaugh Maps (K-maps)

> **Karnaugh Map (K-map):** A graphical tool that reorganizes a Boolean function's truth table into a grid format where adjacent cells differ by only one variable. This structure

> enables visual grouping (combining) of 1's to simplify Boolean expressions.

- **Structure:**
  - Composed of squares, each representing a minterm.
  - Cells are arranged so that adjacent ones (horizontally or vertically) differ by a single bit (Gray code ordering).
  - Can be viewed as a reorganized truth table or a warped Venn diagram.

A. SOP: -

A \ B	$\overline{B}$ 0	B 1
$\overline{A}$ 0	$\overline{A}.\overline{B}$	$\overline{A}.B$
A 1	$A.\overline{B}$	A.B

B. POS: -

A \ B	B 0	$\overline{B}$ 1
A 0	A+B	$A+\overline{B}$
$\overline{A}$ 1	$\overline{A}+B$	$\overline{A}+\overline{B}$

Uses of Karnaugh Maps

> **Key Uses:**
>
> - **Simplification:** Derive optimum or near-optimum SOP (Sum of Products) or POS (Product of Sums) expressions.
> - **Optimization:** Minimize the literal cost and gate input cost in circuit implementations.
> - **Visualization:** Make the relationships between minterms clear, thereby aiding in the identification of simplification opportunities.
> - **Design:** Serve as an instructive tool for manually optimizing small digital circuits before applying computer-aided techniques.

Two-Variable K-Maps

- **Basic Layout:**
  - A 2-variable K-map has 4 cells corresponding to the minterms for variables x and y:
    - $m_0 : x'y$'
    - $m_1 : x'y$
    - $m_2 : xy$'
    - $m_3 : xy$
- **Adjacency:**
  - Cells adjacent either horizontally or vertically differ by only one variable.
  - This property allows adjacent cells containing 1's to be grouped for simplification.

Two Variable K-Maps

- Minterm $\mathbf{m_0}$ and minterm $\mathbf{m_1}$ are "**adjacent**"
  - They differ in the value of the variable **y**
- Similarly,
  - Minterms $\mathbf{m_0}$ and $\mathbf{m_2}$ differ in the **x** variable
  - Minterms $\mathbf{m_1}$ and $\mathbf{m_3}$ differ in the **x** variable
  - Minterms $\mathbf{m_2}$ and $\mathbf{m_3}$ differ in the **y** variable

	y = 0	y = 1
x = 0	m_0 x'y'	m_1 x'y
x = 1	m_2 xy'	m_3 xy

K-Maps and Truth Tables

- **Relationship:**
  - A K-map is simply a reordering of a truth table to expose adjacent minterm groupings.

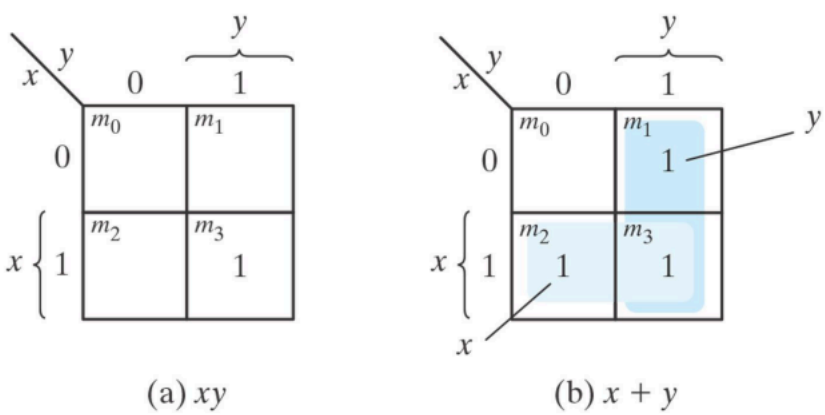
- This organization helps in directly translating truth table information into simplified Boolean expressions.

- **Representation:**
  - Values from a truth table are entered into the K-map, marking 1's for minterms where the function is true.
  - These marked cells are then grouped to form simplified product terms.

Input Values (x,y)	Function Value F(x,y)
0 0	a
0 1	b
1 0	c
1 1	d

	y = 0	y = 1
x = 0	a	b
x = 1	c	d

K-Map Function Representation



(a) xy (b) $x + y$

Three-Variable K-Maps

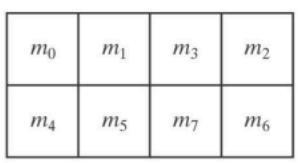
- **Layout:**
  - A 3-variable K-map contains 8 cells.

- The arrangement is designed so that each cell is adjacent to those that differ by a single variable change.
- Commonly, variables are ordered such that one dimension (e.g., rows) represents one variable while columns represent the other two in Gray code order.

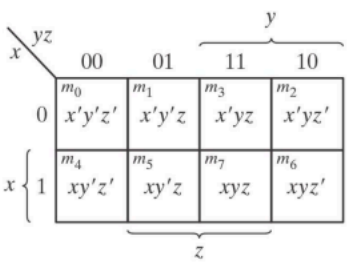
- **Adjacency in 3-Variable Maps:**
  - Allows grouping of cells into rectangles containing 2, 4, or 8 cells (powers of 2) for minimization.
  - The map can be visualized as a cylinder or a torus where the edges wrap around, preserving adjacency.

Three Variable Maps

- A three-variable K-map:



(a)



(b)

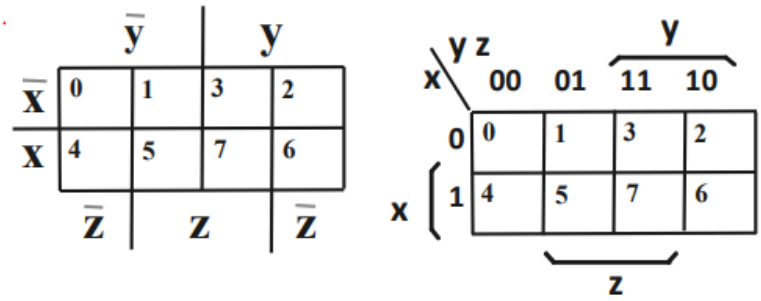
- Note that if the binary value for an index differs in one-bit position (Gray code sequence), the minterms are adjacent on the K-Map

Alternative Map Labeling

> **Alternative Labeling:** Adjusting the labels or ordering of variables in a K-map can make certain groups more apparent. This includes:
>
> - Changing the sequence of variable representation.
> - Using different orientations to emphasize adjacent groupings.

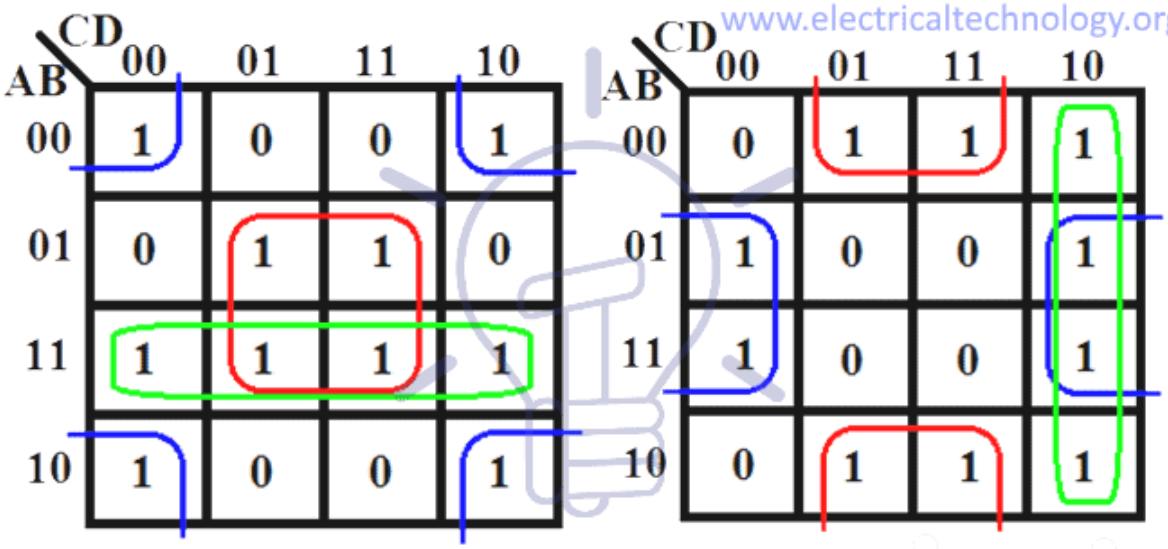
- **Purpose:**
  - Facilitates easier identification of common patterns.

- Enhances clarity in reading product terms from the map.



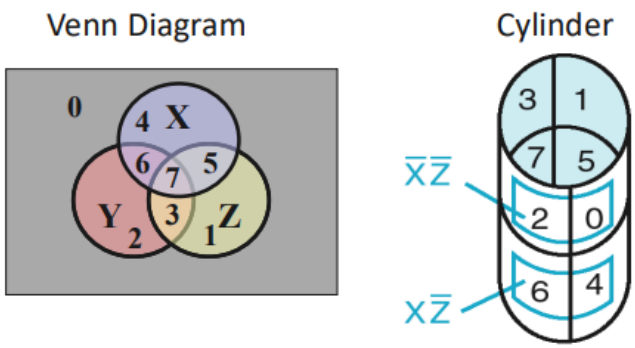
Combining Squares (Grouping)

- **Concept:**
  - **Grouping:** The process of combining adjacent 1's (marked cells) in the K-map to form larger rectangles.
  - **Goal:** Reduce the number of literals (variables) in each product term.
- **Grouping Guidelines:**
  - **Single Cell:** Represents a minterm with all variables.
  - **Pair of Adjacent Cells:** Can eliminate one variable.
  - **Four Adjacent Cells:** Can reduce a term to a single variable or even represent a constant.
  - **Edge Wrapping:** Cells on the edges of the K-map are considered adjacent if they wrap around.



Three-Variable Maps

- Topological warps of 3-variable K-maps that show *all* adjacencies:



Final Summary & Key Takeaways

- **Karnaugh Maps (K-maps)** are powerful tools for Boolean function optimization, particularly effective for functions with a small number of variables.
- **Two-variable and three-variable maps** provide a structured method for visualizing and simplifying Boolean expressions.
- **Grouping (combining squares)** reduces the number of literals in product terms, thereby minimizing circuit complexity.

- **Alternative labeling and visualization techniques** help in recognizing patterns and adjacencies that may not be immediately obvious.
- **Practical Applications:**
  K-maps are used to derive simplified SOP and POS forms, which lead to lower literal and gate input costs in digital circuit implementations.

This note consolidates the key concepts from slides 51 to 67, equipping you with a clear understanding of how to apply K-map techniques for Boolean function optimization in digital system design.

7. Karnaugh Map Simplification & Prime Implicant Optimization

Objective & Scope

This note focuses on methods for simplifying Boolean functions using Karnaugh maps (K-maps) and optimizing their representations through prime implicants. We cover techniques for three-variable and four-variable K-maps, learn how to identify minterms and maxterms, practice prime implicant extraction, and explore an optimization algorithm to select a cost-effective solution. This material corresponds to slides 15 to 23.

Three-Variable Map Simplification

- **Overview:**
  The goal is to use a 3-variable K-map to simplify a Boolean function by grouping adjacent 1's.
- **Key Concepts:**
  - Grouping adjacent cells in powers of 2 (1, 2, 4, ...) minimizes the number of literals in the product terms.
  - Simplified terms (prime implicants) are derived from these groups.

> **Simplification Principle:** Group adjacent 1's in the K-map to form the largest possible rectangles; each rectangle corresponds to a product term with fewer variables.

Three-Variable Map Minterms/Maxterms

- **Minterms and Maxterms:**
  - **Minterm:** A product term where every variable appears exactly once (in true or complemented form).
  - **Maxterm:** A sum term that includes every variable exactly once.
- **Usage in K-maps:**
  Minterms are used in the Sum of Products (SOP) representation; maxterms are used in the Product of Sums (POS) representation.
- **Example:**
  A function may be written as a sum of specific minterms extracted from the K-map, or as a product of its maxterms.

Key Idea: Expressing a function in canonical form provides a starting point for minimization via grouping.

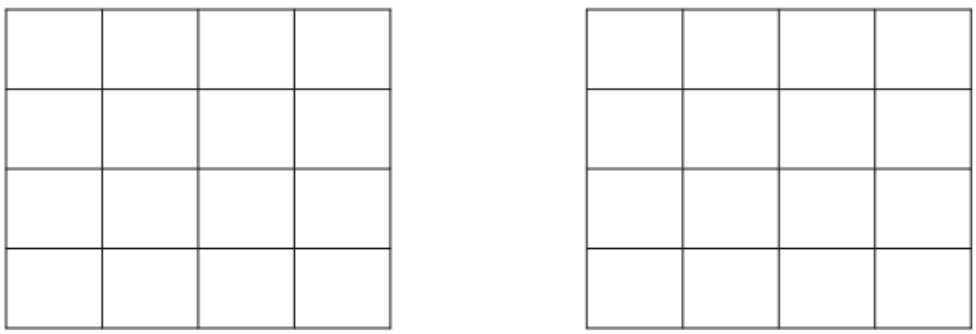
Four-Variable Maps

- **Introduction to Four-Variable K-Maps:**
  Four-variable K-maps extend the principles of 3-variable maps with 16 cells.
- **Layout and Adjacency:**
  - Cells are arranged so that every adjacent pair differs by only one variable.
  - Enables grouping of 1's into larger rectangles (groups of 2, 4, 8, or 16).

Note: Understanding four-variable maps is essential for functions of higher complexity, as the same grouping principles apply.

Four-Variable Maps

F(A,B,C,D)



Four Variable Terms

- **Definition:**
  A "term" in a Boolean expression derived from a four-variable K-map.
- **Grouping Effects:**
  - A single cell represents a minterm with 4 literals.
  - A pair (2 cells) reduces one variable (3 literals remain).
  - A group of four cells results in a term with 2 literals.
  - Larger groups (e.g., eight cells) can reduce the term to a single literal.
- **Optimization Impact:**
  Grouping reduces the literal count, lowering both the complexity and the hardware cost of the resulting circuit.

Optimization Tip: Always look for the largest possible grouping to minimize the expression.

Karnaugh-map Usage

- **Procedure for Using K-Maps:**
  - Plot the function's output (1's for true minterms) into the K-map.

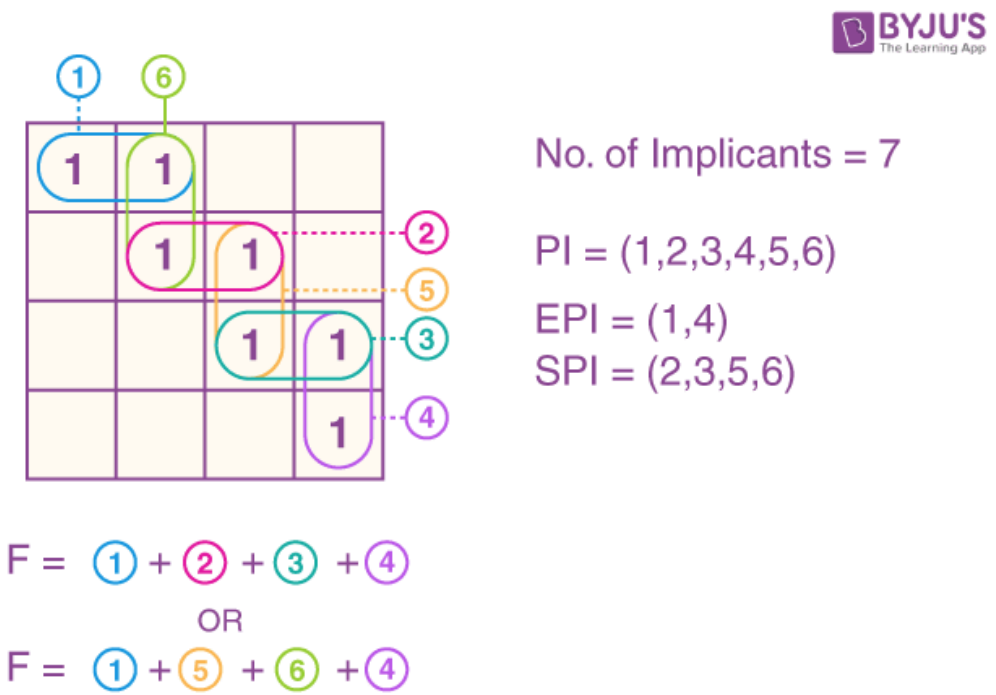
    - Circle or highlight the largest rectangular groups of 1's (the groups must contain 2^n cells).
    - Translate each group into its corresponding product term.
- **Benefits:**
    - Reduces the number of terms and literals.
    - Provides a visual method for function minimization.

Usage Guidelines: The groups (or prime implicants) must cover all 1's in the map, and overlapping groups can sometimes yield a more optimal solution.

Example of Prime Implicants

- **Prime Implicant:**
  A group (rectangle) on the K-map that cannot be combined with adjacent groups to form a larger group.
- **Identification:**
    - Mark all groups of 1's.
    - Identify which groups cover 1's that no other group covers (these are essential prime implicants).
- **Example Process:**
  For a given function on a 3-variable or 4-variable map, list all potential groups and then narrow them down to prime implicants based on their coverage and size.

> **Remember:** Prime implicants are the building blocks of a minimized Boolean expression.



Optimization Algorithm

- **Optimization Process:**
  1. **Find all prime implicants:** List every possible grouping on the K-map.
  2. **Identify essential prime implicants:** Determine which groups cover minterms uniquely.
  3. **Select a minimum cost set:** From the remaining non-essential prime implicants, choose the ones that cover all minterms with the lowest overall cost (considering literal and gate input costs).
- **Selection Rule:**
  - Minimize overlap among selected prime implicants.
  - Ensure each chosen prime implicant includes at least one minterm not covered by another.
- **Goal:**
  Obtain a simplified Boolean expression that minimizes hardware implementation cost.

> **Optimization Insight:** The algorithm ensures that the final solution is not only logically correct but also cost-effective in practical circuit design.

Final Summary & Key Takeaways

- **Karnaugh Maps** are a powerful graphical tool for Boolean function simplification.
- **Grouping in K-maps** reduces the number of literals in an expression, which directly lowers the complexity and cost of digital circuits.
- **Prime Implicants** represent the core simplified groups that cannot be further combined.
- **Optimization Algorithms** help select the best combination of prime implicants, balancing coverage and cost.

This note consolidates the key points from slides 15 to 23, offering a clear pathway from K-map simplification to prime implicant optimization for effective Boolean function reduction.

8. Digital Logic Optimization and Karnaugh Map Techniques

Introduction

Digital systems often require optimized logic circuits to reduce complexity, lower manufacturing costs, and improve performance. Simplification techniques such as Karnaugh maps (K-maps) help transform complex Boolean functions into simpler, more cost-effective forms. By applying systematic methods for grouping minterms and leveraging selection rules, designers can minimize the number of required logic gates and inputs.

Selection Rules and Prime Implicants

The foundation of K-map optimization lies in identifying **prime implicants**—groups of adjacent 1s (minterms) that can be combined to form simplified product terms. A key strategy involves:

- **Choosing prime implicants that minimize overlap:** Each selected implicant should cover at least one minterm not shared with any other implicant.
- **Ensuring minimal literal cost:** The goal is to reduce the number of literals (variables or their complements) in each product term, thus lowering the overall gate input

cost.

This selection rule is critical for deriving a final expression that is both optimal and implementable with fewer gates.

Simplifying Four-Variable Functions

For functions defined over four variables, the Karnaugh map provides a clear visual tool to group minterms:

- **Grouping Adjacent Minterms:** By forming groups (or rectangles) that contain 2, 4, or 8 cells, the number of literals in each resulting product term is reduced.
- **Extracting Simplified Expressions:** Depending on how the groups are formed, you can derive either a Sum-of-Products (SOP) or a Product-of-Sums (PoS) expression.

The process involves visually identifying the largest possible groups that cover all 1s in the map while avoiding unnecessary overlap.

Converting to Product-of-Sums (PoS)

When a design requires a Product-of-Sums form—often for specific gate implementations like NOR-only circuits—the following steps are used:

- **Derive the Complement:** First, express the complement of the function in SOP form.
- **Apply De Morgan's Law:** Complement the expression to convert it into PoS form.

This approach yields a PoS expression that can simplify the circuit design by reducing the number of required gate inputs.

Techniques for Functions with More Variables

As functions grow to five or more variables, Karnaugh maps become more complex. To manage this:

- **Partition the Map:** Divide the map into sections based on the value of one or more variables. For example, a five-variable function may be split into two four-variable maps corresponding to a variable being 0 or 1.

- **Apply Standard Grouping:** Within each partition, the usual grouping techniques are used to combine adjacent minterms.

This segmentation enables the application of familiar two-level optimization techniques even for higher-variable functions.

Incorporating Don’t Care Conditions

In many practical designs, certain input combinations never occur or their outputs are irrelevant. These are marked as **don’t care conditions**:

- **Flexible Grouping:** Don’t cares can be treated as either 0 or 1 in the K-map, allowing for larger groups that reduce the overall literal count.
- **Cost Reduction:** By including don’t care cells in groups, the final logic expression often has fewer terms and lower gate input cost.

For example, in a BCD (Binary-Coded Decimal) circuit, only the codes 0000 to 1001 are valid. The remaining combinations (1010 to 1111) are don’t cares, which can be used to simplify the circuit.

Optimized Selection with Don’t Cares

Integrating don’t care conditions into the selection process further enhances optimization:

- **Larger Prime Implicants:** By including don’t care cells, groups can be expanded, which simplifies the final expression.
- **Minimized Overlap:** The selection rule is applied with the additional flexibility of choosing groups that cover both required minterms and don’t care conditions.

This method leads to a design with even lower gate input costs and reduced overall complexity.

Practical Application: 4-Bit Prime Number Detector

One practical application of these optimization techniques is the design of a 4-bit prime number detector:

- **Function Definition:** The detector identifies prime numbers by outputting a high signal when the 4-bit input corresponds to a prime number.

- **Mapping Minterms:** The function is defined by minterms corresponding to prime numbers (e.g., 2, 3, 5, 7, 11, 13). Using a K-map, these minterms are grouped and simplified.
- **Resulting Expression:** The final simplified expression requires fewer gates, making the detector both efficient and cost-effective.

Practical Application: 4-Bit BCD Prime Number Detector

Adapting a prime number detector for BCD inputs requires additional considerations:

- **Valid Input Range:** BCD inputs range from 0000 to 1001, so the K-map is constructed only for these values.
- **Handling Don't Cares:** Inputs outside the valid range (typically 1010 to 1111) are marked as don't cares. These conditions are used to simplify the grouping on the K-map.
- **Optimized Design:** The resulting logic function is simpler, ensuring the detector works accurately within the decimal range while using minimal hardware resources.

Advanced Optimization Algorithms

The process of Boolean function optimization can be summarized by the following steps:

- **Identify All Prime Implicants:** List every possible grouping of adjacent 1s in the K-map.
- **Determine Essential Prime Implicants:** Select those groups that cover minterms which no other group covers.
- **Select a Minimal Cover:** Choose a combination of prime implicants that covers all required minterms with minimal overlap.
- **Calculate Gate Input Cost:** Evaluate the final expression in terms of the number of gate inputs required, aiming to minimize this cost.

These steps ensure that the final implementation is both optimal and efficient, balancing simplicity with functionality.

Conclusion

Karnaugh map techniques are vital for simplifying Boolean functions in digital circuit design. By carefully applying selection rules, managing don't care conditions, and optimizing groupings in both SOP and PoS forms, designers can create circuits that are not only functionally correct but also cost-effective and efficient. These methods underpin many modern digital systems, contributing to better performance and reduced resource usage in practical applications.

9. Exclusive OR, Adder Circuits, and Digital Addition

Exclusive OR (XOR) Fundamentals

> **XOR:** The Exclusive OR (XOR) operation outputs true only when the inputs differ. Its Boolean expression is given by:
>
> $$F = X \cdot Y' + X' \cdot Y$$

- **Truth Table:**

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

> **Remark:** Some useful identities of XOR include:
>
> - $X \oplus 0 = X$

- $X \oplus X = 0$
- $X \oplus Y = Y \oplus X$
- $X \oplus 1 = X'$
- Associativity: $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$

Exclusive OR

Remember Exclusive-OR or XOR gate?

F=XY' + X'Y

X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

Applications of XOR in Digital Logic

Binary Addition and the Half Adder

> **HALF ADDER:** A half adder is a digital circuit that adds two one-bit binary numbers. It uses an XOR gate to compute the sum and an AND gate to generate the carry.

- **Equations:**
  - Sum:

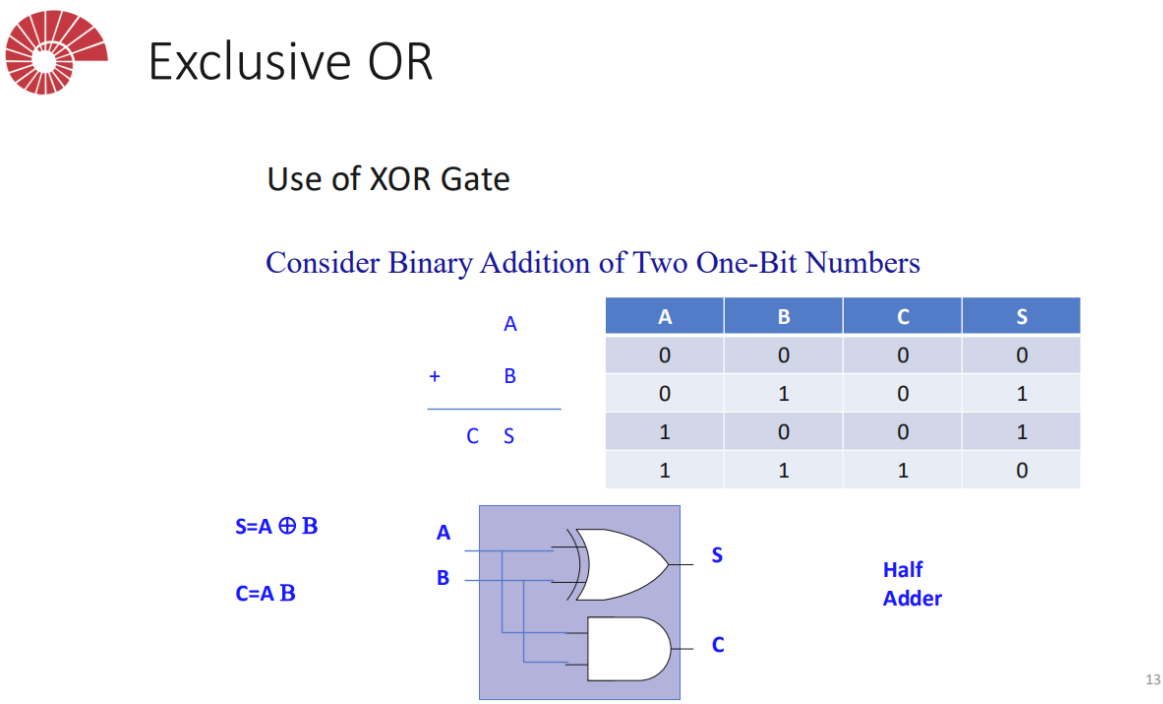
$$S = A \oplus B$$

  - Carry:

$$C = A \cdot B$$

- **Truth Table for Half Adder:**

A	B	$S = A \oplus B$	$C = A \cdot B$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Extending XOR to Multi-Input Functions

> **MULTI-INPUT XOR:** A multi-input XOR gate computes the parity (odd or even) of its inputs. For example, a 3-input XOR function is defined as:
>
> $$F = X \oplus Y \oplus Z$$
>
> and can be implemented as $(X \oplus Y) \oplus Z$.

- **Properties:**
  - The output is 1 if an odd number of inputs are 1 (odd function).

- For a 4-input XOR, the output is 1 if the number of ones is odd.
- The Exclusive-NOR (XNOR) function is the complement of XOR and outputs true when the number of ones is even.

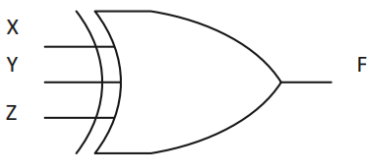
> **Remark:** Multi-input XOR functions are widely used for parity checking and error detection, although they may incur longer delay lines in large-scale implementations.

Exclusive OR

Extending 2 input XOR gate to 3 input XOR gate

F= X⊕ Y ⊕Z

F= (X⊕ Y) ⊕Z



K-Map

X \ YZ	00	01	11	10
0	0	1	0	1
1	1	0	1	0

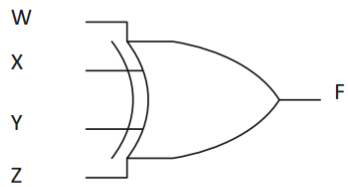
F = XY'Z' + X'Y Z' + XYZ + X'Y' Z

Checker Board Pattern!

15

Exclusive OR

How about 4 input XOR gate



F= W ⊕ X ⊕ Y ⊕Z

F= (W ⊕ X⊕ Y) ⊕Z

Checker Board Pattern Again!

XOR= Odd Function

WX \ YZ	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Full Adder Circuit

> **FULL ADDER:** A full adder is a digital circuit that adds three bits (two significant bits and an input carry) to produce a sum and an output carry.

- **Equations:**
  - Sum:

$$S = (A \oplus B) \oplus C_{in}$$

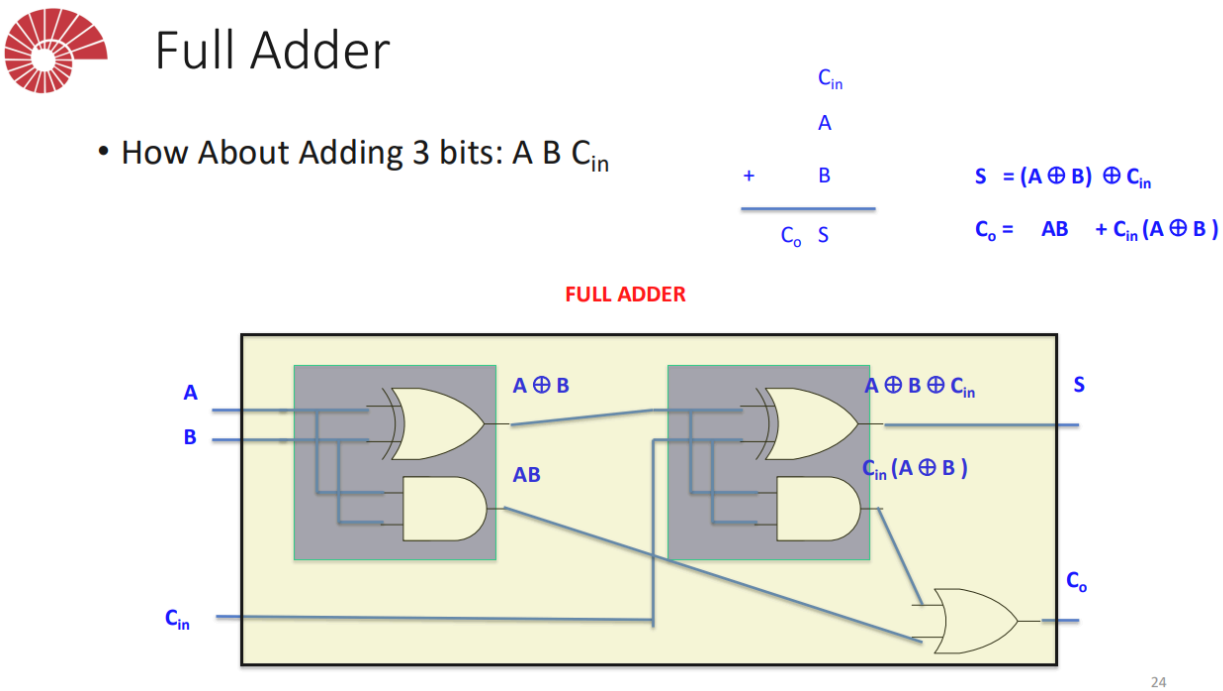
  - Carry-out:

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

- **Truth Table for Full Adder:**

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

A	B	C_{in}	S	C_{out}
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



Larger-Scale Addition

> **Definition:** Multi-bit addition (such as adding 32-bit numbers) is achieved by cascading full adders. The least significant bit (LSB) is typically computed using a half adder, while full adders are used for the remaining bits.

- **Concept:**
  - Each full adder receives a carry input from the previous (less significant) stage.
  - The final carry-out from the most significant stage represents an overflow if present.

> **Remark:** A 32-bit adder can be constructed by connecting 31 full adders in series after a half adder for the LSB, resulting in a combinational circuit with 33 outputs (including the final carry).

Conclusion

This comprehensive note has covered the critical concepts and applications of the XOR operation in digital logic. Key takeaways include:

- The fundamental operation and identities of XOR.
- How XOR is used in designing half adders for binary addition.
- The extension of XOR to multi-input functions for parity checking.
- The construction and functioning of full adders, which are essential for multi-bit addition.
- The overall architecture of larger-scale adders, which are fundamental in digital system design.

Understanding these concepts is vital for designing efficient digital circuits and performing accurate binary arithmetic operations.

10. Digital Decoders: Architecture, Expansion, and Applications in Circuit Design

32-Bit Adder Hardware

> **32-Bit Adder Hardware:** A digital circuit designed to add two 32-bit binary numbers. This hardware typically uses a half adder for the least significant bit and cascaded full adders for the remaining bits, resulting in a 33-bit sum.

- **Further Understanding:**
  - The design leverages modular construction to simplify complex arithmetic operations.
  - It is a building block for larger adders, such as 64-bit adders, by combining two 32-bit units.

Larger Scale Subtraction

> **Larger Scale Subtraction:** A method to subtract binary numbers by converting the subtrahend to its two's complement and then adding it to the minuend.

- **Further Understanding:**
  - This approach simplifies hardware design by allowing the use of an adder circuit for both addition and subtraction.
  - It minimizes the need for separate subtraction circuitry, reducing complexity and cost.

Two's Complement of B

> **Two's Complement:** A binary representation for negative numbers, obtained by inverting all bits of a number and adding one.

- **Further Understanding:**
  - Using two's complement simplifies the arithmetic operation by turning subtraction into addition.
  - It is the standard method for representing signed numbers in most digital systems.

32-Bit Subtractor

> **32-Bit Subtractor:** A circuit that subtracts one 32-bit binary number from another using the two's complement method.

- **Further Understanding:**
  - It uses the same architecture as the 32-bit adder by first converting the subtrahend into its two's complement.
  - The result includes a carry (or borrow) signal which can be used for error detection or overflow checking.

4-Bit Adder/Subtractor

> **4-Bit Adder/Subtractor:** A compact arithmetic unit capable of performing both addition and subtraction on 4-bit numbers.

- **Further Understanding:**

- It typically incorporates a control signal to select between addition and subtraction modes.
- This unit is often used in educational examples to illustrate basic binary arithmetic and overflow conditions.

Overflow

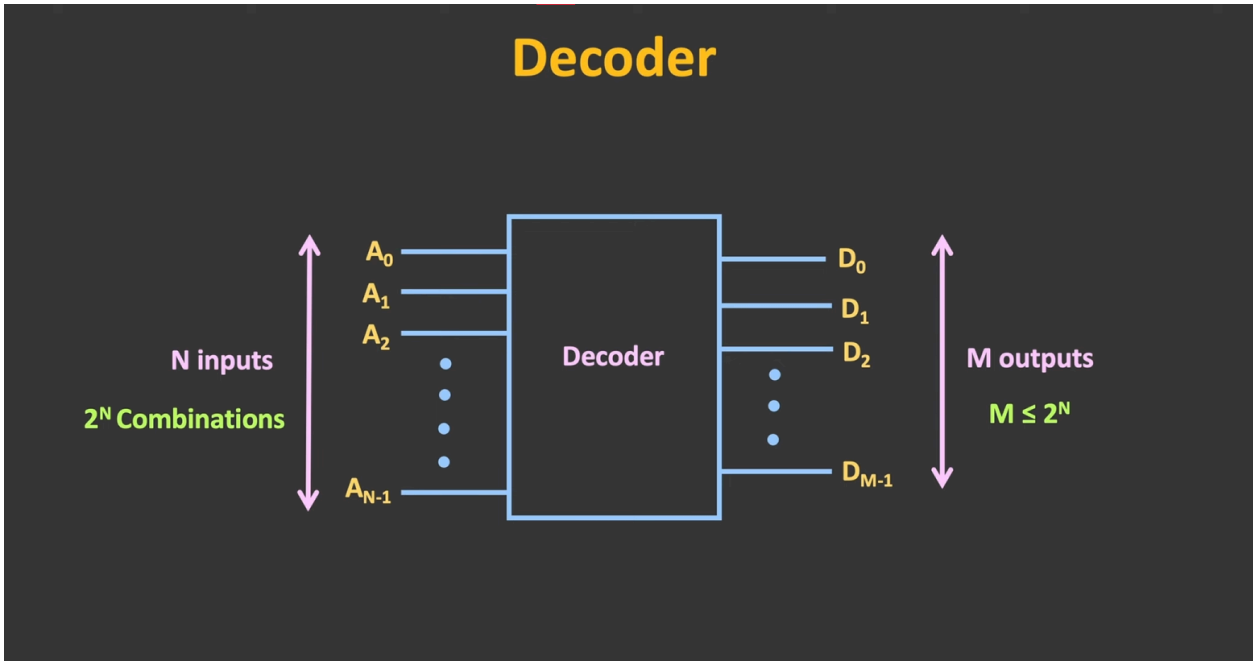
> **Overflow:** A condition that occurs when the result of an arithmetic operation exceeds the maximum value representable with a given number of bits.

- **Further Understanding:**
  - In a signed 4-bit system (range $[-8, 7]$), overflow is detected when the carry into the most significant bit differs from the carry out.
  - Proper overflow detection is essential to ensure the integrity of arithmetic operations in digital systems.

Decoders

> **Decoders:** Combinational circuits that convert n-bit input codes into up to 2^n unique output lines.

- **Further Understanding:**
  - They are crucial in digital systems for tasks such as memory addressing, instruction decoding, and signal routing.
  - A decoder activates a single output corresponding to the binary value of the input.



Binary 2-to-4 Decoder

> **Binary 2-to-4 Decoder:** A specific decoder that takes 2 input bits and produces 4 unique outputs.

- **Key Logic Equations:**
  - $m_0 = I_1' I_0'$
  - $m_1 = I_1' I_0$
  - $m_2 = I_1 I_0'$
  - $m_3 = I_1 I_0$
- **Further Understanding:**
  - It forms the foundation for more complex decoding schemes and is often used to demonstrate the basics of decoder operation.

Decoder

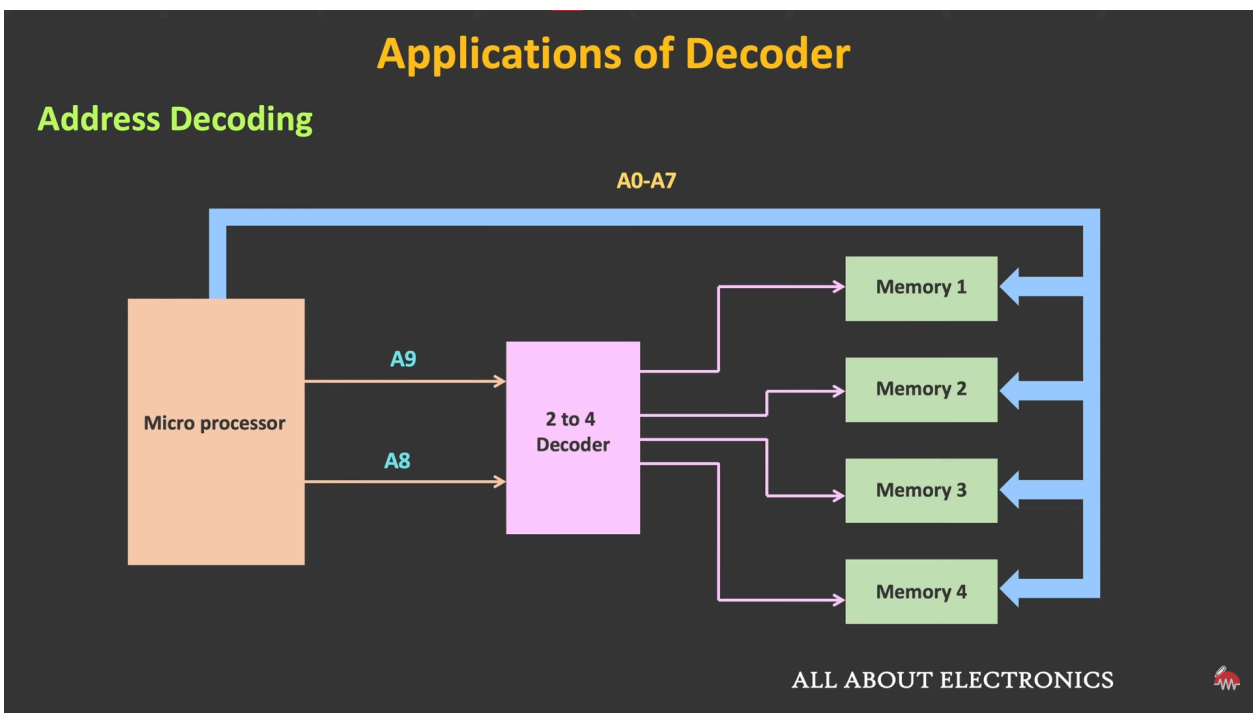
> **Decoder:** A circuit that maps encoded input signals to a unique active output, ensuring that only one output is activated at any time.

- **Further Understanding:**
  - This functionality is essential for control and selection tasks in digital circuits, such as activating memory cells or peripheral devices.

Addressing

> **Addressing:** The process of selecting specific memory locations or devices using decoder circuits.

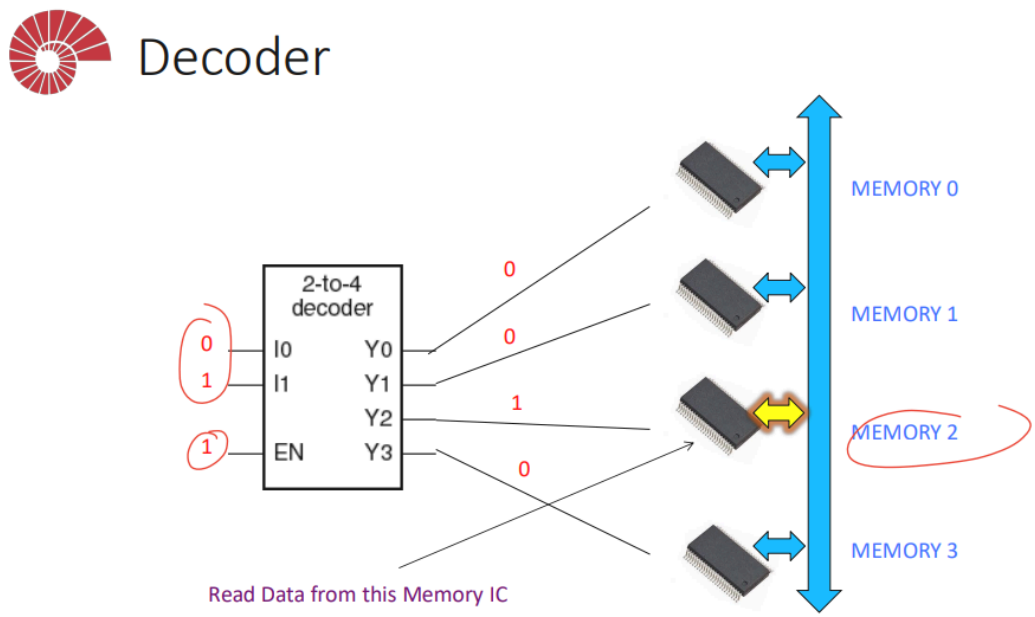
- **Further Understanding:**
  - Decoders translate binary addresses into specific enable signals that activate the correct memory module or peripheral.
  - Efficient addressing is critical for the performance of microprocessor-based systems.



2-to-4-Decoder Logic Diagram

> **2-to-4-Decoder Logic Diagram:** A schematic representation showing how two input bits generate four outputs using logic gates.

- **Further Understanding:**
  - The diagram visually explains the correlation between input combinations and the activation of one unique output.
  - It is a valuable tool for understanding the internal workings of a basic decoder.



Decoder Expansion

> **Decoder Expansion:** A technique for constructing larger decoders (e.g., a 3-to-8 decoder) by using multiple smaller decoders (e.g., 2-to-4 decoders) with enable inputs.

- **Further Understanding:**
  - This modular approach allows designers to scale decoder functionality without increasing complexity exponentially.
  - It is widely used in systems requiring a large number of unique output signals, such as memory addressing.

Decoder Applications

> **Decoder Applications:** Practical implementations of decoder circuits in digital systems.

- **Further Understanding:**
  - **Memory Systems:** Select different memory banks or rows.
  - **I/O Systems:** Enable specific devices within a microprocessor environment.
  - **Instruction Decoding:** Activate functional units based on the instruction code.
  - **Display Systems:** Drive segmented displays (e.g., seven-segment displays).

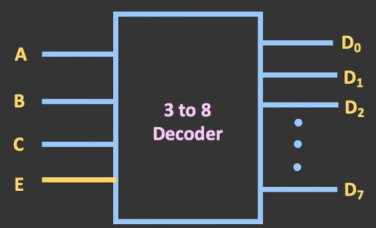
Combinational Functions and Circuits

> **Combinational Functions and Circuits:** Logic circuits where the outputs depend solely on the current inputs, with no memory elements.

- **Further Understanding:**
  - These circuits include decoders, encoders, multiplexers, and others that form the backbone of digital system design.
  - They are essential for performing arithmetic, data routing, and control logic, ensuring that digital systems operate efficiently and reliably.

3 to 8 Decoder

Decoder with Enable Input



Truth Table

A	B	C	E	D0	D1	D2	D3	D4	D5	D6	D7
X	X	X	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	1	0	1	0	0	0	0	0	0
0	1	0	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0	0
1	0	1	1	0	0	0	0	0	1	0	0
1	1	0	1	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

ALL ABOUT ELECTRONICS

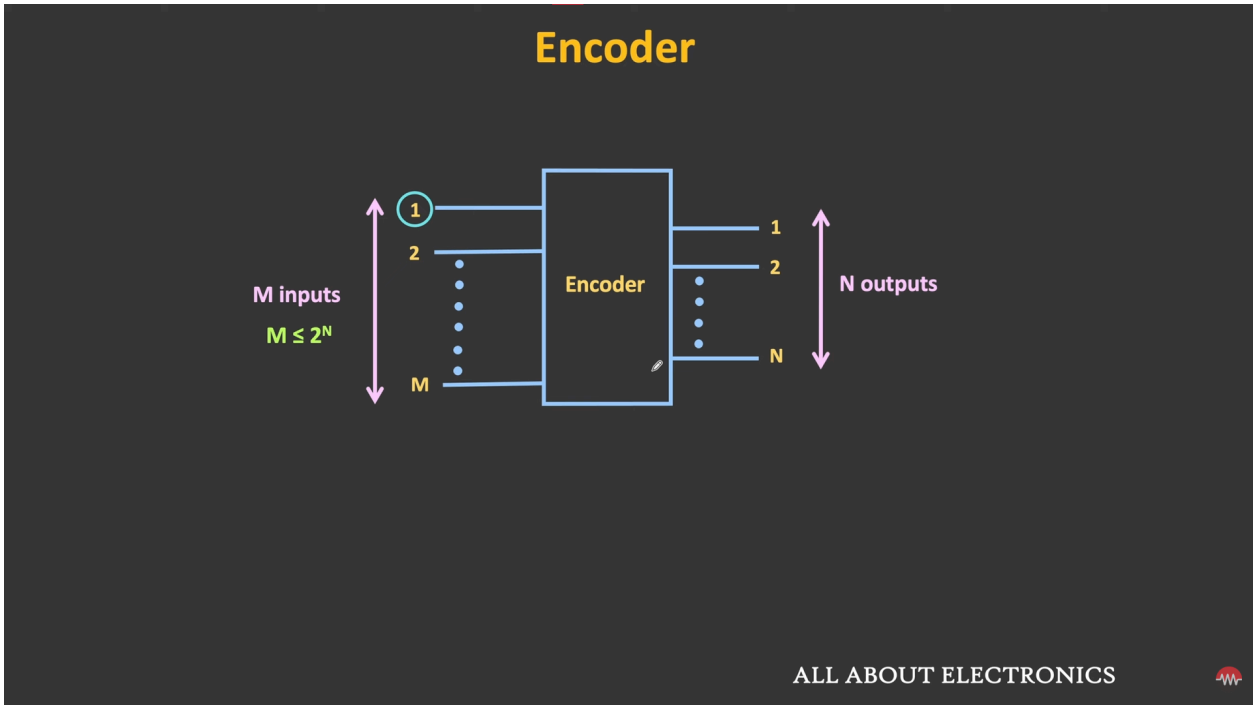
11. Encoders, Selecting Functions and Multiplexers

Encoders and Decoders

Encoders vs. Decoders

> **ENCODER:** A circuit that converts multiple input signals into a compact binary representation.
> **DECODER:** A circuit that converts coded inputs into a one-hot output, effectively reversing the encoding process.

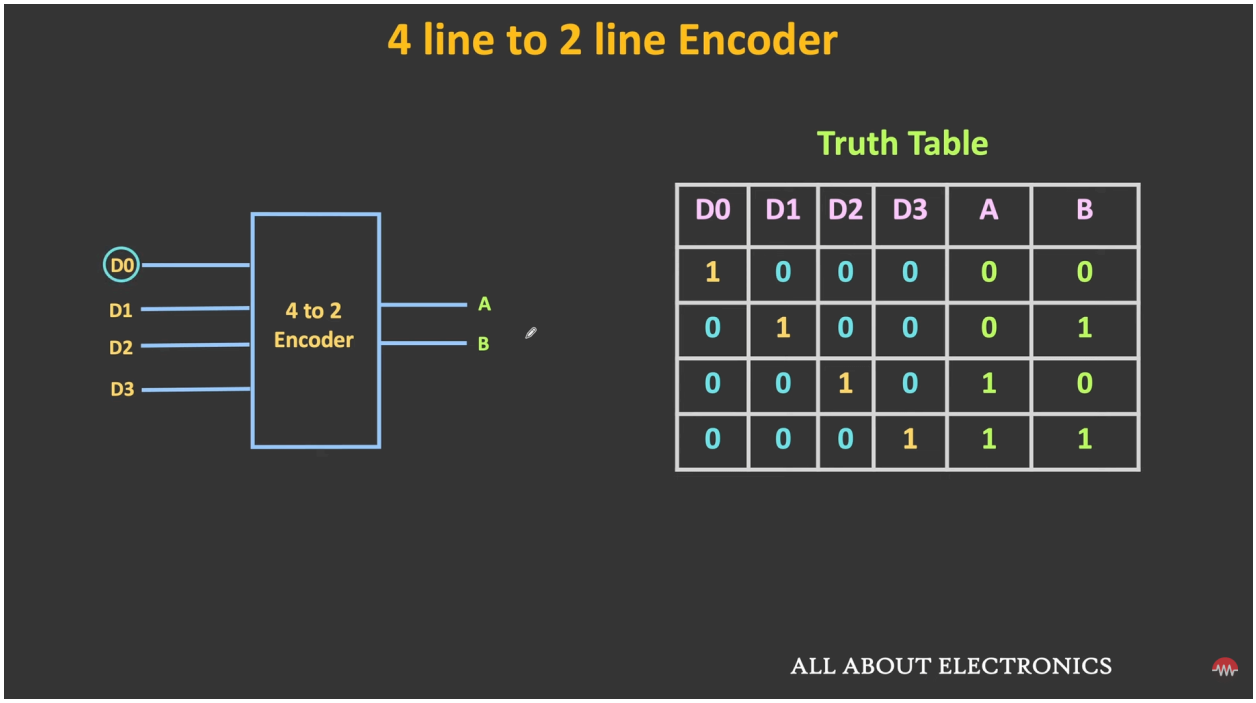
Encoders are used to reduce the number of signal lines by representing the active input with a binary code, while decoders expand a coded input into a set of distinct output lines.

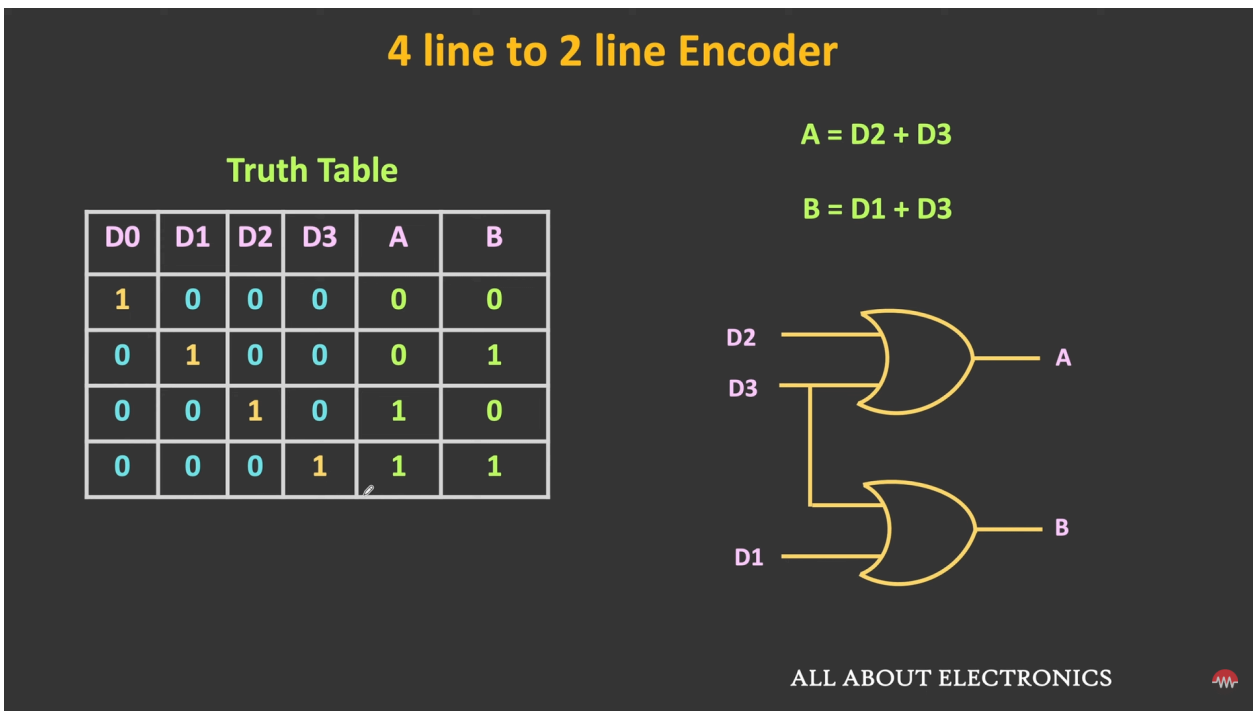


Binary Encoders

> **BINARY ENCODER:** A device that takes 2^n input lines and produces an n-bit binary output corresponding to the active input.

For example, a 4-to-2 binary encoder translates 4 input lines into a 2-bit output.





Priority Encoders

In real-world applications, more than one input may be active simultaneously. Priority encoders resolve such conflicts by assigning higher priority to inputs with higher indices.

> **PRIORITY ENCODER:** An encoder that outputs the binary code of the highest-priority active input.

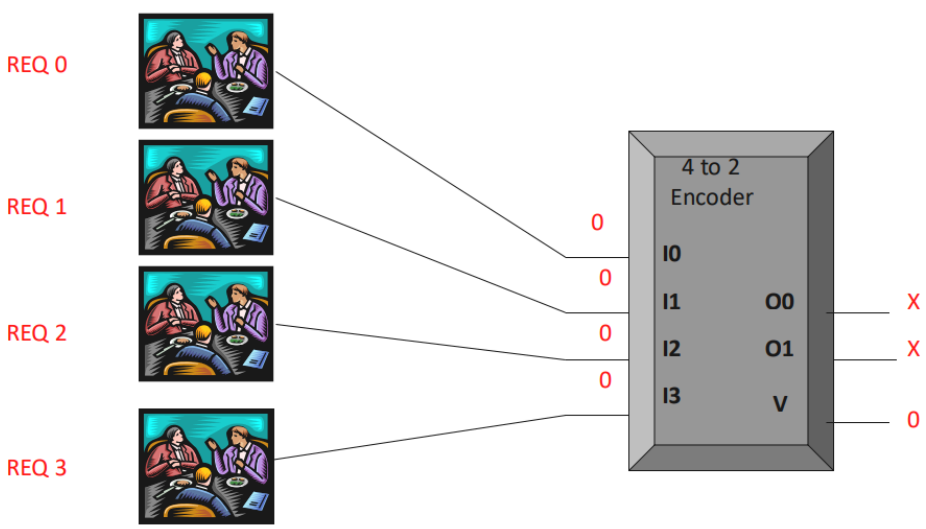
Priority Encoder

TABLE 3-6
Truth Table of Priority Encoder

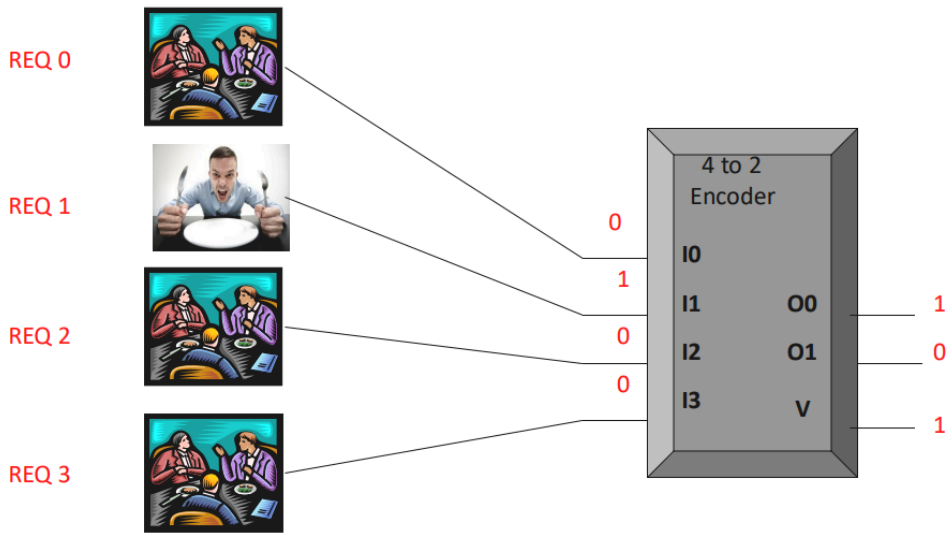
Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Table 3-6 Truth Table of Priority Encoder

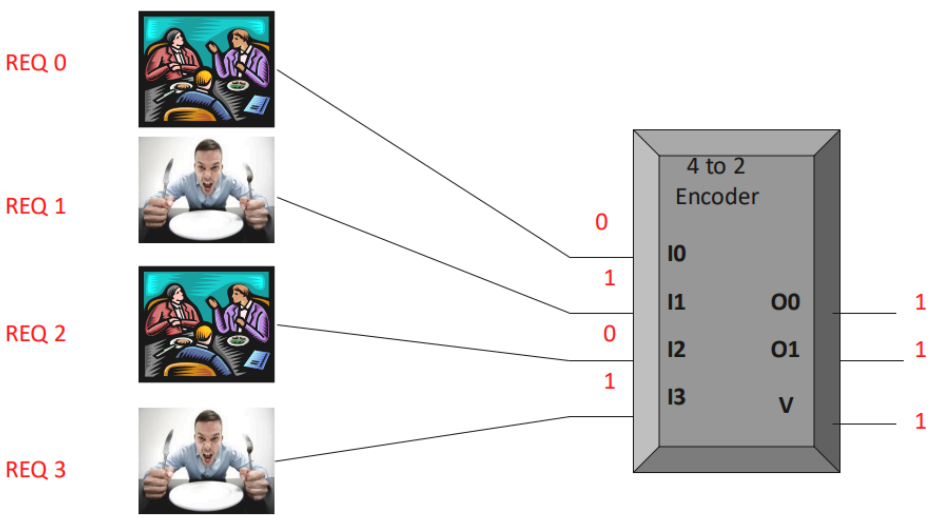
Encoder



Encoder



Priority Encoder



Selecting Functions and Multiplexers

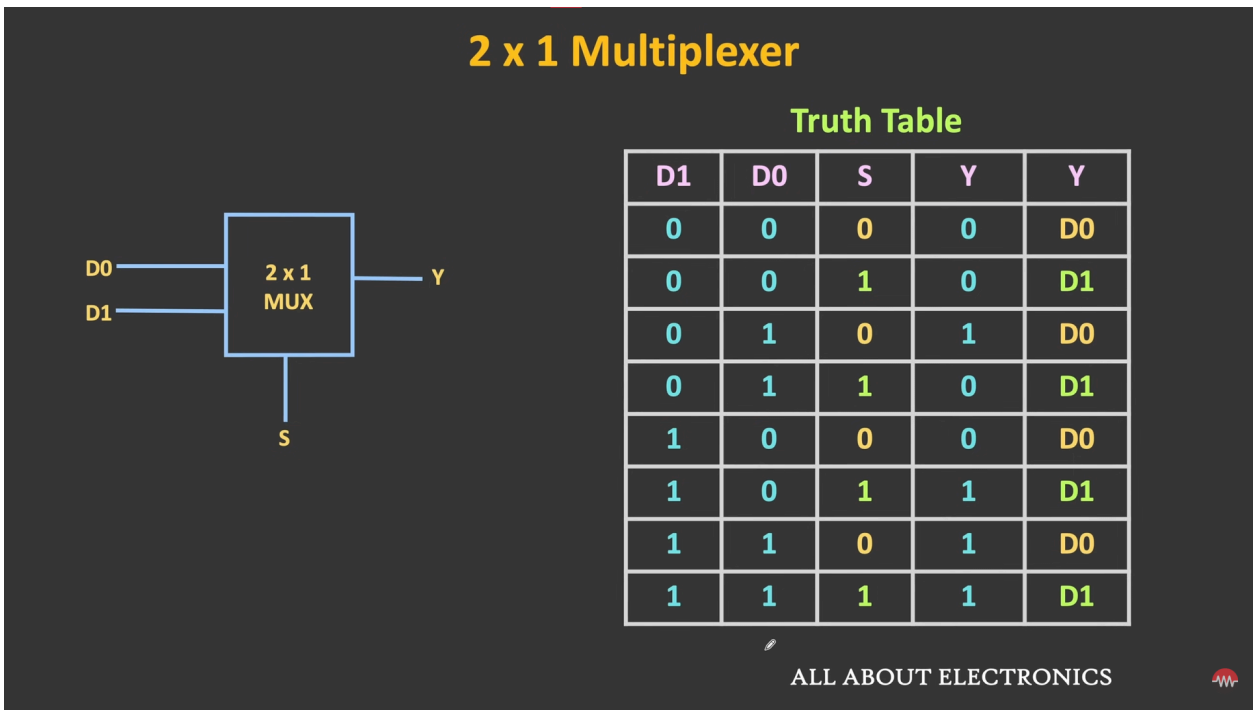
Multiplexers (MUX)

> **MULTIPLEXER:** A combinational circuit that selects one of many input lines and forwards the chosen input to a single output based on control signals.

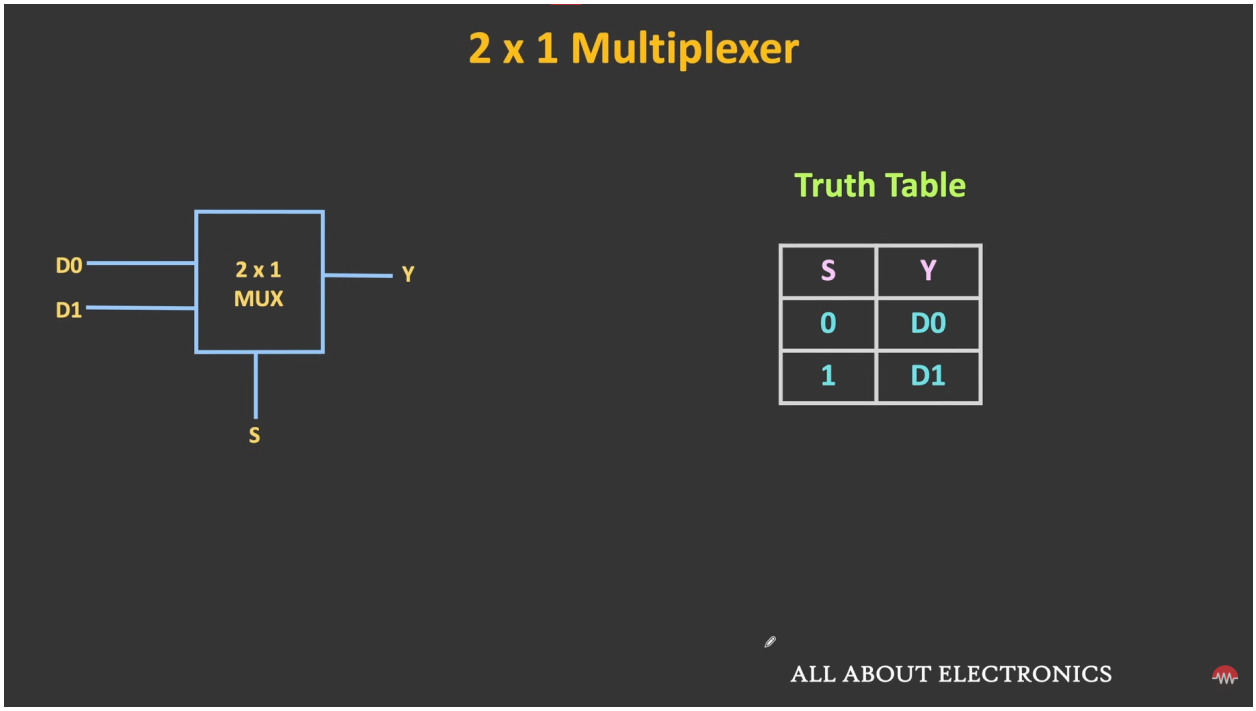
Key points:

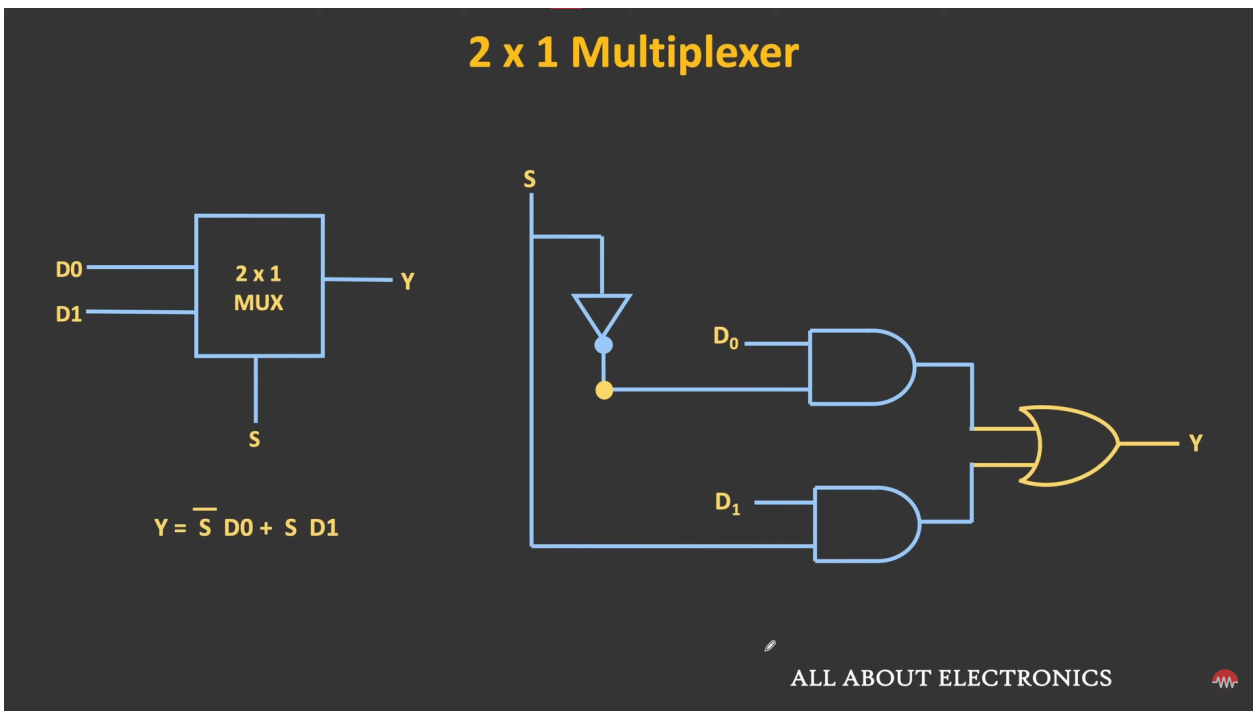
- **Control Lines:** An n-bit control input selects among 2^n possible inputs.
- **Example:** A 4-to-1 multiplexer uses 2 control bits to select one input from four.

Multiplexers are crucial for routing data in digital systems and can simplify the implementation of complex Boolean functions.



D1	D0	S	Y	Y
0	0	0	0	D0
0	0	1	0	D1
0	1	0	1	D0
0	1	1	0	D1
1	0	0	0	D0
1	0	1	1	D1
1	1	0	1	D0
1	1	1	1	D1





Combinational Circuit Implementation Using MUX

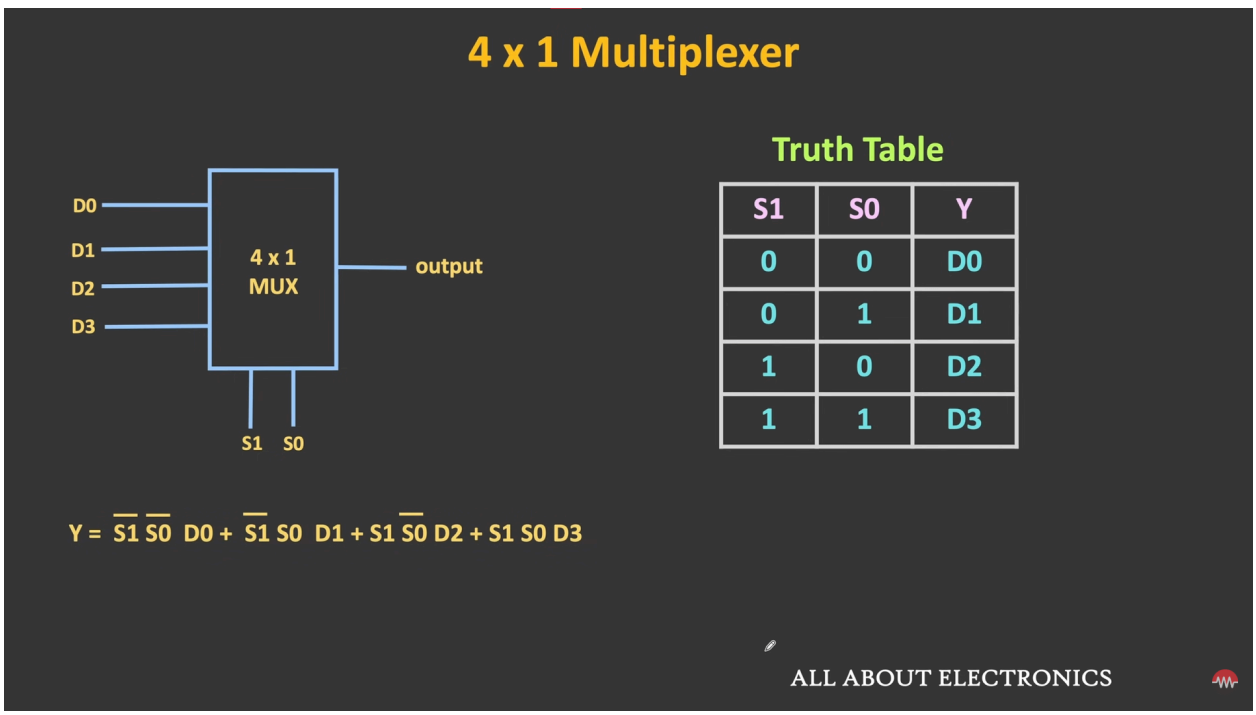
Multiplexers can be used to directly implement Boolean functions.

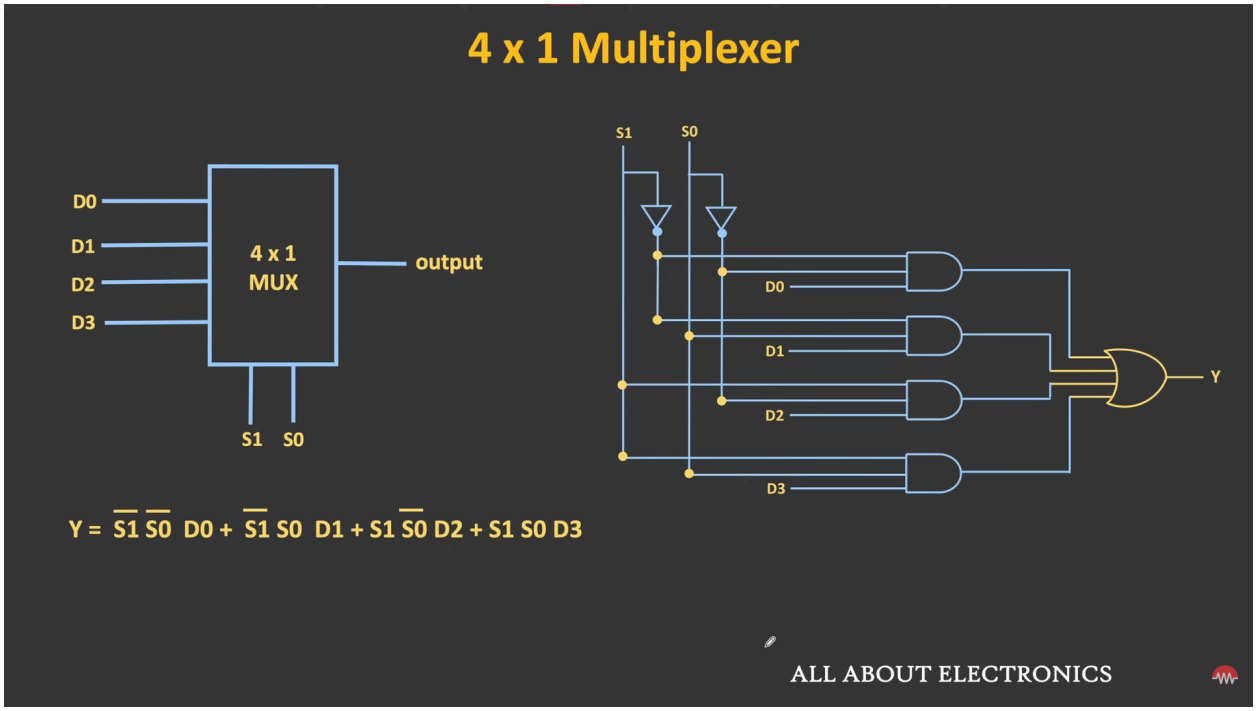
Example:

To implement the function

$$F(X, Y, Z) = \sum m(1, 2, 6, 7)$$

a 4-to-1 multiplexer is used where the control lines select the appropriate input corresponding to the minterms.

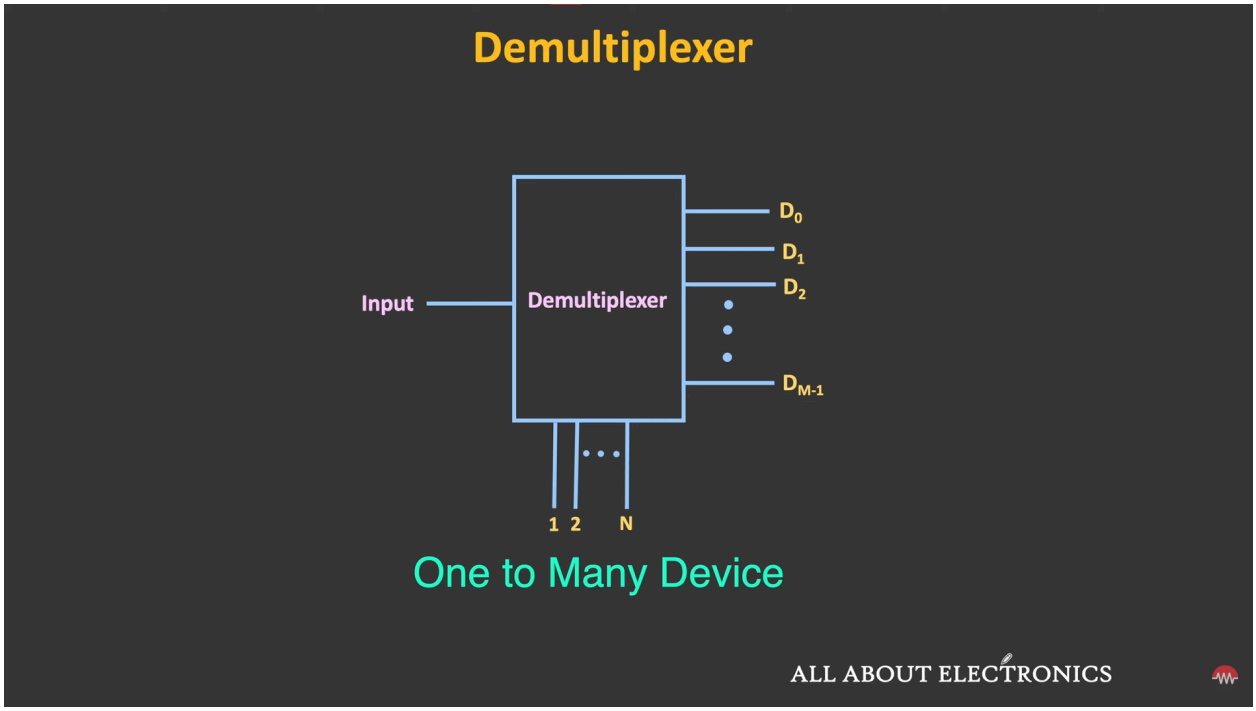


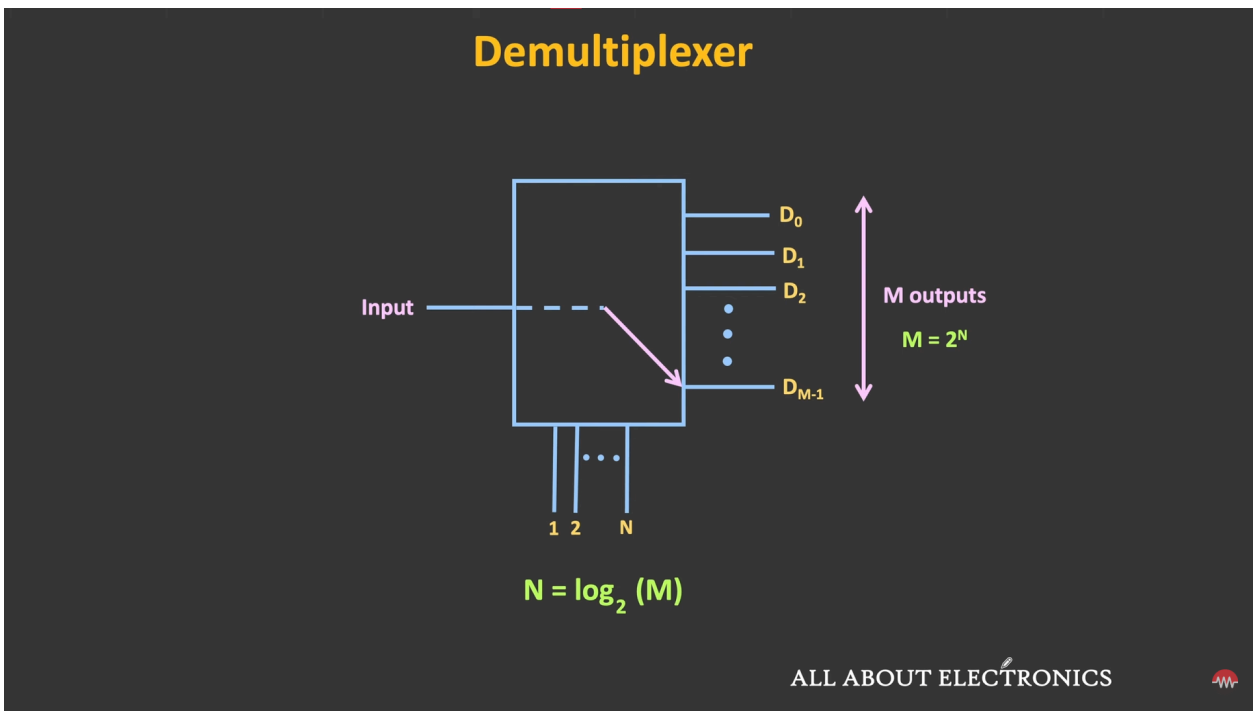


Demultiplexers

> **DEMULTIPLEXER**: The inverse of a multiplexer; it takes a single input and routes it to one of many outputs based on control signals.

Demultiplexers are used in applications such as memory addressing and data distribution, where a single source must be directed to one of several destinations.





Final Summary & Takeaways

- **Encoders** convert multiple signals into a compact binary form, whereas **decoders** expand coded inputs into distinct outputs.

- **Priority encoders** handle multiple active inputs by assigning precedence, with specific Boolean equations governing their behavior.
- **Multiplexers** and **demultiplexers** are essential for data selection and distribution, simplifying complex circuit implementations.
- Understanding these combinational circuits is critical as they form the foundation for more advanced sequential logic and digital system design.

12. Solutions for Midterm Sample Questions

Question 1: Number Representations and Arithmetic

Part (a): Converting Decimal Numbers to 6-Bit Representations

For a 6-bit system:

- **Sign-Magnitude and 1's Complement** have a range of –31 to +31 (5-bit magnitude).
- **2's Complement** has a range of –32 to +31.

1. Decimal **12**

- **Binary (Magnitude):**
  12 in binary (using 5 bits) is:

$$12 = 01100$$

- **Sign-Magnitude:**
  Since 12 is positive, the sign bit is 0.

$$\text{Sign-Magnitude: } 0\,01100 \quad \Rightarrow \quad 001100$$

- **1's Complement:**
  Positive numbers remain unchanged.

$$\text{1's Complement: } 001100$$

- **2's Complement:**
  For positive numbers, it is the same as the standard binary representation.

$$\text{2's Complement: } 001100$$

2. Decimal **19**

- **Magnitude in Binary:**
  19 in binary (5 bits):

$$19 = 10011$$

- **Sign-Magnitude:**
  For negative numbers, set the sign bit to 1.

$$\text{Sign-Magnitude: } 1\,10011 \quad \Rightarrow \quad 110011$$

- **1's Complement:**
  First, represent 19 in 6-bit positive form:

$$19 = 010011$$

  Then, flip all bits:

$$\text{1's Complement: } 010011 \rightarrow 101100$$

- **2's Complement:**
  Add 1 to the 1's complement:

$$\text{2's Complement: } 101100 + 1 = 101101$$

3. Decimal **32**

- **Sign-Magnitude and 1's Complement:**
  The maximum magnitude representable is 31, so -32 is **not representable** in these systems.

$$\text{Sign-Magnitude: NA}, \quad \text{1's Complement: NA}$$

- **2's Complement:**
  For 6 bits, -32 is representable.
  In 2's complement, the most negative number is represented by a 1 followed by all 0s:

$$\text{2's Complement: } 100000$$

Part (b): Arithmetic Operations in 8-Bit 2's Complement

Before performing arithmetic, we must sign-extend the given numbers to 8 bits.

Operation 1: $1101 + 010111$

- **First Operand:**

  1101 (4-bit) has MSB = 1, so it is negative.

  In 4-bit 2's complement, 1101 represents:

$$-(16 - 13) = -3.$$

  Sign-extend to 8 bits (replicate the sign bit):

$$1101 \rightarrow 11111101.$$

- **Second Operand:**

  010111 (6-bit) has MSB = 0 (positive).

  Sign-extend to 8 bits by adding two zeros:

$$010111 \rightarrow 00010111.$$

- **Addition in 8-Bit 2's Complement:**

$$\begin{array}{ll} 11111101 & (-3) \\ 00010111 & (23) \\ \hline 00010100 & (20) \end{array}$$

Result:

00010100 in 8-bit 2's complement, which equals decimal 20.

Operation 2: $0111 - 11101$

- **First Operand:**

  0111 (4-bit) is positive.

  Sign-extend to 8 bits:

  $$0111 \to 00000111 \quad (7).$$

- **Second Operand:**

  11101 (5-bit) has MSB = 1 (negative).

  Sign-extend to 8 bits (prepend three 1's):

  $$11101 \to 11111101 \quad (-3) \quad (\text{in 5-bit 2's complement}, 11101 = -3).$$

- **Subtraction:**

  $7 - (-3) = 7 + 3 = 10.$

  Represent 10 in 8-bit 2's complement:

  $$10 = 00001010.$$

  Result:

  00001010 in 8-bit 2's complement, which equals decimal 10.

Part (c): Left-Shift Multiplication and Overflow Detection

Concept:

Multiplying a number by 2 in 2's complement arithmetic is equivalent to shifting its binary representation one bit to the left. This shift inserts a 0 at the least significant bit

and discards the bit that overflows from the most significant position.

Procedure:

1. **Left Shift:**
   For an n-bit 2's complement number x, shifting left one bit gives a new binary number that represents $2x$ (provided there is no overflow).

2. **Example:**
   Consider the 8-bit number 00110110 (which is decimal 54).
   Left shift by one bit:

$$00110110 \rightarrow 01101100.$$

   01101100 represents decimal 108, which is exactly 54×2.

3. **Overflow Detection Rule:**
   Overflow occurs when the result of the shift cannot be represented within the fixed number of bits.

> **OVERFLOW RULE:** In 2's complement, overflow is detected if the sign bit (MSB) of the original number does not match the sign bit of the shifted result.
>
> For example, if a positive number (MSB = 0) becomes negative (MSB = 1) after shifting, then overflow has occurred.

Question 2: Canonical forms, K-map optimization, and hardware implementation using multiplexers

This section provides formal, step-by-step solutions for optimizing the Boolean function

$$F(A, B, C, D) = (\overline{A} + \overline{C})(\overline{A} + B + \overline{D})(A + C + \overline{D})$$

with the don't-care condition

$$d(A, B, C, D) = \sum m(9, 10).$$

The problem is divided into four parts (a)–(d).

(a) Product-of-Maxterms Representation

Objective: Find the canonical product-of-maxterms form (Π-form) of F.

Approach:

A Boolean function F in product-of-sums (POS) form is 0 if at least one of its factors is 0. We analyze each factor to determine the conditions for $F = 0$:

1. **Factor 1:** $\overline{A} + \overline{C}$

   This is 0 when both $\overline{A} = 0$ and $\overline{C} = 0$, i.e.

   $$A = 1 \quad \text{and} \quad C = 1.$$

   This condition covers all minterms with $A = 1$ and $C = 1$ (independent of B and D).

2. **Factor 2:** $\overline{A} + B + \overline{D}$

   This is 0 when

   $$\overline{A} = 0,\ B = 0,\ \overline{D} = 0 \quad \Rightarrow \quad A = 1,\ B = 0,\ D = 1.$$

   Minterms: $m(9)$ (for $C = 0$) and $m(11)$ (for $C = 1$).

   (Note that $m(11)$ is already included.)

3. **Factor 3:** $(A + C + \overline{D})$

   This is 0 when

   $$A = 0,\ C = 0,\ \overline{D} = 0 \quad \Rightarrow \quad A = 0,\ C = 0,\ D = 1.$$

   Minterms: $m(1)$ (for $B = 0$) and $m(5)$ (for $B = 1$).

Union of all conditions:

The zeros of F occur for minterms:

$$\{1,\ 5,\ 9,\ 10,\ 11,\ 14,\ 15\}.$$

Thus, the product-of-maxterms (canonical POS form) is:

$$F = \Pi M(1, 5, 9, 10, 11, 14, 15).$$

(b) Optimized SOP Expression via 4-Variable K-Map

Given:

$d(A, B, C, D) = \sum m(9, 10)$ (don't cares)

F is 0 for minterms $\{1, 5, 9, 10, 11, 14, 15\}$; hence, $F = 1$ for the remaining minterms:

$$\{0, 2, 3, 4, 6, 7, 8, 12, 13\}.$$

K-Map Setup:

We use the standard 4-variable K-map (rows: AB in Gray code order $00, 01, 11, 10$; columns: CD in order $00, 01, 11, 10$). The assignments are as follows:

- **Row 00 (A=0,B=0):**
  - $m(0) = 0000 : \ 1$
  - $m(1) = 0001 : \ 0$
  - $m(3) = 0011 : \ 1$
  - $m(2) = 0010 : \ 1$
- **Row 01 (A=0,B=1):**
  - $m(4) = 0100 : \ 1$
  - $m(5) = 0101 : \ 0$
  - $m(7) = 0111 : \ 1$
  - $m(6) = 0110 : \ 1$
- **Row 11 (A=1,B=1):**
  - $m(12) = 1100 : \ 1$
  - $m(13) = 1101 : \ 1$
  - $m(15) = 1111 : \ 0$
  - $m(14) = 1110 : \ 0$
- **Row 10 (A=1,B=0):**

- $m(8) = 1000 : 1$
- $m(9) = 1001 : X$ (don't care)
- $m(11) = 1011 : 0$
- $m(10) = 1010 : X$ (don't care)

Grouping for SOP:

1. **Group 1:**

   A 2×2 block covering cells in rows 00 and 01, columns 10 and 11 (minterms $m(2), m(3), m(6), m(7)$).

   Common: $A = 0$ (since rows 00 & 01) and $C = 1$ (columns 10 and 11).

   Prime Implicant: $A'C$.

2. **Group 2:**

   A vertical group in column 00 covering rows 00 and 01 (minterms $m(0)$ and $m(4)$).

   Common: $A = 0,\ C = 0,\ D = 0$ (B varies).

   Prime Implicant: $A'C'D$'.

3. **Group 3:**

   A horizontal pair in row 11, columns 00 and 01 (minterms $m(12)$ and $m(13)$.

   Common: $A = 1,\ B = 1,\ C = 0$.

   Prime Implicant: ABC'.

4. **Group 4:**

   A pair in row 10, columns 00 and (if we assign $m(9)$ as 1) column 01 (minterm $m(8)$ and don't care $m(9)$).

   Common: $A = 1,\ B = 0,\ C = 0$.

   Prime Implicant: $AB'C$'.

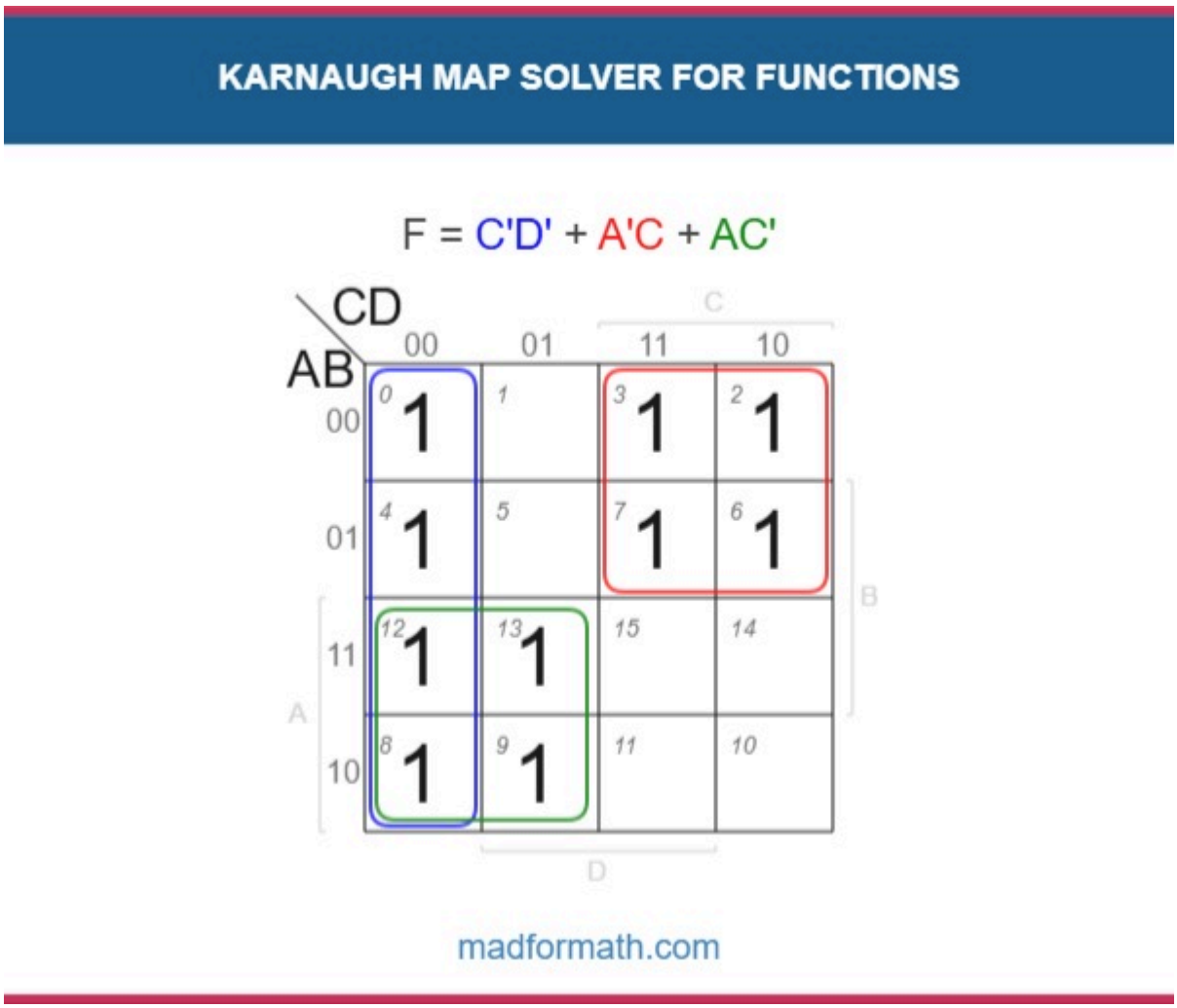
Combining Groups:

Notice that Groups 3 and 4 can be combined:

$$ABC' + AB'C' = AC'(B + B') = AC'.$$

Thus, the simplified SOP expression becomes:

$$F = A'C + A'C'D' + AC'.$$



Further Simplification:

Factor A' from the first two terms:

$$F = A'(C + C'D') + AC'.$$

Using the identity $C + C'D' = C + D$' (since if $C = 0$, then $C'D' = D$'; if $C = 1$, the sum is 1), we obtain:

$$F = A'(C + D') + AC'.$$

Gate Input Cost:

Counting literal occurrences:

- In $A'(C + D')$: literals A', C, D' → 3 inputs.
- In AC': literals A, C' → 2 inputs.
- Final OR (summing two product terms) → 2 inputs.

Thus, an estimated total cost is $3 + 2 + 2 = 7$ inputs.

(c) Optimized POS Expression via 4-Variable K-Map

Objective:

Obtain the minimal product-of-sums (POS) expression for F using the given don't cares.

Zeros of F:

From part (a), $F = 0$ for minterms: $\{1, 5, 9, 10, 11, 14, 15\}$.

Treat the don't cares $m(9)$ and $m(10)$ as 0 to facilitate grouping.

K-Map Grouping for Zeros:

1. **Group 1:**

   Minterms $m(1)$ (00,01) and $m(5)$ (01,01).

   Common values:

   - $A = 0$ (both rows have $A = 0$),
   - $C = 0$ (column 01 gives $C = 0$),
   - $D = 1$.

   Maxterm: For zeros, if a variable is 0 in all cells, include it in non-complemented form; if 1, include its complement.
   → A (since $A = 0$), C (since $C = 0$), and D' (since $D = 1$).
   Result: $(A + C + D')$.

2. **Group 2:**

   Common values:

- $A = 1$,
- $C = 1$,

Maxterm: A' (since $A = 1$), C' (since $C = 1$).
Result: $(A' + C')$.

3. **Group 3:**

   Common values:

   - $A = 1$,
   - $B = 0$,
   - $D = 1$.

     Maxterm: A' (since $A = 1$), B (since $B = 0$), D' (since $D = 1$).
     Result: $(A' + B + D')$.

Thus, the optimized POS expression is:

$$F = (A + C + D')(A' + C')(A' + B + D')$$

Gate Input Cost: 11.

(d) Implementation Using a 4-to-1 Multiplexer and a Single NOT Gate

Objective:

Implement F (with don't cares) using only a 4-to-1-line multiplexer and one NOT gate.

Strategy:

We start with the simplified SOP form from part (b):

$$F = A'(C + D') + AC'.$$

This expression depends on A and one of C or D but not on B. We can choose select lines such that the expression becomes independent of one variable.

Step 1: Choose Select Variables

Let's choose A and C as the select lines of the 4-to-1 multiplexer. Then the MUX will implement F as:

$$F = f_{AC}(D),$$

with A and C determining which data input is selected.

Step 2: Determine the Output for Each Combination of (A, C)

Using the simplified expression:

- **Case 1:** $A = 0$
  Then $F = (C + D')$.
  - When $C = 0$: $F = 0 + D' = D$'.
  - When $C = 1$: $F = 1 + D' = 1$.
- **Case 2:** $A = 1$
  Then $F = C$' (independent of D).
  - When $C = 0$: $C' = 1$.
  - When $C = 1$: $C' = 0$.

Step 3: Assign Multiplexer Data Inputs

The 4-to-1 MUX has 4 data inputs corresponding to select line combinations (A, C) as follows (using binary order 00, 01, 11, 10 where the order of bits is A (MSB) and C (LSB)):

- For $(A, C) = 00$: $A = 0,\ C = 0 \rightarrow F = D$' .
- For $(A, C) = 01$: $A = 0,\ C = 1 \rightarrow F = 1$.
- For $(A, C) = 11$: $A = 1,\ C = 1 \rightarrow F = 0$.
- For $(A, C) = 10$: $A = 1,\ C = 0 \rightarrow F = 1$.

Step 4: Hardware Implementation

- **Select Lines:** Connect A and C to the multiplexer select inputs.
- **Data Inputs:**
  - $I_0 = D$' (requires the single NOT gate to invert D).

    - $I_1 = 1$ (logic high).
    - $I_2 = 0$ (logic low).
    - $I_3 = 1$ (logic high).
- The output of the multiplexer will yield F.

This design meets the requirement: only one NOT gate is used (to generate D'), and the entire function F is implemented with a single 4-to-1 MUX.

Midterm Sample

midterm-sample.pdf

13. Sequential Logic and Memory

Introduction to Digital Systems and Sequential Circuits

Big Picture of Digital Systems

- **Digital Systems** are broadly classified into:
  - **Combinational Logic:** Systems without memory where outputs depend solely on current inputs.
  - **Sequential Logic:** Systems that incorporate memory elements, where outputs depend on both current inputs and past history.

> **DIGITAL SYSTEMS:** Systems that process binary signals; sequential systems specifically integrate storage elements (latches or flip-flops) with combinational logic to implement state-dependent operations.

Introduction to Sequential Circuits

- A sequential circuit is composed of:
  - **Storage Elements (Memory):** Such as latches or flip-flops to hold state.
  - **Combinational Logic:** Which computes next state and output based on current inputs and stored state.

> **SEQUENTIAL CIRCUITS:** Digital circuits where outputs are functions of both present inputs and the stored (past) state. This design enables complex operations such as counting, memory storage, and control flow.

Memory in Digital Systems

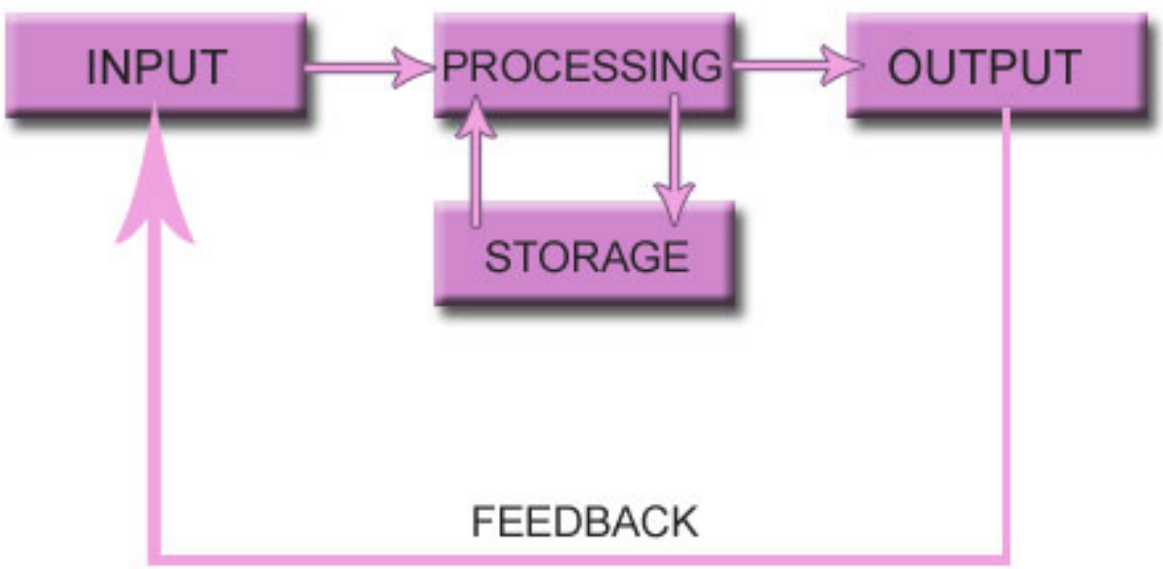
Role of Memory

- **Memory** is essential for maintaining state in a digital system.
- Inherent delays in gates can unintentionally store a value, but these effects are temporary.
- To store data indefinitely, feedback is introduced.

> **MEMORY:** The capability of a circuit to store information; it is achieved by feeding outputs back to inputs, thereby maintaining a stable state over time.

Extending Storage with Feedback

- **Feedback Mechanism:** Feeding the output back into the input loop allows a system to "hold" a value indefinitely.



SOURCE: WWW.TEACH-ICT.COM

Latches: The Basic Storage Element

Overview of Latches

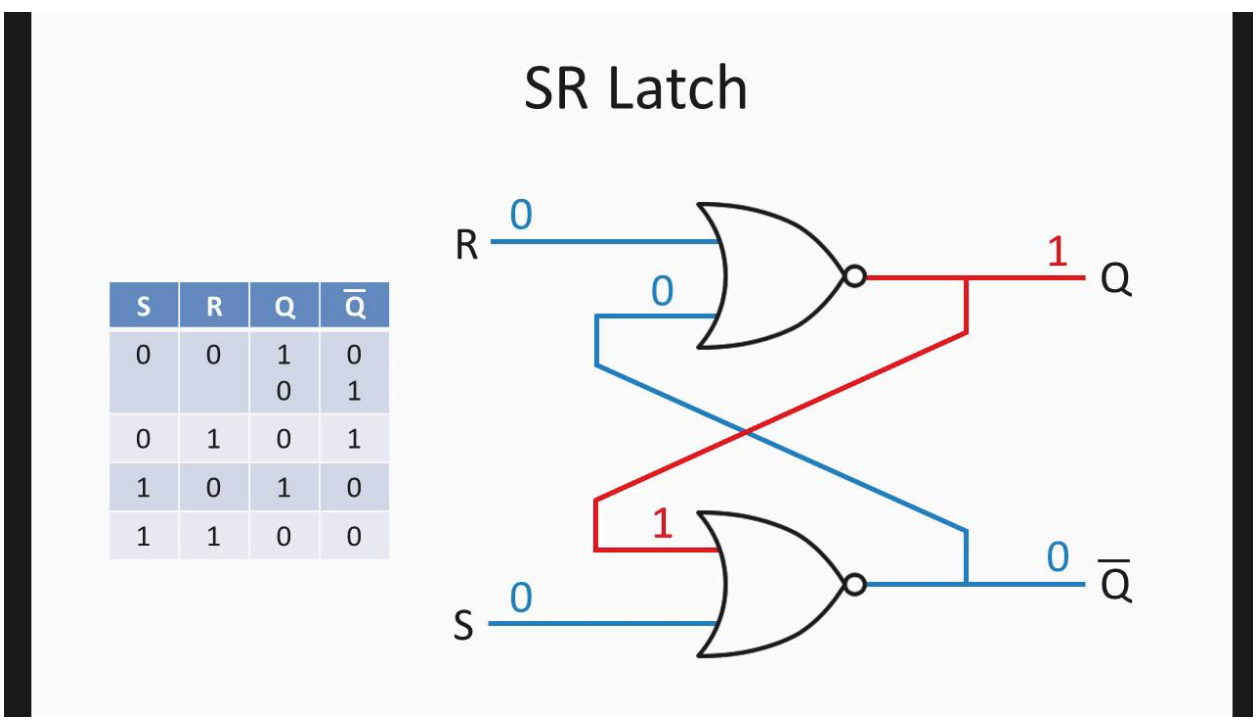
- **Latches** are the simplest storage elements in sequential circuits.
- They hold a binary state (0 or 1) until an external signal causes a change.

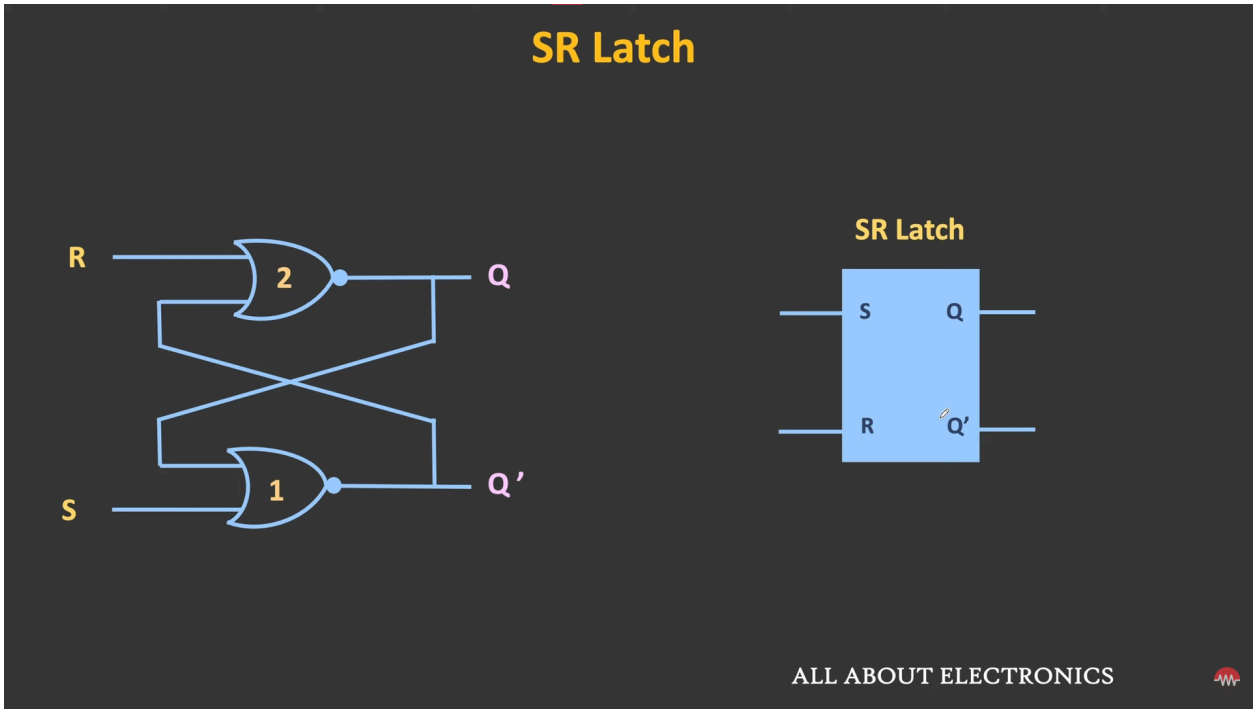
> **LATCH:** A bistable circuit that maintains its state until altered by an input signal, serving as a fundamental memory element in digital systems.

Basic (NOR) S-R Latch

- **Construction:** Uses NOR gates to implement the Set-Reset (S-R) functionality.
- **Inputs and Operations:**
  - **S = 1, R = 0:** Sets Q to 1.
  - **S = 0, R = 1:** Resets Q to 0.
  - **S = 0, R = 0:** Holds the current state.
  - **S = 1, R = 1:** Typically an undefined or forbidden state.

> **S-R LATCH (NOR):** A basic latch using NOR gates where S (set) and R (reset) determine the output Q.

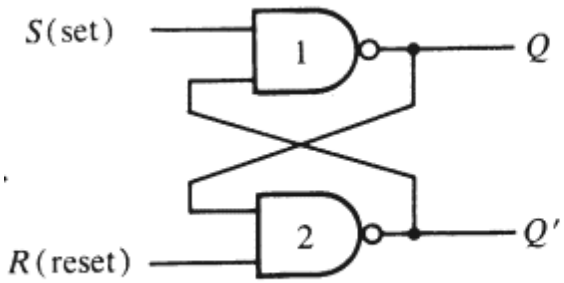




Basic (NAND) S'-R' Latch

- **Construction:** Formed by cross-coupling two NAND gates.
- **Inputs:** Active-low inputs (S' and R').
- **Operations:**
  - **S' = 0, R' = 1:** Sets Q to 1.
  - **S' = 1, R' = 0:** Resets Q to 0.
  - **S' = 1, R' = 1:** Maintains the current state.
  - **S' = 0, R' = 0:** Forbidden input condition.

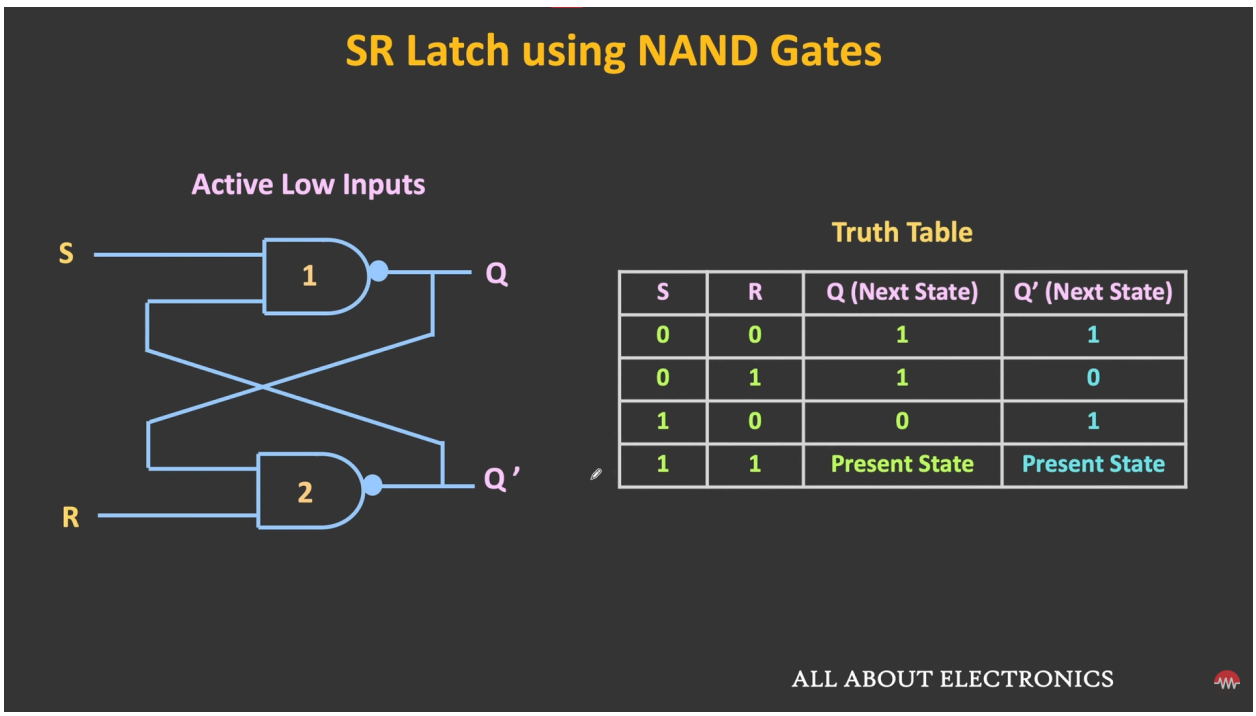
> **S'-R' LATCH (NAND):** A latch that uses NAND gates with active-low signals, ensuring stable operation by avoiding ambiguous states.



(a) Logic diagram

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	

(b) Truth table

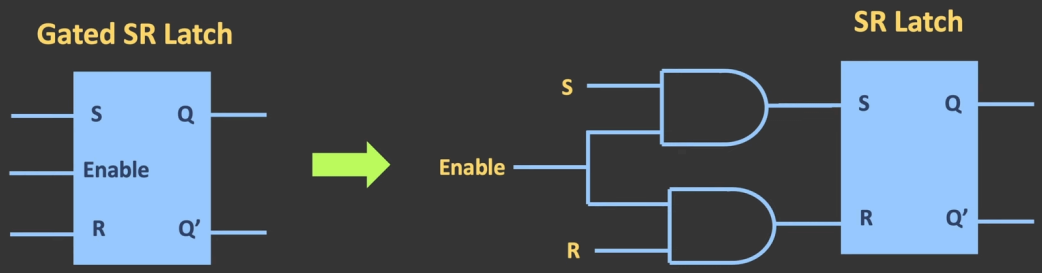


Clocked (Gated) S-R Latch

- **Enhancement:** Incorporates a clock signal (C) to control when S and R are observed.
- **Operation:** The latch updates its state only when the clock is high.

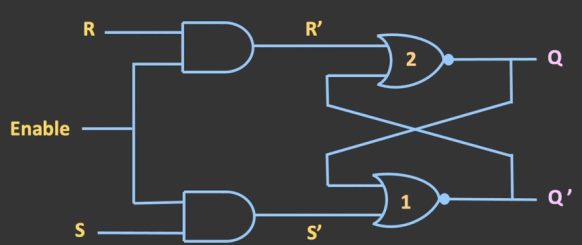
> **CLOCKED S-R LATCH:** A variant of the S-R latch where a clock input enables state changes, providing synchronization in sequential circuits.

Gated SR Latch



ALL ABOUT ELECTRONICS

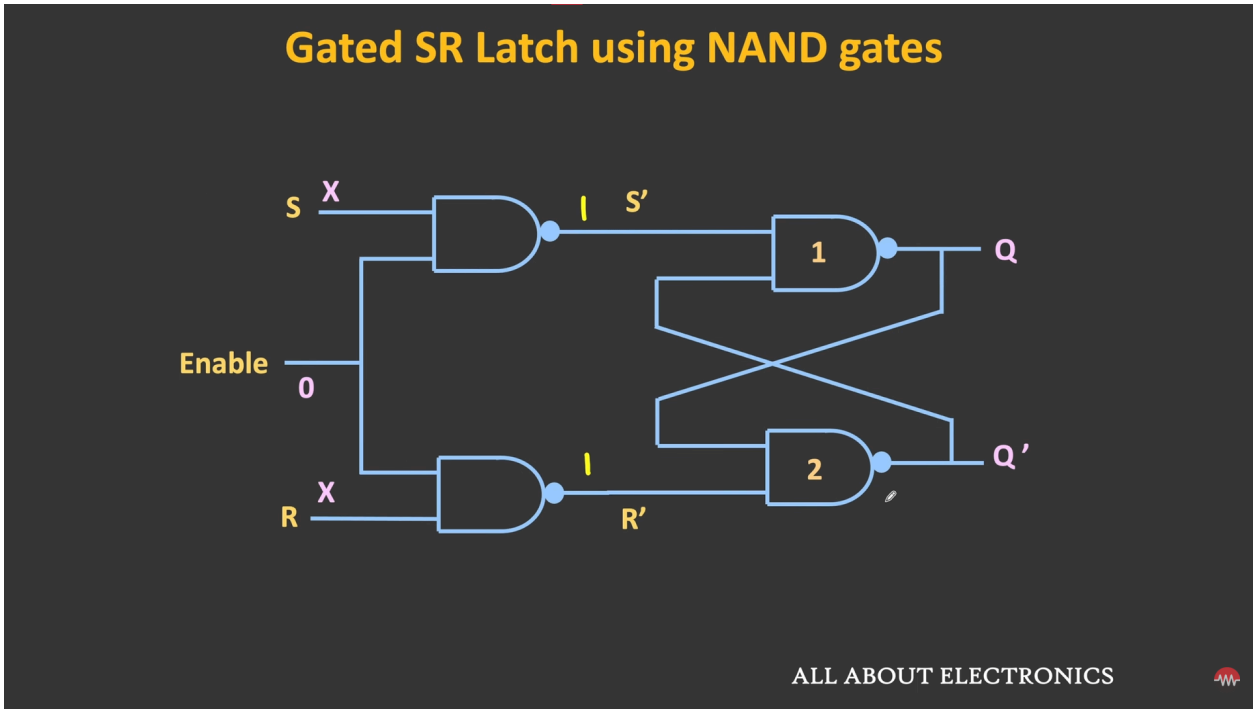
Gated SR Latch

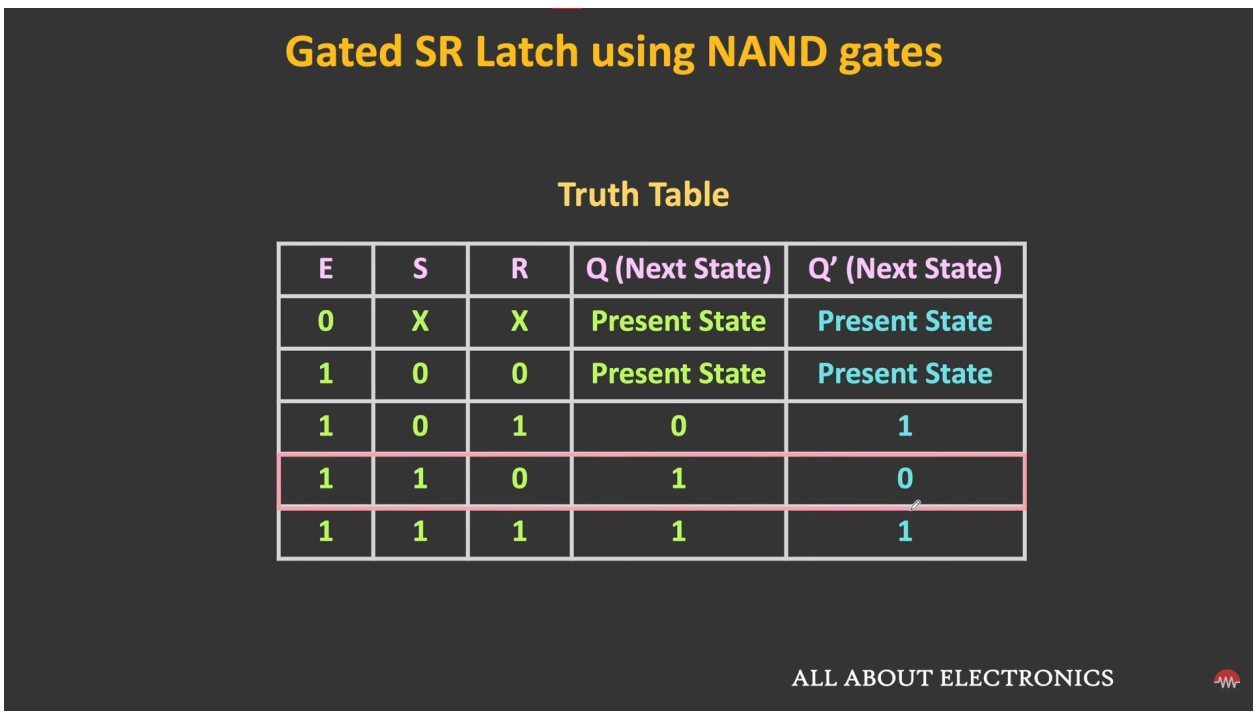


Truth Table

E	S	R	Q (Next State)	Q'(Next State)
0	X	X	Present State	Present State
1	0	0	Present State	Present State
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

ALL ABOUT ELECTRONICS





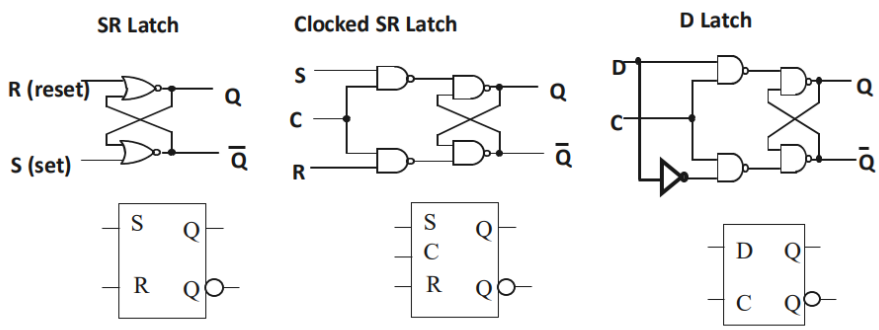
D Latch

- **Derivation:** Obtained by adding an inverter to the S-R latch to eliminate indeterminate states.

- **Operation:**
  - **Input D:** Directly drives the output Q when the clock is active.
  - **Characteristic:** No ambiguous state exists, making the latch predictable.

> **D LATCH**: A latch that captures the value of the input D under a clock condition, ensuring unambiguous data storage.

Latches



S	R	Q+	Inst
1	0	1	Set
0	1	0	Reset
0	0	Q	Hold
1	1		UD

C	S	R	Q+	Inst
1	1	0	1	Set
1	0	1	0	Reset
1	0	0	Q	Hold
1	1	1		UD
0	x	x	Q	Hold

C	D	Q+	Inst
0	x	Q	Hold
1	0	0	Reset
1	1	1	Set

13

Final Summary & Takeaways

- **Sequential Circuits** integrate memory elements with combinational logic to manage state and process sequences of inputs.
- **Latches** are fundamental storage devices, with various implementations (NOR-based, NAND-based, clocked, and D latches) ensuring reliable memory.
- **Feedback** plays a critical role in extending storage duration, making it possible to hold a value indefinitely.

14. Flip-Flops & Sequential Circuit Analysis

Flip-Flops

The Latch Timing Problem

> **LATCH TIMING PROBLEM:** In a clocked D-latch with feedback, as long as the clock input C = 1, changes in output Q immediately feed back to the input D, causing uncontrolled oscillation or multiple updates within one clock pulse.

Flip-Flop Timing (1's-Catching)

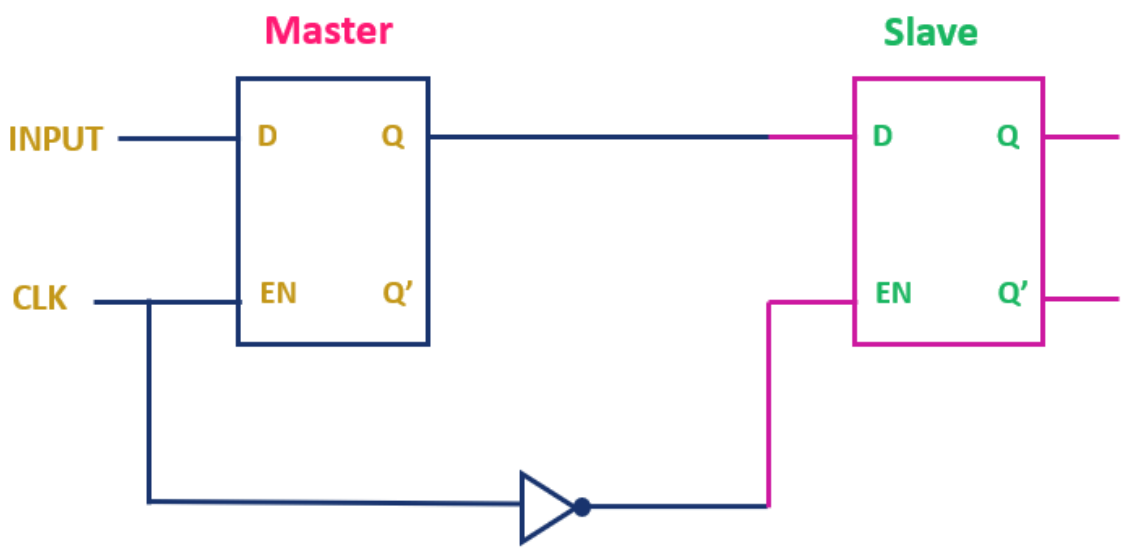
> **1's-CATCHING PROBLEM:** If the data input (S or D) changes while the latch is transparent (C = 1), the master latch may set or reset multiple times during the same pulse, capturing unintended "1's."
>
> - **Example:** With Q = 0, a brief high on D while C = 1 can propagate to Q, then back to D, toggling multiple times.
> - **Consequence:** Results in extra transitions, metastability risk, and dependence on pulse-width.

- **Solution:** Use **edge-triggered architecture** so inputs are ignored when C is steady; only the clock edge causes a single update.

Master-Slave Flip-Flop

MASTER-SLAVE FF: Two clocked latches in series, with the slave latch driven by the inverted clock.

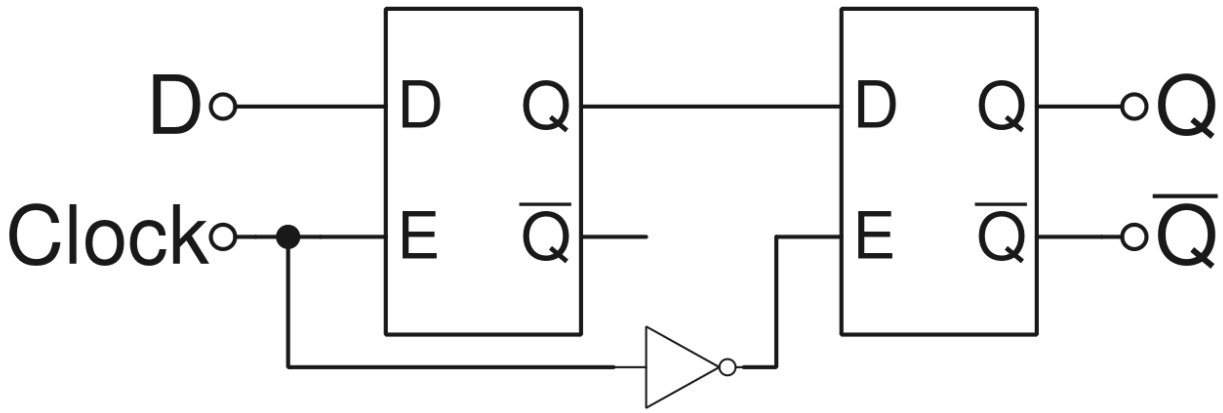
- **Master (C = 1):** Captures input.
- **Slave (C = 0):** Transfers master's output to Q.
- **Benefit:** Master and slave are never transparent at the same time, preventing intra-pulse feedback.

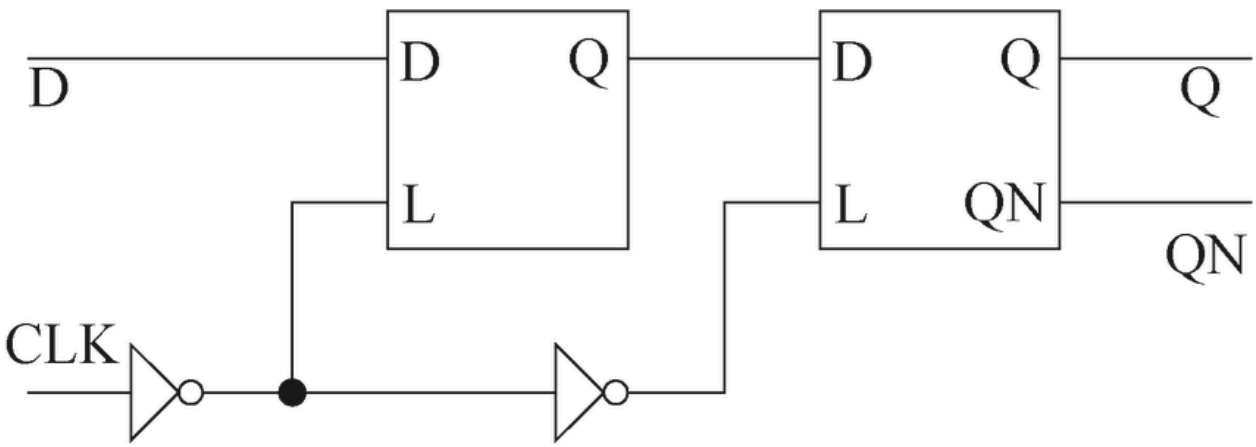


Edge-Triggered D Flip-Flop

EDGE-TRIGGERED FF: Updates output only on a clock edge, ignoring input changes at other times.

- **Negative-edge triggered:** Updates on falling clock.
- **Positive-edge triggered:** Updates on rising clock (via an inverter on C).
- **Remark:** Simplifies input to a single D line, avoiding forbidden states and reducing delay.

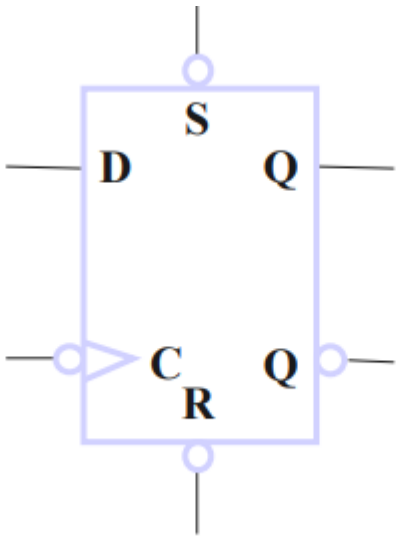




Standard Symbols & Direct Inputs

STANDARD STORAGE SYMBOLS:

- **D-Flip-Flop:** Single data input, clock, Q/$\bar{Q}$ outputs.
- **S-R Flip-Flop:** Separate S and R inputs (forbidden high-high combination).
- **DIRECT INPUTS:** Asynchronous Set (S_a) and Reset (R_a) for initialization outside clocked operation.



Sequential Circuit Analysis

General Model

> **SEQUENTIAL CIRCUIT MODEL:**
>
> - **State Variables:** Stored in flip-flops (vector **S**(t)).
> - **Next State:** **S**(t+1) = f(**S**(t), X(t)).
> - **Outputs:** Y(t) = g(**S**(t), X(t)).

Analysis Example

Given: Input x(t), state bits A(t), B(t), output y(t).

$$A(t+1) = A(t)\,x(t) \;+\; B(t)\,x(t)$$
$$B(t+1) = A'(t)\,x(t)$$
$$y(t) = x'(t)\,\big(B(t) + A(t)\big)$$

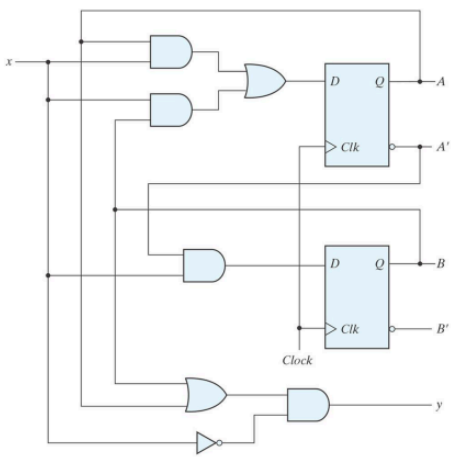
> **Solution Steps:**
>
> 1. Derive next-state and output Boolean expressions.
> 2. Construct the state-table with Present State, Input, Next State, Output.
> 3. Simplify expressions if desired.

State-Table & State-Diagram

> **STATE TABLE:** Lists all Present State & Input combinations with corresponding Next State & Output.
>
> - **Columns:** Present State | Input | Next State | Output
> - **STATE DIAGRAM:** Directed graph where nodes are states; arcs labeled "input/output" show transitions.

SC Analysis Example

- Boolean equations for the functions:
  - A(t+1) = A(t)x(t) + B(t)x(t)
  - B(t+1) = A'(t)x(t)
  - y(t) = x'(t)(B(t) + A(t))



SC Analysis Example: State Table

- The state table can be filled in using the next state and output equations:

$A(t+1) = A(t)\,x(t) + B(t)\,x(t)$

$B(t+1) = A'(t)\,x(t)$

$y(t) = x'(t)\,(B(t) + A(t))$

Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

SC Analysis Example: Alternate State Table

$A(t+1) = A(t)\,x(t) + B(t)\,x(t)$

$B(t+1) = A'(t)\,x(t)$

$y(t) = x'(t)\,(B(t) + A(t))$

Present State		Next State				Output	
		x = 0		x = 1		x = 0	x = 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

State-Table Characteristics

STATE-TABLE CHARACTERISTICS:

- Divided into four sections: Present State, Input, Next State, Output.

> - Treats Present State & Input as “inputs” to the truth table; Next State & Output as “outputs.”
> - Enables systematic enumeration of all behavior for design and verification.

Final Summary & Takeaways

- **Latch vs. Flip-Flop:** Edge-triggered FFs eliminate the 1’s-catching issue inherent in level-sensitive latches.
- **Master-Slave vs. Edge-Triggered:** Master-slave uses back-to-back latches; edge-trigger updates only on a clock transition.
- **Analysis Workflow:**
  1. Write state equations.
  2. Build the state table.
  3. Draw the state diagram.
- **Key Pitfalls:**
  - Omitting asynchronous resets leads to undefined start states.
  - Mislabeling clock edges causes timing failures.
  - Failing to enumerate all table entries can hide invalid or unused states.

15. Finite State Machines: State Diagrams, Models, and Representations

1. State Diagrams

1.1 Definition & Components

> **STATE DIAGRAM:** Graphical FSM representation where:
>
> - Each **state** is a circle labeled with a state name.
> - **Directed arcs** show transitions from Present State → Next State.
> - **Labels on arcs** indicate the **input** causing that transition.
> - **Output labels** appear either:
>   - **On each circle** (Moore): output depends only on state.
>   - **On each arc** (Mealy): output depends on state + input.

1.2 Small vs. Large Circuits

- **Small:** State diagrams are intuitive and easier to follow than tables.

- **Large:** Diagrams become cluttered; tables or code may be preferable.

2. Moore & Mealy Models

2.1 Moore Model

> **MOORE FSM:** Outputs are a function only of the current state.
>
> - Output label placed inside each state circle.

2.2 Mealy Model

> **MEALY FSM:** Outputs are a function of state & input.
>
> - Output label placed on each transition arc ("input/output").

2.3 Mixed Models

> In practice, designs sometimes mix Moore and Mealy conventions.

3. Example Diagrams & Tables

3.1 Example State Diagrams

- Top: Mealy diagram with arcs labeled "x=1 / y=1", "x=0 / y=0", etc.
- Bottom: Moore diagram with circles labeled "A/0", "B/1", etc.

3.2 Example State Tables

Present	Next State (x=0)	Next State (x=1)	Output (x=0)	Output (x=1)
0	0	1	0	0
1	0	2	0	1

Present	Next State (x=0)	Next State (x=1)	Output
A/0	B	A	0
B/1	B	C	1

Final Summary & Takeaways

- **State diagrams** provide intuitive FSM visualization; use tables when diagrams grow complex.
- **Moore vs. Mealy:** Choose based on timing and output-dependence requirements.
- **Design workflow:** Follow the six-step procedure to go from spec → implementation → verification.

16. FSM Design & Sequence Detection

Sequential-Circuit General Model

> **MODEL:**
>
> - **State vector** S(t) stored in flip-flops.
> - **Next state** S(t+1) = f(S(t), X(t)).
> - **Outputs** Y(t) = g(S(t), X(t)?)

Six-Step Design Procedure

1. **Formulation:** Draw state diagram or table from specification.
2. **State Assignment:** Map each abstract state → binary code.
3. **Obtain State Table:** List Present State, Input → Next State, Output.
4. **Obtain Equations:**
    - (a) Flip-flop input equations from Next State entries.
    - (b) Output equations from Output entries.

- (c) Optimize via Karnaugh maps or Boolean simplification.

5. **Technology Mapping:** Implement equations with gates & actual FFs.
6. **Verification:** Confirm FSM behavior matches original spec.

Case Study: Sequence Recognizer (1101)

Problem Statement

> **TASK:** Output 1 whenever the input sequence "1101" appears (including overlaps).

Mealy Implementation

State Diagram

- States A→B→C→D track partial matches.
- Output 1 on transition that completes "1101".

State Table

Present	x=0 → (Next, y)	x=1 → (Next, y)
A	(A, 0)	(B, 0)
B	(A, 0)	(C, 0)
C	(D, 0)	(C, 0)
D	(A, 0)	(B, 1)

Moore Implementation

Extended State Diagram

- Add extra state E to produce output 1 in Moore style.
- Outputs on circles: A/0, B/0, C/0, D/0, E/1.

Moore State Table

State	Code	x=0 → Next	x=1 → Next	Output
A	000	A	B	0

State	Code	x=0 → Next	x=1 → Next	Output
B	001	A	C	0
C	011	D	C	0
D	010	A	E	0
E	110	A	C	1

Unused-State Handling

> **MINIMAL RISK:** Redirect illegal/unused codes → safe (reset) state.
>
> **MINIMAL COST:** Treat unused entries as “don’t cares” to simplify logic, but risk undefined behavior if ever entered.

Summary

- **Sequence recognizer:** Illustrates both Mealy (compact, output on arc) and Moore (requires extra state) designs.
- **State assignment:** Heuristic choices can greatly affect hardware cost and reliability.
- **Unused states:** Must be handled deliberately to avoid metastability or safe-mode failures.

17. State Assignment & Minimization

Obtain State & Output Equations

1. State/Output Table

Present State	Code ($Q_1Q_2Q_3$)	Next State (x=0)	Next State (x=1)	Output (y)
A	000	A (000)	B (001)	0
B	001	A (000)	C (011)	0
C	011	D (010)	C (011)	0
D	010	A (000)	E (110)	0
E	110	A (000)	C (011)	1

> **Remark:** From this table, derive for each flip-flop input the Boolean equation in terms of Q_1, Q_2, Q_3, and x, and the output $y = g(Q_1,Q_2,Q_3,x)$.

Decomposed State Assignment

> **DECOMPOSED STATE ASSIGNMENT:**

> - **Minimal Risk:** Unused (illegal) codes → safe/reset state on occurrence.
> - **Minimal Cost:** Mark unused codes as “don’t care” in next-state logic to simplify equations (assumes they never occur).

State Minimization

1. Purpose

> **STATE MINIMIZATION:** Reduce gates or flip-flops by merging equivalent states.

2. Equivalent States

> **EQUIVALENT STATES:** Two states are equivalent if, for every input sequence, they produce identical outputs and transition to equivalent next states.

3. Strategy

- Partition states by output behavior under x=0/1.
- Iteratively refine partitions until no further splits occur.
- Merge equivalent states in the state table/diagram.

State-Minimization Example

1. Original Table

Present	x=0 → Next, y	x=1 → Next, y
A	A,0	B,0
B	C,0	D,0
C	A,0	D,0
D	E,0	F,1
E	A,0	F,1
F	G,0	F,1
G	A,0	F,1

2. Identify Equivalences

- **E ≡ G** (same outputs/transitions)
- **F ≡ D**

3. Reduced Table

Present	x=0 → Next, y	x=1 → Next, y
A	A,0	B,0
B	C,0	D,0
C	A,0	D,0
D/F	D,0	D,1
E/G	A,0	D,1

Post-Minimization Steps

1. **Draw Reduced State Diagram** with merged states.
2. **Assign Codes** to reduced states (e.g., binary or one-hot).
3. **Derive New Equations** for D-inputs and y from the reduced table.
4. **Optimize Logic** using Karnaugh maps or Boolean simplification.

Final Summary & Takeaways

- **State/Output Table → Equations:** Systematic derivation for each flip-flop and output.
- **Unused-State Handling:** Choose minimal risk (redirect) or minimal cost (don't cares).
- **Minimization Workflow:**
    1. Partition by output behavior.
    2. Refine by next-state equivalence.
    3. Merge equivalent states → reduced FSM.
    4. Redesign logic for improved area/performance.
- **Common Pitfalls:**

- Neglecting unused codes can cause undefined behavior.
- Failing to fully refine partitions may miss further merges.
- Reassigning codes without considering transition adjacency can increase logic complexity.

18. Sequential-Circuit Fundamentals & Flip-Flops

1. Sequential-Circuit Model

> **SEQUENTIAL CIRCUIT MODEL:**
>
> - **State vector** S(t) stored in an array of flip-flops.
> - **Next state** S(t+1) = f(S(t), X(t)), a Boolean function of current state and inputs.
> - **Outputs** Y(t) = g(S(t), X(t)), a Boolean function of state (and sometimes inputs).

2. Latches & Flip-Flop Architectures

2.1 Latch Timing Problem

> **LATCH-TIMING PROBLEM:** In a level-sensitive D-latch, when clock C = 1 the feedback path Q→D allows multiple toggles within one pulse, causing oscillation.

2.2 Master-Slave Flip-Flop

> **MASTER-SLAVE FF:** Two back-to-back latches; master enabled when C = 1, slave when C = 0.

> - Breaks feedback loop—Q updates once per cycle.

2.3 Edge-Triggered Flip-Flop

> **EDGE-TRIGGERED FF:** Updates output only on a clock transition (rising or falling), ignoring input while clock is steady.

3. JK Flip-Flop

> **BEHAVIOR:** Like S-R FF but allows J=K=1, which toggles Q.

Characteristic Table

J	K	Q(t)	Q(t+1)	Operation
0	0	0	0	Hold
0	0	1	1	Hold
0	1	0	0	Reset
0	1	1	0	Reset
1	0	0	1	Set
1	0	1	1	Set
1	1	0	1	Toggle
1	1	1	0	Toggle

> **Characteristic Equation:**
>
> $$Q(t+1) = J\,\overline{Q(t)} \;+\; \overline{K}\,Q(t)$$

Excitation Table

Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

J-K Flip-Flop Descriptors

- Characteristic Table

J	K	Q(t+1)	Operation
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q}(t)$	Complement

- Characteristic Equation

$$Q(t+1) = J\overline{Q} + \overline{K}Q$$

- Excitation Table

Q(t)	Q(t+1)	J	K	Operation
0	0	0	X	No change
0	1	1	X	Set
1	0	X	1	Reset
1	1	X	0	No Change

4. T Flip-Flop

> **BEHAVIOR:** Single input T toggles Q when T = 1; holds when T = 0.

Characteristic Table

T	Q(t)	Q(t+1)	Operation
0	0	0	Hold
0	1	1	Hold
1	0	1	Toggle
1	1	0	Toggle

> **Characteristic Equation:**
>
> $$Q(t+1) = T \oplus Q(t)$$

Excitation Table

Q(t)	Q(t+1)	T
0	0	0

Q(t)	Q(t+1)	T
0	1	1
1	0	1
1	1	0

Final Summary & Takeaways

- **FSM Model:** Defined by next-state function f and output function g stored in flip-flops.
- **Latch vs. FF:** Edge-triggered FFs solve level-sensitive feedback issues.
- **JK & T FFs:** Characterize with tables and equations; excitation tables guide input design.

19. Registers & Bus-Based Transfer Structures

1. Registers & Design Models

1.1 Register Definition

> **REGISTER:** A collection of binary storage elements (flip-flops) that holds an n-bit vector. Can be defined by a state table, but typically treated as a storage vector for data movement and simple processing.

2. Register-Storage & Load Control

2.1 Expectations vs. Reality

> **EXPECTATIONS:** Register holds data across clock cycles; loading must be controlled.
>
> **REALITY:** A plain D-FF register loads on every clock edge.

2.2 Load Control Techniques

1. **Clock Gating**

> **CLOCK GATING:** AND the global clock with a Load signal to enable loading only when Load=1.
>
> **Problem:** Gated clocks introduce skew and timing hazards.

2. **Load-Controlled Feedback**

> **FEEDBACK CONTROL:** Keep clock free; use a 2:1 multiplexer on each D-input to select between current Q (hold) or new input (load).
>
> **Benefit:** Eliminates clock skew; cost is extra MUX logic.

3. Register-Transfer Operations & Notation

3.1 Register-Transfer Concept

> **REGISTER TRANSFER OPERATION:** Movement or processing of data between registers under control signals; each elementary step is a microoperation.

3.2 Notation

- **Register name:** R_n, PC, IR
- **Bit range:** R(7:0), PC(H), PC(L)
- **Transfer arrow:** $R_1 \leftarrow R_2$
- **Parallel ops:** $R_1 \leftarrow R_2,\ R_3 \leftarrow R_4$
- **Memory reference:** $R_0 \leftarrow M[AR]$

4. Conditional Transfers

> **CONDITIONAL TRANSFER:** K_1: ($R_2 \leftarrow R_1$) executes only if control $K_1 = 1$.
>
> Example timing: only one clock edge with K_1=1 causes the move.

5. Microoperations

5.1 Categories

> **Types:**

> - **Transfer**: move data between registers
> - **Arithmetic**: add, subtract, increment, etc.
> - **Logical**: bitwise OR, AND, EXOR, NOT
> - **Shift**: logical/arithmetic left/right

5.2 Arithmetic Microoperations

Symbolic	Description
$R_0 \leftarrow R_1 + R_2$	Addition
$R_0 \leftarrow R_1'$	One's complement
$R_0 \leftarrow R_1' + 1$	Two's complement
$R_0 \leftarrow R_2 - R_1$	Subtraction
$R_0 \leftarrow R_0 + 1$	Increment
$R_0 \leftarrow R_0 - 1$	Decrement

5.3 Logical Microoperations

Symbolic	Description
$R_0 \leftarrow R_1'$	Bitwise NOT
$R_0 \leftarrow R_1 \vee R_2$	Bitwise OR
$R_0 \leftarrow R_1 \wedge R_2$	Bitwise AND
$R_0 \leftarrow R_1 \oplus R_2$	Bitwise EXOR

> **Example:** For R_1=1010 1010 and R_2=1111 0000, compute all four operations.

5.4 Shift Microoperations

Symbolic	Description
$R_0 \leftarrow SL\ R_1$	Shift left (zero-fill)
$R_0 \leftarrow SR\ R_1$	Shift right (zero-fill)

6. Transfer-Structure Architectures

6.1 Multiplexer-Based Transfers

> **MUX-BASED:** Each register input has a dedicated MUX to select its source.

6.2 Bus-Based Transfers

> **BUS-BASED:** A shared bus driven by a single MUX or by multiple 3-state drivers; feeds many registers.

6.3 Dedicated vs. Shared Structures

- **Dedicated MUX:** Highly flexible, high hardware cost.
- **Shared Bus + MUX:** Lower cost, limited simultaneous transfers.
- **Shared Bus + 3-State Buffers:** Further cost reduction, drivers must tri-state when not active.

7. Three-State Buffers & Hi-Z Logic

7.1 Hi-Impedance Outputs

> **Hi-Z:** "Open-circuit" state that allows multiple drivers to share a bus safely. When output = Hi-Z, it neither drives 0 nor 1.

7.2 3-State Buffer

> **Behavior:**
>
> - **EN = 0:** OUT = Hi-Z
> - **EN = 1:** OUT = IN
> - Variations: bubbles invert IN or EN.

EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

7.3 Bus Contention & Resolution

> **Rule:** On a shared wire, at most one 3-state driver may be enabled; all others must be Hi-Z.
>
> **Valid combinations for two drivers B_1, B_0:**
>
> - (0, Hi-Z) → 0
> - (1, Hi-Z) → 1
> - (Hi-Z, 0) → 0
> - (Hi-Z, 1) → 1
> - (Hi-Z, Hi-Z) → Hi-Z

Final Summary & Takeaways

- Registers are *n* bit state vectors realized by D-FF arrays.
- Load control achieved via clock gating or input MUXes.
- Register-transfer notation formalizes data movement and processing.
- Microoperations provide elementary arithmetic, logic, and shift functions.
- Transfer structures trade off flexibility vs. hardware cost: dedicated MUX, shared bus, 3-state.
- Hi-Z logic enables bus sharing; only one driver active at a time is safe.

20. Registers and Register Transfer Operations

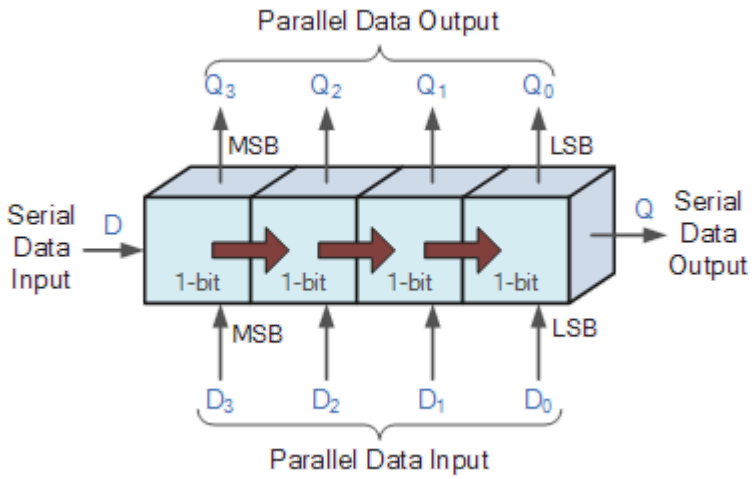
Register Fundamentals

Definitions

> **REGISTER:** A collection of binary storage elements storing a vector of bits.
>
> **MICROOPERATION:** An elementary operation (e.g., load, shift) performed on register contents.

Four-Bit Register Examples

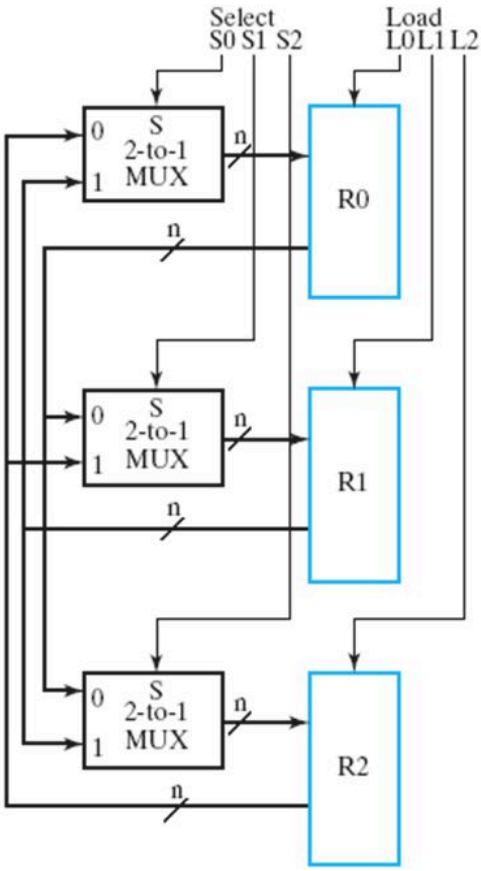
- **Parallel Load Register:** All bits load simultaneously from inputs.
- **Shift Register:** Bits shift left or right by one position per clock.



Register Transfer Architectures

Dedicated MUX-Based Transfer

> Each register input has its own multiplexer, allowing arbitrary source-to-destination moves in one cycle.



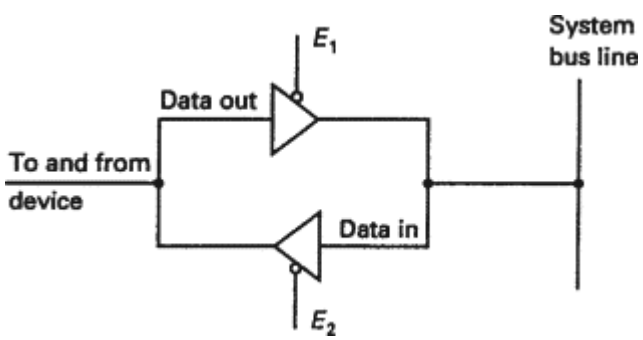
(a) Dedicated multiplexers

Dedicated MUX – based Transfer

- Multiplexer connected to each register input produces a very flexible structure
- Characterize the simultaneous transfers possible with this structure

Bus-Based Transfers

> **Multiplexer Bus:** Single bus with a large multiplexer at the driver side, limiting simultaneous transfers.
>
> **Three-State Bus:** Uses tri-state buffers on each register output sharing a common bus.



Final Summary & Takeaways

- Registers store and process data via microoperations.
- Transfer architectures trade flexibility (dedicated MUX) for cost (bus-based).
- Shift registers support serial data movement with minimal hardware.

21. Counters, Shift Registers, and Serial Transfer

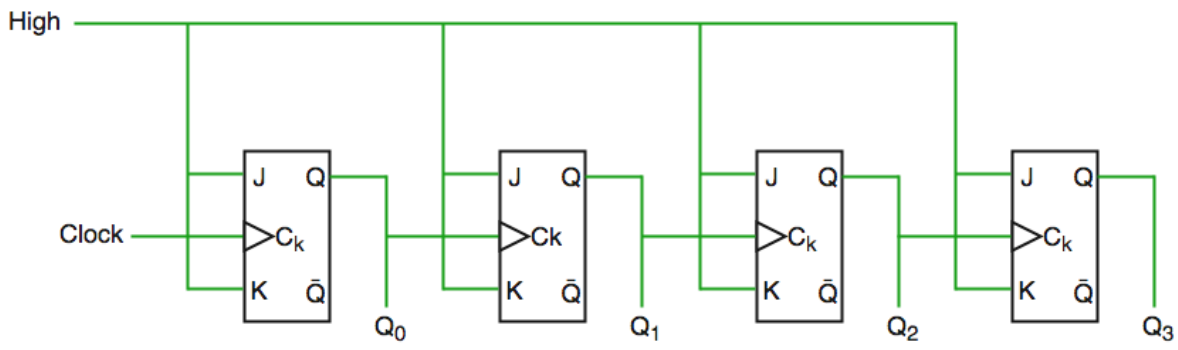
Counters

Definitions

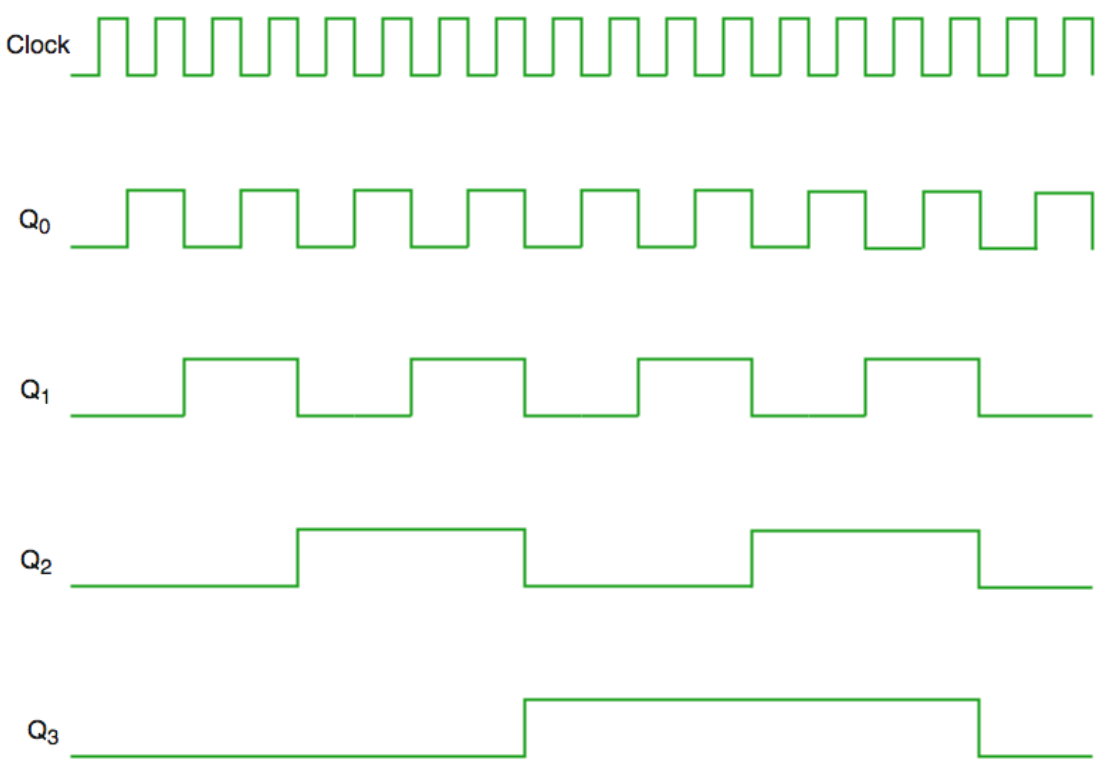
> **COUNTER:** A register sequence that advances through a predefined state sequence on clock pulses.

Ripple vs. Synchronous Counters

- **Ripple Counter:** Clock feeds only LSB; toggling ripples through bits (not fully synchronous).
- **Synchronous Counter:** Common clock to all flip-flops; combinational logic determines next state.



(a) Asynchronous counter



(b) Timing Diagram

Counter Variants

- **Up/Down Counter:** Counts up or down based on control input.
- **Modulo-N Counter:** Resets or loads on terminal counts to achieve non-power-of-two sequences.
- **Parallel-Load Counter:** Supports loading arbitrary values via parallel inputs.

Shift Registers and Serial Transfer

Shift Register Operations

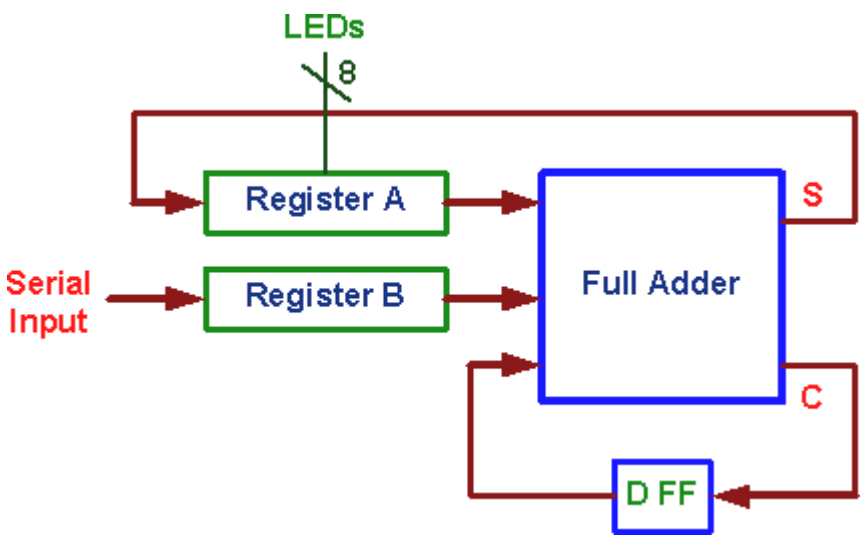
> **Shift Left/Right (sl/sr):** Moves bits toward MSB/LSB; new bit may be zero or provided externally.

Parallel-Load and Hold

> Adding multiplexers enables select between shift, load, and hold operations in the same register structure.

Serial Transfer Example

> **Serial Addition:** Performs A ← A + B one bit at a time using two shift registers and a single adder.



Final Summary & Takeaways

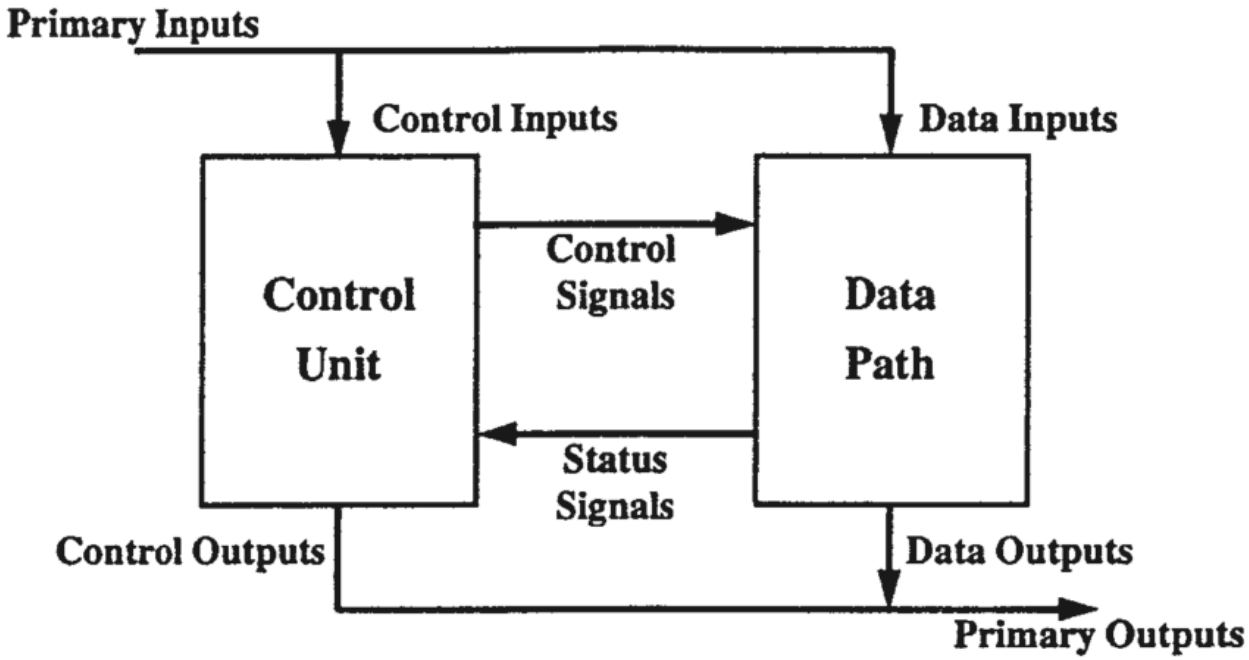
- Counters derive from register structures with combinational next-state logic.
- Synchronous design improves timing predictability over ripple counters.
- Shift registers and serial operations trade throughput for hardware efficiency.
- Parallel-load capability enhances flexibility for control applications.

22. Programmable Computer and Control Unit

Datapath & Control

> **DATAPATH:** Performs data transfer and processing operations, composed of a register file, function unit (ALU, shifter), and buses.
>
> **CONTROL UNIT:** Determines operation sequencing by generating control signals from external inputs and status flags.

- **Control Inputs:** External control signals and status outputs from the datapath.
- **Control Outputs:** Signals enabling multiplexers, register loads, and function-unit operations.



Control Unit Types

> **PROGRAMMABLE CONTROL UNIT:** Features a program counter (PC), instruction memory (ROM/RAM), and decision logic to fetch and decode microinstructions.
>
> **NON-PROGRAMMABLE CONTROL UNIT:** Hardwired sequencer without instruction fetch; suited to fixed-operation datapaths.

Datapath Components

> **REGISTER FILE:** A bank of registers with multiplexed inputs for microoperations.
>
> **FUNCTION UNIT:** ALU and shifter, with a function-select multiplexer (FS codes).
>
> **BUSES:** Shared A, B, and D buses using either dedicated multiplexers or three-state buffers.

Final Summary & Takeaways

- **Datapath** comprises the register file, function unit (ALU/shifter), and shared buses to move and process data.
- **Control Unit** generates sequencing signals from external inputs and status flags to orchestrate microoperations.

- **Control Inputs/Outputs:** Inputs include external controls and datapath status; outputs drive multiplexers, register loads, and function-unit operations.
- **Programmable vs. Non-Programmable:**
  - **Programmable Control Unit:** Uses a PC and microinstruction memory for flexible sequencing.
  - **Non-Programmable Control Unit:** Hardwired logic for fixed-operation control with lower hardware overhead.
- **Datapath Components Recap:**
  - **Register File:** Multiplexed inputs for microoperations.
  - **Function Unit:** ALU/shifter with function-select codes.
  - **Buses:** Implemented via dedicated multiplexers or tri-state buffers to route data.

23. Algorithmic State Machines and ASM Design

Algorithmic State Machines

> **ASM:** A structured, flowchart-like method to specify states, decisions, and microoperations in sequential circuits.

ASM Primitives

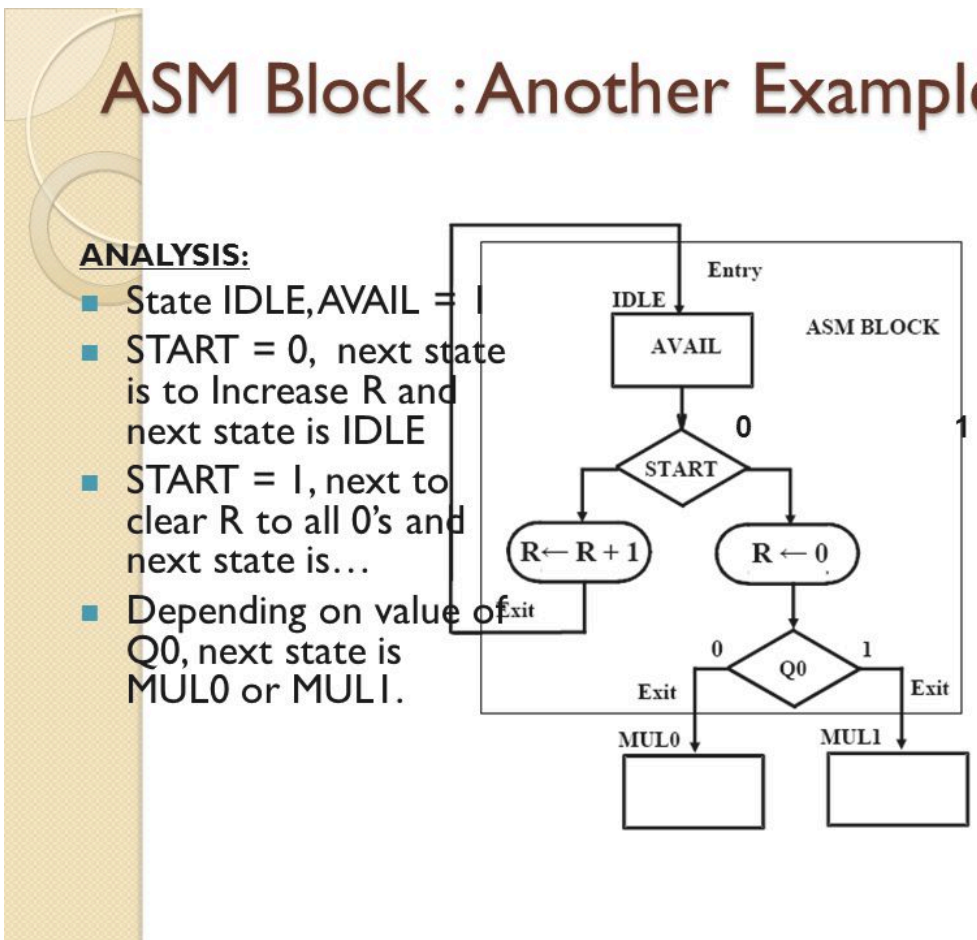
1. **State Box (Rectangle):** Denotes the current state and its register-transfer operations.
2. **Scalar Decision Box (Diamond):** Branches on a single input condition (TRUE/FALSE).
3. **Vector Decision Box (Hexagon):** Branches on an n-bit input vector, with up to 2^n exit paths.
4. **Conditional Output Box (Oval):** Specifies outputs or actions triggered under decision conditions.

ASM Blocks & Timing

> **ASM BLOCK:** A state box plus its connected decision and output boxes, representing one control cycle.
>
> **ASM TIMING:** Outputs are asserted during the state; register transfers occur on the clock's rising edge as the machine exits the state.

ASM Design Process

1. **Identify States:** Enumerate all functional states of the control unit.
2. **Construct Flowchart:** Use ASM primitives to map transitions, decisions, and outputs.
3. **Define Microoperations:** Assign register transfers and control-word field values for each state.
4. **Implement Hardware:** Translate the ASM into control logic or microprogram memory.



24. Design Examples

Greatest Common Divisor

Description

The Euclidean greatest common divisor (gcd) algorithm finds the largest positive integer that divides two input values without a remainder. Mathematically:

- gcd(a, a) = a
- gcd(a, b) = gcd(a − b, b) if a > b
- gcd(a, b) = gcd(a, b − a) otherwise

ASM Components

- **States**:
    1. IDLE
    2. COMP
    3. DONE
- **Registers**: A, B
- **Inputs**: X, Y, GO
- **Outputs**: DONE (pulsed when result is ready), A (holds gcd result)

Operation Sequence

1. **IDLE**
   - Wait for GO = 1, while loading initial values:
     - A ← X
     - B ← Y
   - When GO asserts, transition to **COMP**.
2. **COMP**
   - Compare A and B:
     - If A > B, then A ← A − B.
     - Else if B > A, then B ← B − A.
   - Repeat until A = B:
     - When A = B, that value is gcd(X, Y).
   - Transition to **DONE**.
3. **DONE**
   - Pulse DONE output (0→1) to indicate completion.
   - Hold A as final gcd.
   - Return to **IDLE** on reset or next GO pulse.

Guessing Game

Description

A simple interactive game where a rotating 1-of-3 LED pattern moves at ~5 Hz. The player presses one of three push-buttons (G_1, G_2, G_3) to "guess" which LED is currently lit. If a wrong button is pressed, an error LED is asserted. Play pauses until the button is released, then resumes rotation.

ASM Components

- **Inputs**: G_1, G_2, G_3 (push buttons)
- **Outputs**:

- L_1, L_2, L_3 (1-of-3 rotating LED pattern)
- ERR (red LED, asserted on wrong guess)

- **Clock Frequency**: ~5 Hz for rotating pattern

Operation Sequence

1. **Rotate Pattern**
   - On each clock tick, shift the single "1" among $L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow L_1$ in a circular fashion.
2. **Guess Detection**
   - If any G_i is pressed:
     - Check if $L_i = 1$.
       - If matched, do nothing (correct guess).
       - If unmatched, assert ERR = 1.
   - While G_i remains pressed, halt rotation (pattern frozen).
3. **Resume Play**
   - Once the pressed button is released ($G_i = 0$), clear ERR, then resume rotation from current LED.

attachment:59059461-b0a4-4036-aa21-07f740164583:elec205-w14-s25.pdf

---

25. Final Exam Review

Contents

Sequential Circuits Overview

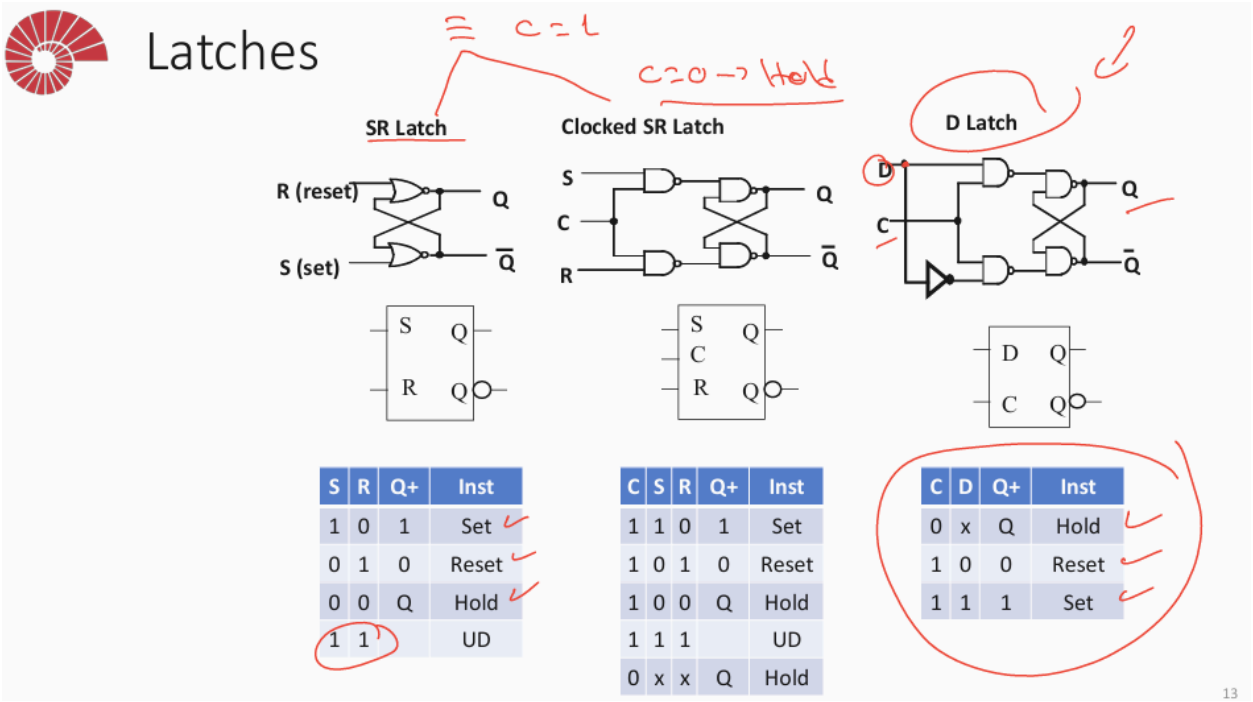
- **Definition:** Outputs depend on current inputs **and** stored state.
- **Model:**

```
Inputs (X) → Combinational Logic → Flip-Flops (state Q) → Output
s (Y)
                  ↑
                  └── Q feedback
```

- **Next-State Function:** Q(t+1) = f(X(t), Q(t))
- **Output Function:**
  - Mealy: Y(t) = g(X(t), Q(t))
  - Moore: Y(t) = h(Q(t))

Latches & Flip-Flops

1. **SR Latch (NOR/NAND):**
   - Stores 1 bit.
   - S=1 R=0 → Set (Q=1); S=0 R=1 → Reset (Q=0); S=R=0 → Hold; S=R=1 → Forbidden.
2. **Clocked SR Latch:**
   - Gate S/R with clock C.
   - When C=1, latch is transparent; C=0, holds state.
3. **D Latch:**
   - Derived from SR latch: D=1 → Set; D=0 → Reset when C=1; C=0 → Hold.
4. **Master–Slave & Edge-Triggered D FF:**
   - Two latches in series: master enabled on C=1, slave on C=0.
   - Edge-triggered FF updates only on clock transition.
5. **JK & T Flip-Flops:**
   - **JK:** J=K=0 → Hold; J=1 K=0 → Set; J=0 K=1 → Reset; J=K=1 → Toggle.
   - **T:** T=0 → Hold; T=1 → Toggle.



Sequential Analysis & Design

1. **Analysis Steps:**
   - Given flip-flop type and logic, derive next-state and output equations.
   - Build a state table: (Present State, Input) → (Next State, Output).
   - Draw state diagram (Mealy: arc labeled X/Y; Moore: state labeled /Y).
2. **Design Procedure:**
   a. **Specification → State Diagram** (identify states, inputs, outputs).
   b. **State Assignment:** Choose binary codes (Gray code often minimizes logic).
   c. **State Table:** List all (PS, X) → (NS, Y).
   d. **Equations:**
      - For D-FF: $D_i = Q_i(t+1)$.
      - For JK: use excitation table to find J,K.
      - For T: $T = Q \oplus Q(t+1)$.
      - Output logic: depends on PS (Moore) or (PS+Input) (Mealy).

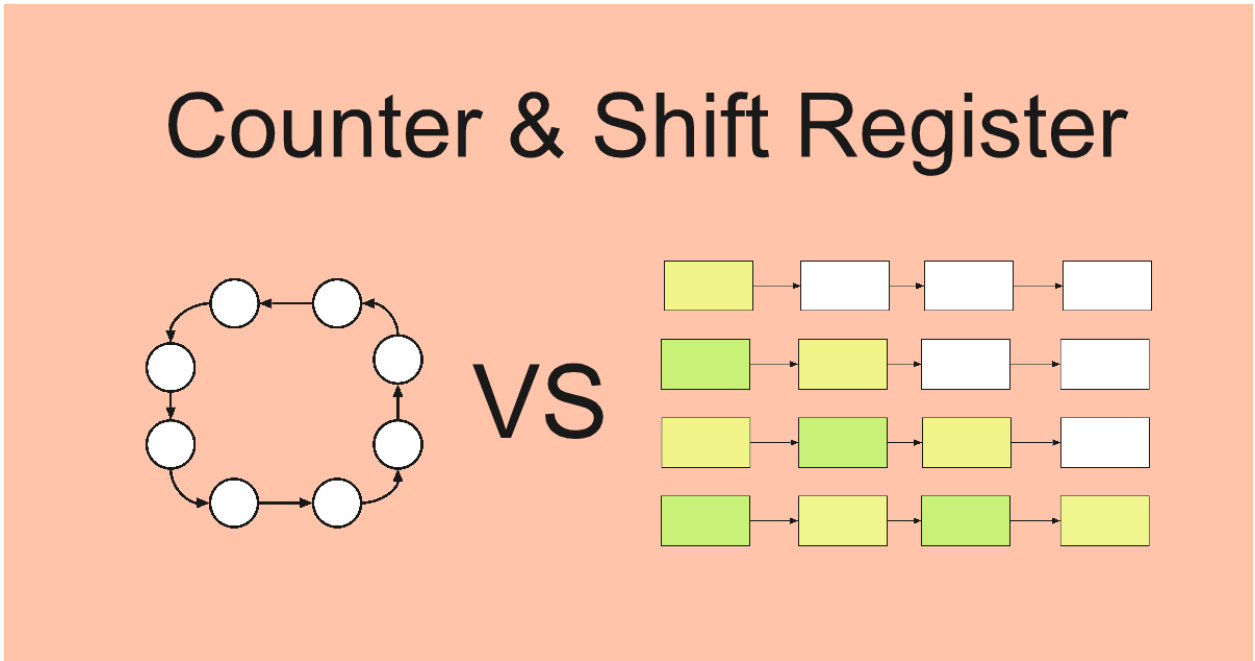
e. **Simplify:** Karnaugh maps or Boolean algebra.

f. **Implementation & Verification:** Map to gates/FFs, simulate/state-table check.

3. **Example (Sequence Detector "1101"):**
   - States: S0(no bits), S1("1"), S2("11"), S3("110").
   - Transitions labeled with input/output.
   - Assign codes (e.g., 00, 01, 11, 10); derive D1,D0 and y = Q1 · ¬Q0 · x.

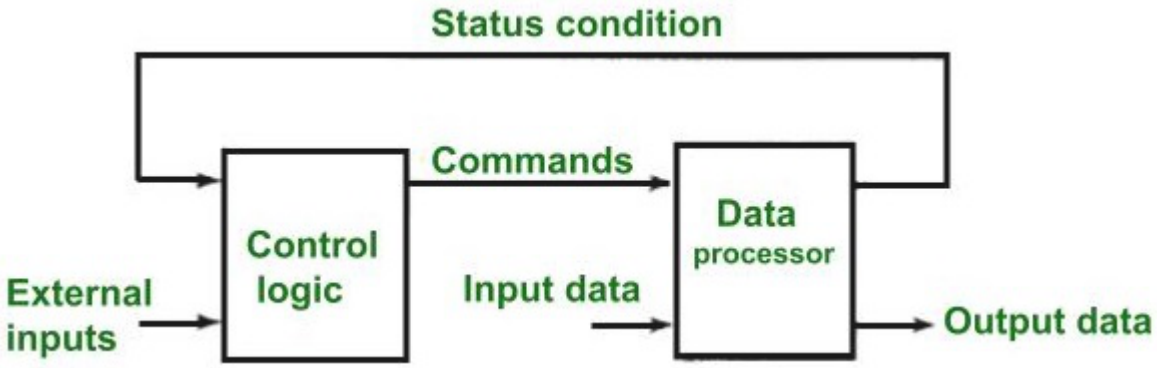
Registers, Counters & Shift Registers

1. **Registers:**
   - n D-FFs storing n-bit word.
   - Microops: load, increment, decrement, bitwise NOT, logical/arithmetic ops via ALU.
2. **Transfer Structures:**
   - **MUX-Based:** Each register input has a MUX to select source.
   - **Bus-Based:** One shared bus, each register output enabled via tri-state buffer.
3. **Shift Registers:**
   - **SISO:** Serial in → shift → serial out.
   - **SIPO:** Serial in → after n clocks, parallel outputs ready.
   - **PISO:** Parallel load → shift out serially.
   - **PIPO:** Standard register (parallel load/read).
4. **Counters:**
   - **Ripple (Asynchronous):** Clock drives LSB; each stage toggles on previous output edge (slower).
   - **Synchronous:** All FFs share clock; next-state logic determines toggles.
   - **Up/Down, Mod-m, BCD:** Use combinational logic to reset or invert as needed.
   - **Example:** 4-bit synchronous up-counter with T-FFs: T_i = Q0 · Q1 · … · Q_{i-1}.



Algorithmic State Machines (ASMs)

- **ASM Chart Symbols:**
  1. **State Box:** Clocked microoperations inside.
  2. **Decision Box:** Conditional test (Yes/No paths).
  3. **Conditional Output Box:** Output action based on decision.
- **Steps to Build ASM:**
  1. Identify clocked actions and decisions.
  2. Draw state boxes with microops.
  3. Add decision diamonds for condition checks.
  4. Label transitions (e.g., “if x=1 → next state; else → another state”).
  5. Derive flip-flop/input equations from ASM.
- **Example (Euclid GCD):**
  - IDLE: wait GO, load A←X,B←Y. → COMP
  - COMP: if A>B, A←A−B; else if A<B, B←B−A; else → DONE
  - DONE: DONE=1 for one cycle, then return to IDLE.



Datapath & Control Basics

1. **Datapath Components:**
   - **Register File:** Multiple registers with two read ports (Bus A, Bus B) and one write port (Bus D).
   - **Mux B:** Select between register B or immediate.
   - **ALU:** Arithmetic/logic operations on Bus A and Mux B output.
   - **Shifter:** Shift/rotate operations on Bus B.
   - **Mux F:** Select ALU or Shifter output.
   - **Mux D:** Select between Mux F or external data for writing.
   - **Status Flags:** Z (zero), N (negative), C (carry), V (overflow).
2. **Control Word Fields (example 16 bits):**
   - [15–13] DA: Destination register (write).
   - [12–10] AA: Source for Bus A.
   - [9–7] BA: Source for Bus B.
   - [6] MB: 0→ register B, 1→ immediate.
   - [5–2] FS: ALU/Shifter function select.
   - [1] MD: 0→ Mux F, 1→ external data.
   - [0] RW: 1→ enable write to DA on rising clock edge.
3. **Function Select (FS) Examples:**

- 0000: F←A (pass A)
- 0010: F←A+B (add)
- 0100: F←A−B (subtract)
- 0110: F←A∧B (AND)
- 0111: F←A∨B (OR)
- 1010: F←shift right(B)
- 1011: F←shift left(B)
- 1100: F←0 (zero)
- 1101: F←1 (one)

4. **Microinstruction Sequencing:**
   - Control word loaded each cycle.
   - On clock: assert control signals → datapath executes ALU/Shifter → Mux D selects data → if RW=1, write to register.
   - Next microaddress: either increment or branch based on flags/inputs.