Digital Logic

Unit 1: Introduction

1.1 Digital Signals and Waveforms

Definition
A **digital signal** is a discrete signal that represents data in binary form (0s and 1s). Unlike analog signals, digital signals do not vary continuously but have specific discrete levels.

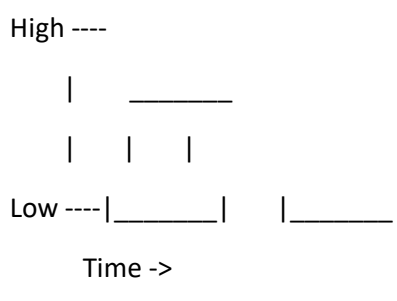
Characteristics of Digital Signals

1. **Binary Representation**: Two distinct voltage levels:
    - High voltage = 1
    - Low voltage = 0
2. **Discrete Time Intervals**: The signal is sampled at fixed intervals.
3. **Noise Resistance**: Less affected by noise compared to analog signals.
4. **Waveforms**: Typically represented as square waves.

Waveform Representation
A digital waveform alternates between high and low states.

Key Parameters	Description
Amplitude	Voltage levels (high and low).
Time Period (T)	Duration of one complete cycle.
Frequency (f)	f=1T, the number of cycles per second.
Duty Cycle	Duty Cycle=Time HighTotal Time×100.

Diagram:



1.2 Digital Logic and Operation

Definition
Digital logic is the foundation of digital systems. It uses logic gates (AND, OR, NOT, etc.) to process binary data (0s and 1s).

Key Components

1. **Logic Gates**: Perform basic operations on binary data:
    - AND, OR, NOT
    - NAND, NOR, XOR, XNOR (derived gates).
2. **Boolean Algebra**: A mathematical framework to simplify logic expressions.
3. **Combinational Circuits**: Circuits where the output depends only on the current input (e.g., Adders, Multiplexers).
4. **Sequential Circuits**: Circuits with memory elements where the output depends on the current input and past states (e.g., Flip-Flops, Counters).

Applications of Digital Logic

- Arithmetic operations.
- Control systems.
- Signal processing.

1.3 Digital Computers and Integrated Circuits (IC)

Digital Computers
A **digital computer** operates using digital signals and processes binary data. It consists of the following key components:

1. **Input Unit**: Converts user data into digital signals.

2. **Central Processing Unit (CPU)**: Processes data using logic circuits.
3. **Memory Unit**: Stores instructions and data.
4. **Output Unit**: Converts digital data into human-readable form.

Integrated Circuits (ICs)
An **Integrated Circuit (IC)** is a compact electronic circuit made by embedding multiple components (transistors, resistors, etc.) into a single chip.

Types of ICs

1. **Small Scale Integration (SSI)**: Few gates per chip.
2. **Medium Scale Integration (MSI)**: Hundreds of gates per chip.
3. **Large Scale Integration (LSI)**: Thousands of gates per chip (used in CPUs).
4. **Very Large Scale Integration (VLSI)**: Millions of gates per chip (used in advanced processors).

Applications

- Computers.
- Telecommunication systems.
- Control systems in electronics.

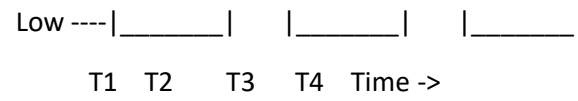
1.4 Clock Waveform

Definition
A **clock waveform** is a periodic signal used to synchronize operations in digital circuits. The clock ensures that all components operate in coordination.

Characteristics

1. **Periodicity**: Alternates between high (1) and low (0) at regular intervals.
2. **Frequency**: Determines the speed of operation of the circuit (f=1T).
3. **Duty Cycle**: Fraction of time the clock remains high in one cycle.

Diagram of Clock Waveform



Applications

1. Timing control in processors and memory.
2. Synchronization in sequential circuits.
3. Frequency division in counters.

Unit2: Number System

1.1 Introduction to Number Systems

A **number system** defines a set of values to represent quantities. It is a mathematical notation for representing numbers of a given set. Digital computers operate on different number systems, with the most common ones being:

- **Decimal (Base 10)**
- **Binary (Base 2)**
- **Octal (Base 8)**
- **Hexadecimal (Base 16)**

Each number system is characterized by its **base (radix)**, which is the total number of unique digits, including zero, used to represent numbers.

1.2 Types of Number Systems

1.2.1 Decimal Number System

- Base: 10
- Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Positional Value: Each digit's position represents a power of 10.
- Example: $23710=(2\times102)+(3\times101)+(7\times100)=200+30+723710$ $=(2\times102)+(3\times101)+(7\times100)=200+30+7$.

1.2.2 Binary Number System

- Base: 2
- Digits: 0, 1
- Used in: Digital electronics and computers for logic representation.
- Positional Value: Each digit represents a power of 2.
- Example: $10112=(1\times23)+(0\times22)+(1\times21)+(1\times20)=8+0+2+1=111010112$ $=(1\times23)+(0\times22)+(1\times21)+(1\times20)=8+0+2+1=1110$.

Diagram: Binary Representation

plaintext

Copy code

+-----+-----+-----+-----+ | 8 | 4 | 2 | 1 | +-----+-----+-----+-----+ | 1 | 0 | 1 | 1 | +-----+-----+-----+-----+ Binary Number: 1011

1.2.3 Octal Number System

- Base: 8
- Digits: 0, 1, 2, 3, 4, 5, 6, 7
- Positional Value: Each digit represents a power of 8.
- Example: $7458=(7\times82)+(4\times81)+(5\times80)=448+32+5=485107458$ $=(7\times82)+(4\times81)+(5\times80)=448+32+5=48510$.

1.2.4 Hexadecimal Number System

- Base: 16
- Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (10), B (11), C (12), D (13), E (14), F (15)
- Positional Value: Each digit represents a power of 16.
- Example: $3F716=(3\times162)+(15\times161)+(7\times160)=768+240+7=1015103F716$ $=(3\times162)+(15\times161)+(7\times160)=768+240+7=101510$.

1.3 Conversion Between Number Systems

To convert numbers between systems, different techniques are used:

1.3.1 Binary to Decimal Conversion

Sum the positional values:
Example: 11012→(1×23)+(1×22)+(0×21)+(1×20)=131011012→(1×23)+(1×22)+(0×21)+(1×20)=1310.

1.3.2 Decimal to Binary Conversion

Repeatedly divide the number by 2 and record remainders.
Example: Convert 19101910:

1. 19÷2=919÷2=9 remainder 11
2. 9÷2=49÷2=4 remainder 11
3. 4÷2=24÷2=2 remainder 00
4. 2÷2=12÷2=1 remainder 00
5. 1÷2=01÷2=0 remainder 11

Binary: 1910=1001121910=100112.

Diagram: Decimal to Binary Conversion

plaintext

Copy code

19 ÷ 2 = 9 R 1 9 ÷ 2 = 4 R 1 4 ÷ 2 = 2 R 0 2 ÷ 2 = 1 R 0 1 ÷ 2 = 0 R 1 Binary Representation: 10011

Unit 3: Combinational Logic Design

2.1 Introduction to Boolean Algebra

Boolean algebra, developed by George Boole, is the foundation of digital logic. It involves mathematical operations on binary values (0 and 1), which represent **False** and **True**, respectively.

2.2 Basic Boolean Operations

2.2.1 AND Operation

- Symbol: · or AND
- Rule: A·B=1 if and only if A=1 and B=1.
- Truth Table:
  | A | B | A·B | |—|—|———-| | 0 | 0 | 0 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 |

2.2.2 OR Operation

- Symbol: + or OR
- Rule: A+B=1 if either A=1 or B=1.
- Truth Table:
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

2.2.3 NOT Operation

- Symbol: $\overline{A}$ or NOT
- Rule: $\overline{A}$=1 if A=0, and $\overline{A}$=0 if A=1.
- Truth Table:
A	$\overline{A}$
0	1
1	0

2.3 Laws of Boolean Algebra

2.3.1 Commutative Laws

- A+B=B+A
- A·B=B·A

2.3.2 Associative Laws

- (A+B)+C=A+(B+C)
- (A·B)·C=A·(B·C)

2.3.3 Distributive Laws

- A·(B+C)=(A·B)+(A·C)
- A+(B·C)=(A+B)·(A+C)

2.4 Simplification Using Boolean Laws

Simplification of Boolean expressions reduces circuit complexity.

Example
Simplify A·(A+B):

1. Apply Distributive Law: A·A+A·B
2. Simplify using A·A=A: A+A·B
3. Apply Absorption Law: A.

2.5 De Morgan's Theorems

De Morgan's Theorems are crucial for simplifying complemented expressions:

1st Theorem:
$A \cdot B^{-} = A^{-} + B^{-}$

2nd Theorem:
$A + B^{-} = A^{-} \cdot B^{-}$

Proof for 1st Theorem

A	B	A·B	A^{-}	B^{-}	$A^{-}+B^{-}$	$A \cdot B^{-}$
0	0	0	1	1	1	1
0	1	0	1	0	1	1
1	0	0	0	1	1	1
1	1	1	0	0	0	0

Diagram: Basic Logic Gate Symbols

AND Gate

A ----| & |---- Output (A · B)

B ----|

OR Gate

A ----|≥1|---- Output (A + B)

B ----|

NOT Gate

A ----|>o---- Output (¬A)

Unit 4: Counters and Registors

3.1 Introduction to Logic Gates

Logic gates are the building blocks of digital circuits. They perform basic logical functions and are implemented using diodes, transistors, or integrated circuits. Each gate corresponds to a Boolean function and has a specific truth table.

3.2 Types of Basic Logic Gates

3.2.1 AND Gate

- **Symbol**: ·
- **Function**: Outputs 1 if all inputs are 1.
- **Truth Table**:
  | Input A | Input B | Output A·B | |———|———|————————–| | 0 | 0 | 0 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 |

Diagram:

A ----| & |---- Output

B ----|

3.2.2 OR Gate

- **Symbol**: +
- **Function**: Outputs 1 if at least one input is 1.
- **Truth Table**:
  | Input A | Input B | Output A+B | |———|———|————————-| | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 1 |

Diagram:

A ----|≥1|---- Output

B ----|

3.2.3 NOT Gate

- **Symbol**: $\bar{A}$
- **Function**: Inverts the input.

- **Truth Table**:
  | Input A | Output $\overline{A}$ | |———|———————| | 0 | 1 | | 1 | 0 |

Diagram:

A ----|>o---- Output

3.3 Universal Gates

Universal gates can be used to implement any Boolean function.

3.3.1 NAND Gate

- **Symbol**: $A \cdot \overline{B}$
- **Function**: Outputs 0 only if all inputs are 1.
- **Truth Table**:
  | Input A | Input B | Output $\overline{A \cdot B}$ | |———|———|———————————-| | 0 | 0 | 1 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 |

Diagram:

A ----| & |----|>o---- Output

B ----|

3.3.2 NOR Gate

- **Symbol**: $A+\overline{B}$
- **Function**: Outputs 1 only if all inputs are 0.
- **Truth Table**:
  | Input A | Input B | Output $\overline{A+B}$ | |———|———|———————————–| | 0 | 0 | 1 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 0 |

Diagram:

A ----|≥1|----|>o---- Output

B ----|

3.4 Exclusive Gates

3.4.1 XOR Gate

- **Symbol**: $A \oplus B$

- **Function**: Outputs 1 if inputs are different.
- **Truth Table**:
  | Input A | Input B | Output $A \oplus B$ | |———|———|——————-| | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 |

Diagram:

A ----|>1|---- Output

B ----|

3.4.2 XNOR Gate

- **Symbol**: $A \oplus B^{-}$
- **Function**: Outputs 1 if inputs are the same.
- **Truth Table**:
  | Input A | Input B | Output $A \oplus B^{-}$ | |———|———|———————–| | 0 | 0 | 1 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 |

Diagram:

A ----|>1|----|>o---- Output

B ----|

3.5 Implementation of Logic Gates

- **AND Gate with NAND Gates**:
  To implement A·B using NAND:

1. Connect A and B to a NAND gate.
2. Invert the output using another NAND gate.

- **OR Gate with NOR Gates**:
  To implement A+B using NOR:

1. Invert both A and B using NOR gates.
2. Apply NOR on the results.

Unit 5: Sequential Logic Design

Chapter 4: Karnaugh Maps (K-Maps)

4.1 Introduction to Karnaugh Maps

A **Karnaugh Map (K-Map)** is a visual method used to simplify Boolean expressions without using Boolean algebra. It provides a systematic way of minimizing logic circuits by grouping terms to eliminate redundant variables.

4.2 Structure of K-Maps

- **Rows and Columns**: Represent combinations of variables.
- **Cells**: Contain the output value (0 or 1) for the corresponding input combination.
- The number of cells equals 2n, where n is the number of variables.

4.3 Example: 2-Variable K-Map

For 2 variables A and B:

- Possible combinations: 00,01,10,11
- K-Map structure:

A\B	0	1
0	0	1
1	1	0

4.4 Example: 3-Variable K-Map

For 3 variables A,B,C:

- Possible combinations: 000,001,010,011,100,101,110,111
- K-Map structure:

AB\C	0	1
00	0	1

AB\C	0	1
01	1	0
11	1	1
10	0	1

4.5 Simplification Using K-Maps

Steps:

1. **Plot the truth table**: Write the values of the Boolean function for all input combinations.
2. **Map the values**: Transfer the output values (1 or 0) into the K-Map cells.
3. **Group adjacent 1s**: Create groups of 1s in powers of 2 (1, 2, 4, 8, etc.).
4. **Derive simplified expression**: Write a simplified Boolean equation for each group.

4.6 Example Problem

Simplify F(A,B,C)=Σ(1,3,5,7) using a 3-variable K-Map.

1. **Truth Table**:
   | A | B | C | F | |———|———|———|———| | 0 | 0 | 0 | | 0 | 0 | 1 | 1 | | 0 | 1 | 0 | 0 | | 0 | 1 | 1 | 1 | | 1 | 0 | 0 | 0 | | 1 | 0 | 1 | 1 | | 1 | 1 | 0 | 0 | | 1 | 1 | 1 | 1 |
2. **K-Map Representation**:
   | AB\C | 0 | 1 | |————————–|——|——| | **00** | 0 | 1 | | **01** | 0 | 1 | | **11** | 0 | 1 | | **10** | 0 | 1 |
3. **Group Adjacent 1s**:

- Group (0,1),(1,1),(1,1),(0,1).

4. **Simplified Equation**:
   F=A+BC.

4.7 K-Map for 4 Variables

For 4 variables (A,B,C,D), the K-Map has 16 cells, arranged as:

AB\CD	00	01	11	10
00	F1	F2	F3	F4
01	F5	F6	F7	F8
11	F9	F10	F11	F12
10	F13	F14	F15	F16

Simplification follows the same grouping process.

4.8 Benefits of K-Maps

- Simplifies expressions to reduce circuit complexity.
- Visual representation aids in identifying patterns.
- Reduces errors in Boolean simplification.

Chapter 5: Combinational Circuits

5.1 Introduction to Combinational Circuits

A **combinational circuit** is a type of digital circuit where the output depends solely on the current input values. There is no memory or storage involved, and these circuits are built using logic gates.

5.2 Characteristics of Combinational Circuits

1. Output depends only on the present input values.
2. No feedback paths (no connection from output back to input).
3. Built using basic gates like AND, OR, NOT, NAND, NOR, XOR, and XNOR.

5.3 Types of Combinational Circuits

5.3.1 Adders Adders perform arithmetic addition of binary numbers.

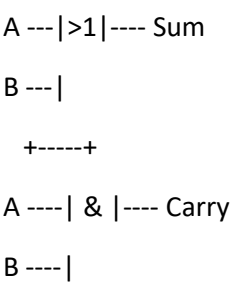
Half Adder

- Adds two bits and produces a sum and a carry.

- **Inputs**: A, B
- **Outputs**: Sum=A⊕B, Carry=A·B
- **Truth Table**:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Circuit Diagram:



Full Adder

- Adds three bits (two inputs and a carry-in).
- **Inputs**: A, B, Cin
- **Outputs**: Sum=A⊕B⊕Cin, Cout=(A·B)+(Cin·(A⊕B)).
- **Truth Table**:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0

A	B	Cin	Sum	Cout
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

5.3.2 Multiplexers (MUX) A multiplexer selects one input from multiple inputs and forwards it to the output, based on the selection lines.

- **Inputs**: n data inputs, log2(n) select lines.
- **Example**: A 4-to-1 MUX has 4 data inputs and 2 select lines.

Truth Table for 4-to-1 MUX:

Select Lines S1,S0	Output Y
00	D0
01	D1
10	D2
11	D3

Diagram:

D0 ----\

D1 ----|\

D2 ----| >--- Output

D3 ----|/

S0 --|

S1 --|

5.3.3 Decoders A decoder converts n-bit binary input into 2n outputs. Each output corresponds to one of the possible combinations of the input.

- **Example**: A 3-to-8 decoder generates 8 outputs for 3 input lines.

Truth Table for 3-to-8 Decoder:

A	B	C	O0	O1	O2	O3	O4	O5	O6
0	0	0	1	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0
…	…	…	…	…	…	…	…	…	…

5.4 Applications of Combinational Circuits

- Arithmetic operations (adders, subtractors).
- Data selection and routing (multiplexers).
- Encoding and decoding (encoders, decoders).
- Data comparison (comparators).

Chapter 6: Sequential Circuits

6.1 Introduction to Sequential Circuits

Unlike combinational circuits, **sequential circuits** depend on both the current inputs and the history of inputs. This is achieved by using memory elements to store past states.

- **Key Characteristics**:

1. Output depends on current and previous inputs.
2. Memory elements store the state of the circuit.
3. Built using logic gates and flip-flops.

6.2 Types of Sequential Circuits

6.2.1 Synchronous Sequential Circuits

- Operate with a clock signal.
- Changes occur at specific intervals, determined by the clock pulse.

6.2.2 Asynchronous Sequential Circuits

- Do not operate with a clock signal.
- Changes occur whenever inputs change.

6.3 Basic Building Blocks of Sequential Circuits

6.3.1 Flip-Flops

Flip-flops are bistable devices (can hold one of two stable states: 0 or 1) used to store a single bit of data.

Types of Flip-Flops

6.3.1.1 SR Flip-Flop

- Inputs: S (Set), R (Reset)
- Outputs: Q, $\overline{Q}$
- Function:
    - S=1,R=0: Sets Q=1.
    - S=0,R=1: Resets Q=0.
    - S=0,R=0: Holds the previous state.
    - S=1,R=1: Invalid state.

Truth Table:

S	R	Q	Q^{-}
0	0	Previous	Complement
0	1	0	1
1	0	1	0
1	1	Invalid	Invalid

Diagram:

```
S ----| & |----\
      |       |---- Q
R ----| & |----/
```

6.3.1.2 D Flip-Flop

- Input: D (Data)
- Output: Q
- Function: Captures the value of D on the clock's rising or falling edge.
- Eliminates the invalid state of SR flip-flop.

Truth Table:

D	Q
0	0
1	1

6.3.1.3 JK Flip-Flop

- Inputs: J, K
- Outputs: Q, Q^{-}

- Function:
    - J=1,K=0: Sets Q=1.
    - J=0,K=1: Resets Q=0.
    - J=1,K=1: Toggles Q.

Truth Table:

J	K	Q (Next State)
0	0	Hold
0	1	0
1	0	1
1	1	Toggle

6.3.1.4 T Flip-Flop

- Input: T (Toggle)
- Output: Q
- Function: Toggles Q on each clock pulse if T=1; holds the state if T=0.

Truth Table:

T	Q (Next State)
0	Hold
1	Toggle

6.4 Registers and Counters

6.4.1 Registers

A register is a group of flip-flops used to store multiple bits of data.

- **Types**:

1. Shift Registers: Shift stored data left or right.
2. Parallel Registers: Store data in parallel form.

6.4.2 Counters

Counters are sequential circuits that count pulses or events.

- **Types**:
    1. Asynchronous (Ripple) Counters: Count without a clock signal.
    2. Synchronous Counters: Count with all flip-flops triggered by the same clock.

6.5 Applications of Sequential Circuits

1. **Registers** for temporary data storage.
2. **Counters** for event counting.
3. **Frequency Dividers** for clock signal generation.
4. **Finite State Machines (FSMs)** for implementing specific algorithms in hardware.

Chapter 7: Finite State Machines (FSMs)

7.1 Introduction to Finite State Machines

A **Finite State Machine (FSM)** is a computational model used to design sequential logic circuits. It consists of a finite number of states, transitions between those states, and outputs determined by the states and inputs.

7.2 Components of FSM

1. **States**: Define the current condition of the system.
2. **Inputs**: External signals that influence transitions.
3. **Outputs**: Determined by the state (and sometimes inputs).
4. **Transitions**: Rules or conditions for moving from one state to another.

7.3 Types of FSMs

7.3.1 Mealy Machine

- Output depends on both the current state and input.
- Faster response since the output can change immediately with the input.

Mealy Machine Representation:

- **State Diagram**: States are represented by circles; transitions show input/output.

7.3.2 Moore Machine

- Output depends only on the current state.
- Simpler design but slower response since the output changes only after the state transition.

Moore Machine Representation:

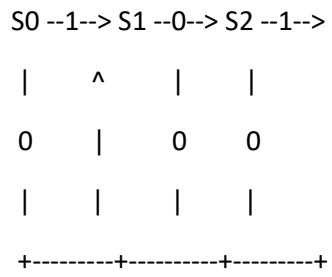
- **State Diagram**: States are represented by circles with outputs inside; transitions show only inputs.

7.4 Example: FSM for Sequence Detector

Design an FSM to detect the sequence “101” in a binary stream.

Steps:

1. **Define States**:
    - S0: Initial state.
    - S1: Detected the first ‘1’.
    - S2: Detected ’10’.
    - S3: Detected ‘101’ (final state).
2. **State Transition Table**:
   | Current State | Input X | Next State | Output | |———————|———————|———————|———–|
   | S0 | 0 | S0 | 0 | | S0 | 1 | S1 | 0 | | S1 | 0 | S2 | 0 | | S1 | 1 | S1 | 0 | | S2 | 0 | S0 | 0 | | S2 | 1 | S3 | 1 | | S3 | 0 | S2 | 0 | | S3 | 1 | S1 | 0 |
3. **State Diagram**:



4. **Implementation Using Flip-Flops**:
   Use a combination of D flip-flops and logic gates to implement the transition and output logic.

7.5 Applications of FSMs

1. **Control Systems**: Traffic lights, elevators, vending machines.
2. **Pattern Recognition**: Sequence detectors.
3. **Data Communication**: Protocol handling in networks.
4. **Hardware Implementation**: Control units in processors.

Chapter 8: FSM Design Examples

8.1 Example 1: Traffic Light Controller

Problem Statement

Design an FSM to control a traffic light with three lights: **Green (G)**, **Yellow (Y)**, and **Red (R)**. The lights should transition in the sequence:
G→Y→R.

Steps to Design

Step 1: Define States

- S0: Green Light ON.
- S1: Yellow Light ON.
- S2: Red Light ON.

Step 2: Define Transitions
The transitions occur based on a clock pulse.

- S0→S1: After 5 clock pulses (Green to Yellow).
- S1→S2: After 2 clock pulses (Yellow to Red).
- S2→S0: After 5 clock pulses (Red to Green).

Step 3: State Transition Table

Current State	Clock Count	Next State	Outputs (G, Y, R)
S0	0 to 5	S1	1, 0, 0

Current State	Clock Count	Next State	Outputs (G, Y, R)
S1	6 to 7	S2	0, 1, 0
S2	8 to 12	S0	0, 0, 1

Step 4: State Diagram

```
[S0: G=1, Y=0, R=0] --5--> [S1: G=0, Y=1, R=0]
        ^                    |
        |                    v
        +---------5---------- [S2: G=0, Y=0, R=1]
```

Step 5: Implementation

- Use a **3-bit counter** to track clock pulses and generate control signals.
- Use combinational logic to define state transitions.

Circuit Diagram Overview:

```
Clock ----> Counter ----> State Logic ----> Output Logic
```

8.2 Example 2: Sequence Detector (Detecting “110”)

Problem Statement

Design an FSM to detect the sequence “110” in a binary input stream.

Steps to Design

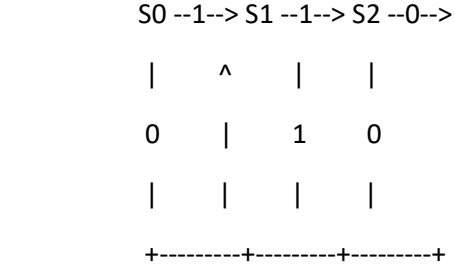
Step 1: Define States

- S0: Initial state (no input detected).
- S1: Detected first ‘1’.
- S2: Detected “11”.
- S3: Detected “110” (final state).

Step 2: State Transition Table

Current State	Input X	Next State	Output
S0	0	S0	0
S0	1	S1	0
S1	0	S0	0
S1	1	S2	0
S2	0	S3	1
S2	1	S2	0
S3	0	S0	0
S3	1	S1	0

Step 3: State Diagram



Step 4: Implementation

- Use 2 flip-flops to represent states S0,S1,S2,S3.
- Use combinational logic to define state transitions and outputs.

Circuit Diagram Overview:

- Input: X, Clock.
- Outputs: Detected sequence flag.

8.3 Example 3: Elevator Control System

Problem Statement

Design an FSM for a 3-floor elevator control system with the following conditions:

1. The elevator can move up or down one floor at a time.
2. Stops at the floor if a request is made.

Steps:

1. Define states for each floor (F0,F1,F2).
2. Include transitions based on button press (Up or Down).
3. Implement with flip-flops and output logic.