ÇUKUROVA ÜNİVERSİTESİ

Department of Computer Engineering

CEN 263 Digital Design

Autumn 2024

Lecture 6

Week 6 Outlines

- SystemVerilog modelling (if-else & case)
- Introduction to Finite State Machines (FSMs)
- Recap with Questions

SystemVerilog modelling - **if statements**

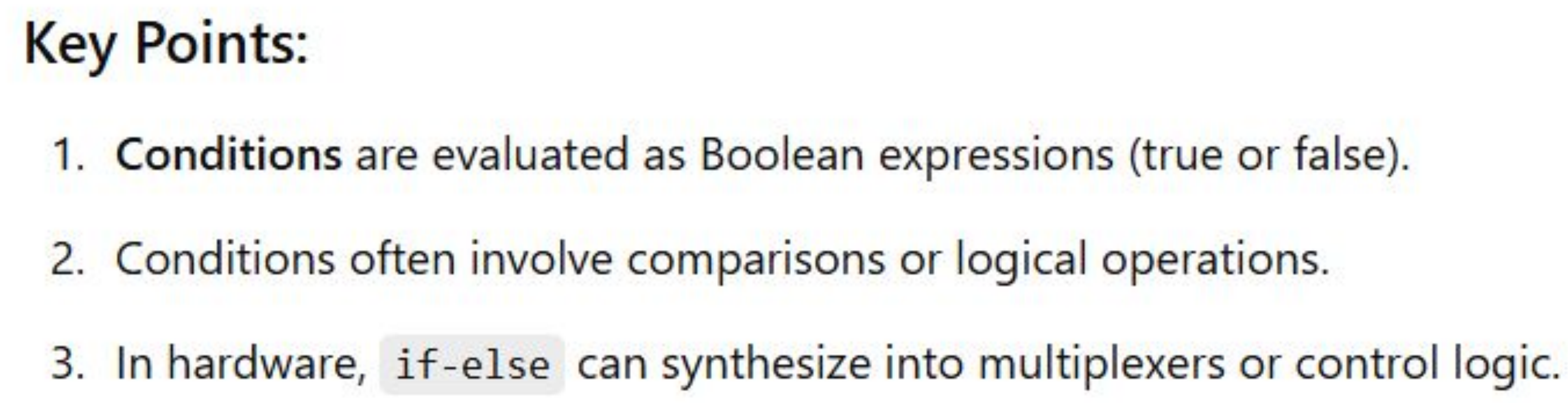
always/process statements may also contain **if statements**. The if statement may be followed by an else statement. If all possible input combinations are handled, the statement implies combinational logic; otherwise, it produces sequential logic

SystemVerilog modelling

The `if-else` statement in SystemVerilog is a conditional construct used to execute certain blocks of code based on a condition's truth value.

It is similar to what you find in other programming languages but tailored to hardware description and verification.

SystemVerilog modelling



Key Points:

1. **Conditions** are evaluated as Boolean expressions (true or false).
2. Conditions often involve comparisons or logical operations.
3. In hardware, `if-else` can synthesize into multiplexers or control logic.

SystemVerilog modelling

Ex: Elevator Control System

Suppose we want to decide whether the elevator should go up, down, or stay based on the current and target floors.

```
module elevator_controller(input logic [3:0] current_floor,
                           input logic [3:0] target_floor,
                           output logic move_up, move_down);

    always_comb begin
        if (current_floor < target_floor) begin
            move_up   = 1'b1;
            move_down = 1'b0;
        end else if (current_floor > target_floor) begin
            move_up   = 1'b0;
            move_down = 1'b1;
        end else begin
            move_up   = 1'b0; // Stay at the current floor
            move_down = 1'b0;
        end
    end

endmodule
```

SystemVerilog modelling

Ex: Temperature Control System

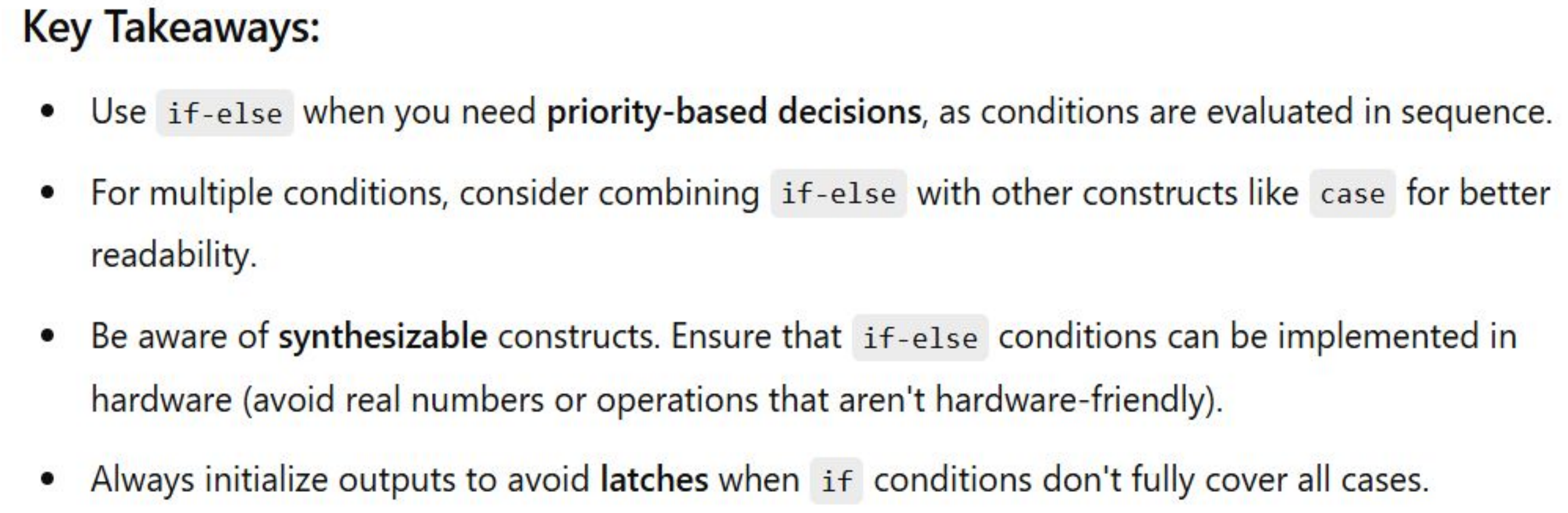
A thermostat that turns on heating or cooling based on the current temperature compared to a setpoint.

```
module thermostat(input logic [7:0] temperature,
                  input logic [7:0] setpoint,
                  output logic heat_on, cool_on);

    always_comb begin
        if (temperature < setpoint) begin
            heat_on = 1'b1;
            cool_on = 1'b0;
        end else if (temperature > setpoint) begin
            heat_on = 1'b0;
            cool_on = 1'b1;
        end else begin
            heat_on = 1'b0; // Turn off both if the temperature is at the setpoint
            cool_on = 1'b0;
        end
    end

endmodule
```

SystemVerilog modelling



Key Takeaways:

- Use `if-else` when you need **priority-based decisions**, as conditions are evaluated in sequence.
- For multiple conditions, consider combining `if-else` with other constructs like `case` for better readability.
- Be aware of **synthesizable** constructs. Ensure that `if-else` conditions can be implemented in hardware (avoid real numbers or operations that aren't hardware-friendly).
- Always initialize outputs to avoid **latches** when `if` conditions don't fully cover all cases.

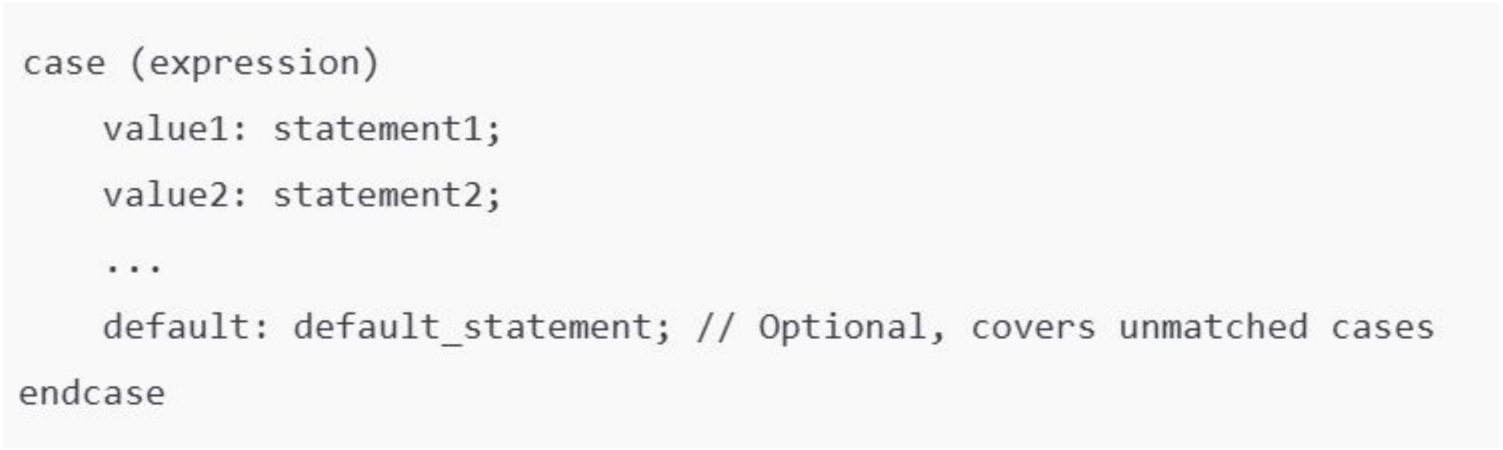
SystemVerilog modelling - **Case Statements**

A better application of using the always/process statement for combinational logic is a seven-segment display decoder that takes advantage of the case statement that must appear inside an always/process statement.

SystemVerilog modelling

The `case` statement in SystemVerilog is a **conditional branching construct** used to execute different blocks of code based on the value of an expression. It is similar to a multi-way `if-else` but often more concise and easier to read, especially when dealing with multiple discrete conditions.

SystemVerilog modelling



```
case (expression)
    value1: statement1;
    value2: statement2;
    ...
    default: default_statement; // Optional, covers unmatched cases
endcase
```

SystemVerilog modelling

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                  abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of the default base, base 10).

The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In SystemVerilog, `case` statements must appear inside `always` statements.

SystemVerilog modelling

Ex: Elevator Floor Indicator

```
module elevator_floor_indicator(
    input logic [2:0] floor,       // 3-bit input for floor number
    output logic [7:0] floor_display // 8-bit output for display encoding
);
    always_comb begin
        case (floor)
            3'b000: floor_display = 8'b00000001; // Floor 0
            3'b001: floor_display = 8'b00000010; // Floor 1
            3'b010: floor_display = 8'b00000100; // Floor 2
            3'b011: floor_display = 8'b00001000; // Floor 3
            3'b100: floor_display = 8'b00010000; // Floor 4
            3'b101: floor_display = 8'b00100000; // Floor 5
            3'b110: floor_display = 8'b01000000; // Floor 6
            3'b111: floor_display = 8'b10000000; // Floor 7
            default: floor_display = 8'b00000000; // No floor (invalid)
        endcase
    end
endmodule
```

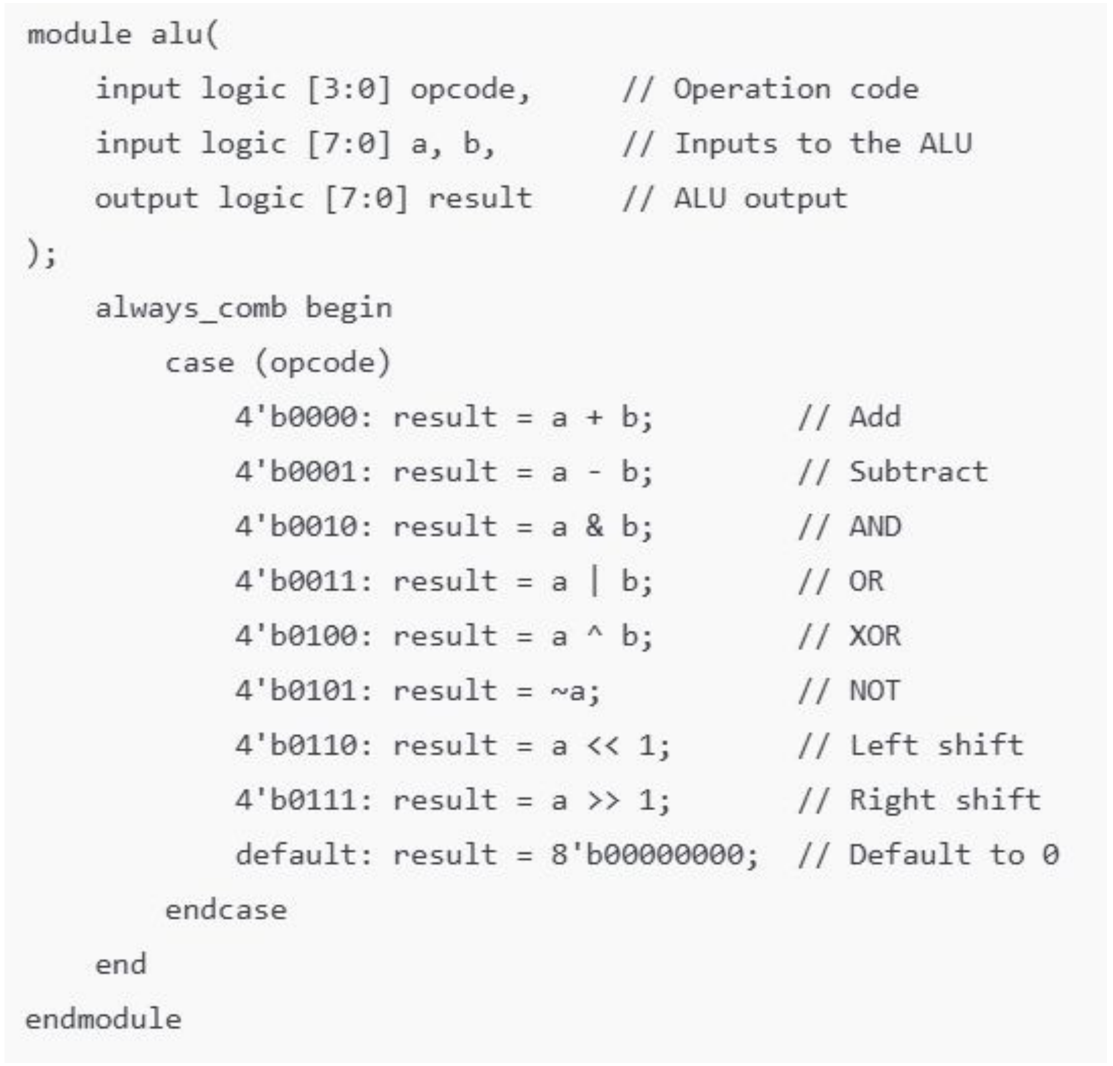
SystemVerilog modelling

Ex: ALU Operation Selector

An **Arithmetic Logic Unit (ALU)** performs different operations based on a control signal.

SystemVerilog modelling

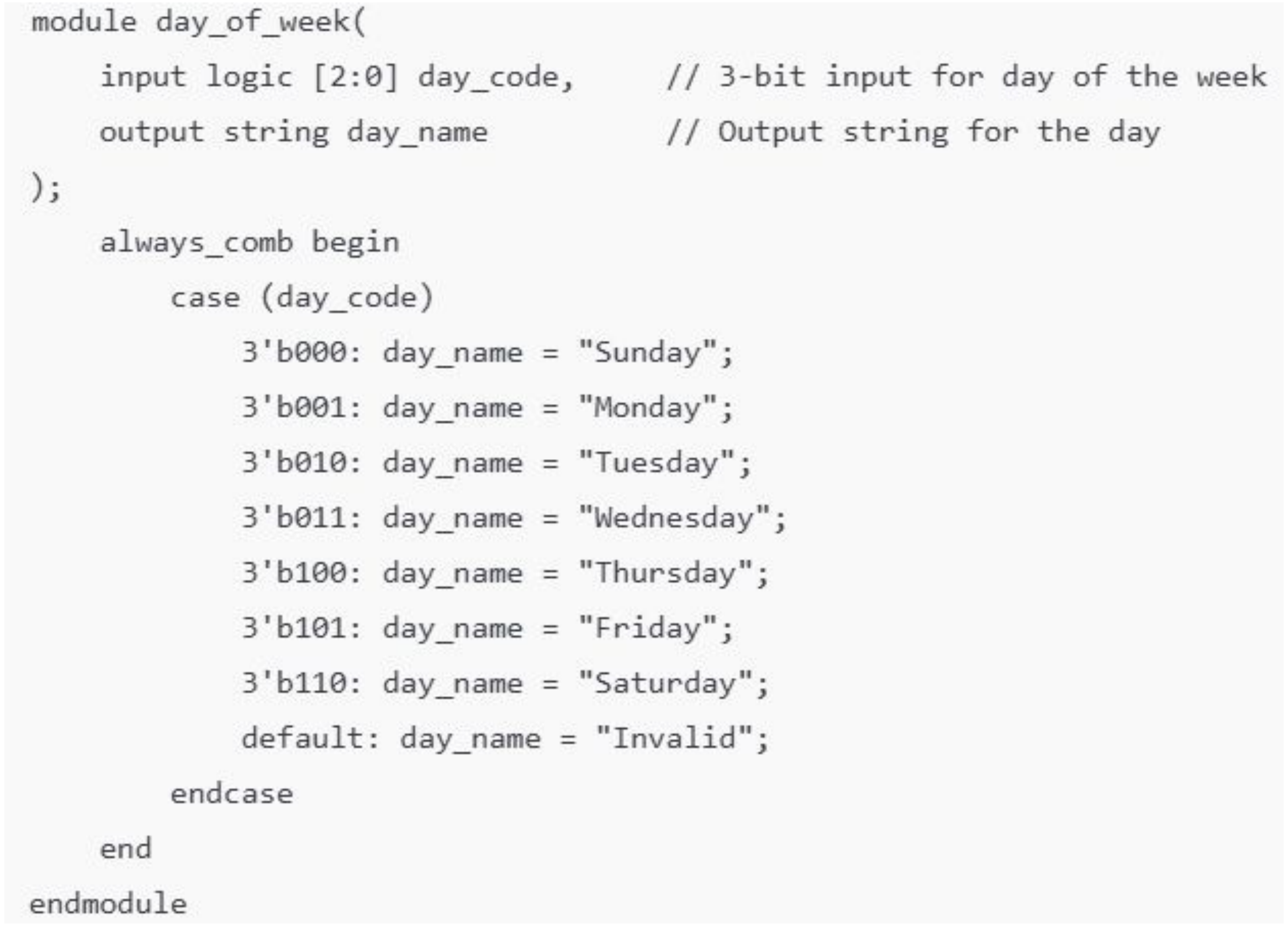
Ex: ALU Operation Selector



```
module alu(
    input logic [3:0] opcode,     // Operation code
    input logic [7:0] a, b,       // Inputs to the ALU
    output logic [7:0] result     // ALU output
);
    always_comb begin
        case (opcode)
            4'b0000: result = a + b;         // Add
            4'b0001: result = a - b;         // Subtract
            4'b0010: result = a & b;         // AND
            4'b0011: result = a | b;         // OR
            4'b0100: result = a ^ b;         // XOR
            4'b0101: result = ~a;            // NOT
            4'b0110: result = a << 1;        // Left shift
            4'b0111: result = a >> 1;        // Right shift
            default: result = 8'b00000000;  // Default to 0
        endcase
    end
endmodule
```

SystemVerilog modelling

Ex: Day of the Week Decoder



```
module day_of_week(
    input logic [2:0] day_code,     // 3-bit input for day of the week
    output string day_name          // Output string for the day
);
    always_comb begin
        case (day_code)
            3'b000: day_name = "Sunday";
            3'b001: day_name = "Monday";
            3'b010: day_name = "Tuesday";
            3'b011: day_name = "Wednesday";
            3'b100: day_name = "Thursday";
            3'b101: day_name = "Friday";
            3'b110: day_name = "Saturday";
            default: day_name = "Invalid";
        endcase
    end
endmodule
```

Key Features of `case`:

1. **Mutually Exclusive Evaluation**:
   - Only one case is executed. Once a match is found, other cases are ignored.
   - Unlike `if-else`, the conditions in `case` are based on exact matches, not ranges or comparisons.
2. **Default Case**:
   - Acts as a fallback if no match is found. It's good practice to include a `default` for complete coverage.
3. **Synthesizability**:
   - `case` maps well to hardware constructs like decoders, multiplexers, or state machines.
4. **Case Variants**:
   - `unique` **case**: Ensures only one case is valid and throws an error if multiple cases match.
   - `priority` **case**: Enforces priority between cases, similar to an `if-else` chain.

Applications of `case`:

1. **State Machines**:
   - Used for controlling behavior in finite state machines (FSMs).
2. **Instruction Decoders**:
   - Common in processors to interpret opcode instructions.
3. **Control Logic**:
   - Used in decision-making circuits like traffic lights or elevators.
4. **Multiplexers**:
   - Easily implemented using `case` by selecting among multiple inputs.

What are FSMs?

What are FSMs?

Finite State Machines (FSMs) are fundamental models of computation used extensively in digital design.

They are used to design systems where the output depends on a sequence of events or states.

FSMs are a crucial topic in digital design because they integrate sequential logic (flip-flops, latches) with combinational logic.

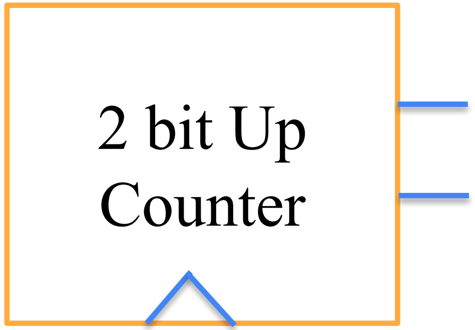
Specification

- Combinational Logic
  - Truth Table
  - Boolean Expression
  - Logic Diagram (No feedback loops)

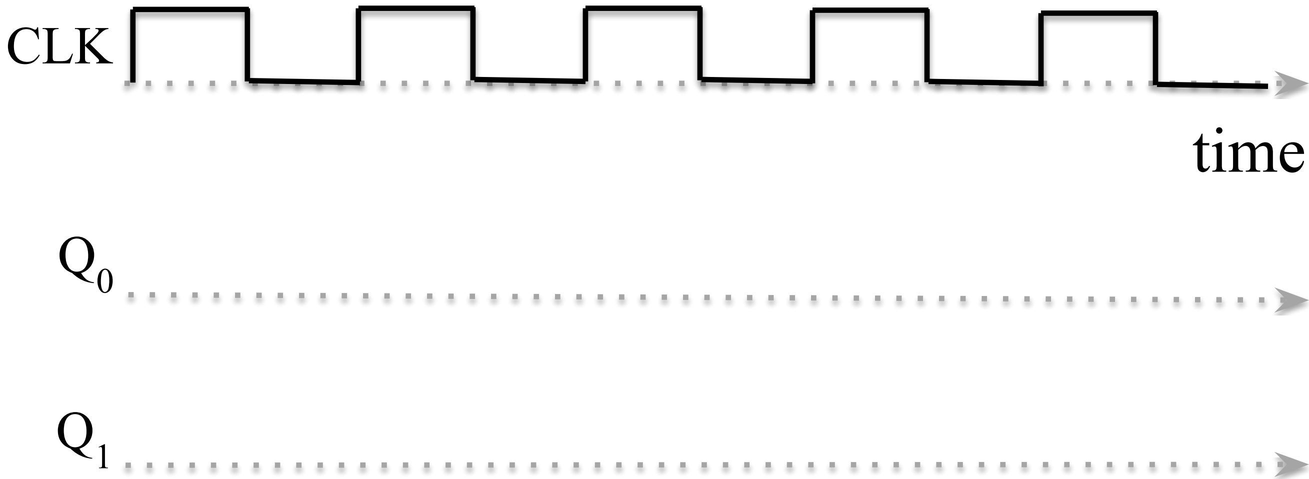
- Sequential Networks:
  - State Diagram, State Assignment, State Table
  - Excitation Table and Characteristic Expression
  - Logic Diagram (FFs and feedback loops)

State: What is it? Why do we need it?

Symbol/ Circuit



Behavior over time



What is the expected output of the counter over time?

Finite State Machines: Describing circuit behavior over time

Symbol/ Circuit

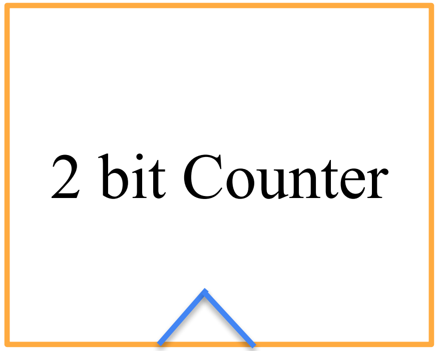
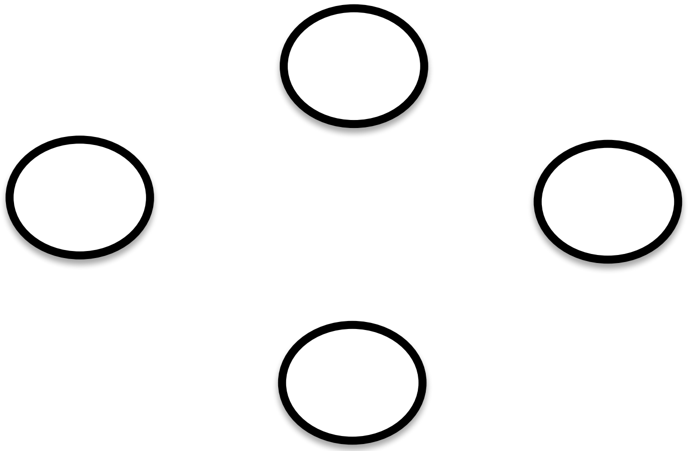
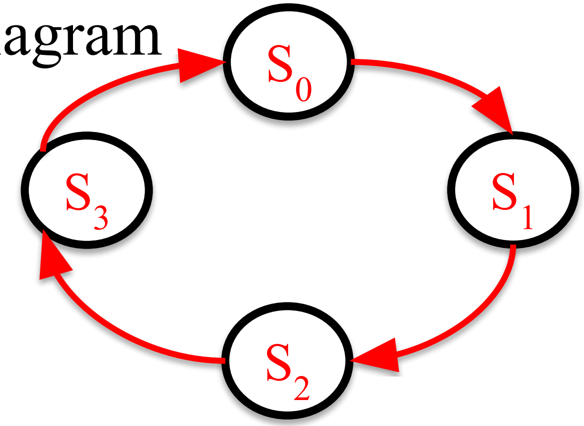


Diagram that depicts behavior over time



Implementing the 2 bit counter

State Diagram



State Table: Symbol

Current state	Next State
S_0	S_1
S_1	S_2
S_2	S_3
S_3	S_0

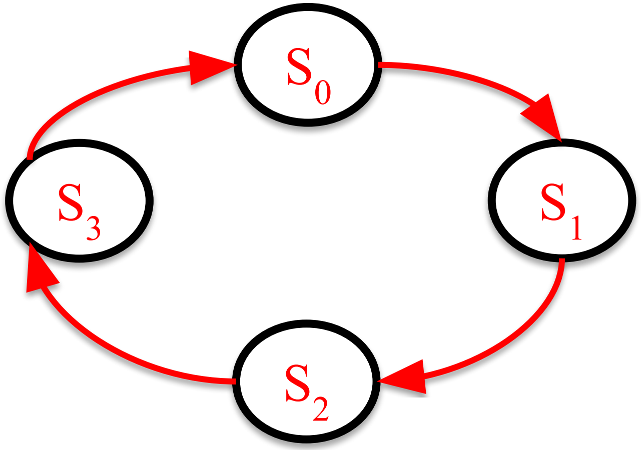
State Assignment

State	Q_1	Q_0
S_0	0	0
S_1	0	1
S_2	1	0
S_3	1	1

$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$

State Table: Binary

Implementing the 2 bit counter



State Diagram

State Table: Symbol

Current state	Next State
S_0	S_1
S_1	S_2
S_2	S_3
S_3	S_0

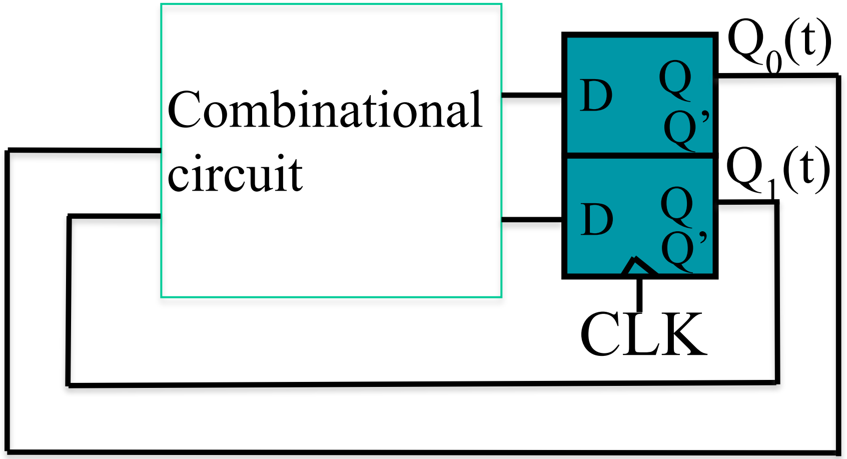
$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

State Table

State Table

$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

$$D_0(t) = Q_0(t)'$$
$$D_1(t) = Q_0(t)\, Q_1(t)' + Q_0(t)'\, Q_1(t)$$

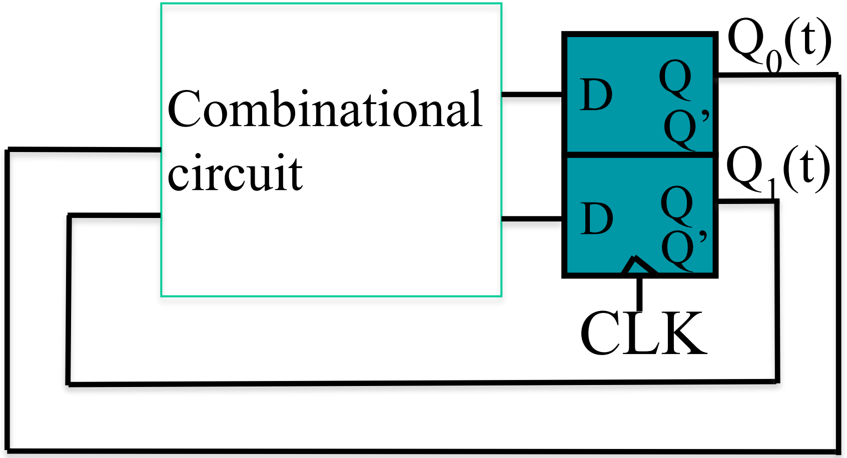


Circuit with 2 flip flops

State Table

$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

$$D_0(t) = Q_0(t)'$$
$$D_1(t) = Q_0(t)\,Q_1(t)' + Q_0(t)'\,Q_1(t)$$



Circuit with 2 flip flops

To implement a 2-bit up counter, you need **2 flip-flops**.

Explanation:

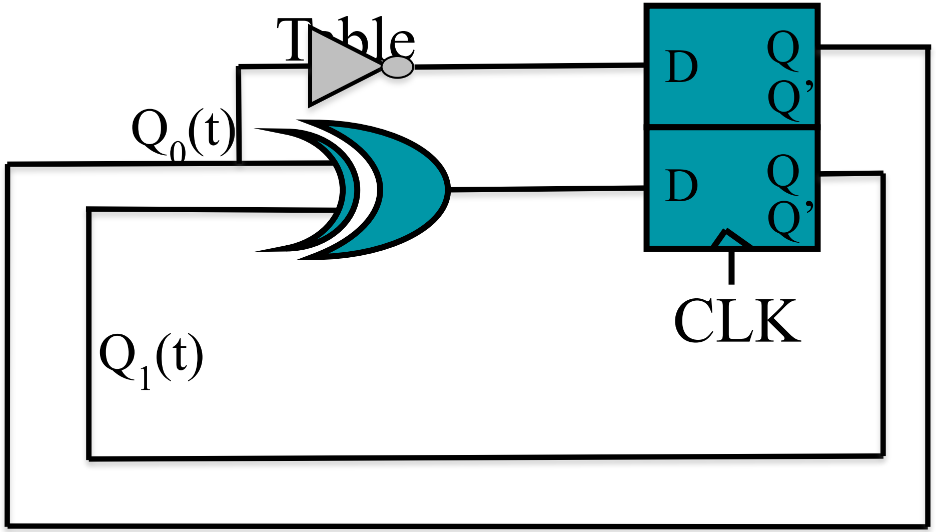
- A flip-flop stores one bit of information.
- A 2-bit counter requires 2 bits to represent its state: $00, 01, 10$, and 11.
- Thus, 2 flip-flops are sufficient to store and transition between these 4 states.

$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

State Table

Truth table→K map→Switching function

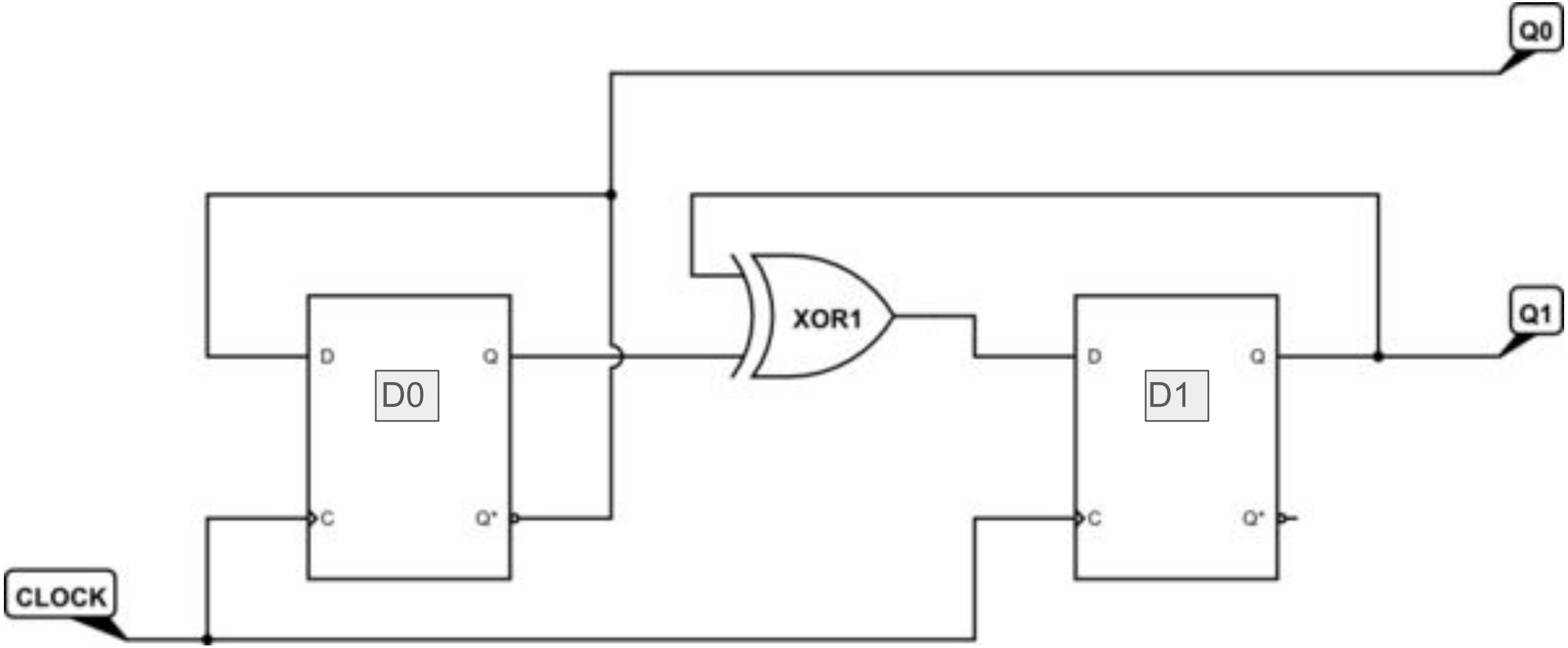
$Q_0(t+1) = Q_0(t)'$
$Q_1(t+1) = Q_0(t)\, Q_1(t)' + Q_0(t)'\, Q_1(t)$



Implementation of 2-bit counter

We store the current state using D-flip flops so that:

- •Inputs to the combinational circuit don't change while the next output is being computed
- •The transition to the next state only occurs at the rising edge of the clock


Q0
Q1
XOR1
D
Q
D0
C
Q*
D
Q
D1
C
Q*
CLOCK

What are FSMs?

An FSM is a **model of a system** with:

- **States**: Distinct conditions or situations the system can be in.
- **Transitions**: Rules that govern how the system moves from one state to another.
- **Inputs**: Signals or data that affect state transitions.
- **Outputs**: Signals or data produced based on states (and sometimes inputs).
- **Reset Mechanism**

Types of FSMs

FSMs are two types: **Moore Machine** and **Mealy Machine**

a) Moore Machine

- The **output depends only on the current state**.
- Outputs change only when the state changes.
- Simpler to design but sometimes less flexible.

Example: Traffic Light Controller

- States: `Red`, `Green`, `Yellow`.
- Outputs: Lights (Red = ON, Green = OFF, Yellow = OFF, etc.).

Types of FSMs

FSMs are two types: **Moore Machine** and **Mealy Machine**

b) Mealy Machine

- The **output depends on both the current state and inputs**.
- Outputs can change during state transitions.
- More compact in implementation but potentially more complex.

Example: Door Lock Controller

- States: `Locked`, `Unlocked`.
- Inputs: Correct/Incorrect Password.
- Outputs: Door state and buzzer signal.

FSM Components

An FSM design can be broken into the following key elements:

- **State Diagram**
  - A graphical representation of states and transitions.
  - Nodes represent states; arrows represent transitions.
  - Conditions for transitions are labeled on the arrows.
- **State Table**
  - A tabular form listing states, inputs, next states, and outputs.
  - Useful for systematic FSM implementation.

FSM Components

An FSM design can be broken into the following key elements:

- **Transition Logic**
  - The combinational logic that dictates how states change based on inputs.
- **Flip-Flops or Memory Elements**
  - Used to store the current state.

FSM Design Process

Understand the Problem:

- Clearly define the states, inputs, and desired outputs.

Draw the State Diagram:

- Identify all states and draw transitions between them based on input conditions.

Create the State Table:

- For each state and input combination, define the next state and output.

FSM Design Process

Encode States:

- Assign binary codes to each state (e.g., 00, 01, 10...).
- The number of flip-flops required depends on the number of states.

Implement Transition Logic:

- Write Boolean expressions or use a tool like SystemVerilog for state transitions and outputs.

Test the FSM:

- Simulate the FSM to ensure it meets the design requirements.

Example Scenario: Card Reader Security System

Imagine designing a security system for an ATM card reader.

The card reader scans a series of binary bits (representing the magnetic stripe data) and needs to detect a specific security code pattern, `101`.

This pattern serves as a marker for validating the card's data integrity.

If the system detects the `101` sequence in the input stream, it flags the card as valid and triggers further processing.

Example Scenario: Card Reader Security System

FSM Operation in Context

- When a card is inserted, the ATM starts reading its magnetic stripe.
- The FSM processes the binary data bit by bit.
- If the sequence `101` is detected, the system flags the card as valid (`detected = 1`).
- If an incorrect sequence is encountered, the FSM resets to scan for a new valid sequence.

Example Scenario: Card Reader Security System

System Requirements

- **Input**
- **Output**
- **States**
- **Transitions**
- **Reset Mechanism**

Example Scenario: Card Reader Security System

System Requirements

- **Input**: A continuous stream of binary data (`serial_in`) from the card's magnetic stripe.
- **Output**
- **States**
- **Transitions**
- **Reset Mechanism**

Example Scenario: Card Reader Security System

System Requirements

- **Input**: A continuous stream of binary data (`serial_in`) from the card's magnetic stripe.
- **Output:** A signal (`detected`) that goes high when the sequence `101` is detected.
- **States**
- **Transitions**
- **Reset Mechanism**

Example Scenario: Card Reader Security System

System Requirements

- **Input**: A continuous stream of binary data (`serial_in`) from the card's magnetic stripe.
- **Output:** A signal (`detected`) that goes high when the sequence `101` is detected.

States / Transitions:

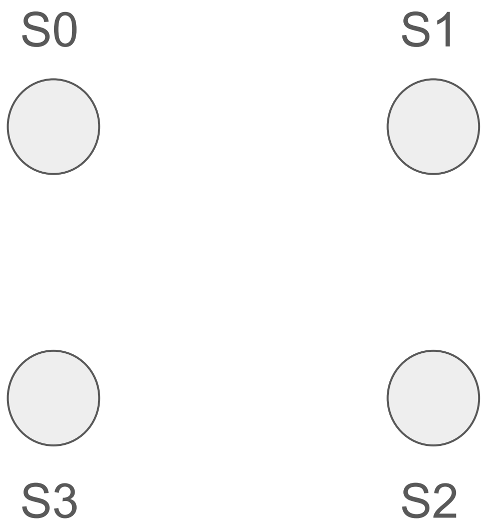
- `S0` (Idle): No part of the sequence matched.
- `S1`: First `1` detected.
- `S2`: `10` detected.
- `S3`: Sequence `101` detected (output = 1).

Example Scenario: Card Reader Security System

System Requirements

States / Transitions:

- `S0` (Idle): No part of the sequence matched.
- `S1`: First `1` detected.
- `S2`: `10` detected.
- `S3`: Sequence `101` detected (output = 1).



Example Scenario: Card Reader Security System

System Requirements

States / Transitions:

- `S0` (Idle): No part of the sequence matched.
- `S1`: First `1` detected.
- `S2`: `10` detected.
- `S3`: Sequence `101` detected (output = 1).

Reset Mechanism: If the sequence is not detected, the FSM resets and scans for a new sequence.

Example Scenario: Card Reader Security System

Enhanced Real-World Features

In a practical implementation, this FSM could be extended with:

1. **Timeout Feature**: If the sequence isn't detected within a specific time frame, reject the card.
2. **Error Detection**: If an invalid pattern is repeatedly encountered, raise a security alert.
3. **Sequence Variations**: Expand the FSM to detect other patterns (e.g., `1101` for different card types).

Example Scenario: Card Reader Security System

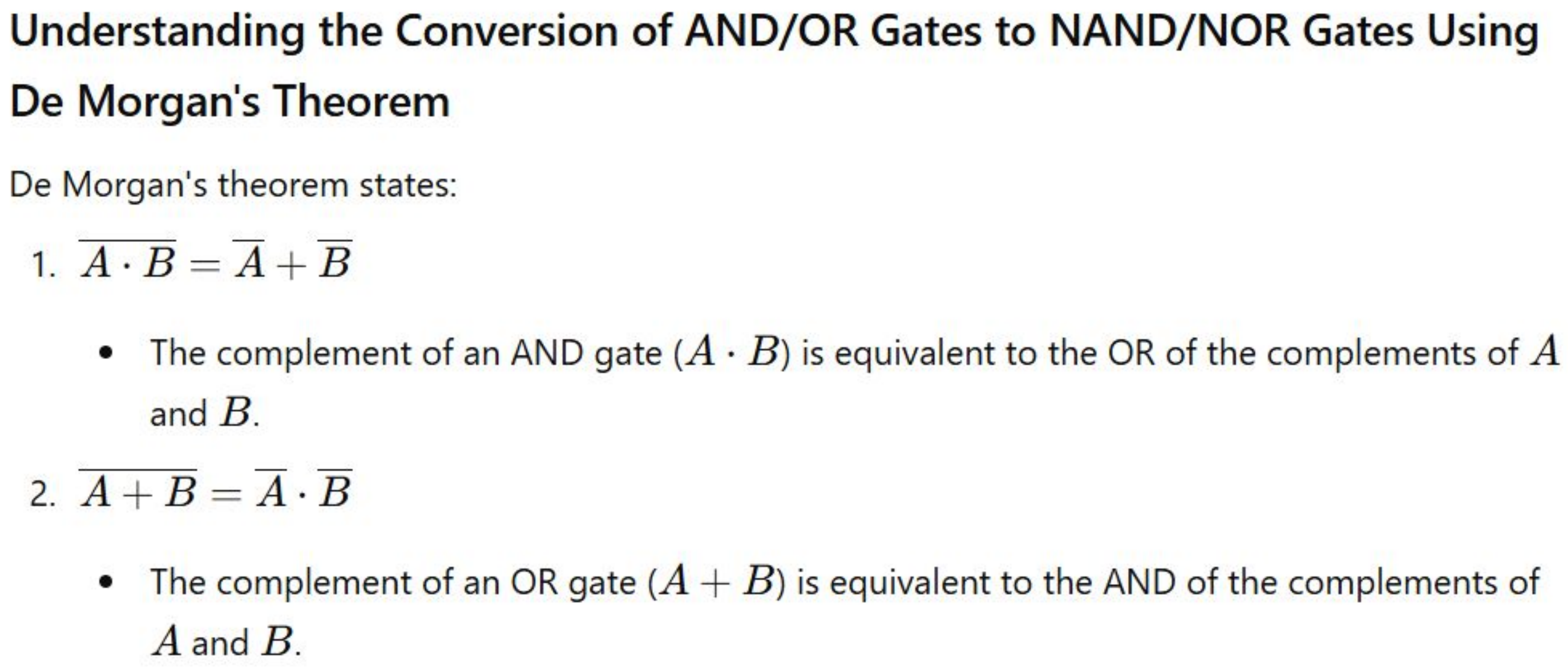
Other Real-World Applications of Sequence Detector FSMs

- **Communication Protocols**:
  - Detect start/stop bit sequences in UART, I2C, or SPI communication.
- **Pattern Recognition**:
  - Use in barcode scanners to validate specific patterns.
- **Security Systems**:
  - Detect unauthorized access codes in door lock systems.

Break!

Next; Recap with Questions

Universal Gates

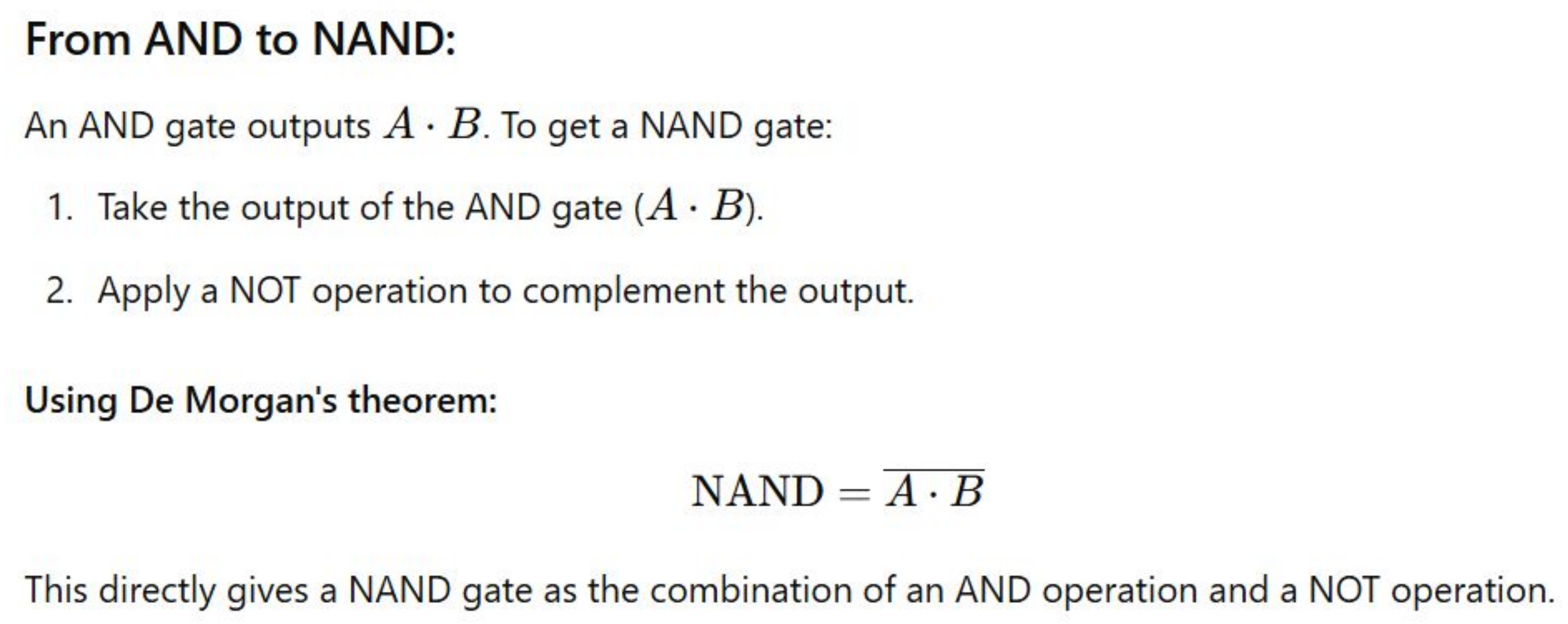


Understanding the Conversion of AND/OR Gates to NAND/NOR Gates Using De Morgan's Theorem

De Morgan's theorem states:

1. $\overline{A \cdot B} = \overline{A} + \overline{B}$
   - The complement of an AND gate ($A \cdot B$) is equivalent to the OR of the complements of A and B.
2. $\overline{A + B} = \overline{A} \cdot \overline{B}$
   - The complement of an OR gate ($A + B$) is equivalent to the AND of the complements of A and B.

Universal Gates



From AND to NAND:

An AND gate outputs $A \cdot B$. To get a NAND gate:

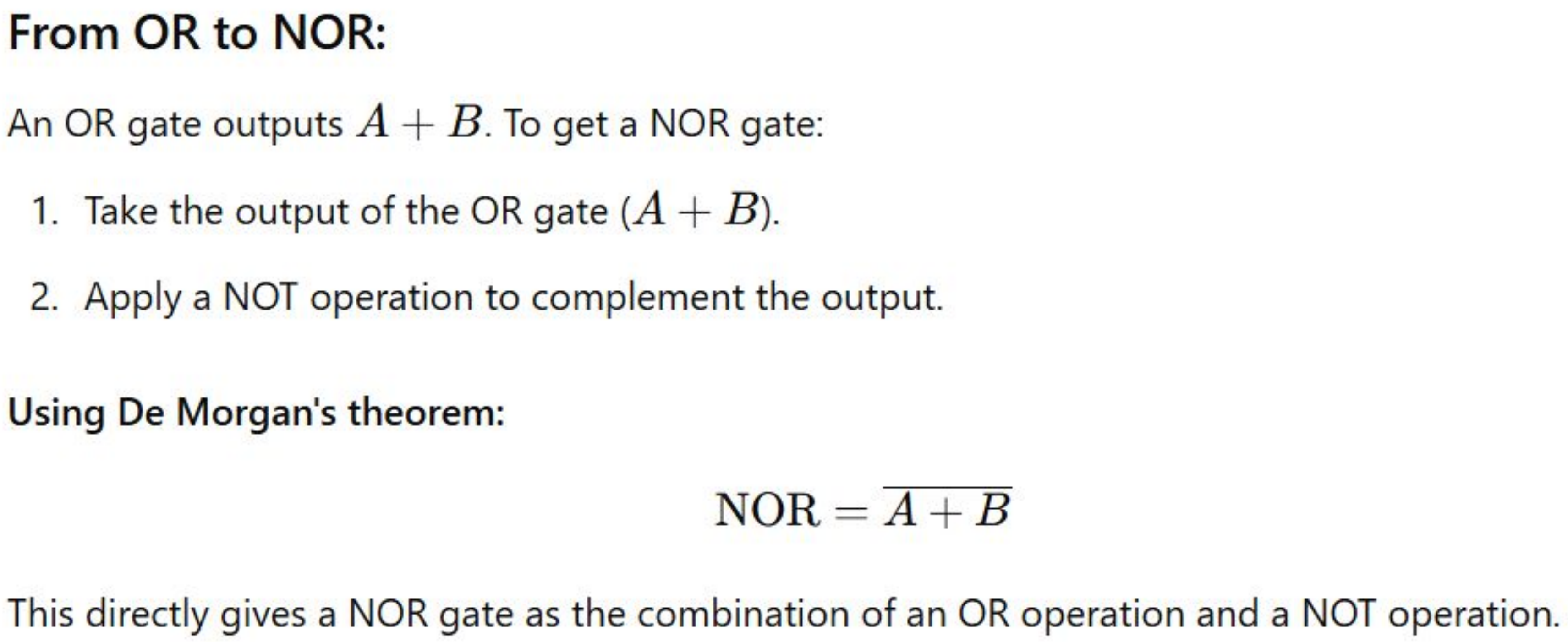
1. Take the output of the AND gate ($A \cdot B$).
2. Apply a NOT operation to complement the output.

Using De Morgan's theorem:

$$\text{NAND} = \overline{A \cdot B}$$

This directly gives a NAND gate as the combination of an AND operation and a NOT operation.

Universal Gates



From OR to NOR:

An OR gate outputs $A + B$. To get a NOR gate:

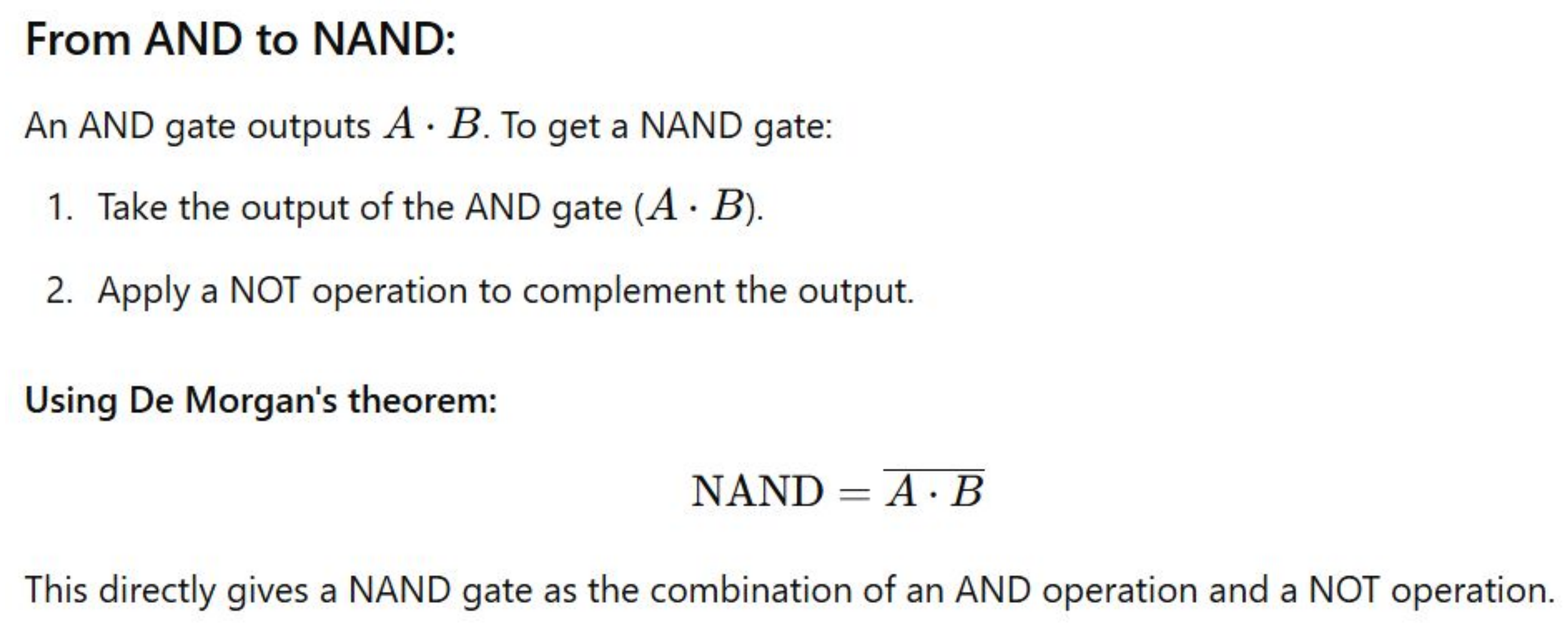
1. Take the output of the OR gate ($A + B$).
2. Apply a NOT operation to complement the output.

Using De Morgan's theorem:

$$\text{NOR} = \overline{A + B}$$

This directly gives a NOR gate as the combination of an OR operation and a NOT operation.

Universal Gates



From AND to NAND:

An AND gate outputs $A \cdot B$. To get a NAND gate:

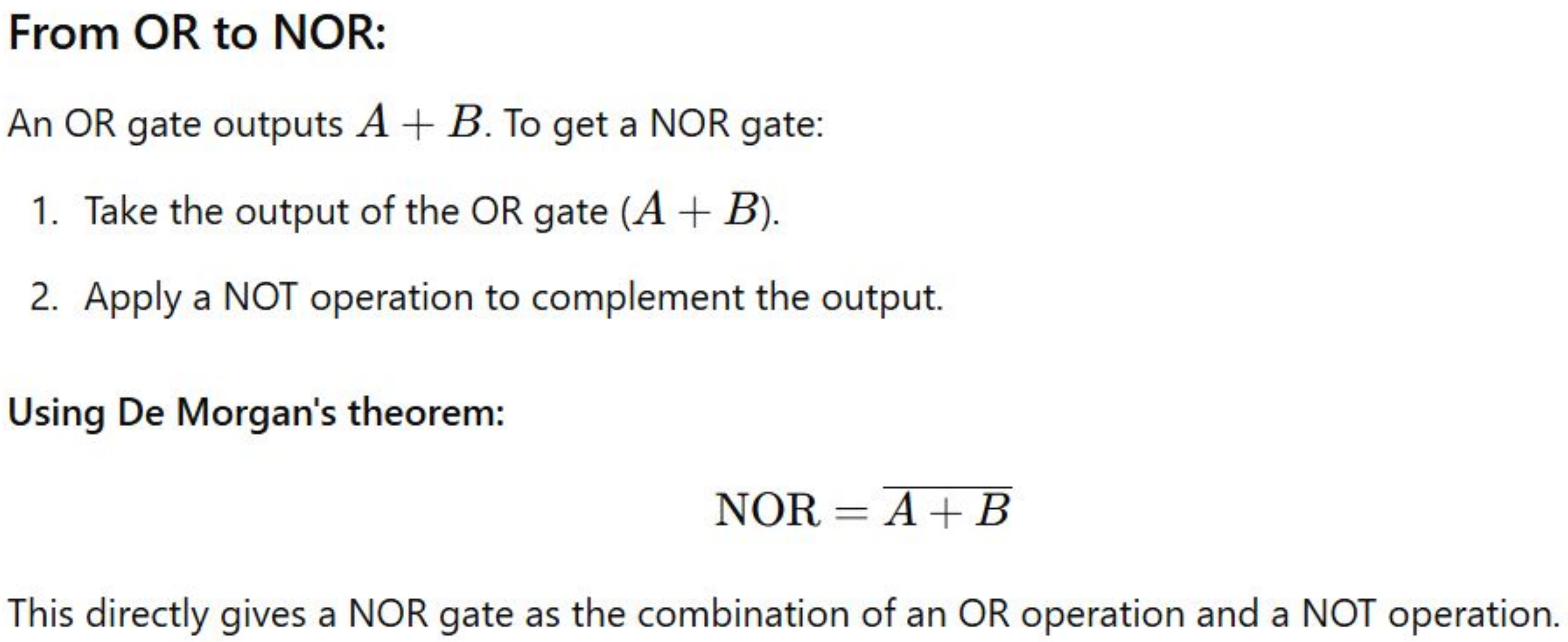
1. Take the output of the AND gate ($A \cdot B$).
2. Apply a NOT operation to complement the output.

Using De Morgan's theorem:

$$\text{NAND} = \overline{A \cdot B}$$

This directly gives a NAND gate as the combination of an AND operation and a NOT operation.

Universal Gates



From OR to NOR:

An OR gate outputs $A + B$. To get a NOR gate:

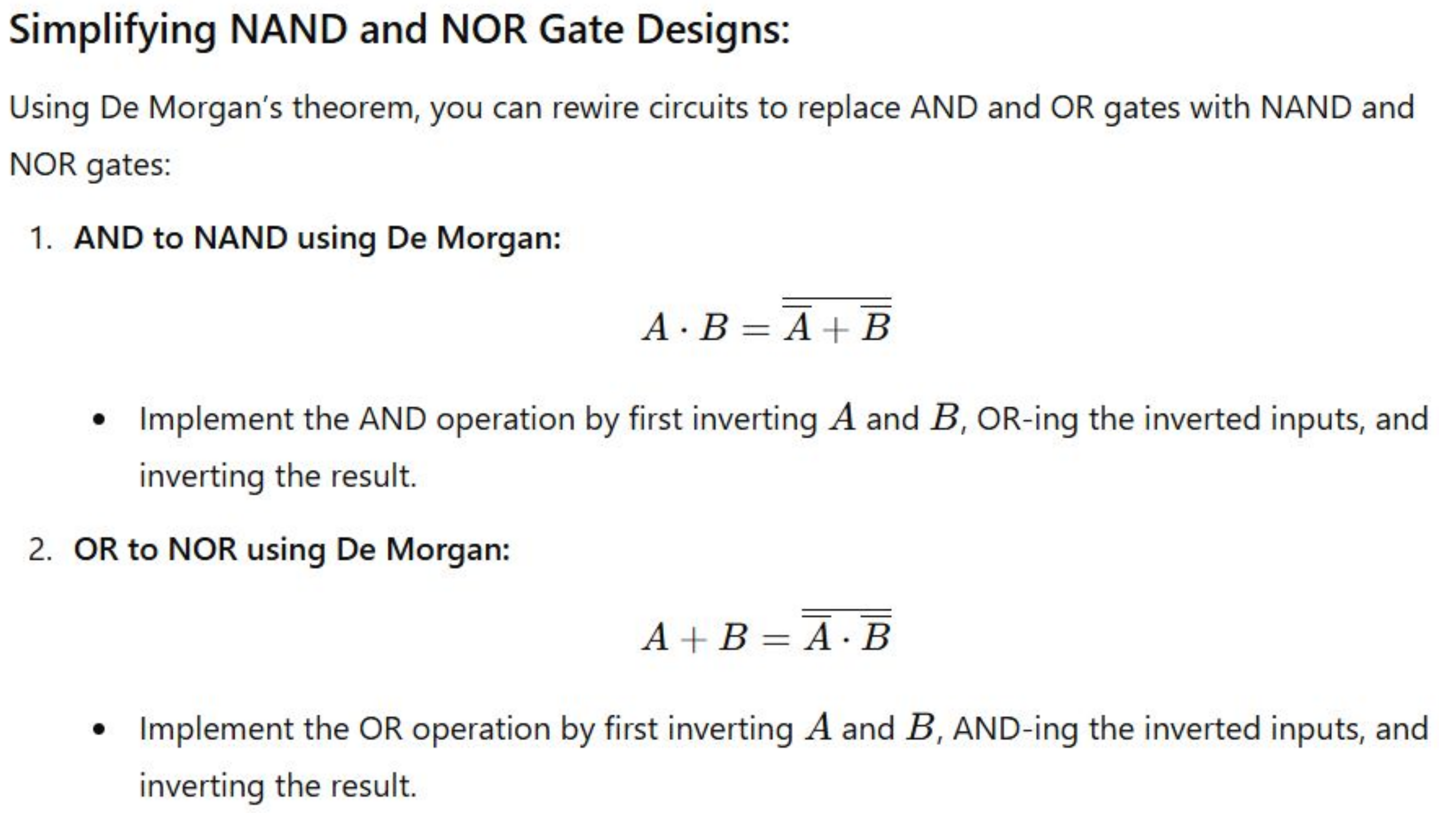
1. Take the output of the OR gate ($A + B$).
2. Apply a NOT operation to complement the output.

Using De Morgan's theorem:

$$\text{NOR} = \overline{A + B}$$

This directly gives a NOR gate as the combination of an OR operation and a NOT operation.

Universal Gates



Simplifying NAND and NOR Gate Designs:

Using De Morgan's theorem, you can rewire circuits to replace AND and OR gates with NAND and NOR gates:

1. **AND to NAND using De Morgan:**

$$A \cdot B = \overline{\overline{A} + \overline{B}}$$

   - Implement the AND operation by first inverting A and B, OR-ing the inverted inputs, and inverting the result.

2. **OR to NOR using De Morgan:**

$$A + B = \overline{\overline{A} \cdot \overline{B}}$$

   - Implement the OR operation by first inverting A and B, AND-ing the inverted inputs, and inverting the result.

Ex1: Design a line follower robots using gates

A line follower is a robot that can follow a track of black lines on a white background.

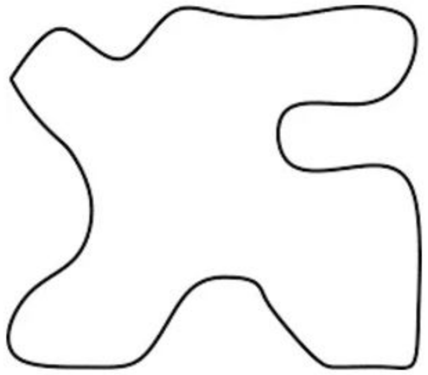


Figure 1. Track of black lines

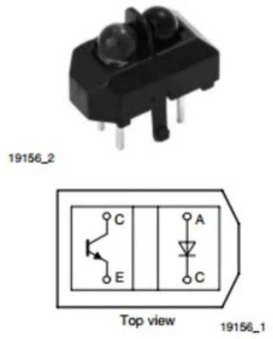
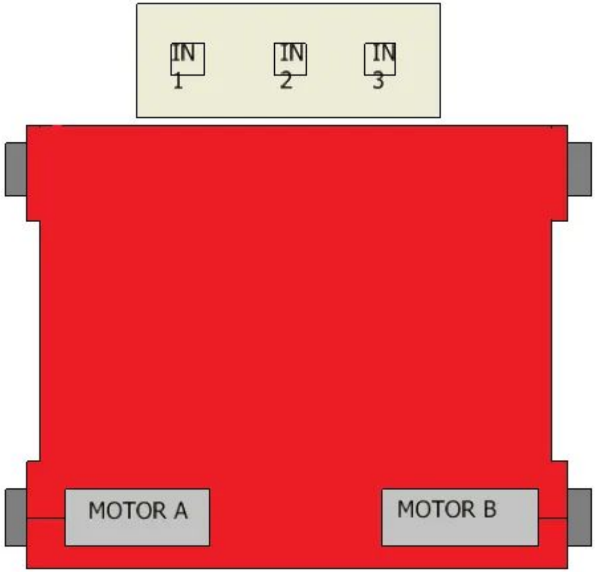


Figure 2. The schematic of the TCRT5000

Design a line follower robots using gates

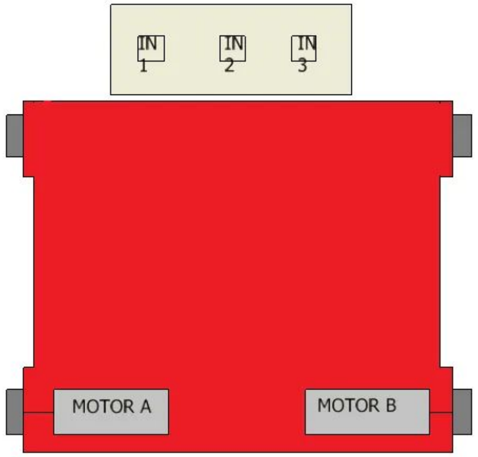
A line follower is a robot that can follow a track of black lines on a white background.



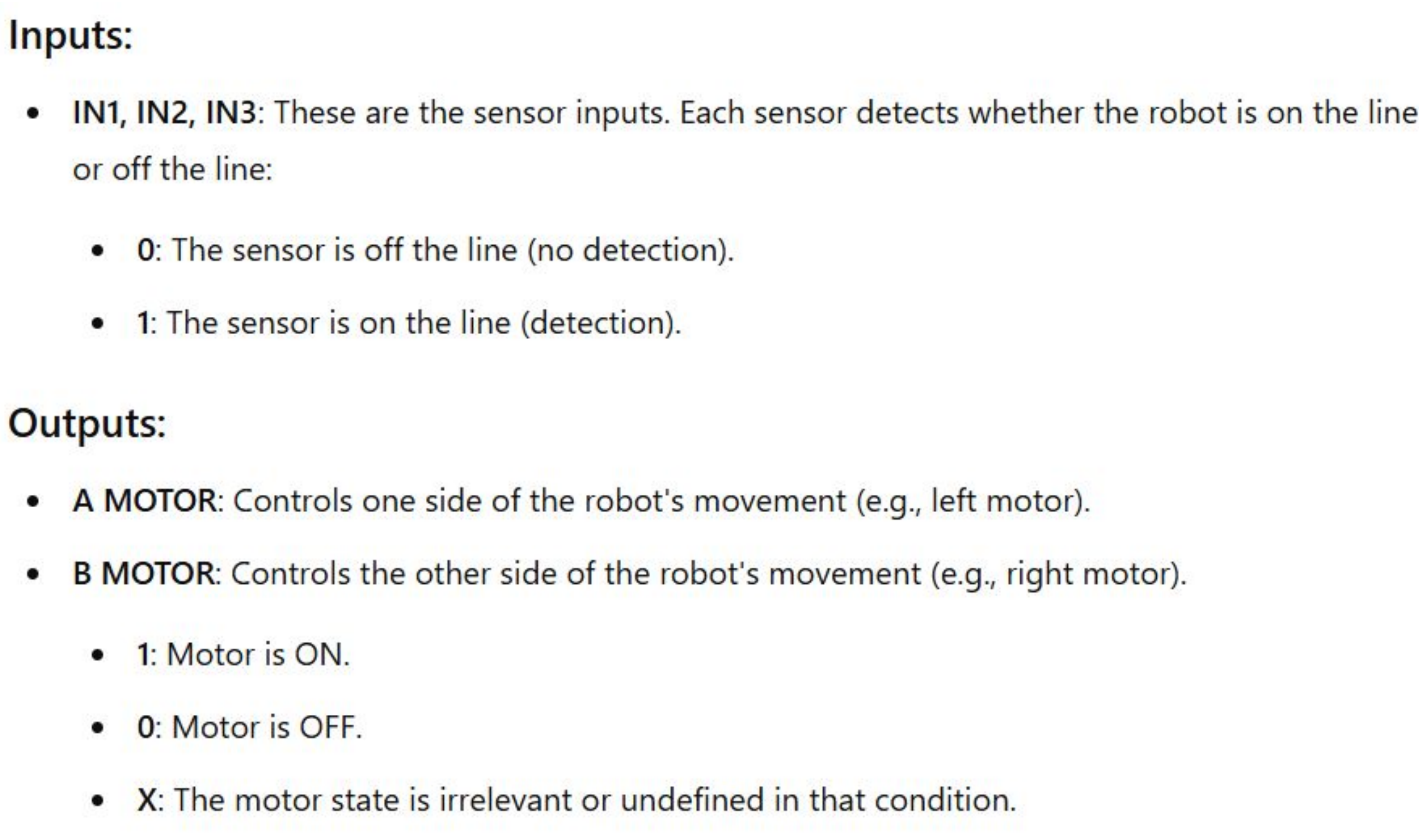
Design a line follower robots using gates

Motor at 1 means that the motor is spinning, and at 0 means that it is stopped.

In order to keep a simple design, we will only drive two motors.



Design a line follower robots using gates



Inputs:

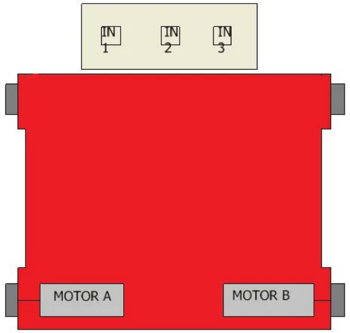
- **IN1, IN2, IN3**: These are the sensor inputs. Each sensor detects whether the robot is on the line or off the line:
  - **0**: The sensor is off the line (no detection).
  - **1**: The sensor is on the line (detection).

Outputs:

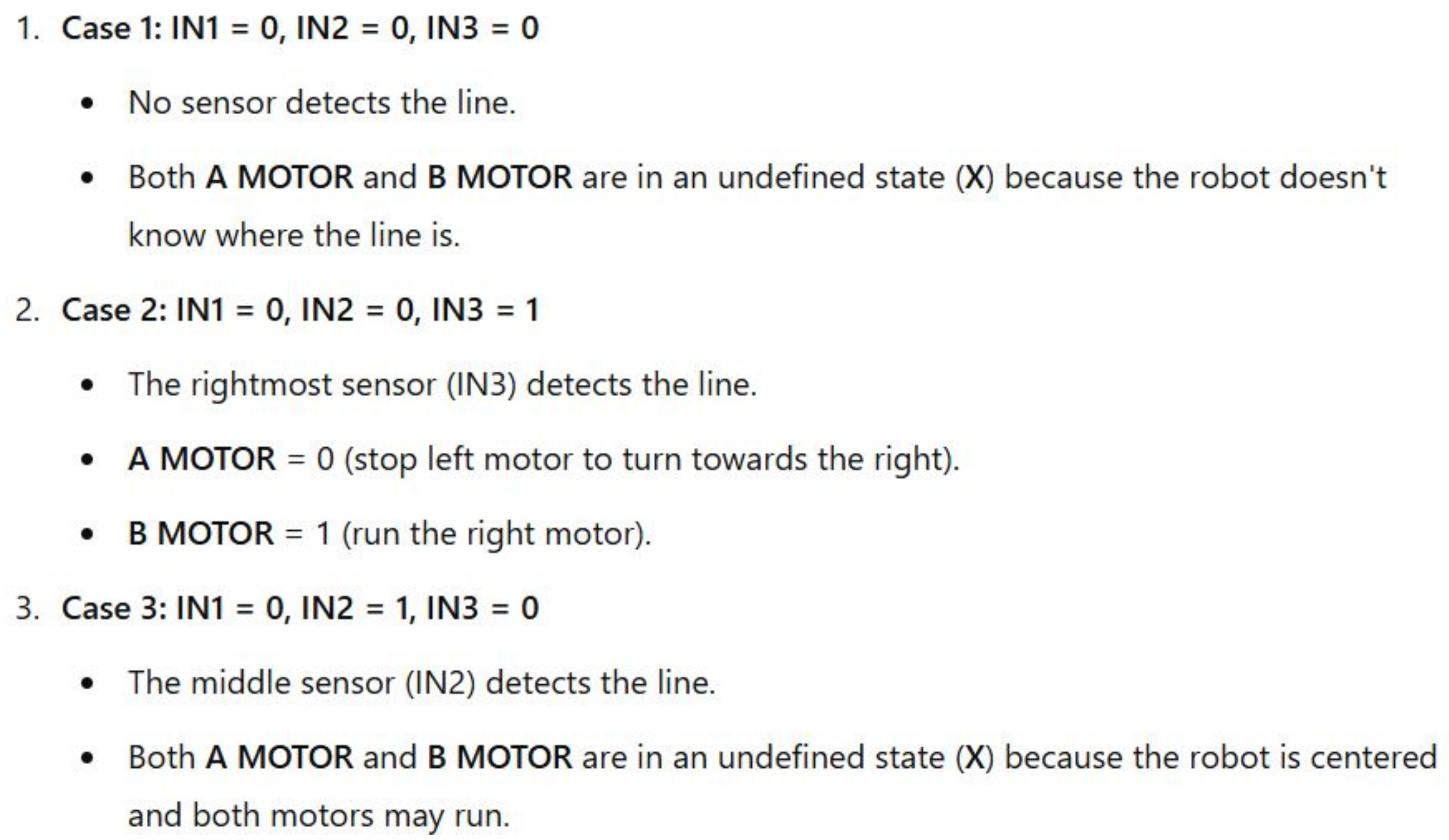
- **A MOTOR**: Controls one side of the robot's movement (e.g., left motor).
- **B MOTOR**: Controls the other side of the robot's movement (e.g., right motor).
  - **1**: Motor is ON.
  - **0**: Motor is OFF.
  - **X**: The motor state is irrelevant or undefined in that condition.

Design a line follower robots using gates

IN 1	IN 2	IN 3	A MOTOR	B MOTOR
0	0	0	X	X
0	0	1	0	1
0	1	0	X	X
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	X	X



Design a line follower robots using gates

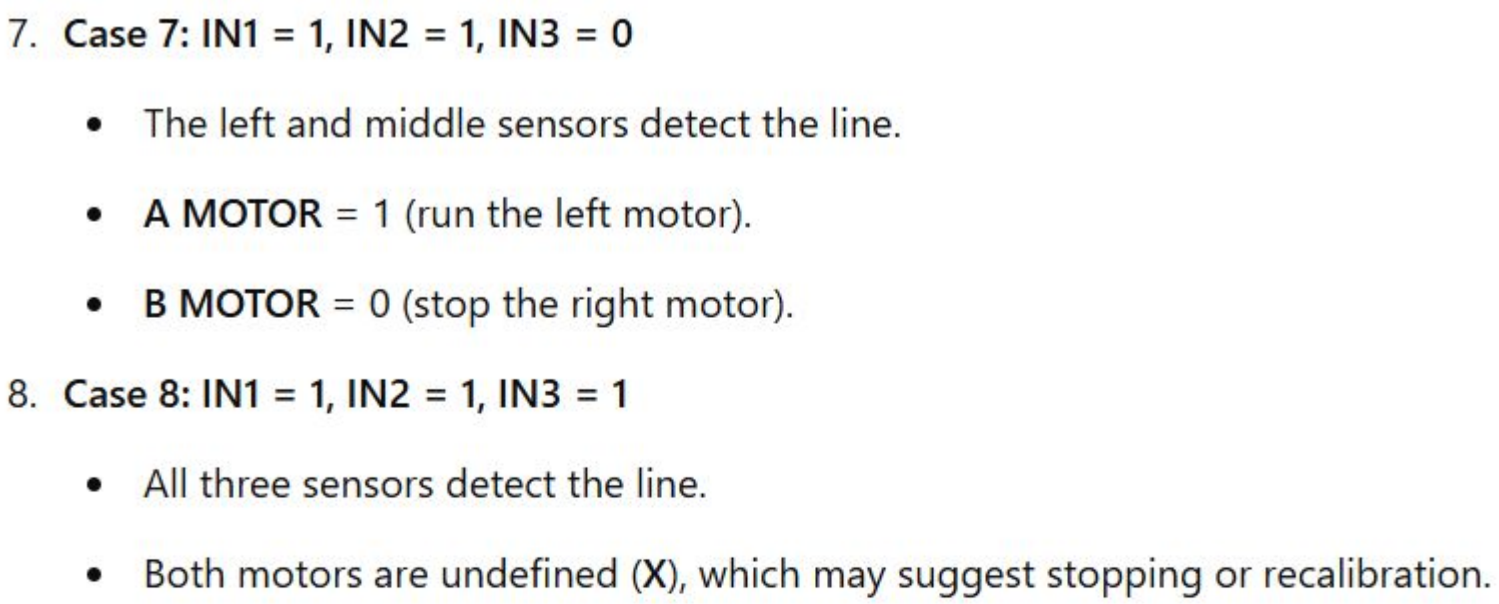


1. **Case 1: IN1 = 0, IN2 = 0, IN3 = 0**
   - No sensor detects the line.
   - Both **A MOTOR** and **B MOTOR** are in an undefined state (**X**) because the robot doesn't know where the line is.
2. **Case 2: IN1 = 0, IN2 = 0, IN3 = 1**
   - The rightmost sensor (IN3) detects the line.
   - **A MOTOR** = 0 (stop left motor to turn towards the right).
   - **B MOTOR** = 1 (run the right motor).
3. **Case 3: IN1 = 0, IN2 = 1, IN3 = 0**
   - The middle sensor (IN2) detects the line.
   - Both **A MOTOR** and **B MOTOR** are in an undefined state (**X**) because the robot is centered and both motors may run.

Design a line follower robots using gates

4. **Case 4: IN1 = 0, IN2 = 1, IN3 = 1**
   - The middle and right sensors detect the line.
   - **A MOTOR** = 0 (stop left motor).
   - **B MOTOR** = 1 (run right motor).
5. **Case 5: IN1 = 1, IN2 = 0, IN3 = 0**
   - The leftmost sensor (IN1) detects the line.
   - **A MOTOR** = 1 (run the left motor to turn towards the left).
   - **B MOTOR** = 0 (stop the right motor).
6. **Case 6: IN1 = 1, IN2 = 0, IN3 = 1**
   - The left and right sensors detect the line.
   - Both motors are undefined (**X**), possibly indicating a need for recalibration.

Design a line follower robots using gates



7. **Case 7: IN1 = 1, IN2 = 1, IN3 = 0**
   - The left and middle sensors detect the line.
   - **A MOTOR** = 1 (run the left motor).
   - **B MOTOR** = 0 (stop the right motor).
8. **Case 8: IN1 = 1, IN2 = 1, IN3 = 1**
   - All three sensors detect the line.
   - Both motors are undefined (**X**), which may suggest stopping or recalibration.

Design a line follower robots using gates

Motor A

Row IN 1 Columns IN2, IN3	00	01	11	10
0	X	0	0	X
1	1	1	X	1

$AMotor = IN1$

Design a line follower robots using gates

Motor B

Row IN 1 Columns IN2, IN3	00	01	11	10
0	X	1	1	X
1	0	1	0	0

$BMotor = IN\,1' + IN\,2' IN\,3$

Ex2: Rover Team

An aircraft belonging to Cukurova University will go to the rover competition organised in the USA. Whether the aircraft will take off or not and the time of departure depends on the rover team finishing the their work pieces.
For example, if 3 sub-teams finish their work on time, the aircraft takes off with 2 units waiting. Design the circuit that generates the standby and take-off outputs of this aircraft.

Number of SubTeam	Delay/Standby	Take off
0	0	0
1-2	3	1
3	2	1
4	0	1

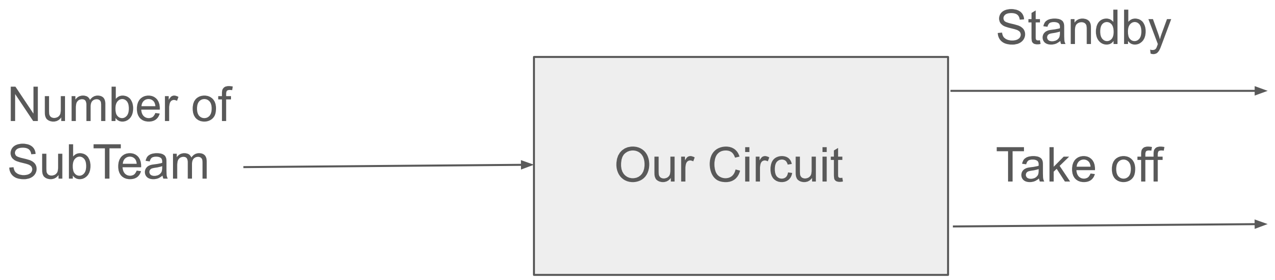
Ex2: Rover Team

- If there are 4 sub-teams in total, how many bits are used to represent these 4 teams?
- How many bits are used to represent standby?
- How many bits are used to represent take off?

Number of SubTeam	Delay/Standby	Take off
0	0	0
1-2	3	1
3	2	1
4	0	1

Ex2: Rover Team

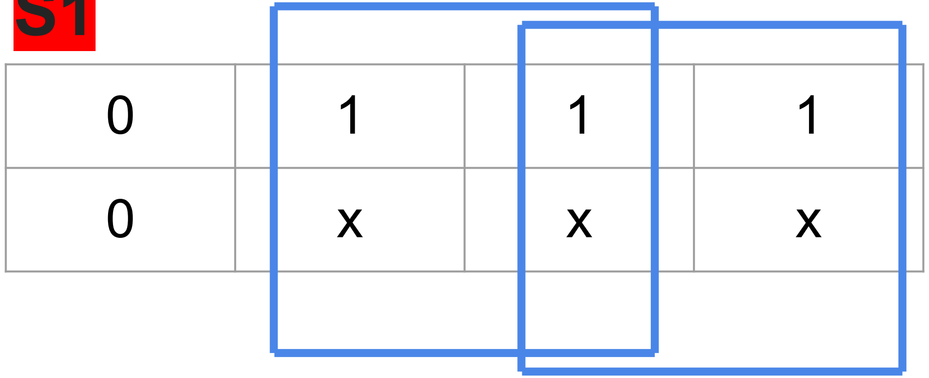
- Subteam = 3 bits
- Standby = 2 bits
- Take off = 1 bit



Ex2: Rover Team

Inputs	outputs		
a b c	S1	S0	T
0 0 0	0	0	0
0 0 1	1	1	1
0 1 0	1	1	1
0 1 1	1	0	1
1 0 0	0	1	1
1 0 1	x	x	x
1 1 0	x	x	x
1 1 1	x	x	x

S1



0	1	1	1
0	x	x	x

S1 = B + C

Ex2: Rover Team

Inputs	outputs		
a b c	S1	S0	T
0 0 0	0	0	0
0 0 1	1	1	1
0 1 0	1	1	1
0 1 1	1	0	1
1 0 0	0	1	1
1 0 1	x	x	x
1 1 0	x	x	x
1 1 1	x	x	x

S0

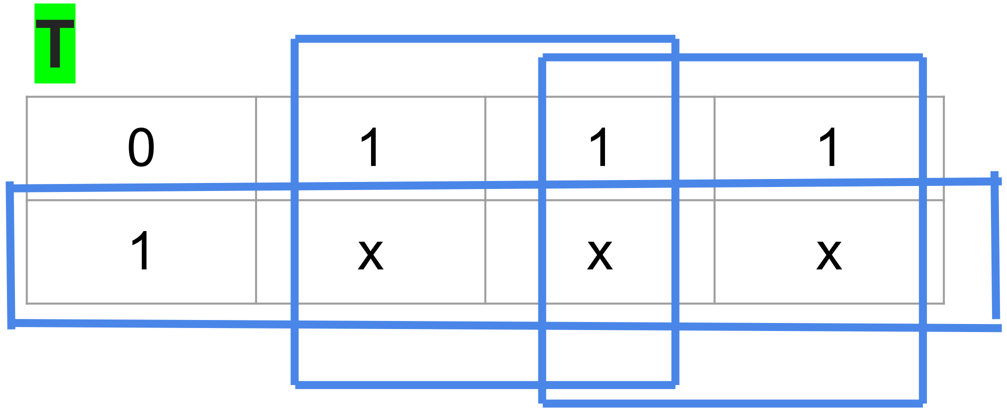
0	1	0	1
1	x	x	x

S0 = B’C + BC’

Ex2: Rover Team

Inputs	outputs		
a b c	S1	S0	T
0 0 0	0	0	0
0 0 1	1	1	1
0 1 0	1	1	1
0 1 1	1	0	1
1 0 0	0	1	1
1 0 1	x	x	x
1 1 0	x	x	x
1 1 1	x	x	x

T



0	1	1	1
1	x	x	x

T = A + B + C

Ex2: Rover Team

Inputs	outputs		
a b c	S1	S0	T
0 0 0	0	0	0
0 0 1	1	1	1
0 1 0	1	1	1
0 1 1	1	0	1
1 0 0	0	1	1
1 0 1	x	x	x
1 1 0	x	x	x
1 1 1	x	x	x

S1 = B + C

S0 = B’C + BC’

T = A + B + C

Ex3: Rover Team - Advanced

Cukurova University pays compensation for planes that do not take off on time. The compensation is proportional to the amount of standby and weather conditions (0=poor, 1=good).

Compensation = 4(Standby)^2 + 8(weather conditions)

For example, a plane standby for 3 units when the weather is 1 good will pay 4 x 9 + 8 = 44 units cost. Design the circuit that calculates the compensation.

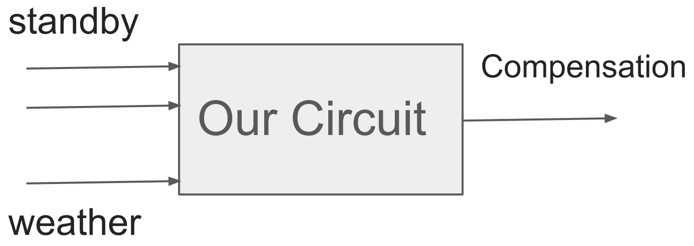
Number of SubTeam	**Delay/Standby**	**Take off**
0	0	0
1-2	3	1
3	2	1
4	0	1

Ex3: Rover Team - Advanced

Cukurova University pays compensation for planes that do not take off on time. The compensation is proportional to the amount of standby and weather conditions (0=poor, 1=good).

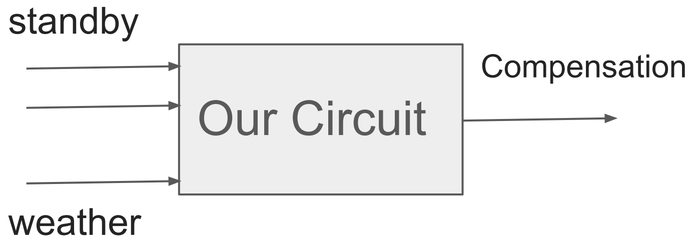
Compensation = 4(Standby)^2 + 8(weather conditions)

For example, a plane standby for 3 units when the weather is 1 good will pay 4 x 9 + 8 = 44 units cost. Design the circuit that calculates the compensation.



Number of SubTeam	**Delay/Standby**	**Take off**
0	0	0
1-2	3	1
3	2	1
4	0	1

Ex3: Rover Team - Advanced



S1	S0	W	C5	C4	C3	C2	C1	C0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0
0	1	0	x	x	x	x	x	x
0	1	1	x	x	x	x	x	x
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	0
1	1	0	1	0	0	1	0	0
1	1	1	1	0	1	1	0	0

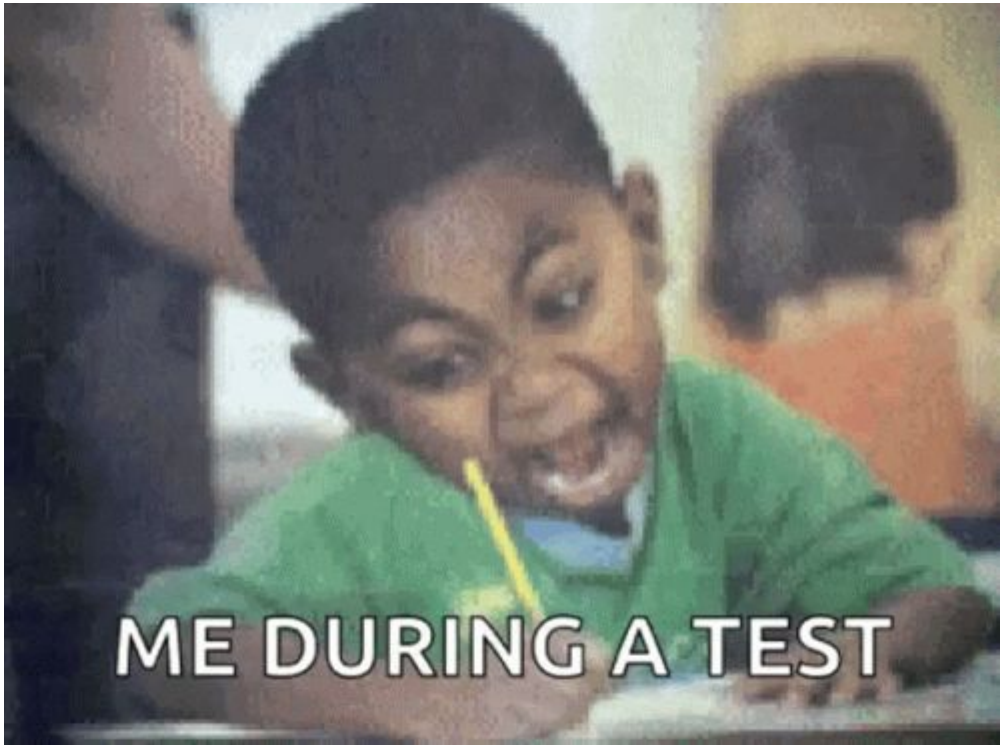
Recap - Textbook

Ch1:Number Systems
Ch1:Logic Gates

Ch2: Boolean Equations
Ch2: Boolean Algebra
Ch2: From Logic to Gates
Ch2: Karnaugh Maps
Ch2: Combinational Building Blocks
Ch2: Timing - Delays

Ch3: Latches and Flip-Flops
Ch3: Synchronous Logic Design
Ch3: Finite State Machines

Ch4: Hardware Description Languages - SystemVerilog



Good luck with your exams!