Group_9(6)_Fundamental_of_digital_logic_with_verilog_design_2_2.1-2.11

1 Introduction

Binary circuits dominate digital systems due to their simplicity. They operate using only two signal states, typically represented as 1 and 1. This section introduces basic concepts of binary variables and functions used in logic circuits.

2 Binary Switches and Variables

A fundamental binary element is a switch that can be either open or closed. If a switch is controlled by an input variable x, it behaves as follows:

- $x = 1$: Switch is open.
- $x = 2$: Switch is closed.
- Logical functions can be implemented using combinations of switches.

Figure 1 illustrates this concept.

A simple application is controlling a lightbulb with a switch. When the switch is closed ($x = 2$), current flows, and the light turns on ($L = 1$). When open ($x = 0$), the light turns off ($L = 0$). This relationship is expressed as:

$$L(x) = x \tag{1}$$

which defines a binary function.

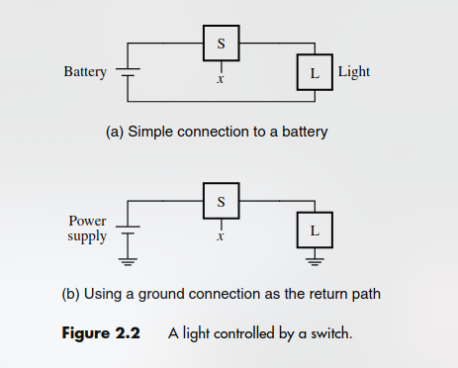


Figure 1: A binary switch representation.

3 Series and Parallel Switch Configurations

When multiple switches control a circuit, they can be connected in series or parallel:

- **Series Connection**: Light turns on only when both switches are closed.

$$L(x_2, x_2) = x_1 \cdot x_2 \tag{2}$$

- **Parallel Connection**: Light turns on when at least one switch is closed.

$$L(x_2, x_2) = x_1 + x_2 \tag{3}$$

- Series connections represent logical AND operations.
- Parallel connections represent logical OR operations.

- Digital circuits use combinations of these configurations to implement complex logic functions.

Figure 2 demonstrates these configurations.

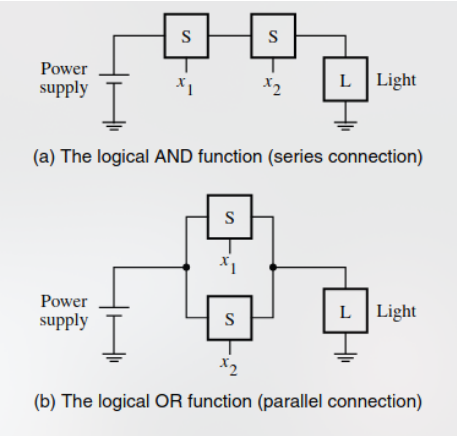
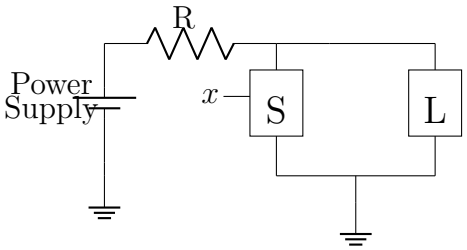


Figure 2: Series and parallel switch connections.

4 Inversion

A switch can be used in an inverting circuit where closing the switch turns off the light, and opening it turns it on. The circuit is designed such that the switch is connected in parallel with the light, meaning that a closed switch short-circuits the light.



The logical behavior of this circuit is expressed as:

$$L(x) = \bar{x} \tag{4}$$

where:

$$L = 1 \text{ if } x = 0,$$
$$L = 0 \text{ if } x = 1.$$

This represents the complement (which is used interchangeably with inverse) operation, commonly denoted as the NOT operation. Different notations for complement include:

$$\bar{x} = \ x' = !x = \sim x. \tag{5}$$

The NOT operation is essential in digital circuits for implementing control logic, where an action should take place when a condition is not met.

The complement operation can also be applied to more complex expressions. For example, if

$$f(x_1, x_2) = x_1 + x_2, \tag{6}$$

then the complement of f is

$$\overline{f}(x_1, x_2) = \overline{x_1 + x_2}. \tag{7}$$

This expression yields the logic value 1 only when $x_1 = x_2 = 0$.

5 Truth Tables

The fundamental logic operations AND, OR, and complement can be represented using truth tables. A truth table lists all possible combinations of input values and their corresponding output values.

For two variables, the AND and OR operations are defined as:

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

For three variables, the truth table expands as follows:

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

These operations can be extended to n variables, where the AND function outputs 1 only if all variables are 1, while the OR function outputs 1 if at least one variable is 1.

6 Logic Gates and Networks

Logic gates are the building blocks of digital circuits. They implement logical functions and are constructed using transistors. Digital circuits consist of interconnected logic gates that process input signals and produce desired outputs. Understanding how to analyze and design these networks is essential for effective circuit design.

6.1 Basic Logic Gates

Below are the diagrams and descriptions of the fundamental logic gates used in digital circuits.

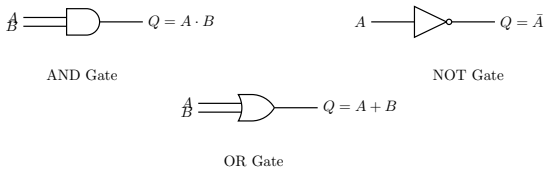


Figure 3: Basic Logic Gates: AND, OR, and NOT

6.2 Analysis of a Logic Network

Analyzing a logic network involves understanding the function performed by a combination of gates. The process includes:

- **Identifying Gates and Connections**: Recognize the type of each logic gate and how they are connected.
- **Tracing Signal Paths**: Follow input signals through each gate to determine their effects.
- **Constructing Truth Tables**: Establish a table that shows all possible input combinations and their corresponding outputs.
- **Verifying Network Behavior**: Ensure that the logic network meets its intended design requirements.

6.3 Example Analysis: Two-Level Logic Network

Consider the following circuit with an AND gate and an OR gate:

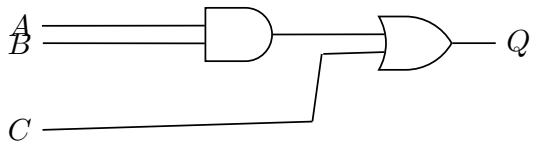


Figure 4: Two-level logic network with AND and OR gates.

Analysis Steps:

- **Step 1: Identify Gates**: The network consists of an AND gate followed by an OR gate.
- **Step 2: Trace Signals**: Input A and B are combined using AND, and the result is combined with input C using OR.
- **Step 3: Construct Truth Table**:

A	B	C	Q
0	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	1
1	1	1	1

Table 1: Truth Table for Two-Level Logic Network

7 Boolean Algebra

Boolean algebra provides a formal framework for designing and analyzing logic circuits. Developed by George Boole in the mid-19th century, it uses binary values (0 and 1) to represent logic levels. Boolean algebra simplifies complex logic expressions and aids in circuit design.

7.1 The Venn Diagram

Venn diagrams visualize the relationships between sets and are used to represent Boolean operations graphically.

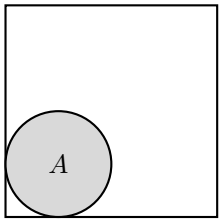


Figure 5: Set representation for a variable in Boolean algebra.

Boolean operations can be represented as intersections, unions, and complements in the Venn diagram.

7.2 Notation and Terminology

Key notations in Boolean algebra:

- **Complement** ($\bar{x}$): Inverts the value of x.
- **AND** ($x \cdot y$): True if both x and y are true.
- **OR** ($x + y$): True if at least one of x or y is true.

Boolean expressions describe logic functions and can be simplified using algebraic rules.

7.3 Precedence of Operations

Operations in Boolean algebra follow a precedence order:

1. **NOT** ($\bar{x}$): Performed first.
2. **AND** ($x \cdot y$): Performed second.
3. **OR** ($x + y$): Performed last.

Parentheses can override this precedence.

7.4 Duality Principle

The duality principle states that every Boolean expression remains valid if we interchange AND and OR operations and swap 0 and 1.

7.5 Boolean Algebra Properties

Important properties include:

- **Commutative Property**: $x + y = y + x$ and $x \cdot y = y \cdot x$
- **Associative Property**: $(x + y) + z = x + (y + z)$
- **Distributive Property**: $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
- **Idempotent Law**: $x + x = x$ and $x \cdot x = x$
- **Identity Law**: $x + 0 = x$ and $x \cdot 1 = x$
- **Complement Law**: $x + \bar{x} = 1$ and $x \cdot \bar{x} = 0$

7.6 Example Simplification

Simplify the expression $x \cdot (x + y)$:

$$\begin{aligned} x \cdot (x + y) &= x \cdot x + x \cdot y \quad \text{(Distributive Property)} \\ &= x + x \cdot y \quad \text{(Idempotent Law)} \\ &= x \quad \text{(Absorption Law)} \end{aligned}$$

This simplification shows that $x \cdot (x+y)$ is equivalent to x.

8 Synthesis Using AND, OR, and NOT Gates

8.1 Introduction

The process whereby we begin with a description of the desired functional behaviour and then generate a cirucit that realizes theis behavious is called *synthesis*. Logic circuit synthesis involves designing a circuit that implements a given Boolean function using fundamental logic gates: AND, OR, and NOT. We can express the required behaviour of the function using truth tables.

8.2 Sum-of-Products and Product-of-Sums

If a function f is specified in the for of a truth table, then an expression that realizes f can be obtained by considering either the rows in the table for which $f = 1$ or rows for which $f = 0$.

Minterms - For a function of n variables, a product term in which each of the n variables appears once is called Minterm.

Sum-of-Products form - A function f can be represented by an expression that is a sum of minterms, where each minterm is ANDed with the value of f for the corresponding valuation of input variables. This form of representation of functions is called the sum-of-products form. If each product term is a minterm, then the expression is called *Canonical sum-of-products.*

After deriving the Canonical form of function, we can manipulate the expression whose corresponding circuit has the lowest cost.

Cost - A good indication of cost of a logical circuit is the total number of gates plus the total number of inputs to all gates in the circuit.

Maxterm - The complements of the minterms are called maxterms.

Product-of-Sums - The expression that represents a function f in the form of a product of terms is in Product-of-Sums form. If the terms are maxterms, then it is called *canonical product-of-sums* form.

8.3 Logic Minimization

Logic expressions obtained directly from truth tables may not always be optimal in terms of gate count or complexity. Boolean algebra and Karnaugh maps are commonly used to simplify expressions and reduce circuit complexity.

9 NAND and NOR Logic Networks

9.1 Introduction

NAND and NOR gates are obtained by complementing the output of AND and OR gates, respectively. NAND and NOR gates are widely used in digital logic design because of their simplicity in implementation and their universality—they can be used to construct any Boolean function.

9.2 NAND as a Universal Gate

A NAND gate performs the operation:

$$\text{NAND}(x_1, x_2) = \overline{x_1 x_2}$$

According to DeMorgan's theorem:

$$\overline{x_1 x_2} = \overline{x_1} + \overline{x_2}$$

This shows that a NAND gate of variables x_1 and x_2 is equivalent to first complementing each of the variables and then ORing them.

9.3 NOR as a Universal Gate

A NOR gate performs:

$$\text{NOR}(x_1, x_2) = \overline{x_1 + x_2}$$

By DeMorgan's theorem:

$$\overline{x_1 + x_2} = \overline{x_1}\,\overline{x_2}$$

This shows that a NOR gate of variables x_1 and x_2 is equivalent to ANDing both inputs and then complementing them.

9.4 Converting Logic Networks

Expressing a function in terms of a sum-of-products or product-of-sums format leads to a logic circuit that has an AND-OR or an OR-AND structure. Any such network can be implemented using only NAND gates or NOR gates, respectively. This can be done by replacing each connection between an AND gate and an OR gate by a connection that includes two inversions of the signal, one inversion at the output of the AND gate and the other at the input of the OR gate. It would have no effect on out put as complement of complement of the given signal is the signal itself. As we invert the output of the AND gate, it becomes a NAND gate and we feed the inverted inputs to the OR gate, which makes it a NAND gate. A similar process is following for NOR gate. This transformation is useful for hardware optimization since NAND and NOR gates are easier to fabricate.

9.5 Design Examples

9.5.1 Three-Way Light Control

- Say there is a large room with 3 lights, with corresponding switches. Let x_1, x_2, x_3 denote the state of each switch.
- The light is OFF when all switches are open. The light is ON if exactly one switch is closed, and OFF if two (or no) switches are closed.
- Let state of the light be represented by the function $f(x_1, x_2, x_3)$.

We shall try to represent$f(x_1, x_2, x_3)$ in sum-of-products form

Sum-of-Products Form

The function f is expressed as:

$$f = m_1 + m_2 + m_4 + m_7 = \overline{x_1} x_2 \overline{x_3} + x_1 \overline{x_2 x_3} + \overline{x_1 x_2} x_3 + x_1 x_2 x_3$$

This expression cannot be simplified further.

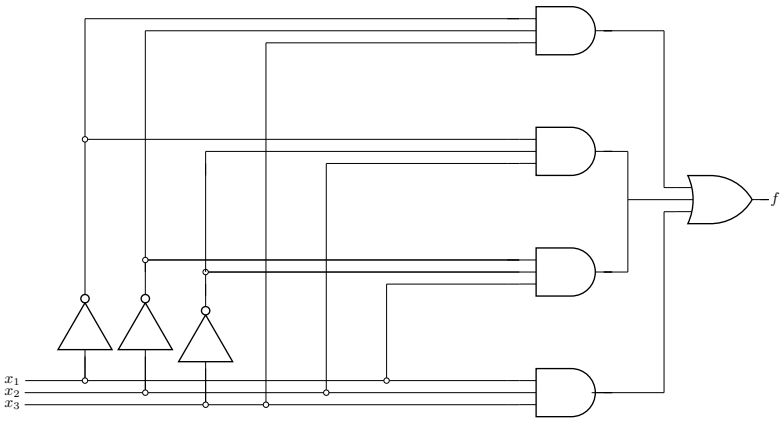


Figure 6: Sum-of-Products

Product-of-Sums Form

The function f can also be expressed as:

$$f = M_0 \cdot M_3 \cdot M_5 \cdot M_6 = (x_1 + x_2 + x_3)(x_1 + \overline{x_2} + \overline{x_3}) \\ (\overline{x_1} + x_2 + \overline{x_3})(\overline{x_1} + \overline{x_2} + x_3)$$

This form has the same cost as the sum-of-products realization.

Truth Table

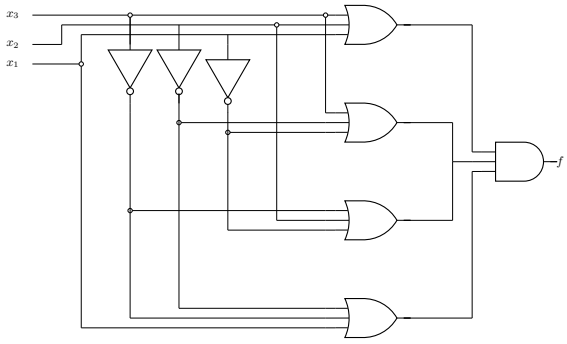


Figure 7: Product-of-Sums

Truth table for the 3-way light control is as follows,

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

9.5.2 Multiplexer Circuit

- In computer systems, it is often necessary to select data from exactly one of several sources.Consider two data sources, represented by input signals x_1 and x_2, which change over time.

- A circuit is needed to produce an output that matches either x_1 or x_2, depending on a selection control signal s.

The circuit has three inputs: x_1, x_2, and s. The output f is defined as:

$$f = \begin{cases} x_1 & \text{if } s = 0, \\ x_2 & \text{if } s = 1. \end{cases}$$

The Sum-of-Products form of the function is,

$$f(s, x_1, x_2) = sx_1x_2 + sx_1\overline{x_2} + \overline{s}x_1x_2 + \overline{sx_1}x_2.$$

Using the distributive property,

$$f = \overline{s}x_1(\overline{x_2} + x_2) + s(\overline{x_1} + x_1)x_2$$

Applying theorems of boolean algebra $x + \overline{(}x) = 1$,

$$f = sx_1 + \overline{s}x_2.$$

- A circuit implementing this function is called a **2-to-1 multiplexer**. It selects one of two data inputs (x_1 or x_2) based on the control signal s.

- Multiplexers are widely used in digital systems, and larger multiplexers (e.g., 4-to-1, 8-to-1) can be built by extending this concept. A 4-to-1 multiplixer would require two control signals, while an 8-to-1 multiplexer would require three.

The truth table for the 2-to-1 multiplexer is as follows:

s	x_1	x_2	$f(s, x_1, x_2)$
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

A 2-to-1 multiplexer can be built using the basic AND, OR, NOT gates

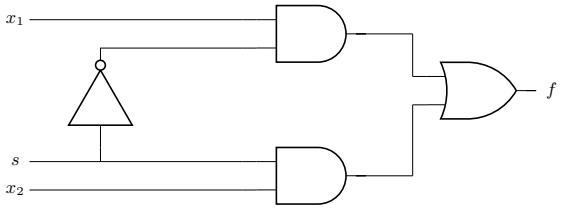


Figure 8: 2-to-1 Multiplexer

9.6 Summary of CAD Tools for Logic Circuit Design

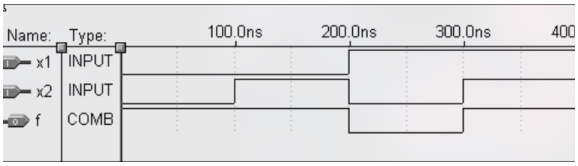
Logic circuits in complex systems, such as modern computers, are designed using sophisticated **CAD tools** rather than manual methods. These tools are packaged into a **CAD system**, which includes tools for:

- Design entry
- Synthesis and optimization
- Simulation
- Physical design

9.6.1 Design Entry

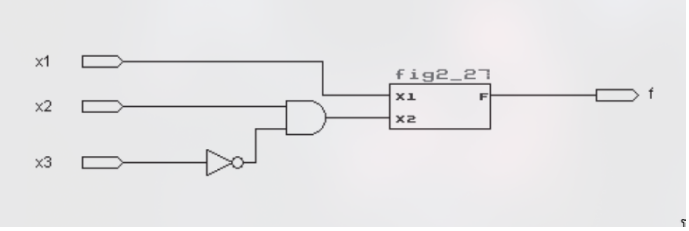
Design entry is the first step in the design process, where the circuit's functionality is described. Three common methods are:

1. **Truth Tables**: Suitable for small circuits or subcircuits. CAD tools can transform truth tables or even waveforms into logic gate networks. One example of a waveform being given as input is, It is not suitable for



larger circuits.

2. **Schematic Capture**: Uses graphical symbols to represent logic gates and interconnections. Supports hierarchical design, where subcircuits are represented as symbols. One example of a circuit represented using Schematic Capture is, It is a much better way of representing larger circuits, but its drawbacks are differences in UI, functionality of different software. This means repeated training when switching tools. Another drawback to note is that the GUI becomes awkward in case of larger circuits.



3. **Hardware Description Languages (HDLs)**: They are similar to computer programming languages except, HDL's are used to describe hardware rather than software. Two HDL's are IEEE standard - **Verilog** and **VHDL**. In comparission to Schematic capture, a HDL such as verilog provies *portability*. This means that a circuit specified in Verilog can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the Verilog specification. Both small and large logic circuit designs can be efficiently represented in Verilog code. Verilog has been used to define circuits such as microprocessors with millions of transistors. Verilog design entry can be combined with other methods. For example, a schematic- capture tool can be used in which a subcircuit in the schematic is described using Verilog.

9.6.2 Synthesis Process

Synthesis is the process of generating a logic circuit from a high-level description, such as a truth table, schematic, or Verilog code. CAD tools automate this process, which includes:

- Translating Verilog code or schematic diagrams into logic expressions.
- Generating logic equations from truth tables.
- Logic Optimization

Logic Optimization

The initial logic expressions provided by synthe-

sis tools are often not optimal (reflecting on the designer's input). **Logic optimization** is a critical step where the synthesis tools manipulate these expressions to produce a more efficient circuit. Optimization goals depend on the design requirements and the target hardware technology.

9.6.3 Functional Simulation

After completing design entry and synthesis, it is essential to verify that the designed circuit functions as expected. This is done using a **functional simulator**, which evaluates the circuit's behavior based on:

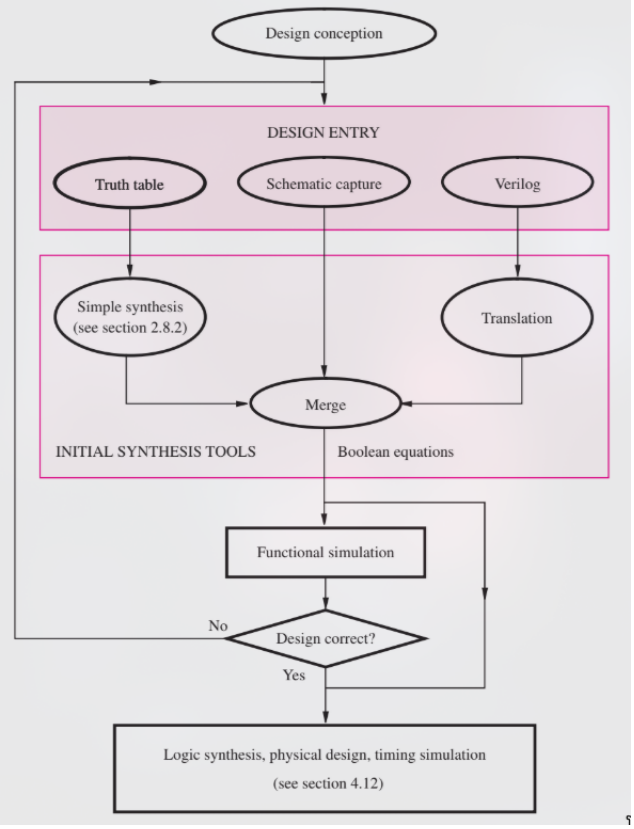
- Logic equations generated during synthesis (before optimization).
- Input valuations specified by the user.

The functional simulator:

- Applies input valuations to the logic equations.
- Computes the corresponding outputs.
- Presents the results as a truth table or timing diagram.

The user examines these results to ensure the circuit operates correctly. **Drawback:** The functional simulator assumes that the time needed for signals to propagate through the logic gates is negligible. In real logic gates this assumption is not realistic, regardless of the hardware technology chosen for implementation of the circuit.

9.6.4 Summary



10 Introduction to Verilog

Verilog is a hardware description language (HDL) used to model digital circuits. Originally developed for simulation, it is now widely used for synthesis. It was standardized as IEEE 1364-1995 and later updated to IEEE 1364-2001.

10.1 Structural Representation

A circuit can be described in Verilog using gate-level primitives. These include:

```
and (y, x1, x2); // AND gate
or (y, x1, x2, x3); // OR gate
not (y, x); // NOT gate
```

These primitives allow direct mapping of hardware structures.
Example of a simple logic circuit in Verilog:

```
module example1 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    and (g, x1, x2);
    not (k, x2);
    and (h, k, x3);
    or (f, g, h);
```

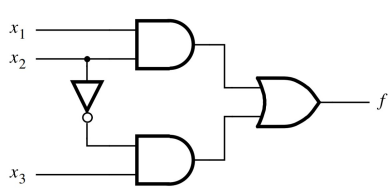


Figure 9: Simple logic function

```
endmodule
```

10.2 Behavioral Representation

Instead of explicitly describing gates, Verilog allows using logical expressions:

```
assign f = (x1 & x2) | (x2 & x3);
```

For sequential circuits, procedural statements are used inside an `always` block:

```
always @(x1 or x2 or x3)
    if (x2 == 1)
        f = x1;
    else
        f = x3;
```

The `reg` keyword is used for variables assigned in procedural blocks.

11 Boolean Algebra and Logic Functions

Boolean algebra provides a mathematical model for digital circuits. Basic operations include:

- AND: $A \cdot B$
- OR: $A + B$
- NOT: $\overline{A}$

Important laws:

$$A + 0 = A, \quad A \cdot 1 = A \quad \text{(Identity Laws)} \tag{8}$$

$$A + A = A, \quad A \cdot A = A \quad \text{(Idempotent Laws)} \tag{9}$$

$$A + \overline{A} = 1, \quad A \cdot \overline{A} = 0 \quad \text{(Complement Laws)} \tag{10}$$

Boolean functions can be simplified using Karnaugh Maps and algebraic manipulation.

12 Verilog Syntax and Coding Style

12.1 Naming Conventions

Variable names must start with a letter and can contain letters, numbers, _, and `$`. Verilog is case-sensitive, so `X` and `x` are different.

12.2 Operators

Verilog supports bitwise and logical operators:

- AND: `&`, OR: `|`, NOT: `~`
- Logical AND: `&&`, Logical OR: `||`
- Bitwise XOR: `^`, XNOR: `^`

12.3 Best Practices

- Use meaningful variable names.
- Use indentation and comments for readability.
- Minimize unnecessary variables and loops.

13 Sequential Circuits

In addition to combinational circuits, Verilog supports sequential circuits using flip-flops and registers.

13.1 D Flip-Flop

A D flip-flop can be described using an `always` block triggered on the clock edge:

```
always @(posedge clk)
    Q <= D;
```

This ensures that the output updates only on the rising edge of the clock.

13.2 Finite State Machines (FSMs)

FSMs are widely used in control circuits. A simple state machine can be implemented as:

```
always @(posedge clk or posedge reset)
    if (reset)
        state <= S0;
    else
        case (state)
            S0: state <= S1;
            S1: state <= S2;
            S2: state <= S0;
        endcase
```

14 Design Verification

14.1 Testbenches

A testbench is used to verify the correctness of a Verilog design. It applies input patterns and checks the outputs:

```
initial begin
    x1 = 0; x2 = 0; x3 = 0;
    #10 x1 = 1; // Apply stimulus after 10 time units
    #10 x2 = 1;
    #10 x3 = 1;
end
```

14.2 Simulation and Debugging

Simulation tools allow observation of waveforms and the detection of errors. Common issues include:

- Incorrect sensitivity lists in `always` blocks.
- Improper use of blocking (`=`) vs. non-blocking (`<=`) assignments.

15 Conclusion

This summary introduced digital circuits, Verilog coding styles, Boolean algebra, and sequential circuit design. For practical implementation, CAD tools and testbenches help verify correctness. Future topics include hardware technologies and optimization techniques.