ECE 253 Lecture Notes

Hei Shing Cheung

Digital and Computer Systems, Fall 2025 ECE253

The up-to-date version of this document can be found at `https://github.com/HaysonC/skulenotes`

Chapter 1

Digital Circuits that Compute, Store, and Control

Introduction

Layers of Computation In hardware, we have the following layers of abstraction:

- Computation
- Adders
- Logic Gates
- Transistors
- Silicon

In this course, we will focus on the first three layers, on top of the logic gate level.

Layer of abstraction At this course, for the digital systems part, we would start from understanding logic gates, all the way to understanding computer architecture, with each level of abstraction hiding the details of the lower level.

1.1 Hierarchy, Modularity, and Regularity

Definiton 1.1.0.1 (Hierarchy)**.** The division of system into a set of modules, then further subdividing each module into smaller modules, and so on, until pieces are *easy* to understand.

Definiton 1.1.0.2 (Modularity)**.** The design principle that modules have well-defined functions and interfaces so they connect easily without unintended side effects.

Definiton 1.1.0.3 (Regularity)**.** The uniformality of modules, such that the reusability of common modules reduces the number of distinct modules to be designed.

1.1.1 Digital Logic Gates

Logic gates are made out of transistors:

Definiton 1.1.1.1 (Transistor)**.** A transistor is a 3-terminal device behaving as a switch. When the voltage on the terminal is HI, the switch is closed, and when the voltage is LO, the switch is open.

Factors Affecting Speed of Digital Circuits

- **Transistors and Electrons take time to switch.** A transistor (State of the Art) takes 2-3 picoseconds to switch. Gates takes 40 ps and an 8-bit adder takes 300 ps.
- **Wires take time to propagate signals.** Signals travel at approximately 2/3 the speed of light in a vacuum, which is about 200,000 kilometers per second in a typical silicon wire.
- **Capacitance** There would be RCL circuits formed by the wires and transistors, which would cause delay.

1.2 Digital Logic Foundations

1.2.1 Number Systems

Definiton 1.2.1.1 (Number System)**.** A number system is a way of representing numbers using a set of symbols (digits) and a base (radix). The base determines the number of unique digits that can be used in the number system.

Common Number Systems You should be familiar with the following number systems:

- Decimal (Base 10): Digits 0-9
- Binary (Base 2): Digits 0-1
- Hexadecimal (Base 16): Digits 0-9, A-F

In computer systems, we use binary to represent information, and we would often use hexadecimal to represent binary numbers in a more compact way - a group of 4 bits (a nibble) can be represented by a single hexadecimal digit.

Example 1.2.1.2 (Binary, Decimal, and Hexadecimal Numbers)**.** Below is a table showing the conversion of binary numbers to decimal numbers, along with their hexadecimal representation.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Table 1.1: Binary to Decimal and Hexadecimal Conversion

Example 1.2.1.3 (Decimal to Binary Conversion)**.** To convert a decimal number to binary, we can use the method of successive division by 2. For example, to convert the decimal number 437 to binary:

$$
\begin{aligned}
437 \div 2 &= 218 && \text{remainder } 1 \\
218 \div 2 &= 109 && \text{remainder } 0 \\
109 \div 2 &= 54 && \text{remainder } 1 \\
54 \div 2 &= 27 && \text{remainder } 0 \\
27 \div 2 &= 13 && \text{remainder } 1 \\
13 \div 2 &= 6 && \text{remainder } 1 \\
6 \div 2 &= 3 && \text{remainder } 0 \\
3 \div 2 &= 1 && \text{remainder } 1 \\
1 \div 2 &= 0 && \text{remainder } 1
\end{aligned}
$$

Reading the remainders from bottom to top, we get the binary representation of 437

Example 1.2.1.4. To convert $(512000)_{10}$ to binary, we recognize that $512000 = 2^9 \times 1000$. We know that $2^9 = 512$ and $1000_{10} = 1111101000_2$ (by method outlined above). Therefore, we can shift the binary representation of 1000 left by 9 bits to get the binary representation of 512000:

$$(512000)_{10} = (1111101000000000000)_2$$

Note An alternative method is to devide by powers of 2.

Fractional Numbers. To represent fractional numbers in binary, we can use the method of successive multiplication by 2 (fixed point representation). Or we can use floating point representation, which is similar to scientific notation in decimal.

1.2.2 Binary Arithmetic and Logic

Binary Arithmetic Binary arithmetic is similar to decimal arithmetic, but it only uses two digits (0 and 1). Addition is associated with a sum and carry.

Binary Addition The rules for binary addition are as follows:

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

Table 1.2: Binary Addition

This could be summerize as the following logic:

$$\text{Sum} = A \oplus B, \quad \text{Carry} = A \cdot B \tag{1.1}$$

Binary Subtraction The rules for binary subtraction are defined using the addition of negative numbers (2's complement):

Definiton 1.2.2.1 (Least Significant Bit (LSB) and Most Significant Bit (MSB))**.** The least significant bit (LSB) is the rightmost bit in a binary number, while the most significant bit (MSB) is the leftmost bit.''

Definiton 1.2.2.2 (2's Complement)**.** The 2's complement of a binary number is obtained by inverting all the bits (1's complement) and adding 1 to the least significant bit (LSB).

Example 1.2.2.3 (Number Inversion)**.** To find the 2's complement of the binary number $(10110010)_2$:

1. Invert all the bits: $(01001101)_2$
2. Add 1 to the LSB:

$$\begin{array}{r} 01001101 \\ +\,00000001 \\ \hline 01001110 \end{array}$$

Therefore, the 2's complement of $(10110010)_2$ is $(01001110)_2$.

Definiton 1.2.2.4 (Logic Function)**.** A logic function $L : \{0,1\}^n \to \{0,1\}$ is a mathematical function that takes n binary inputs and produces a single binary output based on a set of rules.

Definiton 1.2.2.5 (Truth Table)**.** A truth table is a tabular representation of a logic function that lists all possible combinations of input values and their corresponding output values.

Definiton 1.2.2.6 (Boolean Algebra)**.** Boolean algebra is a branch of algebra that deals with binary variables and logical operations. It provides a set of rules and properties for manipulating and simplifying logic functions. The specific rules and properties would be covered in later lectures.

1.2.3 Transistors as Switches

Definiton 1.2.3.1 (Transistor)**.** Transistor operates as a switch. The switch is open only when the gate is high. We denote the state of the gate as $x \in \{0,1\}$, where 0 is LO and 1 is HI. If input end of the switch is HI, the output end could be modeled by the logic function:

$$L(x) = x$$

Example 1.2.3.2 (Serial Transistors)**.** Consider two transistors connected in series, with the input end of the first transistor connected to HI. The output end of the second transistor can be modeled by the following truth table:

x_1	x_2	$L(x_1, x_2)$
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.3: Truth Table for Two Transistors in Series

The logic function can be expressed as:

$$L(x_1, x_2) = x_1 \cdot x_2$$

where $\cdot$ denotes the AND operation.

Example 1.2.3.3 (Parallel Transistors)**.** Consider two transistors connected in parallel, with the input end of both transistors connected to HI. The output end can be modeled by the following truth table:
The logic function can be expressed as:

$$L(x_1, x_2) = x_1 + x_2$$

where $+$ denotes the OR operation.

x_1	x_2	$L(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.4: Truth Table for Two Transistors in Parallel

Example 1.2.3.4. Consider a circuit with a transistor connected to LO and the output end connected to LO, The other ends of the output and the transistor are connected together to a HI (and a resistor). The output end of the circuit can be modeled by the following truth table:

x	$L(x)$
0	1
1	0

Table 1.5: Truth Table for a Transistor Connected to LO

The logic function can be expressed as:

$$L(x) = \overline{x}$$

where $\overline{x}$ denotes the NOT operation.

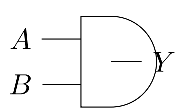
1.2.4 Basic Logic Gates

Definiton 1.2.4.1 (AND Gate)**.** An AND gate outputs 1 only if all inputs are 1. The truth table for a 2-input AND gate is shown in Table 1.3.
The logic function for an AND gate with inputs A and B can be expressed as:

$$L(A, B) = A \cdot B = AB$$

Note when no operator is present, it is assumed to be AND.

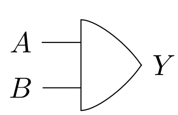
The digital logic symbol for an AND gate is shown below:



Definiton 1.2.4.2 (OR Gate)**.** An OR gate outputs 1 if at least one input is 1. The truth table for a 2-input OR gate is shown in Table 1.4.
The logic function for an OR gate with inputs A and B can be expressed as:

$$L(A, B) = A + B$$

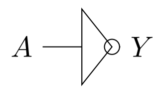
The digital logic symbol for an OR gate is shown below:



Definiton 1.2.4.3 (NOT Gate)**.** A NOT gate outputs the inverse of the input. The truth table for a NOT gate is shown in Table 1.5.
The logic function for a NOT gate with input A can be expressed as:

$$L(A) = \overline{A}$$

The digital logic symbol for a NOT gate is shown below:



1.2.5 Additional Logic Gates

Example 1.2.5.1 (XOR Operation)**.** We have two switches; when both switches are in the same state (both open or both closed), the output is 0. When the switches are in different states (one open and one closed), the output is 1. The truth table for this operation is shown below:

x_1	x_2	$L(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.6: Truth Table for XOR Operation

The logic function can be expressed as:

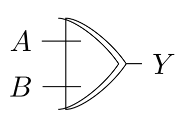
$$L(x_1, x_2) = x_1 \oplus x_2 = \overline{x_1}x_2 + x_1\overline{x_2}$$

where $\oplus$ denotes the XOR operation.

Definiton 1.2.5.2 (XOR Gate)**.** An XOR gate outputs 1 if the inputs are different. The truth table for a 2-input XOR gate is shown in Table 1.6.
The logic function for an XOR gate with inputs A and B can be expressed as:

$$L(A, B) = A \oplus B = \overline{A}B + A\overline{B}$$

where $\oplus$ denotes the XOR operation. The digital logic symbol for an XOR gate is shown below:

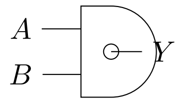


In addition, we have the following gates:

Definiton 1.2.5.3 (NAND Gate)**.** A NAND gate outputs 0 only if all inputs are 1. The truth table for a 2-input NAND gate is as expected for the complement of AND.
The logic function for a NAND gate with inputs A and B can be expressed as:

$$L(A, B) = \overline{A \cdot B} = \overline{A} + \overline{B}$$

The digital logic symbol for a NAND gate is shown below:



Definiton 1.2.5.4 (NOR Gate)**.** A NOR gate outputs 1 only if all inputs are 0. The truth table for a 2-input NOR gate is the dual of the OR gate.
The logic function for a NOR gate with inputs A and B can be expressed as:

$$L(A, B) = \overline{A + B}$$

NAND and NOR Gates are Cheaper NAND gates and NOR gates are cheaper than AND and OR gates because they require fewer transistors to implement. A 2-input NAND gate can be implemented using 4 transistors, while a 2-input AND gate requires 6 transistors (4 for the NAND gate and 2 for the NOT gate). The same applies to NOR and OR gates.

NAND and NOR Gates are Universal (Functionally Complete) Additionally, NAND and NOR gates are universal gates, meaning that any logic function can be implemented using only NAND or NOR gates.

This makes them more versatile and cost-effective for building complex digital circuits.

Commonly Used Logic Operators Below is a table summarizing the commonly used logic operators:

Operator	Symbol	Description
AND	$\cdot$ or adjacency	Outputs 1 if all inputs are 1
OR	$+$	Outputs 1 if at least one input is 1
NOT	$\overline{x}$ or x' or $\sim x$	Outputs the logical negation of the input
XOR	$\oplus$	Outputs 1 if inputs are different
NAND	$\bar{\cdot}$	Outputs 0 if all inputs are 1
NOR	$\overline{+}$	Outputs 0 if at least one input is 1
XNOR	$\overline{\oplus}$	Outputs 1 if inputs are the same

Table 1.7: Commonly Used Logic Operators

1.2.6 Sum of Products (SOP) Form

Definiton 1.2.6.1 (Literal)**.** A literal is a variable or its negation. For example, A and $\overline{A}$ are literals. A literal can be either true or false, and it represents a single value in a logical expression.

Definiton 1.2.6.2 (Product Term)**.** A product term is a logical synonym for AND.

Definiton 1.2.6.3 (Sum Term)**.** A sum term is a logical synonym for OR.

Definiton 1.2.6.4 (Sum of Products (SOP) Form)**.** A logical expression is in sum of products (SOP) form if it is a sum of product terms. For example, the expression $AB + \overline{A}C + BC$ is in SOP form.

Definiton 1.2.6.5 (Minterm)**.** A product term that evaluates to one for exactly one row of the truth table is called a minterm.

Example 1.2.6.6 (Minterm)**.** For a given truth table for x_1, x_2, x_3, the minterms are:

x_1	x_2	x_3	Minterm
0	0	0	$m_0 = \overline{x_1 x_2 x_3}$
0	0	1	$m_1 = \overline{x_1 x_2} x_3$
0	1	0	$m_2 = \overline{x_1} x_2 \overline{x_3}$
0	1	1	$m_3 = \overline{x_1} x_2 x_3$
1	0	0	$m_4 = x_1 \overline{x_2 x_3}$
1	0	1	$m_5 = x_1 \overline{x_2} x_3$
1	1	0	$m_6 = x_1 x_2 \overline{x_3}$
1	1	1	$m_7 = x_1 x_2 x_3$

Table 1.8: Minterms for 3 Variables

Note that each minterm corresponds to a unique combination of input values that produces an output of 1. To create the minterm, you would try to make every literal one.

Definiton 1.2.6.7 (Canonical SOP Form)**.** A logical expression is in canonical SOP form if it is a sum of minterms.

1.2.7 Product of Sums (POS) Form

Definiton 1.2.7.1 (Product of Sums (POS) Form)**.** A logical expression is in product of sums (POS) form if it is a product of sum terms. For example, the expression $(A + B)(\overline{A} + C)(B + C)$ is in POS form.

Definiton 1.2.7.2 (Maxterm)**.** A sum term that evaluates to zero for exactly one row of the truth table is called a maxterm.

Example 1.2.7.3 (Maxterm)**.** For a given truth table for x_1, x_2, x_3, the maxterms are:

x_1	x_2	x_3	Maxterm
0	0	0	$M_0 = (x_1 + x_2 + x_3)$
0	0	1	$M_1 = (x_1 + x_2 + \overline{x_3})$
0	1	0	$M_2 = (x_1 + \overline{x_2} + x_3)$
0	1	1	$M_3 = (x_1 + \overline{x_2} + \overline{x_3})$
1	0	0	$M_4 = (\overline{x_1} + x_2 + x_3)$
1	0	1	$M_5 = (\overline{x_1} + x_2 + \overline{x_3})$
1	1	0	$M_6 = (\overline{x_1} + \overline{x_2} + x_3)$
1	1	1	$M_7 = (\overline{x_1} + \overline{x_2} + \overline{x_3})$

Table 1.9: Maxterms for 3 Variables

Note that each maxterm corresponds to a unique combination of input values that produces an output of 0. To create the maxterm, you would try to make every literal zero.

Definiton 1.2.7.4 (Canonical POS Form)**.** A logical expression is in canonical POS form if it is a product of maxterms.

Theorem 1.2.7.5 (Converting between Canonical Forms)**.** Any logical expression can be converted from canonical SOP form to canonical POS form and vice versa. For $i \in \{0, 1, \ldots, 2^n - 1\}$ and $S \subseteq \{0, 1, \ldots, 2^n - 1\}$, we have:

$$f(x_1, x_2, \ldots, x_n) = \sum_{i \in S} m_i = \prod_{i \notin S} M_i$$

Example 1.2.7.6. We have the following converison:

$$f(x_1, x_2, x_3) = m_1 + m_3 + m_5 + m_7 = M_0 M_2 M_4 M_6$$

1.2.8 Boolean Algebra and Logic Minimization

Definiton 1.2.8.1 (Boolean Algebra)**.** Boolean algebra is a branch of algebra that deals with binary variables and logical operations. It is a effective means to describe logic circuits with a set of rules derived from the axioms of Boolean algebra.

Definiton 1.2.8.2 (Axioms of Boolean Algebra)**.** The axioms of Boolean algebra are a set of fundamental rules that govern the behavior of binary variables and logical operations. The number stems consist only of the set $\{0, 1\}$, with the following axioms:

- $0 \cdot 0 = 0$
- $1 \cdot 1 = 1$
- $0 \cdot A = 0 \cdot 1 = 1 \cdot 0 = 0$ for any A
- if $x = 0$ then $\overline{x} = 1$

Dual Form We can also derive the following logical equivalences from the axioms:

- $A + 0 = A$
- $A + 1 = 1$
- $0 + 1 = 1 + 0 = 1$
- $A + \overline{A} = 1$

where 1 is the multiplicative identity and 0 is the additive identity.

Rules derived from the Axioms of Boolean Algebra The following rules can be derived from the axioms of Boolean algebra:

Theorem 1.2.8.3.

- $x \cdot 0 = 0$ (Annihilation)
- $x \cdot 1 = 1 \cdot x = x$ (Identity)

- $x \cdot \overline{x} = 0$ (Complementation)
- $x \cdot x = x$ (Idempotent)
- $x + 0 = 0 + x = x$ (Identity)
- $x + 1 = 1 + x = 1$ (Annihilation)
- $x + \overline{x} = 1$ (Complementation)

Theorem 1.2.8.4. The following identities can be derived from the axioms of Boolean algebra:

- Commutative Laws:
  - $A + B = B + A$
  - $A \cdot B = B \cdot A$
- Associative Laws:
  - $A + (B + C) = (A + B) + C$
  - $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive Laws:
  - $A \cdot (B + C) = A \cdot B + A \cdot C$
  - $A + (B \cdot C) = (A + B) \cdot (A + C)$

Proof. By perfect induction. We can exhaustively check all possible values of A, B, and C (0 or 1) to verify that both sides of each identity yield the same result. □

Theorem 1.2.8.5 (Covering Theorem)**.** The following is true:

$$x + xy = x$$

and its dual:

$$x(x + y) = x$$

Theorem 1.2.8.6 (Combining Theorem)**.** The following is true:

$$xy + x\overline{y} = x$$

and its dual:

$$(x + y)(x + \overline{y}) = x$$

Theorem 1.2.8.7 (De Morgan's Theorem)**.** The following is true:

$$\overline{xy} = \overline{x} + \overline{y}$$

and its dual:

$$\overline{x + y} = \overline{x} \cdot \overline{y}$$

Proof. By direct proof. We have:

$$\begin{aligned}\overline{xy} &= \overline{x}\overline{y} + \overline{x}y + x\overline{y} \quad \text{(In Canonical SOP Form)}\\ &= \overline{x}\overline{y} + \overline{x}y + x\overline{y} + x\overline{y} \quad \text{(Adding } x\overline{y} \text{ using } x + x = x)\\ &= \overline{x}(\overline{y} + y) + \overline{y}(x + \overline{x}) \quad \text{(Using Distributive Law)}\\ &= \overline{x} \cdot 1 + \overline{y} \cdot 1 \quad \text{(Using Complementation)}\\ &= \overline{x} + \overline{y} \quad \text{(Using Identity)}\end{aligned}$$

□

Theorem 1.2.8.8 (Absorption / Redundancy Theorem)**.** The following is true:

$$x + \overline{x}y = x + y$$

and its dual:

$$x(\overline{x} + y) = xy$$

Proof. By direct proof. We have:

$$\begin{aligned}x + \overline{x}y &= x + \overline{x}y + xy \quad \text{(Adding } xy \text{ using } x + xy = x)\\ &= x(1 + y) + \overline{x}y \quad \text{(Using Distributive Law)}\\ &= x \cdot 1 + \overline{x}y \quad \text{(Using Identity)}\\ &= x + y \quad \text{(Using Combining Theorem)}\end{aligned}$$

□

Summary of Important Theorems The following table summarizes the important theorems/laws in Boolean algebra:

Table 1.10: Summary of Important Theorems/Laws in Boolean Algebra (original form and dual form)

Law / Theorem	**Original Form(s)**	**Dual Form(s)**
Commutative Law	$A + B = B + A$	$A \cdot B = B \cdot A$
Associative Law	$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive Law	$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Identity Law	$A + 0 = A$	$A \cdot 1 = A$
Null Law	$A + 1 = 1$	$A \cdot 0 = 0$
Idempotent Law	$A + A = A$	$A \cdot A = A$
Complement Law	$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
Double Negation Law	$\overline{\overline{A}} = A$	-
De Morgan's Theorem	$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$
Absorption / Redundancy Theorem	$A + \overline{A}B = A + B$	$A(\overline{A} + B) = AB$
Combining Theorem	$AB + A\overline{B} = A$	$(A + B)(A + \overline{B}) = A$
Covering Theorem	$A + AB = A$	$A(A + B) = A$

Logic Minimization The goal of logic minimization is to reduce the number of logic gates and inputs in a digital circuit while maintaining its functionality. This is important because it can lead to cost savings, improved performance, and reduced power consumption. Logic minimization can be achieved through various techniques, including Boolean algebra simplification, Karnaugh maps, and the Quine-McCluskey algorithm.

Theorem 1.2.8.9 (Nand as SOP)**.** And SOP circuit can be implemented using only NAND gates.

Theorem 1.2.8.10 (Nor as POS)**.** A POS circuit can be implemented using only NOR gates.

Example 1.2.8.11 (Gumball Fact)**.** Consider three sensors s_0, s_1, s_2 that detect defects in Gumballs. Those sensors are normally 0, but would be 1 if a defect is detected as follows:

$$\begin{cases} s_0 = 1 & \text{if the Gumball is too small} \\ s_1 = 1 & \text{if the Gumball is too big} \\ s_2 = 1 & \text{if the Gumball is too light} \end{cases}$$

We are to design a circuit that would output 1 if the Gumball is either too large or too small and too light. We can express canonical SOP form as:

$$\begin{aligned} L(s_0, s_1, s_2) &= m_3 + m_4 + m_5 + m_6 + m_7 \\ &= \overline{s_0}s_1s_2 + s_2\overline{s_0s_2} + s_0\overline{s_1}s_2 + s_0s_1\overline{s_2} + s_0s_1s_2 \end{aligned}$$

Using the Combining Theorem

$$= \overline{s_0}s_1s_2 + s_2\overline{s_0s_2} + s_0s_2 + s_0s_1$$

Using the Absorption Theorem

$$= s_2\overline{s_0} + s_0s_2 + s_0s_1$$

Using the Covering Theorem

$$= s_2 + s_0s_1$$

Example 1.2.8.12. *Derive a minimal POS expression for* $f(x_1, x_2, x_3 = \prod M(0, 2, 4)$
We have:

$$\begin{aligned} f(x_1, x_2, x_3) &= M_0M_2M_4 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x_2} + x_3)(\overline{x_1} + x_2 + x_3) \end{aligned}$$

Recognizing the combining theorem $(x + y)(x + \overline{y}) = x$

$$\begin{aligned} &= (x_1 + x_3)(x_2 + x_3)(\overline{x_1} + x_3) \\ &= (x_1 + x_3)(x_2 + \overline{x_1} + x_3) \\ &= (x_1 + x_3)(x_2 + x_3) \end{aligned}$$

Theorem 1.2.8.13 (Transporting POS to SOP)**.** A POS circuit can be implemented using only NAND gates. This is achieved by applying De Morgan's Theorem and the properties of NAND gates. We can use the trick $f = \overline{\overline{f}}$ to convert the POS expression into a form that can be implemented with NAND gates.

Theorem 1.2.8.14 (Transporting SOP to POS)**.** An SOP circuit can be implemented using only NOR gates. This is achieved by applying De Morgan's Theorem and the properties of NOR gates. We can use the trick $f = \overline{\overline{f}}$ to convert the SOP expression into a form that can be implemented with NOR gates.

Example 1.2.8.15. *Implement the function* $f(x_1, x_2, x_3) = \sum m(1, 3, 5, 7)$ *using only NAND gates.* We have:

$$\begin{aligned}
f(x_1, x_2, x_3) &= m_1 + m_3 + m_5 + m_7 \\
&= \overline{x_1}\,\overline{x_2}x_3 + \overline{x_1}x_2x_3 + x_1\overline{x_2}x_3 + x_1x_2x_3 \\
&= x_3(\overline{x_1}\,\overline{x_2} + \overline{x_1}x_2 + x_1\overline{x_2} + x_1x_2) \\
&= x_3(x_1 + x_2) \quad \text{(Using Combining Theorem)} \\
&= \overline{\overline{x_3(x_1 + x_2)}} \quad \text{(Using } f = \overline{\overline{f}}\text{)} \\
&= \overline{\overline{x_3} + \overline{x_1 + x_2}} \quad \text{(Using De Morgan's Theorem)} \\
&= \overline{\overline{x_3} + (\overline{x_1} \cdot \overline{x_2})} \quad \text{(Using De Morgan's Theorem)}
\end{aligned}$$

1.3 Combinational Logic Circuits

Definiton 1.3.0.1 (Combinational Logic Circuit)**.** A combinational logic circuit is a digital circuit that implements a specific logic function using a combination of logic gates. The output of a combinational logic circuit depends only on the current inputs and not on any previous inputs or states.

Definiton 1.3.0.2. Hardware Description Language (HDL) A hardware description language (HDL) is a specialized programming language used to describe the structure, behavior, and operation of electronic circuits and systems. HDLs are used in the design and verification of digital systems, including integrated circuits (ICs) and field-programmable gate arrays (FPGAs). The two most commonly used HDLs are VHDL (VHSIC Hardware Description Language) and Verilog.

1.3.1 Introduction of Verilog

Definiton 1.3.1.1 (Module)**.** A module is a self-contained block of hardware that has inputs and outputs and an internal implementation (behavioral or structural). Modules are the unit of hierarchy in Verilog.

Example 1.3.1.2 (Module Block)**.** A module block in Verilog is defined using the 'module' keyword, followed by the module name and a list of input and output ports. For example:

```
module basic_logic(input logic a, b,
```

```
                     output logic w, x, y, z);
    assign w = a & b; // AND gate
    assign x = a | b; // OR gate
    assign y = ~a;    // NOT gate
    assign z = a ^ b; // XOR gate
endmodule
```

Keywords The keywords used in the module block are:

- `assign`: Used to define continuous assignments for combinational logic.
- `logic`: A type of variable that can hold binary values (0 or 1).

Definiton 1.3.1.3 (Continuous Assignment)**.** A continuous assignment is used to model combinational logic in Verilog. It is defined using the `assign` keyword, followed by the output signal, the assignment operator '=', and the logic expression. Continuous assignments are evaluated whenever any of the input signals change. That is, the output is considered instantaneously updated when the input changes (ignoring propagation delay).

1.3.2 Multiplexers (Mux)

Example 1.3.2.1 (2-1 Multiplexers (Mux))**.** *Design a circuit that controls a light f based on either two switches x and y. The switch that control the light is determined by a control signal s. If $s = 0$, the light is controlled by switch x. If $s = 1$, the light is controlled by switch y.*
We have the following truth table:

s	x	y	f
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

Table 1.11: Truth Table for Mux

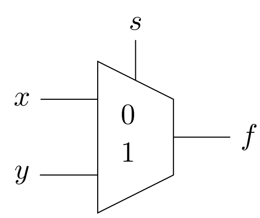
From the truth table, we can derive the following canonical SOP expression:

$$\begin{aligned} f(s,x,y) &= m_2 + m_3 + m_5 + m_7 \\ &= \overline{s}x\overline{y} + \overline{s}xy + s\overline{x}y + sxy \\ &= \overline{s}x(\overline{y} + y) + sy(\overline{x} + x) \quad \text{(Using Distributive Law)} \\ &= \overline{s}x \cdot 1 + sy \cdot 1 \quad \text{(Using Complementation)} \\ &= \overline{s}x + sy \quad \text{(Using Identity)} \end{aligned}$$

The verilog implementation is as follows:

```
module mux2to1(input logic x, y, s,
               output logic f);
    assign f = (~s & x) | (s & y);
endmodule
```

The Diagram of a Mux is a trapezoid shown below:



4-1 Multiplexer (Mux) A 4-1 Mux has 4 data inputs (d_0, d_1, d_2, d_3). The select signal would have two bit (s_0, s_1) to select one of the four data inputs to be outputted. In general, this is call a multibit signal, a.k.a a bus.

Definiton 1.3.2.2 (Bus)**.** A bus is a bundle of signals. It is used to transfer data between different components of a digital system.

Example 1.3.2.3 (Verilog for a 2bit 2-1 Mux)**.** The verilog implementation for a 2bit 2-1 Mux is as follows:

```
module mux2to1_2bit(input logic [1:0] x, y, // 2-bit inputs
                    input logic s,          // select signal
                    output logic [1:0] f); // 2-bit output
    assign f[0] = (~s & x[0]) | (s & y[0]); // LSB
    assign f[1] = (~s & x[1]) | (s & y[1]); // MSB
    // Alternatively, we can use the following single line:
    // assign f = s ? y : x; // If s=1, f=y; else f=x
endmodule
```

As we can see, we can initialize the datatype of a bus using the following syntax:

```
logic [n-1:0] bus_name; // n-bit bus
```

1.3.3 Adders

Definiton 1.3.3.1 (Half Adder)**.** A half adder is a combinational logic circuit that performs the addition of two single-bit binary numbers. It has two inputs, typically denoted as A and B, and two outputs: the sum (S) and the carry (C). The sum output represents the least significant bit of the addition, while the carry output represents any overflow that occurs when both inputs are 1. We start with the following truth table: From the truth table, we can derive the following expressions

A	B	S_1	S_2
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1.12: Truth Table for Half Adder

using canonical SOP form:

$$\begin{aligned} S_0 &= m_1 + m_2 = \overline{A}B + A\overline{B} = A \oplus B \\ S_1 &= m_3 = AB \end{aligned}$$

The Verilog implementation is as follows:

```
module half_adder(input logic A, B,
                  output logic [1:0] S);
    assign s[1] = a & b;
    assign s[0] = a ^ b;
endmodule
```

Definiton 1.3.3.2 (Full Adder)**.** A full adder is a combinational logic circuit that performs the addition of three single-bit binary numbers: two significant bits and a carry-in bit. It has three inputs, we denote the signifcant bit as A_i and B_i, and the carry-in bit as C_i, and two outputs: the sum (S_i) and the carry-out (C_{i+1}). The sum output represents the least significant bit of the addition, while the carry-out output represents any overflow that occurs when the sum exceeds the value that can be represented by a single bit. We start with the following truth table: From the

A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1.13: Truth Table for Full Adder

truth table, we can derive the following expressions using canonical SOP form:

$$
\begin{aligned}
S_i &= m_1 + m_2 + m_4 + m_7 \\
&= \overline{A_i}\overline{B_i}C_i + \overline{A_i}B_i\overline{C_i} + A_i\overline{B_i}\overline{C_i} + A_iB_iC_i \\
&= A_i \oplus B_i \oplus C_i \\
C_{i+1} &= m_3 + m_5 + m_6 + m_7 \\
&= \overline{A_i}B_iC_i + A_i\overline{B_i}C_i + A_iB_i\overline{C_i} + A_iB_iC_i
\end{aligned}
$$

Using the Distributive Law and A = A + A

$$
\begin{aligned}
&= B_iC_i(\overline{A_i} + A_i) + A_iB_i(\overline{C_i} + C_i) + A_iC_i(\overline{B_i} + B_i) \\
&= A_iB_i + B_iC_i + A_iC_i
\end{aligned}
$$

The circuit diagram of a full adder is shown below:

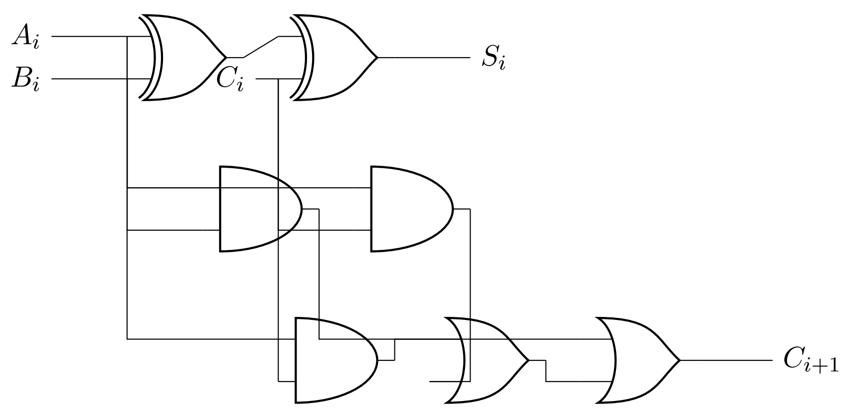


Figure 1.1: Circuit Diagram of a Full Adder

The Verilog implementation is as follows:

```
module full_adder(input logic A_i, B_i, C_i,
                  output logic S_i, C_i_plus_1)
    assign S_i = A_i ^ B_i ^ C_i;
    assign C_i_plus_1 = (A_i & B_i) | (B_i & C_i) | (A_i & C_i);
endmodule
```

Definiton 1.3.3.3 (Hierarchical Verilog Module)**.** Hierarchical Verilog is a verilog module that instantiates other modules within it. This allows for the creation of complex designs by combining simpler modules.

Example 1.3.3.4 (3-bit Ripple Carry Adder)**.** A 3-bit ripple carry adder can be implemented using three instances of the full adder module. The Verilog implementation is as follows:

```
module adder3(input logic [2:0] A, B,
              input logic C_in,
```

```
            output logic [2:0] S,
            output logic C_out);
    logic C_1, C_2; // Internal carry signals

    // Instantiate full adders
    full_adder FA0 (A[0], B[0], C_in, S[0], C_1);
    full_adder FA1 (A[1], B[1], C_1, S[1], C_2);
    full_adder FA2 (A[2], B[2], C_2, S[2], C_out);
endmodule
```

1.4 Digital Storage Elements

1.5 Finite State Machines (FSM)

Chapter 2

Computer Organization and Assembly Language

What is Assembly Language? We know that we can run C/C++ on any computer (Machine Agnostic), but how does the computer understand C/C++? The answer is the compiler that parse it to assembly through:

1. **Front-end Parser:** The front-end parser would parse the C/C++ code into an intermediate representation (IR), which is a low-level representation of the code that is easier to optimize. The front-end parser would also perform optimizations on the IR, such as loop unrolling, inlining, and dead code elimination.
2. **Back-end Parser:** The back-end parser would take the optimized IR and generate assembly code for a specific architectures.

The assembly code is then assembled into machine code, which is a series of 0s and 1s that the computer can understand. The assembly would be specific to the architecture of the computer (machine dependent), which is why we have different assembly languages for different architectures (e.g., x86, RISC-V, ARM).

2.1 Computer Organization

2.2 Assembly Language