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.

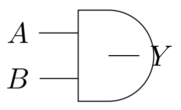
1.2.4 Basic Logic Gates

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.

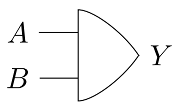
The digital logic symbol for an AND gate is shown below:



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:

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

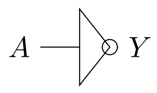
The digital logic symbol for an OR gate is shown below:



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:

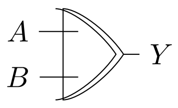
$$L(x_1, x_2) = x_1 \oplus x_2 = \overline{x_1}x_2 + x_1\overline{x_2}$$

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:

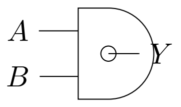


In addition, we have the following gates:

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	$\overline{\cdot}$	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 $i \in \{0, 1, \dots, 2^n - 1\}$ and $S \subseteq \{0, 1, \dots, 2^n - 1\}$, we have:

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

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:

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

□

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:

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

□

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:

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

Using the Combining Theorem

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

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:

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

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

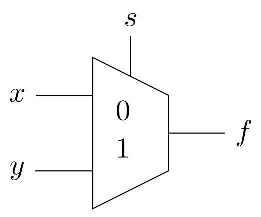
From the truth table, we can derive the following canonical SOP expression:

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

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:

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

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:

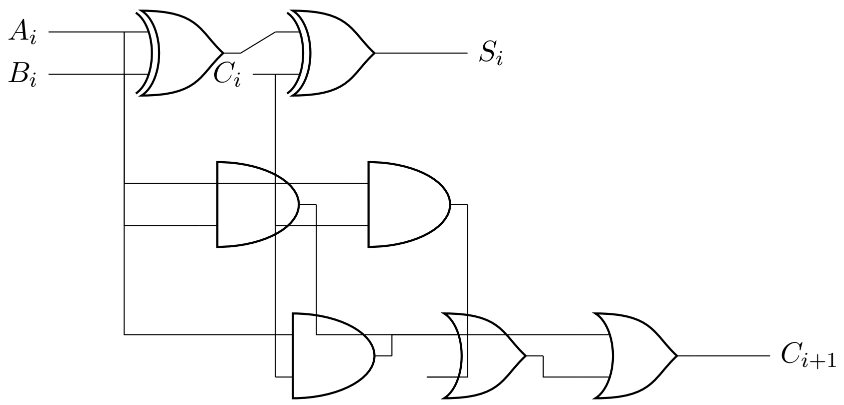


Figure 1.1: Circuit Diagram of a Full Adder

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
```

1.3.4 HEX0 7 Segment Display Decoder

Definiton 1.3.4.1 (Decoder)**.** A decoder is a combinational logic circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. Each output line corresponds to one of the possible combinations of the input lines. When a specific combination of input lines is activated (set to 1), the corresponding output line is activated (set to 1), while all other output lines remain inactive (set to 0). Decoders are commonly used in applications such as memory address decoding, data multiplexing, and digital display systems.

Example 1.3.4.2 (2-bit 7-Segment Display)**.** A 7-segment display is an electronic display device that consists of seven individual segments (labeled h_0 to h_6) that can be illuminated in different combinations to represent numerical digits (0-9) and some alphabetic characters. The truth table for binary inputs x_0, x_1, x_2, x_3 (only x_0 and x_1 are used in this demonstration) to control the 7-segment display is shown below (note that 0 represents an active segment): So we have:

x_1	x_0	h_0	h_1	h_2	h_3	h_4	h_5	h_6
0	0	0	0	0	0	0	0	1
0	1	1	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0

Table 1.14: Truth Table for 7-Segment Display (0-3)

$$
\begin{aligned}
h_0 &= \overline{x_1}x_0 \\
h_1 &= 0 \\
h_2 &= x_1\overline{x_0} \\
h_3 &= h_0 = \overline{x_1}x_0 \\
h_4 &= x_0 \\
h_5 &= x_0 + x_1 \\
h_6 &= \overline{x_1}
\end{aligned}
$$

The Verilog implementation is as follows:

```
module seven_segment(input logic [1:0] x,
                         output logic [6:0] h);
        assign h[0] = ~x[1] & x[0];
        assign h[1] = 1'b0;
        assign h[2] = x[1] & ~x[0];
        assign h[3] = ~x[1] & x[0];
        assign h[4] = x[0];
        assign h[5] = x[0] | x[1];
        assign h[6] = ~x[1];
    endmodule
```

Definiton 1.3.4.3 (If-Else Statement)**.** An if-else statement is a conditional statement that allows for branching in the execution of code based on the evaluation of a boolean expression. In Verilog, if-else statements are used to model combinational logic and can be used within always blocks or initial blocks. The syntax for an implementation of mux using an if-else statement in Verilog is as follows:

```
module mux(input logic x, y, s,
           output logic f);
    always_comb // Always @(x or y or s) is equivalent.
    // The sensitivity list gives the signals
    // that can trigger the always block.
    begin
        if (s == 1'b0) // If statements must go inside always block.
            f = x; // Keyword 'assign' not used in always block.
        else
            f = y;
        end
    end
endmodule
```

Analogy The always block is like the `useEffect(() => { ... }` hook in React. The sensitivity list is like the dependency array in React.

Definiton 1.3.4.4 (Case Statement)**.** A case statement is a control flow statement that allows for multi-way branching based on the value of a single expression. In Verilog, case statements are used to model combinational logic and can be used within always blocks. The syntax for an implementation of a full 7-segment display using a case statement in Verilog is as follows:

```
module seven_segment_case(input logic [3:0] x,
                          output logic [6:0] h);
    always_comb begin
        case (x)
            4'b0000: h = 7'b1000000; // 0
            4'b0001: h = 7'b1111001; // 1
            4'b0010: h = 7'b0100100; // 2
```

```
                4’b0011: h = 7’b0110000; // 3
                4’b0100: h = 7’b0011001; // 4
                4’b0101: h = 7’b0010010; // 5
                4’b0110: h = 7’b0000010; // 6
                4’b0111: h = 7’b1111000; // 7
                4’b1000: h = 7’b0000000; // 8
                4’b1001: h = 7’b0010000; // 9
                default: h = 7’b1111111; // All segments off for invalid input
            endcase
        end
    endmodule
```

1.3.5 Field Programmable Gate Array (FPGA)

Definiton 1.3.5.1 (Gate Array)**.** A gate array is a type of integrated circuit (IC) that consists of a regular array of uncommitted logic gates. These gates can be interconnected to implement various digital functions.

Definiton 1.3.5.2 (Programmable Gate and Lookup Table (LUT))**.** A programmable gate is a logic gate that can be configured to perform different logic functions based on the input signals. A lookup table (LUT) is a memory element that stores the truth table of a logic function. The LUT can be programmed to implement any logic function by storing the corresponding output values for each possible combination of input values.

The implementation of a 2-LUT is via three muxes and 4 memory cells that store the expected output values for each combination of the two input values. The three muxes are used to select the appropriate output value based on the input values. In general, a n-LUT would require 2^n memory cells. By programming the memory cells with the desired output values, we can configure the LUT to implement any logic function of two variables.

Definiton 1.3.5.3 (FPGA Fabric)**.** An FPGA fabric is the underlying architecture of an FPGA that consists of an array of programmable logic blocks (PLBs, i.e. LUTs), switch blocks, and connection blocks. In addition around the FPGA fabric, there are input/output pads (I/O pads) that provide the interface between the FPGA and the pins in the chip package. The FPGA fabric is designed to be highly flexible and reconfigurable, allowing users to implement a wide range of digital circuits and systems.

Files The following files are commonly used in FPGA design:

- **.v** or **.sv** files: These files contain the Verilog or SystemVerilog code that describes the digital circuit or system to be implemented on the FPGA.
- **.sof** files: These files contain the configuration data that is used to program the FPGA. The .sof file is generated by the **synthesis** and place-and-route tools based on the Verilog or SystemVerilog code.

- **.qsf** files: These files contain the pin assignments and other configuration settings for the FPGA. The .qsf file is used by the synthesis and place-and-route tools to ensure that the design is correctly mapped to the physical resources of the FPGA.

CAD Flow The following steps are commonly used in the FPGA design process:

1. **HDL Coding:** Write the Verilog or SystemVerilog code that describes the digital circuit or system to be implemented on the FPGA.
2. **Synthesis:** Use a synthesis tool to convert the HDL code to logic gates.
3. **Functional Simulation:** Use a simulation tool to verify the functionality of the synthesized design. If corrections are needed, go back to the HDL coding step.
4. **Physical Design (Place-and-Route):** Use a place-and-route tool to map the synthesized design to the physical resources of the FPGA.
5. **Timing Analysis:** In reality, it takes time for signal to propagate through the circuit. In our course, we ignore this.
6. **Generate Bitstream:** Generate the .sof file that contains the configuration data for the FPGA.

1.4 Digital Storage Elements

1.4.1 Introduction to Sequential Circuits

Definiton 1.4.1.1 (Sequential Circuit)**.** Sequential circuits are digital circuits whose outputs depend not only on the current inputs but also on the history of past inputs. In other words, sequential circuits have memory and can store information about previous states. This is in contrast to combinational circuits, where the output is solely determined by the current inputs.

We call sequential circuit a circuit that have **state**.

Definiton 1.4.1.2 (Cross-Couple RS-Latch)**.** A cross-coupled RS latch is a basic memory element that can store one bit of information. It consists of two NOR gates connected in a feedback loop. The latch has two inputs, labeled R (reset) and S (set), and two outputs, labeled Q and $\overline{Q}$. The behavior of the latch is as follows:

- When S = 1 and R = 0, the latch is set, and Q = 1 and $\overline{Q} = 0$.
- When S = 0 and R = 1, the latch is reset, and Q = 0 and $\overline{Q} = 1$.
- When S = 0 and R = 0, the latch maintains its previous state (no change).
- When S = 1 and R = 1, this condition is invalid for a NOR-based RS latch, as it would force both outputs to be 0, which violates the requirement that Q and $\overline{Q}$ must always be complements of each other. Additoanlly, this would break the logic for subsequent operations.

The truth table for a cross-coupled RS latch is shown below:

S	R	Q	$\overline{Q}$
0	0	Q_{prev}	$\overline{Q}_{prev}$
0	1	0	1
1	0	1	0
1	1	Invalid (0)	Invalid (0)

Table 1.15: Truth Table for Cross-Coupled RS Latch

Definiton 1.4.1.3 (Timing Diagram)**.** A timing diagram is a graphical representation of the relationship between signals in a digital circuit over time. It shows how the signals change. The timing diagram of a RS Latch could be visualized as follows:

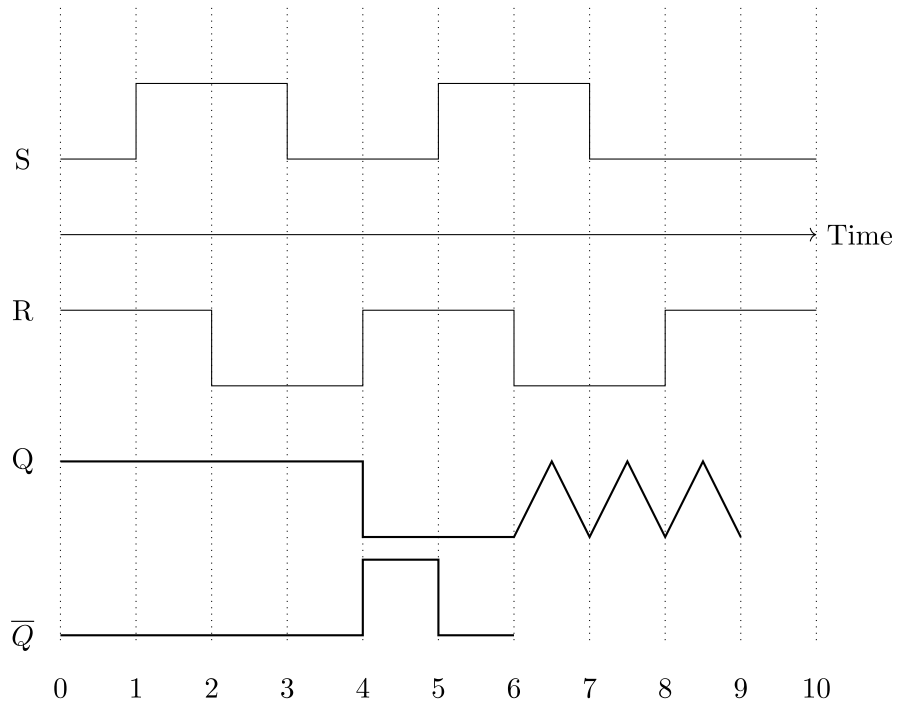


Figure 1.2: Timing Diagram of a Cross-Coupled RS Latch

Definiton 1.4.1.4 (Gated RS-Latch)**.** We can put an AND gate along with the R and S signals, and the circuit will only reset to 0 or set to 1 when the clock is HI. The truth table of a gated RS latch is:

CLK	S	R	$Q(t+1)$
0	X	X	$Q(t)$
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	Invalid

Table 1.16: Truth Table for Gated RS Latch

Additionally, we can use purely NAND gates instead of NOR gates and AND gates, using a technique called bubble pushing. A implementation is shown in the below figure.

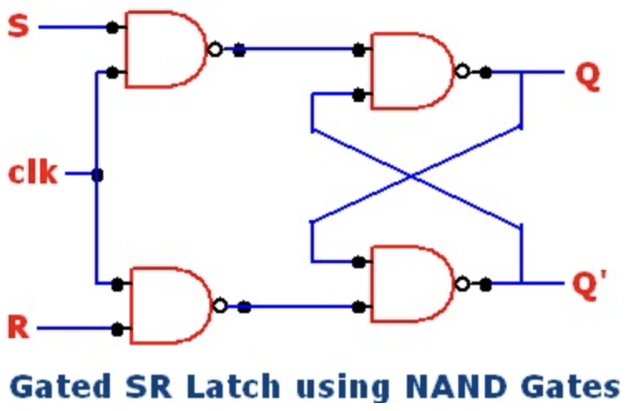


Figure 1.3: Gated SR latch using NAND

Definiton 1.4.1.5 (Gated D-Latch)**.** We can set $D = S = \overline{R}$, so we would prevent $S = R = 1$. So when the clock is HI, Q tracks D (a.k.a. Transparent Mode), and when clock is high, it won't store (a.k.a. Opaque Mode). The truth table is shown as follows:

CLK	D	$Q(t+1)$
0	X	$Q(t)$
1	0	0
1	1	1

Table 1.17: Truth Table for Gated D Latch

The Verilog implementation is as follows:

```
module d_latch(input logic D, CLK,
               output logic Q);
    always_latch begin // this is new in SystemVerilog,
    // dont write else in always_latch
        if (CLK) // Transparent mode
            Q = D;
    end
endmodule
```

Equivalence with a mux If we feed back the output of a mux back to one of the input, this is essentially a gated D latch with the SW as CLK and the input is the D.

Definiton 1.4.1.6 (D-Flip Flops (Registors))**.** A D flip-flop is a type of flip-flop that captures the value of the D input at a specific portion of the clock cycle (rising edge) and holds that value until the next clock cycle. It is done by chaining the output of a D latch to the input of another D-latch. While the second D-latch recognizes the CLK, and the first one recognizes $\overline{\text{CLK}}$.

The verilog implementation is as follows:

```
module d_ff(input logic D, CLK,
            output logic Q);
    always_ff @(posedge CLK) begin
        Q <= D; // Non-blocking assignment
    end
endmodule
```

`postedge` is a keyword used to create a positive edge-triggered FF. We use <= instead of = because we want to use non-blocking assignment. This is because in a sequential circuit, all the FFs are triggered at the same time, and we want to make sure that the value of Q is updated only after all the FFs have been evaluated.

Definiton 1.4.1.7 (8-Bit Register)**.** An 8-bit register is a digital storage device that can hold 8 bits of binary data. It is typically implemented using a series of D flip-flops, where each flip-flop stores one bit of data. The register has a clock input (CLK) that controls when the data is loaded into the register, and an enable input (EN) that determines whether the register should load new data or retain its current value. When EN is high (1), the register loads the data present on the D input into the flip-flops on the rising edge of the clock. When EN is low (0), the register retains its current value regardless of changes on the D input.

The Verilog implementation is as follows:

```
module register8(input logic [7:0] D,
                 input logic CLK, EN,
                 output logic [7:0] Q);
    always_ff @(posedge CLK) begin
        if (EN)
            Q <= D; // Load new data
        // else retain old data
    end
endmodule
```

Resets We have two types of rests to reset the FF to a known state:

- **Synchronous reset:** Active only on the clock edge. If reset = 1 (or resetn = 0) at the triggering edge the flip-flop is forced to the predefined state (usually 0); otherwise it captures D.
- **Asynchronous reset:** Active immediately when asserted, forcing the flip-flop to the predefined state (usually 0) regardless of the clock; normal edge-triggered operation resumes when deasserted.

Definiton 1.4.1.8 (Synchronous Reset D-FF)**.** The Verilog implementation is as follows:

```
module d_ff_sync_reset(input logic D, CLK, RSTn,
                       output logic Q);
    always_ff @(posedge CLK) begin
        if (RSTn == 1'b0)
            Q <= 1'b0; // Reset to 0
        else
            Q <= D; // Normal operation
    end
endmodule
```

Definiton 1.4.1.9 (Asynchronous Reset D-FF)**.** The Verilog implementation is as follows:

```
module d_ff_async_reset(input logic D, CLK, RSTn,
                        output logic Q);
    always_ff @(posedge CLK or negedge RSTn) begin
        if (RSTn == 1'b0)
            Q <= 1'b0; // Reset to 0
        else
            Q <= D; // Normal operation
    end
endmodule
```

Enable Inputs We can add an enable input to a D-FF by adding a mux before the D input. It determines whether data is loaded on the clock edge or the old data is retained. If the enable input is high (1), the data on the D input is loaded into the flip-flop on the rising edge of the clock. If the enable input is low (0), there would be a feedback signal going into the mux, so the old data is retained.

Definiton 1.4.1.10 (Counters)**.** To count binary numbers, we have the follow truth table: From the truth table, we can derive the following rules:

- Q_0 toggles every clock cycle.
$$Q_0(t+1) = Q_0(t) \oplus 1$$
- Q_1 toggles when Q_0 is 1.
$$Q_1(t+1) = Q_1(t) \oplus Q_0(t)$$

CLK Cycle	Binary Count (Q2 Q1 Q0)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Table 1.18: Truth Table for a 3-bit Counter

- Q_2 toggles when Q_1 and Q_0 are both 1.

$$Q_2(t+1) = Q_2(t) \oplus (Q_1(t)Q_0(t))$$

We call this a TQ flip-flop, where T stands for toggle. It is done by connecting the Q output back, to a new XOR gate. The input of that XOR gate is Q and T. And the XOR gate output is connected to the D input of a D-FF. The Verilog implementation is as follows:

```
module counter3(input logic CLK, RSTn,
                output logic [3:0] Q);
    always_ff @(posedge CLK or negedge RSTn) begin
        if (RSTn == 1'b0)
            Q <= 4'b0000;
        else begin
            Q[0] <= Q[0] ^ 1'b1; // Toggle Q0
            Q[1] <= Q[1] ^ Q[0]; // Toggle Q1
            Q[2] <= Q[2] ^ (Q[1] & Q[0]); // Toggle Q2
            Q[3] <= Q[3] ^ (Q[2] & Q[1] & Q[0]); // Toggle Q3
        end
    end
endmodule
```

Example 1.4.1.11 (Clock Reduction)**.** Suppose we want to reduce a 1Mhz clock to a 1Hz clock. We can use a 20-bit counter to achieve this. Since $2^{20} = 1,048,576$, which is slightly more than 1 million, we can use the most significant bit (MSB) of the counter as the output clock. The MSB will toggle every 2^{19} clock cycles, which is approximately 1 second when the input clock is 1MHz. The Verilog implementation is as follows:

```
module clock_divider(input logic CLK_1MHZ, RSTn,
                     output logic CLK_1HZ);
    logic [19:0] counter; // 20-bit counter
    always_ff @(posedge CLK_1MHZ or negedge RSTn) begin
        if (RSTn == 1'b0)
            counter <= 20'b0;
        else
            counter <= counter + 1; // Increment counter
    end
    assign CLK_1HZ = counter[19]; // MSB as 1Hz clock
endmodule
```

In a real hardware implementation, we would feed the bit to a d flip-flop, and that would create the rising and falling edge signals.

1.4.2 Optimizations

Definiton 1.4.2.1 (Karnaugh Map (K-Map))**.** A Karnaugh map (K-map) is a graphical representation of a truth table that is used to simplify boolean algebra expressions. It is a grid-like structure that organizes the minterms of a boolean function in such a way that adjacent cells differ by only one variable. By grouping adjacent cells that contain 1s, we can identify common factors and simplify the boolean expression.

The K-map for a 3-variable function is shown below:

	BC			
A	00	01	11	10
0	m0	m1	m3	m2
1	m4	m5	m7	m6

Figure 1.4: 3-Variable Karnaugh Map

The K-map for a 4-variable function is shown below:

	CD			
AB	00	01	11	10
00	m0	m1	m3	m2
01	m4	m5	m7	m6
11	m12	m13	m15	m14
10	m8	m9	m11	m10

Figure 1.5: 4-Variable Karnaugh Map

The following rules are used to group the minterms in a K-map:

- Groups must contain only 1s.
- Groups must be rectangular and contain 2^n cells (where n is a non-negative integer).
- Groups can wrap around the edges of the K-map.
- Each group should be as large as possible.

- Each 1 in the K-map should be covered by at least one group.

By following these rules, we can create groups of 1s in the K-map and simplify the boolean expression accordingly.

Example 1.4.2.2. Suppose we have the following truth table: We can derive the following

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 1.19: Truth Table for Example

minterms from the truth table:

$$F(A, B, C) = m_1 + m_2 + m_3 + m_5 + m_6 + m_7$$

The K-map for the function is shown below: We can group the minterms as follows:

	BC			
A	00	01	11	10
0	0	1	1	1
1	0	1	1	1

Figure 1.6: Karnaugh Map for Example

- Group of four covering m_2, m_3, m_6, m_7: This group corresponds to $B = 1$ (since B is constant in this group). So this group simplifies to B.
- Group of two covering m_1, m_5: This group corresponds to $A = 0$ and $C = 1$ (since A and C are constant in this group). So this group simplifies to $\overline{A}C$.

Therefore, the simplified boolean expression is:

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

Definiton 1.4.2.3 (Don't Care Conditions)**.** In some cases, certain input combinations may never occur or their output values are irrelevant to the overall function. These input combinations are referred to as "don't care" conditions and are typically represented by an "X" in the truth table. When simplifying a boolean expression using a K-map, we can treat these "don't care" conditions as either 0 or 1, whichever leads to a simpler expression. This flexibility can help in creating larger groups of 1s in the K-map, leading to a more simplified boolean expression.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	1
1	0	0	0
1	0	1	X
1	1	0	1
1	1	1	1

Table 1.20: Truth Table with Don't Care Conditions

Example 1.4.2.4 (Don't Care Conditions)**.** Suppose we have the following truth table with don't care conditions: We can derive the following minterms from the truth table:

$$F(A, B, C) = m_1 + m_3 + m_6 + m_7$$

The K-map for the function is shown below: We can group the minterms as follows:

	BC			
A	00	01	11	10
0	0	X	1	X
1	0	X	1	X

Figure 1.7: Karnaugh Map with Don't Care Conditions

- Group of four covering m_3, m_7 and the don't care conditions: This group corresponds to $C = 1$ (since C is constant in this group). So this group simplifies to C.
- Group of two covering m_1: This group corresponds to $A = 0$ and $B = 0$ (since A and B are constant in this group). So this group simplifies to $\overline{AB}$.

Therefore, the simplified boolean expression is:

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

Gray Code Gray code is a binary numeral system where two successive values differ in only one bit. This property makes Gray code useful in minimizing errors in digital systems, particularly in applications like rotary encoders and digital communication. The following table shows the 3-bit binary and Gray code representations:

Terminologies For a K-map, we have the following terminologies:

- **Implicant:** A product term (AND term) that can be used to cover one or more minterms in the K-map.

Binary	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Table 1.21: 3-bit Binary and Gray Code Representations

- **Prime Implicant:** An implicant that cannot be combined with any other implicant to form a larger implicant. In other words, it is an implicant that covers the maximum number of minterms possible.
- **Essential Prime Implicant:** A prime implicant that covers at least one minterm that is not covered by any other prime implicant. In other words, it is a prime implicant that is necessary to cover all the minterms in the K-map.
- **Cover:** A set of implicants that together cover all the minterms in the K-map.

Example 1.4.2.5 (Implicant and Prime Implicant)**.** Consider the following K-map: We can iden-

	BC			
A	00	01	11	10
0	1	1	0	0
1	1	1	1	0

Figure 1.8: Karnaugh Map for Example

tify the following implicants:

- $A'B'$ covering minterms m_0, m_1
- $A'B$ covering minterms m_2, m_3
- AB' covering minterms m_4, m_5
- AB covering minterms m_6, m_7
- B covering minterms m_2, m_3, m_6, m_7
- A covering minterms m_4, m_5, m_6, m_7

The prime implicants are:

- $A'B'$ (cannot be combined with any other implicant)
- B (cannot be combined with any other implicant)

- A (cannot be combined with any other implicant)

The essential prime implicants are:

- $A'B'$ (covers minterm m_0 which is not covered by any other prime implicant)
- B (covers minterms m_2, m_3 which are not covered by any other prime implicant)
- A (covers minterms m_4, m_5 which are not covered by any other prime implicant)

Therefore, the cover of the K-map is:

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

Example 1.4.2.6. Let $f(x_1, x_2, x_3, x_4) = \sum m(2, 4, 5, 8, 10, 11, 12, 13, 15)$ be a function represented by the following K-map:

	x_1x_2			
x_3x_4	00	01	11	10
00	0	1	1	1
01	0	1	1	0
11	0	0	1	1
10	1	0	0	1

Figure 1.9: Karnaugh Map for Example

Procedure to fine the minimum cost cover

1. Find all prime implicants.
2. Identify all essential prime implicants (EPIs) and include them in the cover.
3. Remove all minterms covered by the EPIs from the K-map.
4. Check and inspect whether it is the minimum cost cover.

Definiton 1.4.2.7 (K-Maps with Maxterm)**.** We can also use K-maps to simplify boolean expressions represented by maxterms. The procedure is similar to that of minterms, but instead of grouping 1s, we group 0s. The following rules are used to group the maxterms in a K-map:

- Groups must contain only 0s.
- Groups must be rectangular and contain 2^n cells (where n is a non-negative integer).
- Groups can wrap around the edges of the K-map.
- Each group should be as large as possible.
- Each 0 in the K-map should be covered by at least one group.

By following these rules, we can create groups of 0s in the K-map and

1.5 Finite State Machines (FSM)

Definiton 1.5.0.1 (State Diagram)**.** A state diagram is a graphical representation of a finite state machine (FSM) that illustrates the states, transitions, inputs, and outputs of the system. It consists of nodes (representing states) and directed edges (representing transitions between states). Each edge is typically labeled with the input condition that triggers the transition and may also include the output associated with that transition.

State diagrams are used to visualize the behavior of sequential systems, making it easier to understand how the system responds to different inputs and how it transitions between various states over time.

Example 1.5.0.2. Cosnider a circult that detect the input sequance $w =$'101' using a FSM. The state diagram is shown below: The FSM has four states: S0, S1, S2, and S3. The initial state is

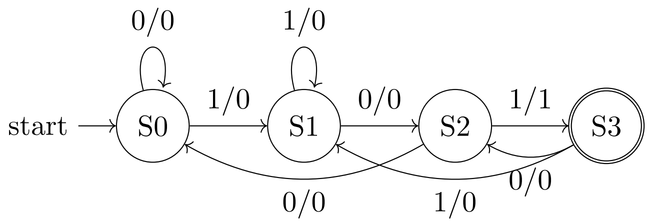


Figure 1.10: State Diagram for FSM that detects '101'

S0, and the accepting state is S3. The transitions between states are labeled with the input/output pairs. For example, from state S0, if the input is 1, the FSM transitions to state S1 and outputs 0. The FSM detects the sequence '101' and outputs 1 when it reaches state S3.

We can also draw its state table as follows:

Current State	$w = 0$	$w = 1$	Output
S0	S0	S1	0
S1	S2	S1	0
S2	S0	S3	0
S3	S2	S1	1

Table 1.22: State Table for FSM that detects '101'

Then, we would assign the states. First the circuit has 4 states, so we need at least 2 flip-flops y_2, y_1 to represent the states. We can assign the states as follows:

- S0 = 00, y_2y_1
- S1 = 01, $y_2\overline{y_1}$
- S2 = 10, $\overline{y_2}y_1$
- S3 = 11, $\overline{y_2y_1}$

Then, we can derive the following state transition table:

Current State ($y_2 y_1$)	$w = 0$	$w = 1$	Output
00	00	01	0
01	10	01	0
10	00	11	0
11	10	01	1

Table 1.23: State Transition Table for FSM that detects '101'

From the state transition table, we can synthesis the circuit using D flip-flops. Let Y_2 and Y_1 be the next state of y_2 and y_1, respectively. We can derive the following equations:

- $Y_1 = w$
- $Z = y_2 y_1$
- $Y_2 = \overline{w} y_1 + w \overline{y_2 y_1}$

Example 1.5.0.3 (Alternative Design)**.** Alternatively, we can design the state diagram in which each state represents the last three values. We can set the following states table: And then we can

$y_1 y_2 y_3$	$w = 0$	$w = 1$	Output
000	000	001	0
001	010	011	0
010	100	101	0
011	110	111	0
100	000	001	0
101	010	011	1
110	100	101	0
111	110	111	0

Table 1.24: State Table for Alternative FSM that detects '101'

draw the following equations:

- $Y_3 = w$
- $Y_2 = y_3$
- $Y_1 = y_2$

The output $Z = y_3 \overline{y_2} y_1$. This design is simpler, but it uses more states (8 states instead of 4 states). This is an example of the trade-off between the number of states and the complexity of the logic.

We call this a shift register, where the input is shifted into the register on each clock cycle. The output is taken from the last bit of the register.

Definiton 1.5.0.4 (Finite State Machine (FSM))**.** A finite state machine (FSM) is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is composed of a finite number of states, transitions between those states, inputs, and outputs. FSMs are used to model the behavior of systems that can be in one of a limited number of states at any given time and can transition from one state to another based on input signals.

There are two main types of FSMs:

- **Moore Machine:** In a Moore machine, the outputs depend only on the current state. The output is associated with each state, and it changes only when the state changes.
- **Mealy Machine:** In a Mealy machine, the outputs depend on both the current state and the current inputs. The output can change immediately in response to changes in the input, even if the state remains the same.

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

2.1.1 Basic Computational Units

Definiton 2.1.1.1 (Arithmetic Logical Unit)**.** An Arithmetic Logical Unit (ALU) is a digital circuit that performs arithmetic and logical operations on binary numbers. It is a fundamental building block of the central processing unit (CPU) in a computer. The ALU takes two binary inputs, performs the specified operation, and produces a binary output.

The ALU can perform a variety of operations, including:

- Arithmetic operations: addition, subtraction, multiplication, division
- Logical operations: AND, OR, NOT, XOR
- Bitwise operations: shift left, shift right
- Comparison operations: equal to, less than, greater than

The specific operations supported by an ALU depend on its design and the architecture of the computer it is used in.

Definiton 2.1.1.2 (Instructions)**.** Instructions are the basic commands that a CPU can execute. They are typically represented in assembly language, which is a low-level programming language that is specific to a particular computer architecture. Each instruction consists of an operation code (opcode) and operands, which specify the data to be operated on.

In RISC-V architecture, instructions are 32 bits long and are divided into different formats based on the type of operation being performed. The most common instruction formats are R-type, I-type, S-type, B-type, U-type, and J-type. Each format has a specific layout for the opcode and operands.

RISC-V instructions can perform a variety of operations, including arithmetic operations (e.g., addition, subtraction), logical operations (e.g., AND, OR), data movement (e.g., load, store), and control flow (e.g., branch, jump).

Definiton 2.1.1.3 (Registers)**.** Registers are small, high-speed storage locations within a CPU that hold data temporarily during the execution of instructions. They are used to store operands, intermediate results, and memory addresses. Registers are faster to access than main memory, which allows for quicker data manipulation and processing.

In RISC-V architecture, there are 32 general-purpose registers (x0 to x31), each 32 bits wide in the RV32I variant. The register x0 is hardwired to zero and cannot be modified. The other registers can be used for various purposes, such as holding function arguments, return values, and temporary data.

Definiton 2.1.1.4 (Intermediates)**.** Intermediates are temporary values that are used during the execution of instructions. They are typically stored in registers and are used to hold the results of operations before they are written back to memory or used as inputs to other operations.

Definiton 2.1.1.5 (Operahands and Instruction Operations)**.** The following is a table of RISC-V instructions and the operations they perform:

The registers are `rX` and `rY`, and `D` is a 12 bit value.

Instruction	Operation	Description
`MV rX, rY`	rX = rY	Move value from rY to rX
`MV rX, #D`	rX = D	Load sign extended immediate D into rX
`ADD rX, rY`	rX = rX + rY	Add rX and rY, store result in rX
`ADD rX, #D`	rX = rX + D	Add immediate D to rX, store result in rX
`SUB rX, rY`	rX = rX - rY	Subtract rY from rX, store result in rX
`SUB rX, #D`	rX = rX - D	Subtract immediate D from rX, store result in rX
`MUL rX, rY`	rX = rX * rY	Multiply rX and rY, store result in rX
`DIV rX, rY`	rX = rX / rY	Divide rX by rY, store quotient in rX

Table 2.1: RISC-V Arithmetic Instructions

Definiton 2.1.1.6 (Opcodes)**.** An opcode (operation code) is a portion of a machine language instruction that specifies the operation to be performed. It is typically represented as a binary or hexadecimal value and is used by the CPU to determine which operation to execute.

In RISC-V architecture, opcodes are 7 bits long and are located in the least significant bits of the instruction. The opcode determines the type of instruction (e.g., arithmetic, logical, load/store, branch) and the specific operation to be performed within that type.

For example, the opcode for the ADD instruction is 0000000, while the opcode for the SUB instruction is 0100000. The CPU uses these opcodes to identify the operation and execute it accordingly.

Definiton 2.1.1.7 (Instrctuion Cycles)**.** An instruction cycle is the process by which a CPU fetches, decodes, and executes an instruction. It consists of several stages, including:

- **Fetch:** The CPU retrieves the instruction from memory.
- **Decode:** The CPU decodes the instruction to determine the operation to be performed and the operands involved.
- **Execute:** The CPU performs the operation specified by the instruction.
- **Write Back:** The CPU writes the result of the operation back to memory or a register.

The instruction cycle is repeated for each instruction in a program until the program is complete.

2.1.2 ALU Control and Datapath

Definiton 2.1.2.1 (Datapath)**.** A datapath is a collection of functional units, registers, and buses that work together to perform data processing operations within a computer system. It is responsible for the flow of data between different components of the system, such as the CPU, memory, and input/output devices.

The datapath typically includes components such as:

- **Registers:** Temporary storage locations for data.
- **ALU (Arithmetic Logic Unit):** Performs arithmetic and logical operations on data.
- **Multiplexers:** Selects between multiple data sources.
- **Buses:** Pathways for data transfer between components.

The datapath works in conjunction with the control unit, which generates control signals to coordinate the operation of the datapath components based on the instructions being executed.

Definiton 2.1.2.2 (Control Path)**.** A control path is a collection of control signals and logic that manage the operation of the datapath in a computer system. It is responsible for generating the necessary control signals to coordinate the activities of the various components within the datapath, such as registers, ALU, multiplexers, and buses.

The control path typically includes components such as:

- **Control Unit:** Generates control signals based on the current instruction being executed.
- **Instruction Decoder:** Decodes the instruction to determine the required operations and generates corresponding control signals.
- **Timing Signals:** Synchronizes the operation of the datapath components with the system clock.

The control path works in conjunction with the datapath to ensure that data is processed correctly and efficiently according to the instructions being executed.

The control signal would cotnrol the following:

- Which operation the ALU would perform.
- Which registers to read from and write to.
- When to read from or write to memory.
- How to route data through the datapath using multiplexers.

2.2 Memory Hierarchy

Definiton 2.2.0.1 (Memory)**.** The CPU uses a memory address to specify the location of the data it wants to access. For example, a 32-bit system specifies a memory address using 32 bits, which amounts to a total addressable memory of around 4GB ($2^3 2$ bytes).

Definiton 2.2.0.2 (Bit, Byte, Word)**.** A **bit** is the smallest unit of data in a computer and can have a value of either 0 or 1.

A **byte** is a group of 8 bits and is the standard unit of data used to represent a character in most computer systems.

A **word** is a group of bytes that is used to represent a larger unit of data, such as an integer or a floating-point number. The size of a word can vary depending on the architecture of the computer, but it is typically 4 bytes (32 bits).

Definiton 2.2.0.3 (Memory Array)**.** A memory array is a collection of memory cells that are organized in a grid-like structure. On a high-level, we think of it as a shape-2 matrix, where each row represents a memory address and each column represents a bit within that address.

Each cell in the array can store a single bit of data (0 or 1). The memory array is typically organized into words, where each word consists of a fixed number of bits.

For example, take a 4-byte word, then the memory array would have 4 columns, and $n - 2$ rows, where n is the total number of bits in the memory. The address format would be, for example:

$$\overbrace{A_n \dots A_2}^{\text{which word}} \qquad \underbrace{A_1 A_0}_{\text{which byte in the word}}$$

2.3 Assembly Language

Definiton 2.3.0.1 (Assembly Language)**.** Assembly language is a low-level programming language that is specific to a particular computer architecture. It provides a human-readable representation of machine code instructions, allowing programmers to write code that can be directly executed by the CPU.

Assembly language uses mnemonics to represent machine instructions, making it easier for humans to understand and write code. Each assembly instruction corresponds to a specific machine code instruction, and the assembly code is typically translated into machine code using an assembler.

List of Assembly Languages The following are some common assembly languages:

- x86 Assembly: Used in Intel and AMD processors. It is a CISC (**C**omplex **I**nstruction **S**et **C**omputer) architecture.
- ARM Assembly: Used in ARM processors, commonly found in mobile devices. It is a RISC (**R**educed **I**nstruction **S**et **C**omputer) architecture.
- RISC-V Assembly: An open-source instruction set architecture (ISA) that is gaining popularity in academia and industry. As in its name, it is a RISC architecture. In this course, we will sue the RISC-V 32I flavor.

RISC vs CISC RISC architectures have a smaller set of instructions, which are designed to be executed quickly and efficiently. CISC architectures have a larger set of instructions, which can perform more complex operations in a single instruction. This can lead to more efficient code in some cases, but it can also make the architecture more complex and harder to optimize.

Definiton 2.3.0.2 (Instruction Set Architecture (ISA))**.** An **I**nstruction **S**et **A**rchitecture (ISA) is a specification that defines the set of instructions that a computer's CPU can execute. It serves as the interface between software and hardware, providing a standardized way for programmers to write code that can be executed on a specific architecture.

The ISA includes details such as:

- The format and encoding of instructions
- The types of operations that can be performed (e.g., arithmetic, logical, data movement)
- The number and types of registers available
- The addressing modes used to access memory
- The conventions for function calls and returns

Different ISAs are designed for different types of processors and applications, and they can vary significantly in terms of complexity and capabilities.

Notably, the instruction set does not define the underlying hardware implementation, which can vary even for processors that implement the same ISA.

2.3.1 Types of Instructions

Definiton 2.3.1.1 (Operand)**.** An operand is a value or data that an instruction operates on. In assembly language, operands can be registers, memory addresses, or immediate values (constants). The operands specify the source and destination of the data for the operation being performed by the instruction.

Definiton 2.3.1.2 (Arithmetic Instructions)**.** Arithmetic instructions are a type of assembly language instruction that perform mathematical operations on numerical data. These instructions typically operate on values stored in registers or memory locations and can include operations such as addition, subtraction, multiplication, and division.

Lets start with high level code:

```
int a = b + c; // addition
int a = b - c; // subtraction
```

The equivalent assembly code using RISC-V instructions would be:

```
ADD rA, rB, rC  # rA = rB + rC
SUB rA, rB, rC  # rA = rB - rC
```

the `rA` occupies the destination operand, while `rB` and `rC` are the source operands.

Definiton 2.3.1.3 (Register Set)**.** A register set is a collection of registers within a CPU that are used for temporary storage of data during the execution of instructions. Registers are small, high-speed storage locations that can hold data such as operands, intermediate results, and memory addresses.

In RISC-V architecture, there are 32 general-purpose registers (x0 to x31), each 32 bits wide in the RV32I variant. The register x0 is hardwired to zero and cannot be modified. The other registers can be used for various purposes, such as holding function arguments, return values, and temporary data.

Definiton 2.3.1.4 (`lw` and `la` Instructions)**.** The `lw` (load word) instruction is used to load a 32-bit word from memory into a register. The syntax for the `lw` instruction is as follows:

```
lw rX, offset(rY)
```

This instruction loads the word located at the memory address calculated by adding the value in register `rY` and the signed immediate `offset` into register `rX`.

The `la` (load address) instruction is used to load the address of a label or variable into a register. The syntax for the `la` instruction is as follows:

```
la rX, LABEL
```

This instruction loads the address of the label or variable `LABEL` into register `rX`.

Example 2.3.1.5 (RISC-V Assembly Code - Rolling Sum)**.** Consider the following C code that uses that performs a rolling sum using pointers of a 4 words array:

```
void rolling_sum(int *arr, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int LIST[4] = {1, 2, 3, 4};
    int result = rolling_sum(LIST, 4);
    return 0;
}
```

The equivalent RISC-V assembly code would be:

```
.data
LIST: .word 1, 2, 3, 4  # Example array

.global _start # label is visible to other files
.text # where the instructions are located

_start:
    la s1, LIST   # Load address of LIST into s1
    lw s2, 0(s1)  # Load first element of LIST into s2
    lw s3, 4(s1)  # Load second element of LIST into s3
    add s2 , s2, s3 # sum = arr[0] + arr[1]
    lw s3, 8(s1)  # Load third element of LIST into s3
    add s2 , s2, s3 # sum += arr[2]
    lw s3, 12(s1)  # Load fourth element of LIST into s3
    add s2 , s2, s3 # sum += arr[3]
    # sum is now in s2
END: ebreak # we use ebreak to end the program, to transfer control to the simulator
```

Definiton 2.3.1.6 (Logic Instructions)**.** Logic instructions are a type of assembly language instruction that perform bitwise operations on binary data. These instructions typically operate on values stored in registers or memory locations and can include operations such as AND, OR, NOT, and XOR.

For example, consider the following high-level code:

```
int a = b & c; // bitwise AND
int a = b | c; // bitwise OR
int a = b ^ c; // bitwise XOR
int a = ~b;    // bitwise NOT
```

The equivalent assembly code using RISC-V instructions would be:

```
AND rA, rB, rC  # rA = rB & rC
OR  rA, rB, rC  # rA = rB | rC
XOR rA, rB, rC  # rA = rB ^ rC
NOT rA, rB      # rA = ~rB; Pseudo-instruction,
# use XORI rA, rB, -1, since -1 = 0xFFFFFFFF
```

where `rA` occupies the destination operand, while `rB` and `rC` are the source operands.

We can also use the immediate values as operands:

```
ANDI rA, rB, #D  # rA = rB & D
ORI  rA, rB, #D  # rA = rB | D
XORI rA, rB, #D  # rA = rB ^ D
```

Definiton 2.3.1.7 (Shift Instructions)**.** Shift instructions are a type of assembly language instruction that perform bitwise shift operations on binary data. These instructions typically operate on values stored in registers and can include operations such as logical shift left (LSL), logical shift right (LSR), and arithmetic shift right (ASR).

For example, consider the following high-level code:

```
int a = b << n; // logical shift left
int a = b >> n; // logical shift right
int a = b >> n; // arithmetic shift right
```

The equivalent assembly code using RISC-V instructions would be:

```
SLL rA, rB, rC  # rA = rB << rC (logical shift left)
SRL rA, rB, rC  # rA = rB >> rC (logical shift right)
SRA rA, rB, rC  # rA = rB >> rC (arithmetic shift right)
```

where `rA` occupies the destination operand, while `rB` is the source operand and `rC` specifies the number of bits to shift.

The arithmetic shift right (ASR) preserves the sign bit (the most significant bit) when shifting right, which is important for signed integers. In contrast, the logical shift right (LSR) fills the vacated bits with zeros, regardless of the sign of the number.

To multiply or divide by powers of two, we can use shift instructions:

```
SLA rA, rB, #D  # rA = rB * 2^D (multiply by 2^D)
SRA rA, rB, #D  # rA = rB / 2^D (divide by 2^D, if it is divisable)
```

We can also use immediate values as operands:

```
SLLI rA, rB, #D  # rA = rB << D (logical shift left)
SRLI rA, rB, #D  # rA = rB >> D (logical shift right)
SRAI rA, rB, #D  # rA = rB >> D (arithmetic shift right)
```

Definiton 2.3.1.8 (Accessing a byte and half word)**.** In RISC-V, we can access individual bytes and half-words (16 bits) using specific load and store instructions.

To load a byte from memory into a register, we use the `lb` (load byte) instruction:

```
lb rX, offset(rY)
```

This instruction loads the byte located at the memory address calculated by adding the value in register `rY` and the signed immediate `offset` into register `rX`. The loaded byte is sign-extended to fill the 32-bit register.

To load a half-word from memory into a register, we use the `lh` (load half-word) instruction:

```
lh rX, offset(rY)
```

This instruction loads the half-word located at the memory address calculated by adding the value in register `rY` and the signed immediate `offset` into register `rX`. The loaded half-word is sign-extended to fill the 32-bit register.

Similarly, to store a byte from a register into memory, we use the `sb` (store byte) instruction:

```
sb rX, offset(rY)
```

This instruction stores the least significant byte of register `rX` into the memory address calculated by adding the value in register `rY` and the signed immediate `offset`.

To store a half-word from a register into memory, we use the `sh` (store half-word) instruction:

```
sh rX, offset(rY)
```

This instruction stores the least significant half-word of register `rX` into the memory address calculated by adding the value in register `rY` and the signed immediate `offset`.