

# Fundamentals of digital systems

Ali EL AZDI

IC BA2 - Mirjana Stojilović

February 19, 2024

## **Introduction**

This document is designed to offer a LaTeX-styled overview of the Fundamentals of Digital Systems course, emphasizing brevity and clarity. Should there be any inaccuracies or areas for improvement, please do not hesitate to reach out to me at ali.elazdi@epfl.ch for corrections. Hoping to provide an experience similar to the one provided by past generations of students in other subjects. Also, for the latest version of this pdf file, as the update on drive may take some time, you can find it on my github repository at the following link:

<https://github.com/elazdi-al/FDS/blob/main/FDS.pdf>

# Contents

<b>Contents</b>	<b>3</b>
<b>1 Number Systems</b>	<b>1</b>
1.1 Digital Representations . . . . .	1
1.1.1 (Non)Redundant Number Systems . . . . .	1
1.1.2 Weighted Number Systems . . . . .	1
1.1.3 Radix Systems . . . . .	1
1.1.4 Fixed and Mixed-Radix Number Systems . . . . .	2
1.1.5 Canonical Number Systems . . . . .	3
1.2 Binary/Octal/Hexadecimal to/from Decimal . . . . .	4
1.2.1 Conversion examples . . . . .	4
1.3 Octal/Hexadecimal to/from Binary . . . . .	6
1.4 Representation of Signed Integers . . . . .	7
1.4.1 Sign-Magnitude Representation (SM) . . . . .	7
1.5 True-and-Complement (TC) . . . . .	8
1.5.1 Mapping . . . . .	8
1.5.2 Unambiguous Representation . . . . .	8
1.5.3 Converse Mapping . . . . .	8
1.6 Two's Complement System . . . . .	9
1.6.1 Sign Detection in Two's Complement System . . . . .	9
1.6.2 Mapping from Bit-Vectors to Values . . . . .	9
1.6.3 Change of Sign in Two's Complement System . . . . .	10
1.7 Range Extension and Arithmetic Shifts . . . . .	10
1.7.1 Range Extension . . . . .	10
1.8 Range Extension Algorithm in Sign-and-Magnitude . . . . .	10
1.8.1 Arithmetic Shifts . . . . .	11
1.8.2 Left Arithmetic Shift in Sign-and-Magnitude System . . . . .	11
1.8.3 Right Arithmetic Shift in Sign-and-Magnitude System . . . . .	11
1.8.4 Left Arithmetic Shift in Two's Complement System . . . . .	12
1.8.5 Right Arithmetic Shift in Two's Complement System . . . . .	12
1.9 Hamming Weight and Distance . . . . .	12
1.9.1 Hamming Weight (HW) . . . . .	12
1.9.2 Hamming Distance (HD) . . . . .	12
1.10 Binary Coded Decimal (BCD) . . . . .	13
1.10.1 BCD Encoding . . . . .	13
1.10.2 Conversion Algorithms . . . . .	13

1.11	Gray Code Conversion Algorithm . . . . .	14
1.11.1	Example . . . . .	14
<b>2</b>	<b>Number Systems (Part II)</b>	<b>15</b>
2.1	Addition and Subtraction of Unsigned Integers . . . . .	15
2.1.1	Addition of Binary Numbers . . . . .	15
2.1.2	Subtracting Two Binary Numbers . . . . .	16
2.2	Overflow and Underflow in Unsigned Binary Arithmetic . . . . .	16
2.2.1	Overflow . . . . .	16
2.2.2	Underflow . . . . .	17
2.3	Two's Complement Addition and Subtraction . . . . .	17
2.3.1	Addition and Subtraction . . . . .	18
2.3.2	Addition . . . . .	18
2.3.3	Subtraction . . . . .	18
2.4	Binary Multiplication . . . . .	18
<b>3</b>	<b>Number Systems (Part III)</b>	<b>20</b>
3.1	Fixed-Point Number Representation . . . . .	20
3.1.1	Examples of Fixed-Point Numbers . . . . .	21
3.2	Concepts of Finite Precision Math . . . . .	22
3.2.1	Precision . . . . .	22
3.2.2	Resolution . . . . .	22
3.2.3	Range . . . . .	22
3.2.4	Accuracy . . . . .	22
3.2.5	Dynamic Range . . . . .	22
3.3	Floating-Point Number Representation . . . . .	23
3.3.1	Significand, Base, Exponent . . . . .	23
3.3.2	Benefits . . . . .	23
3.3.3	Representation . . . . .	24
3.3.4	Biased Representation . . . . .	26
3.3.5	Rounding . . . . .	27
3.4	IEEE 754 Standard . . . . .	29
3.4.1	Special Values . . . . .	30
3.4.2	Overflow, underflow, and others . . . . .	30
<b>4</b>	<b>Number Systems (Part IV)</b>	<b>32</b>
4.1	Fixed-Point Arithmetic . . . . .	32
4.1.1	Addition and Subtraction . . . . .	32
4.1.2	Multiplication . . . . .	32
4.2	In two's complement . . . . .	33
4.3	Floating-Point Arithmetic . . . . .	33
4.3.1	An Example in Binary . . . . .	36
<b>5</b>	<b>Number Systems (Part V)</b>	<b>37</b>
5.1	Low precision computer arithmetic . . . . .	37
5.2	Challenges and limitations . . . . .	37
5.3	Block Floating Point . . . . .	38

<b>6 Digital Logic</b>	<b>39</b>
6.1 Introduction to Digital Logic Circuits . . . . .	39
6.1.1 The simplest binary logic element . . . . .	39
6.1.2 The simplest binary logic element . . . . .	39
6.2 Truth Tables . . . . .	41
6.3 Logic Gates . . . . .	42
6.3.1 AND GATE . . . . .	42
6.3.2 AND GATE (n-variables) . . . . .	42
6.3.3 OR GATE . . . . .	42
6.3.4 OR GATE (n-variables) . . . . .	43
6.3.5 NOT GATE . . . . .	43
6.3.6 DOUBLE NOT GATE (BUFFER) . . . . .	43
6.3.7 NAND GATE . . . . .	43
6.3.8 NOR GATE . . . . .	43
6.3.9 Example of Complex Logic Circuit . . . . .	44
6.4 Analysis of a Logic Network . . . . .	44
6.5 The Venn Diagram . . . . .	45
6.6 Network Equivalence Verification . . . . .	46
6.7 Boolean Algebra . . . . .	47
6.7.1 Axioms . . . . .	47
<b>7 Digital Logic (PART II)</b>	<b>48</b>
7.1 Logic Synthesis . . . . .	48
7.1.1 Minterms and Maxterms . . . . .	48
7.1.2 Examples . . . . .	48
7.1.3 Sum-of-Product (SOP) Form and Product-of-Sum (POS) Form . . . . .	50
7.2 NAND and NOR Logic Networks . . . . .	51
7.2.1 NAND GATE . . . . .	51
7.2.2 NOR GATE . . . . .	51
7.3 Incompletely Defined Functions . . . . .	52
7.3.1 Don't Care Condition . . . . .	52
7.3.2 Example . . . . .	52
7.3.3 Incomplete Functions . . . . .	52
7.3.4 Sum of Products (SOP) Example . . . . .	52
7.4 Even and Odd Detectors (XNOR and XOR Gates) . . . . .	53
7.4.1 XOR Gate . . . . .	53
7.4.2 XNOR Gate . . . . .	53
7.5 Design Example . . . . .	54
7.5.1 Number Display . . . . .	54
7.5.2 Multiplexer . . . . .	55
<b>8 Digital Logic (PART III)</b>	<b>56</b>
8.1 Adders . . . . .	56
8.1.1 Addition of two 1-bit binary numbers . . . . .	56
8.1.2 Binary Addition Examples . . . . .	56
8.1.3 Half-Adder . . . . .	57
8.1.4 Addition of Two N-Bit Binary Numbers . . . . .	58

8.1.5	Addition of Two N-Bit Binary Numbers . . . . .	58
8.1.6	Full-Adder . . . . .	59
8.1.7	Basic Ripple-Carry Adder . . . . .	60
8.2	Subtractors . . . . .	60
8.2.1	Subtraction of Two 1-Bit Binary Numbers . . . . .	60
8.2.2	Binary Subtraction Examples . . . . .	61
8.2.3	Subtraction of Two N-Bit Unsigned Numbers . . . . .	61
8.2.4	Full Subtractor . . . . .	61
8.2.5	N-Bit Ripple-Carry Subtractor . . . . .	62
8.3	Adders-Subtractors in two's complement . . . . .	63
8.4	Fast Adders . . . . .	63
8.4.1	Performance Matters . . . . .	63
8.4.2	Examples of delays . . . . .	64
8.4.3	Summary of the Ripple-Carry Adder-Subtractor . . . . .	65
8.4.4	Carry-Select Adder . . . . .	66
8.5	Shifting . . . . .	66
8.5.1	Barrel Shifter . . . . .	66
8.5.2	Bidirectional Shifting by up to 7 positions . . . . .	68
<b>9</b>	<b>Digital Logic (PART IV)</b> . . . . .	<b>69</b>
9.1	Transistors . . . . .	69
9.1.1	NMOS Transistor Switches . . . . .	69
9.1.2	PMOS Transistor Switches . . . . .	70
9.1.3	Example - CMOS . . . . .	72
9.1.4	CMOS Circuit Structure . . . . .	73
9.1.5	CMOS Circuits - Examples . . . . .	73
9.2	Real Voltage Waveforms . . . . .	75
9.2.1	Logic Values as Voltage Levels . . . . .	75
9.3	Voltage Transfer Characteristic . . . . .	76
9.3.1	CMOS Inverter (NOT gate) . . . . .	76
9.3.2	Ideal Waveforms and Real Waveforms . . . . .	76
9.4	Dynamic Operation . . . . .	78
9.4.1	Fan-In and Fan-Out . . . . .	78
9.4.2	Parasitic Capacitance . . . . .	78
9.4.3	Power Dissipation . . . . .	79
9.5	Hazards in Digital Circuits . . . . .	79
<b>10</b>	<b>Digital Logic and Verilog (PART V)</b> . . . . .	<b>80</b>
10.1	CAD Design Flow . . . . .	80
10.2	Verilog HDL . . . . .	81
10.2.1	Structural Modeling with Logic Gates . . . . .	81
10.2.2	Verilog Syntax . . . . .	82
10.2.3	Modules in Verilog . . . . .	82
10.2.4	Ports in Verilog . . . . .	82
10.2.5	Example - Full adder in Verilog . . . . .	83
10.2.6	Subcircuits in Verilog . . . . .	83
10.2.7	Example - Ripple-Carry Adder in Verilog . . . . .	84

<b>11 Digital Logic and Verilog(PART VI)</b>	<b>85</b>
11.1 Bus Architecture . . . . .	85
11.1.1 Using Multiplexers (MUXEs) . . . . .	85
11.1.2 Without Multiplexers . . . . .	86
11.2 Tri-State Drivers . . . . .	86
11.2.1 Types of Tri-State Drivers . . . . .	86
11.3 Complete Verilog Built-In Gate List . . . . .	86
11.4 Implementing a Bus with Tri-State Drivers . . . . .	87
11.5 Verilog Part 2. . . . .	87
11.5.1 Scalar Signal Values . . . . .	87
11.5.2 Vector Signal Values . . . . .	87
11.5.3 Parameters . . . . .	87
11.5.4 Nets . . . . .	88
11.5.5 Behavioral Modeling in Verilog . . . . .	88
11.5.6 Bit-wise Operators . . . . .	88
11.5.7 Example - Structural to Behavioral Modeling . . . . .	88
11.5.8 Continuous Assignments in Verilog . . . . .	89
11.5.9 Verilog Always Block . . . . .	90
11.5.10 If-Else Statements . . . . .	90
11.5.11 Example - 2-to-1 MUX . . . . .	91
11.5.12 Case Statement . . . . .	91
<b>12 Digital Logic and Verilog(PART VII)</b>	<b>92</b>
12.1 Memory Elements . . . . .	92
12.1.1 Introduction - Example Application: Alarm System Control . . . . .	92
12.1.2 Combinational vs. Sequential Circuits . . . . .	93
12.1.3 Basic Memory Element . . . . .	93
12.2 Latches with a Control Signal . . . . .	95
12.2.1 Gated D Latch . . . . .	95
12.2.2 Flip-Flops . . . . .	95
12.3 Clock . . . . .	95
12.3.1 Clock Signal . . . . .	95
12.3.2 Waveform . . . . .	96
12.4 Verilog Part 3. . . . .	96
12.4.1 Update to Always Block . . . . .	96
12.5 Behavioral Latch and Flip-Flop Models . . . . .	97
12.6 Practical notes . . . . .	100
<b>13 Digital Logic and Verilog(PART VIII)</b>	<b>101</b>
13.1 Finite State Machines . . . . .	101
13.1.1 Mealy State Machine . . . . .	102
13.1.2 Moore State Machine . . . . .	103
13.2 State Machine Analysis . . . . .	104
13.3 State Machine Synthesis . . . . .	108

<b>14 Digital logic and Verilog, Part IX</b>	<b>111</b>
14.1 Sequential Circuits . . . . .	111
14.1.1 Registers . . . . .	111
14.1.2 Registers (Verilog) . . . . .	112
14.2 Shift Registers . . . . .	112
14.2.1 Serial Input, Serial Output Shift Register . . . . .	112
14.2.2 Parallel Input, Parallel Output Shift Register . . . . .	113
14.3 Shift Registers (Verilog) . . . . .	114
14.3.1 Serial input/output Shift Registers . . . . .	114
14.3.2 Parallel input/output Shift Registers . . . . .	114
14.4 Counters . . . . .	115
14.5 Counters (Verilog) . . . . .	117
14.5.1 Counter Up with an Enable Signal . . . . .	117
14.5.2 Counter with Parallel-Load Capability . . . . .	117
14.6 Verilog Part 4. . . . .	118
14.6.1 For Loop . . . . .	118
14.6.2 Logical Operators . . . . .	118
14.6.3 Relational Operators . . . . .	118
14.6.4 Equality Operators in Verilog . . . . .	119
<b>15 Digital logic and Verilog, Part X</b>	<b>120</b>
15.1 Flip-flop Timing Constraints and Parameters . . . . .	120
15.1.1 Synchronous System Design . . . . .	120
15.1.2 Meeting FF Timing Constraints . . . . .	121
<b>16 All of Verilog</b>	<b>123</b>
16.1 Verilog HDL . . . . .	124
16.1.1 Structural Modeling with Logic Gates . . . . .	124
16.1.2 Verilog Syntax . . . . .	124
16.1.3 Modules in Verilog . . . . .	125
16.1.4 Ports in Verilog . . . . .	125
16.1.5 Example - Full adder in Verilog . . . . .	126
16.1.6 Subcircuits in Verilog . . . . .	126
16.1.7 Example - Ripple-Carry Adder in Verilog . . . . .	126
16.2 Complete Verilog Built-In Gate List . . . . .	127
16.3 Verilog Part 2. . . . .	127
16.3.1 Scalar Signal Values . . . . .	127
16.3.2 Vector Signal Values . . . . .	128
16.3.3 Parameters . . . . .	128
16.3.4 Nets . . . . .	128
16.3.5 Behavioral Modeling in Verilog . . . . .	128
16.3.6 Bit-wise Operators . . . . .	128
16.3.7 Example - Structural to Behavioral Modeling . . . . .	129
16.3.8 Continuous Assignments in Verilog . . . . .	129
16.3.9 Verilog Always Block . . . . .	130
16.3.10 If-Else Statements . . . . .	130
16.3.11 Example - 2-to-1 MUX . . . . .	131

16.3.12 Case Statement . . . . .	131
16.4 Verilog Part 3 . . . . .	132
16.4.1 Update to Always Block . . . . .	132
16.5 Registers (Verilog) . . . . .	133
16.6 Shift Registers (Verilog) . . . . .	133
16.6.1 Serial input/output Shift Registers . . . . .	133
16.6.2 Parallel input/output Shift Registers . . . . .	134
16.7 Counters (Verilog) . . . . .	134
16.7.1 Counter Up with an Enable Signal . . . . .	134
16.7.2 Counter with Parallel-Load Capability . . . . .	135
16.8 Verilog Part 4 . . . . .	135
16.8.1 For Loop . . . . .	135
16.8.2 Logical Operators . . . . .	135
16.8.3 Relational Operators . . . . .	136
16.8.4 Equality Operators in Verilog . . . . .	136

## CONTENTS

# Chapter 1

## Number Systems

### 1.1 Digital Representations

In a digital representation, a number is represented by an ordered n-tuple:

The n-tuple is called a **digit vector**, each element is a **digit**

The number of digits  $n$  is called the precision of the representation. (careful! leftward indexing)

$$X = (X_{n-1}, X_{n-2}, \dots, X_0)$$

Each digit is given a **set of values**  $D_i$  (eg. For base 10 representation of numbers,  $D_i = \{0, 1, 2, \dots, 9\}$ )

The **set size**, the maximum number of representable digit vectors is:  $K = \prod_{i=0}^{n-1} |D_i|$

#### 1.1.1 (Non)Redundant Number Systems

A number system is nonredundant if each digit-vector represents a different integer

#### 1.1.2 Weighted Number Systems

The rule of representation if a Weighted (Positional) Number Systems is as follows :

$$\sum_{i=0}^{n-1} X_i W_i$$

where

$$W = (W_{n-1}, W_{n-2}, \dots, W_0)$$

#### 1.1.3 Radix Systems

When weights are in this format :

$$\begin{cases} W_0 = 1 \\ W_{i+1} = W_i R_i \text{ with } 1 \leq i \leq n-1 \end{cases}$$

Also written :  $W_0 = 1, \prod_{j=0}^{i-1} R_j$

### 1.1.4 Fixed and Mixed-Radix Number Systems

In a **fixed-radix system**, all elements of the radix-vector have the same value  $r$  (*the radix*)

The weight vector in a fixed-radix system is given by:

$$W = (r^{n-1}, r^{n-2}, \dots, r^2, r, 1)$$

and the integer  $x$  becomes:

$$x = \sum_{i=0}^{n-1} X_i \times r^i$$

In a **mixed-radix system**, the elements of the radix-vector differ

### Example: Decimal Number System

The decimal number system has the following characteristics:

- Radix  $r = 10$ , it's a fixed-radix system.

The weight vector  $W$  is defined as:

$$W = (10^{n-1}, 10^{n-2}, \dots, 10^2, 10, 1)$$

An integer  $x$  in this system is represented by:

$$x = \sum_{i=0}^{n-1} X_i \times 10^i$$

For example:

$$854703 = 8 \times 10^5 + 5 \times 10^4 + 4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 3 \times 10^0$$

### Examples of Fixed and Mixed radix systems

**Fixed:** The base of number systems.

- Decimal – radix 10
- Binary – radix 2
- Octal – radix 8
- Hexadecimal – radix 16

**Mixed:** An example of a mixed radix representation, such as time:

- Radix-vector  $R = (24, 60, 60)$
- Weight-vector  $W = (3600, 60, 1)$

### 1.1.5 Canonical Number Systems

In a **canonical number system**, the set of values for a digit  $D_i$  is with  $|D_i| = R_i$ , the corresponding element of the radix vector

$$D_i = \{0, 1, \dots, R_i - 1\}$$

Canonical digit sets with fixed radix:

- Decimal:  $\{0, 1, \dots, 9\}$
- Binary:  $\{0, 1\}$
- Hexadecimal:  $\{0, 1, 2, \dots, 15\}$

Range of values of  $x$  represented with  $n$  fixed-radix- $r$  digits:

$$0 \leq x \leq r^n - 1$$

A system with fixed positive radix  $r$  and a canonical set of digit values is called a radix- $r$  conventional number system.

## 1.2 Binary/Octal/Hexadecimal to/from Decimal Conversion Table

The hexadecimal system supplements 0-9 digits with the letters A-F.

*Remark.* Programming languages often use the prefix 0x to denote a hexadecimal number.

Conversion table up to 15.

Decimal	Binary	Octal	Hexadecimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

### 1.2.1 Conversion examples

#### Binary to Decimal:

To convert a binary number to decimal, multiply each bit by two raised to the power of its position number, starting from zero on the right.

Binary:

Decimal:

$$\begin{array}{r} \underline{1011} \\ 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11 \end{array}$$

**Decimal to Binary:**

Let's convert the decimal number  $25_{10}$  to binary.

$$\begin{array}{rcl} 25 \div 2 & = & 12 \text{ remainder } 1 \text{ (LSB)} \\ 12 \div 2 & = & 6 \text{ remainder } 0 \\ 6 \div 2 & = & 3 \text{ remainder } 0 \\ 3 \div 2 & = & 1 \text{ remainder } 1 \\ 1 \div 2 & = & 0 \text{ remainder } 1 \text{ (MSB)} \end{array}$$

Thus, the binary representation of  $25_{10}$  is  $11001_2$  (reading the remainders in reverse).

*Personal Remark* The trick is always to try to answer the question, what's the biggest power of 2 I need to form the number?. For 157, the biggest power would be  $2^7 = 128$ , then  $128 + 64$  is greater than 157,  $128 + 32$  is still greater than 157,  $128 + 16 = 144$ , and so on to obtain :  $128 + 16 + 8 + 4 + 1 = 157$  which can be written as  $2^7 + 2^4 + 2^3 + 2^2 + 2^0 = 157$ . Written in binary as  $10011101_2$

**Octal to Decimal:**

Each octal digit is converted to decimal by multiplying it by eight raised to the power of its position number, starting from zero on the right.

Octal:	<u>257</u>
Decimal:	$2 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 = 128 + 40 + 7 = 175$

**Decimal to Octal:**

To convert the decimal number  $93_{10}$  to octal.

$$\begin{array}{rcl} 93 \div 8 & = & 11 \text{ remainder } 5 \\ 11 \div 8 & = & 1 \text{ remainder } 3 \\ 1 \div 8 & = & 0 \text{ remainder } 1 \end{array}$$

Thus, the octal representation of  $93_{10}$  is  $135_8$  (reading the remainders in reverse).

**Hexadecimal to Decimal:**

To convert the hexadecimal number  $1A3_{16}$  to decimal.

Hexadecimal:	<u>1A3</u>
Decimal:	$1 \times 16^2 + A \times 16^1 + 3 \times 16^0$
	$1 \times 256 + 10 \times 16 + 3 \times 1$
	$256 + 160 + 3$
	419

Here, A in hexadecimal corresponds to 10 in decimal.

**Decimal to Hexadecimal:**

To convert the decimal number  $291_{10}$  to hexadecimal.

$$\begin{aligned} 291 \div 16 &= 18 \quad \text{remainder } 3 \\ 18 \div 16 &= 1 \quad \text{remainder } 2 \\ 1 \div 16 &= 0 \quad \text{remainder } 1 \end{aligned}$$

Thus, the hexadecimal representation of  $291_{10}$  is  $123_{16}$  (reading the remainders in reverse).

## 1.3 Octal/Hexadecimal to/from Binary

### Bit-Vector Representation Summary

- Digit-vectors for binary, octal, and hexadecimal systems are represented using bit-vectors. In binary, 0 and 1 are directly represented as 0 and 1.
- In systems like octal or hexadecimal, a digit is a bit-vector of length  $k$ , where  $k$  is the number of bits needed to represent the base.

$$k = \log_2(r)$$

with  $r$  the radix of the system (eg. 8 for octal conversion).

- For example, the hexadecimal digit  $B$  is represented as the bit-vector 1101 in binary. *We obtain a length 4 bit-vector because the base is 16 and  $\log_2(16) = 4$*

**Binary to Octal:**

To convert a binary number to octal, group every three binary digits into a single octal digit, because  $k = \log_2 8 = 3$ .

Binary:

010 000 100 110

Octal:

$2_8 0_8 4_8 6_8$

**Binary to Hexadecimal:**

To convert a binary number to hexadecimal, group every four binary digits into a single hexadecimal digit, because  $k = \log_2 16 = 4$ .

Binary:

1011 1110 1010 1101

Hexadecimal:

$B_{16} E_{16} A_{16} D_{16}$

**Octal to Hexadecimal:**

Convert the octal number to binary, then group the binary digits in sets of four and convert each group to its hexadecimal equivalent.

Octal:	<u>257</u>
Binary:	010 101 111 (Octal to binary)
Binary grouped:	<u>0101 0111</u>
Hexadecimal:	5 7 (Binary to hexadecimal)

## 1.4 Representation of Signed Integers

### 1.4.1 Sign-Magnitude Representation (SM)

A signed integer  $x$  is represented by a pair  $(x_s, x_m)$ , where  $x_s$  is the *sign* and  $x_m$  is the *magnitude* (positive integer).

The sign (positive, negative) is represented by the most significant bit (MSB) of the digit vector:

0 → positive

1 → negative

The magnitude can be represented as any positive integer. In a conventional radix- $r$  system, the range of  $n$ -digit magnitude is:

$$0 \leq x_m \leq r^n - 1$$

- Examples:

$$\begin{aligned} 01010101_2 &= +85_{10} \\ 01111111_2 &= +127_{10} \\ 00000000_2 &= +0_{10} \\ 11010101_2 &= -85_{10} \\ 11111111_2 &= -127_{10} \\ 10000000_2 &= -0_{10} \end{aligned}$$

Note: The Sign-and-Magnitude representation is considered a redundant system because both  $0000000_2$  and  $1000000_2$  represent zero.

*SM* consists of an equal number of positive and negative integers.

An  $n$ -bit integer in sign-and-magnitude lies within the range (*because of 0's double representation and that MSB is used for the sign*):

$$[-(2^{n-1} - 1), + (2^{n-1} - 1)]$$

Main disadvantage of SM: complex digital circuits for arithmetic operations (addition, subtraction, etc.).

## 1.5 True-and-Complement (TC)

### 1.5.1 Mapping

A signed integer  $x$  is represented by a positive integer  $x_R$ ,  $C$  is a positive integer called the *complementation constant*.

$$x_R \equiv x \pmod{C}$$

For  $|x| < C$ , by the definition of the modulo function, we have:

$$x_R = \begin{cases} x & \text{if } x \geq 0 \quad (\text{True form}) \\ C - |x| = C + x & \text{if } x < 0 \quad (\text{Complement form}) \end{cases}$$

### 1.5.2 Unambiguous Representation

To have an unambiguous representation, the two regions should not overlap, translating to the condition:  $\forall x, \max|x| < \frac{C}{2}$

### 1.5.3 Converse Mapping

Converse mapping:

$$x = \begin{cases} x_R & \text{if } x_R < \frac{C}{2} \quad (\text{Positive values}) \\ x_R - C & \text{if } x_R > \frac{C}{2} \quad (\text{Negative values}) \end{cases}$$

When  $x_R = \frac{C}{2}$ , it is usually assigned to  $x = -\frac{C}{2}$ .

Asymmetrical representation simplifies sign detection.

## 1.6 Two's Complement System

This is the True-and-Complement system with  $C = 2^n$ , where  $n$  is the number of bits used to represent the integer.

Range is asymmetrical:

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

The representation of zero is unique.

### 1.6.1 Sign Detection in Two's Complement System

Since  $|x| < C/2$  and assuming the sign is 0 for positive and 1 for negative numbers:

$$\text{sign}(x) = \begin{cases} 0 & \text{if } x_R < C/2 \\ 1 & \text{if } x_R \geq C/2 \end{cases}$$

Therefore, the sign is determined from the most-significant bit:

$$\text{sign}(x) = \begin{cases} 0 & \text{if } x_{n-1} = 0 \\ 1 & \text{if } x_{n-1} = 1 \end{cases} \quad \text{equivalent to} \quad \text{sign}(x) = x_{n-1}$$

### 1.6.2 Mapping from Bit-Vectors to Values

The value of an integer represented by a bit-vector  $b_{n-1}b_{n-2}\dots b_1b_0$  can be universally expressed as:

$$\text{Value} = (-2^{n-1} \cdot b_{n-1}) + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

where  $b_{n-1}$  is the MSB (sign bit) and is 0 for non-negative numbers and 1 for negative numbers.

#### Examples

$$X = 011011_2 = 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 2 + 1 = 27_{10}$$

$$X = 11011_2 = -1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -16 + 8 + 2 + 1 = -5_{10}$$

$$X = 10000000_2 = -1 \cdot 2^7 = -128_{10}$$

$$X = 10000011_2 = -1 \cdot 2^7 + 1 \cdot 2^1 + 1 \cdot 2^0 = -128 + 2 + 1 = -125_{10}$$

### 1.6.3 Change of Sign in Two's Complement System

The two's complement system represents negative numbers by inverting the bits of their positive counterparts and adding one. This process is equivalent to subtracting the number from  $2^n$ .

For an  $n$ -bit number  $x$ :

$$z = -x = (\sim x) + 1 = C - x_R$$

where  $(\sim x)$  is the bitwise NOT of  $x$  and  $x_R$  is the decimal representation of  $x$ .

### Examples

Converting  $+17$  to  $-17$ :

$$\begin{aligned} +17_{10} &= 00010001_2 \\ -17_{10} &= \overline{00010001_2} + 1 = 11101111_2 \\ 2^8 - 17 &= 256 - 17 = 239 = 11101111_2 \end{aligned}$$

Converting  $-99$  to  $+99$ :

$$\begin{aligned} -99_{10} &= 10011101_2 \\ +99_{10} &= \overline{10011101_2} + 1 = 01100011_2 \quad (TC) \\ 2^8 - 99 &= 256 - 99 = 157 = 01100011_2 \quad (\text{Substracting } 99 \text{ from } 256) \end{aligned}$$

## 1.7 Range Extension and Arithmetic Shifts

### 1.7.1 Range Extension

Performed when a value  $x$  represented by a digit-vector of  $n$  bits needs to be represented by a digit-vector of  $m$  bits, where  $m > n$ .

$x$  is represented as:

$$\begin{aligned} X &= (X_{n-1}, X_{n-2}, \dots, X_1, X_0) \\ Z &= (Z_{m-1}, Z_{m-2}, \dots, Z_1, Z_0) \end{aligned}$$

## 1.8 Range Extension Algorithm in Sign-and-Magnitude

In sign-and-magnitude system, the range-extension algorithm is defined as:

$$\begin{aligned} z_s &= x_s \quad (\text{sign bit}) \\ Z_i &= 0 \quad \text{for } i = m-1, m-2, \dots, n \\ Z_i &= X_i \quad \text{for } i = n-1, \dots, 0 \end{aligned}$$

Example: Consider  $X = 11010101_2 = -85_{10}$ .

The algorithm extends the range of  $X$  by adding zeros to the left of the most significant bit, preserving the sign bit.

### 1.8.1 Arithmetic Shifts

Two elementary transformations often used in arithmetic operations are scaling (multiplying and dividing) by the radix.

In the conventional radix-2 number system for integers:

Left arithmetic shift: multiplication by 2, expressed as  $z = 2x$ .

Right arithmetic shift: division by 2, expressed as  $z = x/2$ , with rounding towards zero when  $x$  is negative.

### 1.8.2 Left Arithmetic Shift in Sign-and-Magnitude System

Algorithm (assuming the overflow does not occur):

$$\begin{aligned} z_s &= x_s \text{ (sign bit retained)} \\ Z_{i+1} &= X_i, \quad \text{for } i = 0, \dots, n-2 \\ Z_0 &= 0 \text{ (insert zero at the least significant bit)} \end{aligned}$$

**Example:**

Given  $X = 100101101_2 = -45_{10}$ ,

The left arithmetic shift  $SL(X)$  would be  $101011010_2 = -90_{10}$ .

### 1.8.3 Right Arithmetic Shift in Sign-and-Magnitude System

Algorithm:

$$\begin{aligned} z_s &= x_s \text{ (sign bit retained)} \\ Z_{i-1} &= X_i, \quad \text{for } i = 1, \dots, n-1 \\ Z_{n-1} &= 0 \text{ (insert zero at the most significant bit)} \end{aligned}$$

**Example:**

Given  $X = 100101101_2 = -45_{10}$ ,

The right arithmetic shift  $SR(X)$  would be  $100010110_2 = -22_{10}$ .

### 1.8.4 Left Arithmetic Shift in Two's Complement System

Algorithm (assuming that overflow does not occur):

$$\begin{aligned} Z_{i+1} &= X_i, \quad \text{for } i = 0, \dots, n-2 \\ Z_0 &= 0 \text{ (insert zero at the least significant bit)} \end{aligned}$$

Examples:

Given  $X = 00001101_2 = 13_{10}$ ,

The left arithmetic shift  $SL(X)$  is  $00011010_2 = 26_{10}$ .

Given  $Y = 11010101_2 = -11_{10}$ ,

The left arithmetic shift  $SL(Y)$  is  $10101010_2 = -22_{10}$ .

### 1.8.5 Right Arithmetic Shift in Two's Complement System

Algorithm (assuming that overflow does not occur):

$$\begin{aligned} Z_{n-1} &= X_{n-1} \\ Z_{i-1} &= X_i, \quad \text{for } i = 1, \dots, n-1 \end{aligned}$$

The most significant bit (MSB) is duplicated to keep the sign of the number the same.

Examples:

For  $X = 001101_2 = 13_{10}$ ,

the right arithmetic shift is  $SR(X) = 000110_2 = 6_{10}$ .

For  $Y = 11010101_2 = -11_{10}$  (in two's complement),

the right arithmetic shift is  $SR(Y) = 111010_2 = -6_{10}$ .

## 1.9 Hamming Weight and Distance

### 1.9.1 Hamming Weight (HW)

The Hamming weight of a binary sequence is the number of symbols that are equal to one (1s).

For example, the Hamming weight of 11010101 is 5, as there are five 1s in the bit sequence.

### 1.9.2 Hamming Distance (HD)

The Hamming distance between two binary sequences of equal length is the number of positions at which the corresponding symbols are different.

For example, the Hamming distance between 11010101 and 01000111 is 3, as they differ in three positions.

## 1.10 Binary Coded Decimal (BCD)

Binary Coded Decimal (BCD) represents decimal numbers where each decimal digit is encoded as a four-bit binary number. This method allows decimal numbers to be represented in a format that is easy for digital systems to process.

### 1.10.1 BCD Encoding

- In BCD, each of the decimal digits 0 through 9 is represented by a four-bit binary number, ranging from 0000 to 1001.
- Binary values from  $1010_2(10_{10})$  to  $1111_2(15_{10})$  are not used in standard BCD encoding.
- For example, 25 is represented as 0010 0101<sub>2</sub>.

### 1.10.2 Conversion Algorithms

#### From BCD to Decimal

To convert a BCD-encoded number to its decimal representation:

1. Initialize i to the highest index of BCD digits (n-1), D to 0.
2. While i is greater than or equal to 0:
  - a. Multiply D by 10.
  - b. Add the decimal value of the BCD digit at index i to D.
  - c. Decrement i.

#### From Decimal to BCD

To convert a decimal number to its BCD representation:

1. Initialize i to 0, D to the decimal number.
2. While D is not equal to 0:
  - a. Calculate D mod 10 and store it as the current BCD digit.
  - b. Divide D by 10.
  - c. Increment i.

## 1.11 Gray Code Conversion Algorithm

**Rule for Conversion:**

For bit  $i$  in the Gray code, look at bits  $i$  and  $i + 1$  in the binary code (bit  $n$  in binary is zero if  $i + 1 = n$ ).

If bits  $i$  and  $i + 1$  in the binary are the same, bit  $i$  in the Gray code is 0.

If they are different, bit  $i$  in the Gray code is 1.

### 1.11.1 Example

To convert the binary number 1011 to Gray code.

let:  $\underline{1011}_2 = b_3b_2b_1b_0$ .

Apply the conversion rule:

$g_3 = b_3$  since there is no  $b_4$  (assume  $b_4 = 0$ ).

$g_2 = b_3 \oplus b_2$ .

$g_1 = b_2 \oplus b_1$ .

$g_0 = b_1 \oplus b_0$ .

Calculate the Gray code bits:

$g_3 = 1 \oplus 0 = 1$  as  $b_4$  doesn't exist and is thus a 0.

$g_2 = 1 \oplus 0 = 1$ .

$g_1 = 0 \oplus 1 = 1$ .

$g_0 = 1 \oplus 1 = 0$ .

The Gray code is:  $g_3g_2g_1g_0 = \underline{1110}_{\text{gray code}}$ .

# Chapter 2

## Number Systems (Part II)

### 2.1 Addition and Subtraction of Unsigned Integers

*Personal Remark.* In case this is not clear, this video explains it pretty well:  
<https://www.youtube.com/watch?v=sJXTo3EZoxM>

#### 2.1.1 Addition of Binary Numbers

To add binary numbers, follow these rules:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10 \text{ (0 and carry 1 to the next higher bit)}$$

$$1 + 1 + 1 = 11 \text{ (1 and carry 1 to the next higher bit)}$$

#### Example:

Adding  $1101_2$  and  $1011_2$ :

$$\begin{array}{r} 1 & 1 & 1 & 0 & (\text{Carry}) \\ 1 & 1 & 0 & 1 \\ + & 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 \end{array}$$

### 2.1.2 Subtracting Two Binary Numbers

Works exactly like subtracting decimal numbers, but with a borrow of 2 instead of 10. The rules for binary subtraction include:

$$0 - 0 = 0$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

$$0 - 1 = 1 \text{ (with a borrow from the next higher bit)}$$

#### Example:

Subtracting  $1000_2$  from  $1101_2$ :

$$\begin{array}{r} 1 & 1 & 0 & 1 \\ - & 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 \end{array}$$

Therefore, the difference between  $1101_2$  and  $1000_2$  is  $101_2$ .

## 2.2 Overflow and Underflow in Unsigned Binary Arithmetic

### 2.2.1 Overflow

Overflow in unsigned binary arithmetic occurs when the sum of two numbers exceeds the maximum value that can be represented by the given number of bits. For an  $n$ -bit unsigned number, the maximum value that can be represented is  $2^n - 1$ . If the result of an addition is greater than this maximum value, the system experiences overflow, leading to an incorrect result.

**Example:** Consider adding two 4-bit unsigned numbers  $1111_2$  and  $0001_2$ :

$$\begin{array}{r} 1 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array}$$

The result  $10000_2$  is a 5-bit number, but only the 4 least significant bits  $0000_2$  are kept in a 4-bit system, leading to overflow.

## 2.2.2 Underflow

Underflow in unsigned binary arithmetic occurs when the result of a subtraction is less than 0, which is not representable in unsigned arithmetic. Since unsigned numbers cannot represent negative values, any operation that would result in a negative value causes underflow.

**Example:** Consider subtracting a larger 4-bit unsigned number  $1010_2$  from a smaller one  $0100_2$ :

$$\begin{array}{r} 0 \ 1 \ 0 \ 0 \\ - 1 \ 0 \ 1 \ 0 \\ \hline \end{array}$$

underflow

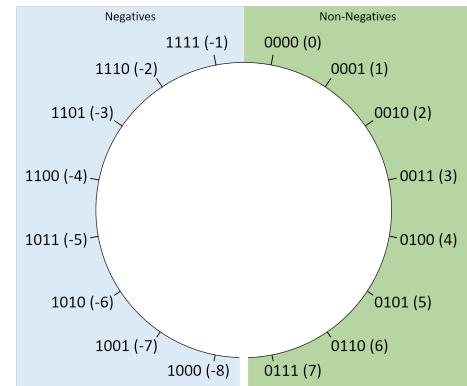
Since the result would be negative, which cannot be represented in unsigned arithmetic, this situation is considered underflow.

## 2.3 Two's Complement Addition and Subtraction

### Graphical Representation

In two's complement arithmetic, a circular graphical representation can be used to illustrate the addition and subtraction of numbers:

- Moving **clockwise** from 0 represents the *addition* of positive numbers.
- Moving **counterclockwise** represents the *subtraction* of positive numbers.
- Crossing the line where the sign changes indicates a *change of sign* from positive to negative or vice versa.



Circular representation of two's complement

### Examples:

The addition of two positive numbers, such as  $2 + 3$ , is shown by moving clockwise from 0 to 2 and then moving 3 more steps clockwise, resulting in 5.

The subtraction of a smaller number from a larger number, such as  $5 - 3$ , is shown by moving clockwise from 0 to 5 and then moving 3 steps counterclockwise, resulting in 2.

### 2.3.1 Addition and Subtraction

#### 2.3.2 Addition

Given two  $n$ -bit numbers  $A$  and  $B$ , their sum in two's complement arithmetic is obtained by directly adding them together as binary numbers:

$$\text{Sum} = A + B \quad (2.1)$$

If there is an overflow, i.e., a carry out of the most significant bit (MSB), it is discarded. The result is also represented in  $n$  bits.

#### 2.3.3 Subtraction

To subtract one  $n$ -bit number  $B$  from another  $A$  using two's complement arithmetic, convert  $B$  to its two's complement and then add it to  $A$ :

1. Find the two's complement of  $B$ , denoted as  $\bar{B}$ , by inverting all the bits of  $B$  and adding 1.
2. Add  $A$  to the two's complement of  $B$ :

$$\text{Difference} = A + \bar{B} \quad (2.2)$$

As with addition, discard any overflow from the MSB.

## 2.4 Binary Multiplication

*Proof.* Let  $X$  and  $Y$  be two numbers, then their product can be represented as:

$$\begin{aligned} X \cdot Y &= X \cdot \left( -Y_{n-1} \cdot 2^{n-1} + X \sum_{i=0}^{n-2} Y_i \cdot 2^i \right) \\ &= -X \cdot Y_{n-1} \cdot 2^{n-1} + X \sum_{i=0}^{n-2} Y_i \cdot 2^i \\ &= -Y_{n-1} \cdot X \cdot 2^{n-1} + Y_{n-2} \cdot X \cdot 2^{n-2} + \dots + Y_2 \cdot X \cdot 2^2 + Y_1 \cdot X \cdot 2^1 + Y_0 \cdot X \cdot 2^0 \end{aligned}$$

□

Binary multiplication operates similarly to decimal multiplication but is performed bit by bit. Here is a clearer example illustrating the multiplication of two binary numbers:

Multiplicand	1101	(This is the number to be multiplied)
Multiplier	$\times$ 0011	(This number multiplies the multiplicand)
	1101	(Multiply by 1, the least significant bit of the multiplier)
+	11010	(Multiply by 1, add one zero to the right, (left shift, $<< 1$ ))
+	000000	(Multiply by 0, add two zeros to the right, (left shift, $<< 2$ ))
+	0000000	(Multiply by 0, add three zeros to the right, (left shift, $<< 3$ ))
	100111	(Sum of the partial products)

# Chapter 3

## Number Systems (Part III)

### 3.1 Fixed-Point Number Representation

Let  $x$  be an integer :  $x = x_{int} + x_{fr}$ , with  $x_{int}$  the integer part and  $x_{fr}$  the fractional part.  
Let  $X$  be a digit-vector:

$$X = (X_{m-1} X_{m-2} \dots X_1 X_0 . X_{-1} X_{-2} \dots X_{-f})$$

where

$X_{m-1}$  to  $X_0$  represent the integer component

$X_{-1}$  to  $X_{-f}$  represent the fractional component

The dot (.) represents the radix point (assumed to be fixed)

For unsigned numbers:

$$x = \sum_{i=-f}^{m-1} X_i \cdot 2^i$$

For signed numbers in two's complement:

$$x = -X_{m-1} \cdot 2^{m-1} + \sum_{i=-f}^{m-2} X_i \cdot 2^i$$

### 3.1.1 Examples of Fixed-Point Numbers

#### Decimal Numbers

Decimal number system with  $m = 5, f = 5$

Example decimal digit vector

$$\begin{aligned} x &= (10077.01690) \\ x &= 1 \cdot 10^4 + 7 \cdot 10^1 + 7 \cdot 10^0 + 1 \cdot 10^{-2} + 6 \cdot 10^{-3} + 9 \cdot 10^{-4} \\ &= 10000 + 70 + 7 + 0.01 + 0.006 + 0.0009 \\ &= 10077.0169 \end{aligned}$$

Most negative (min):

$$x_{\min} = -99999.99999 = \frac{-999999999}{10^5}$$

Largest number (max, positive):

$$x_{\max} = +99999.99999 = \frac{+999999999}{10^5}$$

#### Unsigned Binary Numbers

Unsigned with  $m = 3, f = 4$

Example binary digit vector

$$X = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 5.4375$$

Smallest number (min):

$$x_{\min} = 000.0000_2 = 0$$

Largest number (max):

$$x_{\max} = 111.1111_2 = 7 + \frac{15}{16} = 7,9375$$

#### Sign-and-Magnitude Binary Numbers

Sign-and-magnitude with  $m = 5, f = 3$

Example binary digit vector

$$X = (-1)^1 \cdot (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3}) = -5.75$$

Most negative number (min):

$$x_{\min} = 11111.111_2 = -15 + \frac{7}{8} = -15.875$$

Largest number (max, positive):

$$x_{\max} = 01111.111_2 = 15 + \frac{7}{8} = 15.875$$

## 3.2 Concepts of Finite Precision Math

### 3.2.1 Precision

Let  $X$  be a digit-vector:

$$X = (X_{m-1} X_{m-2} \dots X_1 X_0 . X_{-1} X_{-2} \dots X_{-f})$$

Precision is the maximum number of non zero bits.

$$\text{Precision}(x) = m + f$$

with  $m$  the number of bits for the integer part and  $f$  the number of bits for the fractional part.

### 3.2.2 Resolution

Resolution is the smallest non-zero number that can be represented.

$$\text{Resolution}(x) = 2^{-f}$$

### 3.2.3 Range

The range of a fixed-point number is the difference between the largest and smallest numbers that can be represented.

$$\text{Range}(x) = x_{\max} - x_{\min}$$

In two's complement, the range is given by:

$$\text{Range}(x) = \sum_{i=-f}^{m-2} 2^i - (-2^{m-1})$$

### 3.2.4 Accuracy

Accuracy is the maximum difference between a real value and the represented value.

$$\text{Accuracy}(x) = \frac{\text{Resolution}(x)}{2}$$

### 3.2.5 Dynamic Range

The dynamic range is the ratio of the largest and the smallest positive number that can be represented.

$$\text{Dynamic Range}(x) = \frac{|x_{\max}|}{|x_{\text{positive, nonzero min}}|}$$

*In two's complement :*

$$\text{Dynamic Range}(x) = \frac{2^{m-1}}{2^{-f}} = 2^{m-1+f}$$

### 3.3 Floating-Point Number Representation

*Personal Remark.* Please take the time to really understand this part. It is crucial for the understanding of the rest of the course. Take some time to understand the vocabulary and its meaning.

A real number that is exactly representable in a computer is called a **floating-point number**.

#### 3.3.1 Significand, Base, Exponent

A floating-point number consists of a **significand** (or mantissa), a **base** (or radix), and an **exponent**.

the signed *significand* (also called *mantissa*)  $M^*$

the signed *exponent*  $E$

where  $b$  is a constant called the *base*

$$x = M^* \times b^E$$

This reminds us of the usual scientific notation, base 10:

$+35200 = +3.52 \times 10^4$  (Coefficient)

$-0.099 = -9.9 \times 10^{-2}$  (Exponent)

#### 3.3.2 Benefits

1. Consider 32-bit two's complement signed integers

The dynamic range for a 32-bit two's complement signed integer can be expressed as:

$$\text{Dynamic Range}_1(x) = \frac{|x|_{\max}}{|x_{\text{positive, nonzero}}|_{\min}} = \frac{|-2^{32-1}|}{2^0} = 2^{31} \approx 2 \cdot 10^9$$

2. Consider 32-bit floating-point number, with 24-bits significand and 8-bit exponent in Sign and Magnitude

$$\text{Dynamic Range}_2(x) = \frac{|x|_{\max}}{|x_{\text{positive, nonzero}}|_{\min}}$$

$$= \frac{(2^{23} - 1) \cdot 2^{(2^{8-1})-1}}{2^{-128}} = \frac{(2^{23} - 1) \cdot 2^{127}}{2^{-128}} = (2^{23} - 1) \cdot 2^{255} \approx 5 \cdot 10^{83}$$

3. Dynamic range increase

Comparing the two dynamic ranges:

$$\frac{\text{Dynamic Range}_2(x)}{\text{Dynamic Range}_1(x)} \approx 10^{74}$$

1. Consider a fixed-point number with 8 fractional bits

$$\text{Resolution}_1(x) = 2^{-8}$$

2. Consider 32-bit floating-point number, with 24-bits significand in sign-and-magnitude and 8-bit exponent in two's complement

$$\text{Resolution}_2(x) = 2^0 \cdot 2^{-2(8-1)} = 2^{-2^7} = 2^{-128}$$

3. Improved resolution

$$\frac{\text{Resolution}_2(x)}{\text{Resolution}_1(x)} = \frac{2^{-128}}{2^{-8}} = 2^{-120}$$

### 3.3.3 Representation

We will be focusing on the following digit-vector:

$$X = (\underbrace{S}_{\text{Sign}} \underbrace{E_{m-1}E_{m-2}\dots E_1E_0}_{\text{Exponent}} \underbrace{M_{n-1}M_{n-2}\dots M_0}_{\text{Magnitude}})$$

The floating-point representation becomes

$$x = (-1)^s \times M \times b^E$$

where  $s \in \{0, 1\}$  is the **sign**, and  $M$  is the **magnitude** of the signed significant

**In the rest of the lecture, we assume significand is represented in sign-and-magnitude.**

### Normalization

The goal of normalization is to represent the number such that the magnitude ( $M$ ) is within the range  $1 \leq M < 2$ . This means that the leading digit before the binary point is always 1. Here are examples to demonstrate this process:

#### Positive Example:

Given:  $+1010.1000_2$

Normalize:  $+1.1010_2 \times 2^3$

Decimal Conversion:  $1.3125 \times 8 = 10.5$

Explanation: The binary number  $1010.1000_2$  is normalized by shifting the binary point such that the first digit is 1, and adjusting the exponent ( $2^3$ ) accordingly.

The equivalent decimal number is 10.5.

**Negative Example:**

Given:  $-(0.00000011)_2$

Normalize:  $-1.1_2 \times 2^{-7}$

Decimal Conversion:  $-(1.5)_{10} \times 2^{-7} = -0.01171875$

Explanation: The binary number  $0.00000011_2$  is normalized by shifting the binary point to get a leading 1, and adjusting the exponent ( $2^{-7}$ ) to reflect the shift. The equivalent decimal number is  $-0.01171875$ .

**Why Normalize?**

Normalization removes redundancy from floating-point representation, making it unique. Consider these examples to understand the redundancy in non-normalized representations:

$$\begin{aligned} + (1010)_2 \times 2^{-2} &= 10 \times 2^{-2} = 2.5; \\ + (1.01)_2 \times 2^1 &= 1.25 \times 2^1 = 2.5; \\ + (0101)_2 \times 2^{-1} &= 5 \times 2^{-1} = 2.5; \end{aligned}$$

In these cases, different representations lead to the same decimal value, illustrating redundancy. Normalization ensures that each number has a unique floating-point representation.

**Conclusion:** Floating-point representation is **redundant unless it is normalized**. By normalizing, we ensure a unique and efficient representation for computational purposes.

In normalized floating-point representation, the leading digit is always 1 and is omitted as a hidden bit to save space, with the remaining digits representing the fraction part of the significand :

$$+(101.001)_2 \times 2^{-4} \Rightarrow +(1.01001)_2 \times 2^{-2} \Rightarrow +(.01001)_2 \times 2^{-2}$$

### 3.3.4 Biased Representation

Given a binary number with  $n$  bits, the value of the biased representation can be calculated as follows:

$$x = \left( \sum_{i=0}^{n-1} X_i \cdot 2^i \right) - B$$

Where  $X_i$  represents the  $i^{th}$  bit of the binary number (starting from 0 for the least significant bit), and  $B$  is the bias, which is calculated by:

$$B = 2^{(n-1)} - 1$$

The exponent thus becomes :

$$e = \left( \sum_{i=0}^{n-1} E_i \cdot 2^i \right) - (2^{(n-1)} - 1)$$

For instance, with a 3-bit binary number, the bias  $B$  is  $2^{(3-1)} - 1 = 3$ . Therefore, the biased representation maps binary numbers to the integer range from  $-B$  to  $2^n - 1 - B$ .

#### Example

For a 3-bit binary number, the biased representations would be:

Decimal	Binary	Biased Decimal
7	111	4
6	110	3
5	101	2
4	100	1
3	011	0
2	010	-1
1	001	-2
0	000	-3

Note that the minimum exponent in the biased representation is zero so that the representation of FP zero value is all zeros (zero sign, exponent, and mantissa).

## Summary of the Floating-Point Representation

Let the binary vector :

$$X = (SE_{m-1}E_{m-2}\dots E_1E_0 . M_{n-1}M_{n-2}\dots M_0)$$

( $m$ )-bit exponent

- Biased,  $B = 2^{m-1} - 1$

( $n + 1$ )-bit significand

- Sign-and-magnitude
- Normalized, one hidden bit

$$x = (-1)^S \times \left(1 + \sum_{i=1}^n M_{n-i} 2^{-i}\right) \times 2^{\left(\sum_{j=0}^{m-1} E_j 2^j\right) - (2^{m-1} - 1)}$$

### 3.3.5 Rounding

The result of a floating-point operation is a real number that, to be represented exactly, might require a significand with an infinite number of digits.

For a representation close to the exact result, we perform **rounding**.

Consider the real number  $x_{\text{real}}$  and the consecutive floating-point numbers  $F_1$  and  $F_2$ , such that  $F_1 \leq x_{\text{real}} \leq F_2$ .

We can perform several types of rounding:

- Round to nearest (tie to even)
- Round towards zero (truncation)
- Round towards plus or towards minus infinity

- Round to nearest (tie to even)

$$R_{\text{near}}(x_{\text{real}}) = \begin{cases} F_1, & \text{if } |x_{\text{real}} - F_1| < |x_{\text{real}} - F_2| \\ F_2, & \text{if } |x_{\text{real}} - F_1| > |x_{\text{real}} - F_2| \\ \text{even}(F_1, F_2), & \text{if } |x_{\text{real}} - F_1| = |x_{\text{real}} - F_2| \end{cases}$$

- Round towards zero (truncation)

$$R_{\text{zero}}(x_{\text{real}}) = \begin{cases} F_1, & \text{if } x_{\text{real}} \geq 0 \\ F_2, & \text{if } x_{\text{real}} < 0 \end{cases}$$

- Round towards plus or minus (negative) infinity

$$R_{\text{pinf}}(x_{\text{real}}) = F_2$$

$$R_{\text{ninf}}(x_{\text{real}}) = F_1$$

## Examples of Rounding Methods

Let's consider  $x_{\text{real}} = 2.5$ ,  $F_1 = 2$ , and  $F_2 = 3$  for our examples.

### Round to nearest (tie to even)

$$R_{\text{near}}(2.5) = \text{even}(2, 3) = 2$$

Since both  $F_1$  and  $F_2$  are equidistant from  $x_{\text{real}}$ , we choose the even number which is 2.

### Round towards zero (truncation)

$$R_{\text{zero}}(2.5) = 2$$

Since  $x_{\text{real}}$  is positive, we round towards zero, resulting in  $F_1$ .

### Round towards plus infinity

$$R_{\text{pinf}}(2.5) = 3$$

When rounding towards plus infinity, we choose  $F_2$ .

### Round towards minus infinity

$$R_{\text{ninf}}(2.5) = 2$$

When rounding towards minus infinity, we choose  $F_1$ .

Now let's consider a negative value  $x_{\text{real}} = -2.5$ ,  $F_1 = -3$ , and  $F_2 = -2$ .

### Round towards zero (truncation) for negative value

$$R_{\text{zero}}(-2.5) = -2$$

Since  $x_{\text{real}}$  is negative, we round towards zero, resulting in  $F_2$ .

### Round towards plus infinity for negative value

$$R_{\text{pinf}}(-2.5) = -2$$

When rounding towards plus infinity for a negative number, we choose the larger number, which is  $F_2$ .

### Round towards minus infinity for negative value

$$R_{\text{ninf}}(-2.5) = -3$$

When rounding towards minus infinity for a negative number, we choose the smaller number, which is  $F_1$ .

## 3.4 IEEE 754 Standard

### FP Format in IEEE 754

Exactly what we described:

$(n + 1)$ -bit significand

- Sign-and-magnitude
- Normalized, one hidden bit

$m$ -bit exponent

- Biased,  $B = 2^{m-1} - 1$

Let  $X$  a digit-vector represented as:

$$X = (SE_{m-1}E_{m-2}\dots E_1E_0.M_{n-1}M_{n-2}\dots M_1M_0)$$

Basic and extended formats:

+ Basic formats:

- Single precision (32 bits)
  - \* Sign  $S$ : 1 bit
  - \* Exponent  $E$ : 8 bits
  - \* Fraction  $F$ : 23 bits
- Double precision (64 bits)
  - \* Sign  $S$ : 1 bit
  - \* Exponent  $E$ : 11 bits
  - \* Fraction  $F$ : 52 bits

+ Default round to nearest (ties to even)

### 3.4.1 Special Values

The IEEE 754 standard defines special values with unique bit patterns:

**Floating-point zero:** is represented by all zeros in both the exponent and the significand fields.

Positive zero: 0	00000000	00000000000000000000000000000000
Negative zero: 1	00000000	00000000000000000000000000000000

The most significant bit (the sign bit) differentiates between positive and negative zero.

**Positive and negative infinity:** are represented by all ones in the exponent field and all zeros in the significand field.

Positive infinity: 0	11111111	00000000000000000000000000000000
Negative infinity: 1	11111111	00000000000000000000000000000000

**NaN (Not a Number):** is represented by all ones in the exponent field and a non-zero significand field.

NaN (example): – 11111111 10000000000000000000000000000000

NaN values represent indeterminate or undefined results, such as the square root of a negative number. The sign bit can be either 0 or 1, but the significand must not be all zeros.

### 3.4.2 Overflow, underflow, and others

**Overflow:** Occurs when the rounded value is too large to be represented by the floating-point format.

- The result is set to positive or negative infinity, depending on the sign.

**Underflow:** Happens when the rounded value is too small to be represented.

- Typically, the result is set to a denormalized number or zero.

**Division by zero:** Occurs when a finite non-zero number is divided by zero.

- The result is set to positive or negative infinity, based on the sign of the numerator.

**Inexact result:** Occurs when the result of an operation is not an exact floating-point number.

- The result is rounded to the nearest representable value.

**Invalid:** This flag is set when the result of an operation is not a real number (NaN).

- Examples include the square root of a negative number or the indeterminate form 0/0.

## IEEE 754

### Example: Converting single-precision FP to decimal

Find the decimal equivalent of

$$X = (1\ 01111100\ 0100000000000000000000000000)$$

Solution:

- Sign  $S = 1$ , hence negative.
- Exponent  $E = 01111100$  represents the biased exponent. To find the actual exponent:

$$E_{\text{actual}} = 124 - (2^7 - 1) = 124 - 127 = -3$$

- Mantissa (including the hidden bit)  $M = 1.01$  in binary represents:

$$M = 1 + 2^{-2} = 1.25$$

- Result:

$$x = -1.25 \times 2^{-3} = -0.15625$$

### Example: Converting decimal to single-precision FP

Find the single-precision FP equivalent of  $x = -0.8125$

Solution:

- Sign = 1, negative.
- Fraction bits can be obtained using multiplication by 2.
- Converting 0.8125 to binary by successive multiplication:

$$\begin{aligned} 0.8125 \times 2 &= 1.625 \rightarrow 1 \\ 0.625 \times 2 &= 1.25 \rightarrow 1 \\ 0.25 \times 2 &= 0.5 \rightarrow 0 \\ 0.5 \times 2 &= 1.0 \rightarrow 1 \end{aligned}$$

Stop when the fractional part becomes zero.

- Mantissa  $M$  in binary is 0.1101, normalized is  $1.101 \times 2^{-1}$ .
- Exponent adjustment:

$$E_{\text{actual}} = -1$$

$$E = E_{\text{actual}} + B = -1 + (2^7 - 1) = 126$$

- Result:

$$X = (1\ 0111110\ 1010000000000000000000000000)$$

# Chapter 4

## Number Systems (Part IV)

### 4.1 Fixed-Point Arithmetic

#### 4.1.1 Addition and Subtraction

Let  $x$  and  $y$  be two fixed-point numbers with the same number of integer and fractional bits. The sum and difference of  $x$  and  $y$  can be calculated as follows:

$$x + y = x_{int} + y_{int} + x_{fr} + y_{fr}$$

$$x - y = x_{int} - y_{int} + x_{fr} - y_{fr}$$

#### 4.1.2 Multiplication

010.11	Multiplicand	
$\times$	011.01	Multiplier
<hr/>	000000	First partial product (always zero), sign-extended
+	001011	1 x multiplicand, sign-extended
<hr/>	001011	Intermediate result, sign-extended
+	000000	0 x multiplicand, left-shifted by 1 place and sign-extended
<hr/>	00001011	Intermediate result, sign-extended
+	001011	1 x multiplicand, left-shifted by 2 places and sign-extended
<hr/>	00010111	Intermediate result, sign-extended
+	001011	1 x multiplicand, left-shifted by 3 places and sign-extended
<hr/>	001001111	Intermediate result, sign-extended
+	000000	0 x multiplicand, left-shifted by 4 places and sign-extended
<hr/>	001000.1111	Result

convert to fixed-point now

## 4.2 In two's complement

Given two fixed-point numbers  $x$  and  $y$ , the addition or subtraction in two's complement is given by:

$$x \pm y = \left( -X_{(m_x-1)} 2^{(m_x-1)} + \sum_{i=-f_x}^{m_x-2} X_i 2^i \right) \pm \left( -Y_{(m_y-1)} 2^{(m_y-1)} + \sum_{i=-f_y}^{m_y-2} Y_i 2^i \right) \quad (4.1)$$

Where:

$m_x$  and  $m_y$  are the total number of bits for the integer components of  $x$  and  $y$  respectively.

$f_x$  and  $f_y$  are the number of bits for the fractional components of  $x$  and  $y$  respectively.

$X_{(m_x-1)}$  and  $Y_{(m_y-1)}$  are the sign bits of  $x$  and  $y$ .

The largest integer-part exponent is  $\max(m_x - 1, m_y - 1)$ . Consequently, the number of bits for the resulting integer component is  $\max(m_x, m_y) + 1$ .

The smallest fractional-part exponent is  $\min(-f_x, -f_y)$ . Consequently, the number of bits for the resulting fractional component is  $\max(f_x, f_y)$ .

## Multiplication in Two's Complement

Multiplication on two binary numbers  $x(m_x, f_x)$  and  $y(m_y, f_y)$

$$x \cdot y = (x_{\text{int}} + x_{\text{fr}}) \cdot (y_{\text{int}} + y_{\text{fr}})$$

In two's complement:

$$x \cdot y = \left( -X_{m_x-1} 2^{m_x-1} + \sum_{i=-f_x}^{m_x-2} X_i 2^i \right) \cdot \left( -Y_{m_y-1} 2^{m_y-1} + \sum_{i=-f_y}^{m_y-2} Y_i 2^i \right)$$

The largest integer-part exponent:  $(m_x - 1) + (m_y - 1)$ . Consequently:  $m_{xy} = m_x + m_y$

The smallest fractional-part exponent:  $(-f_x) + (-f_y)$ . Consequently:  $f_{xy} = f_x + f_y$

## 4.3 Floating-Point Arithmetic

Let  $x$  and  $y$  be represented as  $(S_x, M_x, E_x)$  and  $(S_y, M_y, E_y)$

The significands  $M^* = (-1)^S M$  are normalized

Addition/subtraction result is  $z$ , also represented as  $(S_z, M_z, E_z)$ :

$$z = x \pm y = M_x^* \times 2^{E_x} \pm M_y^* \times 2^{E_y}$$

The significand of the result is also normalized

$$z = M_z^* \times 2^{E_z}$$

**Four main steps to compute the result of floating-point addition/subtraction:**

**1. Add/Subtract significand and set exponent:**

- Align the significands by shifting the one with the *smaller* exponent.
- Perform addition/subtraction on the aligned significands.

$$M_z^* = \begin{cases} (M_x^* + (M_y^* \times 2^{(E_y - E_x)})) \times 2^{E_x} & \text{if } E_x \geq E_y \\ ((M_x^* \times 2^{(E_x - E_y)}) + M_y^*) \times 2^{E_y} & \text{if } E_x < E_y \end{cases}$$

$$E_z = \max(E_x, E_y)$$

**2. Normalize the result and update the exponent, if required:**

- Check if the result's significand is within the normalized range.
- If not, shift the significand to the right or left until it is normalized, adjusting the exponent accordingly to maintain the value.

**3. Round the result, normalize, and adjust exponent, if required:**

- Apply rounding rules to the significand to fit within the precision limits.
- After rounding, if the significand overflows (e.g., carries out during addition), normalize the result again and adjust the exponent.

**4. Set flags for special values, if required:**

- Check for overflow or underflow conditions and set flags accordingly.
- Identify and mark results that are special values (e.g., infinity, NaN) based on the operation and input values.

## Step 1: Floating-Point Addition/Subtraction Detailed Algorithm

Algorithm steps:

- Subtract exponents  $d = E_x - E_y$ .
- Align significands:
  - \* Compare the exponents of the two operands.
  - \* Shift right  $d$  positions the significand of the operand with the smallest exponent.
  - \* Select as the exponent of the result the largest exponent.
- Add/subtract signed significands and produce the sign of the result.

## Step 2: Normalize the Result

After the initial addition or subtraction, the result may not always be in the normalized form required by floating-point representation standards. Normalization ensures that the significand (mantissa) is within a specific range, usually just below 1 (for binary floating-point numbers, this means the leading bit is just to the right of the decimal point).

Floating-point operations based on the signs of the operands.

FP operation	Signs of the operands	Effective operation
+	=	add
+	≠	subtract
-	=	subtract
-	≠	add

Steps for normalization:

- If the result of the operation causes the significand to exceed its predefined size (overflow), the significand is shifted to the right, and the exponent is increased accordingly.
- Conversely, if the operation results in a significand that's too small (underflow), the significand is shifted to the left, and the exponent is decreased.
- This process ensures that the floating-point number is as close to its true value as possible within the limits of the representation.

## Step 3: Round the Result

After normalization, the next step is to round the result to fit within the target floating-point format's precision. Rounding is crucial because it affects the accuracy and representation of the result.

Steps for rounding:

- Evaluate the significand's precision and compare it with the format's limit.
- If the significand exceeds the precision limit, round it according to a rounding rule (e.g., round to nearest, round towards zero, round towards positive/negative infinity).
- Common rounding strategies include:
  - \* **Round to Nearest:** Round to the nearest value, with ties going to the nearest even number.
  - \* **Round Down (Towards Zero):** Always round towards zero, truncating any fractional part.
  - \* **Round Up (Away from Zero):** Always round away from zero, increasing the magnitude of the result.
- After rounding, if there's an overflow in the significand (e.g., a carry into a new digit), normalize the result again. This may involve shifting the significand and adjusting the exponent.

## Step 4: Set Flags for Special Values

The final step in floating-point addition or subtraction involves handling special cases and setting flags accordingly. Special values include infinity, not-a-number (NaN), and potential overflow or underflow conditions.

Handling special values:

- **Infinity:** If the result of the operation is too large to be represented in the given floating-point format, set the result to infinity. The sign of infinity depends on the operation and operands.
- **NaN (Not-a-Number):** If the operation involves invalid operations (e.g.,  $0/0$ ,  $\infty - \infty$ ), set the result to NaN. NaN propagates through most floating-point operations.
- **Overflow:** If the result exceeds the maximum representable value, set an overflow flag. The result is typically set to infinity with the appropriate sign.
- **Underflow:** If the result is too small to be represented (closer to zero than the smallest representable value), set an underflow flag. The result may be set to zero or the smallest denormalized number, depending on the format and flags.

### 4.3.1 An Example in Binary

Let's add two binary floating-point numbers:  $1.01 \times 2^3$  and  $1.1 \times 2^2$ . Here's how we do it step by step:

1. **Line Up the Dots:** First, we need to align the exponents. We'll adjust the second number to have the same exponent as the first, by increasing its exponent and shifting its significand to the right:

$$1.1 \times 2^2 = 0.11 \times 2^3$$

Now, both numbers are  $1.01 \times 2^3$  and  $0.11 \times 2^3$ .

2. **Add Them Up:** With the exponents aligned, we can now add the significands:

$$1.01 + 0.11 = 10.00$$

The result in binary is 10.00. Since we're working in binary, 10.00 is actually 2 in decimal.

3. **Make It Look Right:** The result  $10.00 \times 2^3$  is already in the correct format, but let's note that if our result was something like  $1.000 \times 2^4$ , we would need to adjust it to keep it in normalized form.
4. **Round It Off:** Our result doesn't need rounding in this case, but if we had more digits than we could store, we'd round off to the nearest value we could represent.
5. **Check for Special Cases:** There are no special cases here, as our result is a regular binary floating-point number.

So, our final result is  $10.00 \times 2^3$  in binary, which is 8 in decimal.

# Chapter 5

## Number Systems (Part V)

This small chapter concludes the Number Systems chapter with some modern applications of low precision computing.

### 5.1 Low precision computer arithmetic

AI is taking on an increasingly important role

Deep Neural Networks (DNNs) are the most widespread

E.g., Large Language Models (LLM) generate human-like content

Challenge: exponential size growth

GPT3 has 175 billion parameters

Large models mean

A lot of data, many computations

...and we want the result quickly!

Luckily, ML models are tolerant to small errors

### 5.2 Challenges and limitations

32-bit or 64-bit floating-point formats

Arithmetic units are large (many bits  $\Rightarrow$  high area, high energy)

We can put fewer units per chip (e.g., less compute power in GPU)

Poor arithmetic density (in number of ops / 1mm<sup>2</sup>)

Fewer units, fewer computations

The model predictions are accurate, but it takes a long time to compute them

Fixed-point or integer formats

Arithmetic units are smaller and faster (approx. 10x area savings)

Better arithmetic density and lower delays

The errors due to limited dynamic range are too significant for most ML models; the accuracy of their predictions suffers

New number formats are needed: the best of both worlds

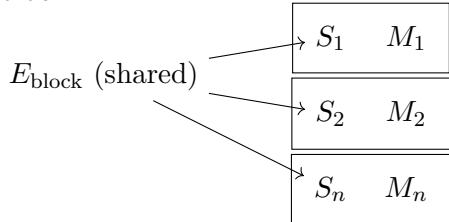
## 5.3 Block Floating Point

### Block Floating Point

Imagine a block (vector) of binary numbers in FP (Floating Point) format, where each vector element has its own S (Sign), M (Mantissa), and E (Exponent).

$S_1$	$E_1$	$M_1$
$S_2$	$E_2$	$M_2$
$\vdots$	$\vdots$	$\vdots$
$S_n$	$E_n$	$M_n$

If the exponents in the block are not too different, we could use a single shared exponent per block.



*In BFP with shared Exponent:*

Find the largest exponent in the block of FP numbers. This will be the shared exponent  $E_{\text{block}}$ .

Calculate the difference  $d_i = E_{\text{block}} - E_i$  between the shared exponent and each of the other exponents  $E_i$  in the block.

Adjust the mantissa by right-shifting the signed mantissa of each number by  $d_i$ . As a result of these adjustments, the mantissa in BFP cannot be normalized, and there is no hidden bit.

# Chapter 6

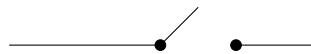
## Digital Logic

### 6.1 Introduction to Digital Logic Circuits

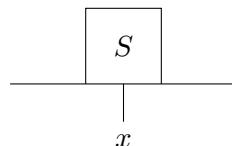
The smallest unit of Digital Information is a binary value 1 or 0.

#### 6.1.1 The simplest binary logic element

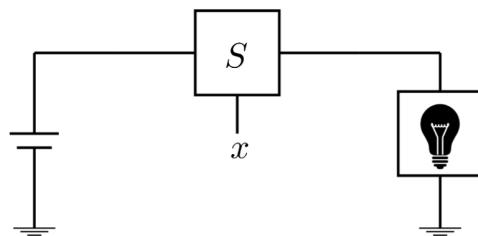
Similarly, a switch can either be open or closed. Let  $x$  an input variable.

- Open ( $x = 0$ ) 
- Closed ( $x = 1$ ) 

The symbol for a switch controlled by an input variable:



#### 6.1.2 The simplest binary logic element

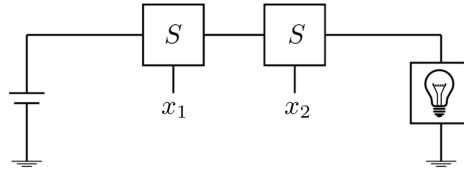


Light controlled by a single switch

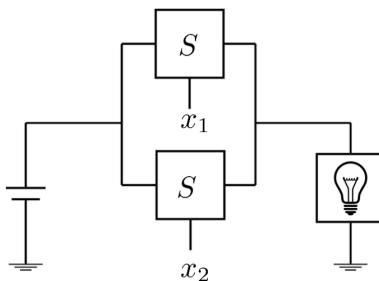
## Two-Variable Logic Functions

### Series and Parallel Connections

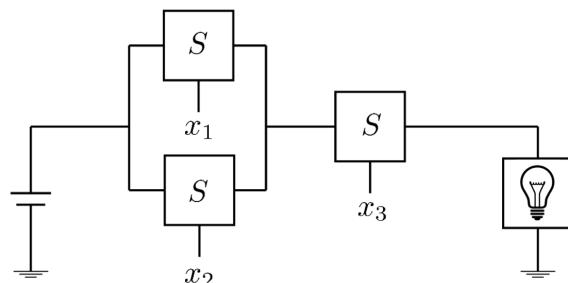
*Remark.* The choice of symbols is not random  $\cdot, +$ , respectively similar to multiplication and addition.



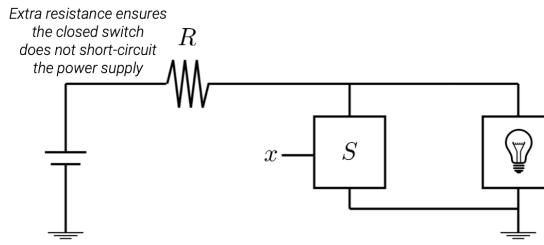
Logic AND function  
 $L(x_1, x_2) = x_1 \cdot x_2$ , with  $\cdot$  the AND operator



Circular representation of two's complement  
 $L(x_1, x_2) = x_1 + x_2$ , with  $+$  the OR operator



A series-parallel connection of three switches  
The corresponding Logic Function is  $L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$   
*Smaller circuits separated by parentheses*



NOT GATE

The corresponding Logic Function is  $L(x) = \bar{x}$ , with  $\bar{x}$  the complement of  $x$ .

## 6.2 Truth Tables

Logical operations can be defined in the form of a truth table

AND

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

where  $L = 1$  if  $x_1 = 1$  and  $x_2 = 1$ ,  $L = 0$  otherwise.

OR

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

where  $L = 1$  if  $x_1 = 1$  or  $x_2 = 1$ , or if  $x_1 = x_2 = 1$ ,  $L = 0$  if  $x_1 = x_2 = 0$ .

$x_1$	$x_2$	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

For  $n$  logical variables we get  $2^n$  rows.

**For example :**

AND

$$L(x_1, x_2, x_3) = x_1 \cdot x_2 \cdot x_3$$

where  $L = 1$  if  $x_1 = x_2 = x_3 = 1$ ,  $L = 0$  otherwise.

OR

$$L(x_1, x_2, x_3) = x_1 + x_2 + x_3$$

where  $L = 0$  if  $x_1 = x_2 = x_3 = 0$ ,  $L = 1$  otherwise.

$x_1$	$x_2$	$x_3$	AND	OR
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

NOT, with  $L(x) = \bar{x}$

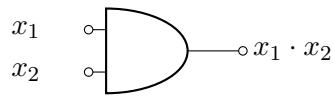
$x$	$\bar{x}$	NOT
0	0	0
0	1	0

## Precedence Table for Logic Operations

Precedence (Priority)	Operator	Operation	Description
1	$\bar{x}$	NOT	Negation of A
2	$x_1 \cdot x_2$	AND	Conjunction of A and B
3	$x_1 + x_2$	OR	Disjunction of A and B

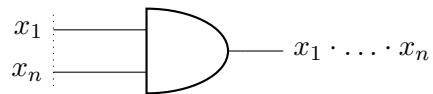
## 6.3 Logic Gates

### 6.3.1 AND GATE



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

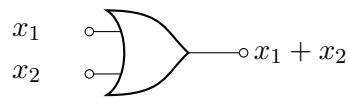
### 6.3.2 AND GATE (n-variables)



$$f(x_1, \dots, x_n) = x_1 \cdot \dots \cdot x_n$$

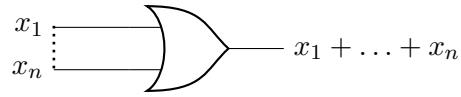
### 6.3.3 OR GATE

True if at least one input is true.



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

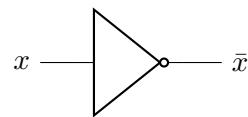
### 6.3.4 OR GATE (n-variables)



$$f(x_1, \dots, x_n) = x_1 + \dots + x_n$$

### 6.3.5 NOT GATE

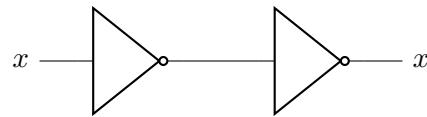
True if input is false, and vice versa.



$$f(x) = \bar{x}$$

### 6.3.6 DOUBLE NOT GATE (BUFFER)

A buffer is a gate that does not change the input signal.



$$f(x) = x = \bar{\bar{x}}$$

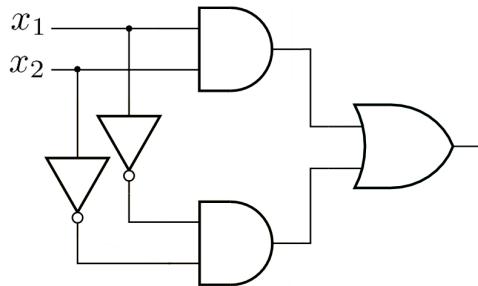
### 6.3.7 NAND GATE



### 6.3.8 NOR GATE



### 6.3.9 Example of Complex Logic Circuit



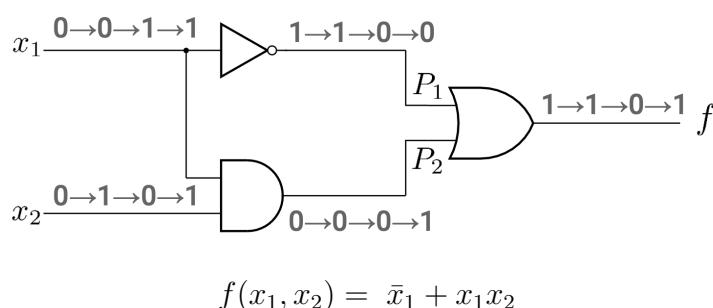
$$f(x_1, x_2) = x_1 x_2 + \bar{x}_1 \bar{x}_2$$

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

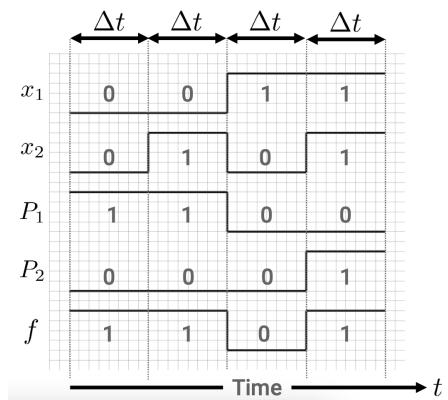
Corresponding logic table

## 6.4 Analysis of a Logic Network

Here we'll be looking at sequences of 0s and 1s sent by the input variables. :



$$f(x_1, x_2) = \bar{x}_1 + x_1 x_2$$



Timing diagram representing the variations in electrical current magnitude over time

## Checking for Equivalence of Logic Networks

Two logic networks, represented by functions  $f$  and  $g$ , are equivalent if:

Their truth tables are identical, which is a verification through perfect induction.

A sequence of algebraic manipulations using Boolean algebra can transform one logic expression into the other.

Their Venn diagrams are the same, providing a simple visual aid for equivalence.

## Finding the Best Equivalent Network (Out of Scope)

The best equivalent logic network is the simplest and cheapest in terms of the number of logic gates used. The process of finding the best equivalent expression is called *minimization* and can be achieved through:

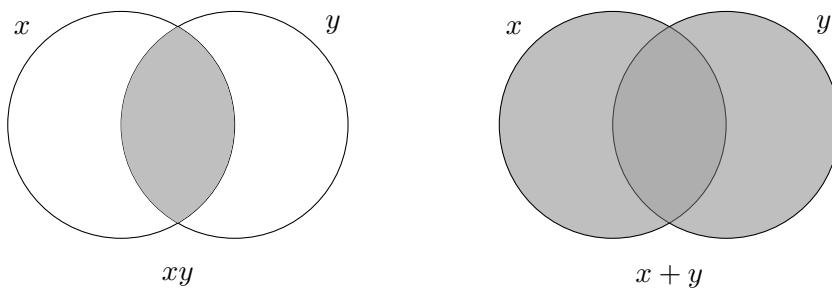
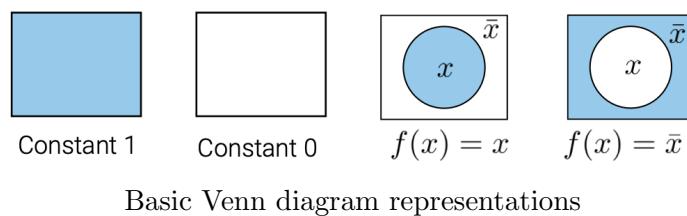
A sequence of algebraic transformations, although it's not always clear which transformations to apply.

Using the Karnaugh map, which is simpler than a truth table but becomes unmanageable by hand for more than 4 inputs.

Automated techniques in synthesis tools (software) which can handle more complex networks efficiently.

## 6.5 The Venn Diagram

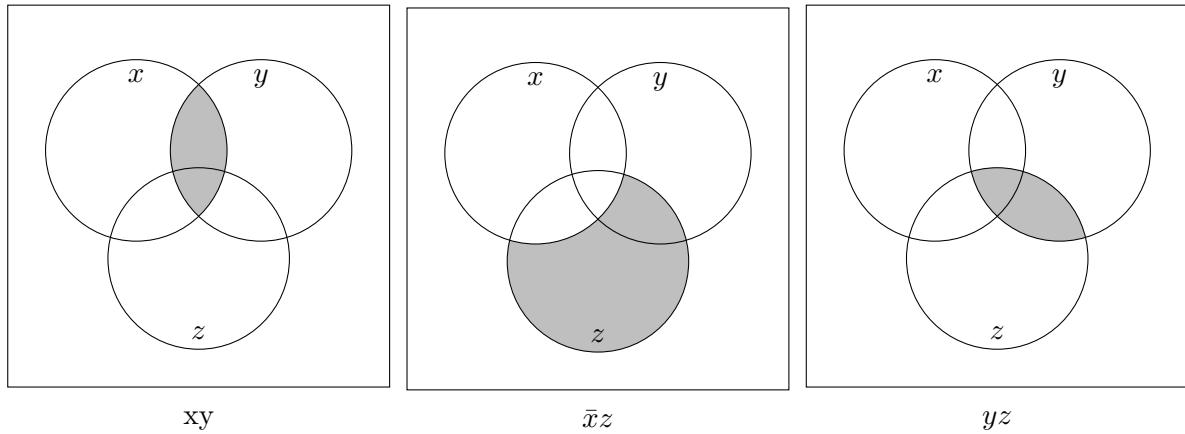
A Venn diagram is a visual representation of the relationships between sets. Each set is represented by a circle. The overlapping parts of circles represent common elements.



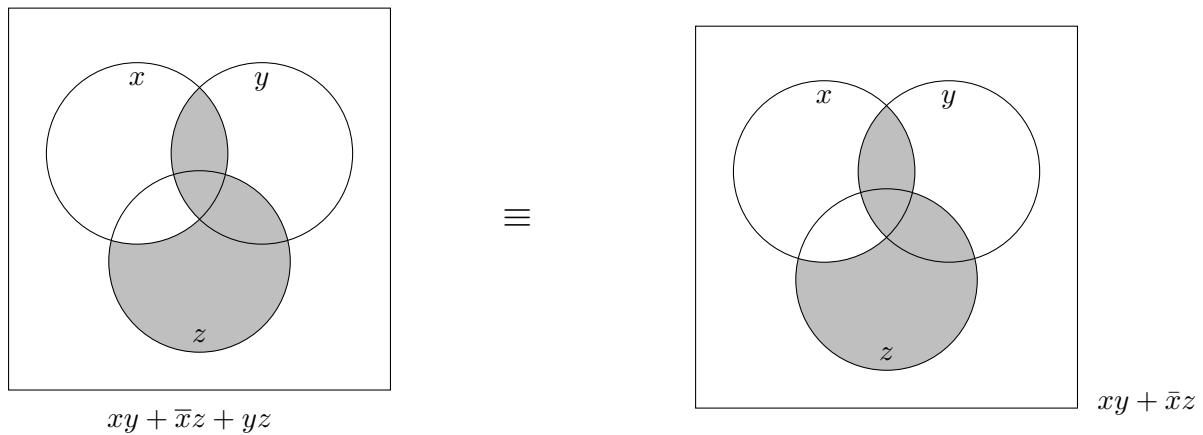
## 6.6 Network Equivalence Verification

Venn Diagram Approach

$$xy + \bar{x}z + yz \stackrel{?}{=} xy + \bar{x}z$$



Thus, the two expressions are equivalent.



## 6.7 Boolean Algebra

### 6.7.1 Axioms

- 1a.  $0 \cdot 0 = 0$
- 1b.  $1 + 1 = 1$
- 2a.  $1 \cdot 1 = 1$
- 2b.  $0 + 0 = 0$
- 3a.  $0 \cdot 1 = 1 \cdot 0 = 0$
- 3b.  $1 + 0 = 0 + 1 = 1$
- 4a. If  $x = 0$ , then  $\bar{x} = 1$
- 4b. If  $x = 1$ , then  $\bar{x} = 0$
- 5a.  $x \cdot 0 = 0$
- 5b.  $x + 1 = 1$
- 6a.  $x \cdot 1 = x$
- 6b.  $x + 0 = x$
- 7a.  $x \cdot x = x$
- 7b.  $x + x = x$
- 8a.  $x \cdot \bar{x} = 0$
- 8b.  $x + \bar{x} = 1$
- 9.  $\bar{\bar{x}} = x$
- 10. Commutative Property:
  - (a) Multiplication:  $x \cdot y = y \cdot x$
  - (b) Addition:  $x + y = y + x$
- 11. Associative Property:
  - (a) Multiplication:  $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
  - (b) Addition:  $x + (y + z) = (x + y) + z$
- 12. Distributive Property:
  - (a) Multiplication over Addition:  

$$x \cdot (y + z) = x \cdot y + x \cdot z$$
  - (b)  $x + y \cdot z = (x + y) \cdot (x + z)$
- 13. Absorption (covering):
  - (a)  $x + x \cdot y = x$
  - b.  $x \cdot (x + y) = x$
- 14. Combining:
  - (a)  $x \cdot y + x \cdot \bar{y} = x$
  - (b)  $(x + y) \cdot (x + \bar{y}) = x$
- 15. DeMorgan's theorem:
  - (a)  $\bar{x} \cdot \bar{y} = \overline{x + y}$
  - (b)  $\bar{x} + \bar{y} = \overline{x \cdot y}$
- 16. Redundancy:
  - (a)  $x + \bar{x} \cdot y = x + y$
  - (b)  $x \cdot (\bar{x} + y) = x \cdot y$
- 17. Consensus:
  - (a)  $x \cdot y + y \cdot z + \bar{x} \cdot z = x \cdot y + \bar{x} \cdot z$
  - (b)  $(x+y) \cdot (y+z) \cdot (\bar{x}+z) = (x+y) \cdot (\bar{x}+z)$

# Chapter 7

## Digital Logic (PART II)

### 7.1 Logic Synthesis

#### 7.1.1 Minterms and Maxterms

*Personal Remark. I changed the expression of the function for the product as it was clearer to my sens*

*Personal Remark.2. This process ressembles a lot what we've seen in AICC I, with CNF and DNF.*

*These two functions correspond to ways of representing binary numbers like we're used but using n-variable vectors. Here  $i$  is the number we're representing  $m_i$  the corresponding "variable" minterm representation and  $M_i$  maxterm's*

Let a product of n variables  $f(x_1, x_2, \dots, x_n) = \prod_{j=1}^n x_j = x_1 \times x_2 \dots \times x_n = i = m_i$  in which each of the n variables only appear once, this is called a **minterm**.

Let a sum of n variables  $f(x_1, x_2, \dots, x_n) = \sum_{j=1}^n x_j = x_1 + x_2 + \dots + x_n = i = M_i$  in which each of the n variables only appear once, this is called a **maxterm**.

#### 7.1.2 Examples

##### Minterms

To make it simple,  $1 \rightarrow x$ ,  $0 \rightarrow \bar{x}$

Let  $n = 3, i = 5$ :

$5 = \underline{101}_2$  thus  $m_5 = x_1 \bar{x}_2 x_3$

Let  $n = 5, i = 3$ :

$3 = \underline{00011}_2$  thus  $m_3 = \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4 \bar{x}_5$

## Maxterms

Maxterms are calculated as  $M_i = \overline{m}_i$

Let  $n = 3, i = 5$ :

Using De Morgan's Law :

We have :

$$m_5 = x_1 \overline{x}_2 x_3$$

thus :

$$M_5 = \overline{m}_5 = \overline{x_1 \overline{x}_2 x_3} = \overline{x}_1 + x_2 + \overline{x}_3$$

Let  $n = 5, i = 3$ :

We have :

$$m_3 = \overline{x}_1 \overline{x}_2 x_3 \overline{x}_4 \overline{x}_5$$

thus :

$$M_3 = \overline{m}_3 = \overline{\overline{x}_1 \overline{x}_2 \overline{x}_3 x_4 x_5} = x_1 + x_2 + x_3 + \overline{x}_4 + \overline{x}_5$$

Row number	$x_1$	$x_2$	$x_3$	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x}_1 x_2 \overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1 \overline{x}_2 x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1 x_2 \overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1 x_2 x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1 \overline{x}_2 \overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1 \overline{x}_2 x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1 x_2 \overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1 x_2 x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

### 7.1.3 Sum-of-Product (SOP) Form and Product-of-Sum (POS) Form

For a function  $f$  represented by a table, the rows where  $f = 1$  are represented by a sum of minterms, and the rows where  $f = 0$  are represented by a product of maxterms.

*Remark* This is not the most optimal implementation of a function.

#### Sum-of-Product (SOP) Form

Consider a function  $f$  of  $n = 3$  variables and the truth table below  
Canonical SOP form:

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6) = \sum m(1, 4, 5, 6)$$

$x_1$	$x_2$	$x_3$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Thus the canonical SOP form :

$$\begin{aligned} f(x_1, x_2, x_3) &= \overline{x_1} \overline{x_2} x_3 + x_1 \overline{x_2} \overline{x_3} + x_1 \overline{x_2} x_3 + x_1 x_2 \overline{x_3} \\ &= (\overline{x_1} + x_1)x_2 x_3 + \overline{x_1}(\overline{x_2} + x_2)x_3 \\ &= 1 \cdot x_2 x_3 + \overline{x_1} \cdot 1 \cdot x_3 \\ &= x_2 x_3 + \overline{x_1} x_3 \end{aligned}$$

#### Product-of-Sum (POS) Form

Consider a function  $f$  of  $n = 3$  variables and the truth table below

$x_1$	$x_2$	$x_3$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$$f(x_1, x_2, x_3) = \prod(M_0, M_2, M_3, M_7)$$

$$f(x_1, x_2, x_3) = M_0 \cdot M_2 \cdot M_3 \cdot M_7$$

$$f(x_1, x_2, x_3) = (x_1 + x_2 + x_3)(x_1 + \overline{x_2} + x_3)(\overline{x_1} + x_2 + \overline{x_3})$$

Complementing f :

$$\overline{f}(x_1, x_2, x_3) = m_0 + m_2 + m_3 + m_7 = \overline{M_0} + \overline{M_2} + \overline{M_3} + \overline{M_7}$$

$$\overline{f}(x_1, x_2, x_3) = \overline{M_0 \cdot M_2 \cdot M_3 \cdot M_7}$$

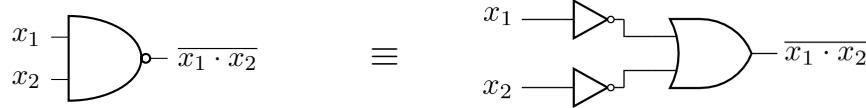
By De Morgan's theorem

$$f(x_1, x_2, x_3) = m_0 + m_2 + m_3 + m_7 = \overline{f}$$

*Personal Remark.* Don't forget the objective here, these are just two ways, equivalent ways, to represent a function only using ORs and ANDs.

## 7.2 NAND and NOR Logic Networks

### 7.2.1 NAND GATE



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

#### Logic Network with NAND Gates

Implement the following function in the SOP form with NAND gates:

$$f = x_2 + \overline{x_1 x_3} \quad (7.1)$$

Algorithm: start by applying double inversion and, then, De Morgan's theorem to simplify the expression:

$$f = x_2 + \overline{x_1 x_3} \quad (7.2)$$

$$= \overline{\overline{x_2} + x_1 \overline{x_3}} \quad (7.3)$$

$$= \overline{\overline{x_2} \cdot \overline{x_1 \overline{x_3}}} \quad (7.4)$$

### 7.2.2 NOR GATE



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

## Logic Network with NOR Gates

Implement the following function in the POS form with NOR gates:

$$f = (x_1 + x_2)(x_2 + x_3) \quad (7.5)$$

Algorithm: start by applying double inversion and, then, De Morgan's theorem to simplify the expression:

$$f = (x_1 + x_2)(x_2 + \overline{x_3}) \quad (7.6)$$

$$= \overline{(x_1 + x_2)(x_2 + \overline{x_3})} \quad (7.7)$$

$$= \overline{(x_1 + x_2)} + \overline{(x_2 + \overline{x_3})} \quad (7.8)$$

## 7.3 Incompletely Defined Functions

### 7.3.1 Don't Care Condition

Don't care conditions occur in logic design when certain input combinations never occur, or their corresponding outputs do not affect the system behavior.

### 7.3.2 Example

Consider  $x_1$  and  $x_2$  as inputs controlling two doors to a lion's cage. Here,  $x_1$  is the control for the outer door and  $x_2$  for the inner door. The truth table is as follows:

$x_1$	$x_2$	$f$	Behavior
0	0	0	Doors closed
0	1	1	Outer closed, inner open
1	0	1	Outer open, inner closed
1	1	X	Doors open (don't care)

The condition where both doors are open is a 'don't care' because it should never happen for safety reasons.

### 7.3.3 Incomplete Functions

A logic function with 'don't care' conditions is termed 'incompletely defined'. These conditions can be exploited to optimize the circuit by choosing don't care outputs to simplify the logic expression.

### 7.3.4 Sum of Products (SOP) Example

For the lion's cage control, with 'don't cares' the function can be expressed as:

$$f = \sum m(1, 2) + D(3)$$

- Assuming  $D(3) = 0$ , the function simplifies to:

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

using 5 gates and 8 inputs.

- Assuming  $D(3) = 1$ , the function simplifies further to:

$$f = x_1 \bar{x}_2 + x_1 = x_1 + x_2$$

using an OR gate and redundancy rules.

*Adding don't care values sometimes allows for a simpler implementation.*

## 7.4 Even and Odd Detectors (XNOR and XOR Gates)

### 7.4.1 XOR Gate

*True when one and only one input is True* The exclusif OR (XOR) gate is represented with the table below :

$x_1$	$x_2$	$f$
0	0	0
0	1	1
1	0	1
1	1	0

and looks like:



$$f = x_1 \oplus x_2$$

### 7.4.2 XNOR Gate

The exclusif NOR (XNOR) gate is represented with the table below : *True when both inputs are the same.*

$x_1$	$x_2$	$f$
0	0	1
0	1	0
1	0	0
1	1	1

and looks like:

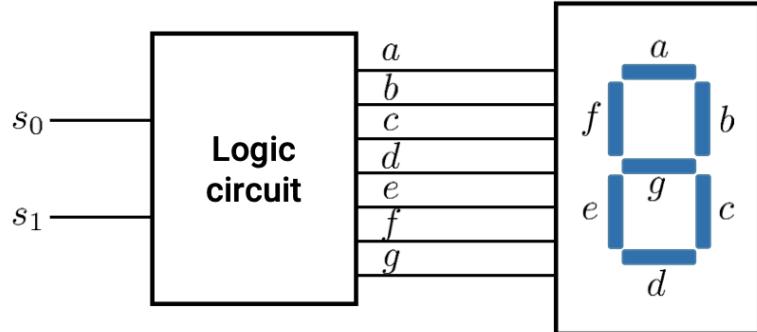


$$f = x_1 \odot x_2$$

## 7.5 Design Example

### 7.5.1 Number Display

Here we will be designing a logic circuit to drive a seven-segment display .



We can derive one logic function per output of the table below:

$S_1$	$S_0$	Output						
		a	b	c	d	e	f	g
0	0	1	1	1	1	1	1	0
0	1	0	1	1	0	0	0	0
1	0	1	1	0	1	1	0	1
1	1	1	1	1	1	0	0	1

$$a(s_0, s_1) = M_1 = s_1 + \overline{s_0}$$

$$b(s_0, s_1) = 1$$

$$c(s_0, s_1) = M_2 = s_1 + s_0$$

$$d(s_0, s_1) = M_1 = s_1 + \overline{s_0} = a(s_0, s_1)$$

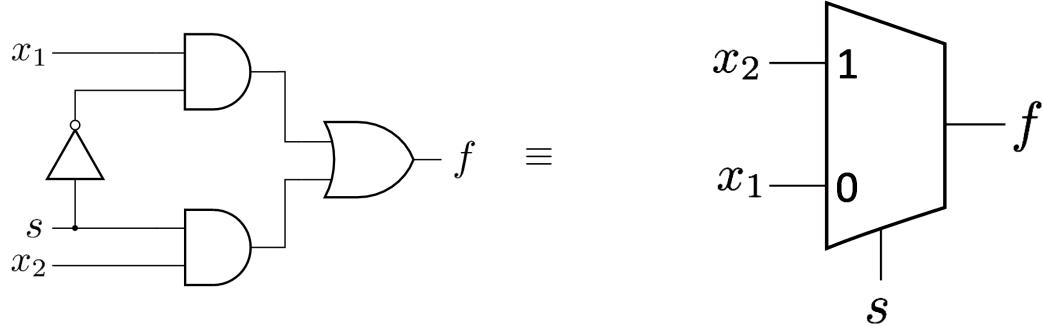
$$e(s_0, s_1) = M_1 \cdot M_3 = m_0 + m_2 = s_1 \overline{s_0} + s_1 s_0 = \overline{s_0}$$

$$f(s_0, s_1) = m_0 = \overline{s_1 s_0}$$

$$g(s_0, s_1) = M_0 \cdot M_1 = m_2 + m_3 = s_1 \overline{s_0} + s_1 s_0 = s_1$$

### 7.5.2 Multiplexer

A *multiplexer*, or MUX, is a circuit that selects one of several inputs to pass to the output based on the value of one or more select inputs.



Reading the truth table, we can derive the following logic functions:

$s$	$x_1$	$x_2$	$f$
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

$$f(s, x_1, x_2) = \bar{s}x_1x_2 + \bar{s}x_1\bar{x}_2 + s\bar{x}_1x_2 + sx_1x_2$$

$$f(s, x_1, x_2) = \bar{s}x_1(\bar{x}_2 + x_2) + s(\bar{x}_1 + x_1)x_2$$

$$= \bar{s}x_1 + sx_2$$

If there are  $n$  inputs to select from, the number of select signals required in a MUX can be determined by:

$$\text{Number of select signals} = \lceil \log_2 n \rceil$$

*Examples:*

Two inputs: one select signal to choose between inputs indexed as '0' and '1'

Four inputs: two select signals (indices 0, 1, 2, 3)

Eight inputs: three select signals (indices 0, 1, 2, ..., 7)

# Chapter 8

## Digital Logic (PART III)

### 8.1 Adders

#### 8.1.1 Addition of two 1-bit binary numbers

In this section, we will be adding two 1-bit binary numbers. The truth table is as follows:

The resulting sum is at most on two bits:

- the rightmost bit is called *sum* (*s*)
- the leftmost bit is called *carry* (*c*); it is produced as a carry-out when both bits being added are logical one

$$\begin{array}{r} & \text{x} \\ & + \quad \text{y} \\ \hline \text{c (carry)} & \text{s (sum)} \end{array}$$

#### 8.1.2 Binary Addition Examples

$$\begin{array}{r} 0 \\ + \quad 0 \\ \hline 0 \quad 0 \end{array}$$

$$\begin{array}{r} 0 \\ + \quad 1 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + \quad 0 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + \quad 1 \\ \hline 1 \quad 0 \end{array}$$

### 8.1.3 Half-Adder

*Personal Remark. The circuits shown in this part might seem a bit overwhelming, I suggest you first try to understand what the full and half adders are, then try to understand the corresponding circuits.*

A half adder is a digital circuit that adds two single-bit binary numbers and outputs a sum and a carry.

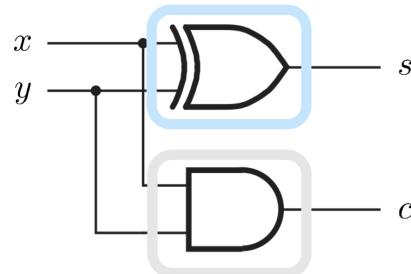
x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Using SOP form, we can derive the following logic functions:

$$s = \bar{x}y + x\bar{y} = x \oplus y$$

$$c = xy$$

Thus the Digital Logic Circuit :



Also represented as :



### 8.1.4 Addition of Two N-Bit Binary Numbers

A binary  $n$ -bit adder computes the sum of two  $n$ -bit numbers  $x$  and  $y$  within the range of  $0 \leq x, y \leq 2^n - 1$ , with a carry-in  $c_{\text{in}} \in \{0, 1\}$ . It produces:

A sum  $s$ , where  $0 \leq s \leq 2^n - 1$ .

A carry-out  $c_{\text{out}} \in \{0, 1\}$ , satisfying the equation:

$$x + y + c_{\text{in}} = 2^n c_{\text{out}} + s \quad (8.1)$$

The solution to the equation is given by:

$$s = (x + y + c_{\text{in}}) \mod 2^n \quad (8.2)$$

$$c_{\text{out}} = \begin{cases} 1 & \text{if } (x + y + c_{\text{in}}) \geq 2^n (\text{overflow}) \\ 0 & \text{otherwise} \end{cases} = \left\lfloor \frac{(x + y + c_{\text{in}})}{2^n} \right\rfloor \quad (8.3)$$

### 8.1.5 Addition of Two N-Bit Binary Numbers

It is impractical to start from the truth tables for  $n$ -bit addition.

Iterative approach:

Add each pair of bits at the position  $i$ , where  $0 \leq i < n$ .

The addition at the bit position  $i$  needs to include a carry-in at the position  $i$  (corresponding to the carry-out at the position  $i - 1$ ).

The 1-bit adder reduces to a primitive module (new block that summarizes the circuit) called *full-adder (FA)* with three binary inputs and two binary outputs such that

$$x_i + y_i + c_i = 2c_{i+1} + s_i.$$

### 8.1.6 Full-Adder

A full adder, on the other hand, is a digital circuit that adds three single-bit binary numbers: the two bits to be added plus an additional carry-in bit. This carry-in bit is the carry from the addition of two lower significant bits. The full adder outputs a sum and a carry-out.

Truth table:

$x_i$	$y_i$	$c_i$	$s_i$	$c_{i+1}$
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

Logical expressions:

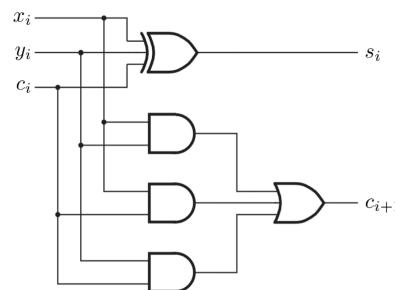
$$\begin{aligned}
 s_i &= \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i \\
 &= (x_i y_i + \bar{x}_i \bar{y}_i) c_i + (\bar{x}_i y_i + x_i \bar{y}_i) \bar{c}_i \\
 &= \overline{(x_i \oplus y_i)} c_i + (x_i \oplus y_i) \bar{c}_i = x_i \oplus y_i \oplus c_i
 \end{aligned}$$

$$\begin{aligned}
 c_{i+1} &= \bar{x}_i y_i c_i + x_i \bar{y}_i c_i + x_i y_i \bar{c}_i + x_i y_i c_i \\
 &= (\bar{x}_i y_i + x_i \bar{y}_i) c_i + x_i y_i (\bar{c}_i + c_i) \\
 &= (x_i \oplus y_i) c_i + x_i y_i \\
 &= x_i y_i + x_i c_i + y_i c_i
 \end{aligned}$$

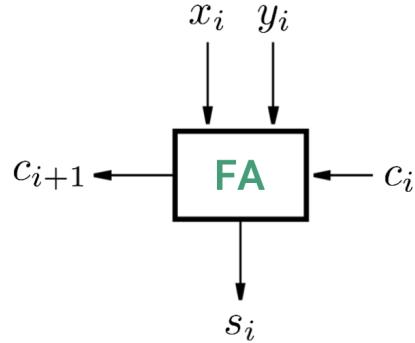
Thus:

$$\begin{aligned}
 s_i &= x_i \oplus y_i \oplus c_i \\
 c_{i+1} &= (x_i \oplus y_i) c_i + x_i y_i = x_i y_i + x_i c_i + y_i c_i
 \end{aligned}$$

With Logic Circuit:



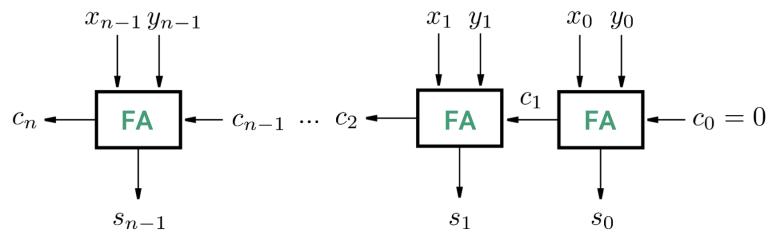
Also represented as :



### 8.1.7 Basic Ripple-Carry Adder

Is a chain of full adders that add two  $n$ -bit binary numbers. It's called a **ripple-carry** adder because the carry-out of each full adder ripples (goes through the full adder chain) into the carry-in of the next full adder.

It looks like this:



## 8.2 Subtractors

*Same but with subtraction and borrows.*

### 8.2.1 Subtraction of Two 1-Bit Binary Numbers

In this section, we will be subtracting two 1-bit binary numbers. The truth table is as follows:

The resulting difference is at most on two bits:

- the rightmost bit is called *difference* (d)
  - the leftmost bit is called *borrow* (b); it is produced as a borrow-out when the subtrahend is greater than the minuend

Let  $b$  be the borrow-in,  $x$  be the minuend, and  $y$  be the subtrahend and  $d$  the difference.

$$\frac{b}{d} = \frac{x}{y}$$

### 8.2.2 Binary Subtraction Examples

$$\begin{array}{r} 0 \\ 0 \\ - 0 \\ \hline 0 \end{array}$$

$$\begin{array}{r} 1 \\ 0 \\ - 1 \\ \hline 1 \end{array}$$

$$\begin{array}{r} 0 \\ 1 \\ - 0 \\ \hline 1 \end{array}$$

$$\begin{array}{r} 0 \\ 1 \\ - 1 \\ \hline 0 \end{array}$$

### 8.2.3 Subtraction of Two N-Bit Unsigned Numbers

It is impractical to start from the truth tables for  $n$ -bit subtraction.

Iterative approach

- Subtract each pair of bits at the position  $i$ ,  $0 \leq i < n$ .
- The subtraction at the bit position  $i$  needs to include a borrow-in at position  $i$  (i.e., borrow-out at the position  $i - 1$ ).

### 8.2.4 Full Subtractor

Subtraction of Two 1-Bit Binary Numbers Taking into Account the Input Borrow

Truth table:

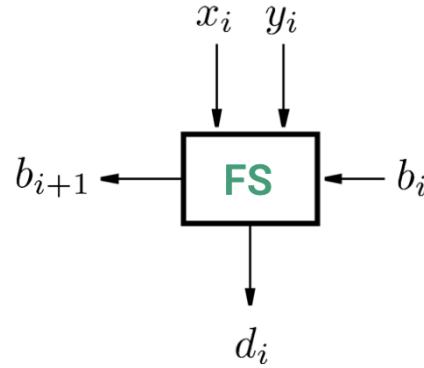
$x_i$	$y_i$	$b_i$	$d_i$	$b_{i+1}$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Logical expressions:

$$\begin{aligned}
 d_i &= \bar{x}_i \bar{y}_i b_i + \bar{x}_i y_i \bar{b}_i + x_i \bar{y}_i \bar{b}_i + x_i y_i b_i \\
 &= (\bar{x}_i \bar{y}_i + x_i y_i) b_i + (\bar{x}_i y_i + x_i \bar{y}_i) \bar{b}_i = \overline{(x_i \oplus y_i)} b_i + (x_i \oplus y_i) \bar{b}_i \\
 &= x_i \oplus y_i \oplus b_i
 \end{aligned}$$

$$\begin{aligned}
 b_{i+1} &= \bar{x}_i \bar{y}_i b_i + \bar{x}_i y_i \bar{b}_i + \bar{x}_i y_i b_i + x_i y_i b_i \\
 &= (\bar{x}_i \bar{y}_i b_i + \bar{x}_i y_i b_i) + (\bar{x}_i y_i \bar{b}_i + \bar{x}_i y_i b_i) + (x_i y_i b_i + x_i y_i b_i) \\
 &= \bar{x}_i b_i + \bar{x}_i y_i + y_i b_i
 \end{aligned}$$

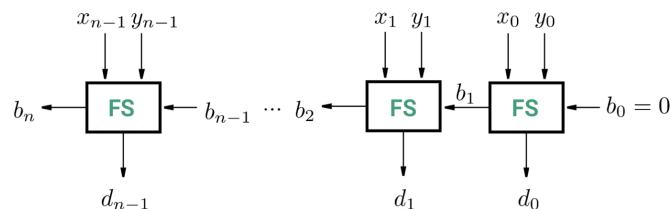
Also represented as :



### 8.2.5 N-Bit Ripple-Carry Subtractor

Subtracting Two N-bit Binary Numbers

Starting from the least-significant digit, we subtract pairs of digits, progressing to the most-significant digit.



### 8.3 Adders-Subtractors in two's complement

Recall that subtracting two numbers in two's complement format requires using the two's complement of one operand:

$$X - Y = X + \bar{Y} + 1$$

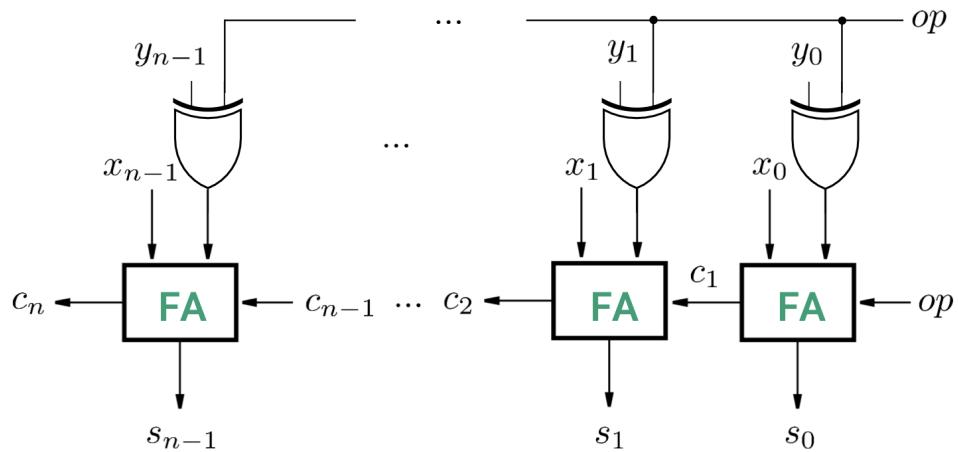
$\bar{Y}$  is obtained by complementing each of the bits of  $Y$

Assume a control signal  $op$  determines which operation to perform  
( $op = 0$ : addition,  $op = 1$ : subtraction)

$op$	$f(X, Y)$
0	$X + Y$
1	$X + \bar{Y} + 1$

$$f(X, Y) = X + \bar{o}p \cdot Y + op \cdot \bar{Y} + op$$

One circuit, able to perform two operations :



### 8.4 Fast Adders

#### 8.4.1 Performance Matters

Addition and subtraction are essential operations in computing systems and their efficiency is crucial for overall system performance.

The value of a system is quantified by the ratio of its performance to cost.

$$\text{value} = \frac{\text{performance}}{\text{price}}$$

### 8.4.2 Examples of delays

#### Full Adder

- Delay to generate the sum

$$\begin{aligned} t(x_i, s_i) &= t(c_i, s_i) = t(\text{XOR}) \\ t(y_i, s_i) &= t(\text{op}, s_i) = 2t(\text{XOR}) \end{aligned}$$

- Delay to generate carry-out

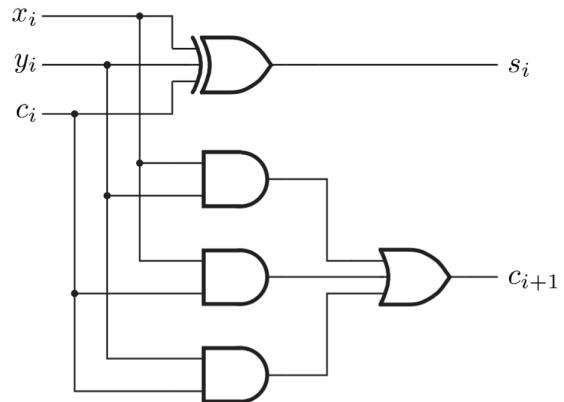
$$\begin{aligned} t(x_i, c_{i+1}) &= t(c_i, c_{i+1}) = t(\text{AND}) + t(\text{OR}) \\ t(y_i, c_{i+1}) &= t(\text{op}, c_{i+1}) = t(\text{XOR}) + t(\text{AND}) + t(\text{OR}) \end{aligned}$$

- Worst-case delay

$$\begin{aligned} t_{\max} &= \max(t(s_i), t(c_{i+1})) \\ &= \max(2t(\text{XOR}), t(\text{XOR}) + t(\text{AND}) + t(\text{OR})) \end{aligned}$$

- If all gates had equal delays:

$$t_{\max} = t(c_{i+1}) = 3 \text{ Gate Delays}$$



#### Full Adder-Subtractor

- Delay to generate the sum

$$\begin{aligned} t(x_i, s_i) &= t(c_i, s_i) = t(\text{XOR}) \\ t(y_i, s_i) &= t(\text{op}, s_i) = 2t(\text{XOR}) \end{aligned}$$

- Delay to generate carry-out

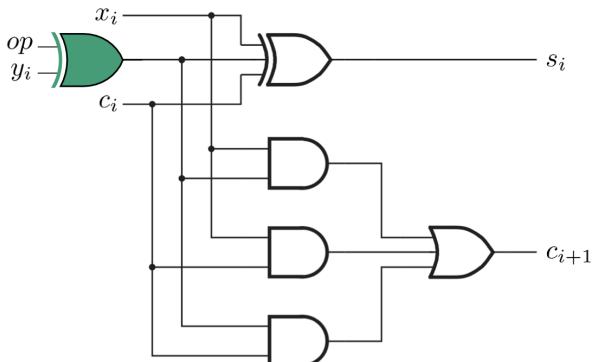
$$\begin{aligned} t(x_i, c_{i+1}) &= t(c_i, c_{i+1}) = t(\text{AND}) + t(\text{OR}) \\ t(y_i, c_{i+1}) &= t(\text{op}, c_{i+1}) = t(\text{XOR}) + t(\text{AND}) + t(\text{OR}) \end{aligned}$$

- Worst-case delay

$$\begin{aligned} t_{\max} &= \max(t(s_i), t(c_{i+1})) \\ &= \max(2t(\text{XOR}), t(\text{XOR}) + t(\text{AND}) + t(\text{OR})) \end{aligned}$$

- If all gates had equal delays:

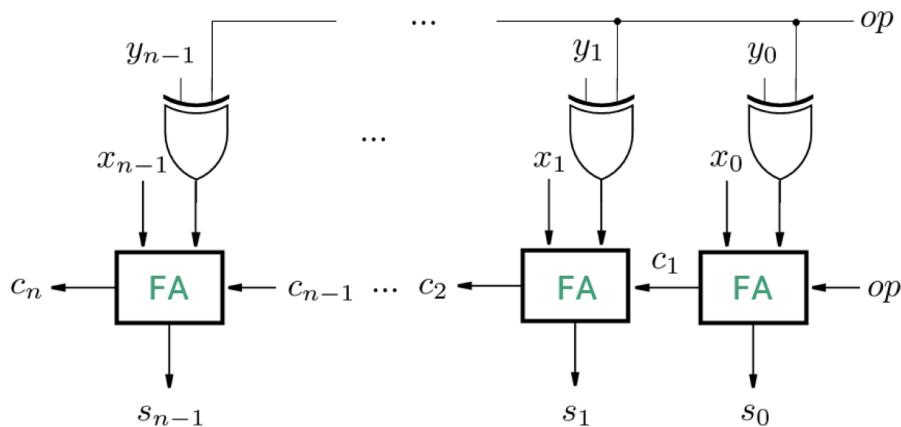
$$t_{\max} = t(c_{i+1}) = 3 \text{ Gate Delays}$$



### 8.4.3 Summary of the Ripple-Carry Adder-Subtractor

The Ripple-Carry Adder-Subtractor is a fundamental component in digital circuits for performing binary addition and subtraction. The key concept to understand includes the **worst-case delay** which is pivotal in determining the efficiency of the component.

- Inputs  $\mathbf{X}$ ,  $\mathbf{Y}$ , and  $op$  are immediately available, hence no delay due to waiting.
- It is assumed that all gates involved have identical delay times for simplification.
- The worst-case delay, also known as *critical path delay (CPD)*, is crucial for assessing performance.
- Given  $n$  bits, the worst-case delay for finding the sum or difference can be calculated as  $(2n + 1)$  gate delays.
- While the Ripple-Carry Adder-Subtractor is a simple and effective component, it is not the most efficient in terms of speed. (With the increasing number of bits , the adder delay increases and the computation becomes prohibitively slow)



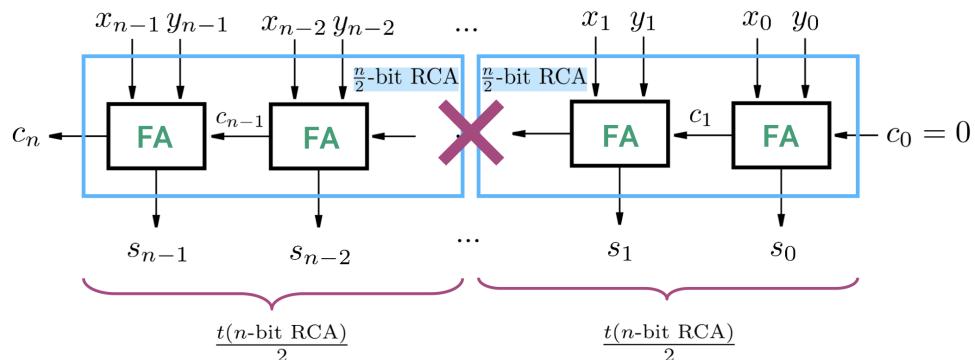
### 8.4.4 Carry-Select Adder

*First steps in parallel computing...*

In a Basic Ripple Carry Adder, the carry-in is sent to each full adder of the system until the last bit. This sequential processing of carry signals results in a significant delay, especially for large bit widths, because each full adder must wait for the carry input from its predecessor before it can complete its operation.

A Carry Select Adder aims to fix this delay issue by improving the speed of carry propagation. It does this by having the second half of the adder calculate two sets of sums and carries for each block—once assuming the carry-in is 0, and once assuming the carry-in is 1. This is done in parallel to the operation of the first half of the adder.

When the actual carry-in for the second half becomes available from the first half, a multiplexer selects the correct set of sums and carries for each block based on the actual carry-in value. This method allows the Carry Select Adder to reduce the overall computation time because it eliminates the need to wait for the carry to propagate through all the bits, significantly speeding up the addition process for multi-bit numbers.



## 8.5 Shifting

### 8.5.1 Barrel Shifter

- **Direction:** It can move bits to the left or to the right.

- **Type of shift:**

*Logical shift:* Move bits and fill the new space with zeros.

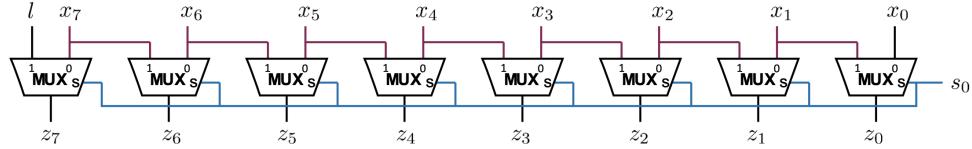
*Arithmetic shift:* Keep the sign of a binary number while shifting.

*Circular/rotation shift:* Move bits around in a circle, where the bits that fall off one end come back at the other end.

- **Amount of shift:** How many places you want to move the bits (0 to n-1).

## Shift Right by one position

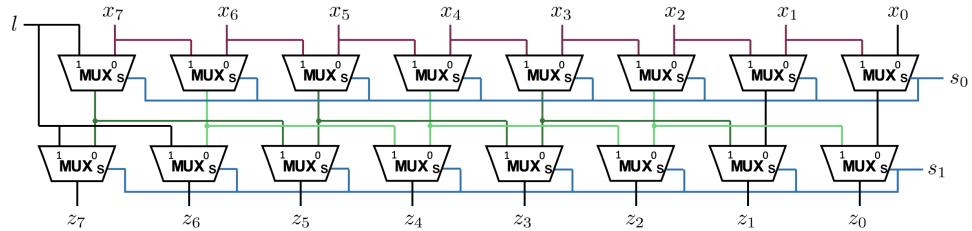
if  $l = 0$ , the shift is Logical (resets the leftmost bit to 0), if  $l = x_{n-1}$ , the shift is Arithmetic (conserve the sign of the number).



Thus, the corresponding truth table:

$s_0$	$z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0$
0	$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$
1	$lx_7 x_6 x_5 x_4 x_3 x_2 x_1$

## Shift Right by up to Three positions



Thus, the corresponding table :

$s_1$	$s_0$	$z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0$
0	0	$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$
0	1	$lx_7 x_6 x_5 x_4 x_3 x_2 x_1$
1	0	$llx_7 x_6 x_5 x_4 x_3 x_2$
1	1	$lllx_7 x_6 x_5 x_4 x_3$

### 8.5.2 Bidirectional Shifting by up to 7 positions

The circuit implements a bidirectional shift register enabling shifts left or right by up to seven positions. It consists of:

Inputs  $\text{in}0$  to  $\text{in}7$  and outputs  $\text{out}0$  to  $\text{out}7$ .

Control signals  $S0$ ,  $S1$ ,  $S2$  for the shift magnitude, and  $\text{dir}$  for the direction.

A logical input 1 for leftward shifts.

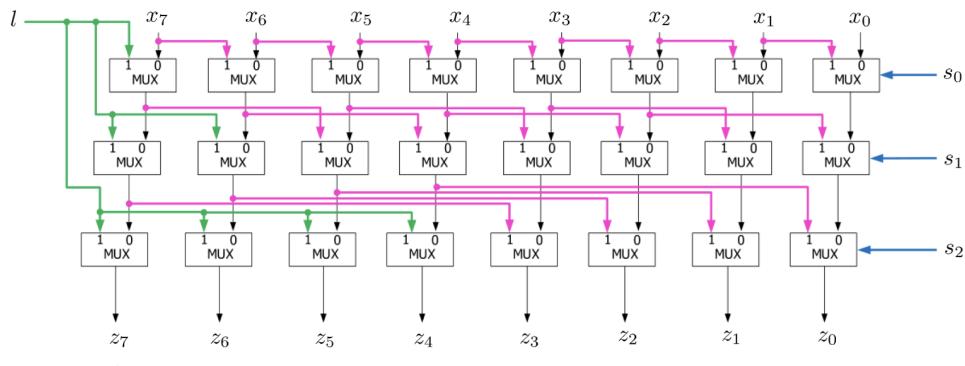
Shift operation:

**Right Shift:** With  $\text{dir}$  high, each MUX passes the value from the left input to the right output.

**Left Shift:** With  $\text{dir}$  low, each MUX passes the value from the right input to the left output, with  $\text{out}7$  taking the value of 1.

The signals  $S0$ ,  $S1$ , and  $S2$  determine the shift amount, with binary encoding representing 0 to 7 positions.

The circuit (*which might look a bit scary at first...*) looks like this:



S2	S1	S0	dir	Operation
0	0	0	0	No shift (or shift by 0) to the left
0	0	0	1	No shift (or shift by 0) to the right
0	0	1	0	Shift 1 position to the left
0	0	1	1	Shift 1 position to the right
0	1	0	0	Shift 2 positions to the left
0	1	0	1	Shift 2 positions to the right
0	1	1	0	Shift 3 positions to the left
0	1	1	1	Shift 3 positions to the right
1	0	0	0	Shift 4 positions to the left
1	0	0	1	Shift 4 positions to the right
1	0	1	0	Shift 5 positions to the left
1	0	1	1	Shift 5 positions to the right
1	1	0	0	Shift 6 positions to the left
1	1	0	1	Shift 6 positions to the right
1	1	1	0	Shift 7 positions to the left
1	1	1	1	Shift 7 positions to the right

Table 8.1: Truth Table for the Bidirectional Shift Register

# Chapter 9

## Digital Logic (PART IV)

### 9.1 Transistors

A Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) is a transistor used for amplifying or switching electronic signals. It has three terminals:

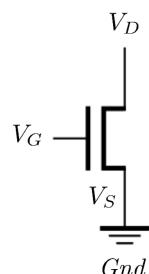
- **Drain (D):** Current exits the transistor, with voltage  $V_D$ .
- **Gate (G):** Controls the transistor operation, with voltage  $V_G$ .
- **Source (S):** Current enters the transistor, with voltage  $V_S$ .

The gate voltage ( $V_G$ ) controls the current flow between the drain and source. When  $V_G$  exceeds a certain threshold, an electric field is created that allows current to flow, making the MOSFET act as an efficient electronic switch.

#### 9.1.1 NMOS Transistor Switches

NMOS works like a switch that turns on to let electricity flow when you apply a certain voltage.

The digital representation of it looks like this :



- Open ( $x = 0$ );  $V_G = 0$



- Closed ( $x = 1$ );  $V_G = V_{DD}$

$$V_D = 0 \text{ V}$$

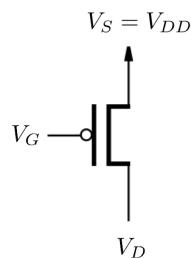


### 9.1.2 PMOS Transistor Switches

*For french people, P like Porte, you push the door (need energy) to open it.*

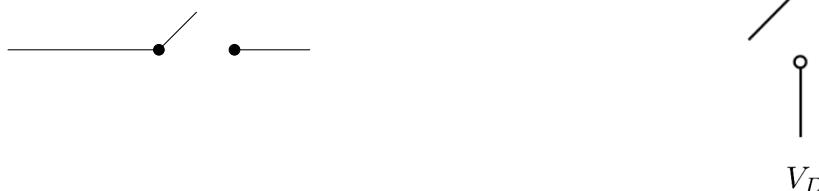
PMOS works like a switch that turns on to let electricity flow when you apply a certain voltage.

The digital representation of it looks like this :



- Open ( $x = 1$ );  $V_G = V_{DD}$

$$V_{DD}$$



- Closed ( $x = 0$ );  $V_G = 0$

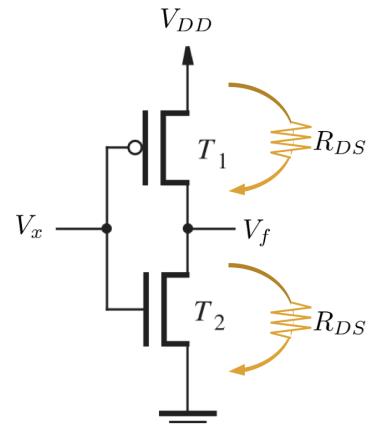


### 9.1.3 Example - CMOS

PMOS (T1) is connected from power supply  $V_{DD}$  to the output  $V_f$ .

NMOS (T2) is connected from ground to the output  $V_f$ .

The input voltage  $V_x$  controls the gates of both transistors.



When  $V_x$  is low:

- T1 conducts (on) since PMOS is active when gate voltage is less than the source.
- T2 does not conduct (off) as NMOS requires gate voltage higher than the source to be active.
- Thus,  $V_f$  is connected to  $V_{DD}$ , resulting in a high output (logical '1').

When  $V_x$  is high:

- T1 does not conduct (off).
- T2 conducts (on).
- Consequently,  $V_f$  is pulled to ground, producing a low output (logical '0').

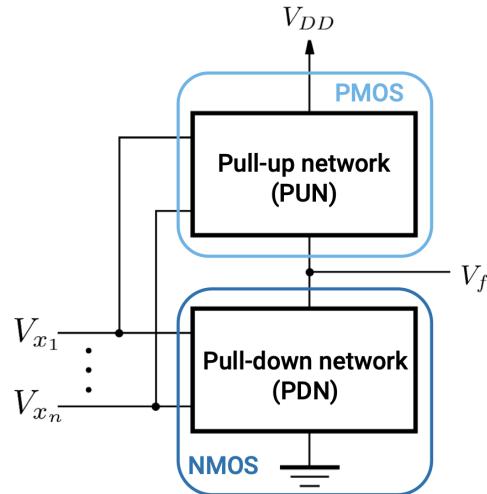
CMOS circuits are power efficient as they draw significant power only during the transition between states (dynamic power consumption), not while in a steady state.

Thus, the truth table:

$x$	$T_1$	$T_2$	$f$
0	ON	OFF	1
1	OFF	ON	0

**The circuit operates as an inverter. (NOT GATE)**

### 9.1.4 CMOS Circuit Structure



Complementary functions performed by a pull-up and pull-down network:

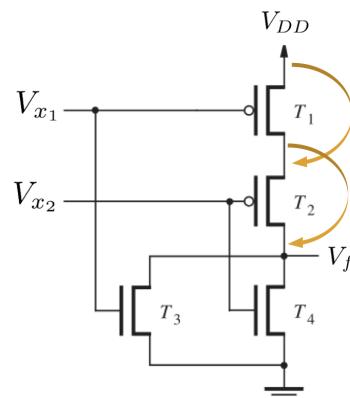
- Pull-up composed of **PMOS**
- Pull-down composed of **NMOS**

Pull-up and pull-down networks are *dual* to one another and have an *equal* number of transistors

### 9.1.5 CMOS Circuits - Examples

*Personal Remark.* Make sure you're able to quickly identify NMOS and PMOS and how they work as this really helps understanding how we construct the table for more complex CMOS circuits

#### NOR Gate



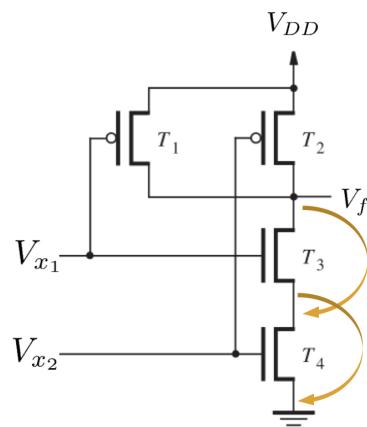
Thus the truth table:

$x_1$	$x_2$	$T_1$	$T_2$	$T_3$	$T_4$	$V_f$
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	0
1	0	OFF	ON	ON	OFF	0
1	1	OFF	OFF	ON	ON	0

Cost (size, area): four transistors

### NAND Gate

- Find the functionality of the given CMOS gate



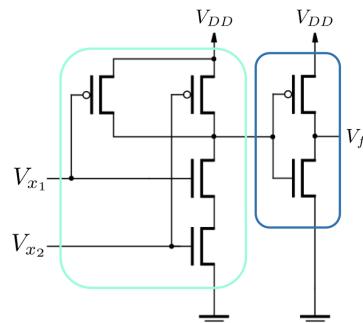
Thus the truth table:

$x_1$	$x_2$	$T_1$	$T_2$	$T_3$	$T_4$	$f$
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

Cost (size, area): four transistors

### AND Gate

A NAND gate (green) followed by a NOT (blue) gate, forming an AND gate.



Thus the truth table:

$x_1$	$x_2$	$f$
0	0	0
0	1	0
1	0	0
1	1	1

Cost (size, area): six transistors

## 9.2 Real Voltage Waveforms

### 9.2.1 Logic Values as Voltage Levels

Binary values (0, 1) in digital circuits are represented as voltage levels

- 0: low (voltage)
- 1: high (voltage)

Thresholds:

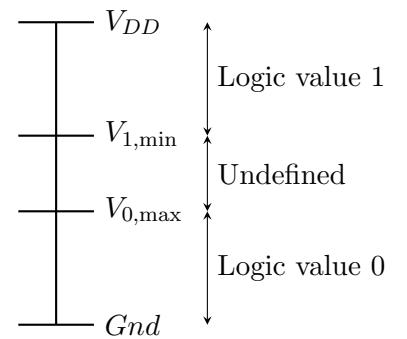
- $V_{0,\max}$ , the max voltage level that the circuit must interpret as *low*
- $V_{1,\min}$ , the min voltage level that the circuit must interpret as *high*

Exact threshold values depend on the technology; typically:

- $V_{0,\max} \approx 0.4V_{DD}$
- $V_{1,\min} \approx 0.6V_{DD}$

Range ( $V_{0,\max}, V_{1,\min}$ ) is undefined

- Logic signals take those intermediate voltage values only while transitioning from one logic value to another.



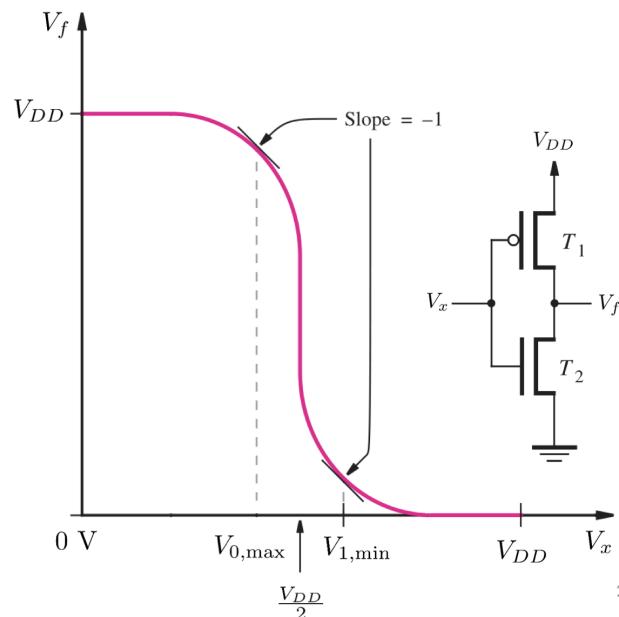
## 9.3 Voltage Transfer Characteristic

### 9.3.1 CMOS Inverter (NOT gate)

The input-output voltage relationship in a real CMOS inverter is summarized by the voltage transfer characteristic.

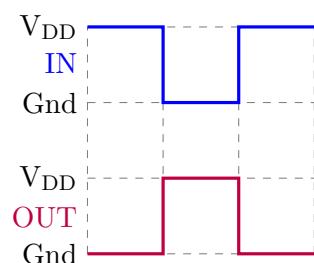
When the slope of the curve is  $-1$ , we have:

- the maximum input voltage that the inverter will interpret as low
- the minimum input voltage that the inverter will interpret as high



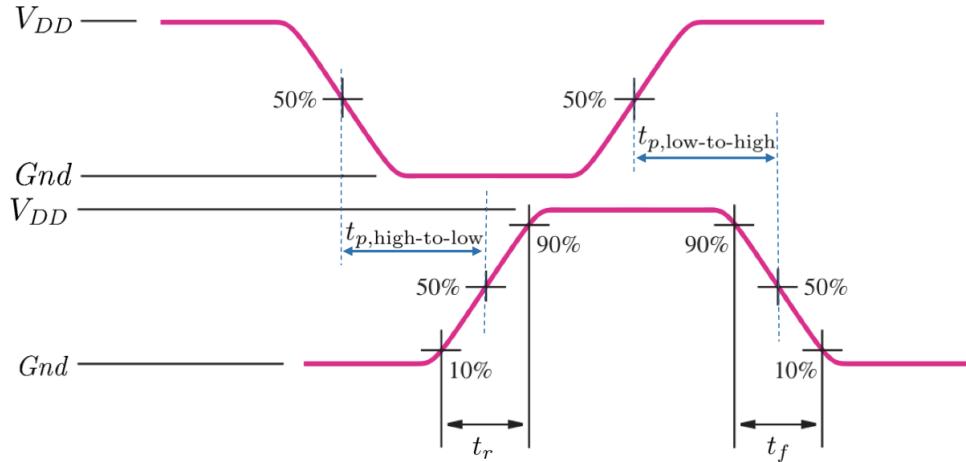
### 9.3.2 Ideal Waveforms and Real Waveforms

Until now, we have considered ideal waveforms in digital circuits like such :



In real gates, the transition delay between GND and V<sub>DD</sub> is not instantaneous. Also, the output is not produced instantaneously, there is a **propagation delay**.

For example, let's consider the Real Waveform of a NOT gate:



With  $t_r$  the rise time,  $t_f$  the fall time,  $t_{p,\text{high-to-low}}$  the propagation delay from high to low, and  $t_{p,\text{low-to-high}}$  the propagation delay from low to high.

Propagation delay  $t_p$  is the time when the voltage is at 50% of the  $V_{DD}$ .

**In general,**  $t_{p,\text{high-to-low}} \neq t_{p,\text{low-to-high}}$

## 9.4 Dynamic Operation

In digital circuits, the operation and performance significantly depend on various factors like the design's ability to handle multiple inputs and outputs, as well as the inherent physical properties of the components used.

### 9.4.1 Fan-In and Fan-Out

**Fan-In** refers to the maximum number of inputs a gate can handle. *eg. fan-in(AND) = 2*

**Fan-Out** refers to the maximum number of gates a gate can drive. *eg. fan-out(NOT) = 1*

### 9.4.2 Parasitic Capacitance

Parasitic capacitance refers to the unintended and inevitable capacitance present in transistor gates, termed as **gate parasitic capacitance** (this is like unwanted electrical 'baggage' that transistors carry which can slow things down and waste energy).

When multiple transistors connect to a single logic gate input, their individual parasitic capacitances combine to form an overall **equivalent per-input capacitance**  $C_{IN}$  (think of this as the total extra load the logic gate has to deal with). This combined capacitance:

The load capacitance ( $C_{LOAD}$ ) that a logic gate must drive is equal to the input capacitance ( $C_{IN}$ ) of the next gate it's connected to, meaning one gate's output has to push against the next gate's inherent electrical resistance to change its state.

Acts as a load (an electrical burden) on the gate that is outputting a signal to this input, affecting its performance.

The equivalent capacitance at a logic gate's input, denoted  $C_{IN}$ , is the aggregate of parasitic capacitances from all transistor gates connected to it. This capacitance directly impacts the circuit's speed.

For example, we've seen that a NOT gate is a CMOS circuit with an NMOS and PMOS transistor. Thus, the parasitic capacitance of the NOT gate is the sum of the parasitic capacitance of the NMOS and PMOS transistors.

### Capacitive Effects and Circuit Speed

**Charging a Capacitor:** The voltage across a capacitor,  $V_c(t)$ , during charging follows the equation:

$$V_c(t) = V_{DD}(1 - e^{-\frac{t}{RC}})$$

where  $V_{DD}$  is the supply voltage,  $R$  is the resistance, and  $C$  is the capacitance.

**Discharging a Capacitor:** Conversely, during discharging, the voltage decays according to:

$$V_c(t) = V_{DD}e^{-\frac{t}{RC}}$$

The speed of a circuit is inversely related to the fan-out, as higher loads increase the charging time of these parasitic capacitors.

### 9.4.3 Power Dissipation

We calculate  $E_c$  the energy dissipated by the circuits and  $P_D$  the power dissipated by the circuits as follows:

$$E_{total} = E_{charge} + E_{discharge} = 2 \cdot \frac{1}{2} CV^2 = CV^2 \quad (9.1)$$

$$P_D = fCV^2 \quad (9.2)$$

where  $f$  represents the switching frequency,  $C$  the capacitance, and  $V$  the voltage.

## 9.5 Hazards in Digital Circuits

Hazards are unintended behaviors in digital circuits that can lead to errors in operation:

**Static Hazard:** Occurs when a signal unexpectedly changes levels momentarily, despite being intended to remain constant.



**Dynamic Hazard:** occurs when a signal is supposed to change:  $1 \rightarrow 0$  or  $0 \rightarrow 1$ , occurs if such a change involves a short oscillation before the signal settles into its new level.

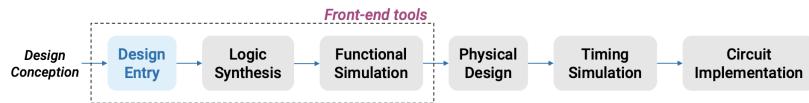


# Chapter 10

## Digital Logic and Verilog (PART V)

### 10.1 CAD Design Flow

The CAD design process encompasses several critical steps, ensuring the accurate realization of digital circuits.



#### Front End Tools:

- **Design Entry:** The journey begins with the design entry, where the initial concept is articulated. This stage leverages intuition and experience, often utilizing Schematic Capture for graphical representations or Hardware Description Languages (HDLs) like Verilog and VHDL for textual descriptions.
- **Logic Synthesis:** The design is then transformed into a logic gate structure, where HDL codes are converted into networks of logic gates. This process not only mirrors the intended circuit functionality through logic expressions but also optimizes the design for speed, size, and power efficiency.
- **Functional Simulation:** This phase verifies the logical correctness of the design by simulating logic functions. Assuming perfect gates, it generates timing waveforms for detailed analysis, ensuring the design operates as expected under ideal conditions.

#### Back End Tools:

- **Physical Design:** Subsequently, the focus shifts to the physical layout, where the logic expressions are mapped onto a chip using available components. This involves the strategic placement of components and routing connections between them.
- **Timing Simulation:** This step is crucial to ensure the design meets all timing constraints, accounting for the physical realities of circuit implementation.
- **Circuit Implementation:** The final step is the circuit's physical realization, where the design is brought to life in hardware form.

## 10.2 Verilog HDL

*Note: See the chapter at the end of the notes for all content on Verilog (Verilog chapter).*

*Personal Remark. this course will absolutely NOT suffice to make you a Verilog expert, please take some time to practice in an empty project, just representing in both structural and behavioral ways the circuits we've seen so far.*

### 10.2.1 Structural Modeling with Logic Gates

In structural modeling, we use predefined modules (*built-in representations of basic logic gates*) to construct complex circuits.

We use this logic gate instantiation statement to create a basic logic gate:

```
1 gate_name [instance_name] (out_port, in_port{, in_port});
```

[] indicates an optional parameter.

() indicates a required parameter.

{ } indicates that additional parameters can be added.

With *gate\_name* as the type of gate :

(*not limited to these...*)

and	nor
nand	buf
xor	not
or	xnor

#### Examples

##### - AND Gate:

```
1     and and1 (out, in1, in2);
```

##### - OR Gate:

```
1     or or1 (out, in1, in2);
```

##### - NOT Gate:

```
1     not not1 (out, in);
```

##### - NAND Gate:

```
1     nand nand1 (out, in1, in2);
```

## 10.2.2 Verilog Syntax

### In a Nutshell

#### Naming Rules:

Begin with a letter.

Include letters, digits, underscore (\_), and dollar sign (\$).

#### Case Sensitivity:

Lowercase and uppercase are distinct (e.g., a ≠ A).

#### Style Guidelines:

Syntax is flexible with white spaces and line breaks.

Prioritize readability with proper indentation.

#### Comments:

Initiated by double slashes (//).

## 10.2.3 Modules in Verilog

A circuit or subcircuit described in Verilog is encapsulated within a module. The module declaration includes the module name and its ports, which are the input and output connections to the module.

```

1 module module_name [(port_name{, port_name})];
2 [input declarations]
3 [output declarations]
4 [wire declarations]
5 [logic gate instantiations]
6 [module instantiations]
7 endmodule

```

*(words in purple are reserved keywords, words after two slashes are comments)*

## 10.2.4 Ports in Verilog

Ports are the input and output connections of a module. They are declared within the module declaration. They can be in the following directions : **Port Types**:

- input – for receiving signals.
- output – for sending signals.
- inout – for bi-directional signal flow.

### Port Declaration:

- Syntax:

```

1 port\direction data\_type [port\_size] port\_name;

```

- Implicit type: Unspecified types default to wire.

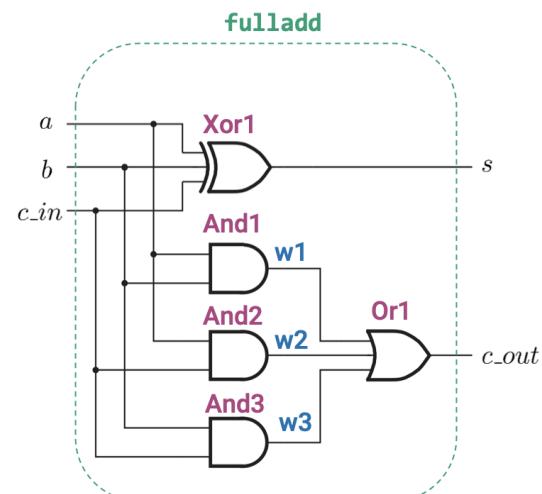
- Vectors: Specify bit-width for multi-bit signals (e.g., [3:0] for 4 bits).

### Examples:

```
input diff;           // Single-bit input named diff
inout [15:0] data;   // 16-bit bidirectional port named data
output [3:0] f;      // 4-bit output named f
```

### 10.2.5 Example - Full adder in Verilog

```
// Structural modeling of a full-
// adder
module fulladd (a, b, c_in, s, c_out)
;
// ----- port definitions -----
input a, b, c_in;
output s, c_out;
// ----- intermediate signals -----
wire w1, w2, w3;
// ----- design implementation -----
and And1 (w1, a, b);
and And2 (w2, a, c_in);
and And3 (w3, b, c_in);
or Or1 (c_out, w1, w2, w3);
xor Xor1 (s, a, b, c_in);
endmodule
```



Which can be simplified to :

```
module fulladd (a, b, c_in, s,
    c_out);
input a, b, c_in;
output s, c_out;
and (w1, a, b);
and (w2, a, c_in);
and (w3, b, c_in);
or (c_out, w1, w2, w3);
xor (s, a, b, c_in);
endmodule
```

### 10.2.6 Subcircuits in Verilog

A Verilog module can be included as a subcircuit in another module.

Modules should be defined in the same source file, or the Verilog compiler must know the locations of the modules.

Module instantiation syntax:

```
{module\_name instance\_name (.port\_name (expression));}
```

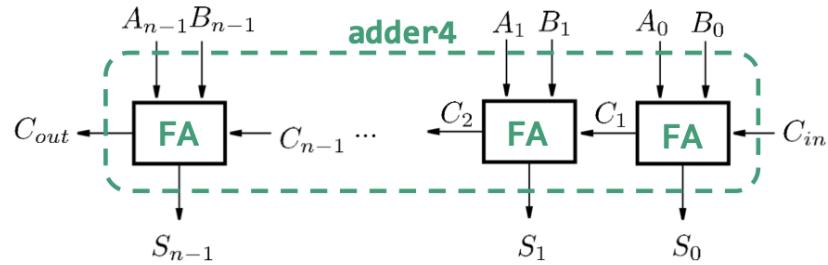
**Notes:** \* module\_name and instance\_name are any valid identifiers.

\* .port\_name specifies the subcircuit's port to be connected.

- \* Omitting `.port_name` is possible if port order is identical to the subcircuit's definition, but not recommended due to error-proneness.

### 10.2.7 Example - Ripple-Carry Adder in Verilog

A four-bit ripple-carry adder is composed of four full-adder stages with interconnections for the carry bits.



Written in VHDL :

```

1 module adder4 (Cin, A, B, S, Cout);
2 // Port definitions
3 input Cin;
4 input [3:0] A, B; // 4-bit vectors
5 output [3:0] S; // 4-bit vector
6 output Cout;
7
8 // Intermediate signals
9 wire [3:1] C; // 3-bit vector for carry bits
10
11 // Design implementation using full-adder stages
12 fulladd stage0 (.c_in(Cin), .a(A[0]), .b(B[0]), .s(S[0]), .c_out(C[1]));
13 fulladd stage1 (.c_in(C[1]), .a(A[1]), .b(B[1]), .s(S[1]), .c_out(C[2]));
14 fulladd stage2 (.c_in(C[2]), .a(A[2]), .b(B[2]), .s(S[2]), .c_out(C[3]));
15 fulladd stage3 (.c_in(C[3]), .a(A[3]), .b(B[3]), .s(S[3]), .c_out(Cout));
16 endmodule

```

# Chapter 11

## Digital Logic and Verilog(PART VI)

### 11.1 Bus Architecture

A bus is a communication system that transfers data between components of a digital device. It is designed to connect parts of the motherboard or to link other hardware. In essence:

- A bus allows sequential data transmission from multiple sources to various destinations.
- Buses usually carry multiple bits simultaneously, typically more than one bit, to transmit data.

Within Verilog design, multiple-bit wires are termed as *vectors*, and single-bit wires are called *scalars*.

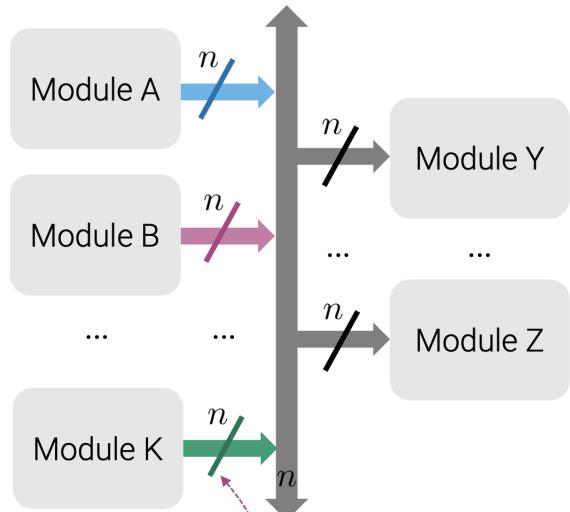
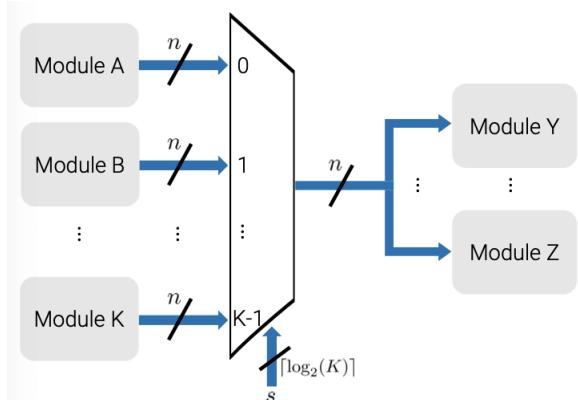


Illustration of an  $n$ -bit bus

#### 11.1.1 Using Multiplexers (MUXes)

Multiplexers, or MUXes, direct signals from several input lines to a single output line. They select which input to transmit based on control signals.

A multiplexer can manage multiple  $n$ -bit data lines by using a select signal of  $m$  bits, where  $m$  is the minimum number of bits required to represent the number of inputs, calculated as  $m = \lceil \log_2(K) \rceil$ .

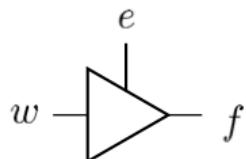


### 11.1.2 Without Multiplexers

Buses enable communication between different modules. However, directly connecting two outputs to one input can cause a short-circuit if they attempt to send conflicting signals. Employing multiplexers or tri-state drivers prevents this by ensuring that only one signal is transmitted at a time.

## 11.2 Tri-State Drivers

A tri-state driver has a data **input**  $w$ , **output**  $f$ , and an **enable**  $e$  input. When inactive, a tri-state driver's output enters a high-impedance state ( $Z$ ), thus three possible states being logical 0, logical 1, or  $Z$ .

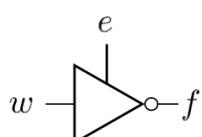


Thus the following table:

e	w	f
0	0	Z
0	1	Z
1	0	0
1	1	1

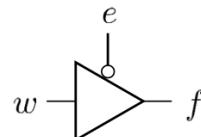
### 11.2.1 Types of Tri-State Drivers

Enable active high with inverted  $f$



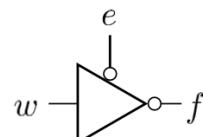
e	w	f
0	0	Z
0	1	Z
1	0	1
1	1	0

Enable active low



e	w	f
0	0	0
0	1	1
1	0	Z
1	1	Z

Enable active low with inverted  $f$



e	w	f
0	0	1
0	1	0
1	0	Z
1	1	Z

## 11.3 Complete Verilog Built-In Gate List

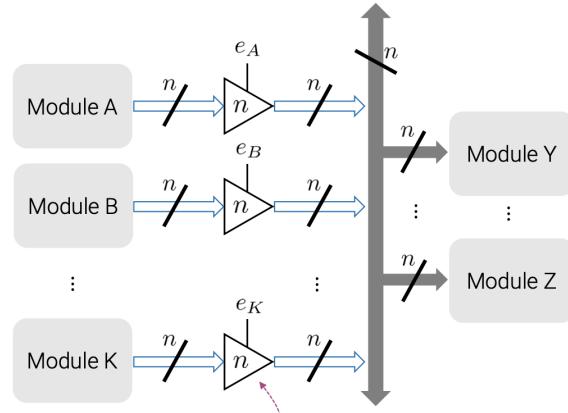
Note: See the chapter at the end of the notes for all content on Verilog (Verilog chapter). Now we can complete the Verilog built-in gate list presented in the last chapter as follows :

and	nor	bufif0(f, w, e)
nand	buf	bufif1(f, w, e)
xor	not	notif0(f, w, e)
or	xnor	notif1(f, w, e)

## 11.4 Implementing a Bus with Tri-State Drivers

In digital systems, a bus with tri-state drivers facilitates selective data transfer between modules without conflict.

Each module connects via a driver that outputs high impedance to avoid bus contention, controlled by unique enable signals. (*simplified: one signal at a time*)



## 11.5 Verilog Part 2.

*Note: See the chapter at the end of the notes for all content on Verilog (Verilog chapter).*

### 11.5.1 Scalar Signal Values

Verilog supports one-bit signals (scalars), each signal can have one of the following values:

Value	Meaning
0	Logic 0
1	Logic 1
x	Unknown logic value or don't care
z	tri-state, high-impedance

### 11.5.2 Vector Signal Values

The value of a vector variable looks like this

```
1 [size] ['radix]constant
```

*size* is the number of bits, *radix* is the base of the number, and *constant* is the value of the number. Supported radices include:

Radix	Meaning
d	decimal <i>default if no radix is specified</i>
b	binary
h	hexadecimal
o	octal

### 11.5.3 Parameters

Parameters are constants that can be used in the module. They are defined as follows:

```
1 parameter name = value;
```

eg.

```
1 parameter c = 8;
```

### 11.5.4 Nets

Nets are used to connect modules. There are two net types:

wires	tri-states
1 <code>wire [3:0] a;</code>	1 <code>tri z1</code>

### 11.5.5 Behavioral Modeling in Verilog

Behavioral modeling in Verilog is one of the three modeling paradigms used in Verilog to describe digital circuits at a high level. This approach allows designers to specify how circuits should behave through procedural statements, similar to software programming. *Simplified: used when too hard to do in structural modeling.*

### 11.5.6 Bit-wise Operators

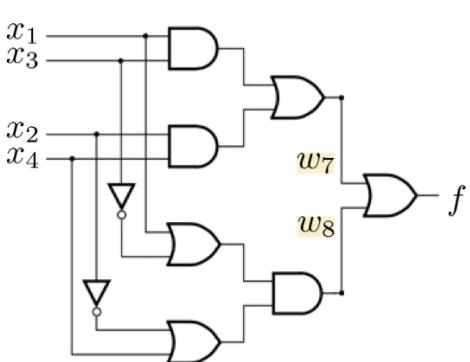
Bit-wise operators in Verilog are used to perform operations on individual bits of a vector. The following are the bit-wise operators available in Verilog:

Operator	Description
&	Bit-wise AND
—	Bit-wise OR
^	Bit-wise XOR
~	Bit-wise NOT

### 11.5.7 Example - Structural to Behavioral Modeling

Here's a circuit we've modeled in structural modeling :

#### Structural



```

1 module my_circuit_structural (
2     input x1, x2, x3, x4,
3     output f
4 );
5     wire w1, w2, w3, w4, w5, w6, w7,
6         w8;
7     and (w1, x1, x3);
8     not (w2, x2);
9     not (w3, x3);
10    and (w4, x2, x4);
11    or (w5, x1, w3);
12    or (w6, w2, x4);
13    or (w7, w1, w4);
14    and (w8, w5, w6);
15    or (f, w7, w8);
endmodule

```

## Behavioral

```

1 module my_circuit_behavioral (
2     input x1, x2, x3, x4,
3     output f
4 );
5     wire w7, w8;
6     assign w7 = (x1 & x3) | (x2 & x4);
7     assign w8 = (x1 | ~x3) & (~x2 | x4);
8     assign f = w7 | w8;
9 endmodule

```

### 11.5.8 Continuous Assignments in Verilog

**Keyword:** assign

#### Definition

Continuous assignments are used in Verilog for assigning values to wires. The assign statement makes the left-hand side (LHS) of the assignment dynamically reflect any changes in the right-hand side (RHS) immediately.

#### Properties

- Executes continuously: The assignment reacts to any change in the RHS variables and updates the LHS accordingly.
- Operates in parallel: The execution order in the code does not affect the simulation behavior.

#### Example

Consider the case where we want to perform a logical OR operation between two signals, w7 and w8, and assign the result to signal f:

```

1 assign f = w7 | w8;

```

In this example, whenever the value of w7 or w8 changes, f is automatically recalculated to reflect the new value.

### 11.5.9 Verilog Always Block

**Keyword:** always

The always block in Verilog is used for modeling combinational and sequential logic. It reacts to changes in input signals, ensuring that the block's internal logic is evaluated whenever necessary.

#### Structure

```

1  always @*
2    [begin]
3      [procedural assignment statements]
4      [if-else statements]
5      [case statements]
6      [while, repeat, and for loops]
7      [task and function calls]
8    [end]
```

In this block, the always keyword is followed by @\*.

### 11.5.10 If-Else Statements

If an expression is true, the code within the begin-end block runs; multiple statements can be grouped in such a block. Optional else if and else clauses, when used, connect to the closest preceding if or else if statement.

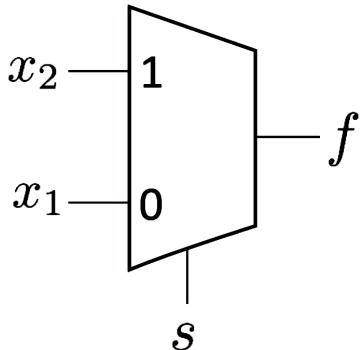
#### Structure

```

1  if (expression1)
2    begin
3      statement;
4    end
5  else if (expression2)
6    begin
7      statement;
8    end
9  else
10   begin
11     statement;
12   end
```

### 11.5.11 Example - 2-to-1 MUX

Using what we've learned so far



```

1 module my_2to1mux ( input w1,
2   w2, s, output reg f );
3   always @*
4     begin
5       if (s == 0)
6         begin
7           f = w1;
8         end
9       else
10      begin
11        f = w2;
12      end
13    end
14 endmodule

```

### 11.5.12 Case Statement

*Personal Remark. Switch in java*

#### Syntax

```

1 case (expression)
2   alternative1:
3     begin
4       statements;
5     end
6   alternative2:
7     begin
8       statements;
9     end
10  default:
11    begin
12      statements;
13    end
14 endcase

```

### Example - Full Adder

Using the truth table of a Full-Adder, let's use a case statement to implement it:

x	y	C <sub>in</sub>	s	C <sub>out</sub>
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

```

1 module fulladd (input x, y, Cin,
2   output reg s, Cout);
3   always @*
4     begin
5       case ({x, y, Cin})
6         3'b000: {s, Cout} = 'b00;
7         3'b001: {s, Cout} = 'b10;
8         3'b010: {s, Cout} = 'b10;
9         3'b011: {s, Cout} = 'b01;
10        3'b100: {s, Cout} = 'b10;
11        3'b101: {s, Cout} = 'b01;
12        3'b110: {s, Cout} = 'b01;
13        3'b111: {s, Cout} = 'b11;
14      endcase
15    end
16 endmodule

```

# Chapter 12

## Digital Logic and Verilog(PART VII)

### 12.1 Memory Elements

#### 12.1.1 Introduction - Example Application: Alarm System Control

Suppose we wish to control an alarm system. The system design involves a **sensor**, an **alarm control circuit**, and an **alarm** indicator. Below is the functional description of the system:



Figure 12.1: Diagram of the Alarm System Control

- The **sensor** detects an undesirable event and sends a **SET** signal (logic high or 1) to the **alarm control circuit**.
- The **alarm control circuit** activates the alarm when it receives the **SET** signal. The **ALARM\_ON** signal becomes 1 (logic high), indicating that the alarm is active.
- The alarm remains active until a **RESET** signal (logic high or 1) is sent to the **alarm control circuit**, turning off the alarm (**ALARM\_ON** becomes 0).
- A memory element in the circuit ensures that the alarm remains active until it is explicitly reset, even if the **SET** signal goes back to low (0).

### 12.1.2 Combinational vs. Sequential Circuits

*Personal Remark.* Please take some time to actually identify the differences between the two as they will be crucial in understanding the rest of the course

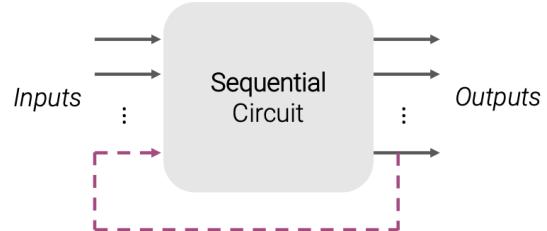
Previously, in **combinational circuits**, outputs only depended on the current inputs.

In **sequential circuits**, outputs are determined by both the current inputs and the circuit's past behavior, due to the inclusion of memory elements.

Combinational Circuits are memory-less; outputs only depend on the current inputs.



Sequential circuits have memory elements, outputs depend on both current inputs and past behavior.

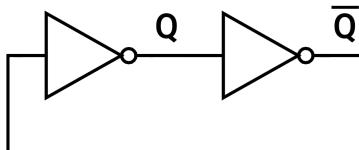


*Storage elements in a circuit hold logic signal values, defining the circuit's state. When inputs change, the circuit may maintain its current state or transition to a new state. Consequently, the circuit evolves through various states over time based on input changes.*

### 12.1.3 Basic Memory Element

#### Bistable Element

We've seen that two Not gates in a series doesn't change the input signal, so, what if we connect the output of the second Not gate to the input of the first Not gate?



$Q$	$\bar{Q}$
0	1
1	0

While this already allows the most basic form of memory, it's not very practical, the given value cannot be changed, and it's not very reliable.

### Set-Reset Latch with Reset Priority

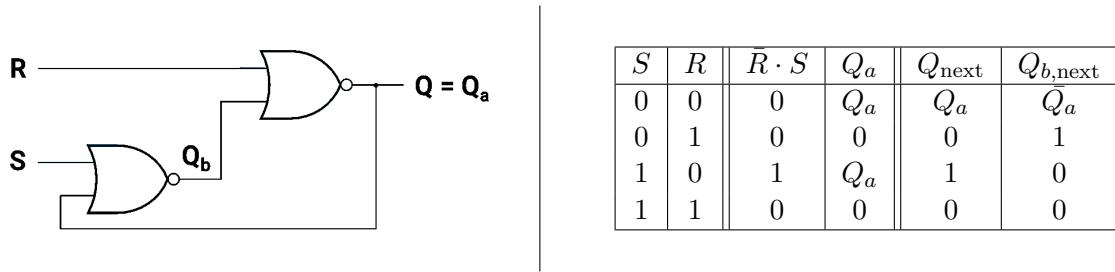
The set-reset latch is a memory element that can store one bit of information. It has two inputs: S (set) and R (reset). The latch retains its state until a new input is received.

#### Components of the Set-Reset Latch:

- *Set Input (S)*: When S is high (1), the latch sets to 1.
- *Reset Input (R)*: When R is high (1), the latch resets to 0.
- *Q Output*: The current state of the latch.
- *Q-bar Output*: The inverse of the current state of the latch.

*What happens when both S and R are high?*

$R = 1, S = 1$ : This is generally considered an unwanted or invalid situation for SR latches because it tries to set and reset the latch at the same time, leading to unpredictable behavior. Some designs define this state specifically to avoid ambiguity.

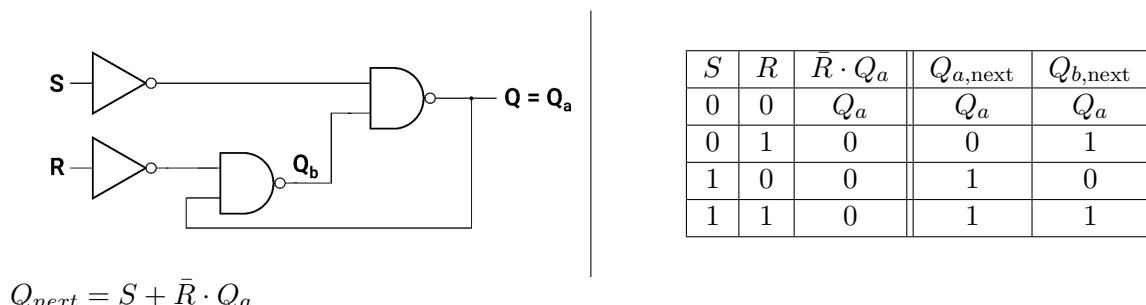


This is how the next state of the latch is calculated, the current state of the latch is  $Q_a$  and  $Q_b$  and the next state is  $Q_{\text{next}}$  and  $Q_{b,\text{next}}$  respectively.

$$\begin{aligned}
 Q_{a,\text{next}} &= \overline{R + Q_b} = \overline{R + \overline{S + Q_a}} \\
 &= \overline{R} \cdot (S + Q_a) \\
 &= \overline{R} \cdot S + \overline{R} \cdot Q_a = Q_{\text{next}} \\
 Q_b &= \overline{S + Q_a}
 \end{aligned}$$

### Set-Reset Latch with Set Priority

The set-reset latch with set priority is similar to the reset-priority latch, but with the priority reversed. The latch sets to 0 when both S and R are high (1).



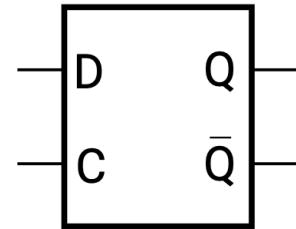
$$Q_{\text{next}} = S + \bar{R} \cdot Q_a$$

## 12.2 Latches with a Control Signal

Gated latches use a control signal to determine when to change their state. When active, the control signal allows the latch to update its state; when inactive, it prevents changes. This helps avoid unwanted oscillations.

### 12.2.1 Gated D Latch

When the control signal C is high, the output Q mirrors changes in input D. Otherwise, Q remains constant, retaining its previous value.

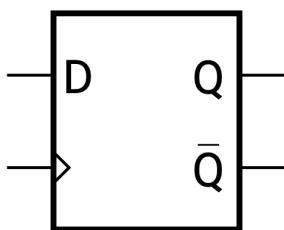


### 12.2.2 Flip-Flops

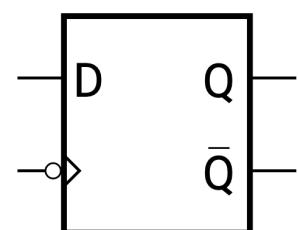
*Personal Remark.* Put simply, the first type requires actual C input to determine when to update, while the other type updates Q in response to an event-driven signal, such as a clock signal (refer to the following section about Clock signals for a clearer explanation).

#### D Flip-Flop

Flip-flops update outputs at specific times, unlike latches. A D flip-flop captures the input from the D signal only during certain clock signal transitions, ensuring stable and predictable outputs. Once the clock triggers (on a rising or falling edge), the output Q immediately matches the input D and maintains this value, unaffected by any further input changes, until the next clock edge.



D Flip-Flop sensitive to the rising edge



D Flip-Flop sensitive to the falling edge

## 12.3 Clock

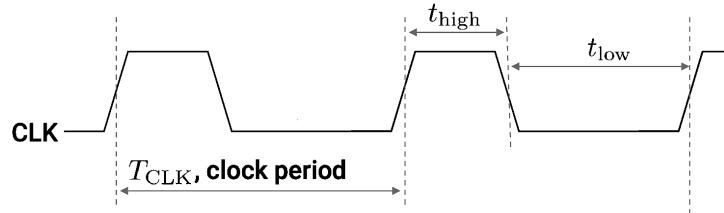
Clocks are essential in digital systems for synchronization purposes. The timing of state changes within the system is orchestrated by these clocks.

### 12.3.1 Clock Signal

A clock signal is a periodic waveform that oscillates between a high state and a low state. It is characterized primarily by its frequency, which dictates how many cycles occur per second, and its duty cycle, which describes the proportion of time the signal is in the high state within a single cycle.

### 12.3.2 Waveform

The waveform of a clock signal can be described as follows:



- **Period:** The period of a clock signal is the duration of one complete cycle of the waveform and is typically measured in seconds or fractions thereof (like nanoseconds, ns). It is denoted by  $T_{CLK}$ .
- **Frequency:** The frequency of a clock signal is defined as the number of complete cycles it performs per second. Typical units of measurement are Hertz (Hz). Calculated as  $f = \frac{1}{T_{CLK}}$ , where  $T_{CLK}$  is the period.
- **Duty Cycle:** The duty cycle is the percentage of the period during which the clock signal remains in the high state (often 50%). Calculated as  $D = \frac{T_{high}}{T_{CLK}} \times 100\%$ .

The transition points of the clock signal, especially the rising edge, often trigger state changes within the digital system, coordinating operations and ensuring the system functions correctly and predictably.

## 12.4 Verilog Part 3.

*Note: See the chapter at the end of the notes for all content on Verilog (Verilog chapter).*

### 12.4.1 Update to Always Block

Updates can take place one after the other, or concurrently, depending on the assignment used : blocking (=) or nonblocking ( $\leq$ ).

#### Blocking Assignment

For example: Here, B is assigned the value of A, then C is assigned the value of B, and finally, D is assigned the value of C. At the end,  $A = B = C = D$ . This is used to model combinational logic (without memory).

```

1 always @ (...sensitivity list...)
2 begin
3   B = A;
4   C = B;
5   D = C;
6 end

```

## Nonblocking Assignment

For example: Changes occur at the same time, so B is assigned the value of A, C is assigned the *old* value of B, and D is assigned the *old* value of C. This is used to model sequential logic (with memory).

```

1  always @ (...sensitivity list...)
2  begin
3      B <= A;
4      C <= B;
5      D <= C;
6  end

```

## always(\*) Blocks

(=) Block Statements should be used.

The `always @*` block is used to model combinational logic, where the output depends only on the current input values. It is sensitive to any change in the input signals.

```

// Example: AND gate
1  always @ (*)
2  begin
3      C = A & B;
4  end

```

## always@(posedge Clock) Blocks

(≤) Nonblocking Statements should be used.

Used to describe sequential logic containing flip-flops

*Note: Remember,  $\leq$  like the arrow in the Control Input symbol of a D Flip-Flop representation, it's a reminder that the assignment is non-blocking.*

- `always@(posedge Clock)` – always at the rising clock edge
- `always@(negedge Clock)` – always at the falling clock edge

## 12.5 Behavioral Latch and Flip-Flop Models

### Basic D Latch

The output may be affected whenever inputs D or C change (hence the \*) If input C is 0, we do nothing, there is no else clause.

```

1  module Dlatch (
2      input D,
3      input C,
4      output reg Q
5  );
6  always @ (*)
7  begin
8      if (C == 1)
9          Q <= D;
10     end
11 endmodule

```

## D Latch w/ Enable and Asynchronous Clear

A D Latch with Enable and Asynchronous Clear is a type of digital storage device used in electronic circuits to store a single bit of data. It has a data input (D), a clock input (C), an enable input (CE), and a clear input (CLR). The output (Q) reflects the input data (D) when both the clock (C) and enable (CE) inputs are active (high). The asynchronous clear (CLR) has the highest priority; when activated (high), it immediately resets the output (Q) to 0, regardless of other input conditions. Thus the Verilog code for this is:

```

1 module Dlatch (input D, input C, input CE, input CLR, output reg Q);
2   always @ (*)
3     begin
4       if (CLR == 1)
5         Q <= 0;
6       else if ((C == 1) && (CE == 1))
7         Q <= D;
8     end
9   endmodule

```

## Positive-Edge-Triggered D FF

D Flip-Flop with positive-edge-triggered clock input. The always block is executed on the positive (rising) CLK edge Q gets set to D

```

1 module Dff (input D, input CLK, output reg Q);
2   always @ (posedge CLK)
3     begin
4       Q <= D;
5     end
6   endmodule

```

## D FF (Positive-Edge-Triggered) with Asynchronous Clear

Works the same as before but the use of posedge here means that the always block is also executed if the CLR signal is high immediately without waiting for the next clock edge.

```

1 module Dff (input D, input CLK, input CLR, output reg Q);
2   always @ (posedge CLK or posedge CLR)
3     begin
4       if (CLR == 1)
5         Q <= 0;
6       else
7         Q <= D;
8     end
9   endmodule

```

*Note: It would be a mistake to omit posedge and execute the always block on any change in CLR. On a 1-to-0 transition on CLR, the else clause would execute and set Q to D even though no CLK edge had occurred*

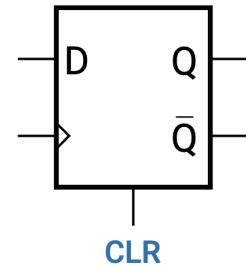
## D FF with Asynchronous Clear and $\sim Q$ Output

The exact same but with an additional output  $\sim Q$  which is the complementary of Q.

```

1 module Dff (input D, CLK, CLR, output
2   reg Q, QN);
3   always @ (posedge CLK or posedge
4     CLR)
5   begin
6     if (CLR == 1) begin
7       Q <= 0;
8       QN <= 1; // complementary
9       output
10    end
11   else begin
12     Q <= D;
13     QN <= ~D; // complementary
14     output
15   end
16 end
17 endmodule

```



## D FF with Asynchronous Clear and Synchronous Preset

A D Flip-Flop with Clock Enable and Synchronous Preset updates its output to the data input (D) at each active clock edge (CE) if enabled, or sets the output to a preset value (S) synchronously if the preset condition is active.

If neither S nor CE is active, the output retains its previous value.

```

1 module Dff (input D, input CLK, input S, input CE, output reg Q);
2   always @ (posedge CLK)
3     if (S == 1)
4       Q <= 1;
5     else if (CE == 1)
6       Q <= D;
7   endmodule

```

## Keywords Summary

Now what we've seen is just a composition of the following keywords, here's a summary of them:

**Asynchronous Clear:** Allows the immediate resetting of the latch or flip-flop output to a low state (0), regardless of the clock or other control signals' states.

**Enable:** An additional control input that allows the latch to either accept new data when high (enabled) or maintain its current state when low (disabled).

**Positive-Edge Triggered:** Indicates that the flip-flop updates its output only at the rising edge of the clock signal. This specificity in timing helps to synchronize the data flow in digital systems, reducing ambiguity and ensuring stability.

**Synchronous Preset:** Similar to asynchronous clear, but the preset operation occurs in sync with the clock. This feature sets the output to a high state (1) during the clock's active edge.

**Complementary Output:** Provides an additional output that is the logical inverse of the main output. This feature simplifies the design of circuits that require the inverse logic state, reducing the need for external inverting components.

## 12.6 Practical notes

For effective sequential systems design, it is crucial to avoid latches due to their susceptibility to glitches, potential for output oscillation, and level sensitivity, which can result in multiple output changes within a single clock period.

Instead, use only D flip-flops and combinational logic. Implement separate `always` blocks, one set for simple D flip-flops and another for potentially complex combinational logic.

Latches (hold of the last value due to bad timing, and if the last value is the initial one, it must be defined) can inadvertently arise in combinational circuits. To prevent this, ensure all variables in `always@(*)` blocks are assigned initial default values, ensuring consistent logic states before proceeding to specific assignments.

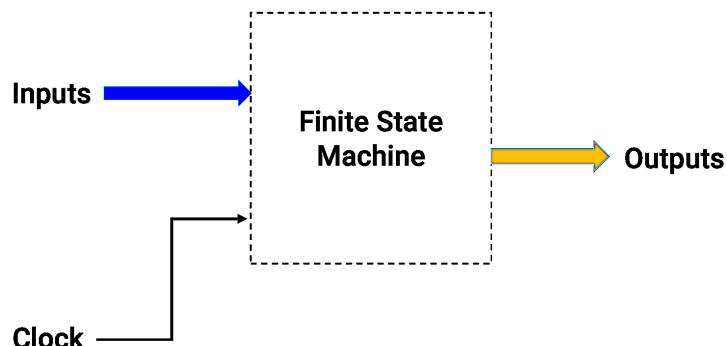
In sequential logic, always employ nonblocking assignments using the `<=` operator in `always` blocks to ensure simulation consistency across multiple blocks, as this method evaluates all right-hand sides before assigning values to the left-hand sides.

# Chapter 13

## Digital Logic and Verilog(PART VIII)

### 13.1 Finite State Machines

Circuits containing both sequential and combinational logic are known as **state machines**.  $2^n$  states can be represented by n-bit binary numbers (nothing new). The number of states being finite we thus call such circuits **finite state machines (FSMs)**



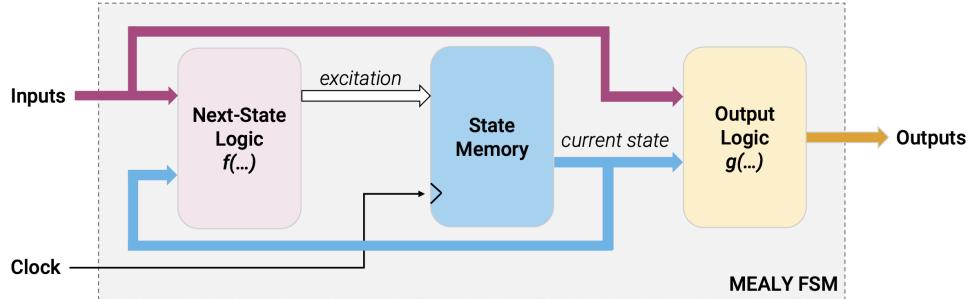
There are two types of FSMs:

- **Mealy Machine:** The output depends on the current state and the current input.
- **Moore Machine:** The output depends only on the current state.

All state transitions are made in sync with clock edges (rising or falling). *Personal Remark: Mnemothechnics: the double o in Moore: Only Output State. (uhm couldn't find anything better)*

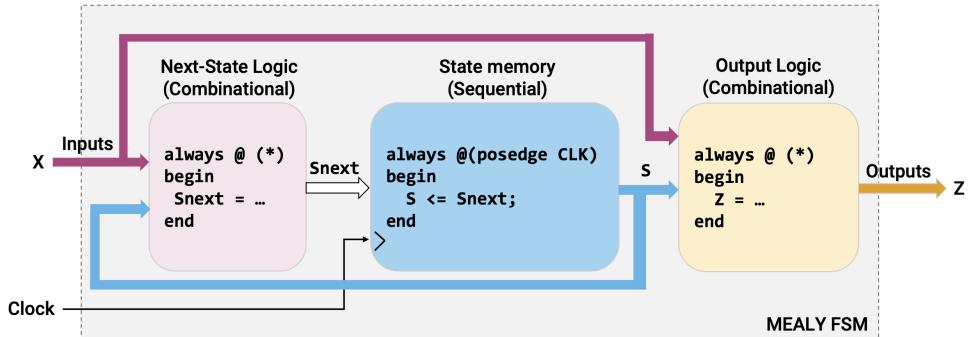
### 13.1.1 Mealy State Machine

In a Mealy state machine, the output is a function of both the current state and the current input.



- The **state memory** is a set of  $n$  flip-flops that store the current state all connected to the same clock (thus updating simultaneously once per clock period).
- The **next state** is a function of inputs and the current state  
eg. Next state =  $f(\text{current state}, \text{input})$
- The **output** is a function of the current state and inputs  
eg. Output =  $g(\text{current state}, \text{input})$

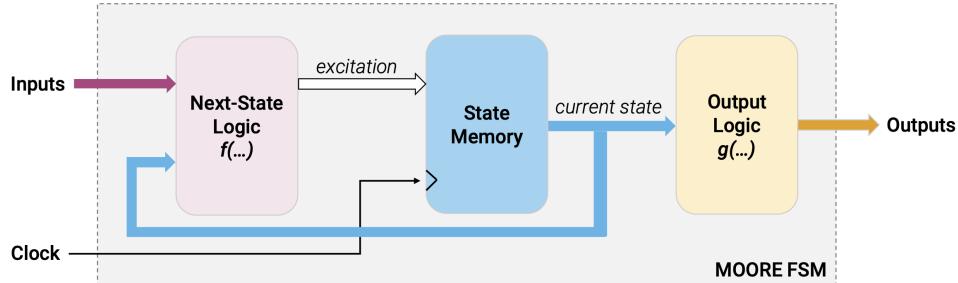
In a more detailed view:



*Note: The sensitivity list of the state machine may be adjusted as we've seen*

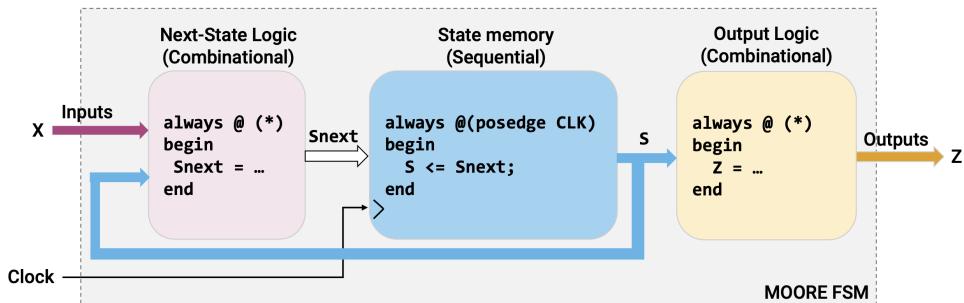
### 13.1.2 Moore State Machine

*Same, but without the input dependency in the output.* In a Moore state machine, the output is a function of the current state only.



- The **state memory** is a set of  $n$  flip-flops that store the current state all connected to the same clock (thus updating simultaneously once per clock period).
- The **next state** is a function of inputs and the current state  
eg. Next state =  $f(\text{current state}, \text{input})$
- (*Difference is here*) The **output** is a function of the current state **only**  
eg. Output =  $g(\text{current state})$

In a more detailed view:



*Note: The sensitivity list of the state machine may be adjusted as we've seen*

## 13.2 State Machine Analysis

Here we'll be looking at how we can analyze a state machine (a circuit containing both sequential and combinational logic) to determine its behavior.

### Algorithm:

Three steps :

**Step 1:** We determine the next-state and output functions eg.  $f()$  and  $g()$

**Step 2:** Use  $f()$  and  $g()$  to construct a state/output table that shows the entirety of the next state and the output of the circuit for all possible inputs and states.

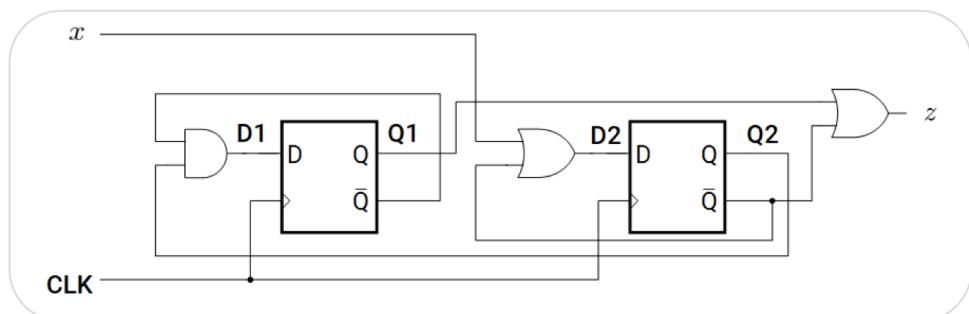
**Step 3 (Optional):** Draw the state diagram of the state machine using the state/output table.

**Step 3 (Optional):** The state diagram is the simplest conceptual method to describe the behavior of a sequential circuit. It is a graph representing circuit states as nodes and the transitions between states as directed edges.

- **Nodes:** Represent the circuit states.
  - Annotated with state names.
  - In Moore FSMs, nodes are also annotated with output values.
- **Edges:** Represent the transitions between states.
  - Annotated with signals causing state transitions.
  - In Mealy FSMs, edges are also annotated with output values.

### Example 1

Let's analyze the state machine below:



As this machine Only Outputs State, it is a Moore state machine.

**Step 1:** Determine the next-state and output functions.

Let  $Q_1$  and  $Q_2$  be the current state, and  $Q_1^*$  and  $Q_2^*$  be the next state. The output is  $z$ .

$$Q_1^* = D_1 = f_1(x, Q_1, Q_2) = \overline{Q}_1 \cdot \overline{Q}_2$$

$$Q_2^* = D_2 = f_2(x, Q_1, Q_2) = x + \overline{Q}_2$$

$$z = g(x, Q_1, Q_2) = Q_1 + \overline{Q}_2$$

State variables	Excitation D2D1		Outputs
	Q2	Q1	
	0	1	x
0 0	1 0	1 0	1
0 1	1 0	1 0	1
1 0	0 1	1 1	0
1 1	1 0	1 0	1

**Step 2:** Construct a state/output table.

Let A, B, C, and D represent the states 00, 01, 10, and 11, respectively.

Current State, S	Next state, S*		Outputs
	S	x	
	0	1	z
A	C	C	1
B	C	C	1
C	B	D	0
D	A	C	1

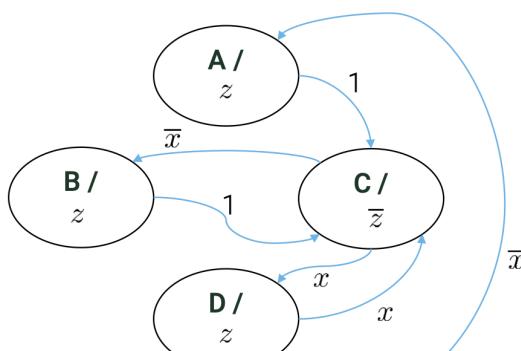
**Step 3:** Draw the state diagram.

**Written in Verilog:**

```

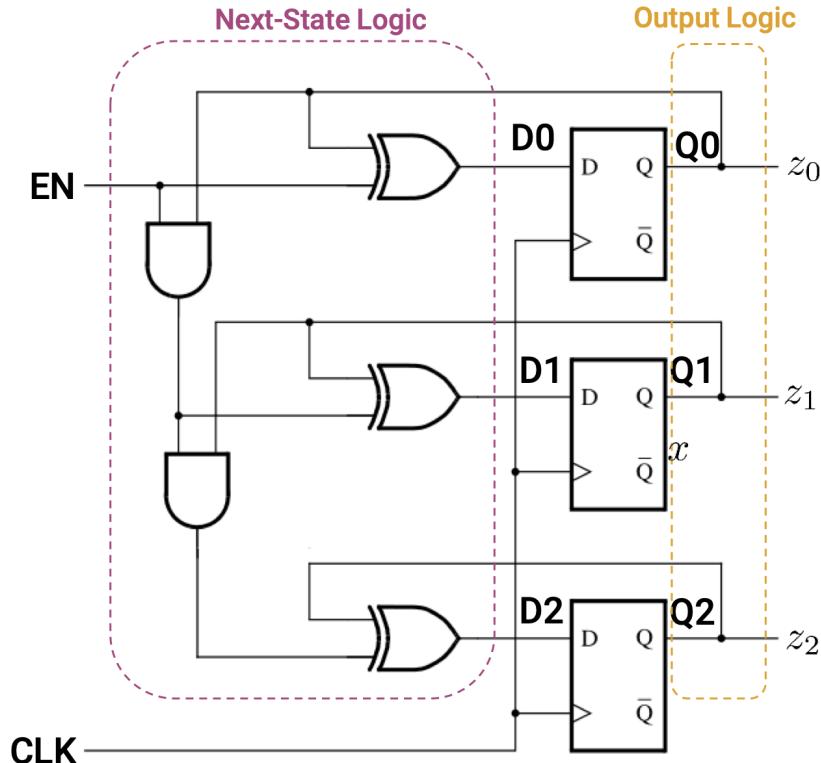
1 module fsm_example1 (input x,
2   input CLK, output reg z);
3   reg [2:1] D, Q;
4
5   // Next-state logic
6   always @ (*) begin
7     D[1] = ~Q[1] & Q[2];
8     D[2] = x | ~Q[2];
9   end
10
11  // State memory
12  always @ (posedge CLK) begin
13    Q <= D;
14  end
15
16  // Output logic
17  always @ (*) begin
18    z = Q[1] | ~Q[2];
19  end
endmodule

```



**Example 2**

Let's analyze the state machine below:



As this machine Only Outputs State, it is a Moore state machine.

**Step 1:** Determine the next-state and output functions.

Let  $Q_0, Q_1, Q_2$  the current state, and  $Q_0^*, Q_1^*, Q_2^*$  be the next state. The output is  $z$ .

$$Q_0^* = D_0 = f_0(EN, Q_0, Q_1, Q_2) = Q_0 \oplus EN$$

$$Q_1^* = D_1 = f_1(EN, Q_0, Q_1, Q_2) = Q_1 \oplus (Q_0, EN)$$

$$Q_2^* = D_2 = f_2(EN, Q_0, Q_1, Q_2) = Q_2 \oplus (Q_0 Q_1 EN)$$

$$z = g_i(Q_i) = Q_i, 0 \leq i \leq 2$$

State variables	Excitation D2D1		Outputs $z$
	$x$		
$Q_2$ $Q_1$ $Q_0$	0	1	
0 0 0	000	001	1
0 0 1	001	010	0
0 1 0	010	011	1
0 1 1	011	100	0
1 0 0	100	101	1
1 0 1	101	110	0
1 1 0	110	111	1
1 1 1	111	000	0

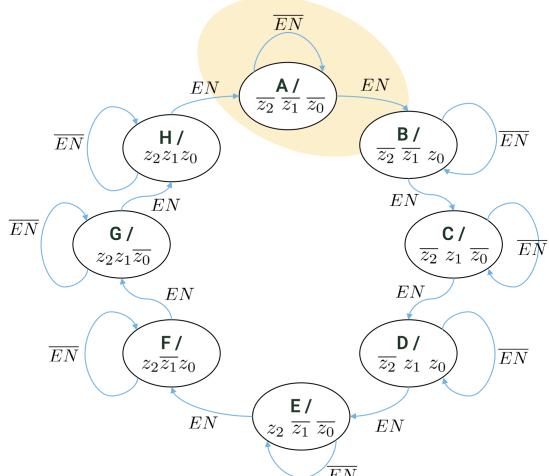
**Step 2:** Construct a state/output table.

Let A, B, C, D, E, F, G, and H represent the states 000, 001, 010, 011, 100, 101, 110, and 111, respectively.

Current State, S	Next state, S*		Outputs $z_2 \ z_1 \ z_0$
	EN 0	EN 1	
A	A	B	0 0 1
B	B	C	0 1 0
C	C	D	0 1 1
D	D	E	1 0 0
E	E	F	1 0 1
F	F	G	1 1 0
G	G	H	1 1 1
H	H	A	0 0 0

**Step 3:** Draw the state diagram.

Written in Verilog:



```

1 module fsm_example2 ( input EN, input
2   CLK, output reg [2:0] z);
3   reg [2:0] D, Q;
4   // Next-state logic
5   always @ (*) begin
6     D[0] = Q[0] ^ EN;
7     D[1] = Q[1] ^ (Q[0] & EN);
8     D[2] = Q[2] ^ (Q[1] & Q[0] & EN);
9   end
10  // State memory
11  always @ (posedge CLK) begin
12    Q <= D;
13  end
14  // Output logic
15  always @ (*) begin
16    z = Q;
17  end
endmodule

```

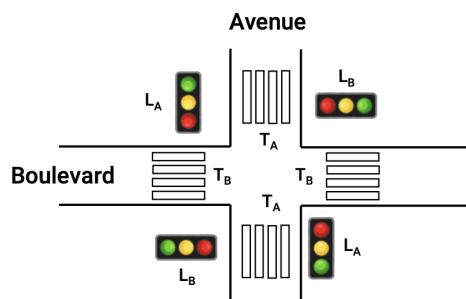
### 13.3 State Machine Synthesis

#### FSM Design Algorithm

- x Identify FSM inputs and outputs.
- x Draw the state diagram.
- x Construct the state/output table.
  - Select state encodings (binary vectors)
  - Rewrite state table with state encodings
  - Write logic expressions for next state
- x Construct output table
  - Select output encodings (binary vectors)
  - Write logic expressions for output
- x Construct the equivalent logic circuit (drawing, Verilog)

#### Example

Consider that we need to design a traffic light controller. Let  $T_A, T_B$  : Signals from the traffic sensors; active when there is traffic  $L_A, L_B$  : Signals for controlling traffic lights



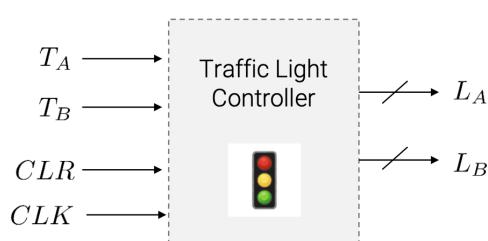
#### Identify FSM inputs and outputs:

Two one-bit inputs from the traffic sensors:  $T_A, T_B$

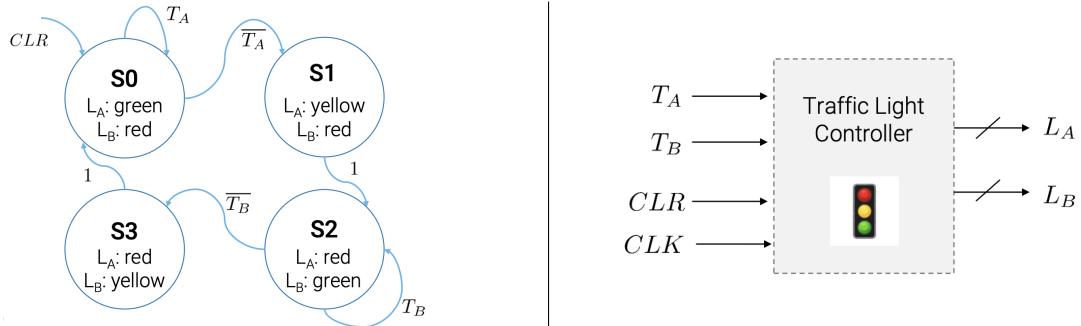
Multiple bits outputs(three states: red, yellow, green):  $L_A, L_B$

CLK, system clock

CLR, asynchronous reset for clearing the state memory



Draw the state diagram:



Construct the state/output table:

Current State, S	Inputs	Next State, S*
S	T <sub>A</sub> T <sub>B</sub>	S*
S0	1 X	S0
S0	0 X	S1
S1	X X	S2
S2	X 1	S2
S2	X 0	S3
S3	X X	S0

X stands for 0/1 (i.e., both options), this simplifies the table and avoids redundancy.

### Select state encodings

There are four states, thus 2 bits to encode all states

State	Encoding
S0	00
S1	01
S2	10
S3	11

Note: Choice of encoding impacts implementation; in practice we let tools infer best encoding (AICC II)

Rewrite state table, write logic expressions for next state:

Current State, S		Inputs		Next State, S*	
Q1	Q0	T <sub>A</sub>	T <sub>B</sub>	D1	D0
0	0	1	X	0	0
0	0	0	X	0	1
0	1	X	X	1	0
1	0	X	1	1	0
1	0	X	0	1	1
1	1	X	X	0	0

Using SOP expressions for the next state :

$$Q_1^* = D_1 = \overline{Q_1}Q_0 + Q_1\overline{Q_0} + Q_1\overline{Q_0} = Q_0 \oplus Q_1$$

$$Q_0^* = D_0 = \overline{Q_1}Q_0T_A + Q_1\overline{Q_0}\overline{T_B}$$

**Construct output table; select output encodings:**

It's a Moore FSM as it Only Outputs State

Three different outputs, hence two bits required to represent them

- $L_A$ : Two bits ( $L_{A1}$ ,  $L_{A0}$ )
- $L_B$ : Two bits ( $L_{B1}$ ,  $L_{B0}$ )

**For the encoding given in the table:**

If  $L_A$  is green:  $L_{A1} = 0$ ,  $L_{A0} = 0$ ;

If  $L_B$  is yellow:  $L_{B1} = 0$ ,  $L_{B0} = 1$ ;

...

Output	Encoding
GREEN	00
YELLOW	01
RED	10

**Construct output table; write logic expressions**

Current State, S	Outputs			
	$L_{A1}$	$L_{A0}$	$L_{B1}$	$L_{B2}$
0 0	0	0	1	0
0 1	0	1	1	0
1 0	1	0	0	0
1 1	1	0	0	1

Thus the Output logic expressions are:  $L_{A1} = Q_1$ ,  $L_{A0} = \overline{Q}_1 Q_0$ ,  $L_{B1} = \overline{Q}_1$

**Written in Verilog:**

```

1 module traffic_light_ctrl (input TA, TB, CLR, CLK, output reg [1:0] LA, LB)
2   ;
3   reg [1:0] D, Q;
4   // Next-state logic
5   always @ (*) begin
6     D[1] = (~Q[1] & Q[0]) | (Q[1] & ~Q[0]);
7     D[0] = (~Q[1] & ~Q[0] & ~TA)
8       | (Q[1] & ~Q[0] & ~TB);
9   end
10  // State memory
11  always @ (posedge CLK or posedge CLR)
12  begin
13    if (CLR == 1)
14      Q <= 0;
15    else
16      Q <= D;
17  end
18  // Output logic
19  always @ (*) begin
20    LA[1] = Q[1]; LA[0] = ~Q[1] & Q[0];
21    LB[1] = ~Q[1]; LB[0] = Q[1] & Q[0];
22  end
endmodule

```

# Chapter 14

## Digital logic and Verilog, Part IX

### 14.1 Sequential Circuits

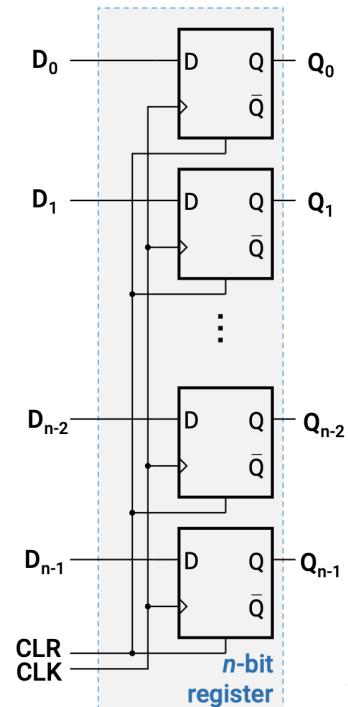
#### 14.1.1 Registers

One flip-flop stores one bit of information; thus, for  $n$  bits we need  $n$  flip-flops.

$n$ -bit structures consisting of flip-flops are commonly referred to as **registers** (with  $2^n$  distinct states).

*Note:*

- Clock (CLK) is common to all flip-flops.
- Reset (CLR) is common to all flip-flops.



### 14.1.2 Registers (Verilog)

*Note: See the chapter at the end of the notes for all content on Verilog (Verilog chapter).*

```

1 module regn (D, CLK, CLR, Q);
2 parameter n = 16; // 16 as an example
3 input [n-1:0] D;
4 input CLR, CLK;
5 output reg [n-1:0] Q;
6 always @ (posedge CLK or posedge CLR)
7 begin
8   if (CLR == 1)
9     Q <= 0;
10  else
11    Q <= D;
12 end
13 endmodule

```

## 14.2 Shift Registers

### 14.2.1 Serial Input, Serial Output Shift Register

*Personal Remark:* You probably thought, What's the difference between this and the classic shifting using MUX we saw in 8.5.1?

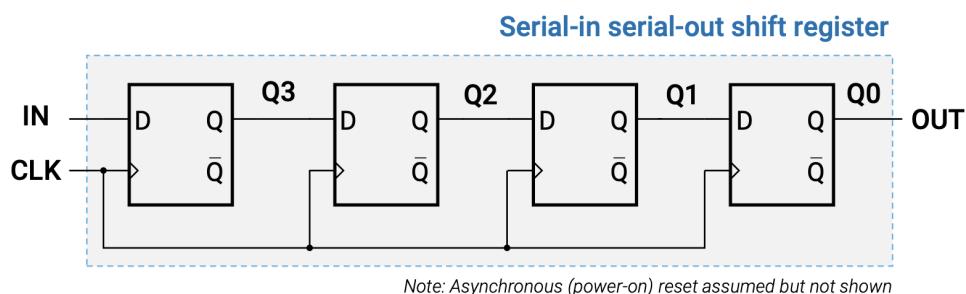
A shift register shifts data one bit per clock cycle using flip-flops, while a MUX shift can shift data by multiple positions in a single clock cycle using multiplexers. Simply put, Shift registers are more efficient for simple, sequential bit shifting operations, while MUX shifts are better for high-speed, multi-bit shifting in complex digital circuits.

Shift registers *shift* their  $n$ -bit value one bit to the left or right.

- Take 1-bit input (serial) and, sometimes,  $n$ -bit input (parallel).
- Produce 1-bit output (serial) or  $n$ -bit output (parallel).

*These are used in various applications, such as:*

*Serial-to-parallel conversion, data storage, arithmetic, frequency division.*



This visually shows the shifting at each clock cycle. (The number to be shifted is the one on the first column of the table), the one to be outputted is the one on the last column.

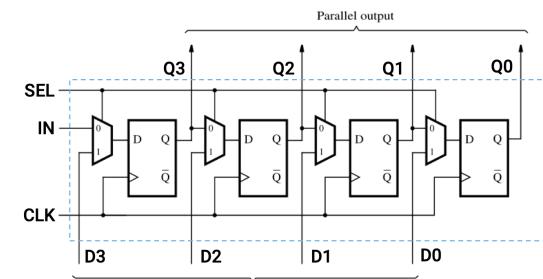
$T_{CLK}$	IN	Q3	Q2	Q1	Q0 (OUT)
t0	1	0	0	0	0
t1	0	1	0	0	0
t2	1	0	1	0	0
t3	1	1	0	1	0
t4	1	1	1	0	1
t5	0	1	1	1	0
t6	0	0	1	1	1
t7	0	0	0	1	1

### 14.2.2 Parallel Input, Parallel Output Shift Register

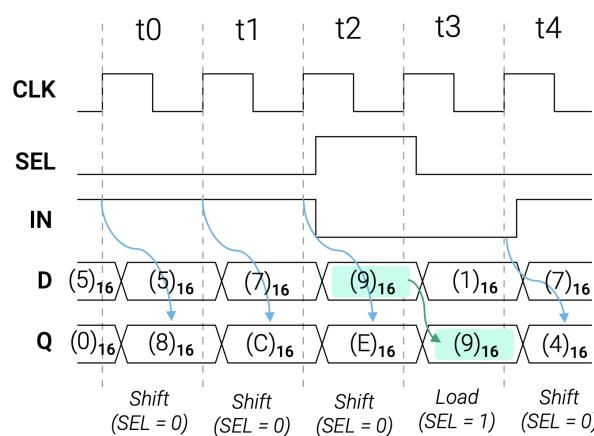
Faster than last one, allows shifting multiple bits at once

Parallel input, parallel output shift registers shift their  $n$ -bit value one bit to the left or right.

- $n$ -bit input D (parallel).
- 1 bit input IN (serial).
- $n$ -bit output (parallel).



The timing diagram for this shift register, numbers in base 16 represent the parallel input and output (each number is actually a 4-bit number input/output to D1,D2,D3,D4)



## 14.3 Shift Registers (Verilog)

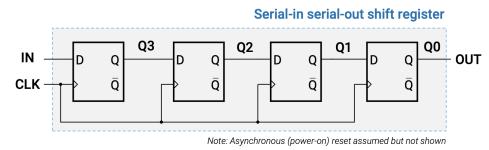
*Note: Section 14.3 covers at the end of the notes for all content on Verilog (Verilog chapter).*

### 14.3.1 Serial input/output Shift Registers

```

1 module shiftfn (IN, CLR, CLK,
2   OUT);
3   parameter n = 4; // 4-bit
4   shifter
5   input IN, CLR, CLK;
6   output OUT;
7   reg [n-1:0] Q; // internal
8   always @ (posedge CLK or
9     posedge CLR)
10 begin
11   if (CLR == 1) Q <= 0;
12   else Q <= {IN, Q[n
13     -1:1]};
14 end
15 assign OUT = Q[0];
16 endmodule

```

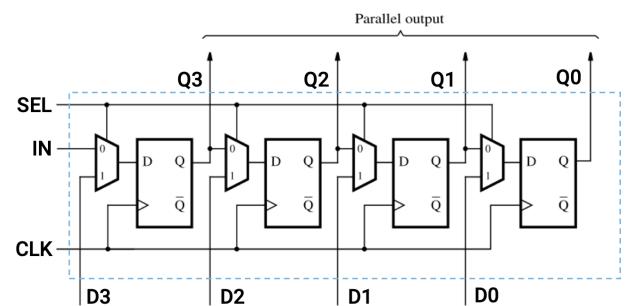


### 14.3.2 Parallel input/output Shift Registers

```

1 module shiftfn (D, IN, SEL, CLR
2   , CLK, Q);
3   parameter n = 4; // 4-bit
4   reg
5   input [n-1:0] D;
6   input IN, SEL, CLR, CLK;
7   output reg [n-1:0] Q;
8   integer k; // loop iterator
9   always @ (posedge CLK or
10    posedge CLR) begin
11    if (CLR == 1) Q <= 0;
12    else if (SEL == 1) Q <= D;
13    else begin
14      for (k = 0; k < n-1; k =
15        k+1) // for loop
16      begin
17        Q[k] <= Q[k+1];
18      end
19      Q[n-1] <= IN;
20    end
21  end
22 endmodule

```



## 14.4 Counters

Simple circuits that increment or decrement their value by 1, often used in various applications such as counting occurrences of events or as timers, can be realized as add/sub circuits, though that approach is generally considered overkill.

### Example - Counter Up with an Enable Signal

The inputs of FF are defined as:

$$\begin{aligned}D_0 &= Q_0 \oplus EN \\D_1 &= Q_1 \oplus (Q_0EN) \\D_2 &= Q_2 \oplus (Q_0Q_1EN) \\D_3 &= Q_3 \oplus (Q_0Q_1Q_2EN)\end{aligned}$$

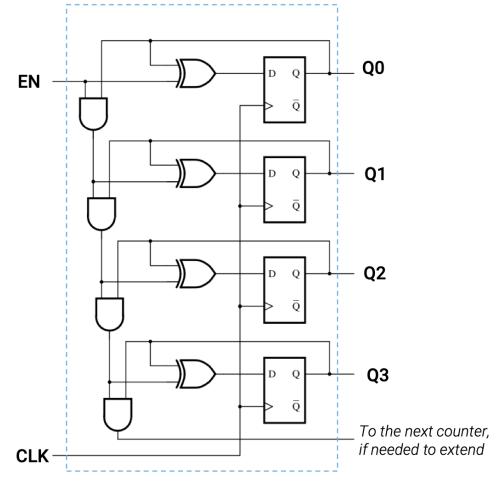
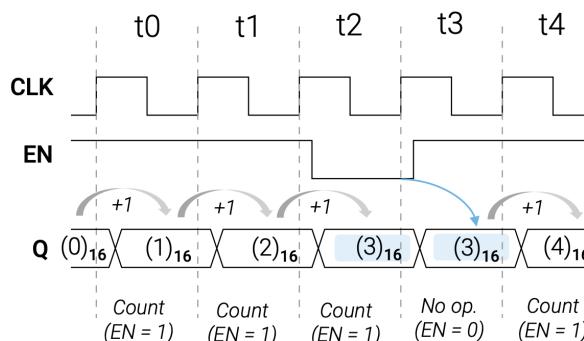
In an n-bit counter case:

$$D_i = Q_i \oplus (Q_0Q_1 \cdots Q_{i-2}Q_{i-1}EN)$$

Modulo  $2^n$  counter:

- Every  $2^n$  clock cycles, the output is zero

With the corresponding timing diagram:



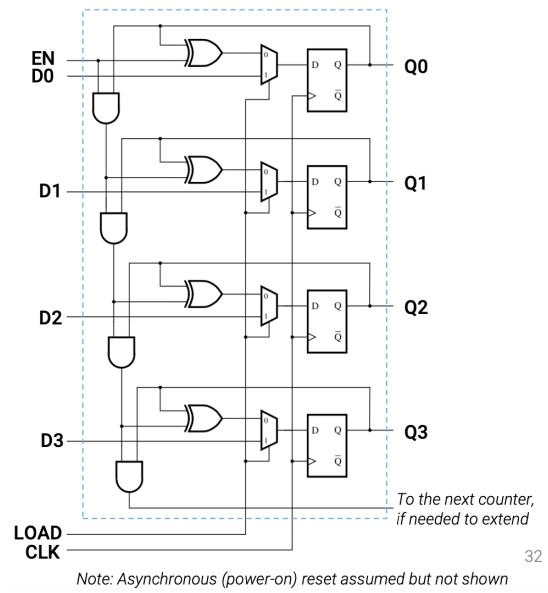
### Example - Counter with Parallel-Load Capability

It is often necessary to start counting with initial value 0, but sometimes it is desirable to start with a different value.

- Parallel-load capability allows that.

In the circuit to the right:

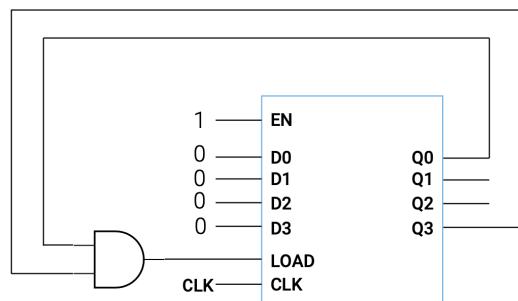
- LOAD = 0: circuit can count
- LOAD = 1: new value D is loaded



### Example - Modulo $m$ ( $m < 2^n$ ) Counter

To create a modulo  $m$  ( $m < 2^n$ ) counter (for example,  $m = 10$ ):

- Start from the counter with parallel-load capability.
- Set load active when the count reaches  $m - 1$  to restart counting from zero the next clock cycle.



## 14.5 Counters (Verilog)

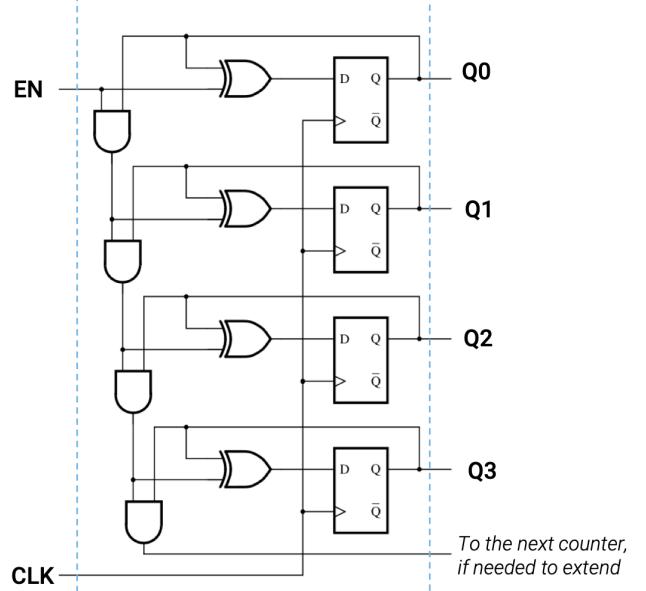
Note: See the chapter at the end of the notes for all content on Verilog (Verilog chapter). Section 14.5 covers this part in details.

### 14.5.1 Counter Up with an Enable Signal

```

1 module upcount (EN, CLR, CLK,
2   Q);
3 parameter n = 4; // 4-bit
4   counter
5   input EN, CLR, CLK;
6   output reg [n-1:0] Q;
7   always @ (posedge CLK or
8     posedge CLR)
9   begin
10    if (CLR == 1) Q <= 0;
11    else if (EN == 1) Q <= Q +
12      1;
13  end
14 endmodule

```



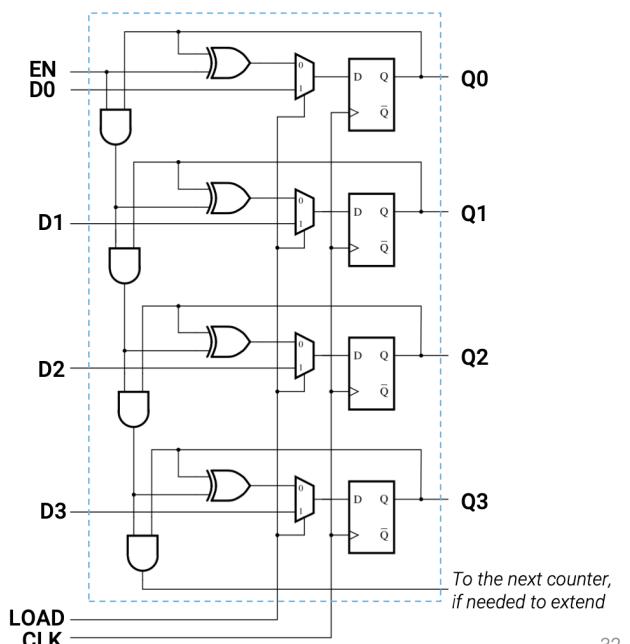
Note: Asynchronous (power-on) reset assumed but not shown

### 14.5.2 Counter with Parallel-Load Capability

```

1 module upcount (D, EN, LOAD,
2   CLR, CLK, Q);
3 parameter n = 4; // 4-bit
4   counter
5   input [n-1:0] D;
6   input EN, LOAD, CLR, CLK;
7   output reg [n-1:0] Q;
8   always @ (posedge CLK or
9     posedge CLR)
10  begin
11    if (CLR == 1) Q <= 0;
12    else if (LOAD == 1) Q <= D
13      ;
14    else if (EN == 1) Q <= Q +
15      1;
16  end
17 endmodule

```



Note: Asynchronous (power-on) reset assumed but not shown

## 14.6 Verilog Part 4.

*Note: See the chapter at the end of the notes for all content on Verilog (Verilog chapter).*

### 14.6.1 For Loop

The **initial index** is evaluated once, before the first loop iteration (e.g.,  $k = 0$ )

In each loop iteration, the **begin-end** block is performed, and then the **increment** statement is evaluated (eg.  $k = k + 1$ )

Finally, the **terminal index** condition is checked

- If true, another loop iteration is performed, otherwise the loop terminates
- Contrary to testing, for synthesis, the **terminal index** condition has to compare the loop index to a **constant value** (e.g.,  $k < 8$ )

```

1 for (initial index; terminal
      index; increment)
2 begin
3   statements;
4 end

```

### 14.6.2 Logical Operators

Logical operators in Verilog are used in conditional expressions, treating operands as true or false to produce an output of 1 (true) or 0 (false). *Ordered in terms of Precedence:*

Operator	Description
!	logical not
==	logical equality
!=	logical inequality
&&	logical and
	logical or

### 14.6.3 Relational Operators

Operator	Description
$a < b$	a less than b
$a > b$	a greater than b
$a \leq b$	a less than or equal to b
$a \geq b$	a greater than or equal to b

These operators have the same precedence level.

If operands are of unequal bit lengths, the smaller operand shall be zero-extended (if unsigned) or sign-extended (if signed) to the size of the larger one. Equal precedence.

#### 14.6.4 Equality Operators in Verilog

Equality operators compare operands bit for bit. The result is 0 if the comparison fails and 1 otherwise.

Operator	Description	Details
a == b	equality	a is equal to b, the result can be 0, 1, or unknown (x or z)
a != b	inequality	a is not equal to b, the result can be 0, 1, or unknown (x or z)
a === b	case equality	a is equal to b, including unknown(x) or high-impedance(z) states
a !=== b	case inequality	a is not equal to b, including unknown(x) and high-impedance(z) states

These equality operators have the same precedence level, which is lower than that of relational operators.

# Chapter 15

## Digital logic and Verilog, Part X

### 15.1 Flip-flop Timing Constraints and Parameters

#### 15.1.1 Synchronous System Design

In practical circuits, the outputs of sequential logic elements (registers) connect to the inputs of other sequential logic elements, typically with some combinational logic between FFs, introducing delays, and for correct circuit operation, timing constraints must be met.

#### D Flip-Flop Input Timing Constraints

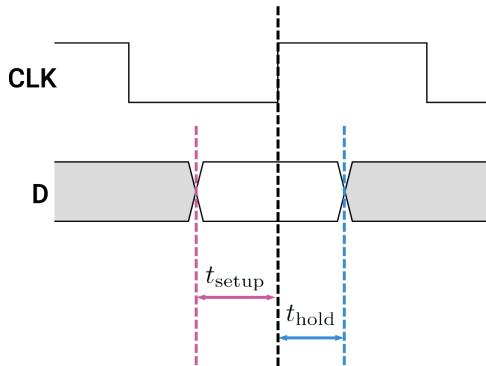
The signal on the input  $D$  of a D Flip-Flop (DFF) must be stable around the **active clock edge** (the edge that captures the signal).

The input  $D$  must remain stable during:

- **Setup time**: the period *before* the active clock edge.
- **Hold time**: the period *after* the active clock edge.

If the input timing constraints are not satisfied, the DFF will not operate correctly.

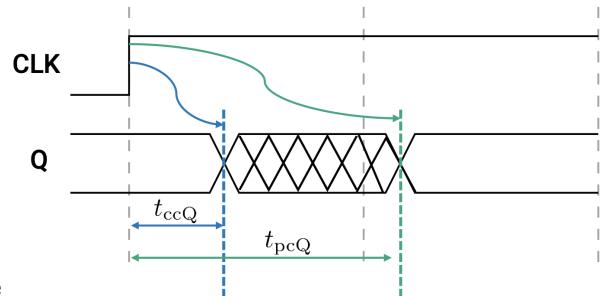
Violating these timing constraints leads to a condition called **metastability**.



## D Flip-Flop Output Timing Parameters

*Remember, Q, the output, CLK, the clock sync Timing Intervals:*

- $t_{ccQ}$  (earliest change): Shown as the blue dashed line on the diagram, indicating the earliest time  $Q$  starts to change after the clock edge.
- $t_{pcQ}$  (latest change): Shown as the green dashed line, indicating the latest time  $Q$  finishes changing after the clock edge.
- The region between these two dashed lines represents the uncertainty interval where the output  $Q$  is in transition, and this time is collectively referred to as  $t_{cQ}$ .



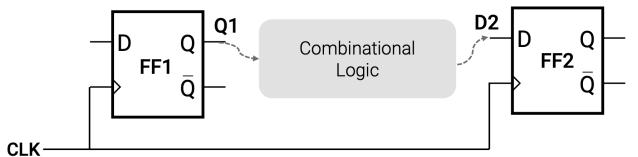
### 15.1.2 Meeting FF Timing Constraints

Sequential logic elements' outputs connect to inputs of other sequential logic elements. *For correct operation, all FFoutput-to-FFinput paths must meet timing constraints.*

The value at input D2 must be stable:

- At least  $t_{\text{setup}}$  before the active clock edge.
- At least  $t_{\text{hold}}$  after the active clock edge.

Longest and shortest delay paths are most important.



### Setup-Time Constraints

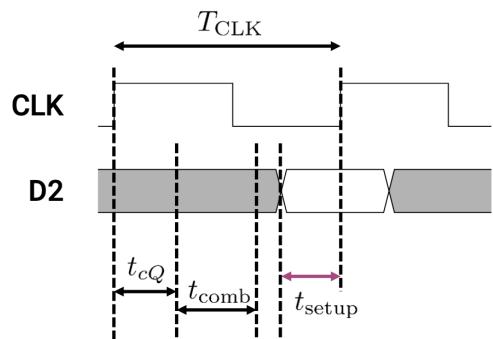
To ensure correct operation, setup-time constraints must be met. This involves the longest possible delay paths:

The total delay should be less than the clock period.

$$\text{Thus, } t_{\text{setup}} + t_{cQ,\max} + t_{\text{comb},\max} \leq T_{\text{CLK}}$$

$t_{cQ,\max}$ : Longest clock-to-Q delay

$t_{\text{comb},\max}$ : Longest path delay through combinational logic



## Hold-Time Constraints

Hold-time constraints ensure stability immediately after the clock edge:

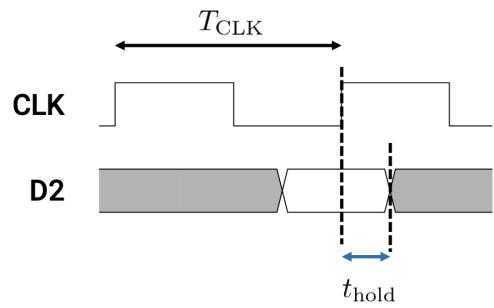
The total delay should be more than the hold time.

Formula:  $t_{cQ,\min} + t_{\text{comb},\min} \geq t_{\text{hold}}$

$t_{cQ,\min}$ : Shortest clock-to-Q delay

$t_{\text{comb},\min}$ : Shortest path delay through combinational logic

Typically,  $t_{\text{hold}} \approx 0$  in real circuits.



# Chapter 16

## All of Verilog

### Concatenation of all Verilog Content

You'll find links to this chapter at the beginning of sections about Verilog.

*Note: This chapter is generated using a Mojo script, you may find some spacing or formatting issues at times, I'm actively working on a fix for this.*

## 16.1 Verilog HDL

*Personal Remark.* this course will absolutely NOT suffice to make you a Verilog expert, please take some time to practice in an empty project, just representing in both structural and behavioral ways the circuits we've seen so far.

### 16.1.1 Structural Modeling with Logic Gates

In structural modeling, we use predefined modules (*built-in representations of basic logic gates*) to construct complex circuits.

We use this logic gate instantiation statement to create a basic logic gate:

```
1 gate_name [instance_name] (out_port, in_port{, in_port});
```

[] indicates an optional parameter.

() indicates a required parameter.

{ } indicates that additional parameters can be added.

With *gate\_name* as the type of gate :

(*not limited to these...*)

and	nor
nand	buf
xor	not
or	xnor

#### Examples

##### - AND Gate:

```
1 and and1 (out, in1, in2);
```

##### - OR Gate:

```
1 or or1 (out, in1, in2);
```

##### - NOT Gate:

```
1 not not1 (out, in);
```

##### - NAND Gate:

```
1 nand nand1 (out, in1, in2);
```

### 16.1.2 Verilog Syntax

#### In a Nutshell

##### Naming Rules:

Begin with a letter.

Include letters, digits, underscore (\_), and dollar sign (\$).

**Case Sensitivity:**

Lowercase and uppercase are distinct (e.g., `a` ≠ `A`).

**Style Guidelines:**

Syntax is flexible with white spaces and line breaks.

Prioritize readability with proper indentation.

**Comments:**

Initiated by double slashes (//).

### 16.1.3 Modules in Verilog

A circuit or subcircuit described in Verilog is encapsulated within a module. The module declaration includes the module name and its ports, which are the input and output connections to the module.

```

1 module module_name [ (port_name{, port_name}) ];
2 [input declarations]
3 [output declarations]
4 [wire declarations]
5 [logic gate instantiations]
6 [module instantiations]
7 endmodule

```

(words in purple are reserved keywords, words after two slashes are comments)

### 16.1.4 Ports in Verilog

Ports are the input and output connections of a module. They are declared within the module declaration. They can be in the following directions : **Port Types:**

- input – for receiving signals.
- output – for sending signals.
- inout – for bi-directional signal flow.

**Port Declaration:**

- Syntax:

```

1 port\_direction data\_type [port\_size] port\_name;

```

- Implicit type: Unspecified types default to wire.
- Vectors: Specify bit-width for multi-bit signals (e.g., [3:0] for 4 bits).

**Examples:**

```

input diff;           // Single-bit input named diff
inout [15:0] data;   // 16-bit bidirectional port named data
output [3:0] f;      // 4-bit output named f

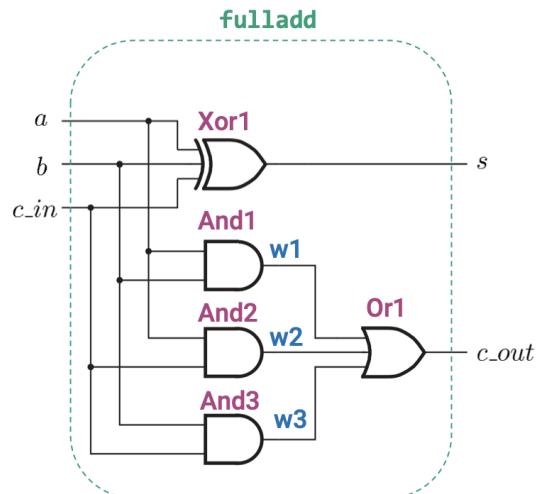
```

### 16.1.5 Example - Full adder in Verilog

```

1 // Structural modeling of a full-
2 // adder
3 module fulladd (a, b, c_in, s, c_out)
4   ;
5   // ----- port definitions -----
6   input a, b, c_in;
7   output s, c_out;
8   // ----- intermediate signals -----
9   wire w1, w2, w3;
10  // ----- design implementation -----
11  and And1 (w1, a, b);
12  and And2 (w2, a, c_in);
13  and And3 (w3, b, c_in);
14  or Or1 (c_out, w1, w2, w3);
15  xor Xor1 (s, a, b, c_in);
16 endmodule

```



Which can be simplified to :

```

1 module fulladd (a, b, c_in, s,
2   c_out);
3   input a, b, c_in;
4   output s, c_out;
5   and (w1, a, b);
6   and (w2, a, c_in);
7   and (w3, b, c_in);
8   or (c_out, w1, w2, w3);
9   xor (s, a, b, c_in);
10 endmodule

```

### 16.1.6 Subcircuits in Verilog

A Verilog module can be included as a subcircuit in another module.

Modules should be defined in the same source file, or the Verilog compiler must know the locations of the modules.

Module instantiation syntax:

```

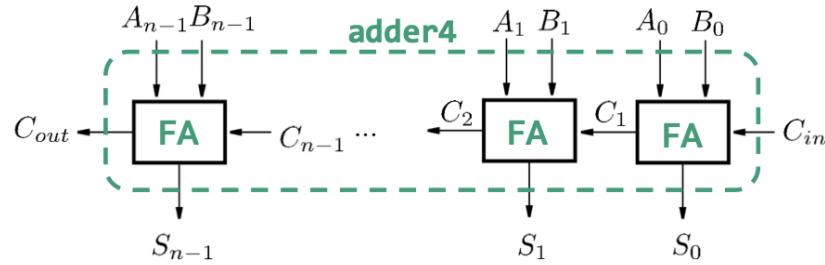
1 {module\_name instance\_name ( .port\_name (expression));}

```

- Notes:**
- \* module\_name and instance\_name are any valid identifiers.
  - \* .port\_name specifies the subcircuit's port to be connected.
  - \* Omitting .port\_name is possible if port order is identical to the subcircuit's definition, but not recommended due to error-proneness.

### 16.1.7 Example - Ripple-Carry Adder in Verilog

A four-bit ripple-carry adder is composed of four full-adder stages with interconnections for the carry bits.



Written in VHDL :

```

1 module adder4 (Cin, A, B, S, Cout);
2 // Port definitions
3 input Cin;
4 input [3:0] A, B; // 4-bit vectors
5 output [3:0] S; // 4-bit vector
6 output Cout;
7
8 // Intermediate signals
9 wire [3:1] C; // 3-bit vector for carry bits
10
11 // Design implementation using full-adder stages
12 fulladd stage0 (.c_in(Cin), .a(A[0]), .b(B[0]), .s(S[0]), .c_out(C[1]));
13 fulladd stage1 (.c_in(C[1]), .a(A[1]), .b(B[1]), .s(S[1]), .c_out(C[2]));
14 fulladd stage2 (.c_in(C[2]), .a(A[2]), .b(B[2]), .s(S[2]), .c_out(C[3]));
15 fulladd stage3 (.c_in(C[3]), .a(A[3]), .b(B[3]), .s(S[3]), .c_out(Cout));
16 endmodule

```

## 16.2 Complete Verilog Built-In Gate List

Now we can complete the Verilog built-in gate list presented in the last chapter as follows :

and	nor	bufif0(f, w, e)
nand	buf	bufif1(f, w, e)
xor	not	notif0(f, w, e)
or	xnor	notif1(f, w, e)

## 16.3 Verilog Part 2.

### 16.3.1 Scalar Signal Values

Verilog supports one-bit signals (scalars), each signal can have one of the following values:

Value	Meaning
0	Logic 0
1	Logic 1
x	Unknown logic value or don't care
z	tri-state, high-impedance

### 16.3.2 Vector Signal Values

The value of a vector variable looks like this

```
1 [size] ['radix]constant
```

*size* is the number of bits, *radix* is the base of the number, and *constant* is the value of the number. Supported radices include:

Radix	Meaning
d	decimal <i>default if no radix is specified</i>
b	binary
h	hexadecimal
o	octal

### 16.3.3 Parameters

Parameters are constants that can be used in the module. They are defined as follows:

```
1 parameter name = value;
```

eg.

```
1 parameter c = 8;
```

### 16.3.4 Nets

Nets are used to connect modules. There are two net types:

wires

```
1 wire [3:0] a;
```

tri-states

```
1 tri z1
```

### 16.3.5 Behavioral Modeling in Verilog

Behavioral modeling in Verilog is one of the three modeling paradigms used in Verilog to describe digital circuits at a high level. This approach allows designers to specify how circuits should behave through procedural statements, similar to software programming. *Simplified: used when too hard to do in structural modeling.*

### 16.3.6 Bit-wise Operators

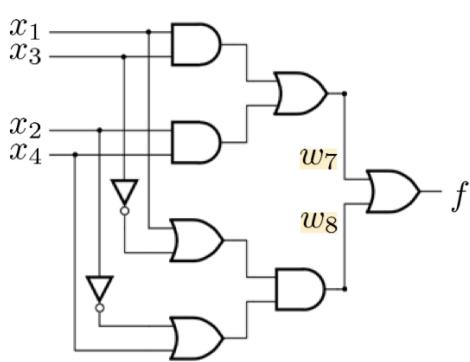
Bit-wise operators in Verilog are used to perform operations on individual bits of a vector. The following are the bit-wise operators available in Verilog:

Operator	Description
&	Bit-wise AND
—	Bit-wise OR
^	Bit-wise XOR
~	Bit-wise NOT

### 16.3.7 Example - Structural to Behavioral Modeling

Here's a circuit we've modeled in structural modeling :

#### Structural



```

1 module my_circuit_structural (
2     input x1, x2, x3, x4,
3     output f
4 );
5     wire w1, w2, w3, w4, w5, w6, w7,
6         w8;
7     and (w1, x1, x3);
8     not (w2, x2);
9     not (w3, x3);
10    and (w4, x2, x4);
11    or (w5, x1, w3);
12    or (w6, w2, x4);
13    or (w7, w1, w4);
14    and (w8, w5, w6);
15    or (f, w7, w8);
endmodule

```

#### Behavioral

```

1 module my_circuit_behavioral (
2     input x1, x2, x3, x4,
3     output f
4 );
5     wire w7, w8;
6     assign w7 = (x1 & x3) | (x2 & x4);
7     assign w8 = (x1 | ~x3) & (~x2 | x4);
8     assign f = w7 | w8;
9 endmodule

```

### 16.3.8 Continuous Assignments in Verilog

**Keyword:** `assign`

#### Definition

Continuous assignments are used in Verilog for assigning values to wires. The `assign` statement makes the left-hand side (LHS) of the assignment dynamically reflect any changes in the right-hand side (RHS) immediately.

#### Properties

- Executes continuously: The assignment reacts to any change in the RHS variables and updates the LHS accordingly.
- Operates in parallel: The execution order in the code does not affect the simulation behavior.

## Example

Consider the case where we want to perform a logical OR operation between two signals, w7 and w8, and assign the result to signal f:

```
1 assign f = w7 | w8;
```

In this example, whenever the value of w7 or w8 changes, f is automatically recalculated to reflect the new value.

### 16.3.9 Verilog Always Block

**Keyword:** always

The always block in Verilog is used for modeling combinational and sequential logic. It reacts to changes in input signals, ensuring that the block's internal logic is evaluated whenever necessary.

#### Structure

```
1 always @*
2   [begin]
3     [procedural assignment statements]
4     [if-else statements]
5     [case statements]
6     [while, repeat, and for loops]
7     [task and function calls]
8   [end]
```

In this block, the always keyword is followed by @\*.

### 16.3.10 If-Else Statements

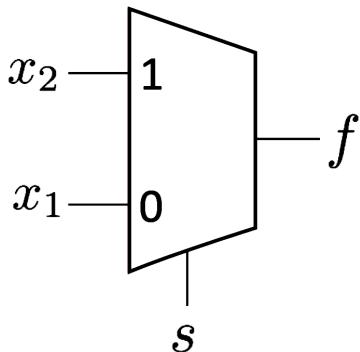
If an expression is true, the code within the begin-end block runs; multiple statements can be grouped in such a block. Optional else if and else clauses, when used, connect to the closest preceding if or else if statement.

#### Structure

```
1 if (expression1)
2 begin
3   statement;
4 end
5 else if (expression2)
6 begin
7   statement;
8 end
9 else
10 begin
11   statement;
12 end
```

### 16.3.11 Example - 2-to-1 MUX

Using what we've learned so far



```

1 module my_2to1mux ( input w1,
2   w2, s, output reg f );
3   always @*
4     begin
5       if (s == 0)
6         begin
7           f = w1;
8         end
9       else
10      begin
11        f = w2;
12      end
13    end
14 endmodule

```

### 16.3.12 Case Statement

*Personal Remark. Switch in java*

#### Syntax

```

1 case (expression)
2   alternative1:
3     begin
4       statements;
5     end
6   alternative2:
7     begin
8       statements;
9     end
10  default:
11    begin
12      statements;
13    end
14 endcase

```

### Example - Full Adder

Using the truth table of a Full-Adder, let's use a case statement to implement it:

x	y	C <sub>in</sub>	s	C <sub>out</sub>
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

```

1 module fulladd (input x, y, Cin,
2   output reg s, Cout);
3   always @*
4     begin
5       case ({x, y, Cin})
6         3'b000: {s, Cout} = 'b00;
7         3'b001: {s, Cout} = 'b10;
8         3'b010: {s, Cout} = 'b10;
9         3'b011: {s, Cout} = 'b01;
10        3'b100: {s, Cout} = 'b10;
11        3'b101: {s, Cout} = 'b01;
12        3'b110: {s, Cout} = 'b01;
13        3'b111: {s, Cout} = 'b11;
14      endcase
15    end
16 endmodule

```

## 16.4 Verilog Part 3.

### 16.4.1 Update to Always Block

Updates can take place one after the other, or concurrently, depending on the assignment used : blocking (=) or nonblocking ( $\leq$ ).

#### Blocking Assignment

For example: Here, B is assigned the value of A, then C is assigned the value of B, and finally, D is assigned the value of C. At the end,  $A = B = C = D$ . This is used to model combinational logic (without memory).

```

1 always @ (...sensitivity list...)
2 begin
3     B = A;
4     C = B;
5     D = C;
6 end

```

#### Nonblocking Assignment

For example: Changes occur at the same time, so B is assigned the value of A, C is assigned the *old* value of B, and D is assigned the *old* value of C. This is used to model sequential logic (with memory).

```

1 always @ (...sensitivity list...)
2 begin
3     B <= A;
4     C <= B;
5     D <= C;
6 end

```

#### always(\*) Blocks

(=) Block Statements should be used.

The `always @(*)` block is used to model combinational logic, where the output depends only on the current input values. It is sensitive to any change in the input signals.

```

1 // Example: AND gate
2 always @ (*)
3 begin
4     C = A & B;
5 end

```

#### always@(posedge Clock) Blocks

( $\leq$ ) Nonblocking Statements should be used.

Used to describe sequential logic containing flip-flops

*Note: Remember,  $\leq$  like the arrow in the Control Input symbol of a D Flip-Flop representation, it's a reminder that the assignment is non-blocking.*

- `always@(posedge Clock)` – always at the rising clock edge
- `always@(negedge Clock)` – always at the falling clock edge

## 16.5 Registers (Verilog)

```

1 module regn (D, CLK, CLR, Q);
2 parameter n = 16; // 16 as an example
3 input [n-1:0] D;
4 input CLR, CLK;
5 output reg [n-1:0] Q;
6 always @ (posedge CLK or posedge CLR)
7 begin
8     if (CLR == 1)
9         Q <= 0;
10    else
11        Q <= D;
12 end
13 endmodule

```

## 16.6 Shift Registers (Verilog)

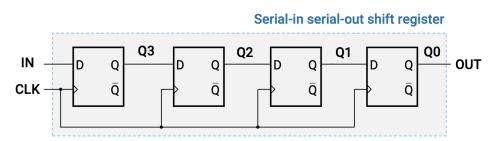
*Section 14.3* covers this part in details.

### 16.6.1 Serial input/output Shift Registers

```

1 module shiftfn (IN, CLR, CLK,
2                 OUT);
3     parameter n = 4; // 4-bit
4         shifter
5     input IN, CLR, CLK;
6     output OUT;
7     reg [n-1:0] Q; // internal
8     always @ (posedge CLK or
9             posedge CLR)
10    begin
11        if (CLR == 1) Q <= 0;
12        else Q <= {IN, Q[n
13            -1:1]};
14    end
15    assign OUT = Q[0];
16 endmodule

```

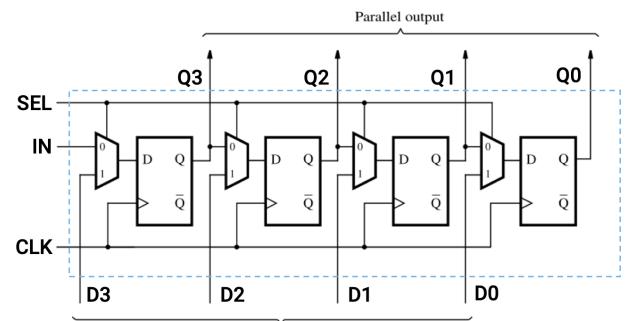


### 16.6.2 Parallel input/output Shift Registers

```

1 module shiftfn (D, IN, SEL, CLR
2   , CLK, Q);
3   parameter n = 4; // 4-bit
4   reg
5   input [n-1:0] D;
6   input IN, SEL, CLR, CLK;
7   output reg [n-1:0] Q;
8   integer k; // loop iterator
9   always @ (posedge CLK or
10    posedge CLR) begin
11      if (CLR == 1) Q <= 0;
12      else if (SEL == 1) Q <= D;
13      else begin
14          for (k = 0; k < n-1; k =
15              k+1) // for loop
16          begin
17              Q[k] <= Q[k+1];
18          end
19      end
20  end
21 endmodule

```



Note: Asynchronous (power-on) reset assumed but not shown

## 16.7 Counters (Verilog)

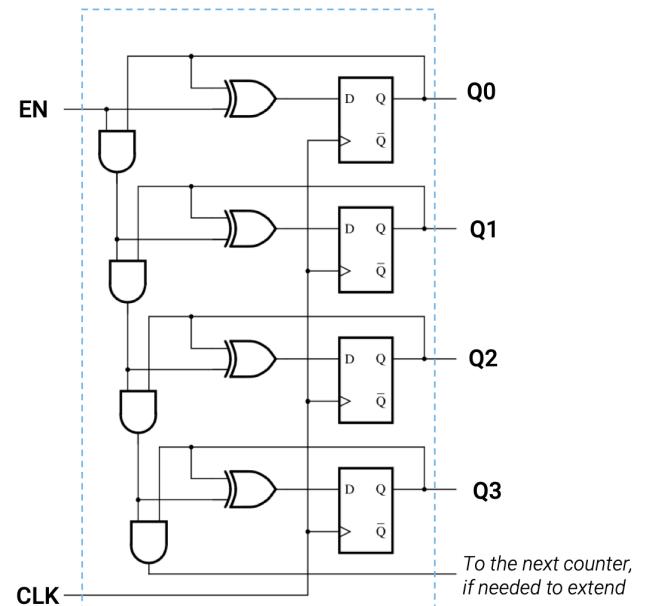
Section 14.5 covers this part in details.

### 16.7.1 Counter Up with an Enable Signal

```

1 module upcount (EN, CLR, CLK,
2   Q);
3   parameter n = 4; // 4-bit
4   counter
5   input EN, CLR, CLK;
6   output reg [n-1:0] Q;
7   always @ (posedge CLK or
8     posedge CLR)
9   begin
10      if (CLR == 1) Q <= 0;
11      else if (EN == 1) Q <= Q +
12          1;
13   end
14 endmodule

```



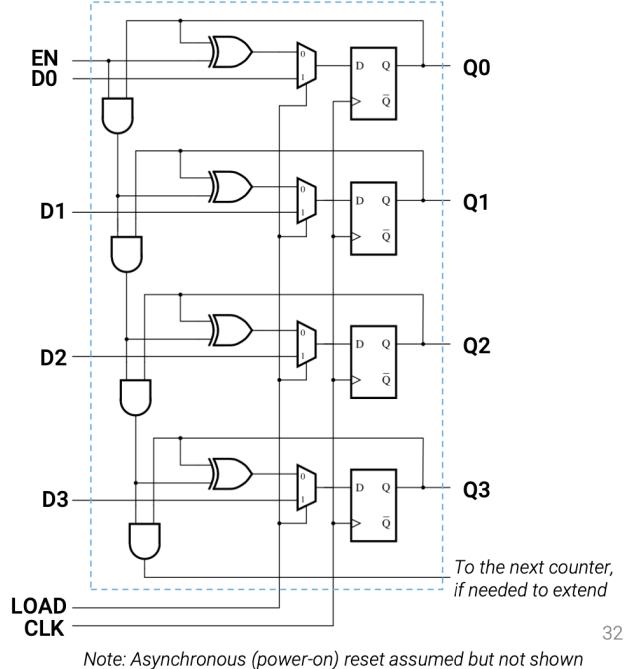
Note: Asynchronous (power-on) reset assumed but not shown

### 16.7.2 Counter with Parallel-Load Capability

```

1 module upcount (D, EN, LOAD,
2   CLR, CLK, Q);
parameter n = 4; // 4-bit
3 counter
4 input [n-1:0] D;
5 input EN, LOAD, CLR, CLK;
6 output reg [n-1:0] Q;
7 always @ (posedge CLK or
8   posedge CLR)
9 begin
10   if (CLR == 1) Q <= 0;
11   else if (LOAD == 1) Q <= D
12   ;
13   else if (EN == 1) Q <= Q +
14     1;
15 end
16 endmodule

```



32

## 16.8 Verilog Part 4.

### 16.8.1 For Loop

The **initial index** is evaluated once, before the first loop iteration (e.g.,  $k = 0$ )

In each loop iteration, the **begin-end** block is performed, and then the **increment** statement is evaluated (eg.  $k = k + 1$ )

Finally, the **terminal index** condition is checked

- If true, another loop iteration is performed, otherwise the loop terminates
- Contrary to testing, for synthesis, the **terminal index** condition has to compare the loop index to a **constant value** (e.g.,  $k < 8$ )

```

1 for (initial index; terminal
2   index; increment)
3 begin
4   statements;
5 end

```

### 16.8.2 Logical Operators

Logical operators in Verilog are used in conditional expressions, treating operands as true or false to produce an output of 1 (true) or 0 (false). *Ordered in terms of Precedence:*

Operator	Description
!	logical not
==	logical equality
!=	logical inequality
&&	logical and
	logical or

### 16.8.3 Relational Operators

Operator	Description
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

These operators have the same precedence level.

If operands are of unequal bit lengths, the smaller operand shall be zero-extended (if unsigned) or sign-extended (if signed) to the size of the larger one. Equal precedence.

### 16.8.4 Equality Operators in Verilog

Equality operators compare operands bit for bit. The result is 0 if the comparison fails and 1 otherwise.

Operator	Description	Details
a == b	equality	a is equal to b, the result can be 0, 1, or unknown (x or z)
a != b	inequality	a is not equal to b, the result can be 0, 1, or unknown (x or z)
a === b	case equality	a is equal to b, including unknown(x) or high-impedance(z) states
a !=== b	case inequality	a is not equal to b, including unknown(x) and high-impedance(z) states

These equality operators have the same precedence level, which is lower than that of relational operators.