Fundamentals of Digital Systems

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`, (W.***) corresponds to the Week number and either Monday (1) or Thursday (2) of the week.

Contents

Contents **3**

1 Number Systems (W1.1) **1**
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 Convertion 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 . . . 11
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) . . . 12
1.10.1 BCD Encoding . . . 12
1.10.2 Conversion Algorithms . . . 13

1.11 Gray Code Conversion Algorithm 13

2 Number Systems (Part II) (W1.2) 14
2.1 Addition and Subtraction of Unsigned Integers 14
2.1.1 Addition of Binary Numbers 14
2.1.2 Subtracting Two Binary Numbers 14
2.2 Overflow and Underflow in Unsigned Binary Arithmetic 15
2.2.1 Overflow 15
2.2.2 Underflow 15
2.3 Two's Complement Addition and Subtraction 16
2.3.1 Addition and Subtraction 16
2.3.2 Addition 16
2.3.3 Subtraction 16
2.4 Binary Multiplication 17

3 Number Systems (Part III) (W2.1) 18
3.1 Fixed-Point Number Representation 18
3.1.1 Examples of Fixed-Point Numbers 19
3.2 Concepts of Finite Precision Math 20
3.2.1 Precision 20
3.2.2 Resolution 20
3.2.3 Range 20
3.2.4 Accuracy 20
3.2.5 Dynamic Range 20
3.3 Floating-Point Number Representation 21
3.3.1 Significand, Base, Exponent 21
3.3.2 Benefits 21
3.3.3 Representation 22
3.3.4 Biased Representation 24
3.3.5 Rounding 25
3.4 IEEE 754 Standard 27
3.4.1 Special Values 28
3.4.2 Overflow, underflow, and others 28

4 Number Systems (Part IV) (W2.2) 30
4.1 Fixed-Point Arithmetic 30
4.1.1 Addition and Subtraction 30
4.1.2 Multiplication 30
4.2 In two's complement 31
4.3 Floating-Point Arithmetic 31
4.3.1 An Example in Binary 34

5 Number Systems (Part V) (W3.1) 35
5.1 Low precision computer arithmetic 35
5.2 Challenges and limitations 35
5.3 Block Floating Point 36

6 Digital Logic (W3.2) 37
6.1 Introduction to Digital Logic Circuits . . . 37
6.1.1 The simplest binary logic element . . . 37
6.1.2 The simplest binary logic element . . . 37
6.2 Truth Tables . . . 39
6.3 Logic Gates . . . 40
6.3.1 AND GATE . . . 40
6.3.2 AND GATE (n-variables) . . . 40
6.3.3 OR GATE . . . 40
6.3.4 OR GATE (n-variables) . . . 41
6.3.5 NOT GATE . . . 41
6.3.6 DOUBLE NOT GATE (BUFFER) . . . 41
6.3.7 NAND GATE . . . 41
6.3.8 NOR GATE . . . 41
6.3.9 Example of Complex Logic Circuit . . . 42
6.4 Analysis of a Logic Network . . . 42
6.5 The Venn Diagram . . . 43
6.6 Network Equivalence Verification . . . 44
6.7 Boolean Algebra . . . 45
6.7.1 Axioms . . . 45

7 Digital Logic (PART II) (W4.1) 46
7.1 Logic Synthesis . . . 46
7.1.1 Minterms and Maxterms . . . 46
7.1.2 Examples . . . 46
7.1.3 Sum-of-Product (SOP) Form and Product-of-Sum (POS) Form . . . 48
7.2 NAND and NOR Logic Networks . . . 49
7.2.1 NAND GATE . . . 49
7.2.2 NOR GATE . . . 49
7.3 Incompletely Defined Functions . . . 50
7.3.1 Don't Care Condition . . . 50
7.3.2 Example . . . 50
7.3.3 Incomplete Functions . . . 50
7.3.4 Sum of Products (SOP) Example . . . 50
7.4 Even and Odd Detectors (XNOR and XOR Gates) . . . 51
7.4.1 XOR Gate . . . 51
7.4.2 XNOR Gate . . . 51
7.5 Design Example . . . 51
7.5.1 Number Display . . . 51
7.5.2 Multiplexer . . . 52

8 Digital Logic (PART III) (W5.1) 53
8.1 Adders . . . 53
8.1.1 Addition of two 1-bit binary numbers . . . 53
8.1.2 Binary Addition Examples . . . 53
8.1.3 Half-Adder . . . 54
8.1.4 Addition of Two N-Bit Binary Numbers . . . 54

8.1.5 Addition of Two N-Bit Binary Numbers . . . 54
8.1.6 Full-Adder . . . 55
8.1.7 Basic Ripple-Carry Adder . . . 55
8.2 Subtractors . . . 55
8.2.1 Subtraction of Two 1-Bit Binary Numbers . . . 56
8.2.2 Binary Subtraction Examples . . . 56
8.2.3 Subtraction of Two N-Bit Unsigned Numbers . . . 56
8.2.4 Full Subtractor . . . 57
8.2.5 N-Bit Ripple-Carry Subtractor . . . 57
8.3 Adders-Subtractors in two's complement . . . 58
8.4 Fast Adders . . . 58
8.4.1 Performance Matters . . . 58
8.4.2 Examples of delays . . . 59
8.4.3 Summary of the Ripple-Carry Adder-Subtractor . . . 60
8.4.4 Carry-Select Adder . . . 61
8.5 Shifting . . . 61
8.5.1 Barrel Shifter . . . 61
8.5.2 Bidirectional Shifting by up to 7 positions . . . 63

9 Digital Logic (PART IV) (W5.2) 64
9.1 Transistors . . . 64
9.1.1 NMOS Transistor Switches . . . 65
9.1.2 PMOS Transistor Switches . . . 66
9.1.3 Example - CMOS . . . 67
9.1.4 CMOS Circuit Structure . . . 68
9.1.5 CMOS Circuits - Examples . . . 68
9.2 Real Voltage Waveforms . . . 70
9.2.1 Logic Values as Voltage Levels . . . 70
9.3 Voltage Transfer Characteristic . . . 71
9.3.1 CMOS Inverter (NOT gate) . . . 71
9.3.2 Ideal Waveforms and Real Waveforms . . . 71
9.4 Dynamic Operation . . . 73
9.4.1 Fan-In and Fan-Out . . . 73
9.4.2 Parasitic Capacitance . . . 73
9.4.3 Power Dissipation . . . 74
9.5 Hazards in Digital Circuits . . . 74

10 Digital Logic and Verilog (PART V) (W6.1) 75
10.1 CAD Design Flow . . . 75
10.2 Verilog HDL . . . 76
10.2.1 Structural Modeling with Logic Gates . . . 76
10.2.2 Verilog Syntax . . . 77
10.2.3 Modules in Verilog . . . 77
10.2.4 Ports in Verilog . . . 77
10.2.5 Example - Full adder in Verilog . . . 78
10.2.6 Subcircuits in Verilog . . . 78
10.2.7 Example - Ripple-Carry Adder in Verilog . . . 79

11 Digital Logic and Verilog(PART VI)(W6.2) 80
11.1 Bus Architecture . . . 80
11.1.1 Using Multiplexers (MUXes) . . . 80
11.1.2 Without Multiplexers . . . 81
11.2 Tri-State Drivers . . . 81
11.2.1 Types of Tri-State Drivers . . . 81
11.3 Complete Verilog Built-In Gate List . . . 82
11.4 Implementing a Bus with Tri-State Drivers . . . 83
11.5 Verilog Part 2. . . . 83
11.5.1 Scalar Signal Values . . . 83
11.5.2 Vector Signal Values . . . 83
11.5.3 Parameters . . . 83
11.5.4 Nets . . . 84
11.5.5 Behavioral Modeling in Verilog . . . 84
11.5.6 Bit-wise Operators . . . 84
11.5.7 Example - Structural to Behavioral Modeling . . . 84
11.5.8 Continuous Assignments in Verilog . . . 85
11.5.9 Verilog Always Block . . . 86
11.5.10 If-Else Statements . . . 86
11.5.11 Example - 2-to-1 MUX . . . 87
11.5.12 Case Statement . . . 87

12 Digital Logic and Verilog(PART VII) (W7.2) 88
12.1 Memory Elements . . . 88
12.1.1 Introduction - Example Application: Alarm System Control . . . 88
12.1.2 Combinational vs. Sequential Circuits . . . 89
12.1.3 Basic Memory Element . . . 89
12.2 Latches with a Control Signal . . . 91
12.2.1 Gated D Latch . . . 91
12.2.2 Flip-Flops . . . 91
12.3 Clock . . . 91
12.3.1 Clock Signal . . . 91
12.3.2 Waveform . . . 92
12.4 Verilog Part 3. . . . 92
12.4.1 Update to Always Block . . . 92
12.5 Behavioral Latch and Flip-Flop Models . . . 93
12.6 Practical notes . . . 96

13 Digital Logic and Verilog(PART VIII) (W8.1) 97
13.1 Finite State Machines . . . 97
13.1.1 Mealy State Machine . . . 98
13.1.2 Moore State Machine . . . 99
13.2 State Machine Analysis . . . 100
13.3 State Machine Synthesis . . . 104

14 Digital logic and Verilog, Part IX (W8.2) 107
14.1 Sequential Circuits . . . 107
14.1.1 Registers . . . 107
14.1.2 Registers (Verilog) . . . 108
14.2 Shift Registers . . . 108
14.2.1 Serial Input, Serial Output Shift Register . . . 108
14.2.2 Parallel Input, Parallel Output Shift Register . . . 109
14.3 Shift Registers (Verilog) . . . 110
14.3.1 Serial input/output Shift Registers . . . 110
14.3.2 Parallel input/output Shift Registers . . . 110
14.4 Counters . . . 111
14.5 Counters (Verilog) . . . 113
14.5.1 Counter Up with an Enable Signal . . . 113
14.5.2 Counter with Parallel-Load Capability . . . 113
14.6 Verilog Part 4. . . . 114
14.6.1 For Loop . . . 114
14.6.2 Logical Operators . . . 114
14.6.3 Relational Operators . . . 114
14.6.4 Equality Operators in Verilog . . . 114

15 Digital logic and Verilog (Part X) (W9.1-W10.1) 116
15.1 Flip-flop Timing Constraints and Parameters . . . 116
15.1.1 Synchronous System Design . . . 116
15.1.2 Meeting FF Timing Constraints . . . 117
15.1.3 Timing Analysis of a Simple Circuit . . . 118
15.1.4 Timing Analysis of a Counter . . . 119
15.2 Metastability . . . 120
15.3 Synchronizer (Shift Register) . . . 120
15.4 Life of D: Lasting One Clock Cycle . . . 121
15.4.1 Born as Q of the Source Flip-Flop (FF) . . . 122
15.4.2 Journey through Combinational Logic . . . 122
15.4.3 Arrives at the Destination FF . . . 122
15.4.4 Waits to be Captured on Clock Edge . . . 123
15.4.5 Gets Captured on the Clock Edge . . . 123
15.4.6 Waits to be Replaced . . . 123
15.5 Clock Skew . . . 124
15.5.1 Clock Delays . . . 124
15.6 Life of D: Lasting One Clock Cycle (with Clock Skew) . . . 125
15.7 Timing Constraints in the Presence of Clock Skew . . . 126
15.7.1 Setup-Time Constraint . . . 126
15.7.2 Hold-Time Constraint . . . 126
15.7.3 Implications of Clock Skew on the Max Operating Frequency . . . 127
15.7.4 Timing Analysis with a Clock Skew . . . 128
15.7.5 Example Analysis with Clock Skew . . . 129

16 Digital logic and Verilog, Part XI (W10.2) 132
16.1 $n - to - 2^n$ Binary Decoders . . . 132
16.2 Memory . . . 134
16.2.1 Abstract View of Memory . . . 135
16.2.2 Word Sizes . . . 135
16.2.3 Memory Capacity . . . 135
16.2.4 Access Protocol . . . 136
16.2.5 Example - 16 x 8 Memory . . . 136
16.2.6 Memory as an Array of DFFs . . . 137
16.3 Verilog Part 5. . . . 138
16.3.1 Arrays . . . 138
16.3.2 Memory Module . . . 138
16.3.3 Parameterized Verilog Modules . . . 138
16.3.4 Verilog Conditional Operator . . . 139

17 Digital Logic and Verilog - Examples - FSM (W10.2) 140
17.1 Moore FSM, with Verilog Implementation . . . 140
17.2 Mealy FSM, with Verilog Implementation . . . 141

18 Digital Logic and Verilog (Part XII) (W11.1) 143
18.1 Swapping of two Registers . . . 143
18.1.1 Using a Bus with Tri-State Drivers (Verilog) . . . 143
18.1.2 Using a Bus with MUXes (Verilog) . . . 145
18.2 Alternative ways to impelment Adders . . . 147
18.2.1 Ripple Carry Adder Using a For Loop (Verilog) . . . 148
18.2.2 Generate Construct (Verilog) . . . 148
18.2.3 Ripple-Carry Adder with a generate Construct (Verilog) . . . 149
18.3 Serial Adders . . . 149
18.3.1 Circuit . . . 149
18.3.2 Serial Adder Mealy FSM . . . 149
18.3.3 Serial Adder Mealy FSM (Verilog) . . . 150
18.4 From Verilg to Circuits . . . 151
18.5 Verilog Reduction Operators . . . 155

19 Computer Architecture (Part I) (W12.1) 157
19.1 A Processor . . . 157
19.2 From Programs to Computers . . . 157
19.3 Hardware-Friendly Programs . . . 158
19.3.1 Translating High-Level Code into Binary . . . 158
19.3.2 From High-Level Programs to Assembly . . . 158
19.4 Under the Hood . . . 160
19.4.1 Data Path . . . 160
19.4.2 Control Path . . . 161
19.5 Data Path + Control Path . . . 163
19.5.1 Data Memory . . . 163
19.6 Complete Processor . . . 163
19.7 Types of Processors . . . 164

19.8 Instruction Set Architecture 164
19.9 RISC V 164

20 Computer Architecture (Part II) (W12.2) 165
20.1 RV32I 165
20.1.1 Registers 165
20.1.2 Computational Instructions 165
20.1.3 Integer Register-Immediate Operations 167
20.1.4 NOP Pseudoinstruction 170

21 Computer Architecture (Part III) (W13.1) 171
21.1 Memory 171
21.1.1 Alignment of Instructions 172
21.1.2 Memory Word and Byte Ordering 172
21.1.3 Memory Read and Write Operations 173

Chapter 1

Number Systems (W1.1)

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)

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

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

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

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

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:

$$W = \left(r^{n-1}, r^{n-2}, \ldots, r^2, r, 1\right)$$

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:

$$W = \left(10^{n-1}, 10^{n-2}, \ldots, 10^2, 10, 1\right)$$

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

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

Canonical digit sets with fixed radix:

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

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 Convertion 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: $\underline{1011}$

Decimal: $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11$

Decimal to Binary:

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

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

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:	$\underline{257}$
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.

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

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:	$\underline{1A3}$
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 convertion).
- 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:	$\underline{010}\,\underline{000}\,\underline{100}\,\underline{110}$
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:	$\underline{1011}\,\underline{1110}\,\underline{1010}\,\underline{1101}$
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:	257
Binary:	010 101 111 (Octal to binary)
Binary grouped:	0101 0111
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 a the most significant bit (MSB) of the digit vector:

$0 \rightarrow$ positive

$1 \rightarrow$ 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:

$$01010101_2 = +85_{10}$$
$$01111111_2 = +127_{10}$$
$$00000000_2 = +0_{10}$$
$$11010101_2 = -85_{10}$$
$$11111111_2 = -127_{10}$$
$$10000000_2 = -0_{10}$$

Note: The Sign-and-Magnitude representation is considered a redundant system because both 00000000_2 and 10000000_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)*:

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

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*.

$$x_R \equiv x \mod C$$

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-Compelement 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 $b_{n-1}b_{n-2}\ldots b_1b_0$ 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:

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

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:

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

1.8 Range Extension Algorithm in Sign-and-Magnitude

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

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

Example: Consider $X = 11010101_2 = -85_{10}$, is equivalent to $Z = 10001010101 = -85_{10}$ in an 8-bit system.

The algorithm extends the range of X by adding zeros to the right 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):

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

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:

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

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):

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

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):

$$Z_{n-1} = X_{n-1}$$
$$Z_{i-1} = X_i, \quad \text{for } i = 1, \ldots, n-1$$

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 $Y = 110101_2 = -11_{10}$ (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 $\underline{0010\ 0101}_2$.

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.

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) (W1.2)

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:

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10$ (0 and carry 1 to the next higher bit)

$1 + 1 + 1 = 11$ (1 and carry 1 to the next higher bit)

Example

Adding 1101_2 and 1011_2:

	1	1	1	0	(Carry)
	1	1	0	1	
+	1	0	1	1	
1	1	0	0	0	

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:

$0 - 0 = 0$

$1 - 0 = 1$

$1 - 1 = 0$

$0 - 1 = 1$ (with a borrow from the next higher bit)

Example

Subtracting 1000_2 from 1101_2:

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

Therefore, the difference between 1101_2 and 1011_2 is 0101_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:

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

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:

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

underflow

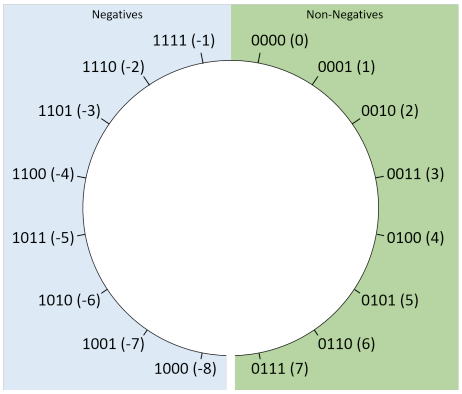
Since the result would be negative, which cannot be represented in unsigned arithmetic, this situation is considered underflow.

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

For $2 + 3$, we move clockwise from 0 to 2 and then we move 3 more steps clockwise, resulting in 5.

$5 - 3$, we move clockwise from 0 to 5 and then we move 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:

$$\text{Sum} = A + B \tag{2.1}$$

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:

$$\text{Difference} = A + \bar{B} \tag{2.2}$$

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:

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

□

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	× 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) (W2.1)

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:

$$X = (X_{m-1}X_{m-2}\dots X_1X_0 \,.\, X_{-1}X_{-2}\dots X_{-f})$$

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

$$x = (10077.01690)$$

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

Most negative (min):

$$x_{\min} = -99999.99999 = \frac{-9999999999}{10^5}$$

Largest number (max, positive):

$$x_{\max} = +99999.99999 = \frac{+9999999999}{10^5}$$

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:

$$X = (X_{m-1}X_{m-2}\dots X_1X_0 \,.\, X_{-1}X_{-2}\dots X_{-f})$$

Precision is the maximum number of non zero bits.

$$Precision(x) = m + f$$

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 difference between the smallest non zero number and the representation of zero.
For fixed point representation:

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

3.2.3 Range

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

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

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

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

3.2.4 Accuracy

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

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

3.2.5 Dynamic Range

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

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

In two's complement :

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

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 consits 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:

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

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:

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

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:

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

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 :

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

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

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)

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

- 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_{real}, we choose the even number which is 2.

Round towards zero (truncation)

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

Since x_{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_{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	00000000000000000000000
Negative zero:	1	00000000	00000000000000000000000

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	00000000000000000000000
Negative infinity:	1	11111111	00000000000000000000000

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

NaN (example):	–	11111111	10000000000000000000000

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

$$X = (1\ 01111100\ 01000000000000000000000)$$

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:

$$0.8125 \times 2 = 1.625 \to 1$$
$$0.625 \times 2 = 1.25 \to 1$$
$$0.25 \times 2 = 0.5 \to 0$$
$$0.5 \times 2 = 1.0 \to 1$$

  Stop when the fractional part becomes zero.

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

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

- Result:

$$X = (1\ 01111110\ 10100000000000000000000)$$

Chapter 4

Number Systems (Part IV) (W2.2)

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	
×	011.01	Multiplier	
	000000	First partial product (always zero), sign-extended	
+	001011	1 x multiplicand, sign-extended	
	001011	Intermediate result, sign-extended	
+	00000	0 x multiplicand, left-shifted by 1 place and sign-extended	
	00001011	Intermediate result, sign-extended	convert to fixed-point now
+	001011	1 x multiplicand, left-shifted by 2 places and sign-extended	
	00010111	Intermediate result, sign-extended	
+	001011	1 x multiplicand, left-shifted by 3 places and sign-extended	
	001001111	Intermediate result, sign-extended	
+	000000	0 x multiplicand, left-shifted by 4 places and sign-extended	
	001000.1111	Result	

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:

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

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: 1.01×2^3 and 1.1×2^2. 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 1.01×2^3 and 0.11×2^3.

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 10.00×2^3 is already in the correct format, but let's note that if our result was something like 1.000×2^4, 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 1.00×2^4 in binary, which is 16 in decimal.

Chapter 5

Number Systems (Part V) (W3.1)

This small chapter concludes the Number Systems chapter with a 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, any 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 ⇒ 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 / $1mm^2$)

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

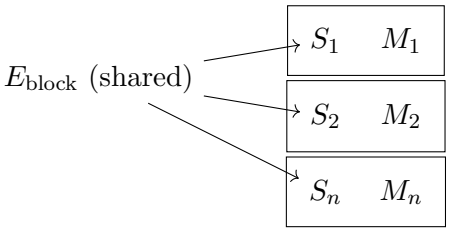
5.3 Block Floating Point

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_{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 (W3.2)

6.1 Introduction to Digital Logic Circuits

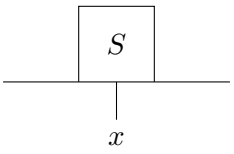
The smallest unit of Digital Information is a binary value 1 or 0.

6.1.1 The simplest binary logic element

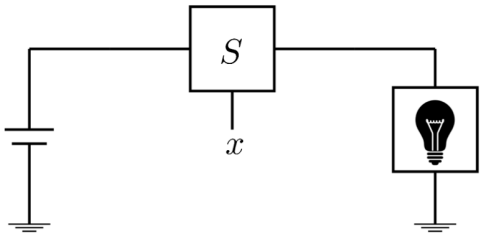
Similarly, a switch can either be open or closed. Let x an input variable.

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

The symbol for a switch controlled by an input variable:



6.1.2 The simplest binary logic element

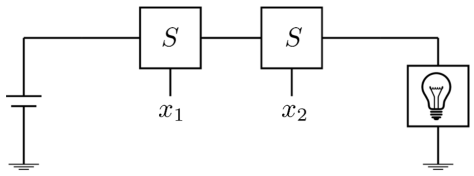


Light controlled by a single switch

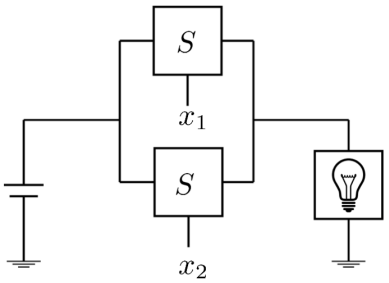
Two-Variable Logic Functions

Series and Parallel Connections

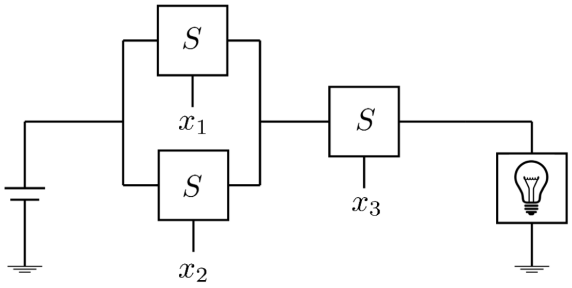
Remark. The choice of symbols is not random $\cdot, +$, respectively similar to multiplication and addition.



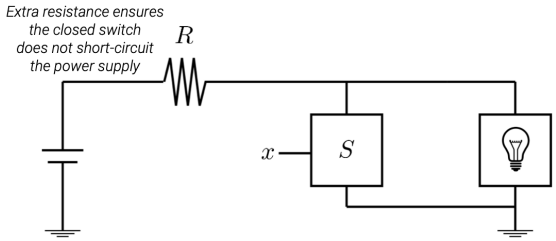
Logic AND function
$L(x_1, x_2) = x_1 \cdot x_2$, with $\cdot$ the AND operator



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

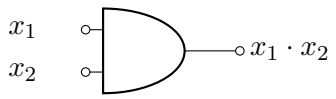
x	$\bar{x}$	NOT
0	0	0
0	1	0

Precedence Table for Logic Operations

Precedence (Priority)	Operator	Operation	Description
1	$\overline{x}$	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

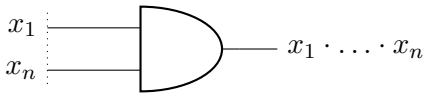
6.3 Logic Gates

6.3.1 AND GATE



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

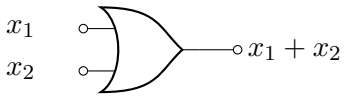
6.3.2 AND GATE (n-variables)



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

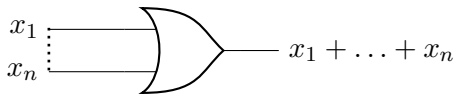
6.3.3 OR GATE

True if at least one input is true.



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

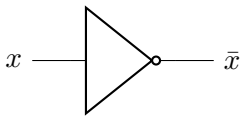
6.3.4 OR GATE (n-variables)



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

6.3.5 NOT GATE

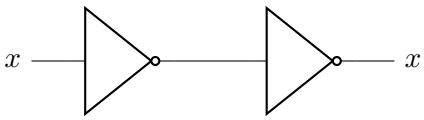
True if input is false, and vice versa.



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

6.3.6 DOUBLE NOT GATE (BUFFER)

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

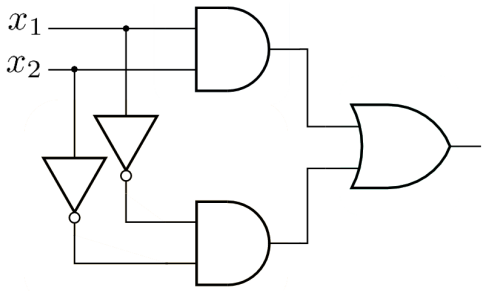


$$f(x) = x = \bar{\bar{x}}$$

6.3.7 NAND GATE

6.3.8 NOR GATE

6.3.9 Example of Complex Logic Circuit



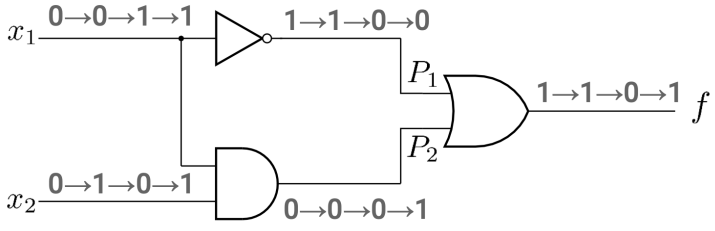
$f(x_1, x_2) = x_1 x_2 + \bar{x}_1 \bar{x}_2$

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

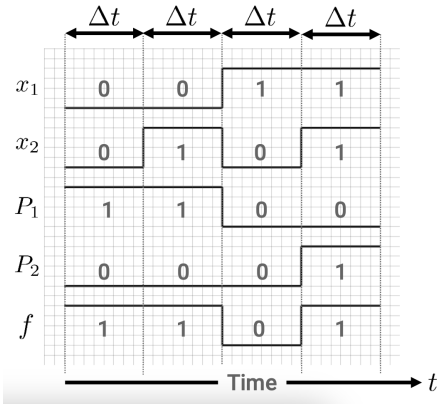
Corresponding logic table

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:

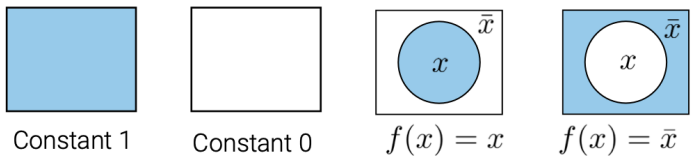
A sequence of algebraic transformations, although it's not always clear which transformations to apply.

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

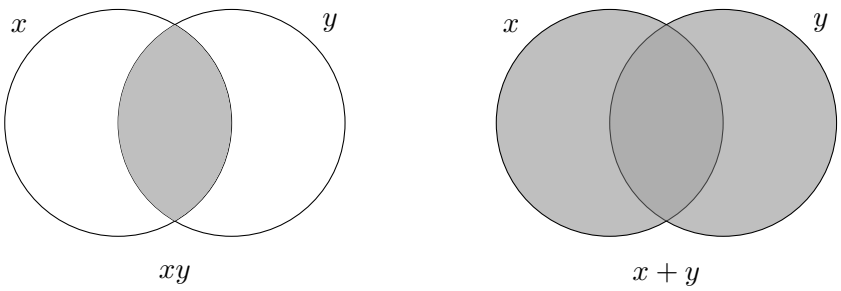
Automated techniques in synthesis tools (software) which can handle more complex networks efficiently.

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.



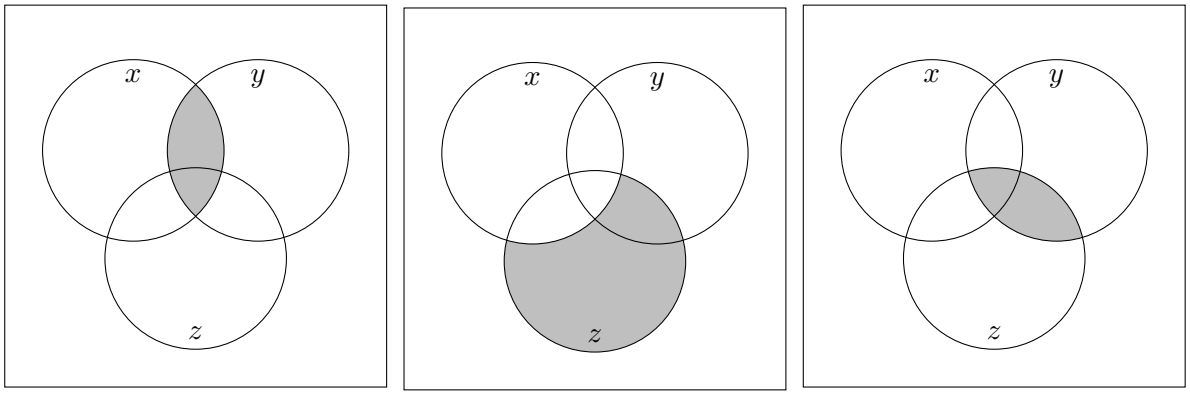
Basic Venn diagram representations



6.6 Network Equivalence Verification

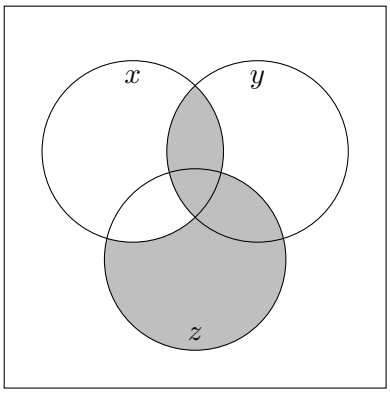
Venn Diagram Approach

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



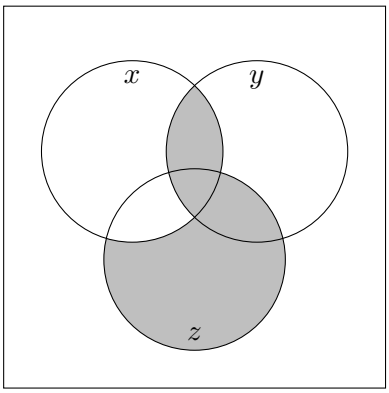
xy $\bar{x}z$ yz

Thus, the two expressions are equivalent.



$xy + \overline{x}z + yz$

$\equiv$



$xy + \bar{x}z$

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) (W4.1)

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

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

7.1.2 Examples

Minterms

To make it simple, $1 \to x,\ 0 \to \overline{x}$
Let $n = 3, i = 5$:
$5 = \underline{101}_2$ thus $m_5 = x_1\overline{x}_2x_3$
Let $n = 5, i = 3$:
$3 = \underline{00011}_2$ thus $m_3 = \overline{x}_1\overline{x}_2\overline{x_3}x_4x_5$

Maxterms

Maxterms are calculated as $M_i = \overline{m}_i$
Let $n = 3, i = 5$:
Using De Morgan's Law :

We have :

$$m_5 = x_1\overline{x}_2x_3$$

thus :

$$M_5 = \overline{m_5} = \overline{x_1\overline{x}_2x_3} = \overline{x}_1 + x_2 + \overline{x}_3$$

Let $n = 5, i = 3$:

We have :

$$m_3 = \overline{x}_1\overline{x}_2x_3\overline{x}_4\overline{x}_5$$

thus :

$$M_3 = \overline{m_3} = \overline{\overline{x}_1\overline{x}_2\overline{x_3}x_4x_5} = x_1 + x_2 + x_3 + \overline{x}_4 + \overline{x}_5$$

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x_1x_2x_3}$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x_1x_2}x_3$	$M_1 = x_1 + x_2 + \overline{x_3}$
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	$m_3 = \overline{x_1}x_2x_3$	$M_3 = x_1 + \overline{x_2} + \overline{x_3}$
4	1	0	0	$m_4 = x_1\overline{x_2x_3}$	$M_4 = \overline{x_1} + x_2 + x_3$
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	$m_6 = x_1x_2\overline{x_3}$	$M_6 = \overline{x_1} + \overline{x_2} + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x_1} + \overline{x_2} + \overline{x_3}$

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

$$\begin{aligned} f(x_1, x_2, x_3) &= \overline{x_1}\overline{x_2}x_3 + x_1\overline{x_2}\overline{x_3} + x_1\overline{x_2}x_3 + x_1x_2\overline{x_3} \\ &= (\overline{x_1} + x_1)x_2x_3 + \overline{x_1}(\overline{x_2} + x_2)x_3 \\ &= 1 \cdot x_2x_3 + \overline{x_1} \cdot 1 \cdot x_3 \\ &= x_2x_3 + \overline{x_1}x_3 \end{aligned}$$

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 :

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

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

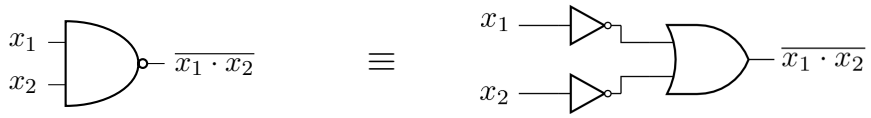
By De Morgan's theorem

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

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:

$$f = x_2 + \overline{x_1 x_3} \tag{7.1}$$

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

$$f = x_2 + \overline{x_1 x_3} \tag{7.2}$$
$$= \overline{\overline{x_2 + x_1 \overline{x_3}}} \tag{7.3}$$
$$= \overline{\overline{x_2}} \cdot \overline{x_1 \overline{x_3}} \tag{7.4}$$

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:

$$f = (x_1 + x_2)(x_2 + x_3) \tag{7.5}$$

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

$$f = (x_1 + x_2)(x_2 + \overline{x_3}) \tag{7.6}$$
$$= \overline{\overline{(x_1 + x_2)(x_2 + \overline{x_3})}} \tag{7.7}$$
$$= \overline{\overline{(x_1 + x_2)} + \overline{(x_2 + \overline{x_3})}} \tag{7.8}$$

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:

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

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

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

using 5 gates and 8 inputs.

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

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

using an OR gate and redundancy rules.

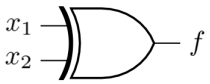
Adding don't care values sometimes allows for a simpler implementation.

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

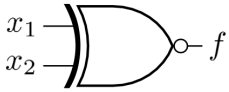


$$f = x_1 \oplus x_2$$

7.4.2 XNOR Gate

The exclusive 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

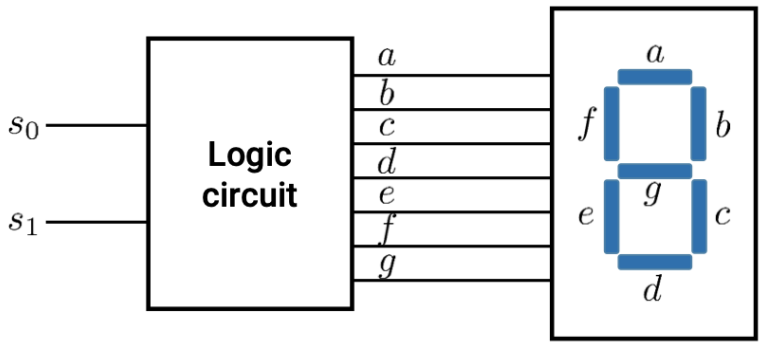


$$f = x_1 \odot x_2$$

7.5 Design Example

7.5.1 Number Display

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



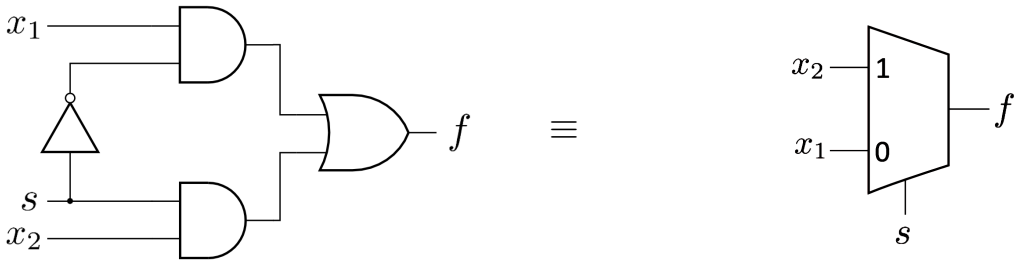
We can derive one logic function per output of the table below:

		Output						
S_1	S_0	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.



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

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

$f(s, x_1, x_2) = \overline{s}x_1x_2 + \overline{s}x_1\overline{x_2} + s\overline{x_1}x_2 + sx_1x_2$
$f(s, x_1, x_2) = \overline{s}x_1(\overline{x_2} + x_2) + s(\overline{x_1} + x_1)x_2$
$= \overline{s}x_1 + sx_2$

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) (W5.1)

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

		x
	+	y
c (carry)		s (sum)

8.1.2 Binary Addition Examples

	0
+	0
0	0

	0
+	1
0	1

	1
+	0
0	1

	1
+	1
1	0

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.

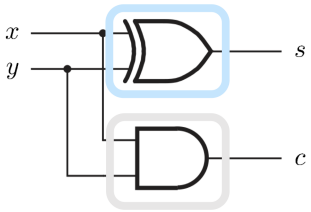
A **half adder** is a digital circuit that adds two single-bit binary numbers and outputs a sum and a carry.

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:

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

$$c = xy$$



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:

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

The solution to the equation is given by:

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

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

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.

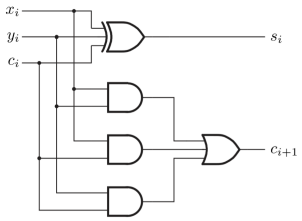
x_i	y_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

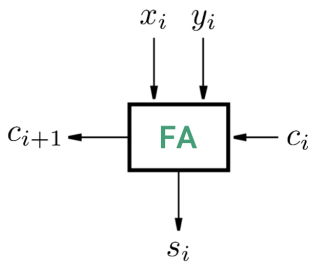
$$
\begin{aligned}
s_i &= \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i\bar{y}_i\bar{c}_i + x_i y_i c_i \\
&= (x_i y_i + \bar{x}_i\bar{y}_i)c_i + (\bar{x}_i y_i + x_i\bar{y}_i)\bar{c}_i \\
&= \overline{(x_i \oplus y_i)}c_i + (x_i \oplus y_i)\bar{c}_i = x_i \oplus y_i \oplus c_i
\end{aligned}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

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

Giving us:

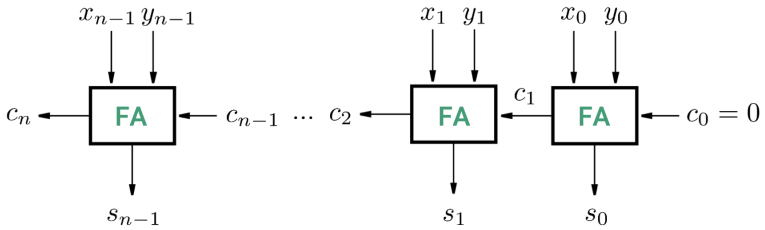
$$
\begin{aligned}
s_i &= x_i \oplus y_i \oplus c_i \\
c_{i+1} &= (x_i \oplus y_i)c_i + x_i y_i = x_i y_i + x_i c_i + y_i c_i
\end{aligned}
$$





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 substraction 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.

$$\begin{array}{rr} & \text{b} \\ & \text{x} \\ - & \text{y} \\ \hline & \text{d} \end{array}$$

8.2.2 Binary Subtraction Examples

$$\begin{array}{rr} & 0 \\ & 0 \\ - & 0 \\ \hline & 0 \end{array}$$

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

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

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

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:

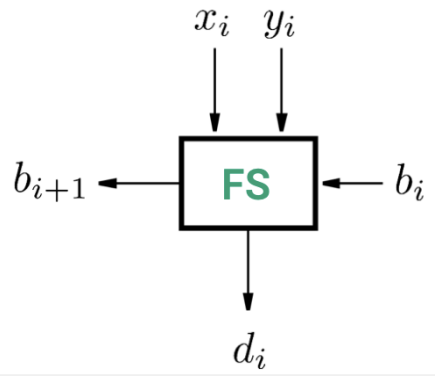
x_i	y_i	b_i	d_i	b_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Logical expressions:

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

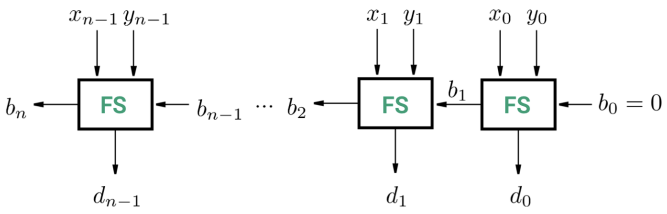
$$\begin{aligned} b_{i+1} &= \bar{x}_i\bar{y}_i b_i + \bar{x}_i y_i \bar{b_i} + \bar{x}_i y_i b_i + x_i y_i b_i \\ &= (\bar{x}_i\bar{y}_i b_i + \bar{x}_i y_i b_i) + (\bar{x}_i y_i \bar{b_i} + \bar{x}_i y_i b_i) + (\bar{x}_i y_i b_i + x_i y_i b_i) \\ &= \bar{x}_i b_i + \bar{x}_i y_i + y_i b_i \end{aligned}$$

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:

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

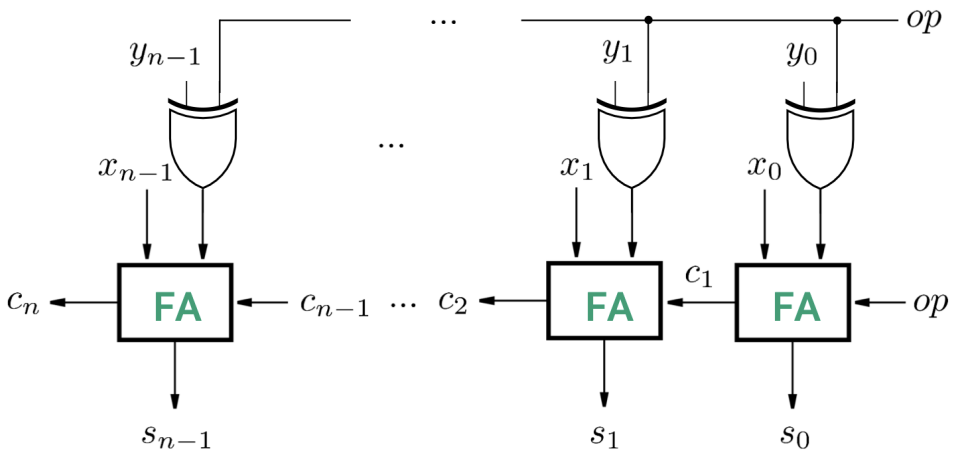
$\overline{Y}$ is obtained by complementing each of the bits of Y

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

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

$$f(X,Y) = X + \overline{op} \cdot Y + op \cdot \overline{Y} + op$$

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.

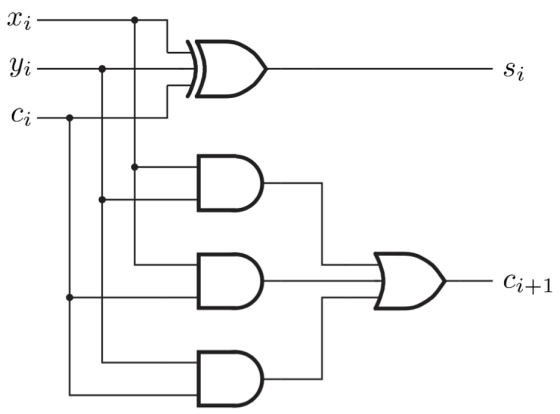
$$\text{value} = \frac{\text{performance}}{\text{price}}$$

8.4.2 Examples of delays

Full Adder

- Delay to generate the sum

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



- Delay to generate carry-out

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

- Worst-case delay

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

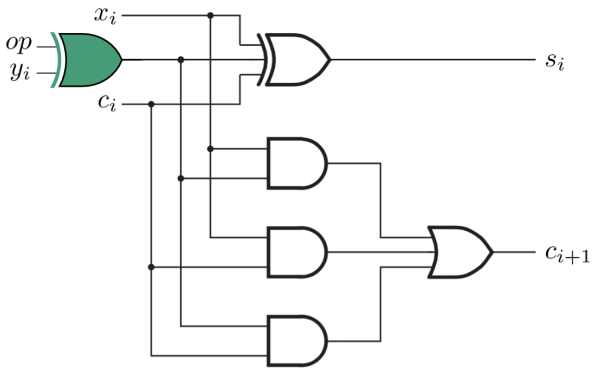
- If all gates had equal delays:

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

Full Adder-Subtractor

- Delay to generate the sum

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



- Delay to generate carry-out

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

- Worst-case delay

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

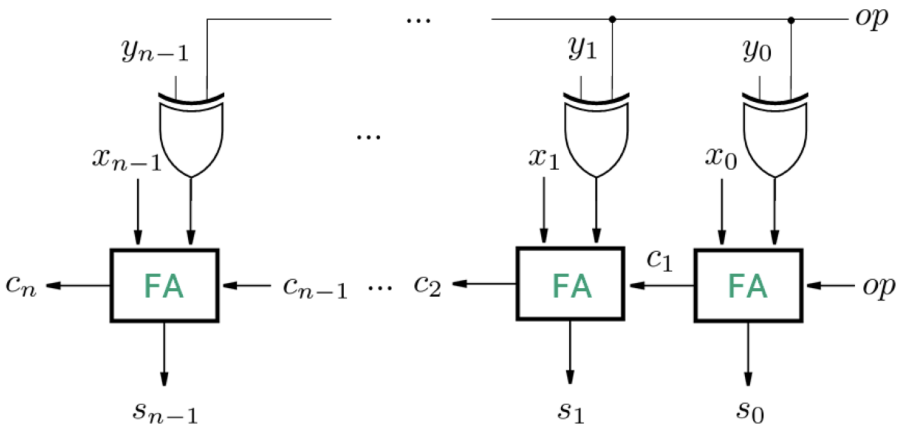
- If all gates had equal delays:

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

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 **X**, **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)

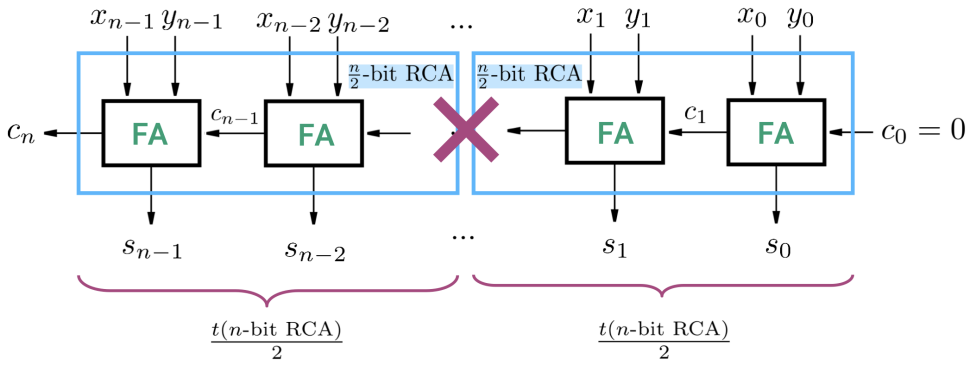


8.4.4 Carry-Select Adder

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.

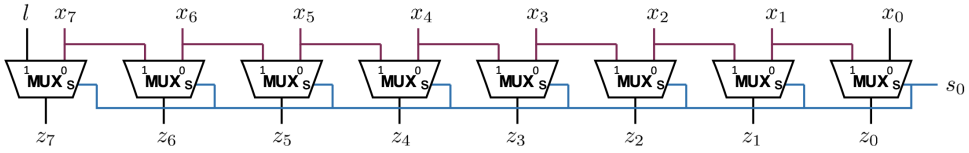
  Arithmetic shift: Keep the sign of a binary number while shifting.

  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

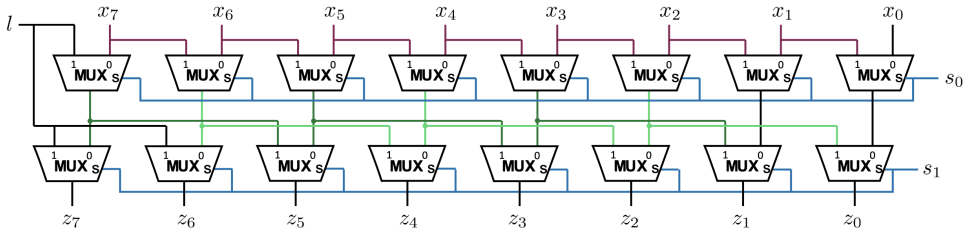
if $l = 0$, the shift is Logical (resets the leftmost bit to 0), if $l = x_{n-1}$, the shift is Arithmetic (conserve the sign of the number).



Thus, the corresponding truth table:

s_0	$z_7z_6z_5z_4z_3z_2z_1z_0$
0	$x_7x_6x_5x_4x_3x_2x_1x_0$
1	$lx_7x_6x_5x_4x_3x_2x_1$

Shift Right by up to Three positions



Thus, the corresponding table :

s_1	s_0	$z_7z_6z_5z_4z_3z_2z_1z_0$
0	0	$x_7x_6x_5x_4x_3x_2x_1x_0$
0	1	$lx_7x_6x_5x_4x_3x_2x_1$
1	0	$llx_7x_6x_5x_4x_3x_2$
1	1	$lllx_7x_6x_5x_4x_3$

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 `in0` to `in7` and outputs `out0` to `out7`.

Control signals `S0`, `S1`, `S2` for the shift magnitude, and `dir` for the direction.

A logical input `l` for leftward shifts.

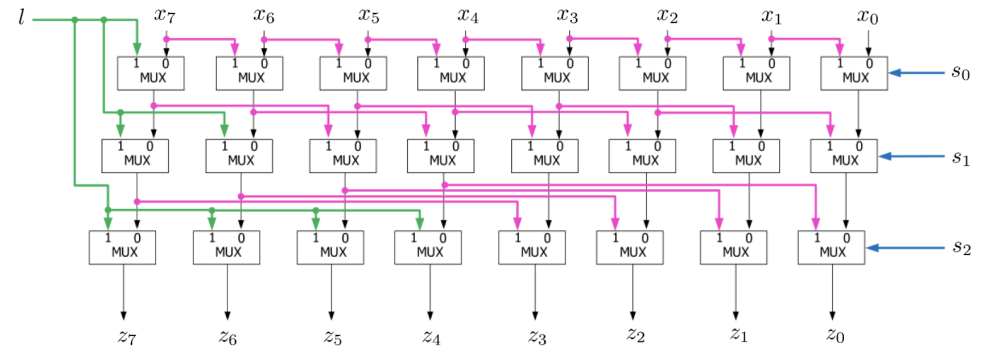
Shift operation:

Right Shift: With `dir` high, each MUX passes the value from the left input to the right output.

Left Shift: With `dir` low, each MUX passes the value from the right input to the left output, with `out7` taking the value of `l`.

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) (W5.2)

9.1 Transistors

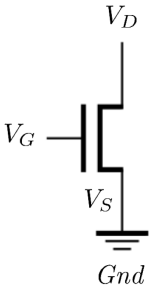
A Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) is a transistor used for amplifying or switching electronic signals. It has three terminals:

- **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.

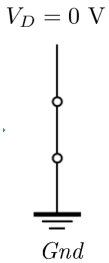
9.1.1 NMOS Transistor Switches

An NMOS is a switch that is open by default, you need to apply a voltage to close it.



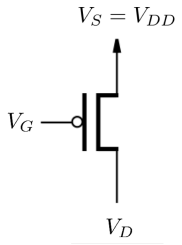
- Open ($x = 0$); $V_G = 0$

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



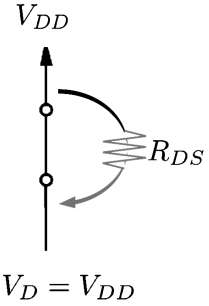
9.1.2 PMOS Transistor Switches

For french people, P like Porte, you push the door (need energy) to open it.
A PMOS is switch that is closed by default, you need to apply a voltage to open it.



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

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

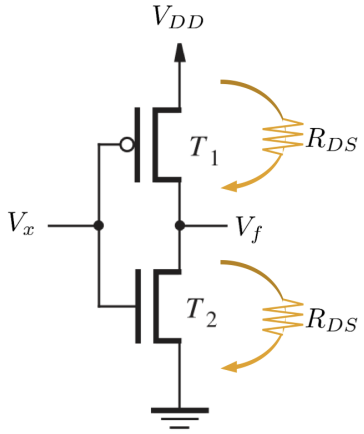


9.1.3 Example - CMOS

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').

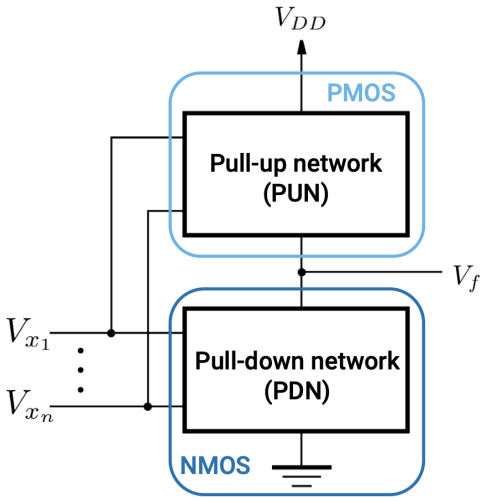
CMOS circuits are power efficient as they draw significant power only during the transition between states (dynamic power consumption), not while in a steady state.

Thus, the truth table:

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

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

9.1.4 CMOS Circuit Structure



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

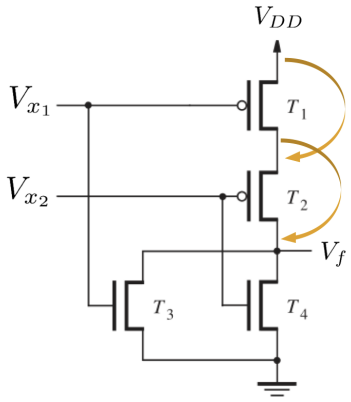
- Pull-up composed of **PMOS**
- Pull-down composed of **NMOS**

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



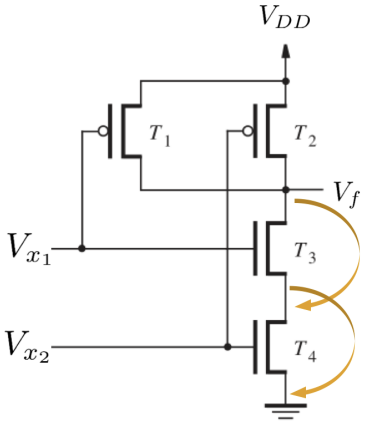
Thus the truth table:

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



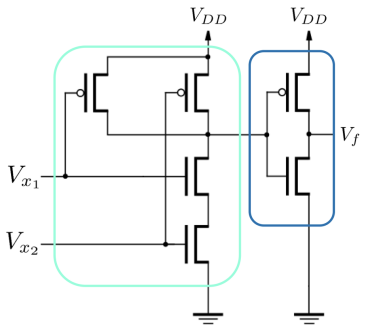
Thus the truth table:

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

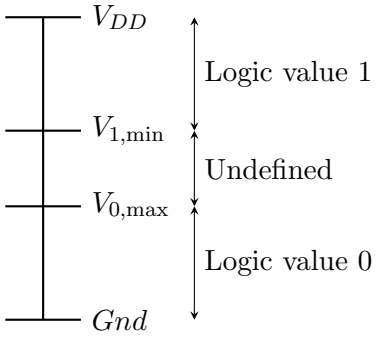
9.2.1 Logic Values as Voltage Levels

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

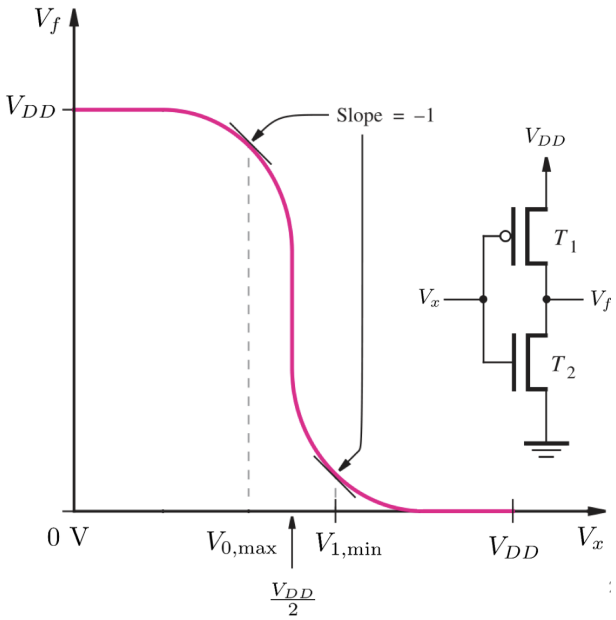
- Logic signals take those intermediate voltage values only while transitioning from one logic value to another.

9.3 Voltage Transfer Characteristic

9.3.1 CMOS Inverter (NOT gate)

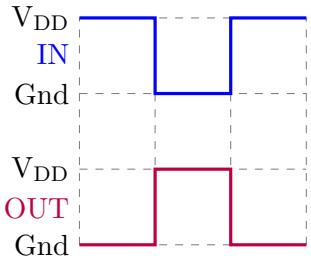
The input-output voltage relationship in a real CMOS inverter is summarized by the voltage transfer characteristic.
When the slope of the curve is -1, we have:

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



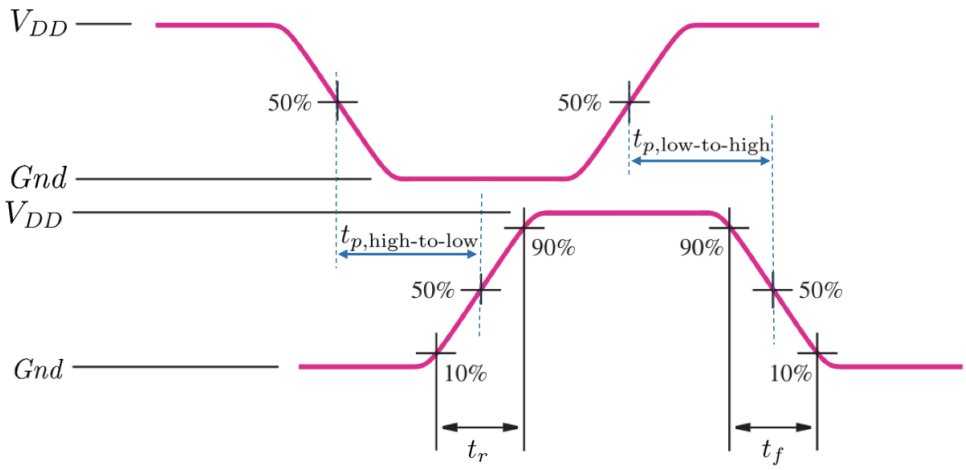
9.3.2 Ideal Waveforms and Real Waveforms

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



In real gates, the transition delay between GND and V_{DD} 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, $t_{p,high-to-low}$ the propagation delay from high to low, and $t_{p,low-to-high}$ 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, $t_{p,high-to-low} \neq t_{p,low-to-high}$

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.* $fan\text{-}in(AND) = 2$

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

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:

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

$$P_D = fCV^2 \tag{9.2}$$

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.

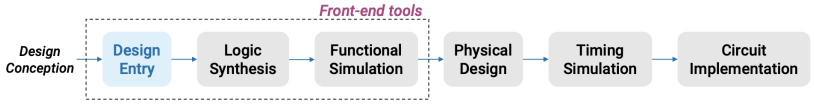
Dynamic Hazard: occurs when a signal is supposed to change: $1 \to 0$ or $0 \to 1$, occurs if such a change involves a short oscillation before the signal settles into its new level.

Chapter 10

Digital Logic and Verilog (PART V) (W6.1)

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

Personal Remark. this course will absolutely NOT suffize 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:

```
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**:

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

- **OR Gate**:

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

- **NOT Gate**:

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

- **NAND Gate**:

```
    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.

```
module module_name [(port_name{, port_name})];
[input declarations]
[output declarations]
[wire declarations]
[logic gate instantiations]
[module instantiations]
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:

```
port\_direction data\_type [port\_size] port\_name;
```

- Implicit type: Unspecified types default to wire.

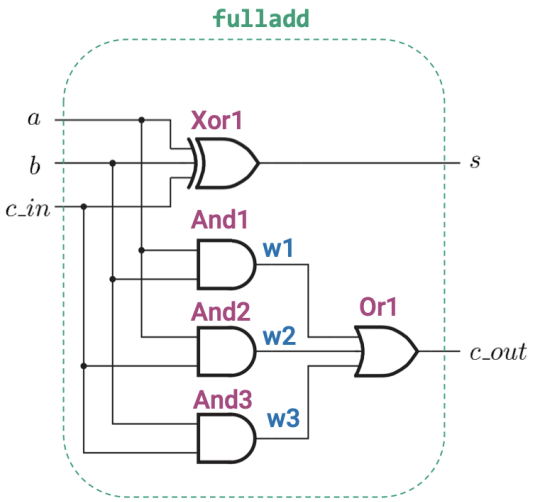
- Vectors: Specify bit-width for multi-bit signals (e.g., [3:0] for 4 bits).

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

```
1  // Structural modeling of a full-
       adder
2  module fulladd (a, b, c_in, s, c_out)
       ;
3  // ----- port definitions -----
4  input a, b, c_in;
5  output s, c_out;
6  // ----- intermediate signals -----
7  wire w1, w2, w3;
8  // ----- design implementation -----
9  and And1 (w1, a, b);
10 and And2 (w2, a, c_in);
11 and And3 (w3, b, c_in);
12 or Or1 (c_out, w1, w2, w3);
13 xor Xor1 (s, a, b, c_in);
14 endmodule
```



Which can be simplified to :

```
1  module fulladd (a, b, c_in, s,
       c_out);
2  input a, b, c_in;
3  output s, c_out;
4  and (w1, a, b);
5  and (w2, a, c_in);
6  and (w3, b, c_in);
7  or (c_out, w1, w2, w3);
8  xor (s, a, b, c_in);
9  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:

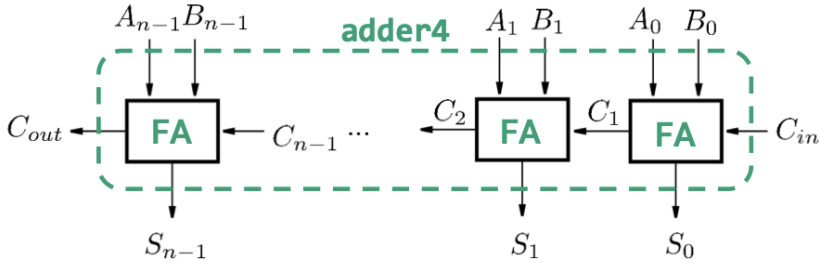
```
1  {module\_name instance\_name ( .port\_name (expression));}
```

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.

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 Verilog :

```
module adder4 (Cin, A, B, S, Cout);
// Port definitions
input Cin;
input [3:0] A, B; // 4-bit vectors
output [3:0] S;   // 4-bit vector
output Cout;

// Intermediate signals
wire [3:1] C; // 3-bit vector for carry bits

// Design implementation using full-adder stages
fulladd stage0 (.c_in(Cin), .a(A[0]), .b(B[0]), .s(S[0]), .c_out(C[1]));
fulladd stage1 (.c_in(C[1]), .a(A[1]), .b(B[1]), .s(S[1]), .c_out(C[2]));
fulladd stage2 (.c_in(C[2]), .a(A[2]), .b(B[2]), .s(S[2]), .c_out(C[3]));
fulladd stage3 (.c_in(C[3]), .a(A[3]), .b(B[3]), .s(S[3]), .c_out(Cout));
endmodule
```

Chapter 11

Digital Logic and Verilog(PART VI)(W6.2)

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*.

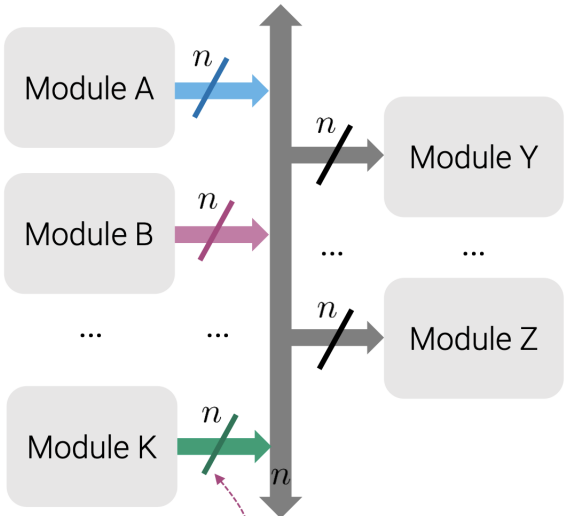
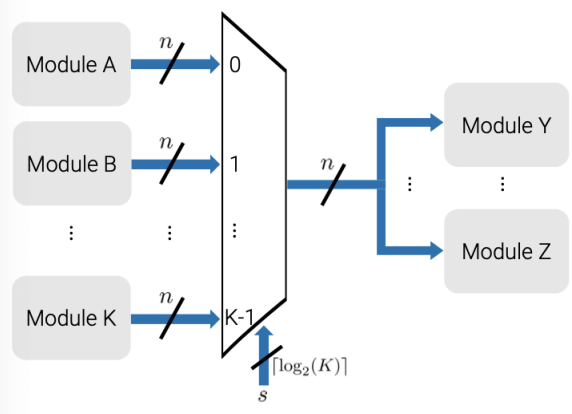


Illustration of an n-bit bus

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 $m = \lceil \log_2(K) \rceil$.

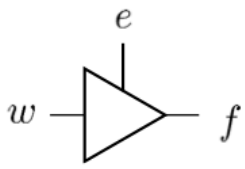


11.1.2 Without Multiplexers

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.*

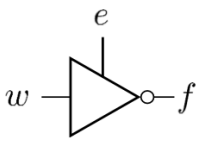


Thus the following table:

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

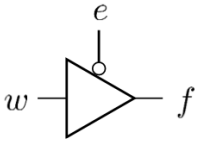
11.2.1 Types of Tri-State Drivers

Enable active high with inverted f



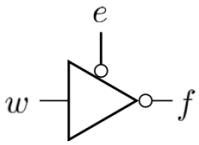
e	w	f
0	0	Z
0	1	Z
1	0	1
1	1	0

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

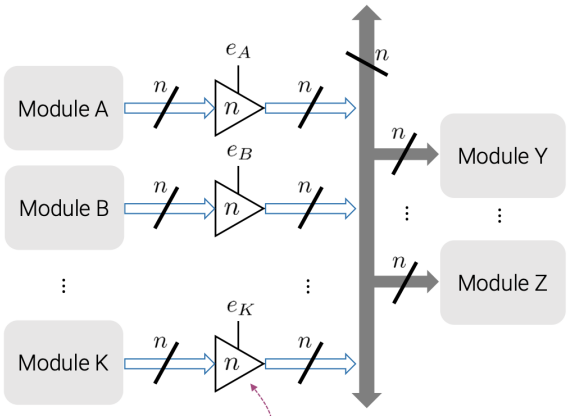
Now we can complete the Verilog built-in gate list presented in the last chapter as follows :

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.

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

```
[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 *default if no radix is specified*
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:

```
parameter name = value;
```

eg.

```
parameter c = 8;
```

11.5.4 Nets

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

wires

```
wire [3:0] a;
```

tri-states

```
tri z1
```

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

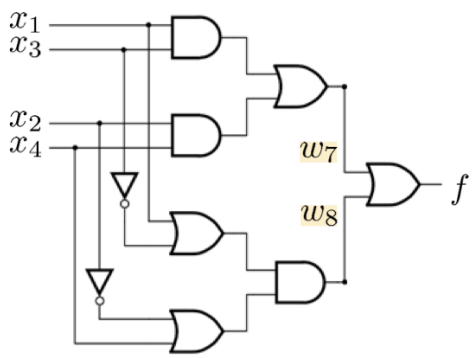
Bit-wise operators in Verilog are used to perform operations on individual bits of a vector. The following are the bit-wise operators available in Verilog:

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




```
module my_circuit_structural (
  input x1, x2, x3, x4,
  output f
  );
  wire w1, w2, w3, w4, w5, w6, w7,
      w8;
  and (w1, x1, x3);
  not (w2, x2);
  not (w3, x3);
  and (w4, x2, x4);
  or (w5, x1, w3);
  or (w6, w2, x4);
  or (w7, w1, w4);
  and (w8, w5, w6);
  or (f, w7, w8);
  endmodule
```

Behavioral

```
module my_circuit_behavioral (
  input x1, x2, x3, x4,
  output f
  );
  wire w7, w8;
  assign w7 = (x1 & x3) | (x2 & x4);
  assign w8 = (x1 | ~x3) & (~x2 | x4);
  assign f = w7 | w8;
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`:

```
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

```
always @*
  [begin]
  [procedural assignment statements]
  [if-else statements]
  [case statements]
  [while, repeat, and for loops]
  [task and function calls]
[end]
```

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

11.5.10 If-Else Statements

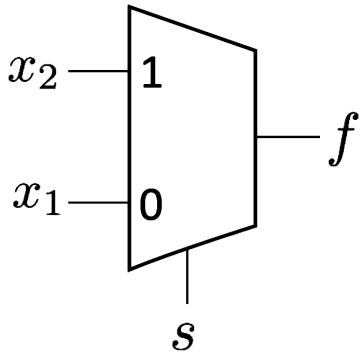
If an expression is true, the code within the begin-end block runs; multiple statements can be grouped in such a block. Optional else if and else clauses, when used, connect to the closest preceding if or else if statement.

Structure

```
if (expression1)
begin
  statement;
end
else if (expression2)
begin
  statement;
end
else
begin
  statement;
end
```

11.5.11 Example - 2-to-1 MUX

Using what we've learned so far




```
module my_2to1mux ( input w1,
    w2, s, output reg f );
  always @*
    begin
    if (s == 0)
    begin
      f = w1;
    end
    else
    begin
      f = w2;
    end
  end
endmodule
```

11.5.12 Case Statement

Personal Remark. Switch in java

Syntax

```
case (expression)
    alternative1:
  begin
        statements;
    end
    alternative2:
  begin
        statements;
    end
    default:
  begin
        statements;
    end
endcase
```

Example - Full Adder

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

x	y	C_{in}	s	C_{out}
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

```
module fulladd (input x, y, Cin,
   output reg s, Cout);
  always @*
  begin
    case ({x, y, Cin})
    3'b000: {s, Cout} = 'b00;
    3'b001: {s, Cout} = 'b10;
    3'b010: {s, Cout} = 'b10;
    3'b011: {s, Cout} = 'b01;
    3'b100: {s, Cout} = 'b10;
    3'b101: {s, Cout} = 'b01;
    3'b110: {s, Cout} = 'b01;
    3'b111: {s, Cout} = 'b11;
    endcase
  end
endmodule
```

Chapter 12

Digital Logic and Verilog(PART VII) (W7.2)

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:

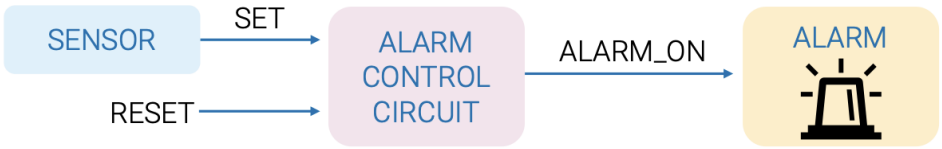


Figure 12.1: Diagram of the Alarm System Control

- 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).

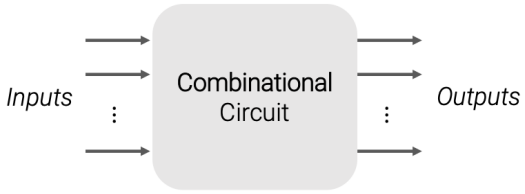
- A memory element in the circuit ensures that the alarm remains active until it is explicitly reset, even if the `SET` signal goes back to low (0).

12.1.2 Combinational vs. Sequential Circuits

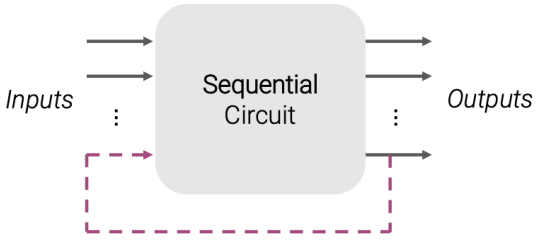
Personal Remark. Please take some time to actually identify the differences between the two as they will be crucial in understanding the rest of the course
Tip: C comes before S in the alphabet, we've seen combinational circuits before sequential circuits. So you can think of combinational as what we've seen so far, and sequential as the new circuit type
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.

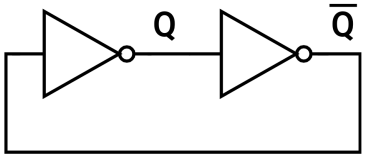


Storage elements in a circuit hold logic signal values, defining the circuit's state. When inputs change, the circuit may maintain its current state or transition to a new state. Consequently, the circuit evolves through various states over time based on input changes.

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	$\overline{Q}$
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.

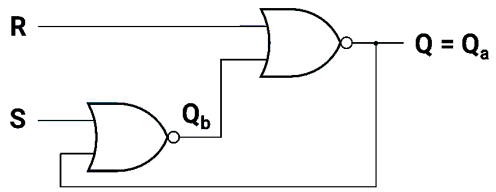
Set-Reset Latch with Reset Priority

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.



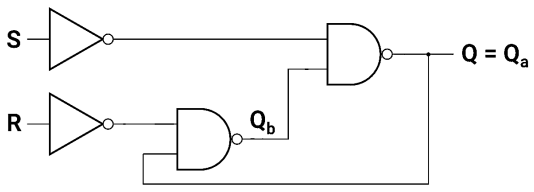
S	R	$\bar{R} \cdot S$	Q_a	Q_{next}	$Q_{b,\text{next}}$
0	0	0	Q_a	Q_a	$\bar{Q}_a$
0	1	0	0	0	1
1	0	1	Q_a	1	0
1	1	0	0	0	0

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_{next} and $Q_{b,\text{next}}$ respectively.

$$Q_{a,next} = \overline{R + Q_b} = \overline{R + \overline{S + Q_a}}$$
$$= \overline{R} \cdot (S + Q_a)$$
$$= \overline{R} \cdot S + \overline{R} \cdot Q_a = Q_{next}$$
$$Q_b = \overline{S + Q_a}$$

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).



S	R	$\bar{R} \cdot Q_a$	$Q_{a,\text{next}}$	$Q_{b,\text{next}}$
0	0	Q_a	Q_a	Q_a
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

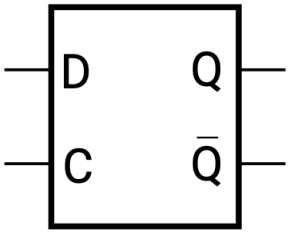
$$Q_{next} = S + \bar{R} \cdot Q_a$$

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.

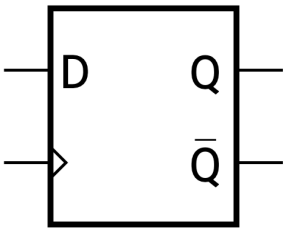


12.2.2 Flip-Flops

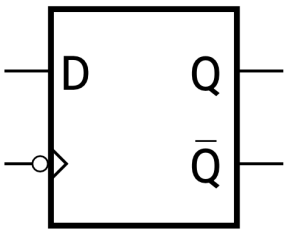
Personal Remark. Put simply, the first type requires actual C input to determine when to update, while the other type updates Q in response to an event-driven signal, such as a clock signal (refer to the following section about Clock signals for a clearer explanation).

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

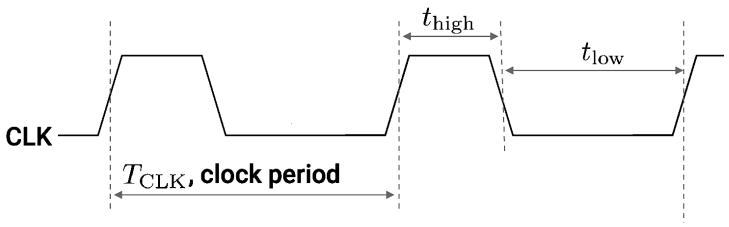
Clocks are essential in digital systems for synchronization purposes. The timing of state changes within the system is orchestrated by these clocks.

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.

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 (≤).

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).

```
always @ (...sensitivity list...)
begin
  B = A;
  C = B;
  D = C;
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).

```
always @ (...sensitivity list...)
begin
  B <= A;
  C <= B;
  D <= C;
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
always @ (*)
begin
  C = A & B;
end
```

always@(posedge Clock) Blocks

(≤) Nonblocking Statements should be used.
Used to describe sequential logic containing flip-flops
Note: Remember, ≤ 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.

```
module Dlatch (
  input D,
  input C,
  output reg Q
  );
  always @ (*)
  begin
    if (C == 1)
    Q <= D;
  end
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:

```
module Dlatch (input D, input C, input CE, input CLR,output reg Q);
  always @ (*)
  begin
    if (CLR == 1)
      Q <= 0;
    else if ((C == 1) && (CE == 1))
      Q <= D;
  end
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

```
module Dff (input D, input CLK, output reg Q);
  always @ (posedge CLK)
    begin
      Q <= D;
    end
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.

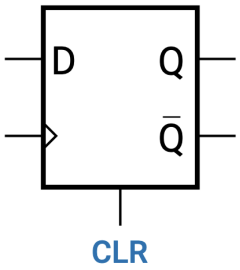
```
module Dff (input D, input CLK, input CLR, output reg Q);
  always @ (posedge CLK or posedge CLR)
    begin
      if (CLR == 1)
        Q <= 0;
      else
        Q <= D;
    end
endmodule
```

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 ∼ Q Output

The exact same but with an additional output ∼ Q which is the complementary of Q.

```
module Dff (input D, CLK, CLR, output reg Q, QN);
  always @ (posedge CLK or posedge CLR)
  begin
    if (CLR == 1) begin
      Q <= 0;
      QN <= 1; // complementary output
    end
    else begin
      Q <= D;
      QN <= ~D; // complementary output
    end
  end
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.

```
module Dff (input D, input CLK, input S, input CE, output reg Q);
  always @ (posedge CLK)
    if (S == 1)
      Q <= 1;
    else if (CE == 1)
      Q <= D;
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.

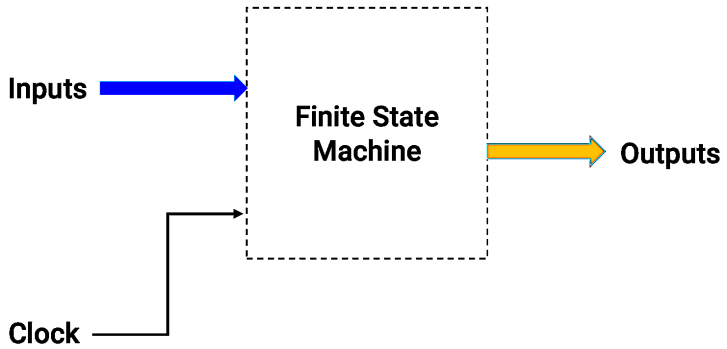
In sequential logic, always employ nonblocking assignments using the <= operator in `always` blocks to ensure simulation consistency across multiple blocks, as this method evaluates all right-hand sides before assigning values to the left-hand sides.

Chapter 13

Digital Logic and Verilog(PART VIII) (W8.1)

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)**



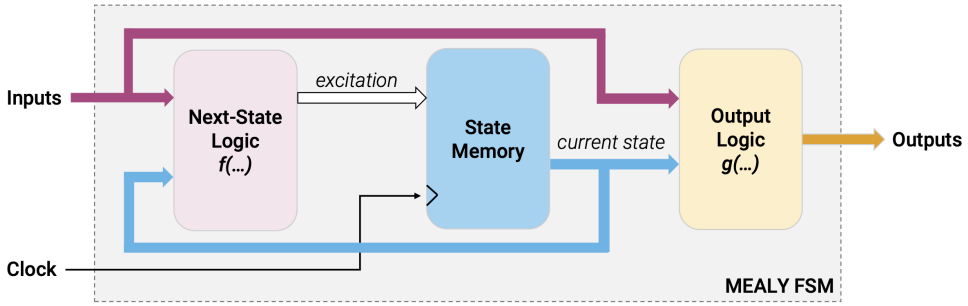
There are two types of FSMs:

- **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: Mnemothecnics: the double o in Moore: Only Output State. (uhm couldn't find anything better)*

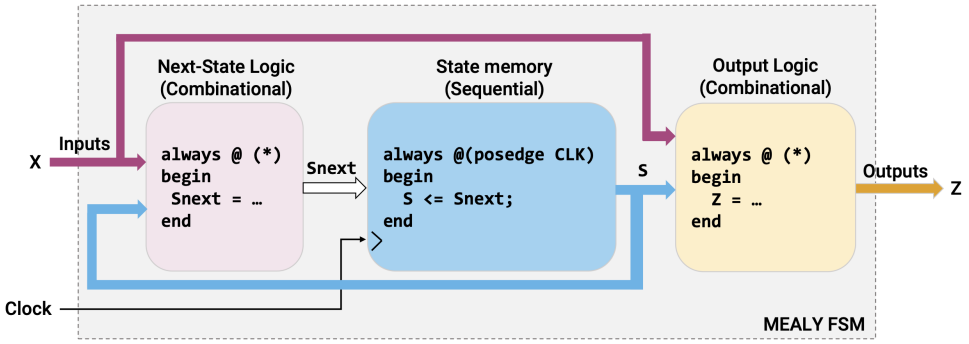
13.1.1 Mealy State Machine

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(current state, input)
- The **output** is a function of the current state and inputs
  eg. Output = g(current state, input)

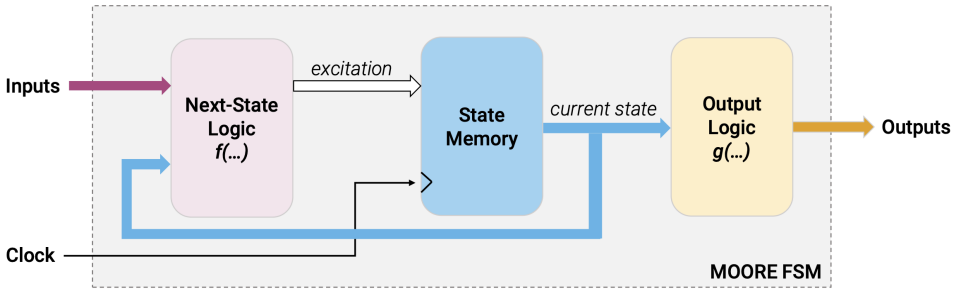
In a more detailed view:



Note: The sensitivity list of the state machine may be ajusted as we've seen

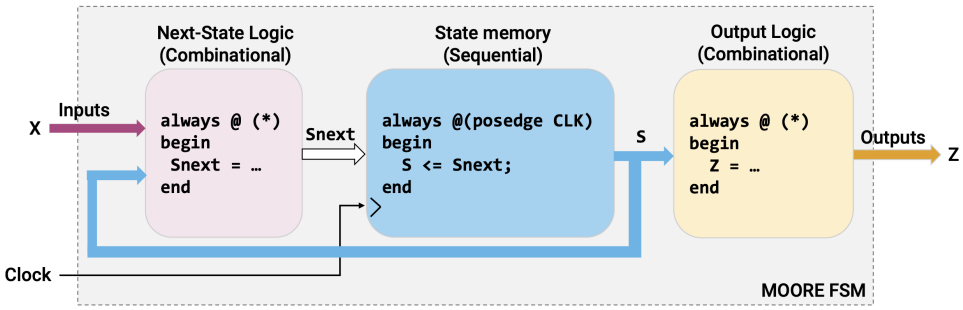
13.1.2 Moore State Machine

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(current state, input)
- *(Difference is here)* The **output** is a function of the current state **only**
  eg. Output = g(current state)

In a more detailed view:



Note: The sensitivity list of the state machine may be ajusted as we've seen

13.2 State Machine Analysis

Here we'll be looking at how he 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 contruct a state/output table that shows the entirety of the next state and the output of the circuit for all possible inputs and states.

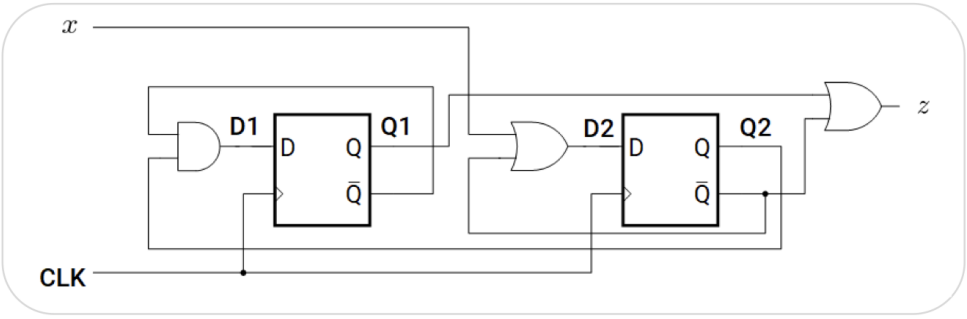
Step 3 (Optional): Draw the state diagram of the state machine using the state/output table.

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.

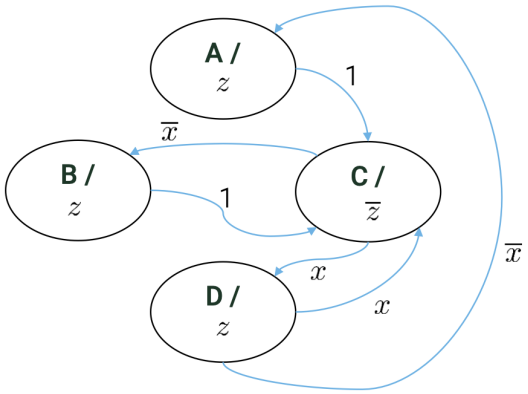
$$Q_1^* = D_1 = f_1(x, Q_1, Q_2) = \overline{Q}_1 \cdot Q_2$$
$$Q_2^* = D_2 = f_2(x, Q_1, Q_2) = x + \overline{Q}_2$$
$$z = g(x, Q_1, Q_2) = Q_1 + \overline{Q}_2$$

State variables	Excitation D2D1		Outputs
Q2 Q1	x		z
	0	1	
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		z
	0	1	
A	C	C	1
B	C	C	1
C	B	D	0
D	A	C	1

Step 3: Draw the state diagram.

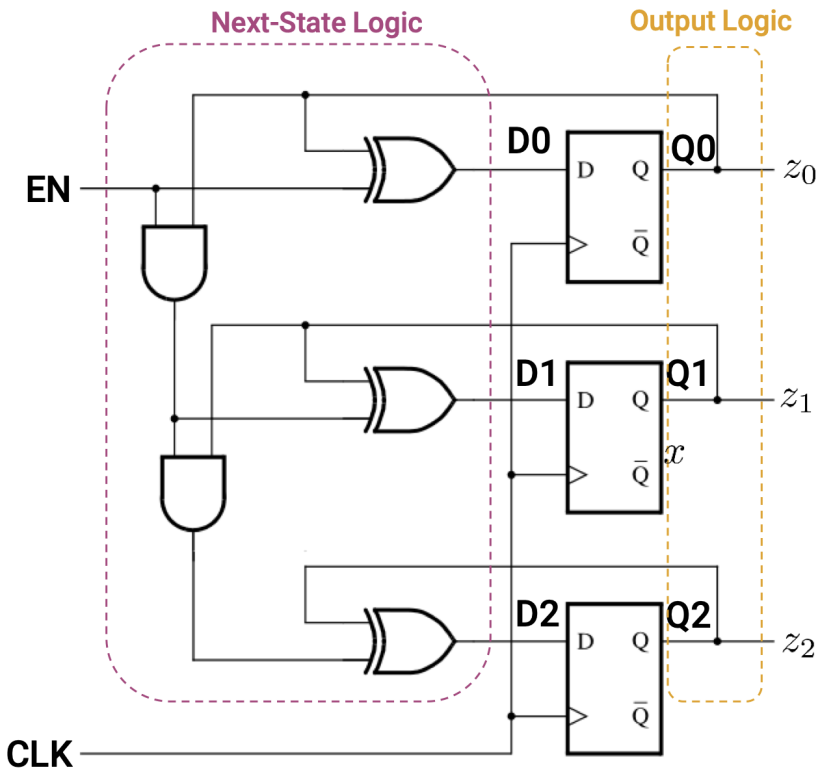


Written in Verilog:

```
module fsm_example1 (input x,
    input CLK, output reg z);
  reg [2:1] D, Q;

  // Next-state logic
  always @ (*) begin
    D[1] = ~Q[1] & Q[2];
    D[2] = x | ~Q[2];
  end

  // State memory
  always @ (posedge CLK) begin
    Q <= D;
  end

  // Output logic
  always @ (*) begin
    z = Q[1] | ~Q[2];
  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.

$$Q_0^* = D_0 = f_0(EN, Q_0, Q_1, Q_2) = Q_0 \oplus EN$$
$$Q_1^* = D_1 = f_1(EN, Q_0, Q_1, Q_2) = Q_1 \oplus (Q_0, EN)$$
$$Q_2^* = D_2 = f_2(EN, Q_0, Q_1, Q_2) = Q_2 \oplus (Q_0 Q_1 EN)$$
$$z = g_i(Q_i) = Q_i, 0 \leq i \leq 2$$

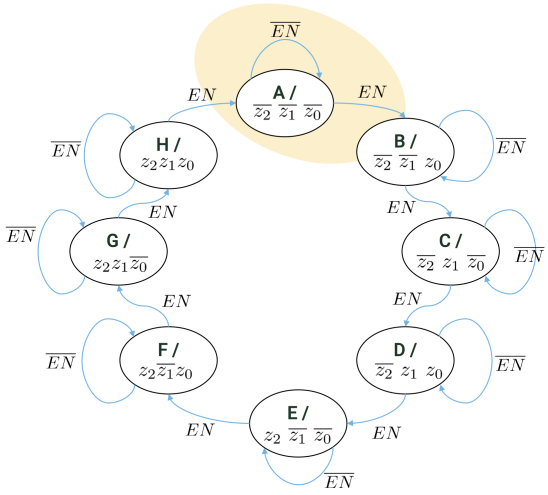
State variables	Excitation D2D1		Outputs
Q2 Q1 Q0	x = 0	x = 1	z
0 0 0	000	001	1
0 0 1	001	010	0
0 1 0	010	011	1
0 1 1	011	100	0
1 0 0	100	101	1
1 0 1	101	110	0
1 1 0	110	111	1
1 1 1	111	000	0

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
S	EN		$z_2\ z_1\ z_0$
	0	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.




```
module fsm_example2 ( input EN, input
    CLK, output reg [2:0] z);
reg [2:0] D, Q;
  // Next-state logic
  always @ (*) begin
    D[0] = Q[0] ^ EN;
    D[1] = Q[1] ^ (Q[0] & EN);
    D[2] = Q[2] ^ (Q[1] & Q[0] & EN);
  end
  // State memory
  always @ (posedge CLK) begin
    Q <= D;
  end
  // Output logic
  always @ (*) begin
    z = Q;
  end
endmodule
```

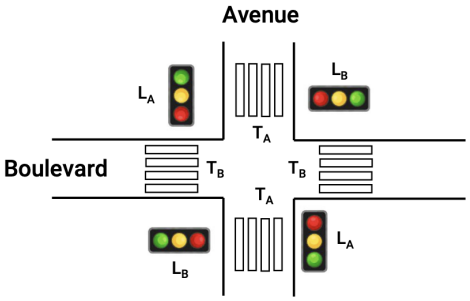
13.3 State Machine Synthesis

FSM Design Algorithm

- x Identify FSM inputs and outputs.
- x Draw the state diagram.
- x Contruct 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 trafic light controller Let T_A, T_B : Signals from the trafic sensors; active when there is traffic L_A, L_B : Signals for controlling traffic lights



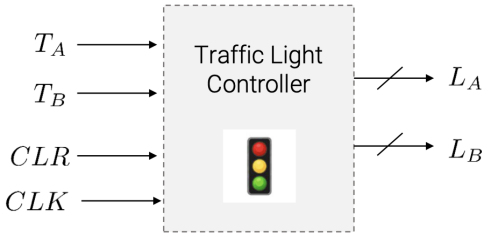
Identify FSM inputs and outputs:

Two one-bit inputs from the traffic sensors: T_A, T_B

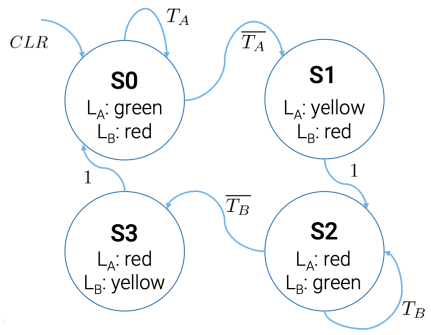
Multiple bits outputs(three states: red, yellow, green): L_A, L_B

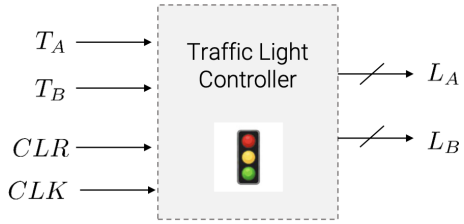
CLK, system clock

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_A	T_B	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_A	T_B	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 :

$$Q_1^* = D_1 = \overline{Q_1}Q_0 + Q_1\overline{Q_0} + Q_1\overline{Q_0} = Q_0 \oplus Q_1$$
$$Q_0^* = D_0 = \overline{Q_1}\overline{Q_0}\overline{T_A} + Q_1\overline{Q_0}\overline{T_B}$$

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			
Q1	Q0	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:

```
module traffic_light_ctrl (input TA, TB, CLR, CLK, output reg [1:0] LA, LB)
   ;
  reg [1:0] D, Q;
  // Next-state logic
  always @ (*) begin
    D[1] = (~Q[1] & Q[0]) | (Q[1] & ~Q[0]);
    D[0] = (~Q[1] & ~Q[0] & ~TA)
    | (Q[1] & ~Q[0] & ~TB);
  end
  // State memory
  always @ (posedge CLK or posedge CLR)
  begin
  if (CLR == 1)
    Q <= 0;
  else
    Q <= D;
  end
  // Output logic
  always @ (*) begin
    LA[1] = Q[1]; LA[0] = ~Q[1] & Q[0];
    LB[1] = ~Q[1]; LB[0] = Q[1] & Q[0];
  end
endmodule
```

Chapter 14

Digital logic and Verilog, Part IX (W8.2)

14.1 Sequential Circuits

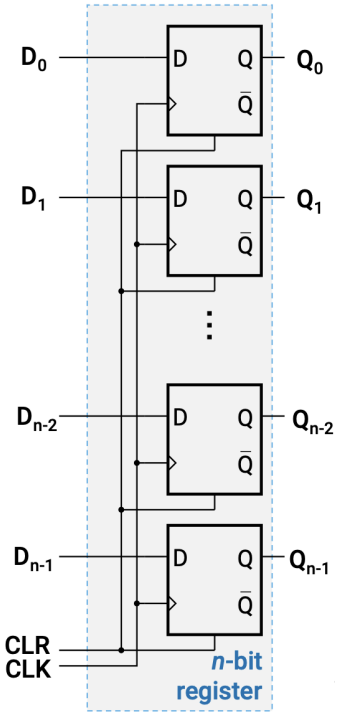
14.1.1 Registers

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)

```
module regn (D, CLK, CLR, Q);
parameter n = 16; // 16 as an example
input [n-1:0] D;
input CLR, CLK;
output reg [n-1:0] Q;
always @ (posedge CLK or posedge CLR)
  begin
    if (CLR == 1)
      Q <= 0;
  else
    Q <= D;
end
endmodule
```

14.2 Shift Registers

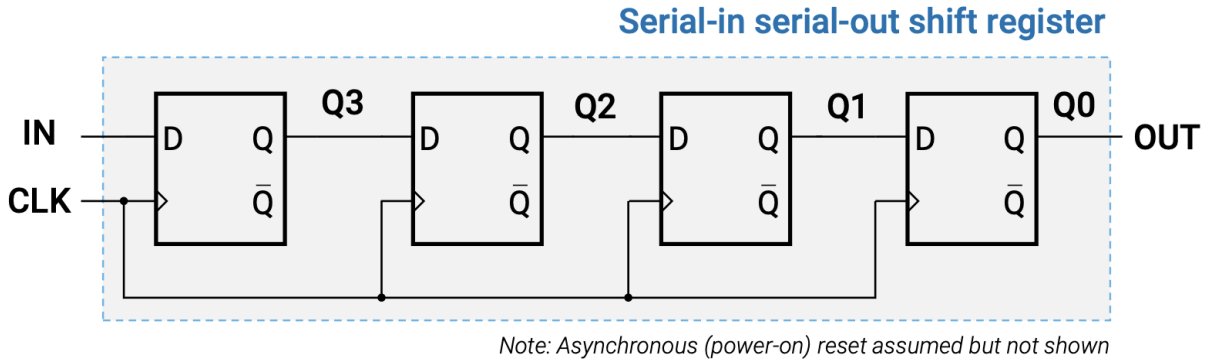
14.2.1 Serial Input, Serial Output Shift Register

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.



Note: Asynchronous (power-on) reset assumed but not shown

This visually shows the shiffting 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.

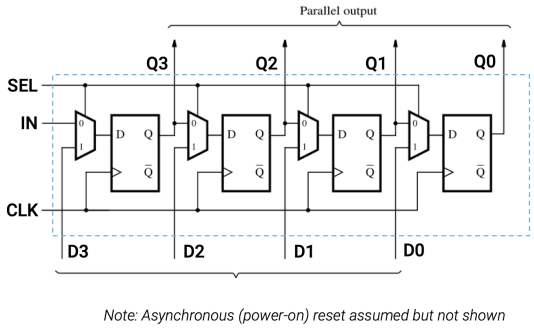
T_{CLK}	IN	Q3	Q2	Q1	Q0 (OUT)
t0	1	0	0	0	0
t1	0	1	0	0	0
t2	1	0	1	0	0
t3	1	1	0	1	0
t4	1	1	1	0	1
t5	0	1	1	1	0
t6	0	0	1	1	1
t7	0	0	0	1	1

14.2.2 Parallel Input, Parallel Output Shift Register

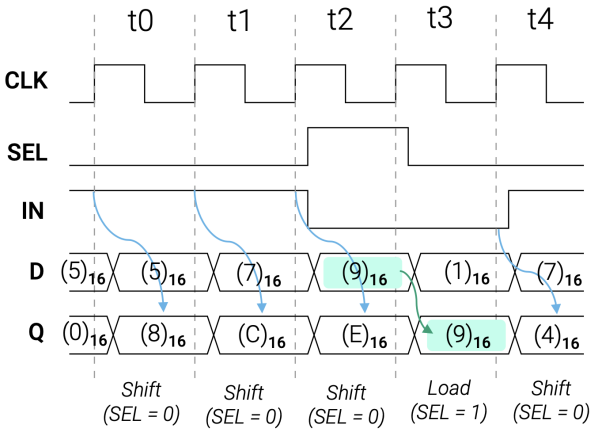
Faster than last one, allows shiffting multiple bits at once

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)

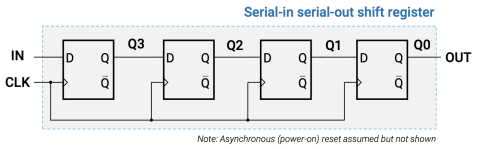


14.3 Shift Registers (Verilog)

Section 14.3 covers this part in details.

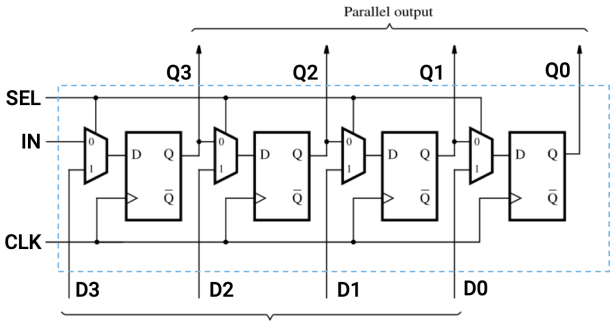
14.3.1 Serial input/output Shift Registers

```
module shiftn (IN, CLR, CLK,
    OUT);
  parameter n = 4; // 4-bit
      shifter
  input IN, CLR, CLK;
  output OUT;
  reg [n-1:0] Q; // internal
  always @ (posedge CLK or
      posedge CLR)
    begin
      if (CLR == 1) Q <= 0;
      else Q <= {IN, Q[n
          -1:1]};
    end
  assign OUT = Q[0];
endmodule
```



14.3.2 Parallel input/output Shift Registers

```
module shiftn (D, IN, SEL, CLR
    , CLK, Q);
  parameter n = 4; // 4-bit
      reg
  input [n-1:0] D;
  input IN, SEL, CLR, CLK;
  output reg [n-1:0] Q;
  integer k; // loop iterator
  always @ (posedge CLK or
      posedge CLR) begin
    if (CLR == 1) Q <= 0;
    else if (SEL == 1) Q <= D;
    else begin
      for (k = 0; k < n-1; k =
          k+1) // for loop
      begin
        Q[k] <= Q[k+1];
      end
      Q[n-1] <= IN;
    end
  end
endmodule
```



Note: Asynchronous (power-on) reset assumed but not shown

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:

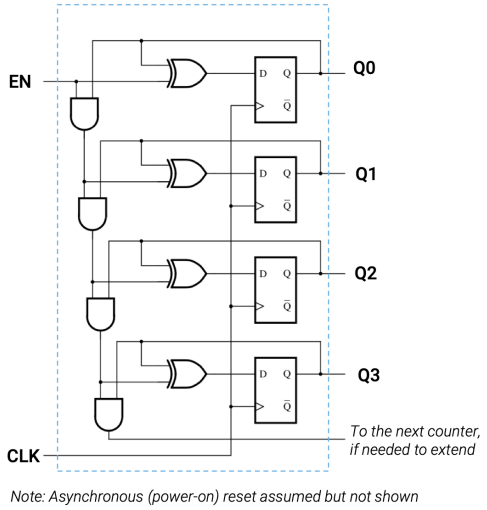
$$D_0 = Q_0 \oplus EN$$
$$D_1 = Q_1 \oplus (Q_0 EN)$$
$$D_2 = Q_2 \oplus (Q_0 Q_1 EN)$$
$$D_3 = Q_3 \oplus (Q_0 Q_1 Q_2 EN)$$

In an n-bit counter case:

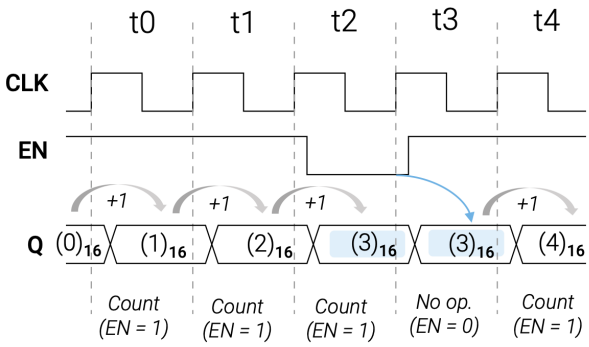
$$D_i = Q_i \oplus (Q_0 Q_1 \cdots Q_{i-2} Q_{i-1} EN)$$

Modulo 2^n counter:

- Every 2^n clock cycles, the output is zero



With the corresponding timing diagram:



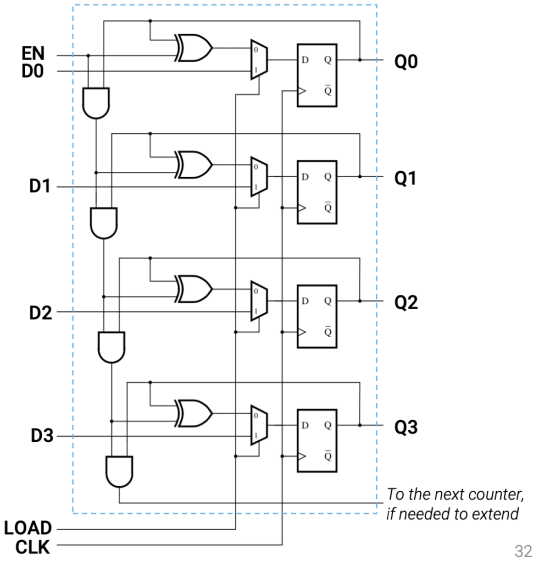
Example - Counter with Parallel-Load Capability

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

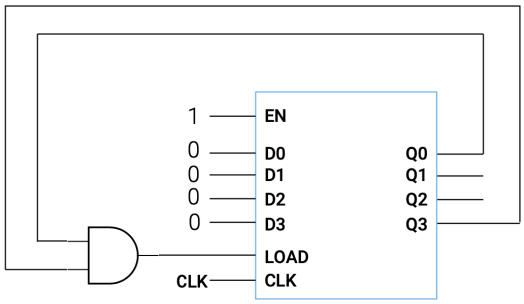


Note: Asynchronous (power-on) reset assumed but not shown

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.

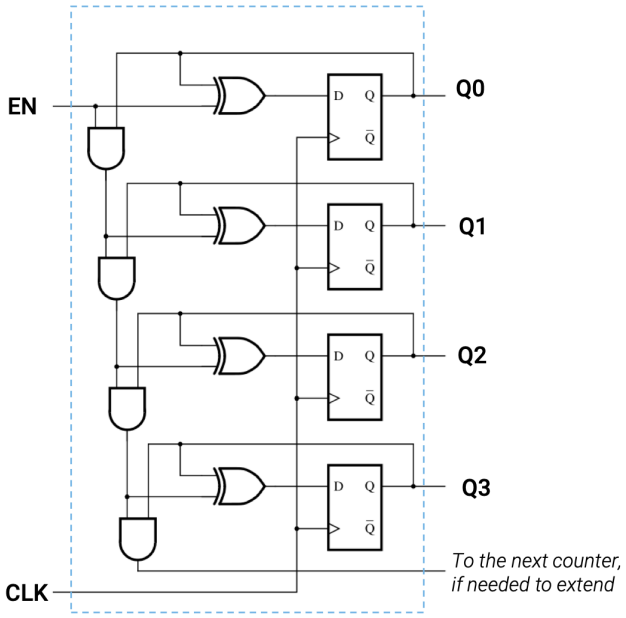


14.5 Counters (Verilog)

Section 14.5 covers this part in details.

14.5.1 Counter Up with an Enable Signal

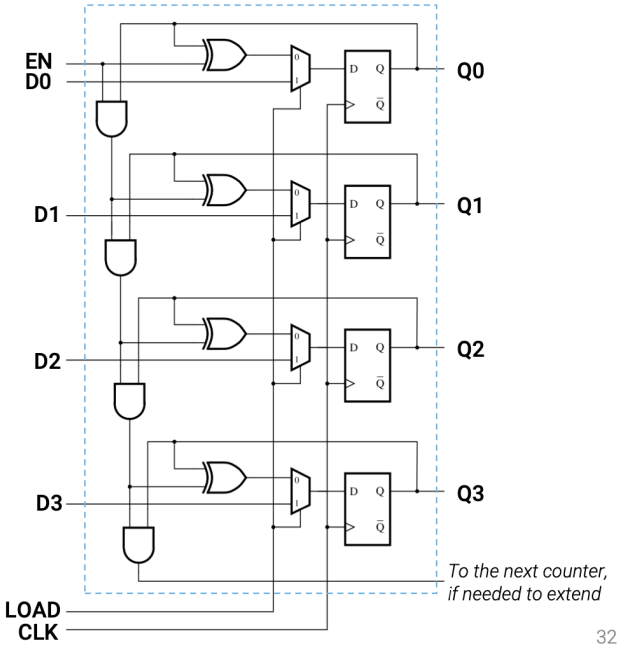
```
module upcount (EN, CLR, CLK,
    Q);
parameter n = 4; // 4-bit
    counter
input EN, CLR, CLK;
output reg [n-1:0] Q;
  always @ (posedge CLK or
      posedge CLR)
  begin
    if (CLR == 1) Q <= 0;
    else if (EN == 1) Q <= Q +
        1;
  end
endmodule
```



Note: Asynchronous (power-on) reset assumed but not shown

14.5.2 Counter with Parallel-Load Capability

```
module upcount (D, EN, LOAD,
    CLR, CLK, Q);
parameter n = 4; // 4-bit
    counter
input [n-1:0] D;
input EN, LOAD, CLR, CLK;
output reg [n-1:0] Q;
  always @ (posedge CLK or
      posedge CLR)
  begin
    if (CLR == 1) Q <= 0;
    else if (LOAD == 1) Q <= D
        ;
    else if (EN == 1) Q <= Q +
        1;
  end
endmodule
```



32

Note: Asynchronous (power-on) reset assumed but not shown

14.6 Verilog Part 4.

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`)

```
for (initial index; terminal
    index; increment)
begin
  statements;
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 <= 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.

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) (W9.1-W10.1)

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

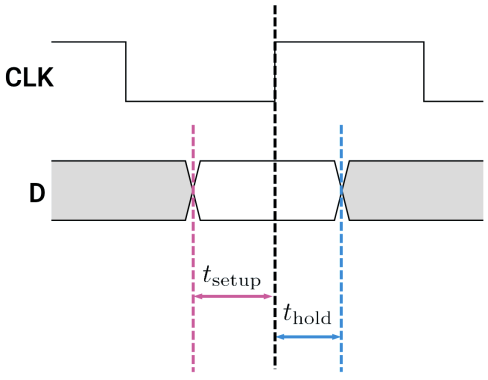
The signal on the input D of a D Flip-Flop (DFF) must be stable around the **active clock edge** (the edge that captures the signal).

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**.

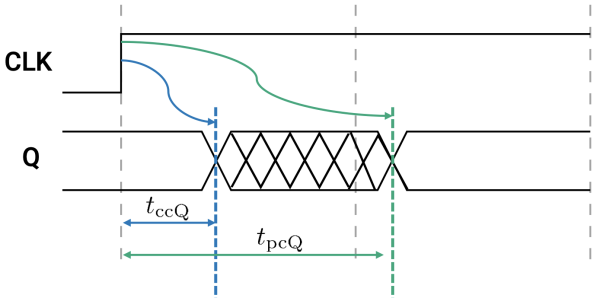


D Flip-Flop Output Timing Parameters

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}.



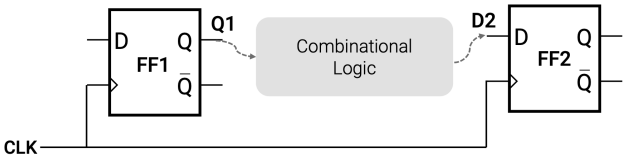
15.1.2 Meeting FF Timing Constraints

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_{setup} before the active clock edge.
- At least t_{hold} after the active clock edge.

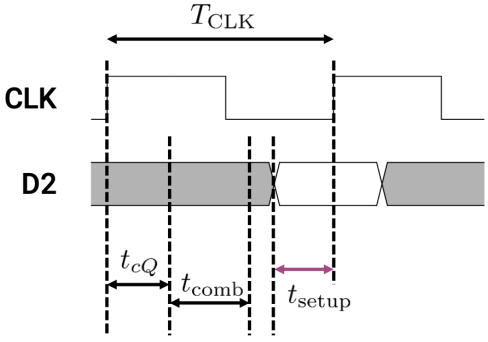
Longest and shortest delay paths are most important.



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.
>
> Thus, $t_{\text{setup}} + t_{cQ,\text{max}} + t_{\text{comb,max}} \leq T_{\text{CLK}}$
>
> $t_{cQ,\text{max}}$: *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:

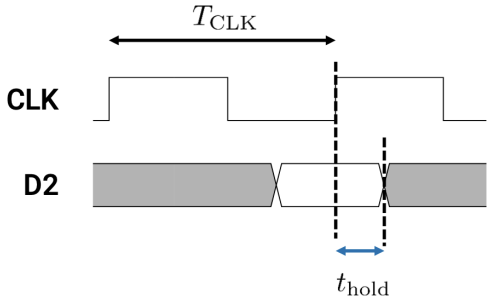
The total delay should be more than the hold time.

Formula: $t_{cQ,\text{min}} + t_{\text{comb,min}} \geq t_{\text{hold}}$

$t_{cQ,\text{min}}$: *Shortest clock-to-Q delay*

$t_{\text{comb,min}}$: *Shortest path delay through combinational logic*

Typically, $t_{\text{hold}} \approx 0$ in real circuits.



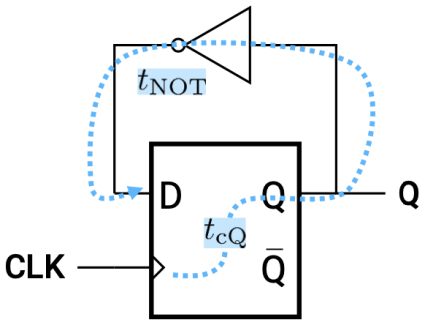
15.1.3 Timing Analysis of a Simple Circuit

Personal Remark: Read the parameters then read the Hold-Time Violations part to better understand what we're trying to achive

Given the following flip-flop (FF) timing parameters:

- Setup time, $t_{setup} = 0.6\,\text{ns}$
- Hold time, $t_{hold} = 0.4\,\text{ns}$
- Clock-to-Q delay, t_{CQ}: $0.8\,\text{ns} \leq t_{CQ} \leq 1.0\,\text{ns}$

and the delay of the NOT gate, $t_{NOT} = 1.1\,\text{ns}$.



Maximum Clock Frequency

To find the maximum clock frequency f_{max} for which the circuit will operate properly, we consider the worst-case delay path:

1. Starts when data is loaded into the FF on the rising clock edge.
2. Propagates to the Q output after t_{CQ}.
3. Propagates through the NOT gate with delay t_{NOT}.
4. Ends at the input D, where it must meet the setup requirement.

Blue path in the upper image

Thus, the shortest allowed clock period T_{min} is given by:

$$T_{min} = t_{setup} + t_{CQ,\text{max}} + t_{NOT} = 0.6\,\text{ns} + 1.0\,\text{ns} + 1.1\,\text{ns} = 2.7\,\text{ns}$$

And the maximum operating frequency f_{max} is:

$$f_{max} = \frac{1}{T_{min}} = \frac{1}{2.7\,\text{ns}} \approx 370.37\,\text{MHz}$$

Hold-Time Violations

To check for hold-time violations, we examine the shortest path between the rising clock edge and the change of input D:

$$t_{min} = t_{CQ,\text{min}} + t_{NOT} = 0.8\,\text{ns} + 1.1\,\text{ns} = 1.9\,\text{ns}$$

Hold-time constraints are satisfied if:

$$t_{min} \geq t_{hold}$$

In this circuit, the hold-time constraints are satisfied as:

$$1.9\,\text{ns} \geq 0.4\,\text{ns}$$

Therefore, the circuit does not suffer from hold-time violations.

15.1.4 Timing Analysis of a Counter

Given the same FF timing parameters as in the previous example:

Setup time, $t_{setup} = 0.6\,\text{ns}$

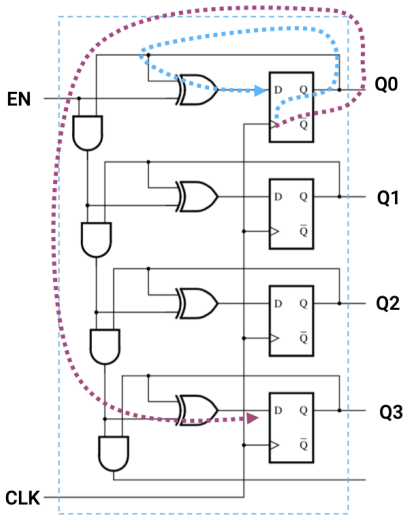
Hold time, $t_{hold} = 0.4\,\text{ns}$

Clock-to-Q delay,
t_{CQ}: $0.8\,\text{ns} \leq t_{CQ} \leq 1.0\,\text{ns}$

With gate delays:

AND gate delay, $t_{AND} = 1.2\,\text{ns}$

XOR gate delay, $t_{XOR} = 1.3\,\text{ns}$



Maximum Operating Frequency

To find the maximum operating frequency f_{max}, we consider the longest path in the circuit:

Paths from output Q_0 to the inputs of FFs:

$$t(Q_0, D_i) = t_{CQ} + i \cdot t_{AND} + t_{XOR}, \quad 0 \leq i \leq 3$$

The longest path is $t(Q_0, D_3) = t_{CQ,\text{max}} + 3 \cdot t_{AND} + t_{XOR} = 1.0\,\text{ns} + 3 \cdot 1.2\,\text{ns} + 1.3\,\text{ns} = 5.9\,\text{ns}$

The shortest clock period that satisfies the setup-time requirements is:

$$T_{min} = t_{setup} + t_{max} = 0.6\,\text{ns} + 5.9\,\text{ns} = 6.5\,\text{ns}$$

Thus, the maximum operating frequency f_{max} is:

$$f_{max} = \frac{1}{T_{min}} = \frac{1}{6.5\,\text{ns}} \approx 153.84\,\text{MHz}$$

Hold-Time Violations

To check for hold-time violations, we examine the shortest paths between the rising clock edge and the change of one of the FF inputs:

Shortest path is $t(Q_i, D_i) = t_{CQ} + t_{XOR}, \quad 0 \leq i \leq 3$

Minimum delay $t_{min} = t_{CQ,\text{min}} + t_{XOR} = 0.8\,\text{ns} + 1.3\,\text{ns} = 2.1\,\text{ns}$

Hold-time requirements are satisfied if:

$$t_{min} \geq t_{hold}$$

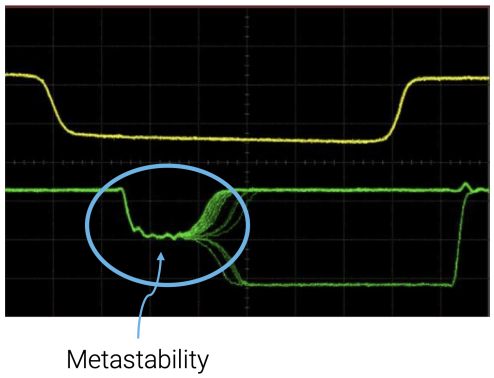
In this circuit, the hold-time requirements are satisfied as:

$$2.1\,\text{ns} \geq 0.4\,\text{ns}$$

Therefore, the circuit does not suffer from hold-time violations.

15.2 Metastability

Metastability occurs in a D flip-flop when the data input D is not stable around the active clock edge.
This causes the output to initially be stuck at an intermediate voltage level before eventually settling to a logic 0 or 1 in an unpredictable manner.



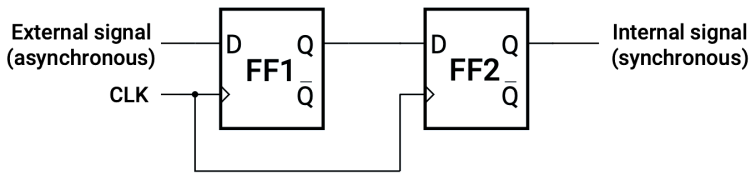
15.3 Synchronizer (Shift Register)

Metastability arises from asynchronous input signals and can be mitigated by using a shift register in the signal path.

FF1's output may enter metastability.

A sufficiently long clock period allows FF1's output to stabilize to 0 or 1 before the next clock edge.

FF2 then operates correctly, preventing metastability from propagating.

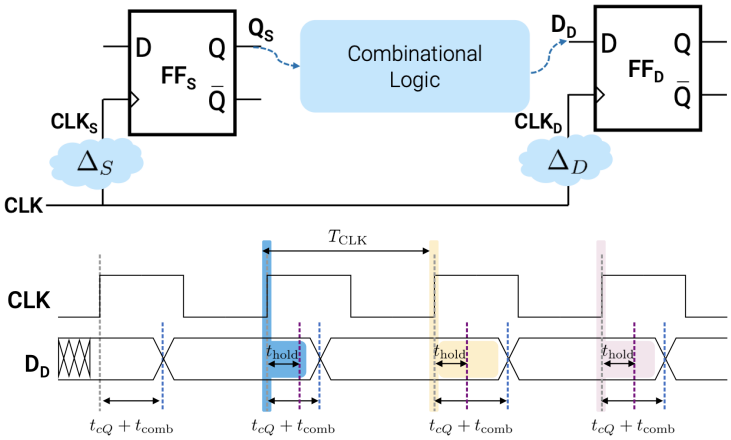


For this part, I took some liberties in the way I organized the ideas presented by the professor as I found that presenting the same idea twice, with then without clock skew, was quite lengthy and redundant. Of course, feel free to take a look at the Digital Logic and Verilog X (extended), if you feel like the professor's approach was better.

15.4 Life of D: Lasting One Clock Cycle

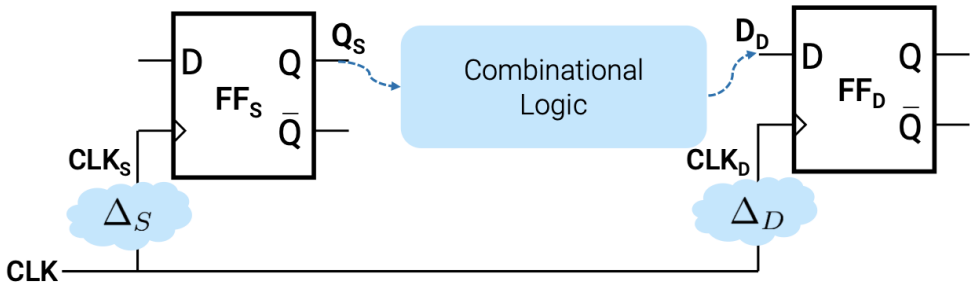
The signal 'D' undergoes a lifecycle within a digital system that repeats every clock cycle. This lifecycle can be described as follows:

1. **Born as Q of the Source Flip-Flop (FF):** The signal D starts its journey as the output (Q) of a source flip-flop.
2. **Journey through Combinational Logic:** After being emitted from the source flip-flop, 'D' travels through various combinational logic components. $t_{CQ} + t_{comb}$
3. **Arrives at the Destination FF:** The signal 'D' reaches the input of the destination flip-flop. $t_{arrival} = t_{CQ} + t_{comb}$
4. **Waits to be Captured on Clock Edge:** At the destination flip-flop, 'D' waits for the next clock edge.
5. **Gets Captured on the Clock Edge:** When the clock edge arrives, the destination flip-flop captures the signal D. $t_{capture} = t_{arrival} + t_{setup}$
6. **Waits to be Replaced:** After being captured, 'D' stays as the output of the destination flip-flop until it is replaced by a new signal. $t_{cycle} = t_{CQ} + t_{comb} + t_{setup} + t_{hold}$



This cycle repeats every clock cycle, ensuring the continuous flow of data through the digital system.

For convinience, the circuit diagram:

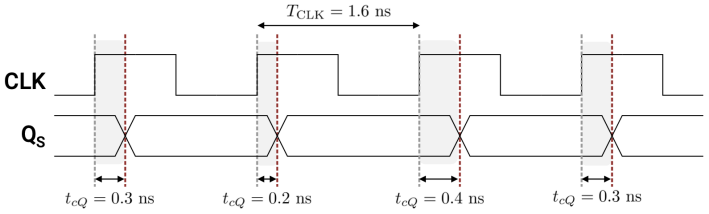


15.4.1 Born as Q of the Source Flip-Flop (FF)

The signal D starts its journey as the output (Q) of a source flip-flop. The transition from the input D to the output Q involves a clock-to-Q delay, t_{CQ}, which can vary between $t_{CQ,min}$ and $t_{CQ,max}$.

> The output Q changes after the clock-to-Q delay.
>
> The clock-to-Q delay can range from $t_{CQ,min}$ to $t_{CQ,max}$.

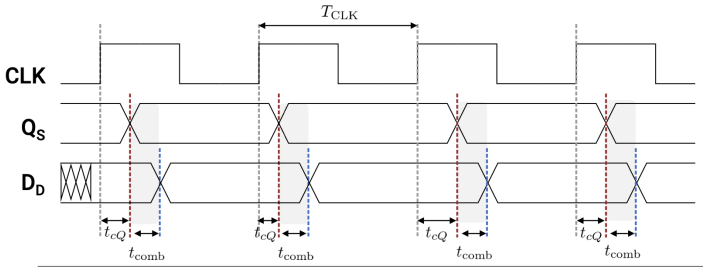
Updating the output of the source FF (Q_S) takes some variable clock-to-Q delay.



15.4.2 Journey through Combinational Logic

After being emitted from the source flip-flop, 'D' travels through various combinational logic components.
This part of the journey involves processing and transformation of the signal.
Time taken: $t_{CQ} + t_{comb}$.

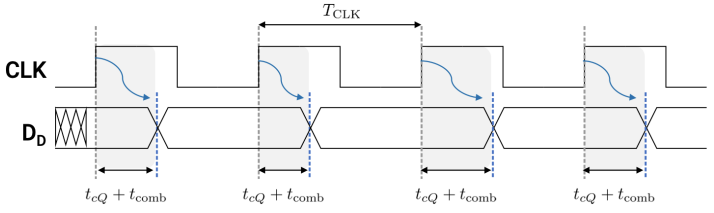
Starts as Q_S- D starts the journey as Q_S.
Ends as D_D- D ends the journey as D_D, the input of the destination flip-flop.
Duration- The journey duration is t_{comb}.



15.4.3 Arrives at the Destination FF

The signal 'D' reaches the input of the destination flip-flop. The arrival time of the signal can be represented as $t_{arrival} = t_{CQ} + t_{comb}$.

> **Arrival Time**: $t_{arrival} = t_{CQ} + t_{comb}$

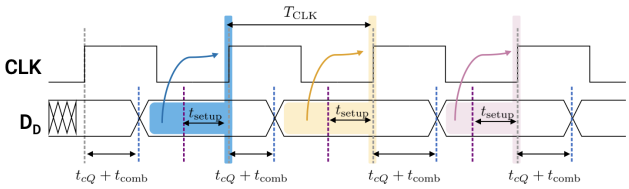


15.4.4 Waits to be Captured on Clock Edge

At the destination flip-flop, 'D' waits for the next clock edge. This waiting period is crucial as the signal is prepared to be synchronized with the system's clock.

Stable During Wait: During the "wait", D_D should be stable.

Minimum Wait Time: The wait must be at least as long as the setup time t_{setup} for the destination FF to capture D correctly.

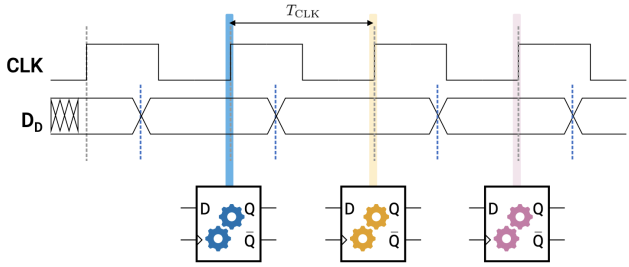


15.4.5 Gets Captured on the Clock Edge

When the clock edge arrives, the destination flip-flop captures the signal D. The capture time can be calculated as $t_{capture} = t_{arrival} + t_{setup}$.

Capture Time: $t_{capture} = t_{arrival} + t_{setup}$

Process: The destination FF takes the value on D_D and works on memorizing it and passing it to the output Q_D.



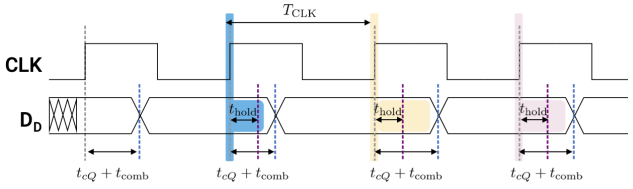
15.4.6 Waits to be Replaced

After being captured, 'D' stays as the output of the destination flip-flop until it is replaced by a new signal in the next clock cycle. The total cycle time is given by $t_{cycle} = t_{CQ} + t_{comb} + t_{setup} + t_{hold}$.

Hold Time: The "wait" must be at least as long as the hold time t_{hold} for the destination FF to capture D_D correctly.

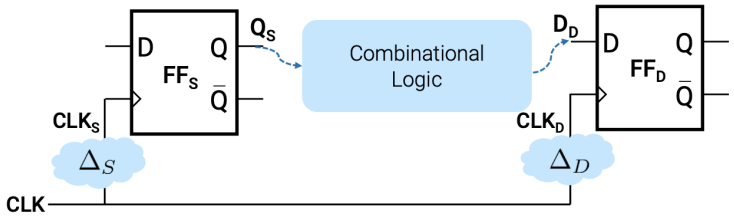
Cycle Time: $t_{cycle} = t_{CQ} + t_{comb} + t_{setup} + t_{hold}$

This cycle repeats every clock cycle, ensuring the continuous flow of data through the digital system.



15.5 Clock Skew

In real circuits, the clock signal may not arrive at all FFs at the same time because of the variations in the delay of the wires that carry it.
This variation in the arrival time of the clock signal between different flip-flops is called clock skew, and **it can lead to timing errors and reduced performance in a circuit.**



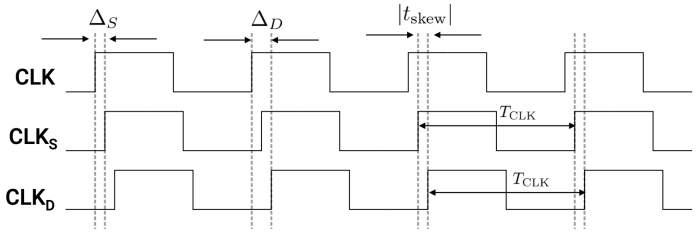
Clock skew is defined as the difference in arrival times of the clock signal at different flip-flops:

$$t_{skew} = \Delta_D - \Delta_S$$

with Δ_i being the arrival time of the clock signal at FF_i.

15.5.1 Clock Delays

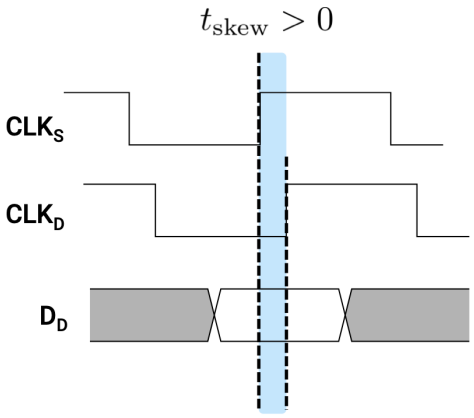
Clock arriving to the flip-flops is delayed with respect to the clock source



Positive Clock Skew

Positive clock skew between CLK_D and CLK_S means that the rising edge of CLK_D is **delayed (late)** with respect to the rising edge of CLK_S.
$t_{\text{skew}} = \Delta_D - \Delta_S > 0$

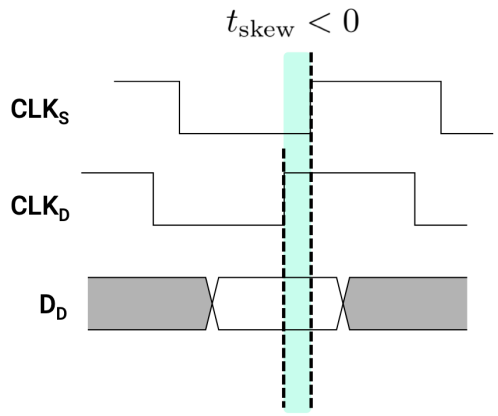
Additional impact of positive clock skew on timing and circuit performance.



Negative Clock Skew

Negative clock skew between CLK_D and CLK_S means that the rising edge of CLK_D is **advanced (early)** with respect to the rising edge of CLK_S. $t_{\text{skew}} = \Delta_D - \Delta_S < 0$

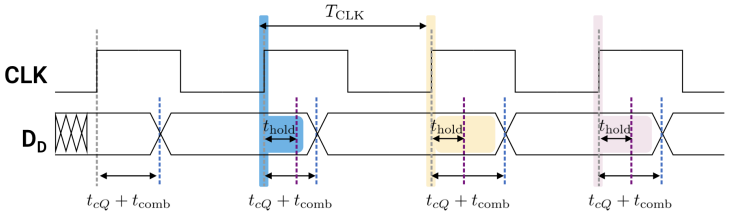
Additional impact of negative clock skew on timing and circuit performance.



15.6 Life of D: Lasting One Clock Cycle (with Clock Skew)

The signal 'D' undergoes a lifecycle within a digital system that repeats every clock cycle. This lifecycle can be described as follows: *Personal Remark: Basically what you should retain from this section*

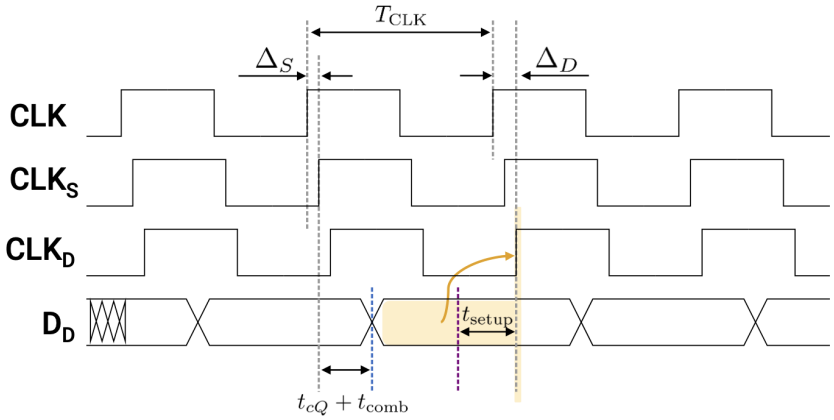
1. **Born as Q of the Source Flip-Flop (FF):** The signal D starts as the output (Q) of a source flip-flop.
2. **Journey through Combinational Logic:** 'D' travels through combinational logic. $t_{CQ} + t_{comb}$
3. **Arrives at the Destination FF:** 'D' reaches the destination flip-flop input. $t_{arrival} = t_{CQ} + t_{comb}$
4. **Waits to be Captured on Clock Edge:** 'D' waits for the next clock edge. **Clock skew affects this wait time.**
5. **Gets Captured on the Clock Edge:** The destination flip-flop captures 'D'. $t_{capture} = t_{arrival} + t_{setup} \pm \boldsymbol{t_{skew}}$
6. **Waits to be Replaced:** 'D' stays as the output until replaced by a new signal. $t_{cycle} = t_{CQ} + t_{comb} + t_{setup} + t_{hold} \pm \boldsymbol{t_{skew}}$



This cycle, adjusted for clock skew, ensures the continuous flow of data through the digital system but may affect timing and performance.

15.7 Timing Constraints in the Presence of Clock Skew

To ensure proper timing in the presence of clock skew, both setup-time and hold-time constraints must be met.



15.7.1 Setup-Time Constraint

The setup-time constraint must satisfy:

$$T_{CLK} + \Delta_D - (\Delta_S + t_{CQ} + t_{comb}) \geq t_{setup}$$

Rewritten for the worst-case scenario:

$$t_{CQ,max} + t_{comb,max} + t_{setup} - (\Delta_D - \Delta_S) \leq T_{CLK}$$

15.7.2 Hold-Time Constraint

The hold-time constraint must satisfy:

$$t_{CQ} + t_{comb} + \Delta_S - \Delta_D \geq t_{hold}$$

Rewritten for the worst-case scenario:

$$t_{CQ,min} + t_{comb,min} - (\Delta_D - \Delta_S) \geq t_{hold}$$

15.7.3 Implications of Clock Skew on the Max Operating Frequency

- **Recall:**

$$t_{CQ,max} + t_{comb,max} + t_{setup} - (\Delta_D - \Delta_S) \leq T_{CLK} = \frac{1}{f_{CLK}}$$

- **Substituting** $t_{skew} = \Delta_D - \Delta_S$:

$$t_{CQ,max} + t_{comb,max} + t_{setup} - t_{skew} \leq T_{CLK} = \frac{1}{f_{CLK}}$$

$$t_{CQ,max} + t_{comb,max} + t_{setup} - t_{skew} = \frac{1}{f_{max,skew=0}} - t_{skew} = \frac{1}{f_{max}}$$

- **Positive skew improves the max frequency**:

$$\frac{1}{f_{max,skew=0}} > \frac{1}{f_{max}} \Rightarrow f_{max} > f_{max,skew=0}$$

- **Negative skew worsens the max frequency**:

$$\frac{1}{f_{max,skew=0}} < \frac{1}{f_{max}} \Rightarrow f_{max} < f_{max,skew=0}$$

What ifs...

- **Q1: What if the circuit cannot work correctly at the desired frequency due to unsatisfied setup-time constraints?**

$$t_{CQ,max} + t_{comb,max} + t_{setup} - t_{skew} \leq T_{CLK} = \frac{1}{f_{CLK}}$$

A1: Redesign the combinational logic to reduce the combinational path delay; try to increase the clock skew. In practice, we let computer-aided design tools do this for us.

- **Implications of Clock Skew on the Max Operating Frequency**

Recall:

$$t_{CQ,max} + t_{comb,max} + t_{setup} - (\Delta_D - \Delta_S) \leq T_{CLK} = \frac{1}{f_{CLK}}$$

Substituting $t_{skew} = \Delta_D - \Delta_S$:

$$t_{CQ,max} + t_{comb,max} + t_{setup} - t_{skew} \leq T_{CLK} = \frac{1}{f_{CLK}}$$

$$t_{CQ,max} + t_{comb,max} + t_{setup} - t_{skew} = \frac{1}{f_{max,skew=0}} - t_{skew} = \frac{1}{f_{max}}$$

Positive skew improves the max frequency:

$$\frac{1}{f_{max,skew=0}} > \frac{1}{f_{max}} \Rightarrow f_{max} > f_{max,skew=0}$$

Negative skew worsens the max frequency:

$$\frac{1}{f_{max,skew=0}} < \frac{1}{f_{max}} \Rightarrow f_{max} < f_{max,skew=0}$$

- **Q2: What if the circuit cannot work correctly due to unsatisfied hold-time constraints?**

$$t_{CQ,min} + t_{comb,min} - t_{skew} \geq t_{hold}$$

A2: Insert buffers on the short combinational paths to increase their delay; try to reduce the clock skew. In practice, we let computer-aided design tools do this for us.

- **Implications of Clock Skew on Meeting the Hold-Time Constraints**

Recall:

$$t_{CQ,min} + t_{comb,min} - (\Delta_D - \Delta_S) \geq t_{hold}$$

Substituting $t_{skew} = \Delta_D - \Delta_S$:

$$t_{CQ,min} + t_{comb,min} - t_{skew} \geq t_{hold}$$

Basically...

Positive skew makes meeting hold-time constraints more difficult

Negative clock skew makes meeting hold-time constraints easier

15.7.4 Timing Analysis with a Clock Skew

Algorithm for Finding $f_{\max}$

Personal Remark: ***Important!!***

1 - Identify all Q-to-D paths

2 - For every such path

(a) Compute its longest combinational delay

(b) Find the shortest clock period that satisfies the path's setup-time constraint

* *Remember to include clock delays in the expressions*

3 - Find the shortest clock period T_{CLK} that satisfies the setup-time constraints of all those paths

4 - Compute $f_{\max} = \frac{1}{T_{\text{CLK}}}$

15.7.5 Example Analysis with Clock Skew

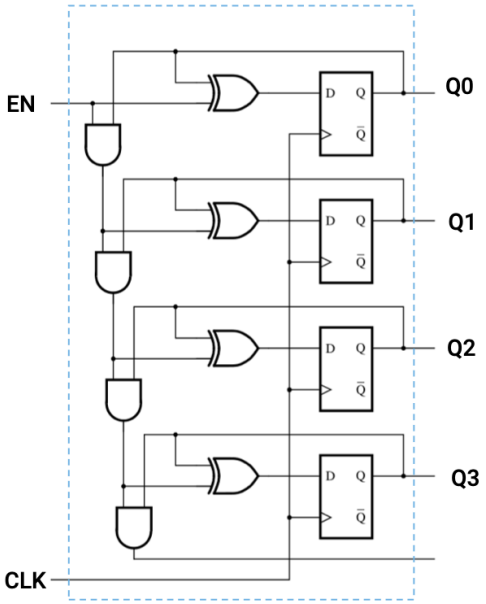
FF timing parameters

$$t_{\text{setup}} = 0.6\,\text{ns};\ t_{\text{hold}} = 0.4\,\text{ns}$$

$$0.8\,\text{ns} \leq t_{cQ} \leq 1\,\text{ns}$$

and gate delays $t_{\text{AND}} = 1.2\,\text{ns}$ and $t_{\text{XOR}} = 1.3\,\text{ns}$
Assume clock delays:

$$\Delta_0 = 0 \quad \Delta_1 = 0 \quad \Delta_2 = 0 \quad \Delta_3 = 2\,\text{ns}$$



(a) Find the max operating frequency $f_{\max}$
(b) Are there hold-time violations in this circuit?

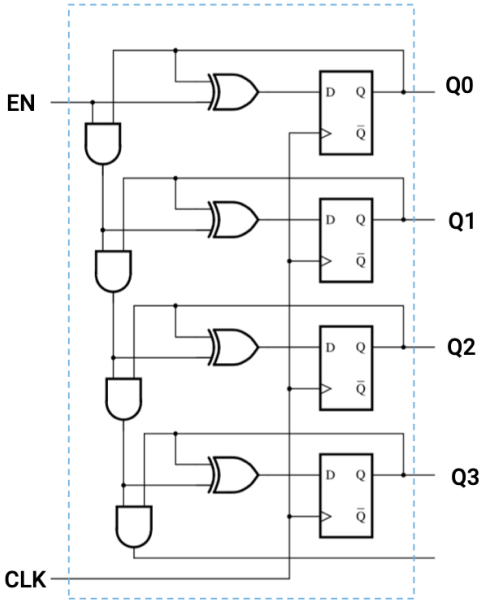
Note: For simplicity, we will assume that EN is available at the rising clock edge.

Step 1 - Identify all Q-to-D paths

We already computed delays of all paths (see earlier example); let us focus on only the paths affected by clock delays

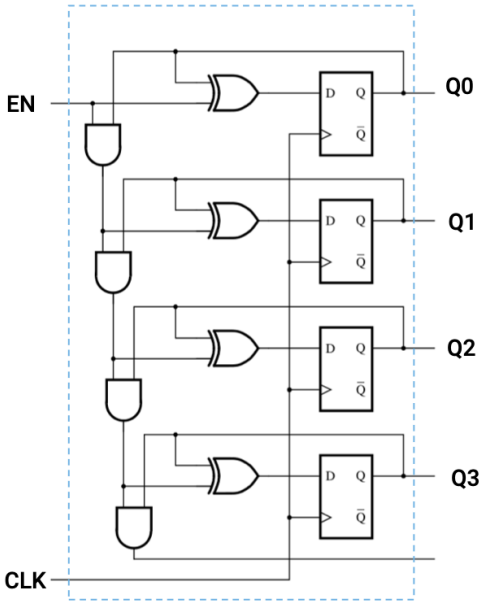
As clock at FF3 is delayed, the Q-to-D paths starting or ending at FF3 are the only ones affected

- Q0 to D3
- Q1 to D3
- Q2 to D3
- Q3 to D3



Step 2a - For all paths, find the longest combinational path delay

- Q0 to D3:
  $t_{\text{comb}} = 3 \times t_{\text{AND}} + t_{\text{XOR}} = 3 \times 1.2 + 1.3 = 4.9\,\text{ns}$
- Q1 to D3:
  $t_{\text{comb}} = 2 \times t_{\text{AND}} + t_{\text{XOR}} = 2 \times 1.2 + 1.3 = 3.7\,\text{ns}$
- Q2 to D3:
  $t_{\text{comb}} = t_{\text{AND}} + t_{\text{XOR}} = 1.2 + 1.3 = 2.5\,\text{ns}$
- Q3 to D3:
  $t_{\text{comb}} = t_{\text{XOR}} = 1.3\,\text{ns}$



Step 2b - Find the shortest clock period that satisfies the path's setup-time constraint

Remember to include clock skew in the expressions

$$t_{cQ,\max} + t_{\text{comb, max}} + t_{\text{setup}} - (\Delta_D - \Delta_S) \leq T_{\text{CLK}}$$

- Q0 to D3: $1 + 4.9 + 0.6 - (2 - 0) = 4.5\,\text{ns}$
- Q1 to D3: $1 + 3.7 + 0.6 - (2 - 0) = 3.3\,\text{ns}$
- Q2 to D3: $1 + 2.5 + 0.6 - (2 - 0) = 2.1\,\text{ns}$
- Q3 to D3: $1 + 1.3 + 0.6 - (2 - 2) = 2.9\,\text{ns}$

Step 3 - Find the shortest period that satisfies all paths

Clock period that satisfies all paths involving FF3:

$$\max(4.5, 3.3, 2.1, 2.9) = 4.5\,\text{ns}$$

Clock period that satisfies other paths is determined by the path with the longest combinational delay:

- Q0 to D2: $t_{\text{comb}} = 2 \times t_{\text{AND}} + t_{\text{XOR}} = 2 \times 1.2 + 1.3 = 3.7\,\text{ns}$

$$Q0\,\text{to}\,D2: \quad 1 + 3.7 + 0.6 + (0 - 0) = 5.3\,\text{ns}$$

Finally, the shortest clock period that satisfies all paths

$$T_{\text{CLK}} = \max(4.5, 5.3) = 5.3\,\text{ns}$$

Step 4 - Compute the max operating frequency

Thus the max operating frequency is:

$$f_{\max} = \frac{1}{T_{CLK}\,\text{ns}} \approx 188\,\text{MHz}$$

Algorithm for Checking Hold-Time Violations

Personal Remark: Again, ***Important!!***

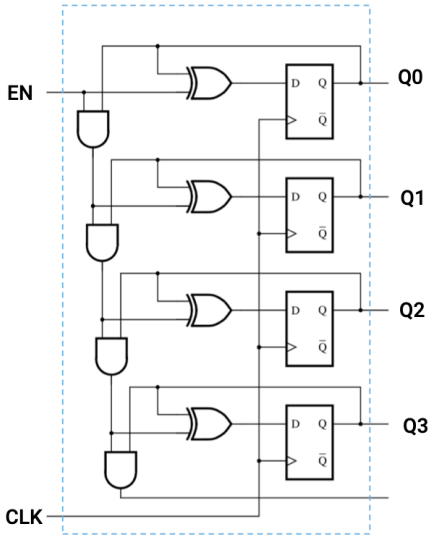
1 Identify all Q-to-D paths

2 For every such path

- Compute its shortest combinational delay
- Verify if hold-time constraint is satisfied
  - *Remember to include clock delays in the expressions*

Step 1 - Identify all Q-to-D paths

We already verified hold-time constraints for all paths (see earlier example);
Therefore, we now focus only on the paths involving FF3, because clock at FF3 is delayed.



Step 2a - Find min combinational path delay

- Q0 to D3:
$t_{\text{comb}} = 3 \times t_{\text{AND}} + t_{\text{XOR}} = 3 \times 1.2 + 1.3 = 4.9\,\text{ns}$
- Q1 to D3:
$t_{\text{comb}} = 2 \times t_{\text{AND}} + t_{\text{XOR}} = 2 \times 1.2 + 1.3 = 3.7\,\text{ns}$
- Q2 to D3:
$t_{\text{comb}} = t_{\text{AND}} + t_{\text{XOR}} = 1.2 + 1.3 = 2.5\,\text{ns}$
- Q3 to D3:
$t_{\text{comb}} = t_{\text{XOR}} = 1.3\,\text{ns}$

Step 2b - Given the clock delay to the source FF, verify if hold-time constraint is satisfied

$$t_{cQ,\min} + t_{\text{comb},\min} - (\Delta_D - \Delta_S) \geq t_{\text{hold}}$$

- Q0 to D3: $0.8 + 4.7 - (2 - 0) = 3.5\,\text{ns} > 0.4\,\text{ns}$
- Q1 to D3: $0.8 + 3.7 - (2 - 0) = 2.5\,\text{ns} > 0.4\,\text{ns}$
- Q2 to D3: $0.8 + 2.5 - (2 - 0) = 1.3\,\text{ns} > 0.4\,\text{ns}$
- Q3 to D3: $0.8 + 1.3 - (2 - 2) = 2.1\,\text{ns} > 0.4\,\text{ns}$

Conclusion
All relevant paths satisfy hold-time constraints.

Chapter 16

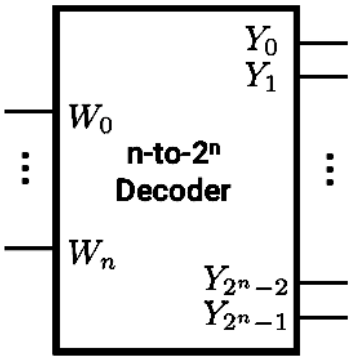
Digital logic and Verilog, Part XI (W10.2)

16.1 $n - to - 2^n$ Binary Decoders

A **decoder** is a logic circuit that takes an n-bit binary input and produces a 2^n-bit output, where only one bit is logic 1 and the rest are logic 0.

Decoding the input: The output bit at the *index* corresponding to the decimal equivalent m of the n-bit input is set to logic 1; all other output bits are logic 0.

Decoder outputs are *one-hot encoded*, meaning only one bit is set to 1 ("hot").



Quick Example for $n = 2$:

- Input: 00 (binary) → Output: 0001 (the 1 is in the 0th position)
- Input: 01 (binary) → Output: 0010 (the 1 is in the 1st position)
- Input: 10 (binary) → Output: 0100 (the 1 is in the 2nd position)
- Input: 11 (binary) → Output: 1000 (the 1 is in the 3rd position)

Example - 2-to-4 Binary Decoder

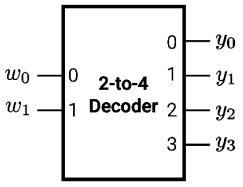
A (binary) decoder with $n = 2$ inputs and $2^n = 4$ outputs

The input $(w_1 w_0)$ corresponds to binary numbers

- $(00)_2 = (0)_{10}$; $(01)_2 = (1)_{10}$; $(10)_2 = (2)_{10}$; $(11)_2 = (3)_{10}$

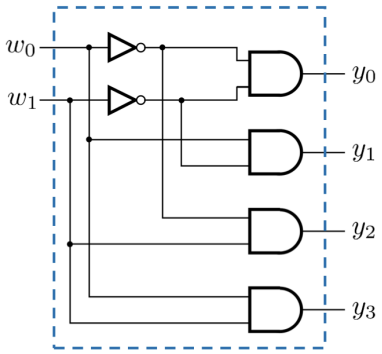
Only one bit in the output vector (y_3, y_2, y_1, y_0) is set

- $(w_1 w_0) = (00)_2 = (0)_{10} \rightarrow y_0 = 1$
- $(w_1 w_0) = (01)_2 = (1)_{10} \rightarrow y_1 = 1$
- $(w_1 w_0) = (10)_2 = (2)_{10} \rightarrow y_2 = 1$
- $(w_1 w_0) = (11)_2 = (3)_{10} \rightarrow y_3 = 1$



w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$y_0 = \overline{w_1 w_0}$$
$$y_1 = \overline{w_1} w_0$$
$$y_2 = w_1 \overline{w_0}$$
$$y_3 = w_1 w_0$$

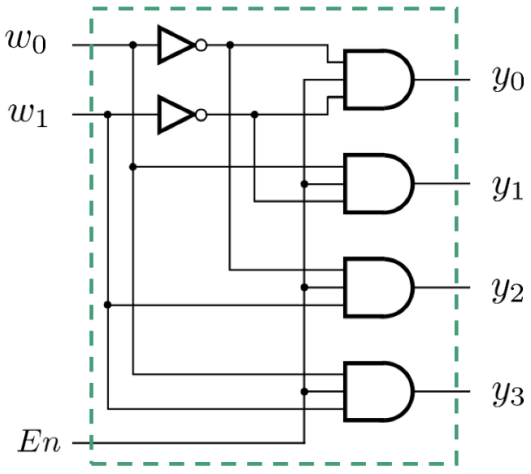


2-to-4 Binary Decoder with an Enable

If the decoder is disabled, $En = 0$, then no output will be set

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	X	X	0	0	0	0

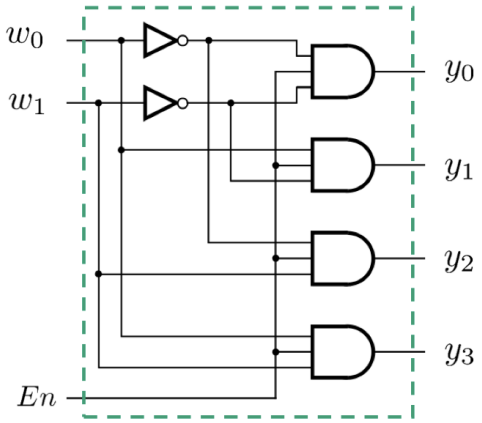
$$y_0 = En \cdot \overline{w_1} \cdot \overline{w_0}$$
$$y_1 = En \cdot \overline{w_1} \cdot w_0$$
$$y_2 = En \cdot w_1 \cdot \overline{w_0}$$
$$y_3 = En \cdot w_1 \cdot w_0$$



2-to-4 Binary Decoder with an Enable (Verilog)

Personal Remark: Obviously, like in Java, the '?' operator is used for one line conditional statements.

```
module two_to_four_dec(W, En, Y);
  input [1:0] W;
  input En;
  output [3:0] Y;
  assign Y[0] = En ? (W == 2'b00)
      : 0;
  assign Y[1] = En ? (W == 2'b01)
      : 0;
  assign Y[2] = En ? (W == 2'b10)
      : 0;
  assign Y[3] = En ? (W == 2'b11)
      : 0;
endmodule
```

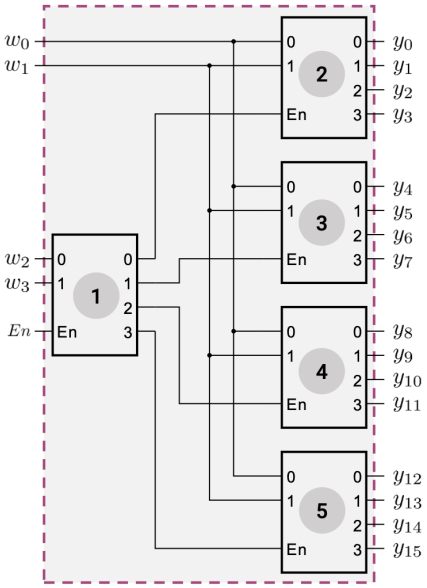


4-to-16 Decoder with an Enable

A 4-to-16 decoder is constructed using a 2-to-4 decoder (DEC1) at the root and four 2-to-4 decoders (DEC2, DEC3, DEC4, DEC5) in the branches:

- Inputs: 4 binary digits ($w_3w_2w_1w_0$) and an enable signal (En).
- w_3w_2 are fed into DEC1.
- DEC1 produces four outputs; each output is fed into a 2-to-4 decoder.
- DEC1 outputs enable DEC2, DEC3, DEC4, DEC5.
- w_1w_0 are fed into the enabled branch decoder.

$$(w_3w_2w_1w_0, En) = (1011, 1)$$
$$OUT_{\text{DEC1[2]}} = 1,$$
$$En_{\text{DEC4}} = 1,$$
$$OUT_{\text{DEC4[3]}} = y_{11} = 1$$



16.2 Memory

Personal Remark: Didn't think the history of memory was worth mentioning (Slides 16-17, Chapter XI) Memory types may vary in terms of:

- Capacity: how many bytes can be stored
- Density: how many bits per unit area (silicon)
- Speed: how much time it takes to read from it or write to it

- Writable or read-only
- Volatile or not: loses contents once the power supply is removed or not

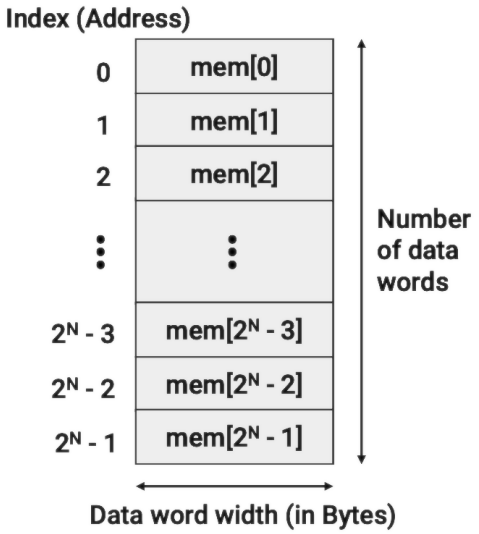
(Out of Scope and Personal) For example, you might be aware of the difference between RAM and ROM. Random Access Memory (RAM) is volatile, meaning it loses its data when the power is turned off, and is often used for fast runtime access. On the other hand, Read-Only Memory (ROM) is non-volatile, meaning it retains its data even when the power is turned off, and is typically used to store firmware or permanent software that does not need to be modified.

16.2.1 Abstract View of Memory

A two-dimensional array of 1-bit memory elements (e.g., DFFs, latches, capacitors)

One memory element `mem[i]`, where `i` is the row index in a two-dimensional array of bits, includes all bits in that row and is referred to as one data word.

In memory terminology, instead of saying the **index**, we say **the address** of the data word



16.2.2 Word Sizes

Data word sizes and common terminology:

- 8 bits = one byte (**Byte, B**)
- 16 bits (also called half-word)
- 32 bits, 64 bits

Memory **capacity** is the total number of data bytes it can store (*important*)

- (**Number of words**) × (**word width in bytes**)

Number of words = 2^N, where **N** is the number of bits of the address

16.2.3 Memory Capacity

Recall: memory **capacity** is the total number of data bytes it can store

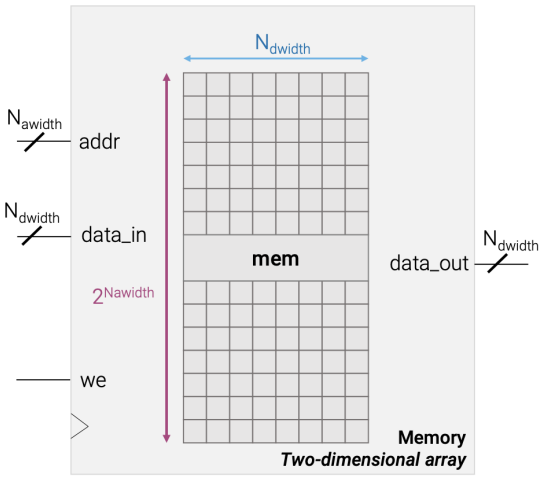
Memory units (Wiki link)

- Kilobyte (**KiB**) = 2^{10} B
- Megabyte (**MiB**) = 2^{20} B
- Gigabyte (**GiB**) = 2^{30} B
- Terabyte (**TiB**) = 2^{40} B

16.2.4 Access Protocol

Many variants exist, differing in control signals, their polarity, number of inputs and outputs, width of inputs and outputs, etc.
In this lecture, we consider the following simple memory access protocol:

> **Synchronous write**: on the rising clock edge, if write enable (we) is active, memory **write** takes place: `mem[addr] = data_in`
>
> **Asynchronous read**: at all times, memory **read** takes place: `data_out = mem[addr]`

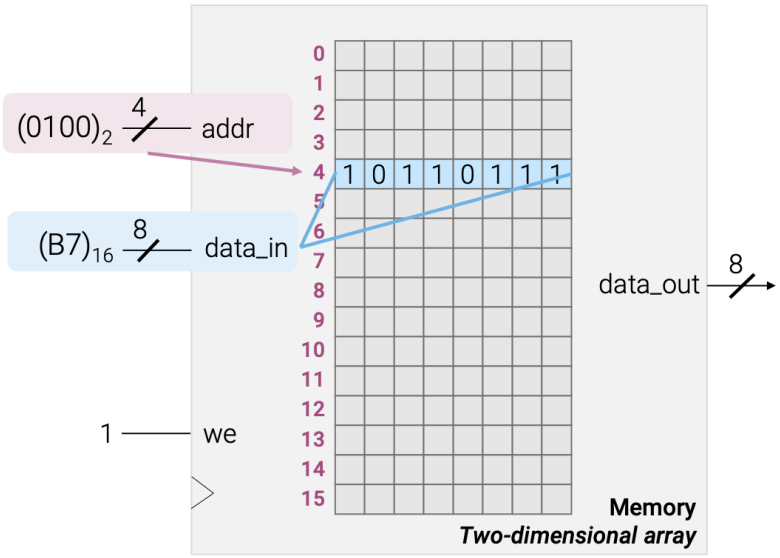


16.2.5 Example - 16 x 8 Memory

Write

- $N_{awidth} = log_2(16) = 4$
- $N_{dwidth} = 8$
- addr $= (0100)_2 = (4)_{10}$
- data_in $= (\mathbf{B7})_{16}$
- we $= 1$

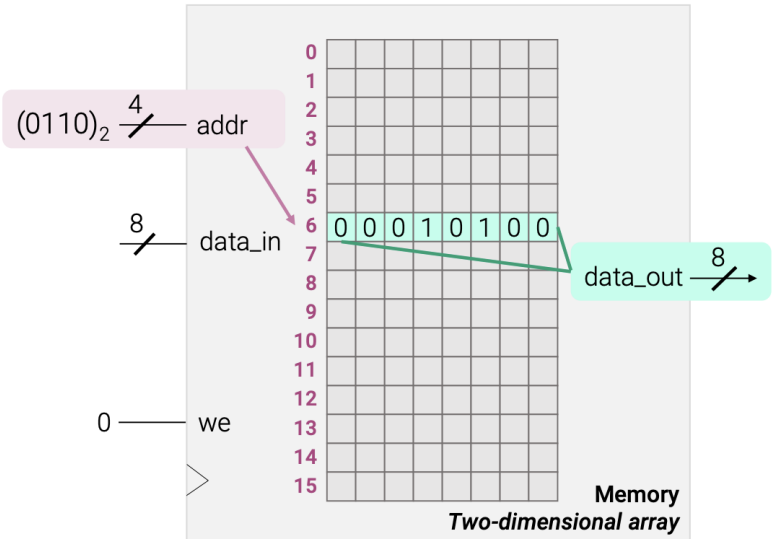
Memory **read** takes place and **old** mem[4] appears at data_out port
Memory **write** takes place and data_in overwrites mem[4]; **new** value of mem[4] $= (\mathbf{B7})_{16}$



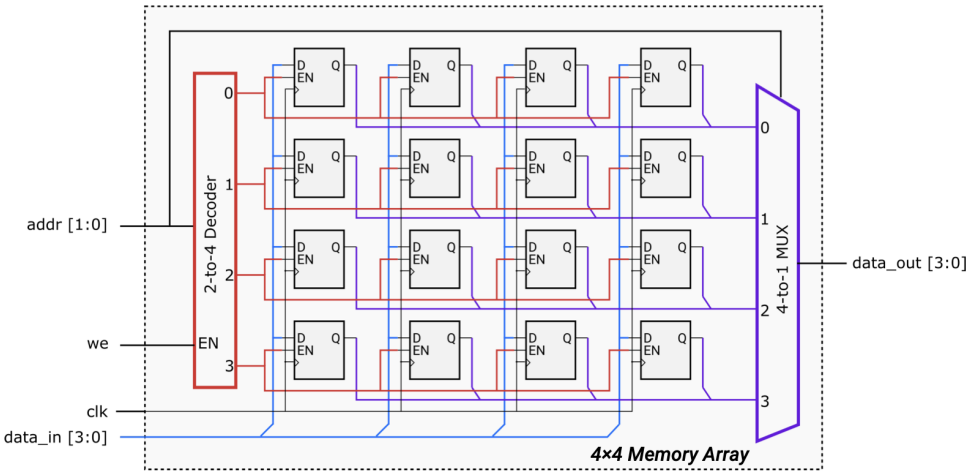
Read

- $N_{awidth} = log_2(16) = 4$
- $N_{dwidth} = 8$
- addr $= (0110)_2 = (6)_{10}$
- we $= 0$

Memory **read** takes place and mem[6] appears at data_out port: $data\_out = mem[6] = (14)_{16}$
Memory **write** is not taking place



16.2.6 Memory as an Array of DFFs



- **Addressing:** 2-bit address (`addr[1:0]`) selects one of 4 rows via 2-to-4 decoder.
- **Write Enable:** `we` enables writing when active.
- **Clock:** `clk` synchronizes writing on rising edge.
- **Data Input:** `data_in[3:0]` provides 4-bit data to write.
- **Memory Cells:** Flip-flops store data; enabled by decoder output.
- **Data Output:** 4-to-1 MUX selects data from addressed row to `data_out[3:0]`.
- The outputs of the address decoder, which enable one entire row of the memory array, are called word lines
- The wires that carry data (input, output, or sometimes a shared in/out bus) are called bit lines

16.3 Verilog Part 5.

16.3.1 Arrays

Two dimensional arrays

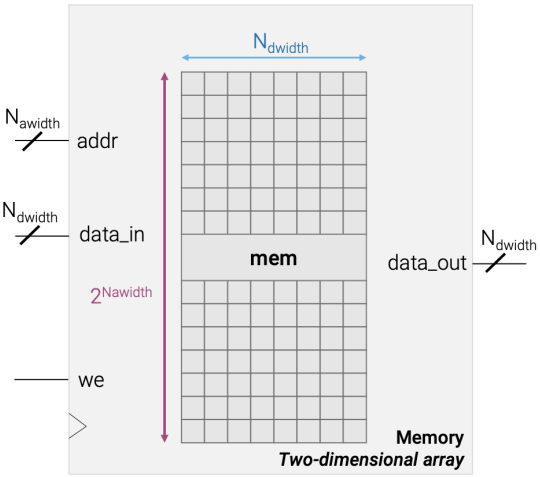
In Verilog, we can declare two-dimensional arrays An array **mem** of **Ndw** data words, where each word has **Ndb** bits, is declared as :

```
reg [Ndb-1:0] mem [Ndw-1:0];
```

16.3.2 Memory Module

A memory module can be defined as such :

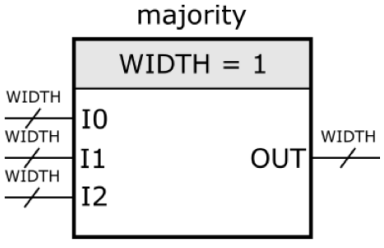
```
module mem (addr, data_in, we, clk
    , data_out);
  parameter Nawidth = 2; //
      arbitrary default
  parameter Ndwidth = 4; //
      arbitrary default
  input we, clk; // write enable
      and clock
  input [Nawidth-1:0] addr;
  input [Ndwidth-1:0] data_in;
  output [Ndwidth-1:0] data_out;

  // memory array
  reg [Ndwidth-1:0] mem [2**
      Nawidth-1:0];
  always @(posedge clk) begin
  if (we) begin
    mem[addr] <= data_in;
  end
  end
  assign data_out = mem[addr];
endmodule
```



16.3.3 Parameterized Verilog Modules

Verilog parameters can be put to good use, to allow instantiated modules to accept inputs and outputs of arbitrary width
Basically allows to create a module that can be used for different word sizes.

```
module majority(I0, I1, I2, OUT);
  parameter WIDTH = 1; // default
      parameter value
  input [WIDTH-1:0] I0, I1, I2;
  output [WIDTH-1:0] OUT;
  assign OUT = (I0 & I1) | (I0 &
      I2) | (I1 & I2);
endmodule
```



Allowing for example, to instantiate using parameter 4, 8 (1 if not specified) :

```
// Instantiating majority module;
// Overriding the default value of WIDTH = 1;
// New value: WIDTH = 4;
majority #(.WIDTH(4)) U1 (.I0(A), .I1(B), .I2(C), .OUT(D));
// Instantiating majority module;
// Overriding the default value of WIDTH = 1;
// New value: WIDTH = 8;
majority #(.WIDTH(8)) U2 (.I0(X), .I1(Y), .I2(Z), .OUT(W));
```

16.3.4 Verilog Conditional Operator

Personal Remark: Like in java

```
A ? B : C
```

- Conditional operator ?: select one of the two alternate expressions (B, C) depending on the value of a logical expression (A)
- If the logical expression (A) is true, it returns the first alternative (B)
- Otherwise, it returns the second alternative (C)

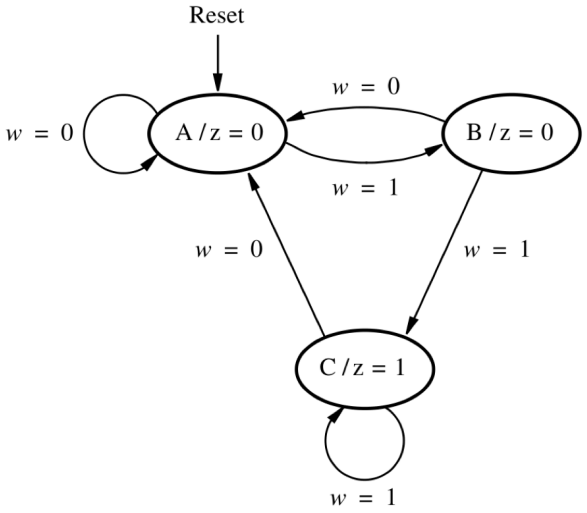
Chapter 17

Digital Logic and Verilog - Examples - FSM (W10.2)

This is a small collection of examples of Finite State Machines (FSMs) and their implementation in Verilog.
Personal Remark: Knowing what a Moore and Mealy machine is, is important to understand this part of the course.

17.1 Moore FSM, with Verilog Implementation

Description of the Machine



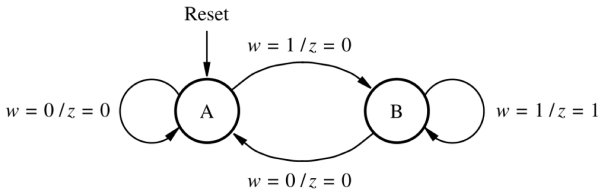
Current State S	Next state S_{next}		Output
S	w		z
	0	1	
A	A	B	0
B	A	C	0
C	A	C	1

- States are A, B, Z, each represented by at least 2 bits.
- Input is 1-bit input w
- Output is 1-bit z
- Reset input is Asynchronous, power-on reset, active high

```
module fsm_Moore (input w, CLK, Reset, output reg z);
reg [1:0] S_next, S;
parameter A = 2'b00, B = 2'b01, C = 2'b10; // states
// Next-state logic
always @ (*) begin
  case (S)
  A: if (w == 0) S_next = A;
    else S_next = B;
  B: if (w == 0) S_next = A;
    else S_next = C;
  C: if (w == 0) S_next = A;
    else S_next = C;
  default: S_next = 2'bxx;
  endcase
end
// State memory
always @ (posedge CLK or posedge Reset) begin
  if (Reset == 1) S <= A;
  else S <= S_next;
end
// Output logic
always @ (*) begin
z = (S == C);
end
endmodule
```

17.2 Mealy FSM, with Verilog Implementation

Description of the Machine



Current State S	Input	Next state S_{next}	Output
S	w	S_{next}	z
A	0	A	0
A	1	B	0
B	0	A	0
B	1	B	1

- States are A, B, represented by at least 1 bit.
- Input is 1-bit input w
- Output is 1-bit z
- Asynchronous, power-on reset, active high

```
module fsm_Mealy (input w, CLK, Reset, output reg z);
  reg S_next, S;
  parameter A = 1'b0, B = 1'b1; // states
  // Next-state logic
  always @ (*) begin
    case (S)
    A: if (w == 0) S_next = A;
      else S_next = B;
    B: if (w == 0) S_next = A;
      else S_next = B;
    default: S_next = 1'bx;
    endcase
  end
  // State memory
  always @ (posedge CLK or posedge Reset) begin
    if (Reset == 1) S <= A;
    else S <= S_next;
  end
  // Output logic
  always @ (*) begin
    z = 0;
    if ((S == B) && (w == 1)) z = 1'b1;
  end
endmodule
```

Chapter 18

Digital Logic and Verilog (Part XII) (W11.1)

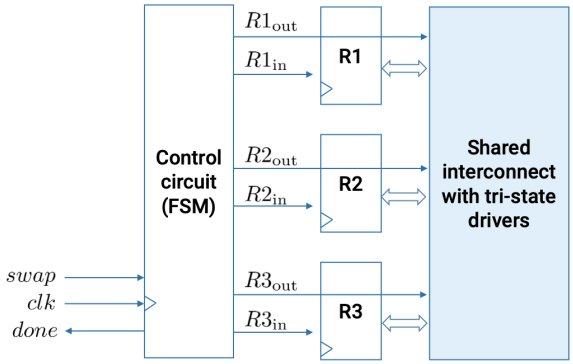
This part of the course might seem a littttttle bit more advanced, as it applies the last few chapters at a higher level.
You do need to recall two things, what a Tri-State Driver is (only allows current to flow when the enable signal is active) and what a Bus with Tri-State drivers is, basically allows multiple modules(things) to be connected to a single "bus", again, current of the bus only flows when the enable signal is active.
Also, don't expect understanding the thinking behind going from an Idea to an Actual Verilog program without having done the exercices

18.1 Swapping of two Registers

Here, we will be considering a system that will allow to swap two registers using one temporary register.

Our first attempt will be using a bus with tri-state drivers, and the second attempt will be using a bus with mux.

18.1.1 Using a Bus with Tri-State Drivers (Verilog)



We have the following FSM States:

```
IDLE: No swapping
R2TOR3: First copy
R1TOR2: Second copy
R3TOR1: Third copy
```

With the following algorithm for swapping:

```
1 - Swap starts → Copy data from R2 to R3
2 - Copy data from R1 to R2
3 - Copy contents of R3 to R1 → Swap ends
```

And an asynchronous power-on reset

Translating this in Verilog gives us,

```
module control (clk, reset, swap, R1in, R1out, R2in, R2out, R3in, R3out,
    done);
  input clk, reset, swap;
  output reg R1in, R1out, R2in, R2out, R3in, R3out, done;
  parameter IDLE = 2'b00, R2TOR3 = 2'b01, R1TOR2 = 2'b10, R3TOR1 = 2'b11;
  reg [1:0] S_next, S;
  // Next-state logic
  always @ (*) begin
    S_next = IDLE; // default
    case (S)
    IDLE: if (swap) S_next = R2TOR3;
      else S_next = IDLE;
      R2TOR3: S_next = R1TOR2;
      R1TOR2: S_next = R3TOR1;
      R3TOR1: S_next = IDLE;
      default: S_next = IDLE;
  endcase
  end
  // State memory
  always @ (posedge clk or posedge reset) begin
      if (reset) S <= IDLE;
      else S <= S_next;
    end
    // Output logic
    always @ (*) begin
      R1in = (S == R3TOR1);
      R1out = (S == R1TOR2);
      R2in = (S == R1TOR2);
      R2out = (S == R2TOR3);
      R3in = (S == R2TOR3);
      R3out = (S == R3TOR1);
      done = (S == R3TOR1);
  end
endmodule
```

We may also define :
An n-bit register module:

```
module regn (D, clk, reset, en,
    Q);
  parameter n = 8; // default
  input [n-1:0] D;
  input clk, reset, en;
  output reg [n-1:0] Q;
  always @ (posedge clk or
     posedge reset) begin
  if (reset) Q <= 0;
  else if (en) Q <= D;
  end
endmodule
```

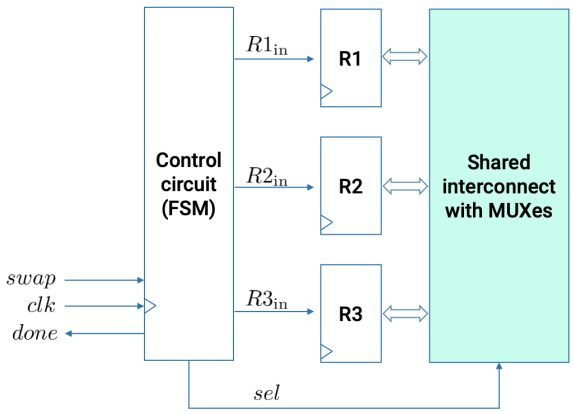
A Tri-State driver bus module:

```
module bustri (w, en, f);
  parameter n = 8; // default
  input [n-1:0] w;
  input en;
  output [n-1:0] f;
  assign f = en ? w : 'bz;
endmodule
```

Now using the FSM controller, the bus and the reg, we can defined the swapping module as such:

```
module regswap (clk, reset, swap);
  parameter width = 8;
  input clk, reset, swap;
  wire wR1in, wR1out, wR2in, wR2out, wR3in, wR3out, wdone;
  wire [width-1:0] wR1, wR2, wR3, wBus;

  // Instantiate controller module
  control controller_module ( .clk (clk), .reset (reset), .swap (swap),
  .R1in (wR1in), .R1out (wR1out), .R2in (wR2in), .R2out (wR2out),
  .R3in (wR3in), .R3out (wR3out), .done (wdone));

  // Instantiate registers
  regn #(.n (width)) R1 (.D (wBus), .clk (clk), .reset (reset), .en (
     wR1in), .Q (wR1));
  regn #(.n (width)) R2 (.D (wBus), .clk (clk), .reset (reset), .en (
     wR2in), .Q (wR2));
  regn #(.n (width)) R3 (.D (wBus), .clk (clk), .reset (reset), .en (
     wR3in), .Q (wR3));

  // Bus with tri-state drivers
  bustri #(.n (width)) bustri1 (.w (wR1), .en (wR1out), .f (wBus));
  bustri #(.n (width)) bustri2 (.w (wR2), .en (wR2out), .f (wBus));
  bustri #(.n (width)) bustri3 (.w (wR3), .en (wR3out), .f (wBus));
endmodule
```

18.1.2 Using a Bus with MUXes (Verilog)



We have the following FSM States:

```
IDLE: No swapping
R2TOR3: First copy
R1TOR2: Second copy
R3TOR1: Third copy
```

With the following algorithm for swapping:

```
1 - Swap starts → Copy data from R2 to R3
2 - Copy data from R1 to R2
3 - Copy contents of R3 to R1 → Swap ends
```

And an asynchronous power-on reset

Translating this in Verilog gives us,

```
module controlbusmux (clk, reset, swap, R1in, R2in, R3in, done, sel);
  input clk, reset, swap;
  output reg R1in, R2in, R3in, done;
  output reg [1:0] sel;
  parameter IDLE = 2'b00, R2TOR3 = 2'b01, R1TOR2 = 2'b10, R3TOR1 = 2'b11;
  reg [1:0] S_next, S;
  // Next-state logic
  always @ (*) begin
    S_next = IDLE;
    case (S)
    IDLE: if (swap) S_next = R2TOR3;
        else S_next = IDLE;
    R2TOR3: S_next = R1TOR2;
    R1TOR2: S_next = R3TOR1;
    R3TOR1: S_next = IDLE;
    default: S_next = IDLE;
    endcase
  end
  // State memory
  always @ (posedge clk or posedge reset) begin
    if (reset) S <= IDLE;
    else S <= S_next;
  end
  // Output logic
  always @ (*) begin
    R1in = (S == R3TOR1);
    R2in = (S == R1TOR2);
    R3in = (S == R2TOR3);
    done = (S == R3TOR1);
    sel = S;
  end
endmodule
```

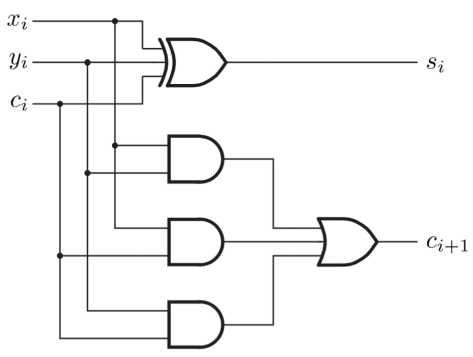
Giving us the following implementation for reg swpping with a bus of MUXes :

```
module regswapbusmux (clk, reset, swap);
    parameter width = 8;
    parameter IDLE = 2'b00, R2TOR3 = 2'b01;
    parameter R1TOR2 = 2'b10, R3TOR1 = 2'b11;
    input clk, reset, swap;
    wire wR1in, wR2in, wR3in, wdone;
    wire [width-1:0] wR1, wR2, wR3;
    wire [1:0] wsel;
    reg [width-1:0] wBus;
    // Instantiate controller module
    controlbusmux controller_module ( .clk (clk), .reset (reset), .swap (
        swap),
    .R1in (wR1in), .R2in (wR2in), .R3in (wR3in),
    .done (wdone), .sel (wsel));
    // Instantiate registers
    regn #(.n (width)) R1 (.D (wBus), .clk (clk), .reset (reset), .en (
        wR1in), .Q (wR1));
    regn #(.n (width)) R2 (.D (wBus), .clk (clk), .reset (reset), .en (
        wR2in), .Q (wR2));
    regn #(.n (width)) R3 (.D (wBus), .clk (clk), .reset (reset), .en (
        wR3in), .Q (wR3));
    // Bus with a multiplexer
    always @ (*) begin
      wBus = wR1;
      case (wsel)
        IDLE: wBus = wR1;
        R2TOR3: wBus = wR2;
        R1TOR2: wBus = wR1;
        R3TOR1: wBus = wR3;
        default: wBus = wR1;
      endcase
    end
endmodule
```

18.2 Alternative ways to impelment Adders

A Full Adder in behavioral Verilog, can be modelled as such:

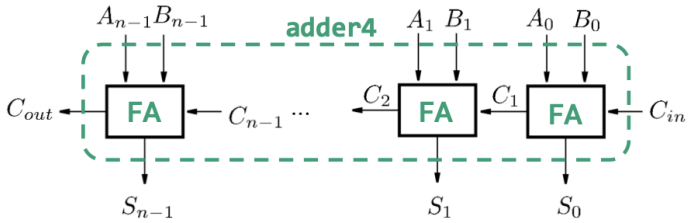
```
module fulladd (a, b, c_in, s,
    c_out);
  input a, b, c_in;
  output s, c_out;
  assign s = a ^ b ^ c_in;
  assign c_out = (a & b) | (a &
      c_in) | (b & c_in);
endmodule
```



18.2.1 Ripple Carry Adder Using a For Loop (Verilog)

To implement a Ripple-Carry Adder, we set the next carry to the C_{out} of the current stage, and calculate the current sum as before.

```
module ripplecarry (Cin, A, B, S,
    Cout);
  parameter n = 32;
  input Cin;
  input [n-1:0] A, B;
  output reg [n-1:0] S;
  output reg Cout;
  reg [n:0] C; // internal wires
  integer k; // loop iterator, an
      integer
  always @(*) begin
    C[0] = Cin;
    for (k = 0; k < n; k = k+1)
        begin
    S[k] = A[k] ^ B[k] ^ C[k];
    C[k+1] = (A[k] & B[k]) | (A[k]
        & C[k]) | (B[k] & C[k]);
  end
  Cout = C[n];
  end
endmodule
```



18.2.2 Generate Construct (Verilog)

The Verilog `generate` construct allows module instantiation within `for` loops and `if-else` statements. Key points include:

- If a `for` loop is included in the `generate` block, the loop index variable must be of type `genvar`.
- A `genvar` is an integer variable that:

  Can only have values ≥ 0.

  Can only be used inside `generate` blocks.
- The `generate` construct enables combining `for` loops with module instantiations effectively.

18.2.3 Ripple-Carry Adder with a generate Construct (Verilog)

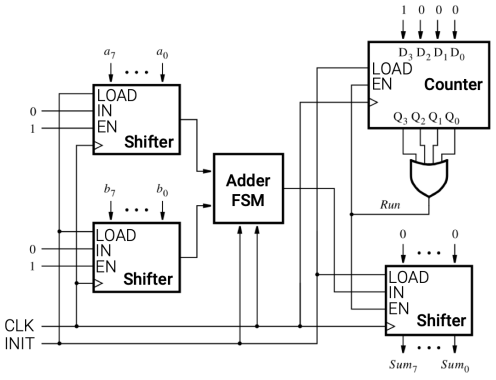
```
module ripplecarrygenerate (Cin, A, B, S, Cout);
  parameter n = 32;
  input Cin;
  input [n-1:0] A, B;
  output [n-1:0] S; // must match the type of fulladd port .s
  output Cout; // must match the type of fulladd port c_out
  genvar g; // generate loop iterator, must have genvar type
  wire [n:0] C; // must match the type of fulladd ports .c_in, c_out
  assign C[0] = Cin;
  // generate block
  generate // optional keyword, helps readability
  for (g = 0; g < n; g = g + 1) begin
  fulladd stage (.a (A[g]), .b (B[g]), .c_in (C[g]), .s (S[g]), .c_out (C[g
      + 1]));
  end
  endgenerate // optional keyword, helps readability
  assign Cout = C[n];
endmodule
```

18.3 Serial Adders

Serial Adders are adders that add bits serially, one bit at a time, if speed is not prority it's a cost effective way to add numbers.

18.3.1 Circuit

- Shift-right registers output one bit of `a` and `b` at a time.
- A down-counter (timer) counts the number of cycles needed for addition, corresponding to the number of bit pairs to add.
- The `INIT` signal loads new values into all shift registers and reinitializes the timer.



18.3.2 Serial Adder Mealy FSM

A Mealy FSM Serial Adder follows the following algorithm:

- Initialize the shift registers and the timer (INIT input)
- Load (in parallel) vectors A and B in their respective shift registers
- Initialize the timer
- Clear the contents of the sum register (all zeros)
- In each clock cycle

    * Add a pair of bits (one bit of A and one bit of B) using the Adder FSM
    * Shift the resulting sum bit into the Sum register
    * Shift the vectors A and B to prepare the next pair of bits
    * Decrement the counter

- Stop when the timer finished counting

The Adder FSM has the following states:
- `S0` = 0
- `S1` = 1

With S_0 and S_1 representing the carry generated by the sum of the previous bits
We get the following State/output tables:

INIT = 0								
Present State **S**	Next state **S_next**				Output **sum**			
	One-bit Inputs **ab**				One-bit Inputs **ab**			
	00	01	10	11	00	01	10	11
S0	S0	S0	S0	S1	0	1	1	0
S1	S0	S1	S1	S1	1	0	0	1

INIT = 1								
Present State **S**	Next state **S_next**				Output **sum**			
	One-bit Inputs **ab**				One-bit Inputs **ab**			
	00	01	10	11	00	01	10	11
S0	S0	S0	S0	S0	0	1	1	0
S1	S0	S0	S0	S0	1	0	0	1

18.3.3 Serial Adder Mealy FSM (Verilog)

```
module serialadderfsm (input a, b, init, reset, clk,output reg sum);
  parameter n = 8;
  parameter S0 = 1'b0, S1 = 1'b1;
  reg S_next, S;
  always @(*) begin
    S_next = S0;
    case (S)
    S0: begin
      if (init) S_next = S0;
      else if (a & b) S_next = S1;
      else S_next = S0;
    end
    S1: begin
      if (init) S_next = S0;
      else if (~a & ~b) S_next = S0;
      else S_next = S1;
    end
    default: S_next = S0;
  endcase
  end
  // State memory, Power-on asynchronous reset
  always @ (posedge clk or posedge reset) begin
    if (reset) S <= S0;
    else S <= S_next;
  end
  // Output logic
  always @(*) begin
    if (S == S0) sum = a ^ b;
    else sum = ~(a ^ b);
  end
endmodule
```

18.4 From Verilg to Circuits

We have been writing Verilog models from their circuit drawing or their description and functionality, we will now be doing the inverse, given a Verilog model we will turn the model back to its circuit and explain its functionality.
Enough with the talking, here's the Verilog program we will be looking at:

```
module guesswhat (clk, resetn, la, s, Data, B, done);
parameter n = 8;
input clk, resetn, la, s;
input [n-1:0] Data;
output reg [3:0] B;
output reg done;

parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;
integer k;
reg [n-1:0] A;
wire z;
reg [1:0] S_next, S;
reg ea, eb, lb;

// State transition logic
always @(*) begin
  S_next = S1;
  case (S)
  S1: if (!s) S_next = S1;
    else S_next = S2;
  S2: if (z == 0) S_next = S2;
    else S_next = S3;
  S3: if (s) S_next = S3;
    else S_next = S1;
  default: S_next = 2'bxx;
  endcase
end

// Output control logic
always @(*) begin
  ea = 0;
  lb = 0;
  eb = 0;
  done = 0;
  case (S)
  S1: lb = 1;
  S2: begin
    ea = 1;
    if (A[0]) eb = 1;
    else eb = 0;
  end
  S3: done = 1;
  endcase
end
```

```
// State register
always @(posedge clk or negedge resetn) begin
if (!resetn) S <= S1;
else S <= S_next;
end

// Output B register
always @(posedge clk or negedge resetn) begin
if (!resetn) B <= 0;
else if (lb) B <= 0;
else if (eb) B <= B + 1;
end

// Register A logic
always @(posedge clk or negedge resetn) begin
if (!resetn) A <= 0;
else if (la) A <= Data;
else if (ea) begin
for (k = n - 1; k > 0; k = k - 1) begin
A[k-1] <= A[k];
end
A[n-1] <= 1'b0;
end
end

// Zero detect logic
assign z = ~| A;
endmodule
// END
```

Now, we can analyse this step by step, starting with the `guesswhat` *declaration:*

```
module guesswhat (clk, resetn, la,
    s, Data, B, done);
parameter n = 8;
input clk, resetn, la, s;
input [n-1:0] Data;
output reg [3:0] B;
output reg done;
parameter S1 = 2'b00, S2 = 2'b01,
   S3 = 2'b10;
integer k;
reg [n-1:0] A;
wire z;
reg [1:0] S_next, S;
reg ea, eb, lb;
//continues...
```

Step 1: Start by analyzing the module interfaces, parameters, and internal signals (wires) and variables (reg, integer).

- The output, `B`, has a width different from the Data input.
- $S1$,$S2$,$S3$, are likely states of an FSM
- integer k is likely a loop iterator
- and A is a register of the same width as Data input

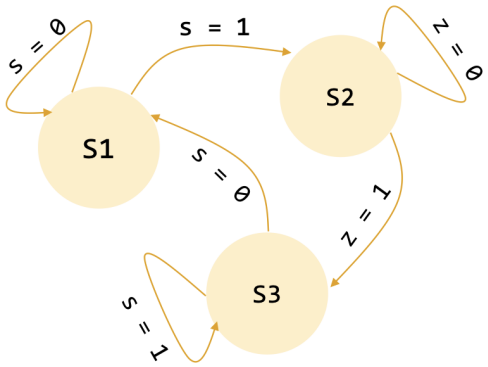
Next State Logic

```
//...
always @(*) begin
  S_next = S1;
  case (S)
    S1: if (!s) S_next = S1;
    else S_next = S2;
    S2: if (z == 0) S_next = S2;
    else S_next = S3;
    S3: if (s) S_next = S3;
    else S_next = S1;
    default: S_next = 2'bxx;
  endcase
end
// continues...
```

Step 2: We analyze the Next State Logic, which is likely part of an FSM.

- The FSM has three states, S1, S2, and S3.
- The FSM transitions are based on the values of the input signals s and z.

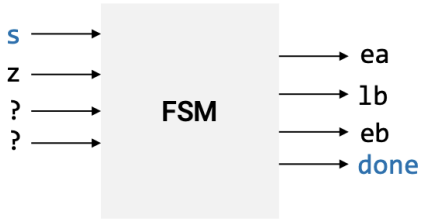
Giving use the following state diagram



FSM Output Logic

```
//...
always @(*) begin
  ea = 0;
  lb = 0;
  eb = 0;
  done = 0;
  case (S)
  S1: lb = 1;
  S2: begin
      ea = 1;
      if (A[0]) eb = 1;
      else eb = 0;
    end
  S3: done = 1;
  endcase
end
// continues...
```

The FSM Output Logic will likelt look like this:



- S1: `lb` = 1, others zero.
- S2 : `ea` = 1, `eb` = A[0], others zero.
- S3 : `done` = 1, others zero.

Sequential Circuits

```
// ...
always @(posedge clk or negedge
    resetn) begin
  if (!resetn) S <= S1;
  else S <= S_next;
  end
// continues...
```

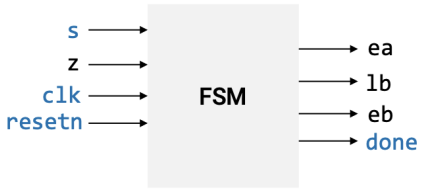
```
// ...
always @(posedge clk or negedge
    resetn) begin
  if (!resetn) B <= 0;
  else if (lb) B <= 0;
  else if (eb) B <= B + 1;
end
// continues...
```

```
  // ...
  always @(posedge clk or negedge
      resetn) begin
    if (!resetn) A <= 0;
    else if (la) A <= Data;
    else if (ea) begin
      for (k = n - 1; k >0; k = k
          - 1) begin
        A[k-1] <= A[k];
      end
      A[n-1] <= 1'b0;
    end
  end
  // continues...
```

FSM State Register

- **negedge** in the sensitivity list:
`resetn` is an asynchronous reset active low, external signal.
  - State after power-on reset: **S1**;

Hence, **S1** is the default state.



B is the output of a counter

- **1b** clears it (loads all zeros)
  * In **S1** state
- **eb** enables it (allows counting)
  * In **S2** state, if **A[0]** is **1**
- **B + 1**: up counter

Shift register A

- **resetn**: asynchronous reset
  - Active low, external signal
- **la** initializes it (loads Data)
  - **la** is an external signal
  - **Data** is also external
- **for** loop performs shift right
  - Most significant bits cleared (replaced by **1'b0**)
- **ea** enables shift right
  - In state **S2**, while up-counter **B** is enabled too

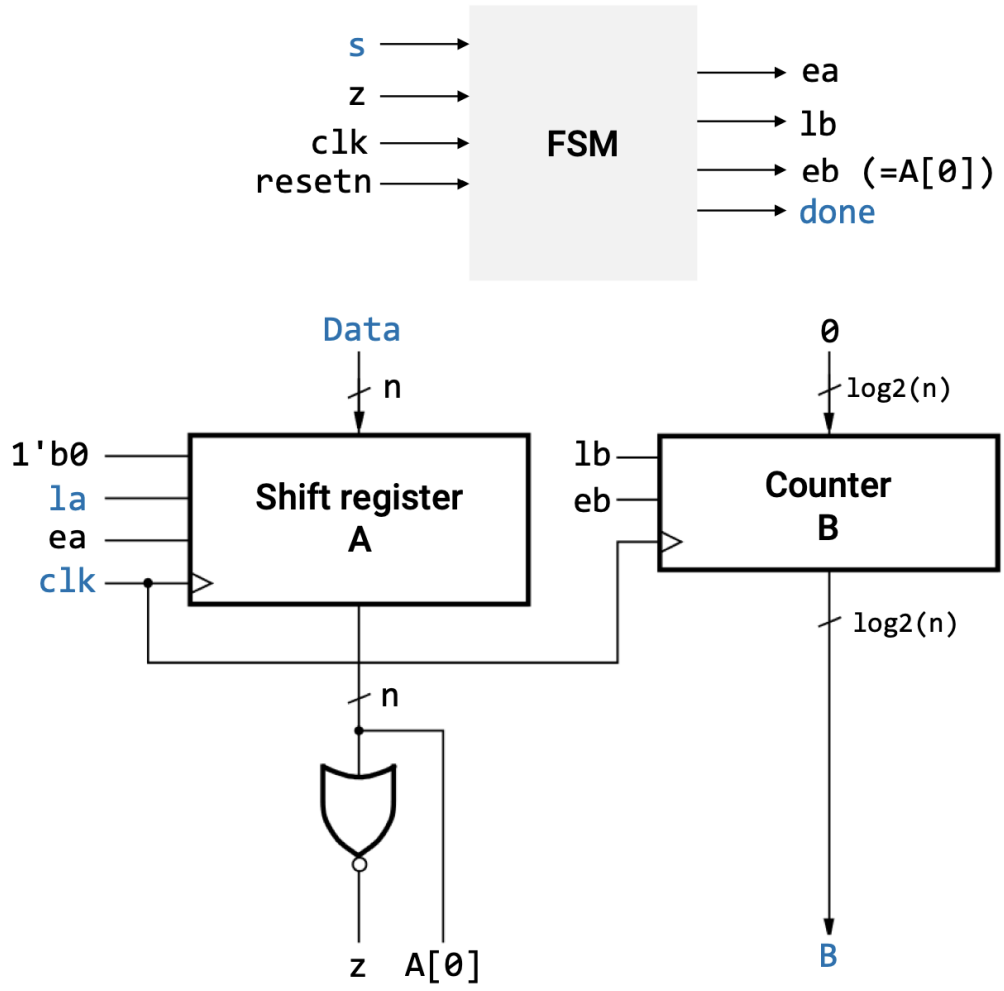
Combinational Logic

Now that we have a broad idea of the circuit, we need to understand when it stops

```
//...
assign z = ~| A;
endmodule
// END
```

The last assignment in the module, $z = \sim |A$, uses the Verilog reduction NOR operator, which sets z to 1 when all bits of vector A (the output of the shift register) are zero.

All of this leads us to the following circuit:



18.5 Verilog Reduction Operators

Reduction operators in Verilog are unary operators that perform bitwise operations across all bits of a vector operand to produce a single-bit result.

Operator	Description
&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction xor
~^	reduction xnor

Examples

Consider the vector $A = 8'b10101111$:

- **Example 1:** $z = \&A$
  `z = ((((((1 & 1) & 1) & 1) & 0) & 1) & 0) & 1) = 0`
- **Example 2:** `z = ~^A`
  `z = ~((((((1 ^ 1) ^ 1) ^ 1) ^ 0) ^ 1) ^ 0) ^ 1) = 1`

Chapter 19

Computer Architecture (Part I) (W12.1)

The knowledge we have built on so far allows to design an en entire simple processor, a simple one, but a processor.

19.1 A Processor

A processor, also known as a Central Processing Unit (CPU), is the central component in any general-purpose computing system, such as phones, laptops, tablets, and servers, responsible for executing software applications, facilitating data processing, and orchestrating data manipulations according to software instructions.

19.2 From Programs to Computers

To understand how a processor work, we need to understand how a program is executed by the computer, and that, on a lower level of abstraction.
Here we will be looking at this program written in the C programming language (still a little high level):

```
1  // variable initialization
2  int data = 0x00123456; // hexadecimal
3  int result = 0;
4  int mask = 1;
5  int count = 0;
6  int temp = 0;
7  int limit = 32;
8  do { // loop
9  temp = data & mask; // bitwise and
10 result = result + temp; // addition
11 data = data >> 1; // shift right
12 count = count + 1; // addition
13 } while (count != limit); //
       condition
```

We may take a look at Reading, Writing, Update and Compute operations;

These correspond to the assignments variable initializations on lines 1-7 and to the assignments and computing on lines 9-13.

We also see a do-while loop with the condition of the count being different to the set limit.

Like in Java, there are controls to execute the program one line after the other, skip some lines, loop, or return to a previous line.

This program counts the number of ones in the 32 bit integer `data` and stores the result in the variable `result`. Giving us `result = 9` for the value of `data = 0x00123456`.

We may represent the do-while loop as follows: *the data variable gets shiffted right at each iteration and the temp varuable stores the least significant bit.*

Step	data	temp	result
Initialization	0000 0000 0001 0010 0011 0100 0101 0110	0	0
1st Loop iteration	0000 0000 0001 0010 0011 0100 0101 0110	0	0
2nd Loop iteration	0000 0000 0000 1001 0001 1010 0010 1011	1	1
3rd Loop iteration	0000 0000 0000 0100 1000 1101 0001 0101	1	2
4th Loop iteration	0000 0000 0000 0010 0100 0110 1000 1010	0	2
5th Loop iteration	0000 0000 0000 0001 0010 0011 0100 0101	1	3
...	...	...	...
31st Loop iteration	0000 0000 0000 0000 0000 0000 0000 0000	0	9
32nd Loop iteration	0000 0000 0000 0000 0000 0000 0000 0000	0	9

19.3 Hardware-Friendly Programs

Hardware does not understand high-level languages like C or Python; it only understands machine language, which consists of binary instructions that direct the CPU to perform specific tasks.

19.3.1 Translating High-Level Code into Binary

This is the process we need to go through to translate the high-level code into binary instructions:

* **Source Code:** Starts with high-level programming language.
* **Compilation/Interpretation:**
  - Compiler translates code to intermediate/binary formats.
  - Interpreter executes source code line-by-line.
* **Optimization:** Enhances code performance/reduces size.
* **Linking:** Combines code with libraries/functions (for compiled languages).
* **Assembly Code Generation (Optional):** Converts to human-readable machine code.
* **Binary Code Generation:** Converts to binary instructions (0s and 1s).

19.3.2 From High-Level Programs to Assembly

Assembly language is a low-level programming language that is a human-readable representation of machine code, it is considered as the closest level of abstraction to machine code.

`Step 1:` Add line numbers and labels

`Step 2:` Assign variables to registers

`Step 3:` Replace each code line with a corresponding machine language instruction

Add line numbers and labels

Number only lines of code that affect program execution, label important lines (e.g., loop body), and use appropriate symbols for comments.

```
      # variable initialization
0     int data   = 0x00123456;
1     int result = 0;
2     int mask   = 1;
3     int count  = 0;
4     int temp   = 0;
5     int limit  = 32;
      do {
6 loop:  temp   = data & mask;
7        result = result + temp;
8        data   = data >> 1;
9        count  = count + 1;
10    } while (count != limit);
```

Assign variables to registers

Assign variables to registers, use comments to indicate the register used for each variable. Assuming registers are named x0, x1, x2, etc., and each register stores 32 bits, we get :

```
      # variable initialization
0     x1   = 0x00123456; # data
1     x2   = 0;          # result
2     x3   = 1;          # mask
3     x4   = 0;          # count
4     x5   = 0;          # temp
5     x6   = 32;         # limit
      do {
6 loop: x5   = x1 & x3;
7       x2   = x2 + x5;
8       x1   = x1 >> 1;
9       x4   = x4 + 1;
10    } while (x4 != x6);
```

Instructions

Let us replace the operations of our program with the corresponding assembly instructions:

Instruction	Description
li	**l**oad a literal (an **i**mmediate) into a variable
and	bitwise and of two variables
add	addition of two variables
addi	addition of a variable and a literal (an **i**mmediate, a value)
srli	shift right logical by a literal (an **i**mmediate)
bne	if two variables are **n**ot **e**qual, go to the line with the label (**b**ranch)

Giving us the following pseudo-code:

Operation name	Destination,	Left operand,	Right operand

```
0        li   x1, 0x00123456
1        li   x2, 0
2        li   x3, 1
3        li   x4, 0
4        li   x5, 0
5        li   x6, 32
                          # do {
6   loop: and  x5, x1, x3    # x5   = x1 & x3
7        add  x2, x2, x5     # x2   = x2 + x5
8        srli x1, x1, 1      # x1   = x1 >> 1
9        addi x4, x4, 1      # x4   = x4 + 1
10       bne  x4, x6, loop   # while (x4 != limit)
```

Complete Assembly Code

We can now complete the assembly code:

```
    li x1, 0x00123456
    li x2, 0
    li x3, 1
    li x4, 0
    li x5, 0
    li x6, 32
loop: and x5, x1, x3
    add x2, x2, x5
    srli x1, x1, 1
    addi x4, x4, 1
    bne x4, x6, loop
```

It is to be noted, though, that a fast 32-bit binary ones counter can be designed with logic gates or shift registers and counters, and that, while processors are very versatile, they are slower than specialized solutions.

19.4 Under the Hood

Sous la capuche
Now the real deal begins, we will be looking at the hardware that executes the assembly code, the processor.
There are two parts to a processor, the Data path, and the Control Path.

19.4.1 Data Path

The Data Path, or `datapath` is the part of the processor that performs arithmetic and logical operations on data. It consists of:

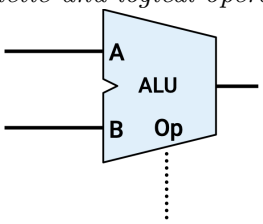
* **Registers:** is an array of registers (i.e., the memory) or the program variables and some other special uses

* **ALU (Arithmetic Logic Unit):** Performs the operations on program variables *E.g., bitwise, logic, shift, comparisons, etc.*

Arithmetic Logic Unit (ALU)

The ALU is the heart of the processor, it performs all the arithmetic and logical operations.

- **A** and **B** are two 32-bit data variables.
- **Op** is a select input that chooses the operation to perform.

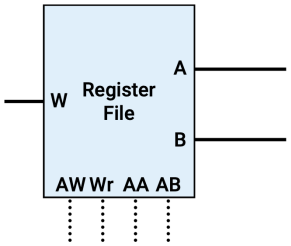


Register File

A register file in a CPU is a small, fast storage area that contains a set of registers, which hold operands and results for quick access during instruction execution. It can read two registers in the same clock cycle and stores intermediate results, operands, memory addresses, and control information for CPU instructions.

`With the following inputs:`
For reading:
- *AA:* **A**ddress of **A** register to be read (output A)
- *AB:* **A**ddress of **B** register to be read (output B)

For writing:
- *W:* Data input, 32-bit data to write into one of the registers (e.g., X register)
- *Wr:* Load input, 1-bit; if active, load value from W into X register
- *AW:* Address of X register to write to



Example of Data Path Execution

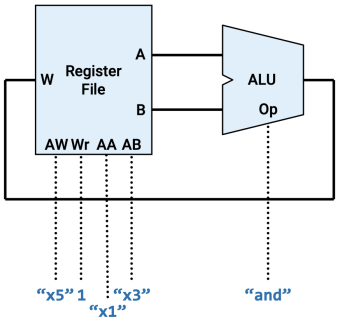
Consider the following instruction:

```
1 and x5, x1, x3
```

This instruction applies the AND operation on x1 and x3 and stores the result in x5.
- The **register file** receives the addresses of the registers to read from (x1, x3) and to write to (x5) from the **control logic** (in the next section of the course).

- The **register** outputs the data values of registers x1 and x3 to the **ALU**, which also receives the operation to perform from the **control logic**.

- The **ALU** performs the operation and sends the result back to the **register file**, which writes it into register x5.



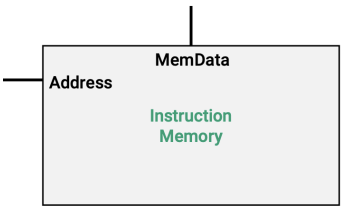
19.4.2 Control Path

The control logic of the processor handles instruction reading, ensures correct sequencing, decodes instructions, sets control signals for the ALU and Register File based on the instruction, and implements the processor's finite-state machine.

Instruction Memory

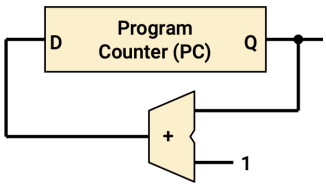
The instruction memory stores the program instructions.

It takes the address of the memory instruction to be read from the **Address** port as an input and outputs the instruction to be executed to the **MemData** port.



Program Counter (PC)

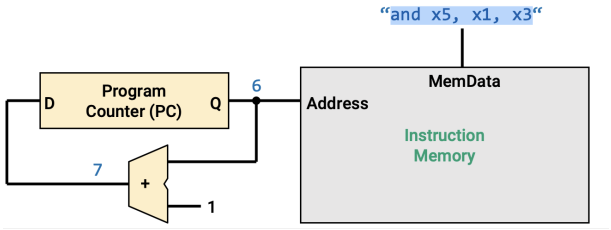
The **Program Counter** register holds the adress of the instruction to be executed and the Adder increments the content of the PC such that the instruction is read on the next clock cycle



Example

For this command

```
1 and x5, x1, x3
```

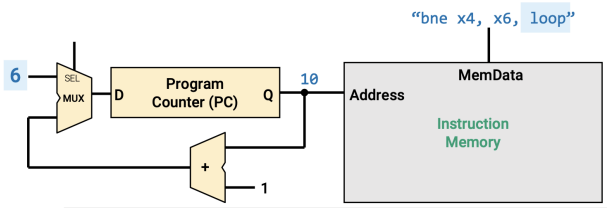


Example of Loop

In a loop execution, we need a **multiplexer** that holds the address of the line at the beginning of the loop and the usual program counter.
The **selector** of the multiplexer depends on the **Control Logic** *(coming in the next section).*

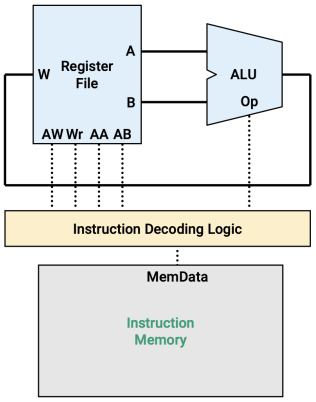
Here for example, the instruction `bne` *tells us that the this is a while loop with the condition that* `x4 != x6`*.*

```
1 loop: and x5, x1, x3
2   ...
3   bne x4, x6, loop # while (x4 !=
      limit)
```



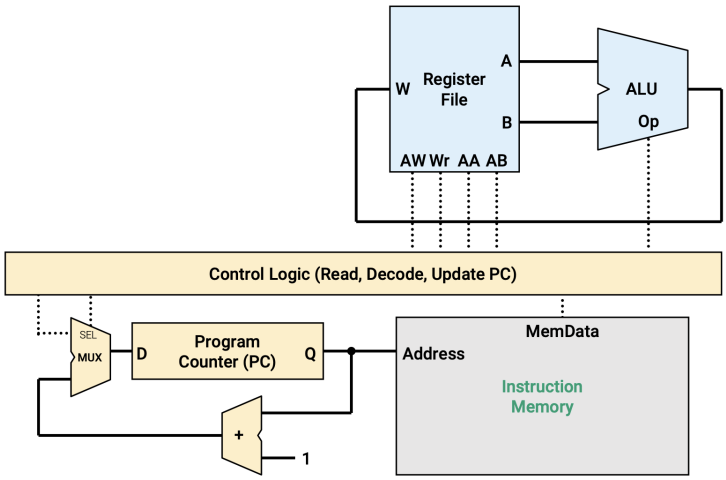
Decoding Instructions

Once the instruction is read from the instruction memory, it must be ***decoded****.*
This means parsing the binary representation of the instruction with the goal of identifying the operation, destination register, and the operands.



19.5 Data Path + Control Path

From what we've learned so far...
Now we have a really really basic Processor, the only problem is that it can only process the limited set of data we can s

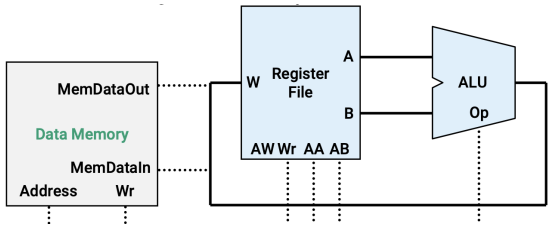


In practice, we need an external memory to store the data only, or both the data and instructions.

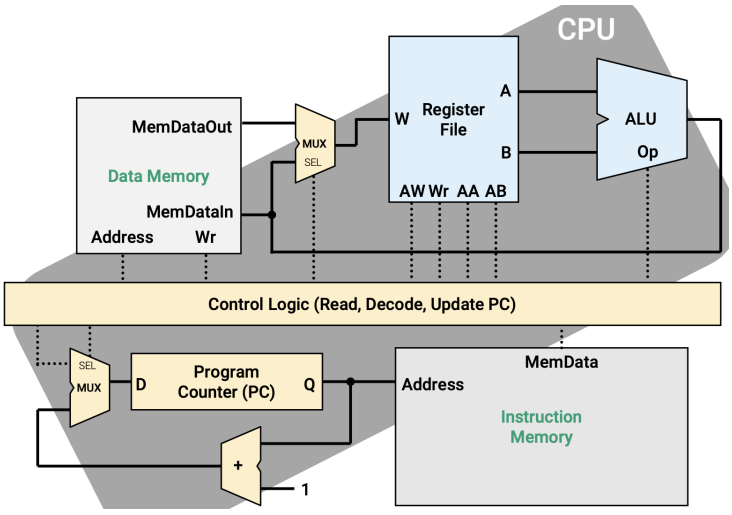
19.5.1 Data Memory

Data memory is a storage area that holds the data to be processed during the execution of a program. It has higher latency than the register file, but it can store a larger amount of data.
Personal Remark. Remember, Register File for intermediate results, currently used operands, etc... Data Memory, for the data to be processed.

- **Address:** Address of the data to be read or written
- **MemDataIn** and **MemDataOut** input and output data ports
- **Wr** load, active when writing to the memory



19.6 Complete Processor



19.7 Types of Processors

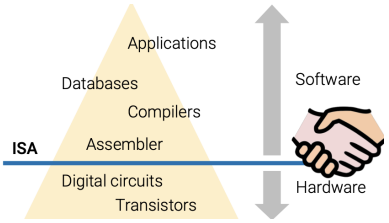
* Harvard architecture
  - Instructions and data memory reside in separate memories
  - See the block diagram on the previous slide
* Von Neumann architecture
  - Instructions and data reside in the same memory
  - We say the memory is unified
  - In general-purpose computers (desktops, laptops, servers, etc.), von Neumann architecture is predominant

19.8 Instruction Set Architecture

The **Instruction Set Architecture** (ISA) defines the set of instructions a CPU can execute and the programming model for software developers, abstracting hardware details to create a consistent interface between hardware and software, allowing CPUs to evolve while maintaining compatibility, though it limits fundamental innovations that would require changes to the ISA.

A common ISA example is IA-32/x86, introduced by Intel, and used in Intel's Core, Pentium, and Xeon series, as well as AMD's Ryzen, Athlon, and EPYC series. Typical details of an ISA include:

- Instruction set: operations the processor can directly execute.
- Instruction encoding: binary representation of instructions.
- Registers: storage for intermediate results and operands.
- Data types and formats: e.g., integers, floating-point numbers, characters.
- Memory addressing modes: how the processor accesses operands from memory.



19.9 RISC V

RISC ISAs are simpler and more regular than CISC, making them easier to implement, with examples including MIPS, Alpha, Sparc, and RISC-V, while most Intel processors are CISC; further advantages of RISC will be explored in the CS-208 Computer Architecture course(force).

RISC-V ISA is modular, centered around a stable base ISA called RV32I for 32-bit integer operations, with optional standard extensions like multiply, single-precision, and double-precision floating-point to customize hardware configurations.

Chapter 20

Computer Architecture (Part II) (W12.2)

Parts of the content we will be covering are actually available in the RV32I Reference Card (so no need to learn the encondings of instructions by heart for example, understanding them, however, is necessary).

20.1 RV32I

RV32I is the base 32-bit integer instruction set architecture of RISC-V, featuring 32-bit wide registers, memory addresses, data words, instructions, and a little-endian memory system, with extendable optional instruction-set extensions.

20.1.1 Registers

In **RV32I**, there are 31 general-purpose registers, **x1** to **x31**. There is a **x0** register that is hardwired to the constant **0**. And a **PC** register that holds the address of the next instruction to be executed. Each register is *32 bits* wide and hold signed integers in **two's complement**.

Register	Name	Description
x0	zero	Hard-wired zero
x1	ra	**Return address*
x2	sp	**Stack pointer*
x3	gp	**Global pointer*
x4	tp	**Thread pointer*
x5-7	t0-2	Temporaries
x8-9	s0-1	Saved registers
x10-11	a0-1	**Function arguments/return values*
x12-17	a2-7	**Function arguments*
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporaries
pc	Program counter	Program counter

All with (*) is Out of Scope.

Out of Scope:
Program variables are commonly kept in t and s registers, the difference between s and t registers is that s registers are preserved across function calls, while t registers are not.

20.1.2 Computational Instructions

There are three types of operations:

- **Integer register-register operations**:
  - These operations involve instructions that use registers to store both operands.
  - They use the **R-type instruction format**.
- **Integer register-immediate operations**:
  - These operations involve instructions where one operand is a register and the other is an immediate value (a constant).
  - They use the **I-type instruction format** and the **U-type instruction format**.
- **NOP pseudoinstruction**:
  - NOP stands for "No Operation" and is a pseudoinstruction that does not perform any operation but can be used for timing purposes or as a placeholder in the instruction sequence.

Integer Register-Register Operations

A Register-Register is an operation that is of R-type. As it's name suggests, it involves executing an operation on data from two registers of the Register File (RF), and saving the result in a register of RF. *Which can be wrote as:*

$$RF[rd] = f(RF[rs1], RF[rs2])$$

With $rs1$, $rs2$, rd, and $f()$ being encoded in the instruction.
These operations are encoded as follows :

funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

* **rs1**: src1 (source), 5-bit index of the first operand register
* **rs2**: src2 (source), 5-bit index of the second operand register
* **rd**: dest (destination), 5-bit index of the result destination register
* **opcode**: $(0110011)_2$, 7-bit operation code (OP)
* **funct3**: 3-bit function code (**AND/OR/XOR/SLL/SRL/ADD/SLT/SLTU/SUB/SRA**)
* **funct7**: 7-bit function code
  - $(0000000)_2$ for **AND/OR/XOR/SLL/SRL/ADD/SLT/SLTU**
  - $(0100000)_2$ for **SUB/SRA**
* **ADD/SUB**: addition/subtraction (Overflows are ignored)
* **AND/OR/XOR**: bitwise logical operations
* **SLL/SRL/SRA**: logical left, logical right, and arithmetic shift right of RF[rs1] by the shift amount specified by bits [4:0] of RF[rs2]
* **SLT/SLTU**: signed/unsigned comparisons
  - RF[rd] = 1, if RF[rs1] ¡ RF[rs2]; otherwise, RF[rd] = 0
  - Origin of the acronym: **set on less than**

The encoding thus looks like this

Instruction	Operation	funct7	rs2	rs1	funct3	rd	opcode
add rd, rs1, rs2	RF[rd] = RF[rs1] + RF[rs2]	0000000	rs2	rs1	000	0110011	ADD
sub rd, rs1, rs2	RF[rd] = RF[rs1] - RF[rs2]	0100000	rs2	rs1	000	0110011	SUB
sll rd, rs1, rs2	RF[rd] = RF[rs1] << RF[rs2]	0000000	rs2	rs1	001	0110011	SLL
slt rd, rs1, rs2	RF[rd] = RF[rs1] $<_S$ RF[rs2]	0000000	rs2	rs1	010	0110011	SLT
sltu rd, rs1, rs2	RF[rd] = RF[rs1] $<_U$ RF[rs2]	0000000	rs2	rs1	011	0110011	SLTU
xor rd, rs1, rs2	RF[rd] = RF[rs1] ^ RF[rs2]	0000000	rs2	rs1	100	0110011	XOR
srl rd, rs1, rs2	RF[rd] = RF[rs1] $>>_U$ RF[rs2]	0000000	rs2	rs1	101	0110011	SRL
sra rd, rs1, rs2	RF[rd] = RF[rs1] $>>_S$ RF[rs2]	0100000	rs2	rs1	101	0110011	SRA
or rd, rs1, rs2	RF[rd] = RF[rs1] | RF[rs2]	0000000	rs2	rs1	110	0110011	OR
and rd, rs1, rs2	RF[rd] = RF[rs1] & RF[rs2]	0000000	rs2	rs1	111	0110011	AND

With $<_S$ stands the signed comparison, $<_U$ the unsigned comparison, $>>_S$ the signed shift right, $>>_U$ the unsigned shift right

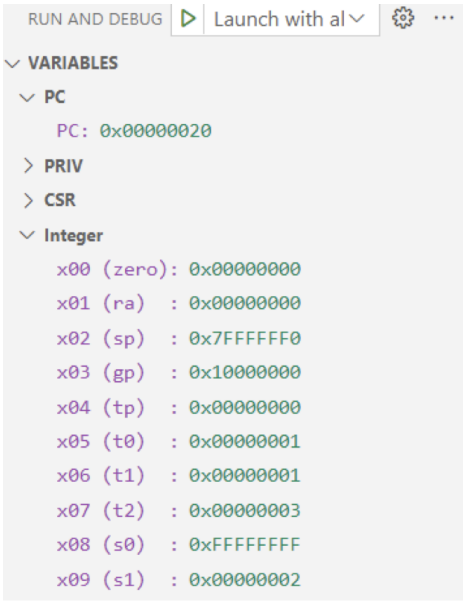
Example

Let's say we're trying to implement, a = b + c + d - e in RISC-V assembly.

The `li` (load immediate) instruction is a pseudo instruction which the assembler translates to addi.

```
# Assuming a, b, c, d are in regs
# x5, x6, x7, x8, respectively
# i.e., t0, t1, t2, s0
# variable initialization
li t0, 0 # a = 0
li t1, 1 # b = 1
li t2, 3 # c = 3
li s0, -1 # d = -1
li s1, 2 # e = 2
add t0, t1, t2 # t0 = temp = b + c = 4
add t0, t0, s0 # t0 = temp + d = 3
sub t0, t0, s1 # a = t0 = temp - e = 1
```

Translated to the following encoding (written in Hexadecimal)



20.1.3 Integer Register-Immediate Operations

Integer immediate operations are I-type operations or U-type operations. As the name suggests, they involve executing an operation on data from a register of the Register File (RF) and an immediate value (coming from the instruction), then saving the result in a register of the RF.

These operations are encoded as follows:
The supported formats are:

- I-type: ADDI/ANDI/ORI/XORI/SLTI/SLTIU/SLLI/SRLI/SRAI
- U-type: LUI/AUIPC

I-type

Two variants here depending on the operation:
For (ADDI/ANDI/ORI/XORI/SLTI/SLTIU):

imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

* **rs1:** src1 (source), 5-bit index of the operand register in the register file (RF)
* **rd:** dest (destination), 5-bit index of the destination register in the RF
* **opcode:** $(0010011)_2$, 7-bit operation code referred to as OP-IMM
* **funct3:** 3-bit function code (ADDI/ANDI/ORI/XORI/SLTI/SLTIU)
* **imm:** 12-bit immediate

- **ADDI**: Adds the sign-extended 12-bit immediate to RF[rs1]. Arithmetic Overflow ignored. The lowest 32bits of the result is saved in RF[rd].
- **ANDI/ORI/XORI**: Perform bitwise AND/OR/XOR on RF[rs1] and the sign-extended 12-bit immediate, result is saved in RF[rd].
- **SLTI** (set less than immediate): RF[rd] = 1 if RF[rs1] $\leq$ immediate (signed), else RF[rd] = 0.
- **SLTIU** (set less than immediate unsigned): RF[rd] = 1 if RF[rs1] $\leq$ immediate (unsigned, sign-extended to 32 bits), else RF[rd] = 0.

For (SLLI/SRLI/SRAI):

imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7

* **rs1:** src1 (source), 5-bit index of the operand register in the register file (RF)
* **rd:** dest (destination), 5-bit index of the destination register in the RF
* **opcode:** $(0010011)_2$, 7-bit operation code also called OP-IMM
* **funct3:** 3-bit function code (SLLI/SRLI/SRLA)
* **imm[4:0]:** 5-bit immediate, the number of bits to shift by, also called shamt
* **imm[11:5]:** 7-bit immediate
  - $(0000000)_2$ for SLLI/SRLI
  - $(0100000)_2$ for SRAI

- **SLLI/SRLI/SRAI**, shift by the **immediate**
  - The operand to be shifted is RF[rs1]
  - The shift amount is encoded in the lower five bits of the immediate field (sufficient for the max being a 32-bit word)
  - The particular shiffting operation is encoded in the higher seven bits of the immediate field
- **SLLI:** shift left logical by the **immediate**
- **SRLI/SRAI:** shift right logical/**arithmetic** by the **immediate**

Example

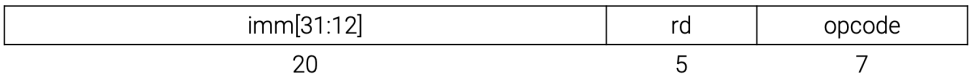
```
# Assuming a, b, c, d are in regs
# x5, x6, x7, x8, respectively
# i.e., t0, t1, t2, s0
# variable initialization
add t0, zero, zero # a = 0
addi t1, zero, 1 # b = 1
addi t2, zero, 3 # c = 3
add s0, zero, zero # d = 0
add s1, zero, zero # temp = 0
add t0, t1, t2 # t0 = b + c = 4
addi s1, s0, 64 # s1 = d + 64 = 64
sub t0, t0, s1 # a = t0 - s1 = -60
```

Register	Decimal	Hex	Binary
x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
x1 (ra)	0	0x00000000	0b00000000000000000000000000000000
x2 (sp)	0	0x00000000	0b00000000000000000000000000000000
x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
x4 (tp)	0	0x00000000	0b00000000000000000000000000000000
x5 (t0)	-60	0xffffffc4	0b11111111111111111111111111000100
x6 (t1)	1	0x00000001	0b00000000000000000000000000000001
x7 (t2)	3	0x00000003	0b00000000000000000000000000000011
x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
x9 (s1)	64	0x00000040	0b00000000000000000000000001000000

Take a look at the online compiler for RISC-V assembly.

U-type

Don't forget there are two types of instructions for an Integer Immediate Operation, the U-type is the second one.



- **U-type format uses only one register, the destination register**
- **rd**: dest (destination), 5-bit index of the destination register in the RF
- **opcode**: 7-bit operation code
  - $(0110111)_2$ for LUI
  - $(0010111)_2$ for AUIPC
- **imm[31:12]**: 20-bit immediate
- **LUI**: **L**oad **u**pper **i**mmediate; used to build 32-bit constants
  - LUI places the 20-bit immediate value in the top 20 bits of the destination register RF[rd], filling the lowest 12 bits with zeros
  - *Don't forget that, `li` is a pseudo command that is translated into one or more instructions (typically LUI and another instruction like ADDI) to load a full 32-bit immediate.* ***It is advised to use li instead of LUI.***
- **AUIPC**: Add upper immediate to the Program Counter (pc) register
  - AUIPC forms a 32-bit offset from the 20-bit immediate value, filling the lowest 12 bits with zeros. Then, it adds this offset to the pc, and places the result in register RF[rd]. *Often used to calculate the address of a global variable, or to jump to a label.*
  - Typically used when one needs to access data or functions located at fixed addresses within the program or memory.

Instruction	**Operation**	**funct7**	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
addi rd, rs1, imm	RF[rd] = RF[rs1] + sext(imm)	0000000	rs2	rs1	000	0110011	ADDI
andi rd, rs1, imm	RF[rd] = RF[rs1] & sext(imm)	0000000	rs2	rs1	000	0110011	ANDI
ori rd, rs1, imm	RF[rd] = RF[rs1] | sext(imm)	0100000	imm	rs1	000	0110011	ORI
xori rd, rs1, imm	RF[rd] = RF[rs1] ^ sext(imm)	0000000	imm	rs1	001	0110011	XORI
slti rd, rs1, imm	RF[rd] = RF[rs1] $<_s$ sext(imm)	0000000	imm	rs1	010	0110011	SLTI
sltiu rd, rs1, imm	RF[rd] = RF[rs1] $<_u$ sext(imm)	0000000	imm	rs1	011	0110011	SLTIU
slli rd, rs1, imm	RF[rd] = RF[rs1] << imm[4:0]	0000000	imm	rs1	100	0110011	SLLI
srli rd, rs1, imm	RF[rd] = RF[rs1] $>>_u$ imm[4:0]	0000000	imm	rs1	101	0110011	SRLI
srai rd, rs1, imm	RF[rd] = RF[rs1] $>>_s$ imm[4:0]	0100000	imm	rs1	101	0110011	SRAI
lui rd, imm	RF[rd] = sext(imm[31:12] << 12)	0000000	imm	rs1	110	0110011	LUI
auipc rd, imm	RF[rd] = pc + sext(imm[31:12] << 12)	0000000	rs2	rs1	111	0110011	AUIPC

With $<_S$ stands the signed comparison, $<_U$ the unsigned comparison, $>>_S$ the signed shift right, $>>_U$ the unsigned shift right

20.1.4 NOP Pseudoinstruction

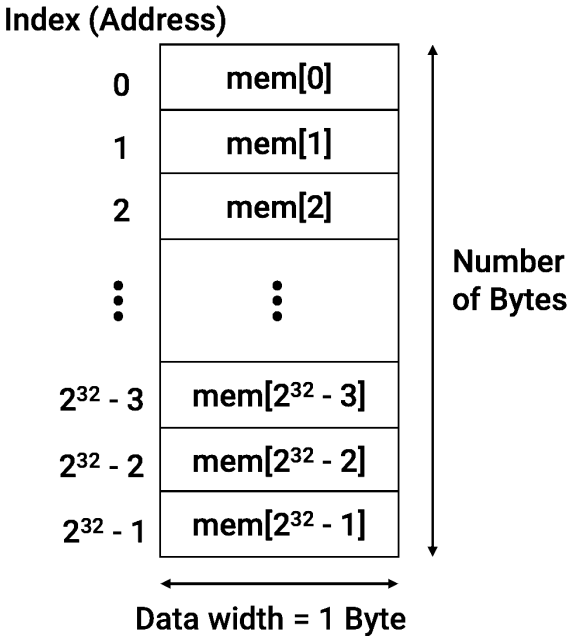
NOP stands for "No Operation" and is a pseudoinstruction that does not perform any operation but can be used for timing purposes or as a placeholder in the instruction sequence.
It does not change any user-visible state besides advancing the program counter.
It is exactly encoded as `addi x0, x0, 0`.

Chapter 21

Computer Architecture (Part III) (W13.1)

21.1 Memory

* **Memory Capacity:** 2^32 bytes.
* **Data Width** = 1 Byte
* **32-bit Adress**
  - **Adress range:** 0 to $2^{32} - 1$
  - **Adress space** is circular (overflows ignored), so that the byte at adresse $2^{32} - 1$ is adjacent to the byte at adress 0.
  - Memory is **byte-addressable**, meaning that each byte of memory has a unique 32-bit address.
* **Von Neuman** architecture is the most commonly used architecture with a unified memory and shared by the program(code), datasets, temporary variables, I/O devices, etc.
* RISC-V data size terminology
  - **Byte:** 8 bits = 1 B
  - **Halfword:** 16 bits = 2 B
  - **Word:** 32 bits = 4 B
  - **Quadword:** 128 bits = 16 B



21.1.1 Alignment of Instructions

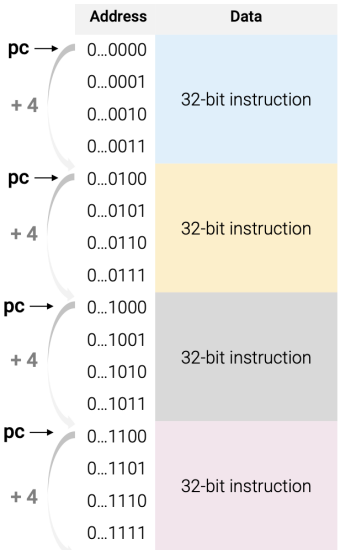
Remember, in RISC-V, instructions are 32-bit

As such, instructions occupy, one **word**.
Thus, with a memory capacity of 2^{32} B, we can store 2^{30} words. Instructions must be naturally aligned on 32-bit boundaries
Valid address ranges: 0-3, 4-7, 8-11, ...
Therefore, instructions span blocks of memory addresses of the format

$$(4k - 4,\ 4k - 3,\ 4k - 2,\ 4k - 1), \quad 1 \leq k \leq 2^{30}$$

Consequently, to prepare for reading the next program instruction from the memory, the value in the PC is increased in steps of four

$$\mathbf{pc = pc + 4}$$



21.1.2 Memory Word and Byte Ordering

A memory word $W = \{w_{31}, w_{30}, \ldots, w_1, w_0\}$ contains four bytes:

$$B_0 = \{b_7, b_6, \ldots, b_0\} = \{w_7, w_6, \ldots, w_1, w_0\}$$
$$B_1 = \{b_7, b_6, \ldots, b_0\} = \{w_{15}, w_{14}, \ldots, w_9, w_8\}$$
$$B_2 = \{b_7, b_6, \ldots, b_0\} = \{w_{23}, w_{22}, \ldots, w_{17}, w_{16}\}$$
$$B_3 = \{b_7, b_6, \ldots, b_0\} = \{w_{31}, w_{30}, \ldots, w_{25}, w_{24}\}$$

These bytes are mapped to the word's four consecutive memory addresses $(4k-4, 4k-3, 4k-2, 4k-1)$, where $1 \leq k \leq 2^{30}$.

There exist two common orderings: **little endian** and **big endian**. The latest RISC-V ISA specification supports both orderings. Originally, only little-endian byte ordering was assumed.

Little-endian
The least significant bits of the word (byte B0, the one at the "little" end of the word) is at the lowest memory address

Address	Byte, bits 7 down to 0
...	...
4k – 4	B0
4k – 3	B1
4k – 2	B2
4k - 1	B3

Big-endian
The most significant bits of the word (byte B3, the one at the "big" end of the word) is at the lowest memory address

Address	Byte, bits 7 down to 0
...	...
4k – 4	B3
4k – 3	B2
4k – 2	B1
4k - 1	B0

Instruction Encoding

Suppose we were to use Little-endian byte ordering to fill the following the table encoding instruction `srl t0, t1, t2` at memory address 0x100:

Address	Data
0x100	?
0x101	?
0x102	?
0x103	?

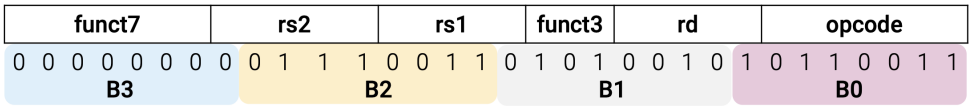
First let's recall srl, is a register-register operation of R-type format, and is thus written as such:

funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

Giving us: *(thanks to the RV32I Reference Card)*

- `funct7` $= 0$
- `rs2` $=$ `t2` $=$ `x7` $= (111)_2$
- `rs1` $=$ `t1` $=$ `x6` $= (110)_2$
- `funct3` $= (101)_2$
- `rd` $=$ `t0` $=$ `x5` $= (101)_2$
- `opcode` $= (0110011)_2$

Diving this in bytes, we get:



thus,

Address	Data
0x100	0xb3
0x101	0x52
0x102	0x73
0x103	0x00

21.1.3 Memory Read and Write Operations

Memory **read**/**write** operations are called **load**/**store**:

Operation	Instructions
Loading signed and unsigned bytes	`lb, lbu`
Loading signed and unsigned halfwords	`lh, lhu`
Loading words	`lw`
Storing bytes	`sb`
Storing halfwords	`sh`
Storing words	`sw`

Signed bytes and halfwords are sign-extended to 32 bits and then written to the destination registers. *(Widening of narrow data allows subsequent integer computation instructions to operate correctly on all 32 bits, even if the natural data types are narrower.)*

Unsigned bytes and halfwords, common in text characters and unsigned integers, are zero-extended to 32 bits and then written to the destination registers.