
ESD - Digital Systems Electronics

Lecture Notes

Riccardo Berta

2025.12.11

Contents

1 The Digital Abstraction	7
1.1 Abstraction	8
1.1.1 Discipline	10
1.1.2 The three -y	11
1.1.3 Digital Electronics	12
1.2 Digitalizazion	13
1.2.1 Sampling	15
1.2.2 Quantization	17
1.2.3 The amount of information	17
1.3 Interaction with the Physical World	18
1.4 Boolean Algebra	20
1.5 Exercises	22
2 Data Representation	23
2.1 Number Systems	23
2.1.1 Decimal System	23
2.1.2 Binary Numbers	24
2.1.3 Decimal to Binary Conversion	26
2.1.4 Hexadecimal Numbers	27
2.2 Data Units and Scaling	29
2.2.1 Bytes, nibble and words	29
2.2.2 Kilo, Mega and Giga	30
2.2.3 Estimate	30
2.3 Arithmetic with Binary Numbers	31
2.3.1 Binary Addition	31
2.3.2 Signed binary numbers	32
2.3.3 Binary Multiplication	36
2.4 Fractional Numbers	37
2.4.1 Fixed-point numbers	37
2.4.2 Floating-point numbers	39
2.5 Exercises	41
3 Logic gates and circuits	44
3.1 Logic gates	45
3.1.1 NOT gate (Inverter)	45
3.1.2 BUFFER gate	45

3.1.3	AND gate	46
3.1.4	OR gate	47
3.1.5	XOR, NAND, NOR Gates	47
3.2	Boolean Algebra	48
3.3	Digital Circuits	49
3.3.1	Combinational circuits	51
3.3.2	Sequential circuits	51
3.3.3	Combinational composition	51
3.4	Complexity	52
3.5	Simulation	53
3.6	Exercises	54
4	Beneath the abstraction	56
4.1	Logic Levels and Noise Margins	57
4.1.1	Logic levels	57
4.1.2	Noise Margins	58
4.1.3	DC Transfer Characteristics	59
4.1.4	Logic Families	61
4.2	Implementing Logic Gates	63
4.2.1	Gears and vacuum tubes	63
4.2.2	Integrated Circuits	64
4.3	Electron Devices	68
4.3.1	Semiconductors	68
4.4	The p-n junction	70
4.4.1	Diode	73
4.4.2	MOS Transistor	75
4.4.3	The switch model	77
4.4.4	CMOS	78
4.4.5	Building Integrated Circuits	79
4.5	CMOS logic gates	80
4.5.1	NOT Gate (Inverter)	80
4.5.2	NAND Gate	81
4.5.3	NOR Gate	82
4.5.4	General Rule for Inverting Logic	82
4.5.5	Transmission Gate	84
4.6	Power consumption	84

5 Boolean Algebra	85
5.1 Fundamental Definitions	85
5.1.1 Variable	86
5.1.2 Function	86
5.1.3 Complement	87
5.1.4 Literals	87
5.1.5 Products (Implicants) and Sums	88
5.2 Canonical Forms	88
5.2.1 Sum-of-Products (SOP)	89
5.2.2 Product-of-Sums (POS)	90
5.3 Axioms and Theorems	91
5.3.1 Axioms	91
5.3.2 Theorems of one variable	92
5.3.3 Theorems of several variables	94
5.3.4 Perfect induction	98
5.4 Simplifying Expressions	99
5.5 Schematic	100
5.5.1 Programmable Logic Arrays (PLAs)	102
5.5.2 Multiple-output circuits	103
5.5.3 Multi-level circuits	105
5.5.4 Bubble pushing	107
5.5.5 Beyond 0 and 1: X and Z	108
5.6 Exercises	109
6 Graphical minimization methods	111
6.1 Geometry of Logic	112
6.1.1 Logical adjacency	112
6.1.2 Unfolding	112
6.2 Karnaugh maps	114
6.2.1 Construction	116
6.2.2 Wrapping around	118
6.2.3 SOP Synthesis	119
6.2.4 POS Synthesis	125
6.2.5 Multiple Minimal Forms	126
6.3 Seven-segment Display Controller	126
6.4 Don't Cares	128
6.5 Quine–McCluskey Method	130
6.5.1 Expansion Phase	131

6.5.2 Combination Phase	134
7 Hardware Description Languages (HDL)	136
7.1 Levels of Abstraction	137
7.1.1 Structural Level (Gate-Level)	137
7.1.2 RTL Level (Data-Flow)	138
7.1.3 Algorithmic Level (Behavioral)	139
7.1.4 Specification	139
7.2 Entities	141
7.2.1 Entity construct	143
7.2.2 Architecture construct	145
7.3 VHDL Computational Model	146
7.4 Types	149
7.4.1 Bit type	150
7.4.2 Integer type	151
7.4.3 IEEE types	151
7.4.4 Resolved types	152
7.4.5 User-defined types	153
7.5 Slice and Concatenate	154
7.5.1 Slice	154
7.5.2 Concatenate	156
7.6 Logical Expressions and Truth Tables	157
7.7 Structural Description	161
7.7.1 Component description	161
7.7.2 Component instantiation	162
7.8 Process	165
7.8.1 If-then-else statement	166
7.8.2 Case statement	168
7.8.3 For statement	169
7.8.4 While statement	170
7.9 Testbench	170
7.10 Simulation and Synthesis	174
7.10.1 Simulation	175
7.10.2 Synthesis	175
8 Combinational Building Blocks	176
8.1 Multiplexers	177
8.1.1 Wider Multiplexers	177

8.1.2	Multiplexer Logic	179
8.2	Decoders	183
8.2.1	Decoder Logic	185
8.3	Adder	185
8.3.1	Half Adder	185
8.3.2	Full Adder	186
8.3.3	Ripple-carry adder	186
8.3.4	Carry-Lookahead Adder	187
8.3.5	Subtractor	192
8.4	Comparators	193
8.4.1	Equality comparator	193
8.4.2	Magnitude comparator	194
8.5	Arithmetic/Logical Unit (ALU)	194
8.5.1	Flags	195
8.5.2	Comparisons	197
8.6	Shifters	198
8.6.1	Logical shifters	198
8.6.2	Arithmetic shifters	198
8.6.3	Shifters as multiplexers	200
8.6.4	Rotators	201
8.7	Multipliers	203
8.8	Dividers	206
8.9	Floating-Point Adder	209
9	Sequential Circuits	211
9.1	Bistable Element	212
9.2	Latches	213
9.2.1	SR Latch	214
9.2.2	D Latch	217
9.3	Flip-Flops	219
9.3.1	D Flip Flop	220
9.3.2	Flip-Flops and Latches comparison	223
9.3.3	Registers	224
9.3.4	Enabled Flip-Flop	226
9.3.5	Resettable Flip-Flop	228
9.4	Transistor level implementation	230
9.5	Problematic sequential circuits	234
9.5.1	Astable circuits	234

9.5.2	Race conditions	235
9.5.3	Asynchronous circuits	237
9.6	Synchronous digital design	237
10	Timing	242
10.1	Combinational Circuits Timing	243
10.1.1	Propagation and Contamination Delays	243
10.1.2	Critical and Short Paths	245
10.1.3	Control-critical and Data-critical circuits	248
10.1.4	Hazards and Glitches	250
10.1.5	Wire delay	254
10.2	Timing of Sequential Logic	255
10.2.1	Dynamic Discipline	256
10.2.2	Aperture Time	257
10.2.3	Clock-to-Q Delays	258
10.3	Synchronous System Timing	259
10.3.1	Setup Time Constraint	260
10.3.2	Hold Time Constraint	262
10.3.3	Timing Analysis	263
10.3.4	Clock Skew	268
10.4	Timing violations	272
10.4.1	Metastability	273
10.4.2	Resolution Time	274
10.4.3	Synchronizers	275
10.4.4	Reliability	278
11	Sequential Building Blocks	279
11.1	Counters	280
11.1.1	Counter by Addition	281
11.1.2	Frequency Divider	281
11.1.3	Digitally Controlled Oscillator	282
11.2	Shift Registers	283
11.2.1	Serial-to-Parallel Conversion	284
11.2.2	Parallel-to-Serial Conversion	284
11.2.3	Scan Chains	285
11.3	Memories	286
11.3.1	Memory Arrays	289
11.3.2	Bit Cell	290

11.3.3	Dynamic Random Access Memory (DRAM)	292
11.3.4	Static Random Access Memory (SRAM)	293
11.3.5	Register Files	294
11.3.6	Volatile memories comparison	295
11.3.7	Read Only Memory (ROM)	296
11.3.8	Programmable ROM	297
11.3.9	Reprogrammable ROM	298
11.4	Programmable Hardware	299
11.4.1	Lookup Tables (LUT)	300
11.4.2	Programmable Logic Array (PLA)	301
11.4.3	Field-Programmable Gate Array (FPGA)	302
11.4.4	Comparison: FPGA vs. ASIC	307

1 The Digital Abstraction

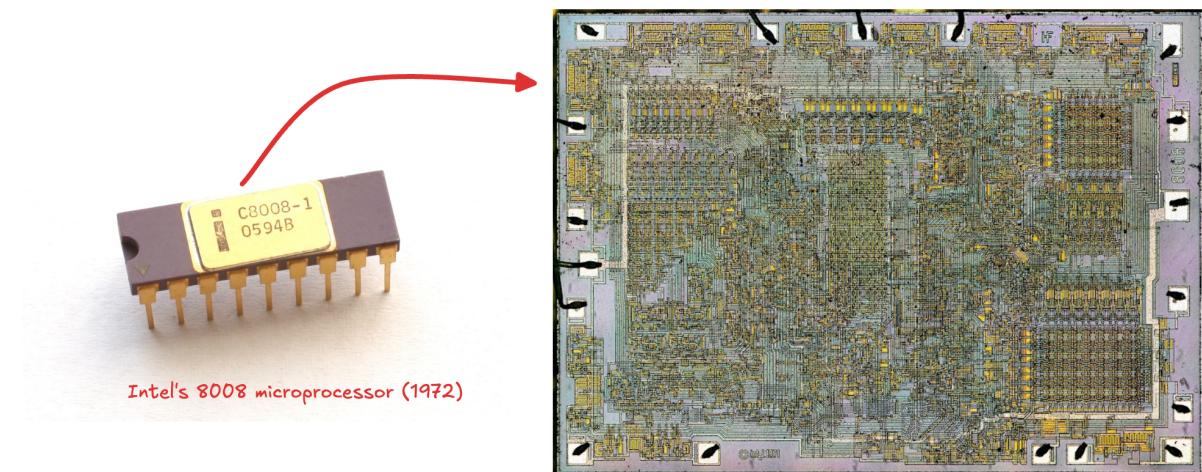
Microprocessors have revolutionized our world over the past three decades. A microprocessor is an **electronic circuit** that acts as the "brain" of a computer. It performs arithmetic, logic, control, and input/output operations by executing instructions stored in a program. Thanks to these devices, today's laptop computers have more capability than room-sized mainframes of the past. Microprocessors are not only found in personal computers: modern automobiles, for example, contain around one hundred of them, controlling everything from engine performance to safety systems. The economic impact has been equally impressive. The worldwide **semiconductor industry** has grown enormously, with sales rising from about 21 billion US dollars in 1985 to over 520 billion dollars in 2023. This growth reflects the **central role of digital technology** in every aspect of our society.

Modern digital systems are built from millions, or even billions, of **transistors**. A transistor is a tiny electronic component that can act as a **switch** or as an **amplifier**. By controlling the flow of current, it serves as the fundamental building block of all electronic circuits. However, the **complexity** of such systems is staggering:

- The first microprocessor, Intel 4004 (1971), had 2.300 transistors.
- The first 32-bit microprocessor, Motorola 68000 (1979), had 68.000 transistors.
- The first 64-bit microprocessor, MIPS R4000 (1991), had 1.35 million transistors.
- The first Pentium processor, Intel Pentium (1993), had 3.1 million transistors.
- The first dual-core processor, AMD Athlon 64 X2 (2005), had 233.2 million transistors.
- The first quad-core processor, Intel Core 2 Quad (2006), had 582 million transistors.
- The first eight-core processor, AMD FX-8150 (2011), had 1.2 billion transistors.

- The first ten-core processor, Intel Core i7-6950X (2016), had 3.2 billion transistors.
- The first twenty-eight-core processor, Intel Xeon W-3175X (2019), had 8.6 billion transistors.
- ...

At the lowest level, the behavior of transistors could be described using **semiconductor device equations**: for example, quantum mechanics and Maxwell's equations for electromagnetic fields, or the drift–diffusion equations that model how electrons and holes move inside the semiconductor material. In principle, one could attempt to write down a huge system of partial differential equations to describe the movement of charges in every transistor. But for a processor with billions of devices, this would mean billions of coupled equations, clearly impossible for any human (or even any computer) to solve in practice. For example, this is a picture of the internal of the Intel's 8008 microprocessor (1972) is the ancestor of modern processors that you may be using right now:



It is composed of only 3.500 transistors, this should give you an idea of the scale of complexity involved in modern microprocessors. Therefore, the key challenge is to **manage complexity**. To design and understand digital systems (like microprocessors), we must learn to think in **layers of abstraction**, focusing on the essential features at each level without being overwhelmed by unnecessary details. This strategy enables engineers to build reliable and sophisticated digital systems while keeping the design process under control.

1.1 Abstraction

The most powerful idea in digital design is **abstraction**: the practice of **hiding details** when they are not important for the task at hand. Without abstraction, the enormous complexity

of modern digital systems would be unmanageable. The following figure shows the stack of abstractions in a typical digital system:

	Software	Operating systems, Programs
	Architecture	Datapath, controllers, instructions, registers
	Logic	Adders, memories
	Digital Circuits	AND, NOT gates
	Analog Circuits	Amplifiers, filters
	Devices	Diodes, transistors
	Physics	Electrons

ESD

At the lowest level, everything is governed by **physics**. The motion of electrons is described by quantum mechanics and Maxwell's equations. These equations are essential for understanding semiconductor behavior, but we do not need to deal with them directly when building circuits.

Instead, we step up one level to **devices**. Here, we work with transistors and diodes. A transistor has well-defined terminals and can be **modeled** as a controllable switch or amplifier. This allows us to ignore the messy details of individual electron movement. From devices, we construct **analog circuits** such as amplifiers and filters, which process continuous voltages and currents.

Narrowing further, we focus on **digital circuits**, where voltages are restricted to discrete ranges representing binary values ("0" and "1"). With these digital building blocks, we can design logic gates such as AND, OR, and NOT. Combining logic gates leads us to more **complex components**, including adders, multiplexers, and memories. At this stage, the designer can think in terms of bits and operations rather than individual transistors.

The next step is the **architecture** level, where datapaths, control units, and pipelines are organized to carry out the fundamental operations of a processor. The architecture level provides the programmer's view of the system: instruction sets, registers, and addressing modes.

Beyond hardware, **operating systems** manage resources such as memory, files, and devices, hiding the complexity of hardware from software developers. At the very top, **application**

software allows users to interact with the machine in a way that is completely independent of the underlying physics and electronics.

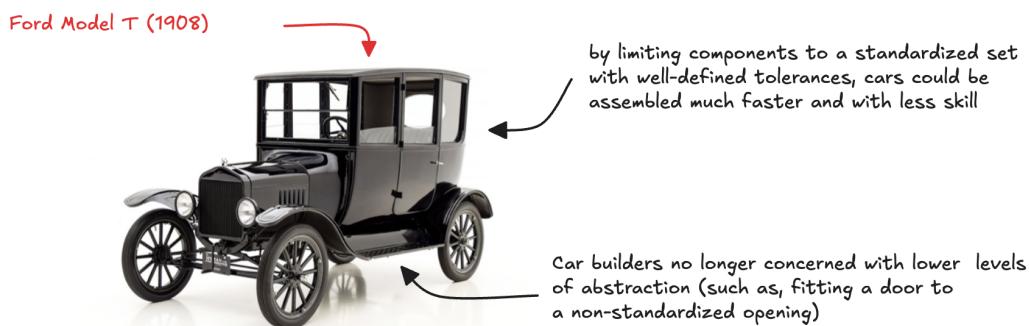
This hierarchical approach means that at each level **we can focus only on the concepts that matter**, while trusting that the lower layers will behave as specified. Abstraction is therefore the essential tool that enables engineers to design, understand, and use digital systems without drowning in unnecessary details.

1.1.1 Discipline

In order to manage complexity effectively, engineers must exercise **discipline** in their design processes. This means **intentionally restricting the design choices available**, so that they can work more productively at a higher level of abstraction. Without discipline, systems would become too complex to design, maintain, or scale.

A classic example of this principle comes from the automotive industry. Before 1908, cars were handcrafted by skilled workers. **Each vehicle was unique, and parts were often adjusted individually to fit together**. This approach was not only **time-consuming**, but also **expensive**, since production depended heavily on craftsmanship and could not easily be scaled.

Henry Ford revolutionized car manufacturing with the introduction of the Model T (1908). Instead of relying on handcrafted uniqueness, he focused on mass production.



Two key innovations were critical:

1. **Interchangeable parts**: components were manufactured according to standardized dimensions and tolerances. This meant that a door or a wheel from one car could fit perfectly onto another car of the same model.
2. **Moving assembly lines**: cars were assembled in a systematic process, where workers specialized in performing a small number of tasks repeatedly. This made production faster, cheaper, and less dependent on individual skill.

By enforcing these **restrictions**, Ford introduced discipline into car design and manufacturing. As a result, car builders no longer had to worry about **low-level issues** such as adjusting each door to a non-standard opening. Instead, they could focus on **higher-level goals**: efficiency, reliability, and cost reduction.

Ford's approach became so influential that he is often quoted with his famous saying: "Any customer can have a car painted any colour that he wants so long as it is black".

This statement illustrates the idea of limiting choices for the sake of productivity. While customers had fewer options, the standardized approach allowed the industry to achieve unprecedented levels of mass production.

In digital systems, we apply the same logic. By **restricting ourselves to well-defined design rules and abstractions**, we can manage complexity and build systems with millions or billions of transistors. Without such discipline, it would be impossible to coordinate the design of modern digital circuits, just as it would have been impossible to build affordable cars without Ford's innovations.

1.1.2 The three -y

A concrete method to apply the principles of abstraction and discipline in digital design is the use of the **three-y** framework: Hierarchy, Modularity, and Regularity. These principles provide a systematic way to design large systems by breaking them down into simpler, more manageable parts.

1. **Hierarchy** means dividing a system into modules, and then further subdividing those modules into smaller pieces until each part is easy to understand. By organizing a design into **layers of detail**, engineers can focus on one level at a time without being overwhelmed by the full complexity.
2. **Modularity** requires that each module has a clear and well-defined function, along with precise interfaces to connect with other modules. This ensures that modules can be designed, tested, and reused independently, without causing unexpected side effects when integrated into a larger system.
3. **Regularity** emphasizes uniformity and reuse. When common modules are used repeatedly, the number of distinct designs that must be created is reduced. This not only saves design effort, but also improves reliability, since the same module can be tested and verified once and then applied many times.

The Ford Model T illustrates these three principles very clearly. A car can be broken down into major components like the chassis, engine, and seats. The engine itself contains smaller

parts such as cylinders, carburetor, and cooling system. The carburetor can be further divided into fuel and air intakes, a throttle, and so forth. This recursive breakdown mirrors the idea of hierarchical decomposition in design. Consider the coupling nut that holds the fuel feed line to the intake elbow. Its diameter, thread pitch, and torque are standardized. Because its function and interface are well-defined, the nut can be manufactured, replaced, and tightened with a standard wrench. This is a perfect analogy for modularity. A standardized nut can be purchased from many different suppliers, as long as it meets the correct specification. This shows how reuse and uniformity reduce the need to reinvent components, saving time and resources.

In digital design, these same principles apply. A microprocessor, for example, is organized hierarchically into modules such as arithmetic circuits, memory units, and control logic. Each module is defined with clear interfaces, ensuring modularity. And regularity ensures that structures like logic gates or memory cells can be replicated billions of times on a chip without requiring unique designs for each instance.

By applying Hierarchy, Modularity, and Regularity, engineers can build systems of enormous complexity in a manageable, reliable, and efficient way.

1.1.3 Digital Electronics

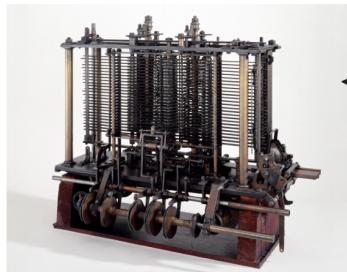
In digital electronics those concepts are applied through the decision to represent information with **discrete values** rather than **continuous ones**.

In an analog circuit, voltages can vary continuously over a range of values. While this can describe physical phenomena with great precision, it also makes circuits harder to design and more sensitive to noise. By contrast, digital circuits restrict voltages to discrete ranges that represent logical values. This restriction makes digital circuits a subset of analog circuits, but one that is much simpler to design and reason about.

This simplification is extremely powerful. By limiting ourselves to digital circuits, we can combine components into sophisticated systems that **ultimately outperform their analog counterparts** in many applications. The shift from analog to digital technologies has transformed our world: televisions, cameras, and telephones that were once analog are now almost entirely digital, with improved performance, flexibility, and reliability.

Digital systems thus represent information with **discrete-valued variables**. An early attempt to harness this idea can be found in Charles Babbage's Analytical Engine (1834–1871):

Charles Babbage's Analytical Engine (1834–1871)



A mechanical device to perform calculations. It represented numbers using discrete positions of gears, reducing continuous mechanical motion to a finite set of states that could be combined to perform complex calculations.

Although mechanical, it embodied a digital representation: gears with ten discrete positions labeled 0 through 9. This illustrates that the abstraction of information into discrete states predates electronics, even if practical digital systems only became feasible with the advent of electronic devices.

Modern electronic digital systems typically use a **binary representation**. In this scheme, a high voltage corresponds to a "1" and a low voltage corresponds to a "0". Each **binary digit** is called a **bit**. Why only two states? The answer is simplicity and robustness: it is much easier to distinguish between two voltage levels than between ten or more.

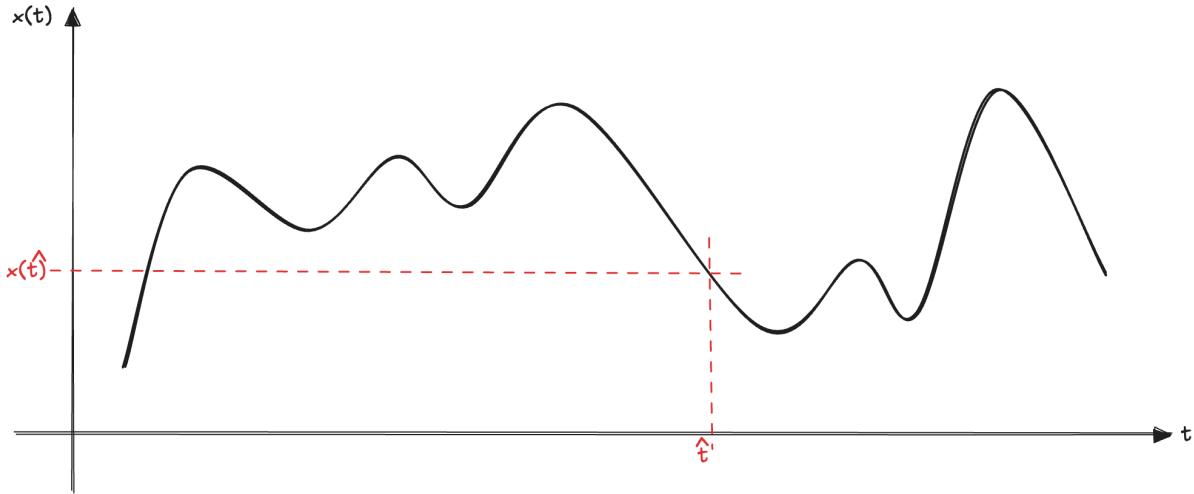
The digital abstraction, therefore, allows engineers to design, build, and understand extremely complex systems by thinking in terms of "0" and "1" rather than dealing with the infinite range of values found in the analog world.

1.2 Digitalization

Most physical phenomena **vary continuously over time**. For example, the temperature of a room, the intensity of light, or the force applied to an object can all take on an infinite number of values. Mathematically, we can represent such a **signal** as a function:

$$x(t)$$

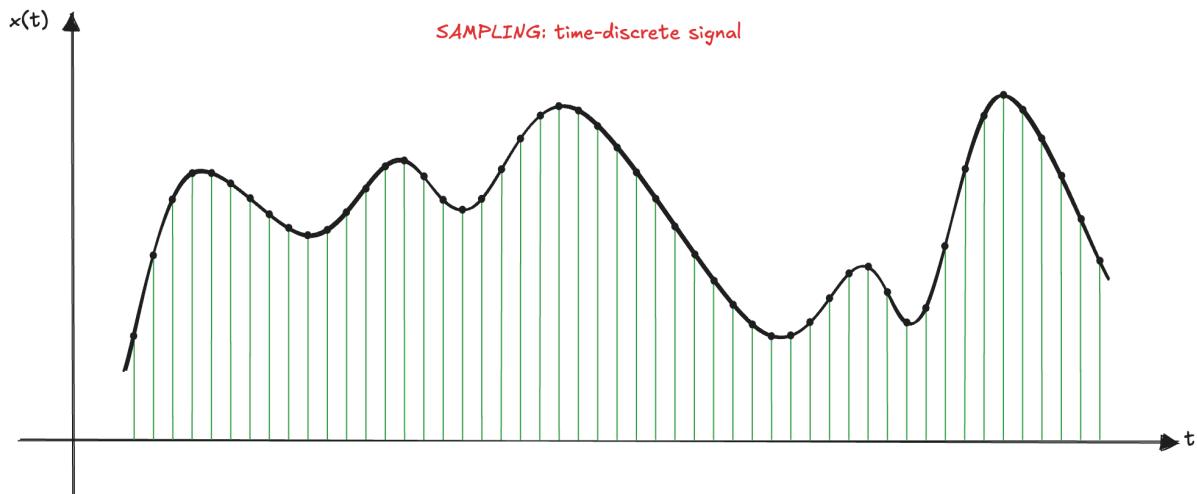
which is continuous in **time** and **amplitude**. Strictly speaking, at the quantum scale, nature is not perfectly continuous: energy, charge, and even light come in discrete packets (quanta). However, these quantum steps are so small compared to human-scale measurements that they can be safely ignored. For several practical purposes in engineering, signals like voltage, force, or temperature can be treated as continuous functions.

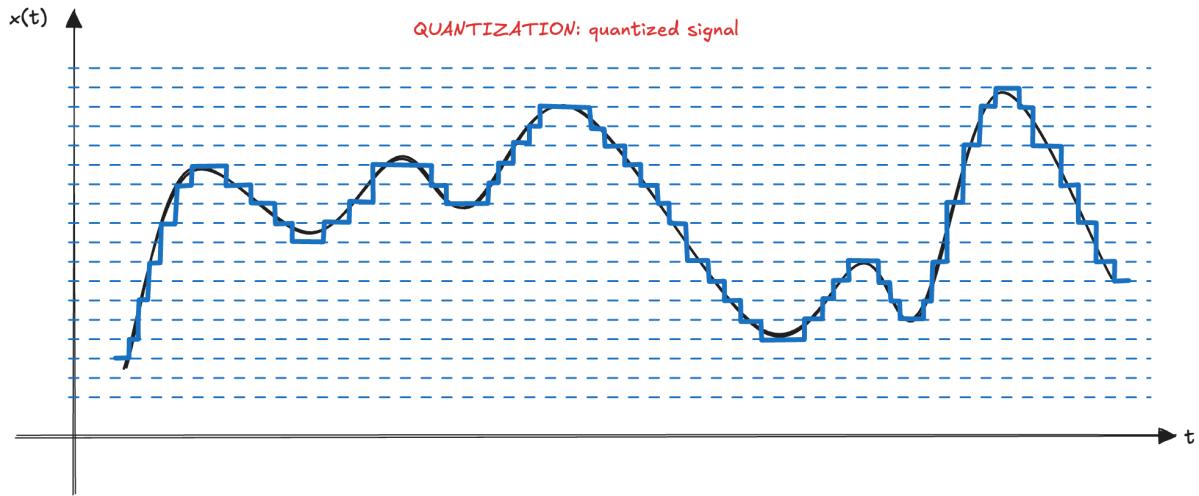


For describing nature, the analog representation seems the most natural and effective, because it preserves the continuity of the physical world. An analog voltage, for instance, can vary smoothly to reflect all possible values of a measured signal.

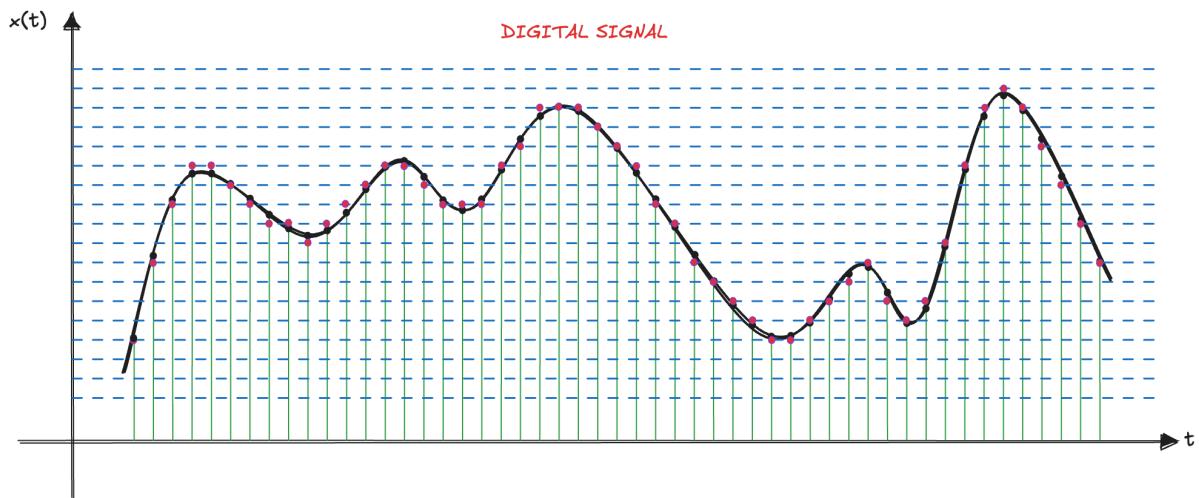
However, digital systems require us to work with discrete values in order to manage the complexity of processing and storing information. . To bridge this gap, we need a way to convert continuous quantities into digital ones. This is the role of an **Analog-to-Digital Converter (ADC)**. In order to represent an analog signal digitally, we must digitize it in two dimensions:

- **in time**, by selecting specific instants at which the signal is measured (a process called **sampling**),
- **in amplitude**, by mapping each measured value to the nearest level in a finite set of possibilities (a process called **quantization**)





Through sampling and quantization, a continuous signal is transformed into a sequence of numbers that a digital circuit can store and process:



This digital representation makes it possible to apply algorithms, store data efficiently, and transmit information reliably, even though it always involves some degree of approximation compared to the original analog signal.

1.2.1 Sampling

Sampling is the process of reducing a continuous-time signal into a **discrete-time signal** by measuring its value at specific instants in time. A **sample** is one of these measured values. If the signal is measured every T seconds, then:

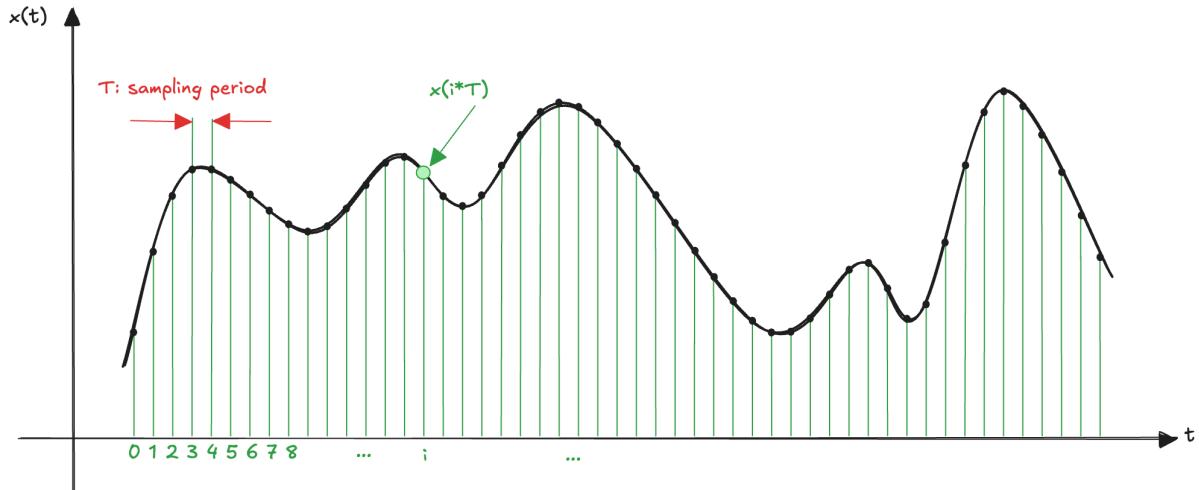
$$t_i = i * T \quad \text{for } i = 0, 1, 2, \dots$$

and the corresponding samples are:

$$\hat{x}[i] = x(t_i) = x(i * T)$$

T is called the **sampling period** (the time between two consecutive samples) and its reciprocal is the **sampling frequency**.

$$f_s = \frac{1}{T} \quad [\text{samples per second, or Hertz}]$$



For example, if we measure the temperature of a room every $T=1$ second, then the sampling frequency is:

$$f_s = \frac{1}{T} = \frac{1}{1\text{s}} = 1\text{Hz}$$

If instead we sample every $T=0.001\text{s}$ (1ms), the frequency becomes:

$$f_s = \frac{1}{0.001\text{s}} = 1000\text{ Hz}$$

However, one key question arises: **which T should we use?** Choosing the right sampling period is essential. If the sampling frequency is too low, the discrete signal will not capture the variations of the continuous one, leading to loss of information (a phenomenon called aliasing). According to the **Nyquist–Shannon sampling theorem**, in order to reconstruct the signal without losing information, the sampling frequency must satisfy:

$$f_s \geq 2f_{\max}$$

where f_{\max} is the highest frequency present in the signal.

1.2.2 Quantization

The continuous range of amplitudes of a signal must be mapped to a finite set of discrete values. Formally, if the input signal has amplitude values in a range

$$[x_{\min}, x_{\max}]$$

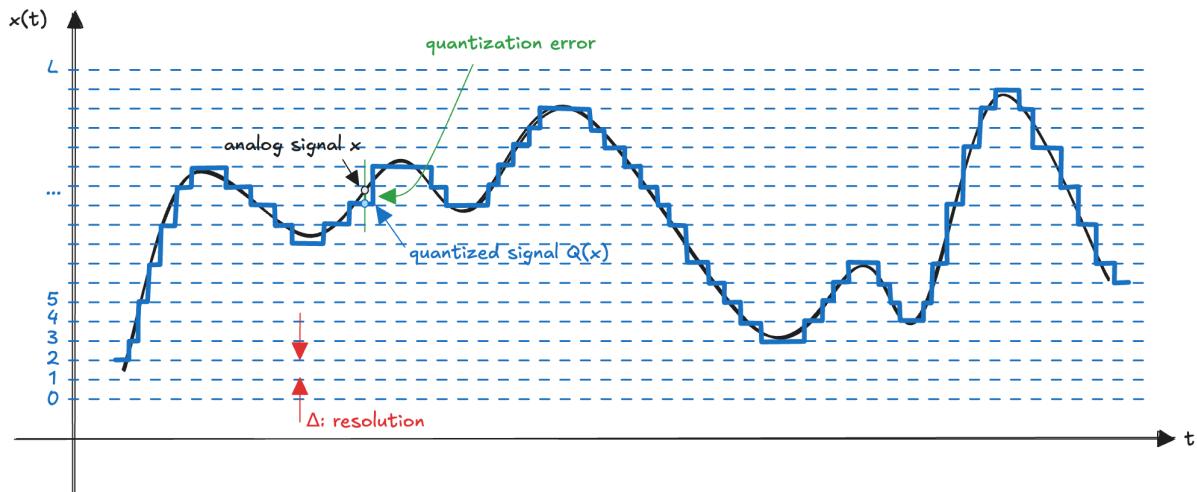
and we choose to represent it with L quantization levels, then each level corresponds to an interval of width:

$$\Delta = \frac{x_{\max} - x_{\min}}{L}$$

where Δ is called the **quantization step size (or resolution)**.

Each measured value is then rounded to the nearest discrete level:

$$Q(x) = \text{round}\left(\frac{x - x_{\min}}{\Delta}\right) \cdot \Delta + x_{\min}$$



In this way, we introduce an error, since the quantized value is only an approximation of the true input:

$$e_q = x - Q(x)$$

The error is bounded by:

$$-\frac{\Delta}{2} \leq e_q \leq +\frac{\Delta}{2}$$

1.2.3 The amount of information

After the continuous signal is sampled in time and quantized in amplitude, the result is a **sequence of discrete values** that can be stored and processed by a digital system. But a natural

question arises: **how many discrete values are needed** to represent the information? To answer this, we must define the **amount of information** carried by a variable. This depends on the **size of the alphabet S** (the number of different symbols available, e.g. 2 for binary, 10 for decimal, 26 for letters) and on the **word length D** (how many symbols we place side by side to form one encoding). The total number N of distinct encodings we can create is:

$$S^D$$

If we want to represent a set of N distinct objects, we must ensure that:

$$S^D \geq N$$

In most cases, we know how many objects N we want to represent, and the size of the alphabet S (for example, binary symbols S=2) and we then need to find the **minimum word length D**, that we call **amount of information**:

$$D \geq \log_S N$$

When the alphabet is binary (S=2), the unit of measurement is the **bit (binary digit)**:

$$D = \log_2 N \quad [\text{bits}]$$

The amount of information of a single binary variable is:

$$D = \log_2 2 = 1 \text{ bit}$$

To represent the 75 alphanumeric characters (uppercase, lowercase, digits, and some special characters), we need:

$$\log_2 75 \approx 6.23$$

which means at least 7 bits are required. This is why the original **American Standard Code for Information Interchange (ASCII)** was designed with 7 bits per character.

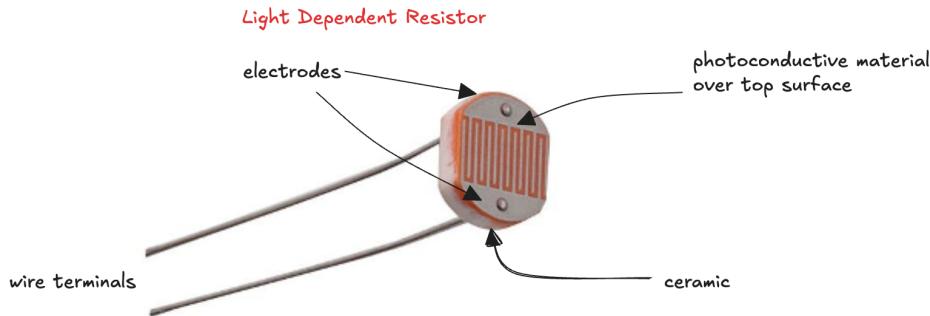
A continuous signal theoretically contains an **infinite amount of information**, since it can take an infinite number of values. However, in practice, physical limitations such as noise and measurement error restrict the effective resolution. For most continuous signals, this practical information content is limited to about 10 to 16 bits.

By quantifying information in this way, we establish a link between the abstract concept of information and the physical requirements of digital systems.

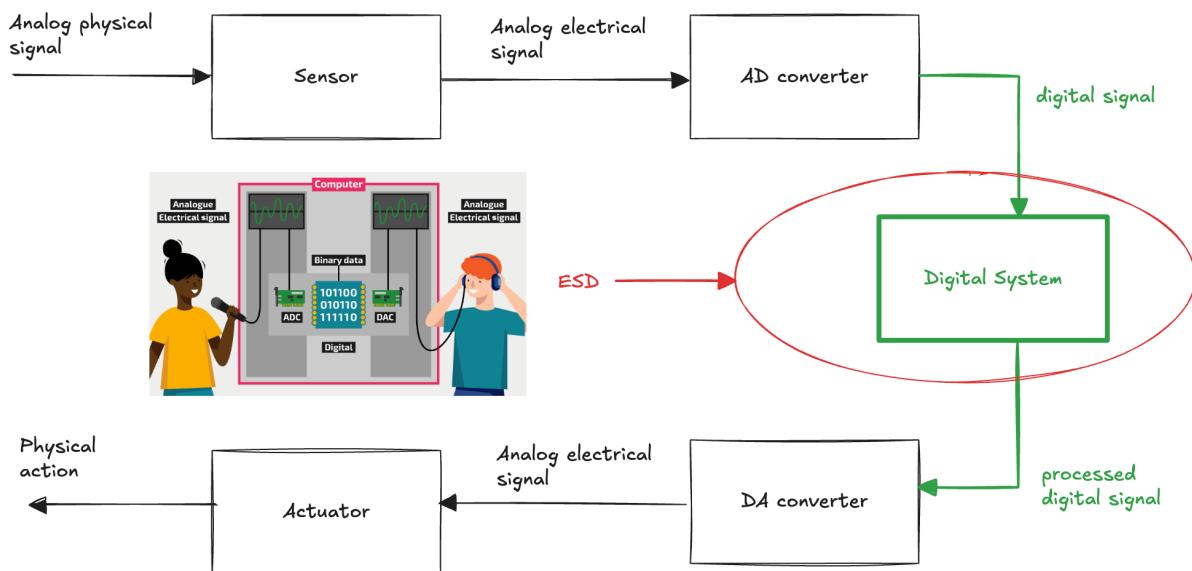
1.3 Interaction with the Physical World

To interact with the **physical world**, digital systems need devices that can sense physical phenomena and produce electrical signals that represent them. These devices are called **sensors**. A sensor converts one form of energy into another. It takes a non-electrical signal (like heat, light,

pressure, or motion) and transforms it into an electrical signal that can be processed. For example, a light sensor is made of a light-sensitive resistor, often called an LDR (Light Dependent Resistor). Its resistance decreases as the intensity of light increases. This change in resistance can be measured as a voltage, providing an electrical representation of the light level:



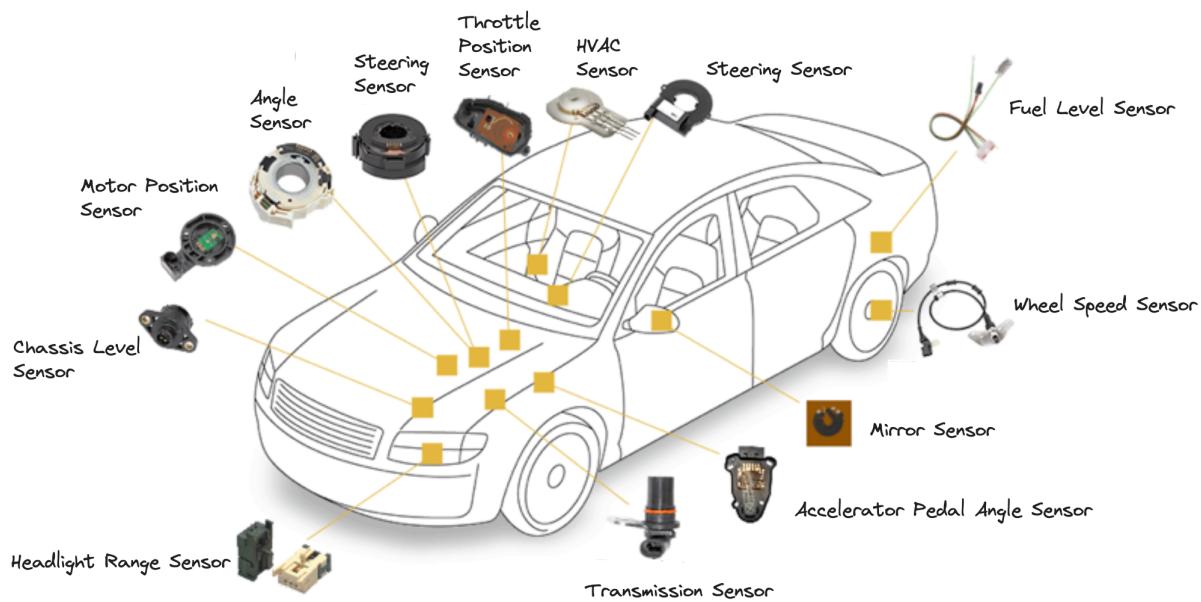
Once a sensor produces an analog signal, the system must digitize it to use it in a digital system. This involves an **Analog-to-Digital Converter (ADC)**, which transforms the continuous voltage into a sequence of digital values. These values can then be processed by a digital processing unit such as a processor or a custom designed digital circuit. Then the processed signal needs to be converted back to the real world (for example, to drive a loudspeaker, a motor, or a display) and a **Digital-to-Analog Converter (DAC)** is used to transform the digital output into a continuous analog signal again that can be used by an actuator to perform a physical action:



For example, modern cars are full of sensors:

- Temperature sensors for engine control

- Position sensors for throttle, accelerator pedal, and steering
- Speed sensors for wheels and transmission
- Light and rain sensors for headlights and wipers
- Pressure sensors for tires and fuel systems
-



All these sensors produce analog signals that are digitized, processed, and used by the vehicle's electronic control units to optimize performance, improve safety, and enhance comfort.

1.4 Boolean Algebra

When working with digital systems, it is often more convenient to leave behind the messy physical world of voltages, currents, and transistors, and instead **operate in the clean and abstract world of mathematics**. The most important mathematical tool for digital design is **Boolean algebra**, named to honor the mathematician **George Boole**, who first defined its principles in the mid-19th century.



George Boole, 1815–1864

Born to working-class parents and unable to afford a formal education, Boole taught himself mathematics and joined the faculty of Queen's College in Ireland. He wrote "An Investigation of the Laws of Thought" (1854), which introduced binary variables and the three fundamental logic operations: AND, OR, and NOT.

Boolean algebra is a form of **discrete mathematics** that deals with variables that can take only two possible values. In digital electronics, these values are naturally associated with the binary digits 0 and 1, which correspond to the two allowed voltage ranges in a digital circuit. **This correspondence between mathematics and hardware is the essence of the digital abstraction:** once we agree that "low voltage" means 0 and "high voltage" means 1, we can reason about circuits entirely in terms of 0s and 1s, **without worrying about electrons or physical devices.**

The beauty of Boolean algebra is that it frees designers from implementation details. A Boolean variable may be physically represented by:

- a voltage level in a transistor circuit,
- the presence or absence of a gear tooth in a mechanical system,
- or even the level of liquid in a hydraulic device.

From the perspective of Boolean algebra, all of these are equivalent, as long as the system consistently distinguishes between two states. This means that **an hardware designer can use Boolean algebra to model, analyze, and optimize circuits.** For example, Boolean expressions can be simplified using algebraic laws such as De Morgan's Theorems or the Distributive Law, which help reduce the number of gates needed in a design. The same principle that allows a programmer to remain abstract also empowers a hardware engineer to manipulate logical relationships efficiently.

It is important to recognize, however, that abstraction is a two-edged sword. While Boolean algebra allows us to forget about the messy analog world, **understanding some of those details can still be useful.** For example, a circuit designer who understands transistor-level characteristics can optimize power consumption or speed.

In summary, Boolean algebra is the **mathematical foundation of digital design.** It gives us a simple language of 0s and 1s, and a set of algebraic rules, that make it possible to design and reason about digital systems at a high level of abstraction. Without it, the complexity of modern electronics (millions or billions of transistors in a single microprocessor) would be unmanageable.

1.5 Exercises

1 - An analog voltage is in the range of 0–5V. If it can be measured with an accuracy of ± 50 mV, at most how many bits of information does it convey?

According to the text we know:

- the analog voltage in range: 0V to 5V
- the accuracy is $\pm 50\text{mV} = \pm 0.05\text{V}$

This means the smallest resolvable interval is $\Delta V = 0.1\text{V}$ (because $\pm 0.05\text{V}$ corresponds to a resolution step of 0.1V)

We count how many distinct intervals fit in the range.

$$N = \frac{5 - 0}{0.1} = 50$$

So we can distinguish 50 different levels. Then we compute the number of bits, using the amount of information definition:

$$D = \log_2 N = \log_2 50 \approx 5.64\text{bits}$$

The analog voltage conveys about 5.64 bits of information.

2 - A classroom has an old clock on the wall whose minute hand broke off. (a) If you can read the hour hand to the nearest 15 minutes, how many bits of information does the clock convey about the time? (b) If you know whether it is before or after noon, how many additional bits of information do you know about the time?

According to the text we know:

- one hour is divided into 4 segments of 15 minutes
- in 12 hours, the hand can point to: $12 * 4 = 48$ distinct positions

The number of bits of information is: $D = \log_2 48 \approx 5.58\text{bits}$

If we also know whether it is before or after noon, that doubles the number of possibilities: $48 * 2 = 96$

So the extra information is:

$$D = \log_2 96 - \log_2 48 = \log_2 \left(\frac{96}{48} \right) = \log_2 2 = 1 \text{ bit}$$

The clock conveys about 5.58 bits and knowing AM/PM gives 1 additional bit.

2 Data Representation

Digital systems process and store information in a form that must ultimately be expressed as numbers. Whether it is text, images, audio, or other signals, all data inside a computer or an digital system is represented using **numbers**. Understanding how different types of numbers are expressed and manipulated is therefore a fundamental step in digital design.

2.1 Number Systems

A number system defines **how digits are combined to represent quantities**, much like the familiar decimal system that we use in everyday life.

While humans are most comfortable with the decimal (base-10) system, digital hardware relies on the **binary (base-2) system**, in which all values are expressed using only two symbols. This is because it is much easier and more reliable to distinguish between two electrical states (such as high and low voltages) than between ten or more. Humans, instead have ten fingers, which naturally leads to a base-10 counting system.

Beyond binary and decimal, other number systems such as **hexadecimal (base-16)** play an important role in digital design. Hexadecimal serve as convenient shorthand for binary numbers, allowing engineers to read and manipulate long binary strings more easily.

Every operation in a digital system (from arithmetic and logic to memory addressing and data transmission) relies on these representations.

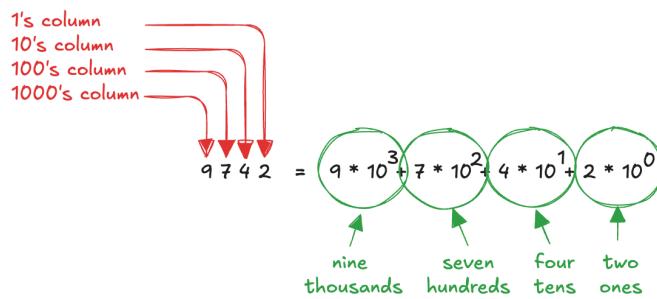
2.1.1 Decimal System

From an early age, we are taught to count and perform arithmetic in the decimal number system. This system is based on the number ten, a natural choice since humans have ten fingers, which historically guided the development of counting methods. The decimal system uses **ten distinct digits**:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Decimal digits can be combined to form larger numbers. The value of a digit depends not only on the digit itself but also on its position within the number (**positional notation**). Each position, or column, represents a power of ten.

For example, consider the number:



Here, the rightmost digit (2) represents two ones, the next digit (4) represents four tens, the next (7) represents seven hundreds, and the leftmost digit (9) represents nine thousands. Each step to the left increases the column weight by a factor of ten.

Because each column is weighted by a power of ten, decimal numbers are referred to as being in **base 10**. In general, a base indicates **how many distinct digits a number system uses** before carrying over to the next column. In decimal, the base is 10, so after the digit 9 comes the carry-over to the next column, forming 10.

An important property of the decimal system is that an N-digit decimal number can represent one of 10^N distinct values. For example:

- a 1-digit number ranges from 0 to 9, giving 10 possibilities.
- a 2-digit number ranges from 0 to 99, giving 100 possibilities.
- a 3-digit number ranges from 0 to 999, giving 1000 possibilities.

Thus, in general, an N-digit decimal number represents values in the range:

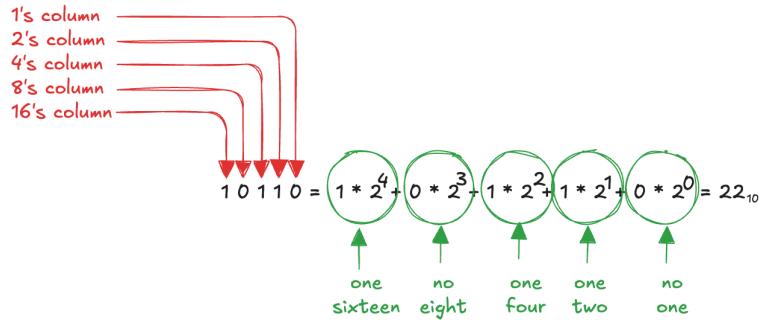
$$[0, (10^N - 1)]$$

By reviewing decimal numbers and their positional representation, we can smoothly transition to other number systems that are better suited to digital electronics.

2.1.2 Binary Numbers

A binary number is made up of **bits** (short for binary digits), each of which can take only one of two possible values: 0 or 1. This simplicity makes binary ideal for electronic systems, where **two distinct voltage levels** can be used to represent the two states reliably.

Just like decimal numbers use powers of 10, binary numbers **use powers of 2**. Each column of a binary number represents a power of 2, and each position has twice the weight of the previous one. The column weights in binary are therefore: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...



This shows how binary numbers can be directly converted to decimal by summing the values of the powers of two where there are 1s.

The number of distinct values that can be represented depends on the number of bits. An N-bit binary number can represent exactly 2^N different values. The table below shows this relationship:

	1-Bit	2-Bit	3-Bit	Decimal
	0	00	000	0
	1	01	001	1
		10	010	2
		11	011	3
			100	4
			101	5
			110	6
			111	7

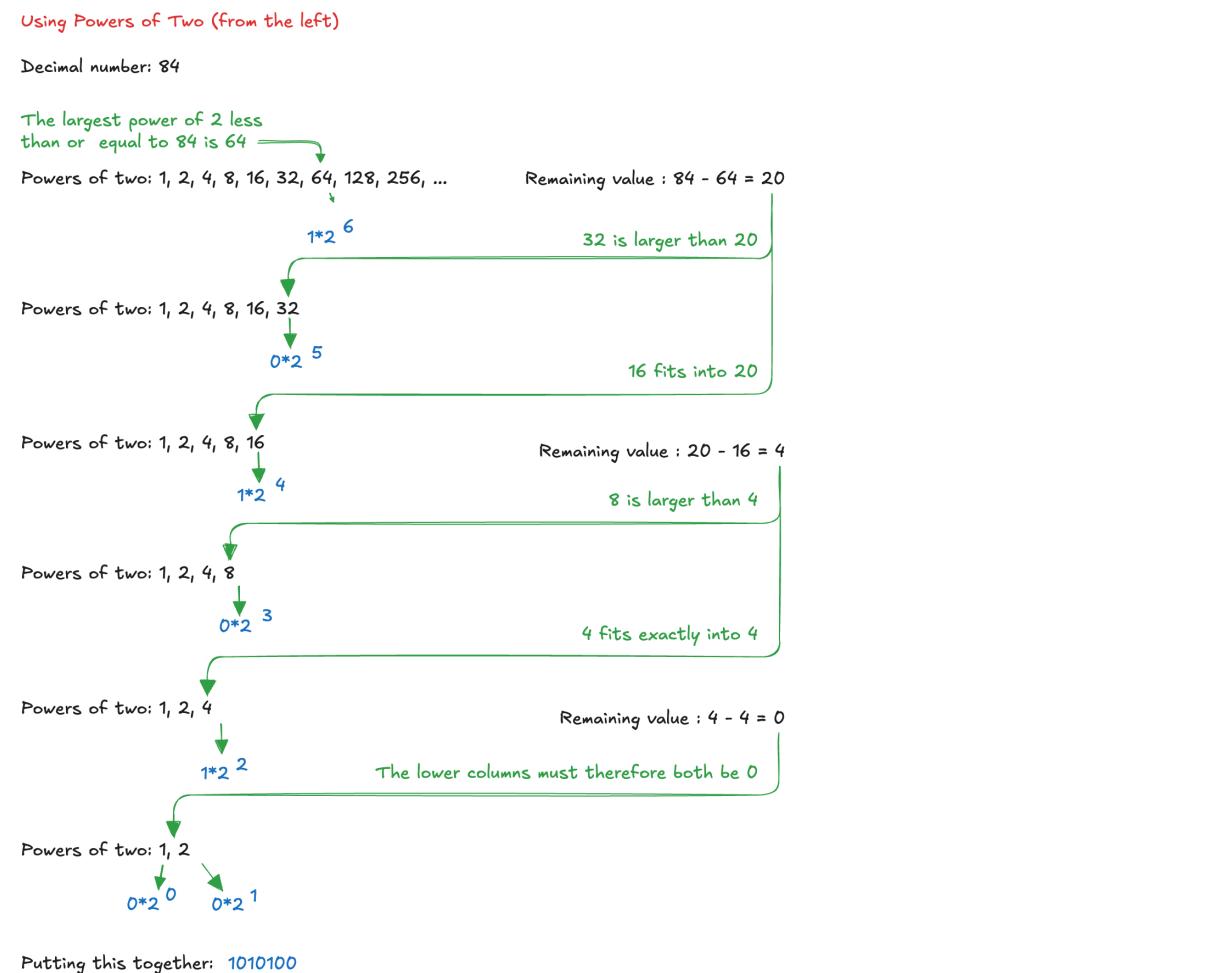
This illustrates how increasing the number of bits exponentially increases the number of representable values. For instance, with 8 bits, we can represent 256 distinct values, ranging from 0 to 255.

Binary numbers are the foundation of all digital electronics. Every operation inside a digital circuit relies on binary representation. By understanding how binary numbers work, we can begin to understand how larger and more complex data structures are built and manipulated within digital systems.

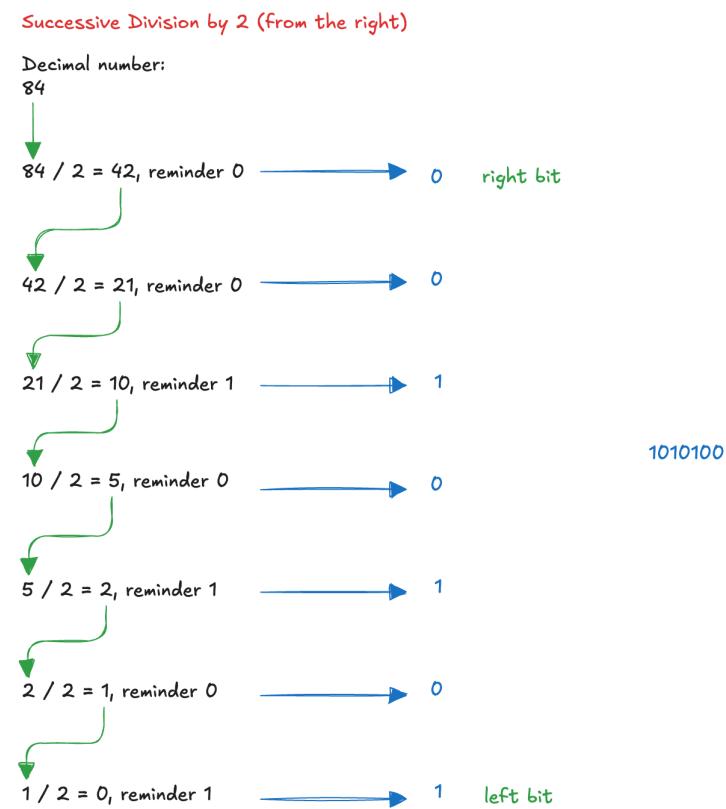
2.1.3 Decimal to Binary Conversion

Digital systems operate in binary, however humans have problems working with binary numbers directly. Therefore, it is essential to understand **how to convert numbers** from the decimal system (base 10) into binary (base 2). There are two common approaches to perform this conversion.

We can **work from the left** using powers of two. This method relies on identifying the largest power of 2 less than or equal to the decimal number and then subtracting it until the number is reduced to zero. For example, to convert 84 into binary:



We can also **work from the right**. This method repeatedly divides the decimal number by 2 and records the remainders, which correspond to the binary digits from right to left. Using the same example:



Both methods give the same answer. The powers of two approach highlights the positional weights of binary digits, while the successive division method is straightforward and often used in manual conversions or algorithms.

2.1.4 Hexadecimal Numbers

Writing binary numbers is often cumbersome. Long strings of 0s and 1s are not only **tedious** to read and write, but also **prone to error**. However, conversion between binary and decimal number is not straightforward. For this reason, engineers commonly use the **hexadecimal number system** as a shorthand for binary.

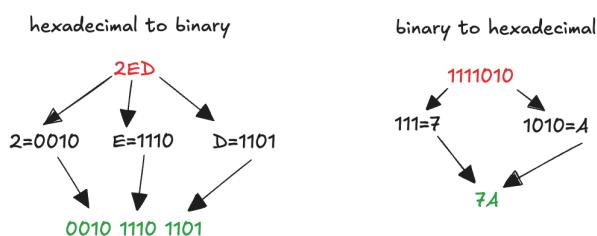
The hexadecimal system is **base 16**, meaning that each digit represents a value from 0 to 15. To cover all 16 possibilities, hexadecimal uses the decimal digits 0 through 9 together with the letters A to F:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Each hexadecimal digit **corresponds exactly to a group of four binary bits**. Since four bits can represent $2^4 = 16$ different values, this mapping is natural and efficient:

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

This direct correspondence makes conversion between hexadecimal and binary particularly easy. To convert hexadecimal to binary, we simply **replace each hexadecimal digit with its four-bit binary equivalent**. To convert binary to hexadecimal, we **group the binary digits into groups of four, starting from the right, and replace each group with the corresponding hexadecimal digit**. For example:



Hexadecimal notation dramatically reduces the length of binary numbers and makes them easier to read and debug. For instance, the 32-bit binary value:

1101011010011101010110101011001

can be written much more compactly as:

D69EAD59

This compactness is why hexadecimal is widely used in programming, digital design, and computer architecture.

2.2 Data Units and Scaling

Data is not only represented in single bits, but also **grouped into larger units** that make storage and processing more efficient.

2.2.1 Bytes, nibble and words

A **byte** is a group of **eight bits**. Since each bit can take two values, a byte can represent one of $2^8 = 256$ different possibilities. For this reason, the size of objects stored in memory is **usually measured in bytes** rather than individual bits.

A smaller grouping of four bits is called a **nibble**. A nibble can represent one of $2^4 = 16$ different values, which corresponds exactly to a single hexadecimal digit. Although the term nibble is less commonly used today, it remains useful when working with hexadecimal representations.

Microprocessors usually handle data in **larger chunks called words**. The size of a word depends on the **architecture** of the processor. For example, most modern computers use 64-bit processors, meaning that the processor can handle 64-bit words in a single operation. In the past, 32-bit processors were standard, and even earlier architectures supported 16-bit or 8-bit words.

Within a binary number, not all bits carry the same importance.

- the bit in the **rightmost position** is the **least significant bit (LSB)**, since it represents the smallest weight (the ones place);
- the bit in the **leftmost position** is the **most significant bit (MSB)**, since it carries the highest weight.



The same terminology applies at the byte level: within a multi-byte word, the **least significant byte (LSB)** is the one containing the lowest-order bits, while the **most significant byte (MSB)** contains the highest-order bits.

2.2.2 Kilo, Mega and Giga

Data sizes are often expressed in **multiples of bytes** using familiar prefixes such as **kilo**, **mega**, and **giga**. Although these prefixes originate from the decimal metric system, in computer science they are usually associated with powers of two:

- a **kilobyte (KB)** is defined as $2^{10} = 1024$ bytes. This value is close to $10^3 = 1000$, so the prefix kilo is used.
- a **megabyte (MB)** corresponds to $2^{20} = 1.048.576$ bytes, which is approximately equal to one million (10^6) bytes.
- a **gigabyte (GB)** is defined as $2^{30} = 1.073.741.824$ bytes, which is close to one billion (10^9) bytes.

This works because the powers of two line up nicely with powers of ten. However, formally, the **IEC (International Electrotechnical Commission)** distinguish between **kibibyte (KiB** = 1024 B), **mebibyte (MiB** = 1024^2 B), **gibibyte (GiB** = 1024^3 B) and **kilobyte (KB** = 1000 B), **megabyte (MB** = 1000^2), **gigabyte (GB** = 1000^3). In this way the distinction between base 2 and base 10 is clear. However, in common usage KB, MB, and GB are still often used with base 2.

2.2.3 Estimate

When working with digital systems, big numbers come up all the time (like memory sizes or data limits). Instead of always reaching for a calculator, we can **estimate them quickly** if we remember just the smaller powers of two:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512

These are easy to remember because each number is just double the previous one. Now suppose we want to know the value of 2^{24} . We can break the exponent into parts that are easier to handle:

$$\begin{array}{ccc} 2^{24}\text{B} & \xrightarrow{\hspace{1cm}} & 2^{20} \cdot 2^4 \\ \downarrow & & \downarrow \quad \downarrow \\ 20+4 & & M \quad 16 \end{array}$$

Here's the trick, 2^{20} is a MB and 2^4 is 16, this gives 16 MB. The exact answer is 16,777,216, but the estimate is close enough for most practical purposes.

From memory addressing to data transfer, digital systems rely on these standardized groupings to ensure efficiency and compatibility across hardware and software.

2.3 Arithmetic with Binary Numbers

Digital systems perform **arithmetic** using binary numbers, where simple rules for addition, subtraction, and multiplication must be adapted to handle the representation of both positive and negative values.

2.3.1 Binary Addition

Binary addition works according to the same principles as decimal addition, but it is actually simpler because there are only two digits. Just as in decimal addition, if the sum of two digits is greater than what a single digit can hold, we **carry over** to the next column.

However, in digital system we cannot store an infinite number of digits. Numbers are represented using a **fixed number of bits**, such as 4 bits, 8 bits, 32 bits, or 64 bits. This limitation means that sometimes the result of an addition is **too large** to fit in the available number of bits. This situation is called **overflow**. For example, a 4-bit number can represent values from 0 to 15, if the result of an addition is larger than 15, it cannot be stored correctly in 4 bits. The extra bit is discarded, producing an incorrect result within the 4-bit limit.

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \text{ (with a carry of 1)} \end{aligned}$$

$$\begin{array}{r} 11 \\ 4277 + \\ 5499 = \\ \hline 9776 \end{array} \quad \begin{array}{r} 11 \\ 1011 + \\ 0011 = \\ \hline 1110 \end{array}$$

← carries →

$$\begin{array}{r} 11 \\ 1101 + \\ 0101 = \\ \hline 10010 \end{array}$$

overflow

Overflow can be **detected** by checking for a carry out of the most significant column. If such a carry exists but cannot be stored, it means the result has exceeded the representable range.

This is an important point. Keep in mind that we are no longer in the purely mathematical realm; we are dealing with real systems that have specific limitations, and overlooking these constraints can lead to serious problems:



The \$7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed. The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.

2.3.2 Signed binary numbers

Of course, we can also consider **subtraction**, which works in a way similar to decimal subtraction. However, our focus is on how these operations are **implemented in hardware**. If we were to support both addition and subtraction directly, we would need **two separate circuits**: an adder and a subtractor. Instead, we can observe that subtraction can be expressed as the addition of a positive number and a negative number, thus reducing the two operations to a single one in hardware. This requires a representation for negative numbers that allows us to reuse the same addition circuitry.

The most straightforward idea is the **sign/magnitude representation**:

- the most significant bit (MSB) is used as the sign bit
- a sign bit of 0 indicates a positive number
- a sign bit of 1 indicates a negative number
- the remaining N-1 bits are used for the magnitude, or absolute value

This system is intuitively appealing because it **resembles our usual way of writing numbers**: a minus sign followed by the magnitude. For example a 4-bits numbers:

- $0101 = +5$
- $1101 = -5$

However, with this **ordinary binary addition does not work** and we have **two representations of zero**, which is a undesirable property:

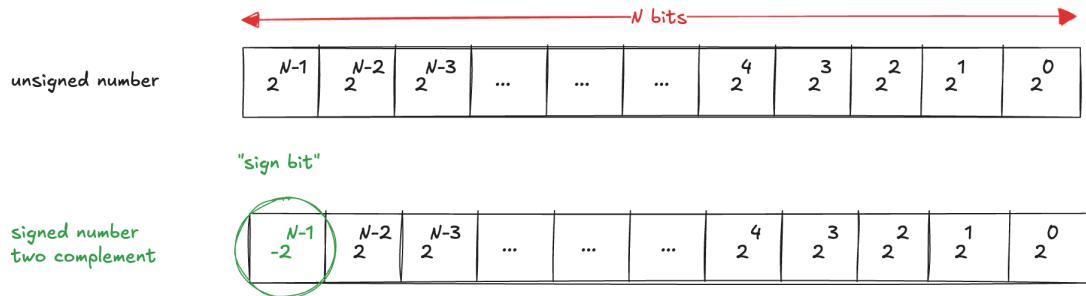
$$\begin{array}{r} -5 + \text{---} \\ 5 = \text{---} \\ \hline 0 \end{array} \quad \begin{array}{r} 1101 + \\ 0101 = \\ \hline 10010 \end{array}$$

nonsense

$$\begin{array}{r} 00000000 \\ 10000000 \\ \hline \end{array} \quad \begin{array}{l} +0 \\ -0 \end{array}$$

weird

Because of these issues, sign/magnitude is rarely used in digital systems. The most common solution is the **two's complement representation**. It works exactly like unsigned binary numbers, except that the **most significant bit (MSB) has a negative weight**:



The following table show a 8-bit explanation:

8-bit Binary value	Unsigned value	Signed value
00000000	0	0
00000001	1	+1
00000010	2	+2
...
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
...
11111110	254	-2
11111111	255	-1

The table illustrates how the same 8-bit binary pattern can be interpreted in two different ways: as an unsigned number (ranging from 0 to 255) or as a signed number (ranging from -128 to +127). If the MSB is 0, the number is non-negative, and the representation is the same as the unsigned case.

Example: $00000010 = +2$.

If the MSB is 1, the number is negative, however the MSB now carries a weight of -128 (for the 8-bit case), while the other bits contribute with their normal positive weights. This means that 10000000 is interpreted as -128, 10000001 as -127, and so on, until 11111111, which represents -1.

Notice that the two's complement representation **covers the range from -128 up to $+127$** . This range is not symmetric: there is one more negative number than positive numbers. The reason is that the MSB has been reassigned a negative weight, allowing us to cover an extra value on the negative side, it is called the **weird number**, because it has no positive counterpart.

Notice that the range of an N-bit two's complement number is

$$[-2^{N-1}, 2^{N-1} - 1]$$

For example, the range of a 8-bit two's complement representation covers the range from -128 up to $+127$. This range is **not symmetric**: there is one more negative number than positive numbers, since there is no negative zero. The most negative value is therefore special, because it has no corresponding positive counterpart. This is sometimes referred to as the **weird number**.

The main advantage of two's complement representation is that **binary addition and subtraction work correctly for both positive and negative numbers**. This makes it possible to implement all arithmetic with a single adder circuit in hardware.

When adding two N-bit numbers in two's complement:

- if the result requires $N+1$ bits, the extra carry out of the most significant bit is simply discarded
- the remaining N bits represent the correct result, provided that no overflow occurs. Consider the following example with 4-bits words:

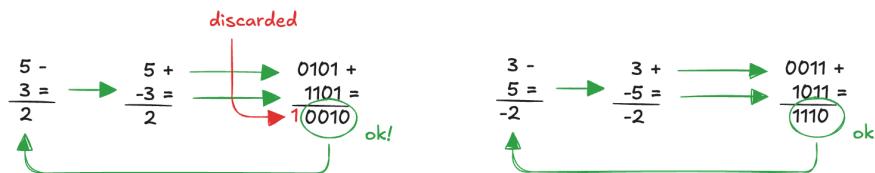


Since only 4 bits are stored, the fifth bit is discarded, leaving 0000, which correctly represents 0.

Subtraction is performed by converting it into addition:

$$A - B = A + (-B)$$

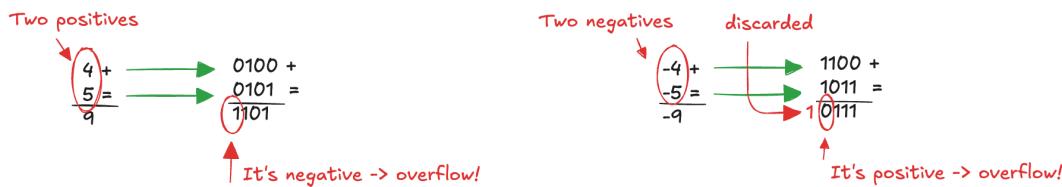
This works because negative numbers are already represented in a way that makes addition valid. Consider the following example using 4-bits words:



Another advantage is that **zero has only one unique encoding**, which avoids the redundancy present in other signed number systems.

However, we need to be careful: **overflow cannot be detected simply by checking the final carry bit** as with unsigned numbers. Fortunately, there is still a reliable way to detect it. The overflow can occur only if we are adding two numbers with the same sign, otherwise the result will be in the correct range. Notice that if we add two numbers with the same sign (both positive or both negative), the result should logically have the same sign:

- adding two positives should never give a negative
- adding two negatives should never give a positive If that happens, it means the true result was too large (for positives) or too small (for negatives) to fit in the available number of bits. The sign "flips" incorrectly because the representation has wrapped around, and this is exactly what we call overflow. For example:

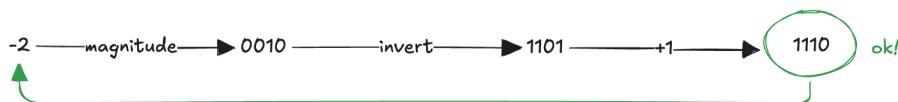


If the same operations had been performed using 5 bits, the correct results would have been obtained.

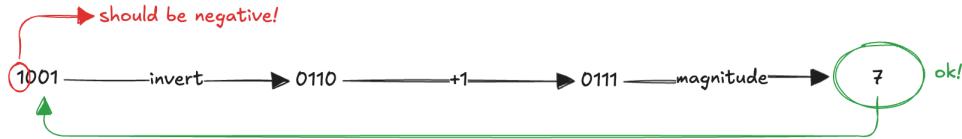
A fast procedure to obtain the two's complement representation of a negative number is straightforward:

- take the binary representation of the positive number (the magnitude)
- invert all the bits (this is called the one's complement)
- add 1 to the result.

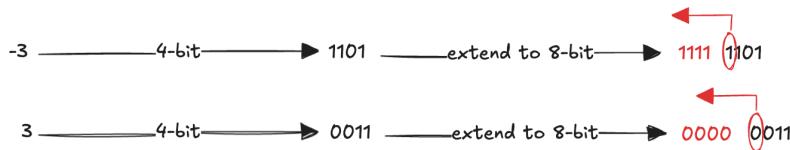
This process converts a positive number into its negative counterpart. Consider the following example, representing -2 in 4-bit two's complement word



We can reverse the procedure to get the magnitude of a negative number. For example, consider the 4-bit binary number 1001. We want to know its decimal value:

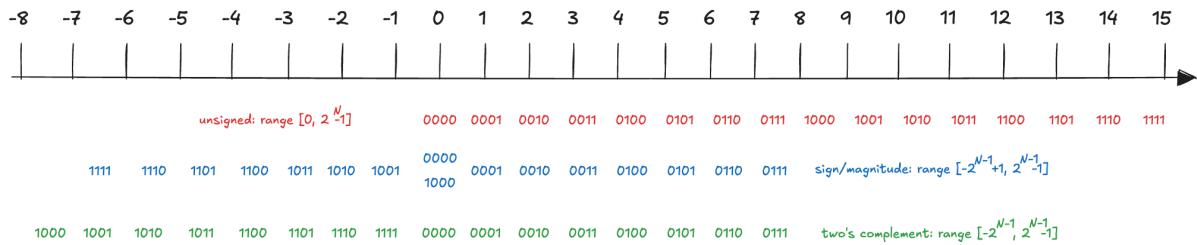


When a two's complement number is represented **with more bits**, its value must remain unchanged. To achieve this, the sign bit (the MSB) is copied into the newly added higher-order positions. This procedure is called **sign extension**. For instance:



Notice that the additional bits simply repeat the original sign bit, ensuring that the value is preserved.

Finally, we can provide a comparison between unsigned, sign/magnitude and two's complement representations in terms of ranges and value:

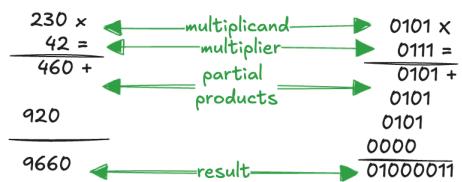


Two's complement is the standard in digital systems because it combines **efficiency** (one circuit for both addition and subtraction), **uniqueness** (only one zero), and **correctness** (arithmetic works seamlessly).

2.3.3 Binary Multiplication

Multiplication in binary follows the same principles as multiplication in decimal. The only difference is that binary numbers involve only two digits and this greatly simplifies the process of forming partial products. When multiplying two binary numbers:

- each bit of the multiplier is considered in turn
- if the bit is 1, the partial product is the entire multiplicand
- if the bit is 0, the partial product is simply 0
- each partial product is shifted according to the position of the multiplier bit, just as in decimal multiplication



In decimal multiplication, partial products may be multiples of the multiplicand, in binary multiplication, **each partial product is either the multiplicand or zero**, making the process simpler. The result of multiplying an M-bit number by an N-bit number may require up to **M+N bits**.

2.4 Fractional Numbers

We have focused on how digital systems represent integers, either signed or unsigned. However, digital systems must also handle fractions (or rational numbers), since many real-world value (such as temperature, voltage, or probabilities) cannot always be expressed as whole numbers. To represent fractions, we can use two main approaches: fixed-point numbers and floating-point numbers.

2.4.1 Fixed-point numbers

In fixed-point representation, numbers are expressed with an **implied binary point** between the integer part and the fractional part. This is directly analogous to the decimal point in ordinary decimal numbers. For example:

$$0110.1100 = 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

Here, the digits to the left of the binary point represent integer values, while the digits to the right represent fractional values.

Just as with integers, fixed-point numbers can also represent negative values. We can use the **sign/magnitude notation**, in which the MSB is reserved for the sign of the number and the remaining bits represent the magnitude of the number. For example:

-2.375 is written as 1010.0110

where the leading 1 denotes the negative sign and the rest of the bits encode the magnitude. Or we can adopt the **two's complement notation**. For example:

-2.375 is written as 1101.1010

Notice that even though any integer number N can be written as a sum of powers of two:

$$78 = 64 + 8 + 4 + 2 = 2^6 + 2^3 + 2^2 + 2^1$$

decimal fractions are **rarely exact in binary**. For example, consider 0.78; in binary it is represented as an infinite series:

$$0.78 = 0.1100010010000111110 \dots$$

This is an infinite repeating expansion in base 2, just like $1/3 = 0.3333 \dots$ in base 10.

The reason is that:

$$0.78 = 78/100 = 39/50 = 39/(2 \times 5^2).$$

In base 2, the denominator includes the factor 5, which is not a power of 2, and therefore the binary expansion does not terminate. Only fractions of the form

$$\frac{k}{2^n}$$

can be represented exactly in binary.

It is important to stress that fixed-point numbers are, at the hardware level, **nothing more than a sequence of bits**. The binary point is not physically present in the representation. Instead, its position is established only by convention. This means that, without prior agreement on the format, there is no way to determine where the binary point should be placed, and thus no way to correctly interpret the number. The interpretation always depends on the chosen representation scheme.

A common way to describe fixed-point formats is the **Q notation**, which makes explicit how many bits are assigned to the integer part and how many bits are assigned to the fractional part. In this notation:

- **Ua.b** denotes an unsigned fixed-point number, where a bits represent the integer part and b bits represent the fractional part;
- **Qa.b** denotes a signed fixed-point number in two's complement, with a bits for the integer part and b bits for the fractional part.

Fixed-point formats are widely used in practice, particularly in **digital signal processing** (DSP), **computer graphics**, and **machine learning** applications. These domains often require real-time computations on embedded hardware platforms and fixed-point arithmetic offers a good compromise between efficiency, precision, and hardware complexity. To give some concrete examples:

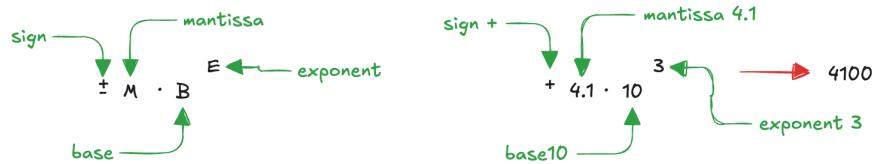
- Q1.15 uses 1 bit for the sign and 15 bits for the fractional part. It is the most common format in DSP applications because it offers high precision for fractional values.
- Q1.31 has 1 sign bit and 31 fractional bits. It is used when extremely high precision is required, for instance in intermediate stages of filtering or other numerical algorithms.

- U8.8 has 8 integer bits and 8 fractional bits in an unsigned representation. This format is frequently employed in sensor readings, where both a reasonable integer range and fractional precision are necessary.

However, fixed-point formats have some limitations. They **require a fixed agreement** on where the binary point lies, and **their range and precision are limited** by the number of bits allocated to the integer and fractional parts.

2.4.2 Floating-point numbers

Floating-point numbers are best understood by drawing an analogy with **scientific notation** in decimal arithmetic. In scientific notation, a number is expressed in the form:



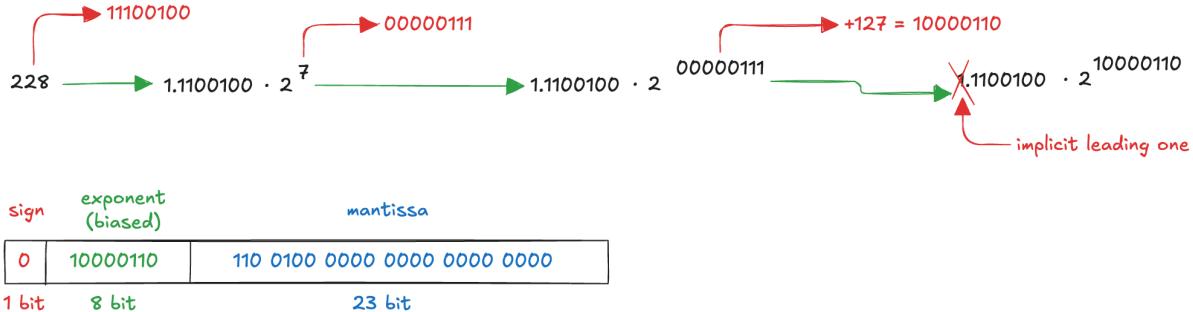
where:

- M is the **mantissa**, which contains the significant digits of the number,
- B is the **base** (10 in the decimal system, 2 in binary systems),
- E is the **exponent**, which shifts the decimal (or binary) point,
- and the leading **sign** indicates whether the number is positive or negative.

In the example, the mantissa is 4.1, the base is 10, and the exponent is 3. Notice that in this form the **decimal point "floats"** so that it always follows the most significant digit of the mantissa.

This idea extends naturally to binary floating-point numbers. By adjusting the exponent, the binary point can be shifted left or right, which makes it possible to represent both very large numbers and very small numbers using the same format. However, there are many reasonable ways to store the information about mantissa, sign and exponent, and historically computer manufacturers developed their own formats independently. This lack of compatibility meant that results produced on one digital system could not easily be interpreted by another, which represented a significant barrier to portability and interoperability. To address this issue, the **Institute of Electrical and Electronics Engineers** (IEEE) introduced the **IEEE 754 Floating-Point Standard** (1985). This standard defines how floating-point numbers are represented and manipulated, and it has since become the universally adopted convention in modern computing. Today, virtually all processors and programming languages rely on IEEE 754, ensuring that floating-point calculations are consistent across platforms. For example, in the 32-bit single-precision format, one bit is allocated for the sign, eight bits for the exponent, and twenty-three

bits for the fraction. The use of a bias (127 in the single-precision case) allows the exponent to represent both positive and negative values. For example, the number 228 can be represented as:



Notice that, in normalized floating-point numbers, the first bit of the mantissa is always 1. This bit, known as the **implicit leading one**, does not need to be stored explicitly, which allows one additional bit of precision to be represented within the available storage.

Notice that floating-point numbers can represent a **much wider range of values** than fixed-point numbers with the same number of bits. However, this flexibility comes at the cost of **increased complexity** in both hardware and software implementations. Floating-point arithmetic is more computationally intensive than fixed-point arithmetic, and it can introduce rounding errors and precision issues that must be carefully managed.

The IEEE 754 standard also defines several special cases:

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000000000000
∞	0	11111111	00000000000000000000000000000000
$-\infty$	1	11111111	00000000000000000000000000000000
NaN	X	11111111	Non-zero

The IEEE 754 standard defines additional formats:

By providing a common and precise representation, IEEE 754 makes it possible to carry out floating-point arithmetic in a consistent way across different architectures. This consistency is essential for scientific computing, graphics, and any domain where reproducibility of numerical results is critical.

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits	Bias
Single	32	1	8	23	127
Double	64	1	11	52	1023
Quad	128	1	15	112	16363

In recent years, the growing importance of machine learning has driven the development of **new floating-point formats**. These formats **trade some precision for reduced storage and faster computation**, which is acceptable in many applications where approximate results are sufficient. Examples include:

- Google BFloat16: 16-bit format with 8 exponent bits and 8 mantissa bits.
- IBM DLFloat16: 16-bit format with 6 exponent bits and 10 mantissa bits.
- NVIDIA FP8: 8-bit format with 4 exponent bits and 4 mantissa bits.
- Tesla CFloat8: 8-bit format with 5 exponent bits and 4 mantissa bits.
- NVIDIA FP4: 4-bit format introduced in the NVIDIA Blackwell GPUs (2024).

These compact formats are not designed for general-purpose arithmetic, but they provide significant performance and memory advantages.

2.5 Exercises

1 - The Babylonians developed the sexagesimal (base 60) number system about 4000 years ago. How many bits of information is conveyed with one sexagesimal digit? How do you write the decimalnumber 4000 in sexagesimal?

- Information per digit:

$$I = \log_2(60) \approx 5.91 \text{ bits}$$

- Convert 4000 to base 60:

$$4000/60 = 66 \text{ remainder } 40$$

$$66/60 = 1 \text{ remainder } 6$$

$$1/60 = 0 \text{ remainder } 1$$

- Result:

$$4000 = 1 \times 60^2 + 6 \times 60^1 + 40 \times 60^0$$

2 - How many different numbers can be represented with 16 bits?

- Each bit can take two values (0 or 1)

- With 16 bits, the total number of distinct patterns is:
 $2^{16} = 65.536$
- If the numbers are unsigned, the range is
0 to 65.535
- If they are signed two's complement, the range is
–32.768 to +32.767

3 - What is the largest unsigned 32-bit binary number?

- An unsigned 32-bit binary number uses all 32 bits for magnitude.
- The largest value occurs when all bits are 1:
11111111 11111111 11111111 11111111
- In decimal, this equals:
 $2^{32} - 1 = 4.294.967.295$
- Usando i multipli:
 $2^{32} = 2^{30} \cdot 2^2 = 4 \text{ GB}$

4 - What is the largest 16-bit binary number that can be represented with (a) unsigned numbers? (b) two's complement numbers? (c) sign/magnitude numbers?

(a) Unsigned numbers

- All 16 bits are used for the magnitude.
- Largest value:
11111111 11111111 = $2^{16} - 1 = 65.535$

(b) Two's complement numbers

- Range for N-bit two's complement:
[2^{n-1} , $(2^{n-1} - 1)$]
- For n = 16:
[–32,768 , +32,767]
- Largest value: 32.767

(c) Sign/magnitude numbers

- 1 bit for sign, 15 bits for magnitude.
- Largest positive magnitude:
0111111111111111 = $2^{15} - 1 = 32.767$

5 - Convert the unsigned binary number 11110000 to decimal and to hexadecimal

- Binary to decimal:

$$11110000 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 = 128 + 64 + 32 + 16 = 240$$

- Binary to hexadecimal:

Group into 4-bit chunks: $1111\ 0000_2 = F0_{16}$

6 - Convert the hexadecimal 3B16 number to decimal

- $3B = 3 \times 16^1 + 11 \times 16^0 = 48 + 11 = 59$

7 - Convert the two's complement binary numbers 011100002 to decimal

- 01110000_2 has MSB = 0, so the number is non-negative.
- $01110000 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 = 64 + 32 + 16 = 112$

8 - Convert the decimal numbers -63 and -132 to 8-bit two's complement number or indicate that the decimal number would overflow the range

- For 8-bit two's complement, the representable range is -128 to $+127$.

- -63 :

$$63 = 00111111$$

Invert bits: 11000000_2

Add 1: 11000001_2

$$\text{Final: } -63_{10} = 11000001_2$$

- -132_{10} :

Since $-132 < -128$, it falls outside the 8-bit two's complement range.

Result: **overflow**

9 – Convert the decimal number -5.75 into IEEE 754 single-precision format

- Sign bit: 1 (negative number)
- Convert integer part: $5 = 101$
- Convert fractional part: $0.75 = .11$
- Combined binary: 101.11
- Normalize: 1.0111×2^2

- Exponent: $2 + 127 = 129 = 10000001$
- Mantissa: 0111000 ... (fill to 23 bits)
- Final: 1 10000001 0111000000000000000000000000000

10 – Convert the IEEE 754 single-precision binary number

0 10000001 1001100000000000000000000000000₂ **into decimal.**

- Sign = 0, so it is positive
- Exponent = 10000001 = 129
- Unbiased exponent = $129 - 127 = 2$
- Fraction bits = 1001100 ...
- Ssignificand = 1.10011₂
- $1.10011 = 1 + 2^{-1} + 2^{-4} + 2^{-5} = 1.59375$
- Value = $1.59375 \times 2^2 = 6.375$

3 Logic gates and circuits

Digital systems are built on the simplest possible foundation: two states, usually represented as 1 and 0. These two values are not just numbers; they can also be interpreted as **TRUE** and **FALSE**, **HIGH** and **LOW** voltage, or even as the **presence** or **absence** of a signal. This abstraction allows us to **ignore the messy details of electronics** and instead **reason in terms of logic**.

At the heart of digital design are **logic gates**, the elementary building blocks that take one or more binary inputs and produce a binary output according to a well-defined rule. By combining gates, we can implement any **logical function**, from simple comparisons to the complex operations inside a microprocessor. While the gates themselves are simple, the real power comes from combining them into larger modules that ultimately form the backbone of computers and digital devices.

3.1 Logic gates

Logic gates are **simple digital circuits** that accept one or more binary inputs and produce a binary output. Each logic gate is represented by a **symbol** that shows how inputs are connected to the output. By convention, the inputs are placed on the left (or sometimes at the top), while the output appears on the right (or at the bottom).

The relationship between the inputs and the output can be described in two equivalent ways:

- A **truth table** explicitly lists all possible input combinations in its left columns and the corresponding output values in the right column. In this way, it provides a complete specification of the gate's behavior.
- A **Boolean equation** is a mathematical expression that uses binary variables and logical operators to capture the same relationship in algebraic form.

Both descriptions are fundamental tools for analyzing and designing digital circuits.

3.1.1 NOT gate (Inverter)

A NOT gate is the simplest type of logic gate. It has one input, usually labeled A, and one output, usually labeled Y. The output is the inverse of the input:

- If A is FALSE (0), then Y is TRUE (1)
- If A is TRUE (1), then Y is FALSE (0)

The graphical symbol for the NOT gate is a triangle with a small circle (called a bubble) at its output, representing the inversion. The behavior can be summarized in the two complementary ways:



The bar over the variable indicates the logical NOT operation, and is read as "Y equals NOT A". Because it reverses the logic value of its input, the NOT gate is also called an inverter.

3.1.2 BUFFER gate

The second one-input logic gate is the buffer. Unlike the NOT gate, the buffer **does not invert the signal**: it simply copies the input to the output:

- If the input A is 0, the output Y is also 0;

- If the input A is 1, the output Y is also 1.

This behavior can be expressed with the Boolean equation and the truth table:

name: BUFFER	symbol: A ——————→ Y	truth table:	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th><th>Y</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td></tr> </tbody> </table>	A	Y	0	0	1	1	boolean equation: Y = A
A	Y									
0	0									
1	1									

From a **purely logical point of view**, the buffer seems redundant, since it performs the same function as a **direct wire connection** between input and output. However, from the **analog point of view**, the buffer provides important benefits. It is capable of delivering large amounts of current, for example to drive a motor, and it can rapidly distribute its output to many other gates in a circuit without signal degradation.

This example highlights the **importance of considering multiple levels of abstraction**. While the digital abstraction suggests that the buffer is unnecessary, at the physical implementation level it plays a critical role in ensuring reliable and efficient circuit operation.

3.1.3 AND gate

The AND gate is a **two-input logic gate** whose output is TRUE only when both inputs are TRUE. In every other case, the output is FALSE:

- If the inputs A is 1 and B is 1, the output Y is 1
- If one of the inputs A or B (or both) is 0, then the output Y is 0.

This behavior is summarized by the following truth table and Boolean equation:

name: AND	symbol: A ——————→ Y	truth table:	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th><th>B</th><th>Y</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	boolean equation: Y = AB
A	B	Y																	
0	0	0																	
0	1	0																	
1	0	0																	
1	1	1																	

The equation can also be written in alternative forms, such as:

$$Y = AB$$

$$Y = A \cdot B$$

$$Y = A \cap B$$

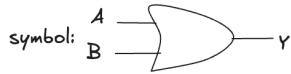
The dot or simple juxtaposition are commonly used in digital design and read as "A AND B". The symbol cap, meaning intersection, is often preferred in logic and mathematics.

The AND gate is one of the most fundamental logic gates and is widely used to model situations where two or more conditions must be simultaneously satisfied.

3.1.4 OR gate

The OR gate is a logic gate that produces a TRUE output if at least one of its inputs is TRUE:

- If either input is 1 (or both are 1), the output will also be 1
- If both inputs are 0, the output will be 0.

name: OR
symbol: 

truth table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

boolean equation: $Y = A + B$

The Boolean equation for the OR gate is written as:

$$Y = A + B \quad Y = A \cup B$$

The plus sign is widely used by digital designers and is read as "A OR B". The union symbol is often preferred in mathematics and logic.

The OR gate is fundamental in digital systems because it models situations in which one condition, or another, or both can lead to a positive result.

3.1.5 XOR, NAND, NOR Gates

The AND, OR, and NOT gates are the fundamental logical operations. With them, we can construct all the digital operations we need. However, besides these basic operations, other logic gates exist. In some cases these gates are more convenient to use, or they allow the design of digital circuits to be simplified. Some of these gates are even **universal**: with them, it is possible to implement any logical function. Among the most common two-input gates we have the XOR (exclusive OR), pronounced "ex-OR", which is TRUE if A or B is true, but not both:

name: XOR
symbol: 

truth table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

boolean equation: $Y = A \oplus B$

It is indicated by a plus sign with a circle around it; the NAND, which performs NOT AND. Its output is TRUE unless both inputs are true:

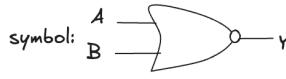
name: NAND
symbol: 

truth table:

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

boolean equation: $Y = \overline{AB}$

and the NOR, which performs NOT OR. Its output is TRUE only if neither A nor B is true:

name: NOR	symbol: 	truth table:	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	boolean equation: $Y = \overline{A+B}$
A	B	Y																	
0	0	1																	
0	1	0																	
1	0	0																	
1	1	0																	

3.2 Boolean Algebra

Boolean algebra is the **mathematical framework** used to represent and manipulate logical expressions. It can be described as an **algebraic system**, which means it is a set of elements together with a set of operations defined on them. In this case, the system is defined by three components:

- the set of values $\{0, 1\}$,
- the operations OR, AND, and NOT,
- the equivalence operator =

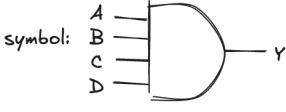
A **Boolean variable** is any discrete variable that can assume only two possible values, 0 or 1, and a **Boolean function** is any expression F that is formed using logical operations. It takes one or more binary inputs

(X_1, X_2, \dots, X_n)

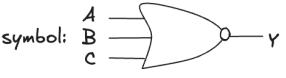
and produces a binary output Y .

The **domain** of a Boolean function is made up of all the 2^n possible combinations of the values of the variables. This domain is **countable**, meaning that it **can be listed explicitly**, this means that a Boolean function can be completely specified by its Boolean expression or by a truth table that lists all possible input combinations and their corresponding output values.

Using the idea of Boolean function, we can extend the basic logic gates in order to handle multiple inputs. An **N-input AND gate** produces a TRUE output only when all N inputs are TRUE. For example, a 4-input AND gate (AND4) outputs 1 only when A, B, C, and D are all equal to 1:

name: AND4	symbol: 	truth table:	truth table:																																																																																					
			<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th>A</th> <th>B</th> <th>C</th> <th>D</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	C	D	Y	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	1	0	0	1	1	0	0	0	1	1	1	0	1	0	0	0	0	1	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
A	B	C	D	Y																																																																																				
0	0	0	0	0																																																																																				
0	0	0	1	0																																																																																				
0	0	1	0	0																																																																																				
0	0	1	1	0																																																																																				
0	1	0	0	0																																																																																				
0	1	0	1	0																																																																																				
0	1	1	0	0																																																																																				
0	1	1	1	0																																																																																				
1	0	0	0	0																																																																																				
1	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																																																																																				

An N-input NOR gate produces a TRUE output when all inputs are FALSE. For example, a 3-input NOR gate outputs 1 only if all three inputs are 0:

name: NOR3	symbol: 	truth table:	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th>A</th> <th>B</th> <th>C</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	C	Y	0	0	0	1	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	0	boolean equation: $Y = \overline{A+B+C}$
A	B	C	Y																																					
0	0	0	1																																					
0	0	1	0																																					
0	1	0	0																																					
0	1	1	0																																					
1	0	0	0																																					
1	0	1	0																																					
1	1	0	0																																					
1	1	1	0																																					

These examples illustrate how basic gates can be generalized to handle more inputs, while still following the same logical definitions. Their behavior can always be described using symbols, Boolean equations, and truth tables.

3.3 Digital Circuits

A circuit is a **network** that processes Boolean variables. It can be viewed as a black box with one or more **input terminals** and one or more **more output terminals**. To describe such a circuit, we use two kinds of specifications:

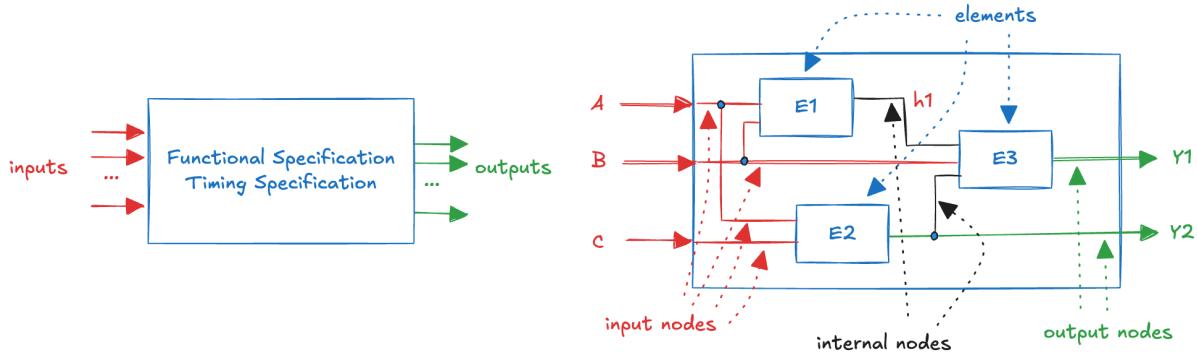
- a **functional specification**, which defines the **logical relationship between inputs and outputs**;
- a **timing specification**, which describes the **delay between a change in the inputs and the corresponding change in the outputs**.

Inside the black box, circuits are built from **nodes** and **elements**:

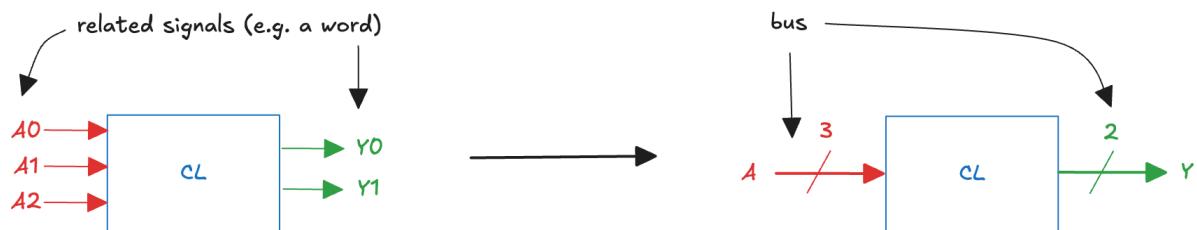
- elements are themselves circuits, each with inputs, outputs, and specifications.
- nodes are the wires that connect elements together, carrying a voltage that represents a Boolean variable.

There are three types of nodes:

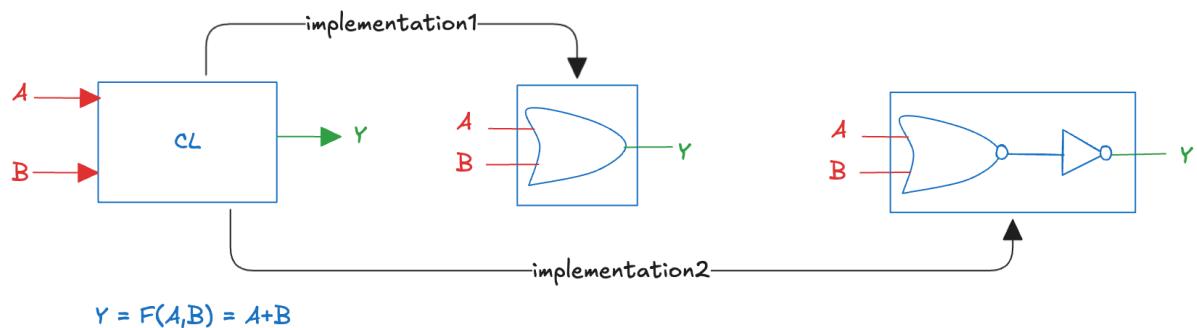
- **input nodes** receive values from the external world,
- **output nodes** deliver values to the external world,
- **internal nodes** carry signals inside the circuit but are not directly connected to the outside.



Sometimes, several nodes are **grouped together** to carry **related signals**. In this case, instead of drawing many separate wires, we use a **bus**, which is simply a bundle of multiple signals treated as a unit. A bus is represented by a single line with a slash and a number next to it, where the number indicates how many signals it contains. This compact notation makes circuit diagrams clearer and easier to read, especially when dealing with multi-bit values such as data words:



Together, nodes and elements form the **internal structure of digital circuits**, while from the outside the system can still be described simply by its inputs, outputs, and specifications. In practice, there are often **multiple possible implementations for the same logical function**:



The **choice** of implementation depends on the **available building blocks** and on **design constraints** such as area, speed, and power consumption. Different circuit realizations may compute the same Boolean function, but they can vary in terms of **efficiency** and **performance**.

Digital circuits can be divided into **combinational and sequential types**, depending on how their outputs are determined.

3.3.1 Combinational circuits

A combinational circuit produces an output that **depends only on the current values of the inputs**. It simply combines the present input values to compute the output, without remembering anything from the past. For this reason, combinational circuits are said to be **memoryless**. A basic logic gate is an example of a combinational circuit.

3.3.2 Sequential circuits

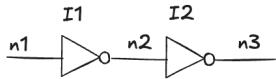
A sequential circuit produces an output that **depends on both the current and the previous values** of the inputs. In other words, the output is influenced by the sequence of inputs over time. Sequential circuits therefore have **memory**, and they can store information about past inputs.

3.3.3 Combinational composition

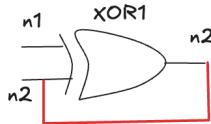
To design digital systems, we often want to build a **large combinational circuit** starting from smaller combinational circuit elements. This modular approach makes it easier to manage complexity and reuse standard building blocks. A circuit is combinational if it is composed of interconnected elements that satisfy the following conditions:

- every element is itself **combinational**,
- every node is **either an input** or is **connected to exactly one output** of an element,
- the circuit contains **no cyclic paths**, meaning that every path through the circuit visits each element at most once.

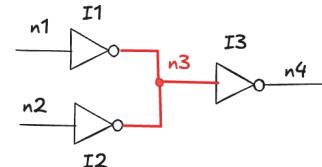
These rules are **sufficient** to guarantee that a circuit is combinational, but they are **not strictly necessary**. In fact, there exist circuits that violate one or more of these conditions and are still combinational. However, to simplify design and analysis, we **restrict ourselves** to combinational composition when building combinational circuits. Here some examples to clarify these concepts:



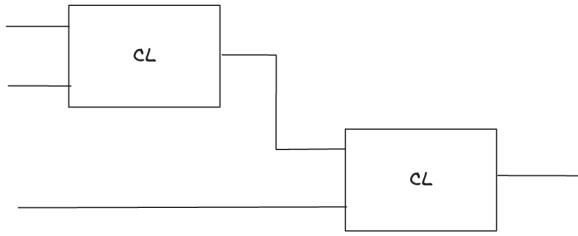
It is combinational circuit: the circuit is made of interconnected elements with no cyclic paths. Each node connects properly to a single output, so the rules are satisfied.



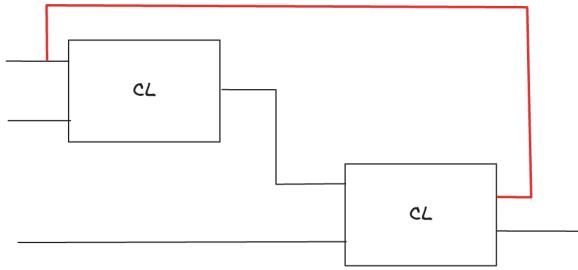
It is not combinational: there is a cyclic path in the circuit. This feedback loop prevents it from being purely combinational.



It is not combinational: node n3 is incorrectly connected to the outputs of two elements (I1 and I2). Since a node must connect to exactly one output, the rule is violated.



It is combinational: two combinational circuits are connected together to form a larger one. Since each sub-circuit follows the rules, the overall circuit is also combinational.



It does not obey the rules: the circuit contains a cyclic path through the two elements. Depending on the actual functions of these elements, the overall circuit may or may not be combinational, but it does not respect the strict rules of combinational composition.

3.4 Complexity

Large circuits, such as microprocessors, can be extremely complex. To manage this complexity, we rely on a few key principles:

- **Abstraction and modularity:** we can view a circuit as a black box with a well-defined interface and function, without worrying about the internal details.
- **Hierarchy:** we build complex systems by combining smaller circuit elements step by step, creating multiple levels of organization.
- **Discipline:** we apply the rules of combinational composition to ensure circuits behave correctly and remain easy to analyze.

Using these principles allows designers to break down a seemingly overwhelming problem into smaller, manageable parts.

The functional specification of a combinational circuit is usually expressed either as a truth table or as a Boolean equation. This raises some important design questions:

- How can we implement the basic gates AND, OR, and NOT?

- How can we derive a Boolean equation from any given truth table?
- How can we use Boolean algebra to simplify Boolean equations?

These questions guide the process of moving from abstract functional specifications to practical, efficient circuit implementations.

3.5 Simulation

Simulation is an essential step between theory and practice. When we design a circuit on paper, we rely on Boolean equations and truth tables to predict how it should behave. However, as circuits grow more complex, analyzing them manually becomes difficult and error-prone. Simulation bridges this gap by allowing us to **test our designs in a safe, controlled, and cost-free environment**. Before building any physical prototype, we can use simulation tools to:

- **Verify correctness:** check that the circuit produces the expected outputs for all possible inputs.
- **Explore behavior over time:** visualize how signals change, and how propagation delays affect the output.
- **Detect errors early:** find logical mistakes or design flaws before hardware is built, avoiding expensive rework.
- **Experiment freely:** try variations, optimize the design, and gain intuition about circuit behavior without worrying about damaging components.

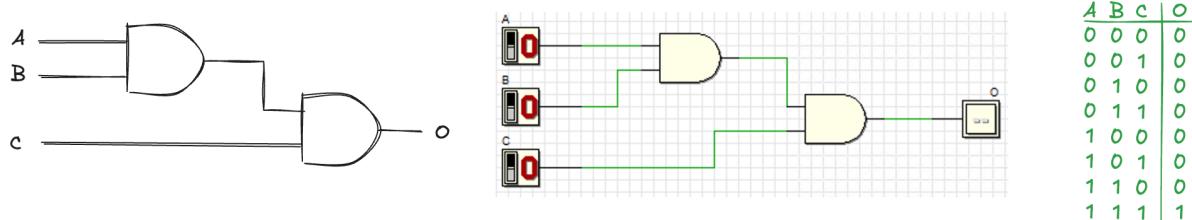
In education, simulation is even more valuable. It provides students with an interactive environment where they can directly connect theory to practice. By **building and testing circuits virtually**, learners see abstract concepts like logic gates, timing, and sequential behavior come to life. Ultimately, simulation is not just a convenience, it is a core methodology of modern digital design, enabling designers and students alike to move confidently from ideas to working systems. The first tool we will use is the **Deeds Digital Circuit Simulator**

DEEDS, short for **Digital Electronics Education and Design Suite**, is a comprehensive educational platform developed at the University of Genoa. It's designed not as a commercial tool but as an intuitive, hands-on environment for learning digital electronics. At the core of Deeds is the Digital Circuit Simulator, a tool tailored to help students design and interact with digital circuits. With its graphical schematic editor, it allows us to drag and drop components like logic gates then wire them intuitively on a canvas. What makes this tool uniquely pedagogical are its simulation modes:

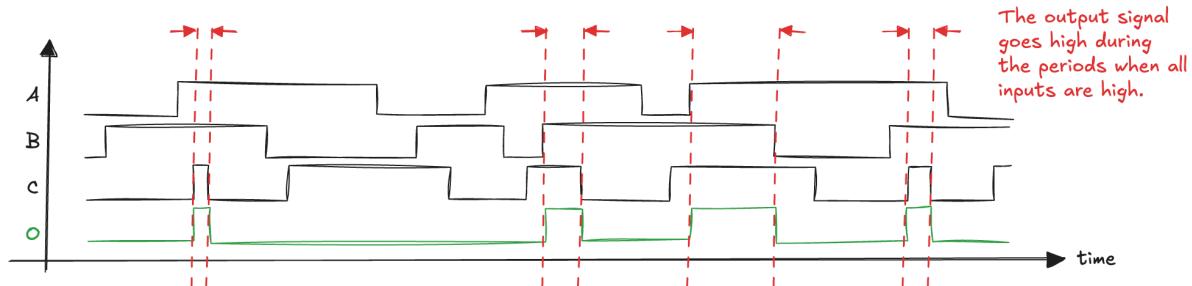
- the **interactive animation mode** lets us flip input switches on the schematic and observe real-time changes in outputs, which is ideal for exploring logic behavior dynamically.

- the **timing Simulation mode** offers a timing-diagram interface, where we can draw input signal sequences and analyze **output waveforms** step-by-step to understand temporal behavior.

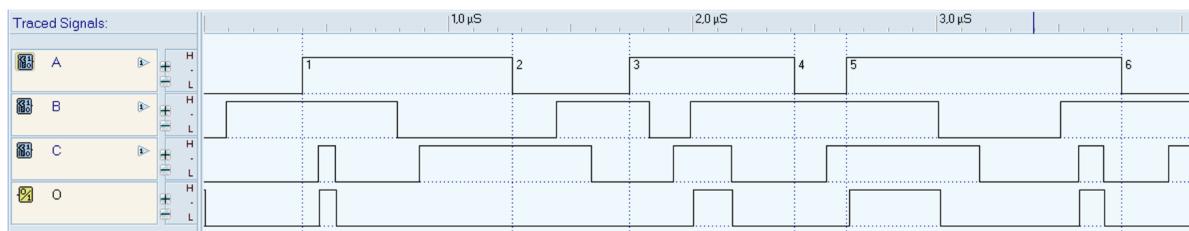
Together, these features bridge the gap between theoretical learning and practical understanding. Let's consider a practical example, we can implement a simple combinational circuit, create it in Deeds, simulate its behaviour and verify that it matches the expected truth table:



We can also examine how the circuit responds as the input signals change, using the simulator to generate the **output waveform** from the given input waveforms. For instance, in the circuit shown we obtain:



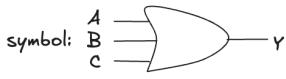
The Deeds simulator provides support for this type of analysis:



3.6 Exercises

1 - Draw the symbol, boolean equation, and truth table for a three-input OR gate, a three-input XOR gate, and a four-input XNOR gate

- three-input OR gate

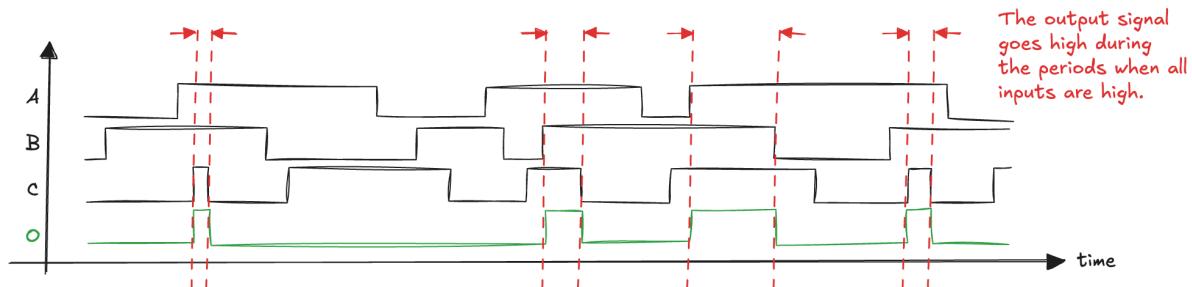
name: OR3 symbol: 

truth table:

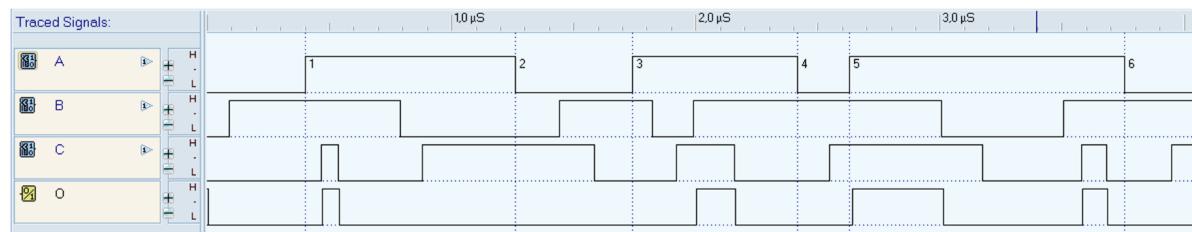
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

boolean equation: $Y = A+B+C$

- three-input XOR gate

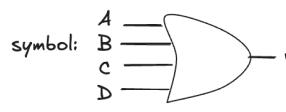


- four-input XNOR gate



2 - Draw the symbol, boolean equation, and truth table for a four-input OR gate, and a three-input XNOR gate

- four-input OR gate

name: OR4 symbol: 

truth table:

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

boolean equation: $Y = A+B+C+D$

- three-input XNOR gate

name: XNOR3 symbol:  truth table:

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

 boolean equation: $Y = \overline{A \oplus B \oplus C}$

3 - A majority gate produces a TRUE output if and only if more than half of its inputs are TRUE. A three-input AND-OR gate produces a TRUE output if both A and B are TRUE or if C is TRUE. Complete a truth table for the two gates.

majority gate →

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

AND-OR gate →

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

4 Beneath the abstraction

A fundamental characteristic of digital systems is that they operate using discrete-valued variables. This means that the information processed inside a digital circuit can only assume a finite set of values. However, **the physical world is not digital**. Real-world variables, such as the voltage on a wire, the angular position of a gear, or the level of fluid in a cylinder, are continuous quantities. They can, at least in principle, take on an infinite number of values within a certain range.

We have already considered part of the problem of mapping the continuous physical world onto discrete digital values when we discussed two fundamental processes: sampling and quantization. Sampling reduces a signal that varies continuously in time into a sequence of values at discrete instants, while quantization reduces an amplitude that could take infinitely many values into a finite set of representable levels. These steps are crucial when we digitize information coming from the outside world—for example, converting a sound wave or temperature measurement into digital data. However, now we have a more deeper issue to address. Even after we have sampled and quantized a signal, we still need to define **how the resulting discrete values are represented physically within a digital system**.

Let us consider an example based on voltage signals, which are the most common way to rep-

resent information in electronic systems. Suppose we want to represent a binary signal A using the voltage on a wire. One simple convention could be:

- A = 0 when the voltage is 0V
- A = 1 when the voltage is 5V

At first sight this might appear straightforward, but in practice, **no physical system is perfectly noise-free** or perfectly stable. Slight variations in voltage are inevitable due to manufacturing tolerances, electromagnetic interference, or fluctuations in power supply. For instance, if the voltage on the wire is 4.97V, we would still like the system to interpret this as A = 1. The question becomes more subtle when the measured voltage is farther away from the expected ideal values. For example, should 4.3V still be considered a logical 1? And what about values like 2.8V or 2.5V? These intermediate cases highlight the need for **design rules** that establish clear **thresholds** between logical 0 and logical 1, while at the same time providing enough **tolerance** to handle noise.

At this point, we also see why **transistors** are indispensable. A transistor is a device that acts as a controllable switch: depending on the input voltage, it either allows or blocks the flow of current. By defining precise voltage thresholds, transistors turn the continuous world of voltages into the binary world of logic. Entire digital systems (processors, memories, and communication circuits) are built from billions of these tiny devices, each faithfully converting continuous electrical signals into the discrete values required by digital abstraction.

This process of defining how a continuous physical signal is interpreted as a discrete logical variable is at the very heart of digital design. Without it, the abstraction of 0 and 1 would collapse under the imperfections of the physical world.

4.1 Logic Levels and Noise Margins

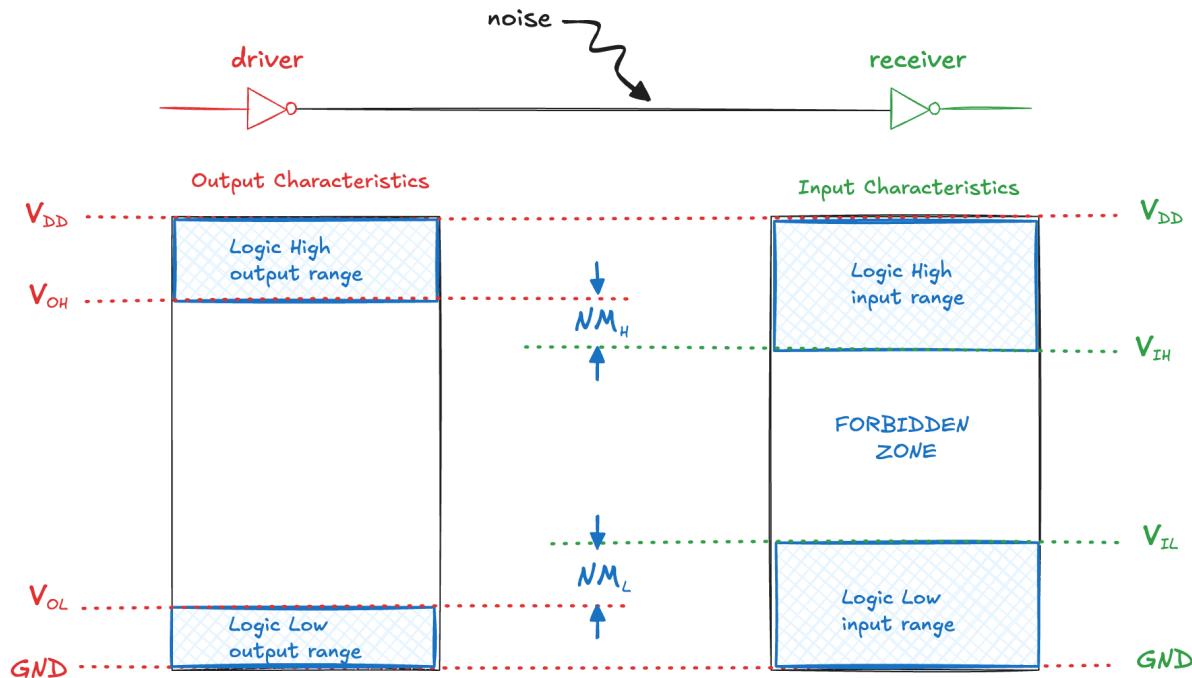
The lowest voltage in a digital circuit is typically 0V, called **ground** or **GND**. The highest voltage comes from the power supply and is denoted as V_{DD} . A binary signal is thus expected to swing between these two reference points. But as we have seen, real circuits are never perfect: voltages will never be exactly 0V or exactly V_{DD} . Manufacturing tolerances, temperature, and electrical noise all contribute to **variations**.

4.1.1 Logic levels

To ensure reliable operation, we introduce **voltage thresholds** that formally define what counts as a logical 0 or a logical 1:

- V_{OL} : the maximum voltage that an output still considered a valid logic 0 can produce
- V_{OH} : the minimum voltage that an output still considered a valid logic 1 can produce
- V_{IL} : the maximum voltage that an input will still interpret as logic 0
- V_{IH} : the minimum voltage that an input will interpret as logic 1

The following diagram illustrates this process: a driver generates an output signal that may vary within the ranges defined by V_{OL} and V_{OH} . This signal is then received as an input, which has its own acceptable ranges defined by V_{IL} and V_{IH} . The overlap between these definitions creates the **safety margins** that make digital circuits reliable even in the presence of noise:



4.1.2 Noise Margins

The **low noise margin** is

$$NM_L = V_{IL} - V_{OL}$$

representing how much noise can be added to a logic 0 signal without the risk of it being seen as a 1

The **high noise margin** is

$$NM_H = V_{OH} - V_{IH}$$

representing how much noise can be added to a logic 1 signal without it being mistaken for a 0.

Between V_{IL} and V_{IH} lies a **forbidden zone**, where the circuit's behavior is undefined. A voltage in this range cannot be reliably classified as either a 0 or a 1.

Defining logic levels and noise margins ensures that digital circuits are **robust against real-world imperfections**. This allows billions of transistors to communicate reliably, even though each one operates in an imperfect analog world. Without noise margins, every small fluctuation in voltage could lead to errors.

To better understand how noise margins guarantee the robustness of digital circuits, let us work through a concrete example. Consider the circuit: the output voltage of the first inverter is connected to the input voltage of the second inverter. Between them, we assume that some noise may be added to the signal, disturbing the transmission. Both inverters are characterized by the following voltage levels:

- Power supply: $V_{DD} = 5V$
- Input thresholds: $V_{IL} = 1.35V, V_{IH} = 3.15V$
- Output levels: $V_{OL} = 0.33V, V_{OH} = 3.84V$

With these values, we can compute the noise margins. The low noise margin is:

$$NM_L = V_{IL} - V_{OL} = 1.35V - 0.33V = 1.02V$$

This means that if the circuit outputs a logic 0, it can tolerate up to 1.02V of noise before the signal risks being misinterpreted as logic 1.

The high noise margin is:

$$NM_H = V_{OH} - V_{IH} = 3.84V - 3.15V = 0.69V$$

This means that when the circuit outputs a logic 1, it can tolerate up to 0.69V of noise before being mistaken for logic 0.

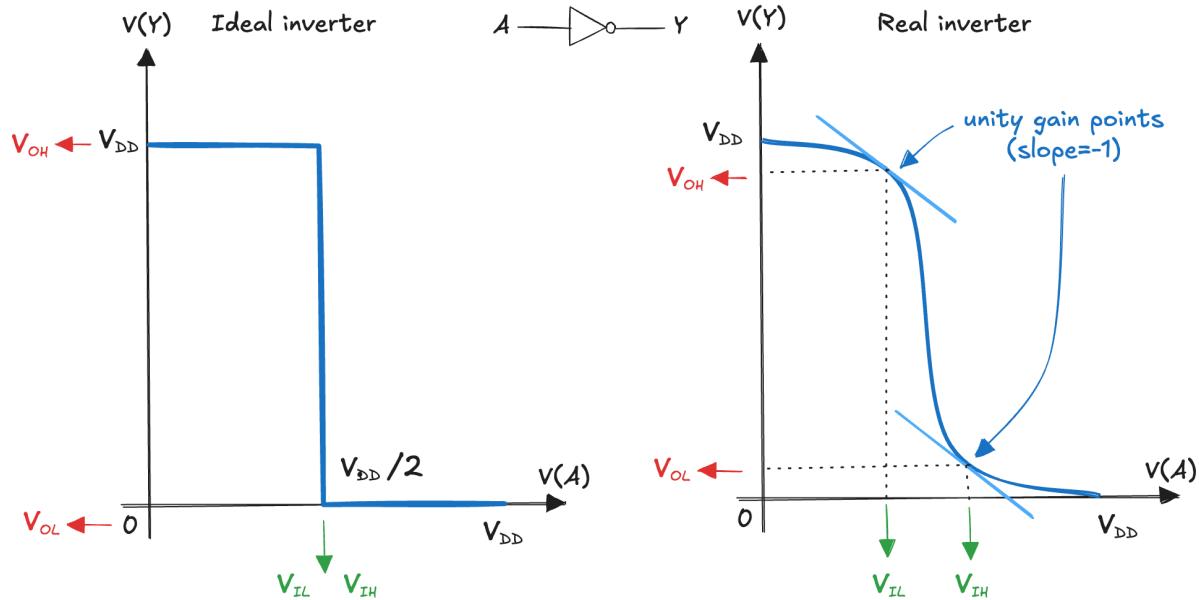
Now suppose that 1 V of noise is injected between the two inverters. When the output is LOW, the circuit can tolerate it, since $NM_L = 1.02V > 1V$, however when the output is HIGH, the circuit cannot tolerate it, since $NM_H = 0.69V < 1V$.

This example demonstrates how the robustness of a digital circuit depends on the size of the noise margins. A well-designed technology ensures that both NM_L and NM_H are **sufficiently large to guarantee reliable communication between gates, even in the presence of disturbances**.

4.1.3 DC Transfer Characteristics

In the previous discussion, we saw that reliable digital systems require us to define logic levels: specific voltage ranges that represent 0 and 1. But where do these values come from? **How**

do we decide the boundaries between logic low and logic high? To answer this, we must **look beneath the digital abstraction** and **examine the analog behavior of a gate**. Even though we think of gates as purely digital devices, in reality their operation is governed by continuous voltages and currents. The tool we use to describe this behavior is the **DC transfer characteristic**, which shows how the output voltage of a gate depends on its input voltage when the input is varied slowly. We can consider an ideal and a real inverter:



If digital abstraction were perfect, an inverter would behave like an **ideal switch**. Its output $V(Y)$ would remain at V_{DD} (logic 1) for all input voltages $V(A)$ below $V_{DD}/2$, and would abruptly jump to 0V (logic 0) as soon as the input exceeded $V_{DD}/2$. In this case:

$$V_{IL} = V_{IH} = \frac{V_{DD}}{2}, \quad V_{OH} = V_{DD}, \quad V_{OL} = 0$$

The digital abstraction would be exact: below the threshold the signal is a 0, above it the signal is a 1. Real circuits, however, **do not switch instantaneously**. The output of a physical inverter decreases gradually as the input increases:

- when $V(A) = 0$, the output is close to V_{DD}
- when $V(A) = V_{DD}$, the output is close to 0
- in between, the transition is **smooth rather than abrupt**

This gradual slope forces us to define more carefully where the boundary between logic 0 and logic 1 lies. The standard approach is to examine the slope of the transfer characteristic. The **critical points** are where the slope start to steepen significantly:

$$\frac{dV(Y)}{dV(A)} = -1$$

These **unity-gain points** define the thresholds:

- for $V(A) < V_{IL}$, the output is guaranteed to be a valid logic 1 (near V_{OH})
- for $V(A) > V_{IH}$, the output is guaranteed to be a valid logic 0 (near V_{OL})
- between V_{IL} and V_{IH} , the output is undefined, this is the forbidden zone.

Defining logic levels in this way usually maximizes the noise margins, ensuring that the digital system remains reliable despite noise and imperfections.

4.1.4 Logic Families

At this point, we have seen that logic levels must be carefully defined, and that noise margins provide robustness against imperfections. But there is a deeper design principle that ensures the entire digital system functions reliably: the **static discipline**. It states that **if a circuit element is given logically valid inputs, it must always produce logically valid outputs**. In other words, as long as the inputs fall within the acceptable ranges for logic 0 or logic 1, the output will also fall into the valid ranges. This guarantee allows us to build complex digital systems from simple components without worrying about the analog details of every transistor or wire.

We can think of the static discipline as a kind of **grammar for digital circuits**. Just as a language has rules of grammar that ensure words combine to form meaningful sentences, the static discipline ensures that gates, when connected correctly, produce valid digital behavior:

- if each “word” (gate) follows the rules, then every “sentence” (the overall circuit) will make sense
- if a “word” violates the rules, the sentence becomes meaningless, just as a gate producing an invalid output voltage can corrupt the whole digital system.

This analogy helps us see why the static discipline is so powerful: it provides a framework that guarantees consistency, enabling complexity to be managed and systems to scale. Adopting the static discipline does mean **sacrificing some freedom**. We cannot arbitrarily connect any analog circuit and expect it to behave digitally. But this restriction is what gives digital design its **simplicity and robustness**:

- it allows engineers to ignore the continuous voltages and currents at the transistor level and instead reason in terms of abstract 0 and 1
- it raises the level of abstraction: from analog electronics to digital logic, where complexity is managed by hiding needless detail
- it ensures that as signals propagate through billions of gates, they remain unambiguous digital values.

Without the static discipline, noise would accumulate, thresholds would drift, and eventually the digital abstraction would collapse. One important consequence is that gates that communicate with each other **must use compatible logic levels**. The actual choice of supply voltage and thresholds is arbitrary, but **consistency is essential**.

For this reason, logic gates are grouped into **logic families**, from the 1970s through the 1990s, four major logic families dominated digital circuit design:

Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75–5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5–6)	1.35	3.15	0.33	3.84
LVTTL	3.3 (3–3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3–3.6)	0.9	1.8	0.36	2.7

Each family is **internally consistent**: any gate in the family can drive any other gate in the same family while obeying the static discipline. Second, families differ in their supply voltages and thresholds, which can create **compatibility issues** when mixing technologies:

Driver	Receiver: TTL	Receiver: CMOS	Receiver: LVTTL	Receiver: LVCMOS
TTL	OK	NO: $V_{OH} < V_{IH}$	MAYBE	MAYBE
CMOS	OK	OK	MAYBE	MAYBE
LVTTL	OK	NO: $V_{OH} < V_{IH}$	OK	OK
LVCMOS	OK	NO: $V_{OH} < V_{IH}$	OK	OK

For example, TTL cannot reliably drive CMOS, because its guaranteed high output voltage is lower than the minimum high input voltage required by CMOS. In other cases, the answer is “maybe” because the margin is small and depends on manufacturing variations and operating conditions.

This shows why logic families are so important: they ensure that, within a family, **gates “snap together” like Lego bricks**, but crossing between families requires caution or additional interfacing circuits.

As technology progressed, a proliferation of logic families with **lower supply voltages** appeared. Moving from 5V to 3.3V, and then to 2.5V, 1.8V, and below, was driven by the need for **lower**

power consumption. Each reduction in supply voltage requires a careful redefinition of logic levels and noise margins, but the static discipline ensures that digital abstraction remains intact.

4.2 Implementing Logic Gates

At this point we understand that at the heart of digital system lie logic gates. These simple building blocks, capable of performing basic logical operations can be combined to form adders, memory elements, processors, and all the digital circuits we can imagine. However, a **logic gate is an abstraction:** it takes one or more binary inputs and produces a binary output. Yet, to become real, it must be **embodied in some physical device** capable of switching between two distinct states. A fundamental question arises: **how can we physically implement a logic gate?**

4.2.1 Gears and vacuum tubes

The history of computing is, in large part, the story of how engineers and scientists have found different ways to realize this abstraction. The earliest serious attempt was Charles Babbage's Analytical Engine in the 19th century. In this machine, **gears and levers** were used to represent logical states and perform operations. The approach worked in principle, but the machines were massive, complex, and slow. **Mechanical computation demonstrated feasibility but not practicality.** In the early 20th century, computation entered the electrical era with the use of **vacuum tubes.** A vacuum tube could act as an electronic switch: depending on the applied voltage, it would allow or block current flow:

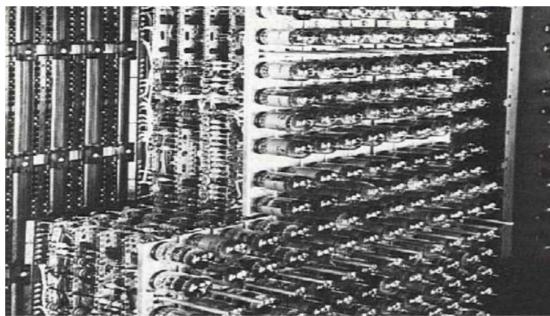


Babbage Differential Engine, 1823



The original triode vacuum tube, 1906

This property enabled the construction of the first electronic computers. Machines such as the IBM 701 (1952) contained **thousands of tubes** and could process thousands of words of data at unprecedented speeds compared to mechanical engines:



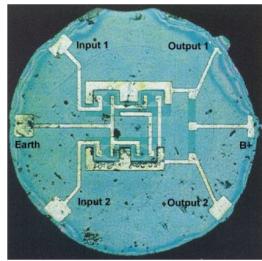
IBM 701 (1952), 2048 words of 36 bit

However, vacuum tubes came with **severe limitations**. They were large, fragile, power-hungry, and unreliable, producing significant heat and often burning out after only a few thousand hours of operation. Entire rooms were required to house tube-based computers, and frequent maintenance was necessary.

Thus, from the very beginning, the central problem in building digital systems has been finding a physical technology that can implement gates **efficiently, reliably, and at scale**. Mechanical gears were too slow, and vacuum tubes were too bulky and unreliable. The solution came in 1947 with the invention of the transistor, a device that would revolutionize electronics by offering a small, robust, and energy-efficient switch: the perfect foundation for digital logic. Yet the true **revolution** came not from the transistor alone, but from the ability to **integrate** large numbers of transistors onto a single chip of silicon. This breakthrough, known as the **integrated circuit (IC)**, transformed electronics by making it possible to build complex digital systems that were compact, fast, and affordable.

4.2.2 Integrated Circuits

The origins of the integrated circuit trace back to two independent inventions at the end of the 1950s. In 1958, Jack Kilby at Texas Instruments demonstrated the first working integrated circuit, in which multiple components were fabricated together on a single piece of semiconductor material. In 1959, Robert Noyce at **Fairchild Semiconductor** went further, patenting a practical method for interconnecting many transistors on a single silicon chip. Noyce's contribution solved the critical issue of wiring and scalability, laying the foundation for modern microelectronics:



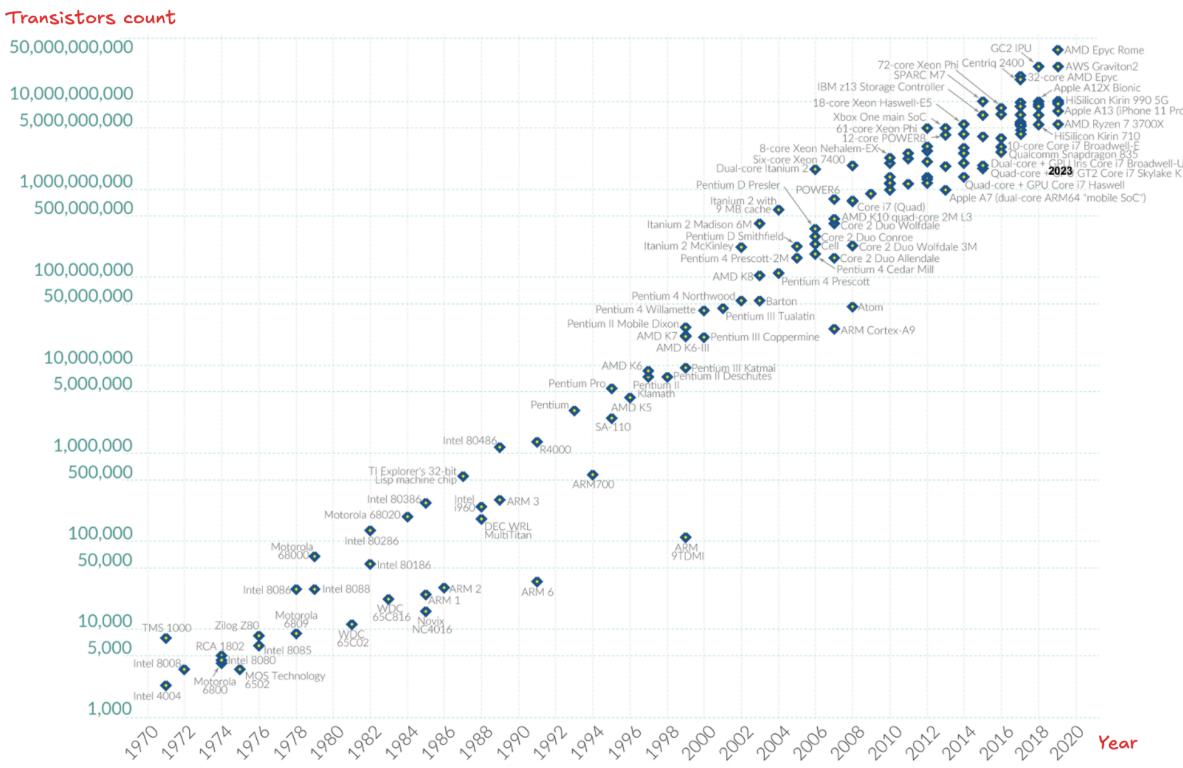
First silicon IC Chip, 1961

Robert Noyce, 1927–1990

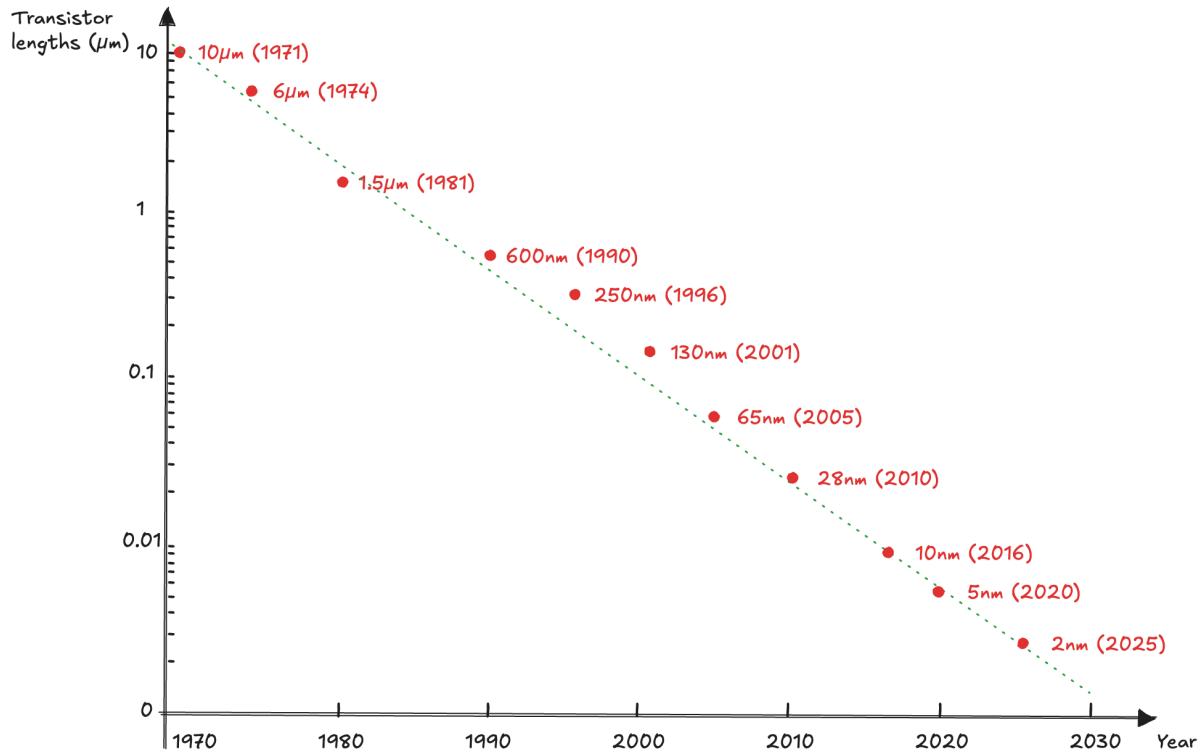
Born in Burlington, Iowa. Received a B.A. in physics from Grinnell College and a Ph.D. in physics from MIT. Nicknamed "Mayor of Silicon Valley" for his profound influence on the industry. Cofounded Fairchild Semiconductor in 1957 and Intel in 1968. Co-invented the integrated circuit. Many engineers from his teams went on to found other seminal semiconductor companies.

The impact was dramatic. In the 1950s, a single transistor could cost around 10 dollars, and building a system with thousands of them was prohibitively expensive. Today, decades of technological progress allow us to place over one billion transistors on a chip as small as 1cm², with the effective cost of each transistor dropping below 10 micro-cents. This **extraordinary increase in density and decrease in cost** has been the engine driving the entire digital revolution.

The invention of the integrated circuit opened the door to unprecedented growth in computing power. But how fast could this growth continue? In 1965, Gordon Moore, cofounder of Intel, made a striking observation: **the number of transistors on an integrated circuit was doubling roughly every year**. A decade later, the rate was refined to a doubling about every two years. This empirical observation became known as **Moore's Law**. It is not a law of physics, but rather an engineering and economic trend. It captures the fact that advances in manufacturing, materials, and design techniques have consistently enabled the semiconductor industry to shrink transistor sizes, allowing more and more devices to be placed on the same area of silicon. The following graph illustrates this phenomenon: from the early 1970s to the present, the number of transistors on microchips has increased from a few thousand to tens of billions, following a remarkably steady exponential trajectory. Each point represents a commercial microprocessor, showing how the trend has held for over five decades:



One of the most visible consequences of Moore's Law has been the **steady shrinking of transistor dimensions**. The figure shows the characteristic minimum feature size (often referred to as the "process node") plotted against time:



Each new generation of technology is defined by the length of the transistor (more specifically, its "gate length"), measured in micrometers (μm) or nanometers (nm). In the early 1970s, commercial microchips used processes with feature sizes of around $10\mu\text{m}$. By the mid-1980s, scaling had already reduced this dimension below $1\mu\text{m}$. The trend continued through the 1990s and 2000s, reaching 250nm (1996), 130nm (2001), and 65nm (2005). More recent generations have pushed into the deep nanometer regime, with 28nm (2010), 10nm (2016), 5nm (2020), and the current development of 2nm nodes or less in 2025.

The consequences have been revolutionary. As transistor counts increased, so did processing speed, memory capacity, and overall performance, while the cost per transistor decreased dramatically. This **exponential growth fueled the information age**, making possible everything from personal computers and smartphones to large-scale data centers. In the last decade, however, **the pace of scaling has slowed since some challenges have emerged**. As transistor dimensions approach the atomic scale, engineers face **fundamental physical barriers**, including: quantum effects, heat dissipation, manufacturing challenges, and material limits. Because of these factors, the continuous scaling that characterized Moore's Law for decades is now slowing down.

How can the industry continue to improve performance when scaling alone is no longer enough? The answer lies in moving "**Beyond Moore**": instead of relying solely on smaller transistors, engineers are exploring new strategies for extending computing capability.

One response has been to place **multiple processors** on a single chip. Instead of making one processor faster, manufacturers create multi-core processors that divide tasks across several cores. This approach has allowed performance to keep improving, even as clock speeds plateaued due to power and heat constraints.

Another innovation is **3D integration**, where circuits are stacked vertically rather than spread out only in two dimensions. This reduces communication delays between components and increases density without requiring further shrinking of individual transistors.

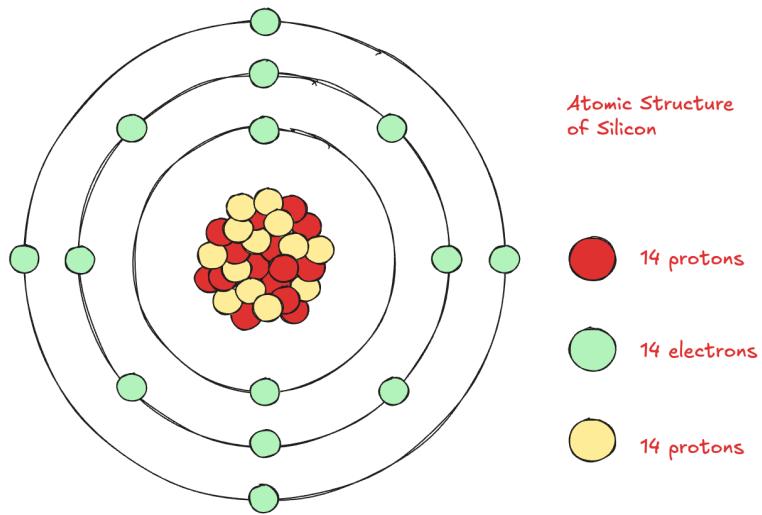
Looking further ahead, researchers are exploring alternative devices and architectures: **new materials** such as graphene, carbon nanotubes, and 2D semiconductors promise faster switching and lower power; **specialized accelerators**, like GPUs and AI chips, achieve massive gains in performance for particular tasks; and **quantum computing** aims to go beyond classical logic, exploiting entirely new principles of information processing.

4.3 Electron Devices

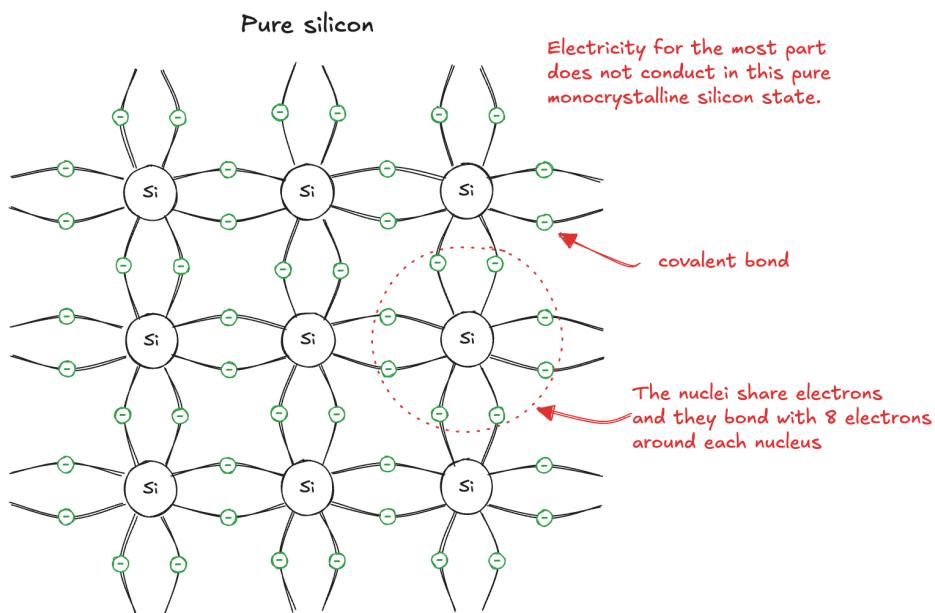
Digital systems are built upon a small set of fundamental building blocks known as **electron devices**. These are components specifically engineered to control the flow of electric charge through materials. Unlike passive elements such as resistors or capacitors, electron devices exploit the properties of semiconductors to actively switch, amplify, and process signals.

4.3.1 Semiconductors

Modern transistors are built from **MOS (Metal-Oxide-Semiconductor) technology**, and the key material used is **silicon (Si)**. Silicon is so widely adopted because of its unique electronic properties. A silicon atom has a nucleus containing 14 protons and typically 14 neutrons, with 14 electrons orbiting the nucleus in shells. Its electron configuration is:



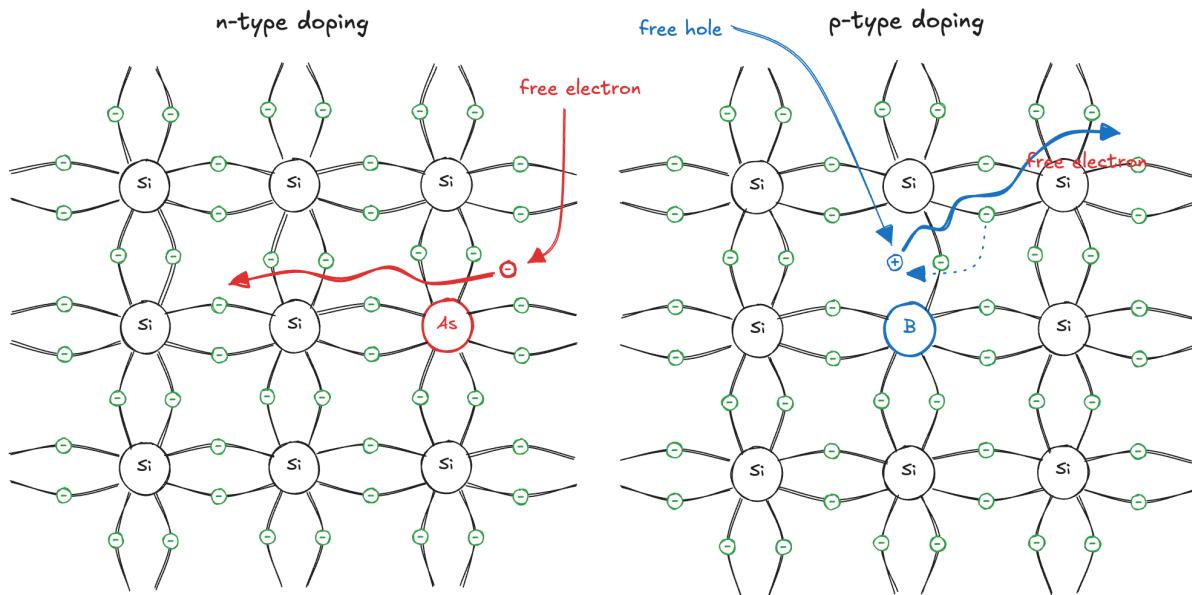
The outermost four electrons, known as **valence electrons**, play a crucial role in chemical bonding. In a crystal structure, each silicon atom forms four covalent bonds with neighboring atoms in a producing a very stable and regular structure, creating a rigid and strong lattice similar to that of diamond:



In this pure form, silicon is actually a **poor conductor**: all the electrons are locked into covalent bonds, leaving no free carriers to conduct electricity.

The magic of **semiconductors** lies in the ability to **tune their conductivity** by adding very small amounts of impurities, called **dopants**. Depending on the dopant, silicon can be made to conduct via two different mechanisms:

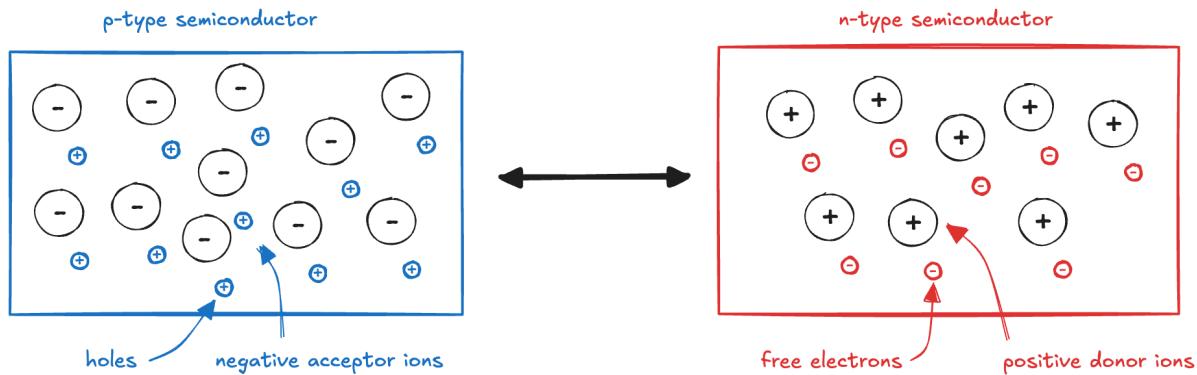
- **n-type doping:** if a dopant atom like arsenic (As) is introduced, it brings **five valence electrons** instead of four. Four of them bond with neighbors, but the extra electron remains free. This **free electron** can move through the lattice, carrying a negative charge.
- **p-type doping:** if a dopant atom like boron (B) is used, it brings only **three valence electrons**. One bond in the lattice is left incomplete, creating a vacancy called a **hole**. An electron from a neighboring atom can move to fill this gap, leaving another hole behind. To the outside observer, it appears as though **the hole itself moves**. Since a hole represents the absence of a negative charge, it behaves like a positively charged particle.



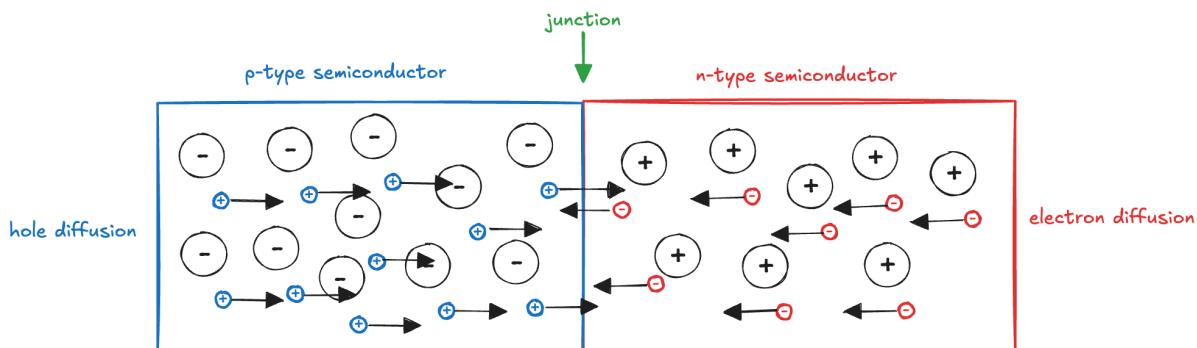
The key property of silicon is that its conductivity can be varied over many orders of magnitude simply by adjusting the concentration of dopants. This flexibility is what makes silicon neither a perfect conductor like copper, nor a perfect insulator like glass, but something in between: a **semiconductor**. This tunable behavior is what allows us to construct the fundamental devices of digital electronics: diodes and transistors.

4.4 The p-n junction

A p–n junction is the **fundamental building block** of all semiconductor devices. It is formed by placing a region of p-type semiconductor in direct contact with a region of n-type semiconductor. The boundary between the two regions is called the **junction**:

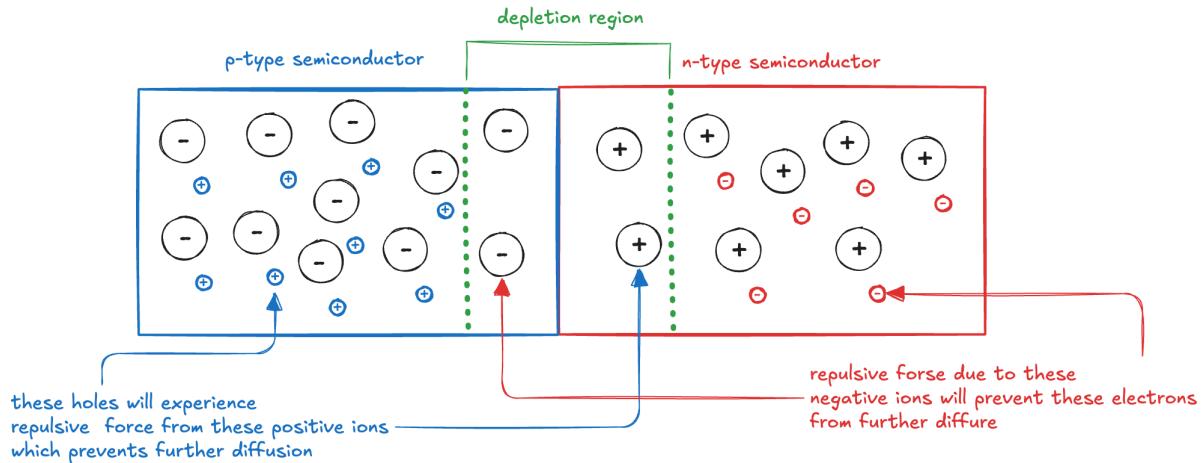


The p-type semiconductor contains **mobile holes as majority carriers**, balanced by an equal number of **immobile negatively charged acceptor ions**. Similarly, the n-type semiconductor contains **free electrons as majority carriers**, balanced by an equal number of **immobile positively charged donor ions**. In each region, the mobile carriers are **uniformly distributed** throughout the material, and overall electrical neutrality is maintained because the number of positive and negative charges is equal. When a p-n junction is formed, **charge carriers begin to diffuse across the boundary**. Holes from the p-type side migrate into the n-type side, while electrons from the n-type side migrate into the p-type side. This diffusion occurs because of a **concentration gradient**: the density of holes is higher on the p-side than on the n-side, and the density of electrons is higher on the n-side than on the p-side:

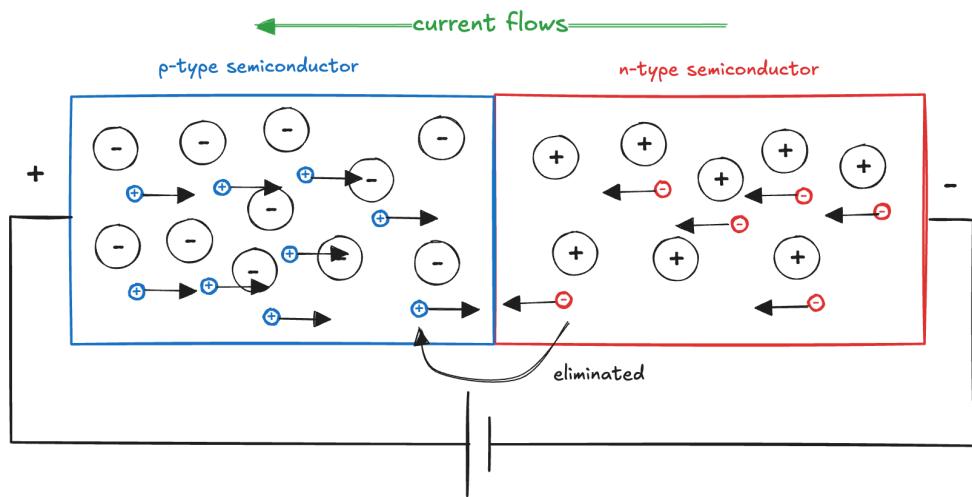


This process is called **diffusion**. As free electrons cross from the n-side into the p-side, they **recombine with holes**. In doing so, they **leave behind positively charged donor ions** (atoms that now have one fewer electron than protons) fixed in the crystal lattice on the n-side. At the same time, the electrons that enter the p-side create **negatively charged acceptor ions** (atoms that now have one more electron than protons). As a result, a region near the junction becomes filled with fixed positive ions on the n-side and fixed negative ions on the p-side. This region is called the **depletion region**, because it is depleted of mobile carriers. The fixed charges create an **internal electric field**: the p-side becomes negative, and the n-side becomes positive. This electric field **opposes further diffusion of charge carriers** across the junction. In other words, the

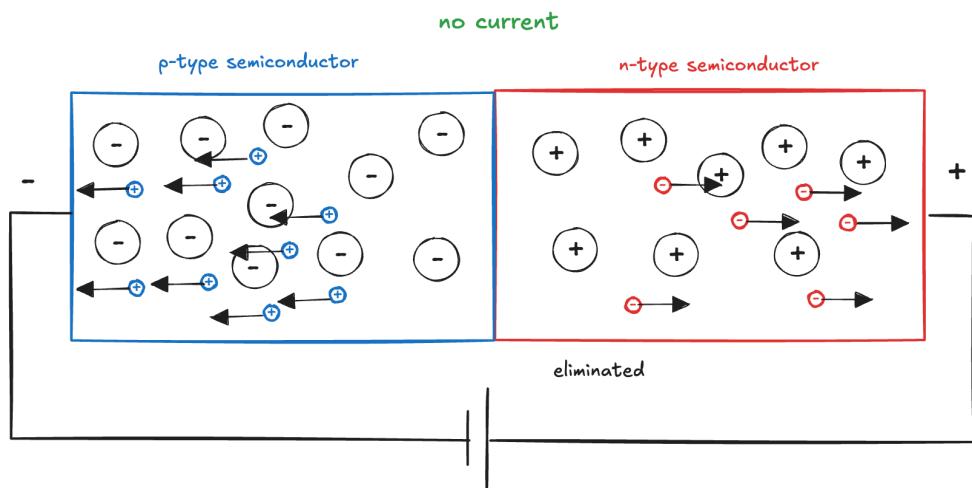
initial diffusion sets up a barrier potential (also called the **built-in potential**) that prevents additional electrons from crossing from n to p and additional holes from crossing from p to n. For a silicon junction, the barrier potential is typically about 0.7 V under equilibrium conditions.



The region around the junction is termed as **depletion region** because the mobile charge carriers (free electrons and holes) have been emptied in this region. When an **external voltage** is applied to a p-n junction such that the **p-side is made positive relative to the n-side (forward bias)**, the applied field works against the built-in potential of the depletion region. As the applied voltage approaches the built-in potential (about 0.7 V for silicon), the barrier is reduced and the depletion region narrows. This allows holes in the p-side to be pushed toward the junction, while electrons in the n-side are also driven toward the junction. When these carriers meet at the boundary, electrons recombine with holes, effectively neutralizing one another. However, the external voltage source continually supplies new carriers: more electrons are injected from the n-side, and more holes are injected from the p-side. This **continuous injection and recombination process sustains a steady current flow** across the junction. The applied voltage maintains the supply of carriers that replace those recombined at the interface, so the current persists as long as the forward bias is applied:



When an external voltage is applied to a p-n junction such that the **p-side is made negative relative to the n-side (reverse bias)**, the applied field acts in the same direction as the built-in potential of the depletion region. As a result, the depletion region widens, and the barrier potential increases. Under these conditions, the majority carriers (holes in the p-side and electrons in the n-side) are pulled away from the junction, preventing them from crossing. This effectively blocks current flow. If the reverse bias is increased beyond a critical value, called the **breakdown voltage**, the junction can suddenly conduct a large current.



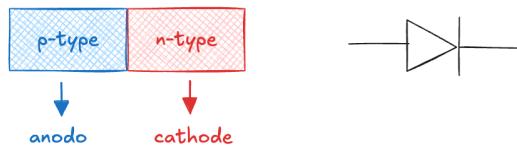
4.4.1 Diode

A **diode** is essentially a junction provided with two external terminals:

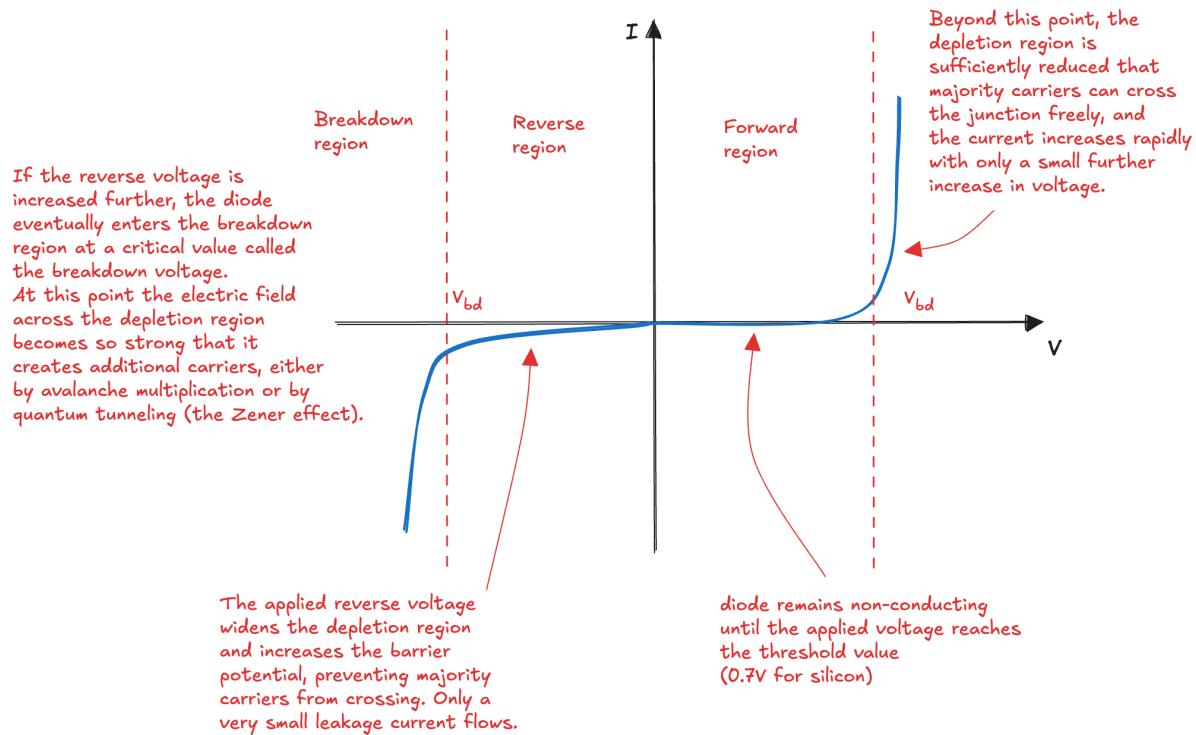
- the **anode**, connected to the p-type region

- the **cathode**, connected to the n-type region.

Encapsulating the junction in a practical component with external contacts allows engineers to exploit its properties in real circuits. In circuit diagrams the diode is represented by a triangle pointing toward a vertical bar: the triangle corresponds to the anode (p-side) and the bar to the cathode (n-side):



It behaves as a **one-way valve for current**. When the anode is made positive with respect to the cathode, the applied voltage counteracts the built-in potential, the depletion region narrows, and majority carriers are able to cross the junction. Once the applied voltage exceeds a certain threshold (about 0.7V in silicon) current flows readily. In the opposite condition, when the anode is negative relative to the cathode, the applied voltage reinforces the barrier potential, the depletion region widens, and majority carriers are repelled. In this case the diode blocks current. The behavior of a diode can be fully understood by looking at its **current-voltage (I-V) characteristic**:



Although simple, the diode plays an essential role in electronics. Its unidirectional conduction property is exploited in rectifiers that convert AC into DC, in clamping and protection circuits, in

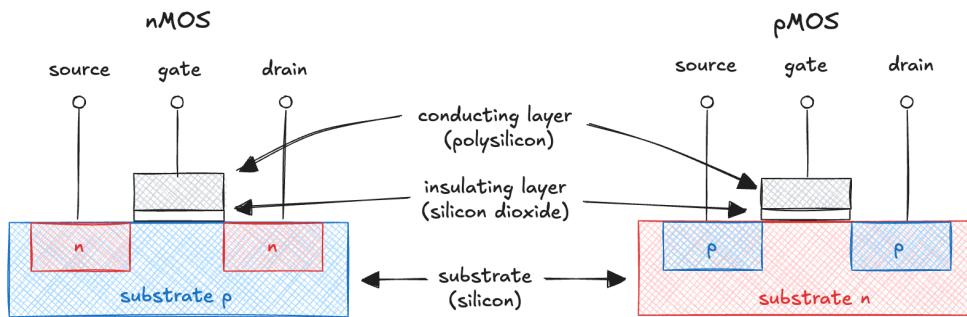
demodulation of signals, and in voltage regulation when operated in breakdown mode (Zener diode). From this elementary device, the foundation was laid for more complex components such as transistors and integrated circuits, but even today the diode remains one of the most indispensable elements in modern circuits.

4.4.2 MOS Transistor

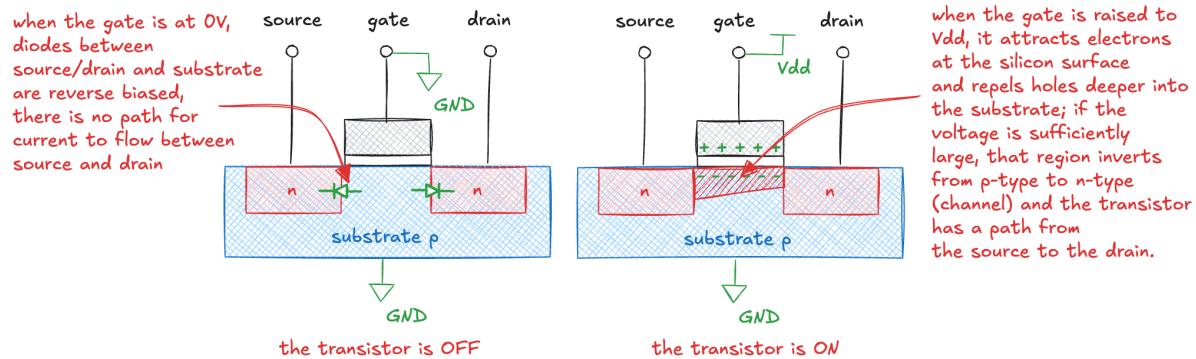
To understand how digital circuits work, we need to examine the **MOSFET (Metal–Oxide–Semiconductor Field-Effect Transistor)**, the most widely used electronic device: every logic gate, memory cell, and microprocessor is ultimately made from billions of MOS transistors. At its core, it is a controllable switch built on a piece of silicon. Imagine starting with a block of semiconductor (the **substrate**). Two regions within this substrate are **heavily doped** to provide a high density of charge carriers. This basic principle comes in two complementary forms:

- **nMOS**: the substrate is p-type, and the two doped regions are n-type, providing free electrons as carriers
- **pMOS**: the substrate is n-type, and the two doped regions are p-type, providing holes as carriers.

These two regions will later serve as the ends of the switch, one acting as the **entry point for carriers** (the **source**) and the other as the **exit point** (the **drain**). Above the space between source and drain lies a thin **insulating layer** of silicon dioxide, and on top of it a **conductive plate** called the **gate**. The gate does not touch the silicon directly; instead, it influences it through an electric field. By applying a voltage to the gate, we can decide whether or not carriers are allowed to flow between source and drain:



The MOS transistor works **like a switch** controlled by voltage rather than by mechanical movement. When the gate voltage is low, no conductive path exists, and the switch is OFF. When the gate voltage exceeds a certain threshold, it creates a **channel** in the silicon beneath, allowing current to flow freely from source to drain, the switch is ON. In order to understand this behavior, we can consider the nMOS device:



When the gate voltage is zero, the p–n junctions between the n-type source/drain and the p-type substrate are reverse biased (since source/drain never goes below the minimum potential). Under these conditions, there is no conductive path between source and drain, and no current can flow through the device. When a positive voltage is applied to the gate relative to the substrate, the electric field created by the gate penetrates through the oxide into the semiconductor. This field repels the holes in the substrate and attracts electrons toward the surface just under the gate. As the gate voltage increases, more electrons accumulate and when the voltage reaches a critical value (**threshold voltage**), the concentration of electrons becomes high enough to invert the conductivity of the region under the gate. The surface of the substrate, originally p-type, is transformed into a thin n-type **channel** that connects the source and the drain. With this inversion layer in place, electrons can now flow from the source to the drain, when a voltage is applied between them. The transistor provides a conductive path controlled entirely by the gate voltage.

The pMOS transistor is the **complementary** counterpart of the nMOS. The substrate is connected to the supply voltage. When the gate–substrate voltage is zero (that is, when the gate is also at the supply voltage), the p–n junctions between the source/drain and the substrate are reverse biased. No channel exists under the gate, and the transistor is OFF. If the gate is brought down toward 0V, the gate–substrate voltage becomes negative. The electric field repels electrons from the surface of the substrate and attracts holes. Once the gate–substrate voltage goes below the threshold voltage, the surface inverts from n-type to p-type, forming a conductive channel that connects source and drain. With this inversion layer in place, holes can flow across the device, and the transistor turns ON.

Summarizing:

- in an nMOS, a positive gate voltage induces an n-type channel for electrons
- in a pMOS, a zero gate voltage induces a p-type channel for holes

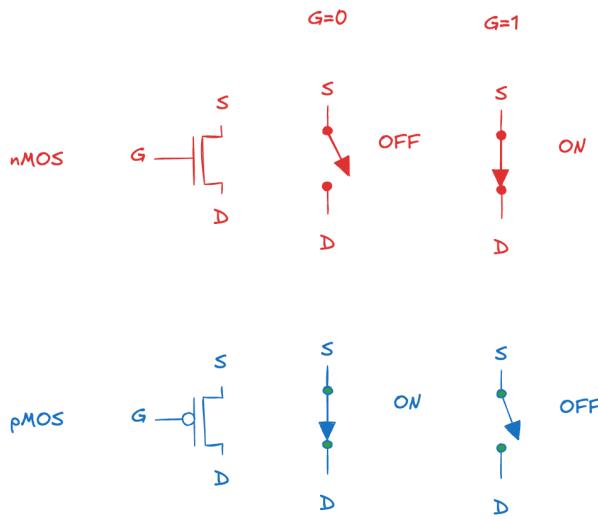
The nMOS and pMOS transistors are represented with simple symbols that indicate their type and connections. Both symbols have three terminals: the gate (control input), the drain

(where current leaves), and the source (where current enters). The substrate is tied to ground for nMOS and to power supply for pMOS, and is omitted from the symbol for simplicity:



4.4.3 The switch model

From the perspective of digital design, a MOS transistor can be understood as a **voltage-controlled switch**. The gate voltage determines whether a conductive path exists between the source and the drain. For an nMOS transistor, the switch is OFF when the gate is at logic 0 (0V), and it turns ON when the gate is at logic 1 (power supply). In other words, an **nMOS conducts when its gate is high**. For a pMOS transistor, the behavior is the opposite. The switch is ON when the gate is at logic 0, and OFF when the gate is at logic 1. A **pMOS conducts when its gate is low**:



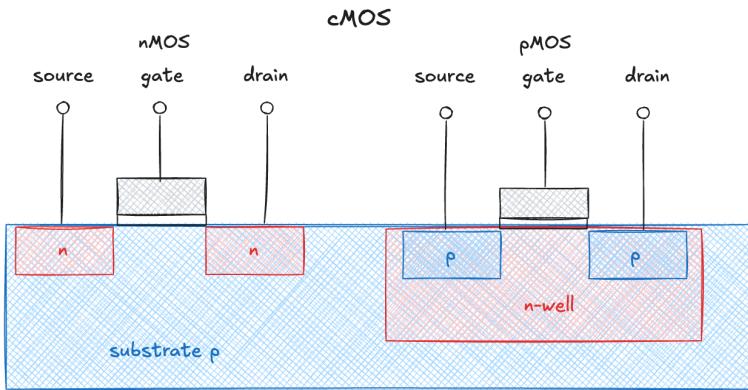
Although MOS transistors can be modeled as switches, they are **not perfect switches**. An nMOS transistor turns on when its gate is at supply voltage. If the source is at 0V, it easily pulls the drain node down to 0, because the gate is still much higher than the substrate and the channel is strong. In this case, the **nMOS "passes" a logic 0 very well**. But if the source is at supply voltage and the drain is supposed to rise toward supply voltage, the situation changes. As the drain node approaches the gate voltage, the electric field under the gate weakens, the inversion layer disappears, and the output stalls below the full supply voltage. An **nMOS cannot pass a strong 1**. The behavior of a pMOS is the exact complement. With its substrate tied to supply

voltage, it turns on when the gate is low. If the source is at supply voltage, the pMOS can pull the drain node cleanly up to the supply voltage, because the gate–substrate field is strongly negative and the channel is robust. It therefore **passes a logic 1 well**. But if the source is at 0, the drain node cannot be pulled all the way to ground. As the node approaches 0, the gate and the channel are at the same potential, the inversion layer fades, and the output stalls above 0. A **pMOS cannot pass a strong 0**. The intuition is simple:

- nMOS transistors act like **strong switches to ground** but **weak switches to high**
- pMOS transistors act like **strong switches to high** but **weak switches to ground**.

4.4.4 CMOS

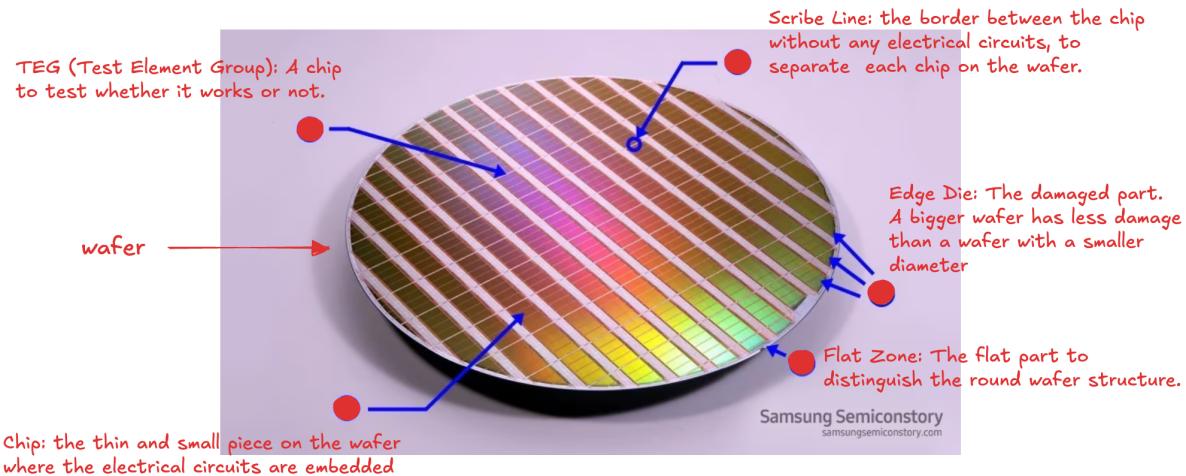
To build digital logic efficiently, **both nMOS and pMOS transistors must be fabricated on the same chip**. The challenge is that each type of transistor requires a different substrate: nMOS devices need a p-type substrate, while pMOS devices need an n-type substrate. The solution is the **n-well process**. Modern manufacturing starts with a p-type wafer, which directly supports nMOS fabrication, and then implants localized n-type regions, called **wells**, into which pMOS transistors are built. This makes it possible to integrate both types of transistors side by side on a single piece of silicon:



This approach is known as **complementary MOS**, or **CMOS**, and it has become the dominant technology for digital electronics. In CMOS circuits, nMOS and pMOS transistors work together to implement logic functions. The nMOS devices provide a strong connection to ground, ensuring that outputs can be pulled fully to 0, while the pMOS devices provide a strong connection to the supply voltage, ensuring that outputs can be pulled fully to high}. In this way, CMOS gates overcome the weaknesses of using only one type of device.

4.4.5 Building Integrated Circuits

Modern integrated circuits are fabricated on thin, flat slices of silicon known as **wafers**. These wafers are cut from large, single-crystal ingots of silicon and polished to extreme smoothness. Today's production wafers typically have a diameter of about 30 cm, allowing thousands of individual circuits to be manufactured at once on a single piece of silicon:



On the surface of the wafer, advanced manufacturing processes create microscopic transistors by layering and patterning materials with **nanometer precision**. Each transistor is only a fraction of a micron in length, which makes it possible to fabricate billions of them at a time at relatively low cost. The ability to manufacture so many devices simultaneously is one of the main reasons why integrated circuits have become so inexpensive and widespread.

Once all the devices and interconnections have been built into the wafer, the wafer is cut into small rectangular sections called **chips** or **dies**. Each chip contains a complete electronic circuit, which may consist of thousands, millions, or even billions of transistors depending on its complexity. These chips are then packaged and tested before being assembled into computers, smartphones, and countless other electronic systems.

Because the features on a modern wafer are so small, even a single dust particle or strand of hair can destroy or degrade a circuit. For this reason, wafers are manufactured in **clean rooms**, highly controlled environments where the air is constantly filtered to remove contaminants. Technicians wear special "**bunny suits**" that cover their entire body, preventing particles from their hair, skin, or clothing from contaminating the wafers:



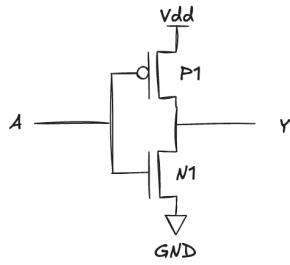
The combination of large wafer sizes, microscopic feature dimensions, and clean-room manufacturing enables the production of integrated circuits at the scale required by today's digital world. From each wafer emerge hundreds or thousands of chips, each capable of performing billions of operations per second, forming the foundation of all modern electronics.

4.5 CMOS logic gates

The true power of CMOS technology emerges when we combine nMOS and pMOS transistors to build logic gates. The basic idea is simple, by arranging nMOS and pMOS networks in a complementary way, we can construct any logic gate that produces full logic levels with very low static power consumption.

4.5.1 NOT Gate (Inverter)

The simplest CMOS circuit is the inverter. It consists of a pMOS at the top, connected to power supply, and an nMOS at the bottom, connected to ground, connected together to form the output:

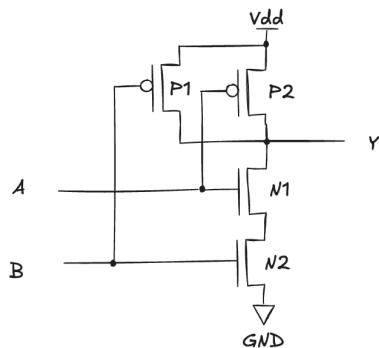


- if the input $A=0$, the nMOS is OFF and the pMOS is ON, so the output is connected to power supply and pulled up to logic 1
- if $A=1$, the nMOS is ON and the pMOS is OFF, so the output is connected to ground and pulled down to logic 0

This complementary action guarantees that the inverter always produces the opposite logic value of its input.

4.5.2 NAND Gate

The NAND gate can be constructed by connecting two nMOS transistors in series and two pMOS transistors in parallel:

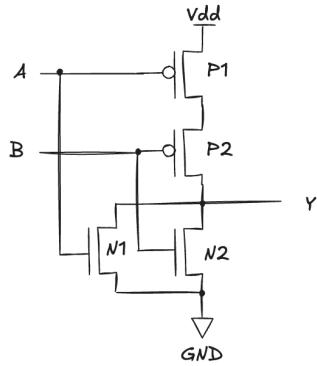


- if both inputs A and B are 1, the two nMOS transistors conduct, creating a path to ground that pulls the output to 0
- in all other cases, at least one of the pMOS transistors is ON, which connects the output to power supply, making it 1

Thus, the output is 0 only when both inputs are 1, which is exactly the NAND function.

4.5.3 NOR Gate

The NOR gate is built as the dual of the NAND. Here, the nMOS transistors are in parallel and the pMOS transistors are in series:

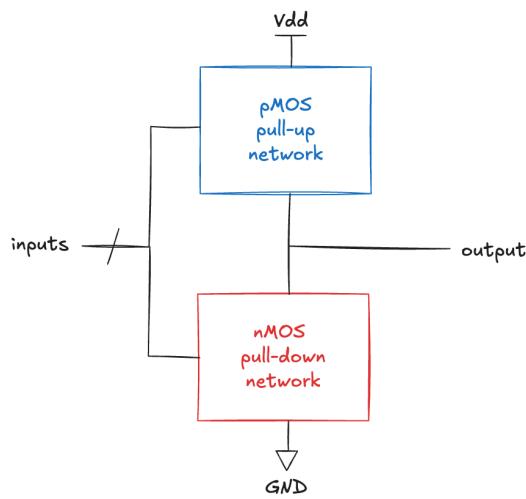


- if either input is 1, one of the nMOS devices conducts, connecting the output to ground and producing a 0
- only when both inputs are 0 do the pMOS transistors conduct together, pulling the output up to 1.

This behavior corresponds to the NOR function.

4.5.4 General Rule for Inverting Logic

From the examples of the inverter, NAND, and NOR, we can **generalize the structure of CMOS logic**. Every gate consists of two complementary parts: a **pMOS pull-up network** that connects the output to power supply, and an **nMOS pull-down network** that connects the output to ground. The input signals are applied to both networks in such a way that exactly one of them conducts in any stable state:



- if the pull-down network is ON, the output is discharged to 0
- if the pull-up network is ON, the output is charged to 1

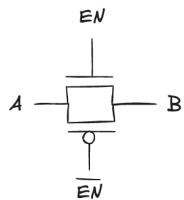
The arrangement of transistors inside each network follows a simple rule. In the nMOS pull-down network, transistors placed in parallel mean the output is pulled down if any input is 1, while transistors in series mean the output is pulled down only if all inputs are 1. The pMOS pull-up network is always built as the logical complement: series in the nMOS network corresponds to parallel in the pMOS network, and parallel in the nMOS network corresponds to series in the pMOS network.

This complementary organization guarantees correct logic behavior and ensures that there is never a direct conducting path between power supply and ground when the output is stable. It is this property that gives CMOS logic its extremely low static power consumption.

Among the basic logic functions, the **NAND and NOR gate is particularly simple and efficient to implement in CMOS**. Their compact configuration requires only four transistors. Historically, this simplicity had major consequences. In early integrated circuit families such as TTL (Transistor-Transistor Logic) and later CMOS, engineers quickly realized that implementing AND or OR directly was more complex and required extra transistors. NAND and NOR, on the other hand, were much cheaper in terms of transistor count and area on the silicon wafer. For this reason, libraries of logic gates in early ICs were based primarily on NAND and NOR. This tradition persists: even today, synthesis tools for digital circuits often map logic into networks of NAND or NOR gates, since these are the most efficient primitives in CMOS.

4.5.5 Transmission Gate

We have seen that nMOS and pMOS transistors, when used individually as switches, are not perfect: nMOS passes logic 0 well but fails to pass a strong 1, while pMOS passes logic 1 well but fails to pass a strong 0. In order to have a good switch in every situation, we can combine them in parallel, so that each device handles the logic level it is best at. This arrangement is known as a **CMOS transmission gate**:



A transmission gate consists of one nMOS and one pMOS connected in parallel between two nodes, say A and B. The gates of the two transistors are driven by **complementary enable signals**: the nMOS gate receives EN, while the pMOS gate receives the inverted signal:

- when $EN=0$, both transistors are OFF. In this case, there is no conducting path between A and B, so the transmission gate is open and the two nodes are electrically disconnected.
- when $EN=1$, both transistors are ON. Now the nMOS provides a strong connection for logic 0, and the pMOS provides a strong connection for logic 1.

As a result, any logic value can pass cleanly from A to B or from B to A. The transmission gate behaves like an almost ideal bidirectional switch. It is often used to implement circuits where an ideal switch is needed.

4.6 Power consumption

Every electronic system **consumes power**, which is the **amount of energy used per unit time**. In portable devices, this directly impacts **battery life**, while in plugged systems it determines electricity costs, heat dissipation, and environmental impact.

A typical smartphone battery stores about 10 watt-hours (Wh) of energy. This means it can supply 1 watt of power for 10 hours, or 2 watts for 5 hours, and so forth. To last a full day on a single charge, the average power consumption of a smartphone must therefore remain under about 1 watt. Laptops typically have larger batteries, in the range of 50 to 100 WWh, and during normal operation consume less than 10 watts, much of which is used by the display.

Power consumption is also important for systems that are continuously plugged into the grid. Drawing more power increases electricity costs and contributes to greenhouse gas emissions.

Furthermore, excessive power use generates heat, and if a system cannot dissipate this heat efficiently, it risks overheating and damaging components.

Digital systems consume two kinds of power: **dynamic power** and **static power**. Dynamic power is used when signals change value, as charging and discharging capacitances requires energy every time a node switches between 0 and 1. Static power, on the other hand, is consumed even when the system is idle and no signals are switching. Static consumption arises from leakage currents inside transistors, which become more significant as devices scale down to nanometer dimensions.

Modern chip designers must carefully manage both forms of power. Dynamic power dominates in active circuits, while static power becomes a critical issue for idle states and standby modes. The balance between performance, battery life, and thermal limits makes power consumption one of the central challenges of digital design.

5 Boolean Algebra

Digital signals can take only a limited number of values. In the simplest and most important case, there are only two values: 0 and 1. To manipulate these binary variables, we use a mathematical framework called **Boolean Algebra**, introduced by the mathematician George Boole in the 19th century. The power of Boolean Algebra is that it allows us to **reason about digital circuits without worrying about their physical implementation**. A Boolean variable might be represented by a voltage (high = 1, low = 0), by the position of a mechanical switch, or even by fluid levels in a hydraulic system. From the point of view of Boolean Algebra, all these are equivalent. This abstraction is extremely useful for **hardware designers**, since we can design logic circuits by manipulating equations instead of analyzing transistors, and for **programmers**, since software can be written using logic (true/false) without needing to understand the physical electronics. Later, we will see how Boolean expressions can be represented with logic gates (AND, OR, NOT, etc.) and how these gates form the basis of all digital systems, from simple calculators to modern microprocessors.

5.1 Fundamental Definitions

Before we can manipulate Boolean expressions or design digital circuits, we need to establish some precise terminology. Boolean algebra, like any branch of mathematics, is built upon a small set of well-defined concepts. These definitions will serve as the foundation for everything that follows, from truth tables to simplified logic circuits.

5.1.1 Variable

A Boolean variable is a **discrete variable** that can take only two possible values. By convention, we represent these values as:

- 0, which corresponds to False,
- 1, which corresponds to True.

This simple idea is extremely powerful because it allows us to **describe logical situations mathematically**. For example, a switch can be either off (0) or on (1), a digital signal can be low (0) or high (1), and a condition in a program can be false or true.

5.1.2 Function

Suppose we have a set of Boolean variables:

$$X_1, X_2, \dots, X_n$$

we can define a Boolean function over these variables, as a rule:

$$f(X_1, X_2, \dots, X_n)$$

that assigns a Boolean value (0 or 1) to every possible combination of its input variables:

- the function's output is always binary (either 0 or 1).
- its domain consists of all possible input combinations of the variables. Since each of the n variables can take only two values, the total number of combinations is 2^n .

For example, if we have two variables, there are $2^2 = 4$ possible combinations:

$$(0,0), (0,1), (1,0), (1,1)$$

Two Boolean functions are said to be **equivalent** if they produce the same output for every possible combination of inputs. Equivalence is important because **the same logical behavior can often be expressed in multiple ways**.

As an example, let's consider a simple Boolean function:

$$f(X_1, X_2) = X_1 \cdot \overline{X}_2$$

This means that the function is true if and only if X_1 is 1 and X_2 is 0. The corresponding truth table is:

X_1	X_2	\overline{X}_2	$f(X_1, X_2)$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

This truth table shows clearly how the Boolean function works. For each of the $2^2 = 4$ possible input combinations, we calculate the output step by step.

5.1.3 Complement

Each variable can appear in two forms:

- **true form**, represented simply by the variable itself (e.g., A)
- **complementary form**, represented by a bar above the variable (e.g., \overline{A})

The **complement** is the logical inverse:

- if $A = 0$, then $\overline{A} = 1$
- if $A = 1$, then $\overline{A} = 0$

This simple inversion is one of the fundamental operations of Boolean logic.

5.1.4 Literals

A literal is **an instance of a variable**, either in its true form or in its complemented form. For example, consider the Boolean function:

$$f(A, B)$$

This function contains two variables: A and B . If we define:

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

this expression contains two literals: A and B , instead, if we define:

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

this expression contains four literals: \overline{A} , B , \overline{B} , A .

It is important not to confuse variables with literals: each time a variable appears in an expression, it counts as a separate literal.

5.1.5 Products (Implicants) and Sums

The **AND** of one or more literals is called a **product** (or sometimes an **implicant**). Examples:

- AB ,
- ABC ,
- B .

All of these are considered products.

A **minterm** is a special kind of product, it **includes all the variables** of the function, each **appearing once** in either true or complemented form. For example, if we have three variables A , B , C :

- ABC ,
- $A\bar{B}C$,
- $\bar{A}BC$,

are all minterms. Each minterm corresponds to exactly **one row of the truth table**.

The **OR** of one or more literals is called a **sum**. For example:

- $A + B$,
- $A + \bar{C}$,
- $A + B + C$.

are all sums.

A **maxterm** is a special kind of sum. It is a sum that **involves all the variables** of the function, each **appearing once** in either true or complemented form. For example, with three variables A , B , C :

$$A + B + C$$

is a maxterm. Just like minterms, each maxterm corresponds to exactly one row of the truth table, but in a different way, a minterm is true for one unique input combination, while a maxterm is false for one unique input combination

5.2 Canonical Forms

To represent Boolean functions in a systematic way from the truth table, we often use canonical forms. These are standard representations that express the output as a combination of all the input variables.

5.2.1 Sum-of-Products (SOP)

A Boolean function of N inputs can always be described by a truth table with 2^N rows, one for each possible combination of input values. To connect the truth table with an algebraic expression, we can exploit the minterm concept.

Let us look at an example with two input variables. The truth table has four rows, and thus four minterms:

A	B	Y	minterm
0	0	0	$m_0 \rightarrow \bar{A} \bar{B}$
0	1	1	$m_1 \rightarrow \bar{A} B$
1	0	0	$m_2 \rightarrow A \bar{B}$
1	1	1	$m_3 \rightarrow A B$

sum of products form

$\bar{A} B + A \bar{B}$

To build a Boolean function from the truth table, we can take **the OR of all the minterms that correspond to rows where the output is equal to 1**. In the example, this means we take the minterms the second and fourth rows:

$$m_1 = \bar{A}B$$

$$m_3 = AB$$

so the function is:

$$Y = \bar{A}B + AB$$

This expression is said to be in **Sum-of-Products (SOP) form**, because it is written as a sum (OR) of products (ANDs). Listing all of them explicitly can become lengthy, especially if the function has many inputs. To make expressions more compact, we can use the **sigma notation**. In this notation, a Boolean function is written as a sum of selected minterms, indicated by their index numbers:

$$f(A, B) = \Sigma(m_1, m_3)$$

The SOP form is important because it provides **a systematic way to write down a Boolean function directly from its truth table**. The method works not only for small functions but also for truth tables with any number of variables. Suppose we have three inputs, the truth table will contain $2^3 = 8$ rows:

A	B	C	Y	minterm
0	0	0	1	$m_0 \rightarrow \bar{A} \bar{B} \bar{C}$
0	0	1	0	$m_1 \rightarrow \bar{A} \bar{B} C$
0	1	0	0	$m_2 \rightarrow \bar{A} B \bar{C}$
0	1	1	0	$m_3 \rightarrow \bar{A} B C$
1	0	0	1	$m_4 \rightarrow A \bar{B} \bar{C}$
1	0	1	1	$m_5 \rightarrow A \bar{B} C$
1	1	0	0	$m_6 \rightarrow A B \bar{C}$
1	1	1	0	$m_7 \rightarrow A B C$

Diagram illustrating the conversion of a truth table to SOP form. The minterms are grouped by rows (rows 0, 1, 2, 3) and columns (columns 0, 1, 2). The output is $\bar{A} \bar{B} \bar{C} + A \bar{B} \bar{C} + A \bar{B} C$.

Therefore, the Boolean function is:

$$Y = \bar{A} \bar{B} \bar{C} + A \bar{B} \bar{C} + A \bar{B} C$$

In sigma notation, we can write this compactly as:

$$Y = \Sigma(m_0, m_4, m_5)$$

Although the SOP form is systematic and always works, **it does not necessarily yield the simplest equation**. In the example above, the function uses three minterms, each of which contains three literals. With Boolean algebra simplification techniques, or with Karnaugh maps, we will later learn how to reduce this expression to a shorter and more efficient form.

5.2.2 Product-of-Sums (POS)

The Product-of-Sums (POS) form is an alternative way of expressing Boolean functions. Instead of using minterms (rows where the function is true), POS relies on **maxterms**, which are associated with rows where the function is false:

A	B	Y	maxterm
0	0	0	$M_0 \rightarrow A+B$
0	1	1	$M_1 \rightarrow A+B$
1	0	0	$M_2 \rightarrow \bar{A}+B$
1	1	1	$M_3 \rightarrow \bar{A}+B$

Diagram illustrating the conversion of a truth table to POS form. The maxterms are grouped by rows (rows 0, 1, 2, 3) and columns (columns 0, 1, 2). The output is $(A+B)(\bar{A}+B)$.

A Boolean function can be written as the AND of all the maxterms corresponding to the rows where the output is false. In the example, it means we take the maxterms for the first and third rows:

$$Y = (A+B)(\bar{A}+B)$$

This is the **Product-of-Sums (POS) form**, since it is a product (AND) of sums (ORs). In pi notation, we can write:

$$f(A, B) = \Pi(M_0, M_2)$$

Just like SOP, the POS form **does not necessarily yield the simplest possible expression**. However, the two forms are complementary:

- SOP is usually more convenient when the function is true for only a few rows.
- POS is often more convenient when the function is false for only a few rows.

This duality between SOP and POS is fundamental and will be useful when we learn simplification techniques.

5.3 Axioms and Theorems

Up to now, we have seen how any Boolean function can be written in a canonical form directly from its truth table. These methods are systematic and **guarantee a correct representation** of the function. However, the resulting **expressions are not always the most efficient**. A function written in canonical form may use many literals and therefore **require a large number of logic gates** when implemented.

5.3.1 Axioms

Boolean algebra is based on a small set of **axioms**, which we assume to be true. From these axioms, we can derive all other simplification rules. The table below lists five fundamental axioms that define the meaning of Boolean variables and the basic operations of NOT, AND, and OR:

Axiom	Statement	Dual Statement	Name
A1	$B = 0$ if $B \neq 1$	$B = 1$ if $B \neq 0$	Binary field
A2	$\bar{0} = 1$	$\bar{1} = 0$	NOT
A3	$0 \cdot 0 = 0$	$1 + 1 = 1$	AND/OR
A4	$1 \cdot 1 = 1$	$0 + 0 = 0$	AND/OR
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$	AND/OR

A key property of Boolean algebra is the **principle of duality**. If in any valid equation we **systematically replace 0 with 1, 1 with 0, AND with OR, and OR with AND**, the new equation will also be **valid**. This principle allows us to generate pairs of related theorems, one being the dual of the other.

These axioms and the principle of duality form the foundation upon which all Boolean simplification techniques are built.

5.3.2 Theorems of one variable

The axioms form the foundation, but to make simplification practical we use a **set of theorems** derived from them. These theorems show how expressions can be reduced in a systematic way, saving gates and making circuits more efficient. Let us begin with the theorems involving a single variable. The following table summarizes five important theorems, together with their dual forms:

Theorem	Statement	Dual Statement	Name
T1	$B \cdot 1 = B$	$B + 0 = B$	Identity
T2	$B \cdot 0 = 0$	$B + 1 = 1$	Null element
T3	$B \cdot B = B$	$B + B = B$	Idempotency
T4	$\overline{\overline{B}} = B$	—	Involution
T5	$B \cdot \overline{B} = 0$	$B + \overline{B} = 1$	Complements

T1 - Identity Theorem: A variable ANDed with 1 is unchanged, and a variable ORed with 0 is also unchanged. In hardware terms, this means that if one input of an AND gate is permanently set to 1, the AND gate can be removed, and the output simply equals the other input:



And if one input of an OR gate is permanently set to 0, the OR gate can also be removed:



This is not only a theoretical simplification but also a **practical one: every unnecessary gate adds cost, power consumption, and delay to a circuit**. Replacing redundant gates with a direct wire connection is always beneficial.

T2 - Null element: A variable ANDed with 0 is always 0, and a variable ORed with 1 is always 1. This means that if one input of an AND gate is fixed to 0, the output will always be 0 regardless of the other input. In circuit terms, the gate is useless and can be replaced by a constant logic 0:



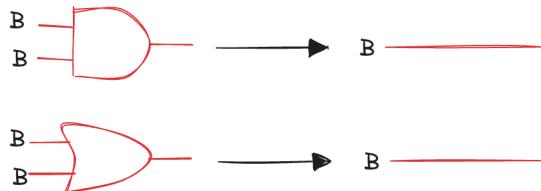
If one input of an OR gate is fixed to 1, the output will always be 1, no matter what the other input is. The OR gate can therefore be replaced by a constant logic 1:



These results are important for simplification because they allow us to eliminate redundant gates when one input is permanently tied to 0 or 1.

T3 - Idempotency: Repeating a variable does not change its value.

The word "idempotent" comes from Latin roots meaning "the same power". In Boolean algebra, it refers to the fact that repeating the same variable in an operation has no additional effect. From a circuit perspective, this means that if both inputs of a gate are connected to the same signal, the gate is unnecessary: the output is identical to the input. The gate can be removed and replaced with a direct wire connection:



This theorem reinforces the idea that gates should never be used to duplicate an input; redundancy does not change the logic and only wastes hardware resources.

T4 - Involution: The complement of the complement returns the original variable.

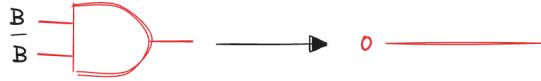
This property is intuitive: the first negation flips the value, and the second negation flips it back. From a circuit perspective, two inverters connected in series cancel each other out and are logically equivalent to a direct wire connection:



T5 - Complements: A variable ANDed with its complement is always 0, while a variable ORed with its complement is always 1.

The complement theorem expresses the fundamental relationship between a variable and its logical opposite. It is true because one of the two inputs must be 0 (for AND) or 1 (for OR). At the hardware level, they show that combining a signal with its inverted form leads to trivial

constants (always 0 or always 1). Such expressions can be replaced by fixed values, reducing unnecessary gates:



5.3.3 Theorems of several variables

While the theorems of one variable provide useful simplifications, most practical Boolean expressions involve several variables. In this case, additional theorems allow us to reorganize and reduce expressions, often leading to much simpler forms. The table below summarizes some of the most important multi-variable theorems, together with their duals:

Theorem	Statement	Dual Statement	Name
T6	$B \cdot C = C \cdot B$	$B + C = C + B$	Commutativity
T7	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$	$(B + C) \cdot (B + D) = B + (C \cdot D)$	Distributivity
T9	$B + (B \cdot C) = B$	$B \cdot (B + C) = B$	Covering
T10	$(B \cdot C) + (B \cdot \bar{C}) = B$	$(B + C) \cdot (B + \bar{C}) = B$	Combining
T11	$(B \cdot C) + (\bar{B} \cdot D) + (C \cdot D) = (B \cdot C) + (\bar{B} \cdot D)$	$(B + C) \cdot (\bar{B} + D) \cdot (C + D) = (B + C) \cdot (\bar{B} + D)$	Consensus
T12	$A \cdot B + \bar{A} = B + \bar{A}$	$(A + B) \cdot \bar{A} = B \cdot \bar{A}$	Absorption
T13	$B_0 \cdot B_1 \cdot B_2 \dots = (\bar{B}_0 + \bar{B}_1 + \bar{B}_2 \dots)$	$B_0 + B_1 + B_2 \dots = (\bar{B}_0 \cdot \bar{B}_1 \cdot \bar{B}_2 \dots)$	De Morgan's Theorem

Together, these theorems give us a powerful toolkit for manipulating Boolean functions into simpler and more efficient forms.

T6 — The Commutativity Theorem

This theorem shows that the **order of the inputs does not matter**, either for AND or for OR.

$$B \cdot C = C \cdot B$$

$$B + C = C + B$$

This property lets us **rearrange terms** without changing the logic of the expression, which is often a first step in aligning terms for further simplification.

T7 — The Associativity Theorem

This theorem shows that the **grouping of inputs does not matter**, either for AND or for OR. Parentheses can be rearranged freely:

$$(B \cdot C) \cdot D = B \cdot (C \cdot D)$$

$$(B + C) + D = B + (C + D)$$

Associativity lets us simplify expressions like $A+B+C$ **without worrying about how the terms are grouped**. This makes Boolean expressions more flexible to handle.

T8 — The Distributivity Theorem

This theorem shows how AND and OR interact: **AND distributes over OR**, and **OR distributes over AND**:

$$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$$

$$(B + C) \cdot (B + D) = B + (C \cdot D)$$

Notice that this property is **different from traditional algebra**, because in ordinary arithmetic addition does not distribute over multiplication. In Boolean algebra, however, the restricted values (0 and 1) make this distribution possible. This theorem is particularly important because it allows us to **factor expressions** (like algebraic factoring) or to **expand them**. Both directions are widely used in simplification.

T9 — The Covering Theorem

This theorem states that when a variable already appears in an expression, any additional term where the same variable is ANDed with another factor is redundant and can be eliminated:

$$B + (B \cdot C) = B$$

$$B \cdot (B + C) = B$$

This result is important since it allows us to **recognize and remove redundant terms**. We can prove this theorem using some of the previous theorems:

$$\$B \cdotdot (B + C) = \$ \text{ (T8 Distributivity)}$$

$$\$B \cdotdot B + B \cdotdot C = \$ \text{ (T3 Idempotency)}$$

$$\$B + B \cdotdot C = \$ \text{ (T8 Distributivity)}$$

$$\$B \cdotdot (1 + C) = \$ \text{ (T2 Null element)}$$

$$B \cdot 1 = B \quad (\text{T1 Identity})$$

B

T10 — The Combining Theorem

This theorem states that when two terms differ only by a variable and its complement, the expression can be simplified to depend solely on the common factor:

$$(B \cdot C) + (B \cdot \bar{C}) = B$$

$$(B + C) \cdot (B + \bar{C}) = B$$

This result is important since it allows us to eliminate variables that do not affect the outcome. We can prove it:

$$(B \cdot C) + (B \cdot \bar{C}) = (T8 \text{ Distributivity})$$

$$B \cdot (C + \bar{C}) = (T5 \text{ Complements})$$

$$B \cdot 1 = (T1 \text{ Identity})$$

B

T11 — The Consensus Theorem

The theorem states that when two terms already cover all possible cases of a third term, that third term is redundant and can be removed without affecting the result:

$$(B \cdot C) + (\bar{B} \cdot D) + (C \cdot D) = (B \cdot C) + (\bar{B} \cdot D)$$

$$(B + C) \cdot (\bar{B} + D) \cdot (C + D) = (B + C) \cdot (\bar{B} + D)$$

Eliminating the redundant term reduces the number of gates without changing the function. We can prove it:

$$(B \cdot C) + (\bar{B} \cdot D) + (C \cdot D) = (T5 \text{ Complements})$$

$$(B \cdot C) + (\bar{B} \cdot D) + (C \cdot D \cdot (B + \bar{B})) = (T8 \text{ Distributivity})$$

$$(B \cdot C) + (\bar{B} \cdot D) + (B \cdot C \cdot D) + (\bar{B} \cdot C \cdot D) = (T6 \text{ Commutativity})$$

$$(B \cdot C) + (B \cdot C \cdot D) + (\bar{B} \cdot D) + (\bar{B} \cdot C \cdot D) = (T8 \text{ Distributivity})$$

$$B \cdot C \cdot (1 + D) + \bar{B} \cdot D \cdot (1 + C) = (T2 \text{ Null element})$$

$$B \cdot C \cdot 1 + \bar{B} \cdot D \cdot 1 = (T1 \text{ Identity})$$

$$(B \cdot C) + (\bar{B} \cdot D)$$

T13 — The Absorption Theorem

This theorem states that when a variable is combined with a term that already includes its complement, the redundant factor can be absorbed, leaving a simpler equivalent expression:

$$A \cdot B + \bar{A} = B + \bar{A}$$

$$(A + B) \cdot \bar{A} = B \cdot \bar{A}$$

We can prove it:

$$A \cdot \overline{B} + \overline{A} \cdot B = (T5 \text{ Complements})$$

$$A \cdot B + \overline{A} \cdot (B + \overline{B}) = (T8 \text{ Distributivity})$$

$$A \cdot B + \overline{A} \cdot B + \overline{A} \cdot \overline{B} = (T8 \text{ Distributivity})$$

$$B \cdot (A + \overline{A}) + \overline{A} \cdot \overline{B} = (T5 \text{ Complements})$$

$$B \cdot 1 + \overline{A} \cdot \overline{B} = (T1 \text{ Identity})$$

$$B + \overline{A} \cdot \overline{B} = (T8 \text{ Distributivity})$$

$$(B + \overline{A}) \cdot (B + \overline{B}) = (T5 \text{ Complements})$$

$$(B + \overline{A}) \cdot 1 = (T1 \text{ Identity})$$

$$B + \overline{A}$$

T13 — De Morgan's Theorems

De Morgan's theorems, named after the British mathematician Augustus De Morgan (1806–1871), who contributed significantly to logic and algebra, describe how the complement (negation) of a product or a sum can be expressed in terms of the complements of the individual terms.



Augustus De Morgan, 1806 – 1871

A British mathematician, born in India. Blind in one eye. His father died when he was 10. Attended Trinity College, Cambridge, at age 16, and was appointed Professor of Mathematics at the newly founded London University at age 22. Wrote widely on many mathematical subjects, including logic, algebra, and paradoxes. De Morgan's crater on the moon is named for him. He proposed a riddle for the year of his birth: "I was x years of age in the year x^2 ."

There are two fundamental forms: the complement of a product is equal to the sum of the complements:

$$\overline{B \cdot C \cdot D} = \overline{B} + \overline{C} + \overline{D}$$

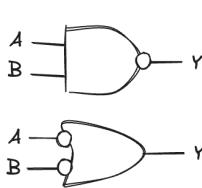
The complement of a sum is equal to the product of the complements:

$$\overline{B + C + D} = \overline{B} \cdot \overline{C} \cdot \overline{D}$$

In words: **the negation of an AND operation is equivalent to an OR of the negated inputs, and the negation of an OR operation is equivalent to an AND of the negated inputs.**

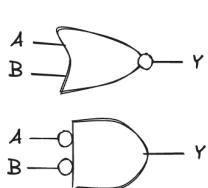
De Morgan's theorems are particularly important because they explain the **relationship between NAND/NOR gates and the basic AND/OR operations**. A NAND gate can be seen as an OR gate with inverted inputs, and a NOR gate can be seen as an AND gate with inverted inputs:

$$y = \overline{A}\overline{B} = \overline{A} + \overline{B}$$



A	B	y
0	0	1
0	1	1
1	0	1
1	1	0

$$y = \overline{A} + \overline{B} = \overline{A}\overline{B}$$



A	B	y
0	0	1
0	1	0
1	0	0
1	1	0

This equivalence allows NAND and NOR gates to be used as **universal gates**, since any digital circuit can be built using only NAND or only NOR gates.

The inversion circle drawn at the input or output of a gate is called a **bubble**. De Morgan's theorem explains how "pushing" a bubble through a gate changes its type: an AND gate becomes an OR gate when bubbles are moved to its inputs, and an OR gate becomes an AND gate when bubbles are moved to its inputs. This "bubble pushing" technique is widely used in circuit simplification and in schematic transformations, because it allows the designer to switch between AND/OR implementations using NAND and NOR gates.

5.3.4 Perfect induction

When working with Boolean algebra, many theorems can be proven algebraically using the axioms and previously established theorems. However, since Boolean variables can only take the values 0 or 1, there is another straightforward method available: perfect induction.

Perfect induction consists of **checking that a theorem holds for all possible input combinations of the variables involved**. If the two sides of the equation produce identical results for every case, then the theorem is proven. This method is direct, mechanical, and especially useful for verifying the correctness of Boolean laws. For example, we can use perfect induction to prove the consensus theorem. We construct a truth table for both sides of the equation:

B	C	D	$Bc + \bar{B}D + CD$	$Bc + \bar{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Since the two columns are identical, the equality holds for all possible values of the inputs. Therefore, the theorem is proved.

Perfect Induction is **simple to understand**, since it only requires evaluating truth tables, and it is **guaranteed to work**, because Boolean variables are **finite** and **discrete**. However, it **becomes impractical** for expressions with many variables, because the number of rows grows as 2^n . In such cases, algebraic simplification is preferred.

5.4 Simplifying Expressions

The theorems of Boolean algebra are not only useful for proving properties, but also provide a **systematic way to simplify Boolean expressions**. Simplification is at the heart of digital design. It transforms truth-table derived SOP or POS forms, which are often large and unwieldy, into compact implementations that use the fewest possible gates.

Whenever we apply Boolean theorems to reduce an expression, the simplified result remains logically equivalent to the original. One may then ask: why bother with simplification if the logical behavior does not change? The main reason is **efficiency**. A simplified expression generally requires **fewer logic gates to implement**. Since each gate consumes physical resources in a circuit, fewer gates mean the design is **smaller, cheaper, and faster**, while also **consuming less power**. Simplification is therefore not just a theoretical exercise, but a fundamental step in optimizing digital hardware. Considering the following example:

$$Y = \overline{A}B + AB$$

At first glance, this may look like it requires two product terms. However, by applying the Combining Theorem (T10), we can simplify it directly to:

$$Y = B$$

The example shows how an apparently more complex expression can collapse into a very simple form when the right theorems are applied. In practice, more complicated functions may require several steps of simplification, but the principle remains the same.

The essential strategy for simplifying an expression in SOP form is to repeatedly look for pairs of terms that differ in only one literal and use the combining rule to eliminate redundancy:

$$PA + \overline{P}A = A$$

where P may represent any Boolean expression. This identity shows that if A appears in both terms, the value of P or its complement becomes irrelevant, and the function depends only on A .

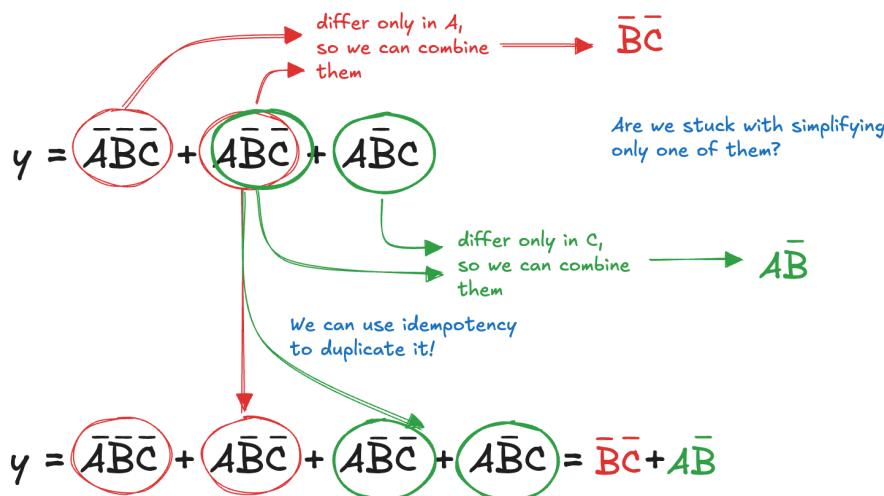
How far can we go with simplification? An expression written in sum-of-products form is considered minimized if it uses the **fewest possible product terms**. When several different expressions use the same number of products, the minimal one is the one containing the **fewest**

total literals. This criterion ensures that the minimized form corresponds to the most efficient implementation of the function.

In this context, a product is called a **prime product** if it cannot be combined with any other product in the expression to form a new product with fewer literals. Every product in a minimal equation must be a prime product. If a product is not prime, it can be merged with another term, which means the expression is not yet fully simplified.

Let us minimize the following expression:

$$f = \bar{A}\bar{B}C + A\bar{B}C + A\bar{B}\bar{C}$$



Simplifying Boolean expressions using only theorems can **sometimes be a matter of trial and error**. Different combinations of theorems may need to be tested before the minimal form emerges, and the process can become complicated as the number of variables grows. For this reason, in addition to algebraic manipulation, designers also rely on a graphical method known as the **Karnaugh map**, that provides a structured, visual way to identify simplifications, making the process faster and more reliable than relying solely on theorem application.

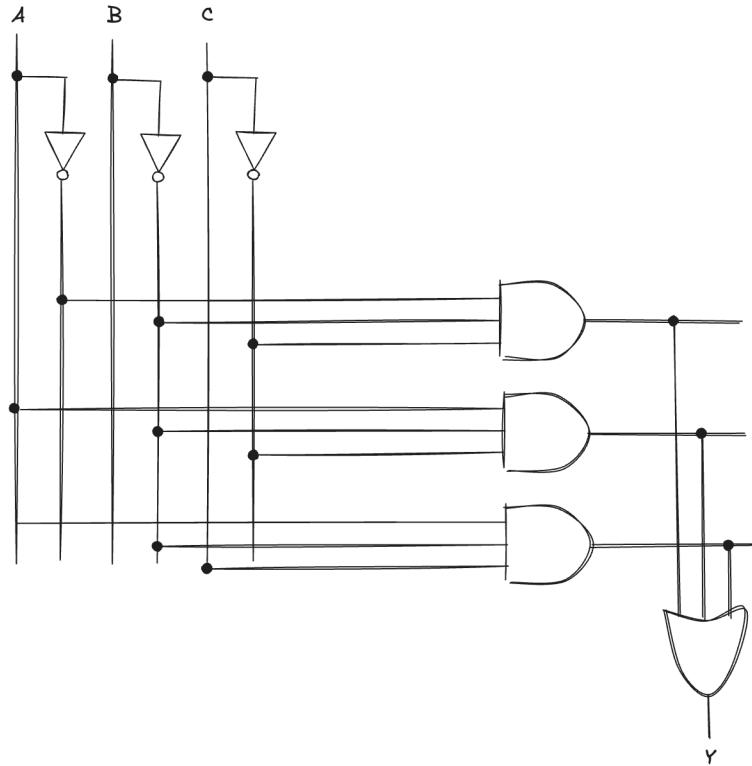
5.5 Schematic

A Boolean expression can be directly implemented as a digital circuit, and the resulting diagram is called a **schematic**. It shows the logic gates that realize the function and the wires that connect them. Consider the function of the previous example:

$$Y = \bar{A}\bar{B}C + A\bar{B}C + A\bar{B}\bar{C}$$

If we implement this function in its canonical sum-of-products form, the schematic requires three AND gates to generate the minterms and one OR gate to combine them, plus inverters for

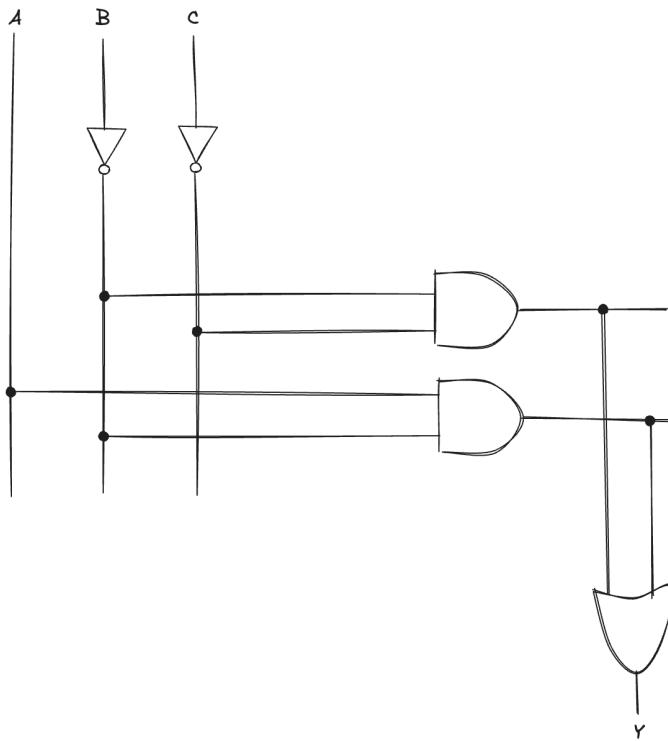
the complemented inputs. The circuit is correct, but it uses many gates and several inputs per gate:



After simplification, the same function reduces to

$$Y = \overline{B}\overline{C} + A\overline{B}$$

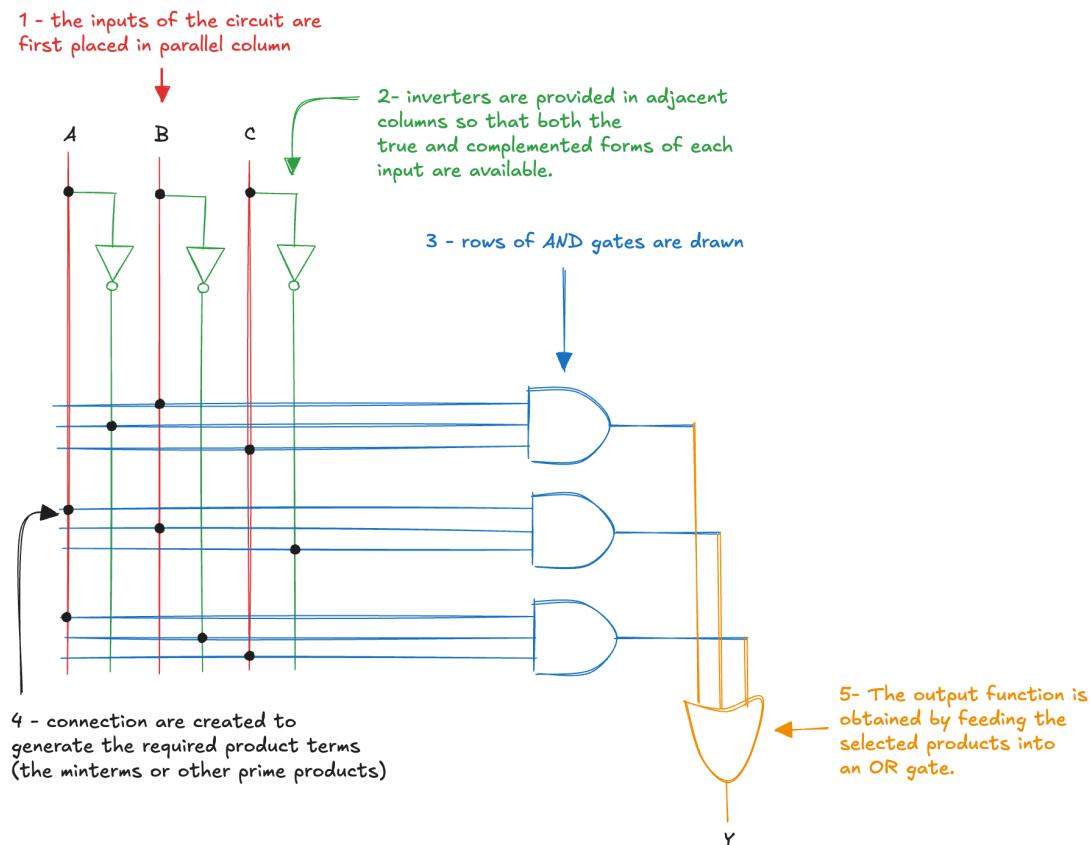
The corresponding schematic is much smaller: only two AND gates and one OR gate are required, along with a single inverter:



This comparison illustrates the central motivation for simplification. Although both circuits implement exactly the same logic, the simplified version requires** significantly less hardware. **Fewer gates mean** a smaller circuit, **which is generally cheaper**** and more **energy efficient**, and may also be **faster** because each signal passes through fewer gate inputs and fewer logic levels. This demonstrates how mathematical manipulation directly influences the cost, size, and performance of real circuits.

5.5.1 Programmable Logic Arrays (PLAs)

Notice from the previous example, that a Boolean equation written in sum-of-products form can be implemented in hardware following a **systematic procedure**:



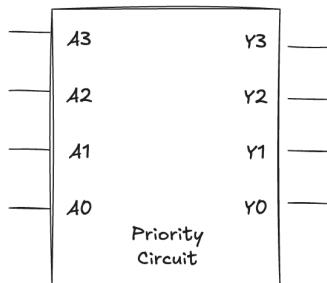
This regular structure of inputs, inverters, AND gates, and OR gates is called a **Programmable Logic Array (PLA)**. The name comes from the fact that the array of gates is arranged in a standard and systematic way, while **the actual logic function implemented depends only on how the connections ("programming") are made between inputs, products, and outputs**.

The advantage of the PLA style is that it provides a **universal template** for implementing any Boolean function. Instead of designing a new schematic from scratch for each function, we can always rely on the same architecture: a layer of input signals with inverters, a layer of AND gates forming products, and a layer of OR gates producing the outputs. The programming step determines which minterms are used and how they are combined.

5.5.2 Multiple-output circuits

In many practical situations, digital circuits are required to generate **more than one output**. Each output is a separate Boolean function of the same set of inputs. To describe such circuits, we can either write an independent truth table for each output, or, more efficiently, place all the outputs in a single truth table that lists every input combination along with the corresponding values of all outputs. For example, we can consider a **priority circuit**, in which multiple inputs

may be active at the same time, but only the highest-priority active input is recognized and produces the corresponding output:

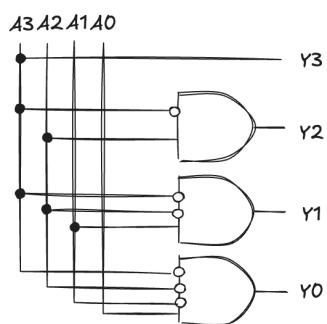


We can derive the Boolean equations for each output by constructing the canonical sum-of-products form and then simplifying with Boolean algebra. However, in practice, experienced designers often take advantage of observations and implement circuits *directly from inspection*, bypassing lengthy algebraic simplifications. In this example, the outputs are true if the corresponding input is true and all higher-priority inputs are false. This leads directly to the following simplified truth table:

high priority				low priority				multi-output			
A3	A2	A1	A0	Y3	Y2	Y1	Y0				
0	0	0	0	0	0	0	0				
0	0	0	1	0	0	0	1				
0	0	1	x	0	0	1	0				
0	1	x	x	0	1	0	0				
1	x	x	x	1	0	0	0				

↑
don't care

The X symbol indicates "**don't care**" conditions, meaning that the output can be either 0 or 1 without affecting the circuit's correctness. This flexibility allows for further simplification of the Boolean expressions. The resulting circuit is:



The schematic for the priority circuit shows how each output is generated with a small number of gates, reflecting the priority structure encoded in the truth table.

5.5.3 Multi-level circuits

Logic functions expressed in sum-of-products form are known as **two-level logic**, since they consist of literals connected to a layer of AND gates, whose outputs are then fed into a single OR gate. This representation is systematic and always correct, but for many functions it may require **an enormous amount of hardware**, as every minterm must be explicitly generated and summed.

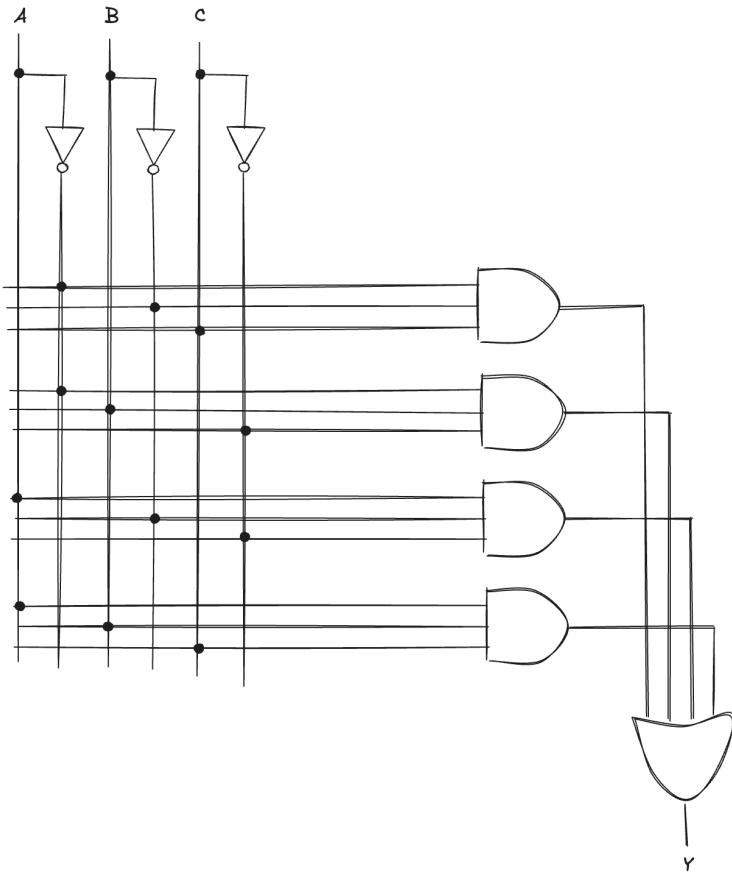
A useful example is the XOR function with three inputs. By definition, an XOR produces a 1 whenever an odd number of inputs are true. Looking at the truth table of:

$$y = A \oplus B \oplus C$$

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The corresponding sum-of-products expression is:

$$Y = \overline{ABC} + \overline{AB}\overline{C} + \overline{A}\overline{BC} + ABC$$



This implementation is correct but requires several AND gates and a large OR gate. A more compact solution emerges if we rewrite the function as

$$A \oplus B \oplus C = (A \oplus B) \oplus C$$

In this form, the circuit can be built simply by cascading two XOR gates, which is far more efficient than the canonical two-level approach.

This example illustrates the essence of **multilevel combinational logic**: by reorganizing Boolean expressions into multiple levels, we can obtain simpler circuits, with fewer gates, reduced hardware cost compared to their two-level counterparts.

However, choosing the **best** multilevel realization of a logic function is not straightforward. The notion of "best" depends on many different criteria: it might mean the circuit with the fewest gates, the fastest response, the shortest design time, the lowest cost, or the least power consumption. Importantly, the best design in one technology may not be the best in another. For example, CMOS circuits generally favor NAND and NOR gates rather than AND and OR gates, because they are simpler and more efficient to implement.

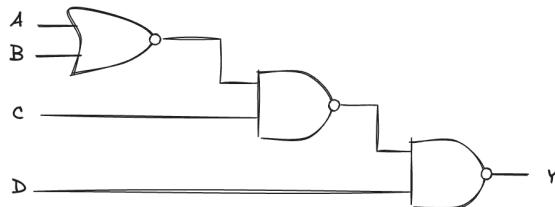
With growing experience, many circuits can be designed efficiently by inspection, as patterns

and simplifications become familiar. Nevertheless, for complex logic functions, the number of possible multilevel implementations grows rapidly, and **Computer-Aided Design (CAD)** tools are essential. These tools can explore a vast space of alternatives and automatically search for the design that satisfies given constraints, using the available building blocks.

5.5.4 Bubble pushing

When working with implementations based on NAND and NOR gates, it is often useful to redraw a circuit in a form where the underlying function can be determined more easily. A systematic method for this simplification is known as **bubble pushing**. The idea is to manipulate the "bubbles" (inversion markers) on the inputs and outputs of gates so that they cancel each other whenever possible, thereby revealing a clearer and more compact representation of the circuit.

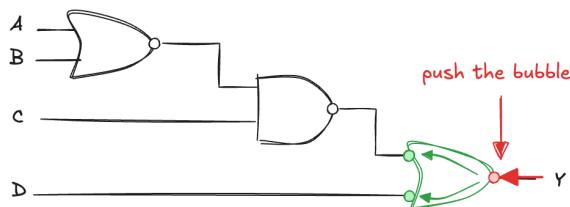
The procedure begins by examining the output of the circuit and then working backward toward the inputs. At each step, **inversion bubbles are pushed back**, using the property that a bubble at the input of one gate can be transferred to the output of the previous gate, provided the corresponding inversion is preserved. In practice, this means we can redraw each gate in such a way that input and output bubbles cancel when they appear in series. For example, consider the following circuit:

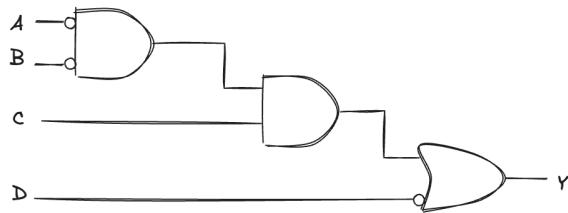
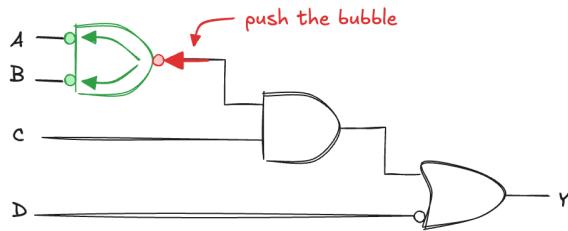
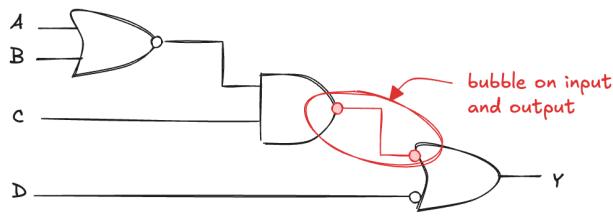


Its direct expression is:

$$Y = \overline{(\overline{A + B})C}D$$

which is correct but not very transparent. By pushing the bubbles back through the gates, we can redraw the circuit step by step:





Finally, the circuit expression becomes:

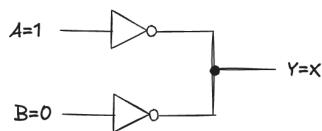
$$Y = \overline{A} \overline{B} C + \overline{D}$$

This technique allows us to read expressions directly in terms of the original signals rather than their complements, and it greatly simplifies circuits that contain large numbers of NAND and NOR gates. The final result is a representation that is both easier to analyze.

5.5.5 Beyond 0 and 1: X and Z

Boolean algebra strictly assumes only two logic levels, 0 and 1. Real circuits, however, may also exhibit **illegal or floating values**, usually denoted as X and Z. These values reflect physical situations that cannot be represented by ideal Boolean logic but occur frequently in hardware.

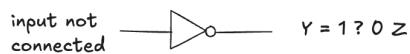
The symbol X indicates that a circuit node has an unknown or illegal value. This typically happens when the node is simultaneously driven to 0 and to 1, a situation known as **contention**:



In such a case the actual voltage lies somewhere between ground and power supply, depending on the relative strength of the driving sources. This is an **error condition** that must be avoided

in real systems. Circuit simulators also use X to represent an **uninitialized node**. It is important to remember that in truth tables X can also mean "don't care", but this is a different meaning that should not be confused with illegal values in circuits.

The symbol Z indicates that a node is being driven by neither a logic HIGH nor a logic LOW; in other words, the node is floating.

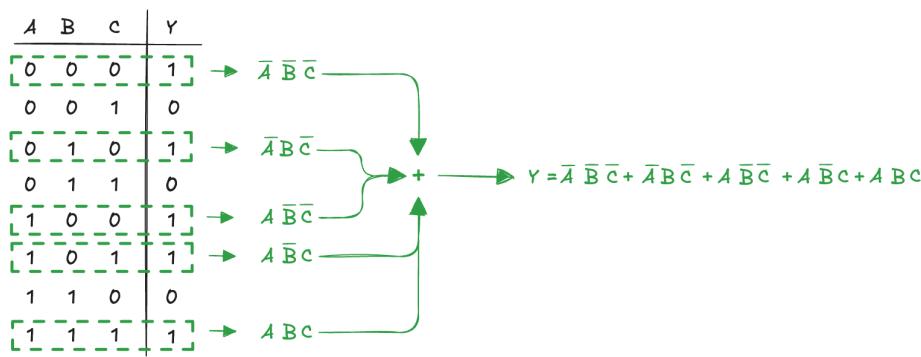


A floating node is **unpredictable**: it might be read as 0, as 1, or as an intermediate voltage. A common mistake that produces a floating node is to **leave an input pin unconnected**. In this case the circuit can **behave erratically**, switching randomly between values or even reacting to static charges from a person's touch. This explains why, in a lab environment, some circuits appear to work correctly only while a student's finger is resting on a chip, the touch slightly biases the floating input.

5.6 Exercises

1 - Write a Boolean equation in sum-of-products canonical form for the following truth tables of 3 variables, then simplify the equation using Boolean algebra rules

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



$$Y = \bar{A} \bar{B} \bar{C} + \bar{A} B \bar{C} + A \bar{B} \bar{C} + A \bar{B} C + A \bar{B} C + A B C$$

Idempotency

combining $\bar{A} \bar{C}$ combining $A \bar{B}$ combining $A C$

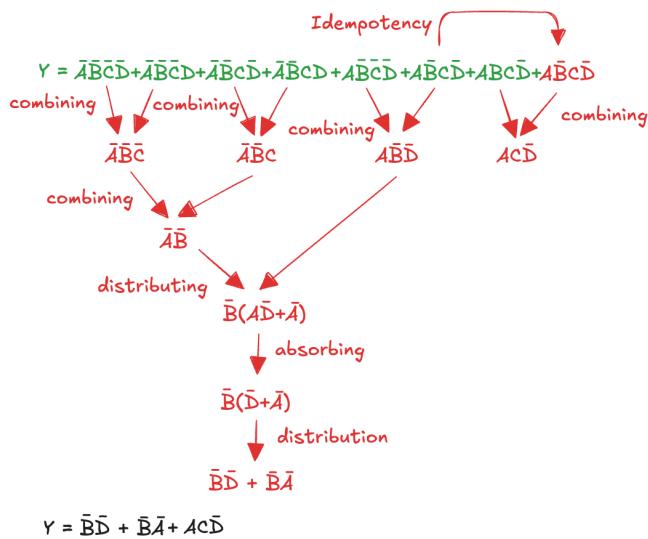
$$Y = \bar{A} \bar{C} + A \bar{B} + A C$$

2 - Write a Boolean equation in sum-of-products canonical form for the following truth tables of 4 variables, then simplify the equation using Boolean algebra rules

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	1	1	0	0
1	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

→ $\bar{A} \bar{B} \bar{C} \bar{D} + \bar{A} \bar{B} \bar{C} D + \bar{A} \bar{B} C \bar{D} + \bar{A} \bar{B} C D + A \bar{B} \bar{C} \bar{D} + A \bar{B} \bar{C} D + A \bar{B} C \bar{D}$



6 Graphical minimization methods

Describing a function through its Boolean expression or truth table makes simplifying them logically an arduous task. **Karnaugh maps**, often abbreviated as **K-maps**, are a **graphical method** for simplifying Boolean equations. They provide a structured way to represent truth tables in a format that makes simplification intuitive. By arranging and grouping minterms according to their geometric relationships, the map highlights logical adjacencies and makes it easy to apply properties such as absorption and combination directly.

- Absorption: $A + AB = A$
- Combination: $AB + A\bar{B} = A$



Maurice Karnaugh, 1924–2022

Graduated with a bachelor's degree in physics from the City College of New York in 1948 and earned a Ph.D. in physics from Yale in 1952. Worked at Bell Labs and IBM from 1952 to 1993 and as a computer science professor at the Polytechnic University of New York from 1980 to 1999.

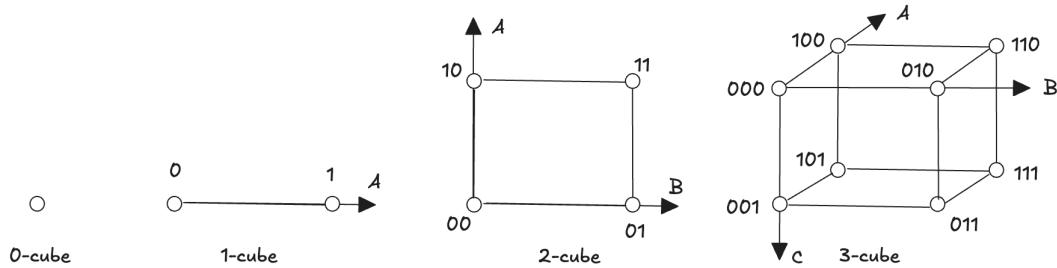
To understand the idea, we need to introduce the concept of logic geometry, which provides a geometric interpretation of binary numbers and their relationships.

6.1 Geometry of Logic

A binary number with n bits can be represented as a **point in an n -dimensional space**. To understand this idea, consider first the set of binary numbers with just one bit (0 and 1). These two possible values can be represented as two points along a **one-dimensional line**. This representation is known as a **1-cube** (or n -cube of order 1). If we had only one possible value (for instance, 0 alone), it would correspond to a **0-cube**, which is simply a single point in a degenerate zero-dimensional space.

If we now consider the set of binary numbers with **two bits** ($\{00, 01, 10, 11\}$), we can represent them as **four points** (the vertices of a **square**) in a two-dimensional space. This figure is called a **2-cube** (or n -cube of order 2). It can be obtained by taking the 1-cube, projecting it into a second dimension, and adding a 0 or 1 as a suffix to each of the original points.

By applying a similar projection method, it is possible to obtain a cube of higher dimension. For example, the representation of all 3-bit binary numbers can be derived by projecting the 2-cube and adding a prefix 0 to the points of the original 2-cube, and a prefix 1 to the points of the projected one. The resulting figure is a **3-cube** (or n -cube of order 3):



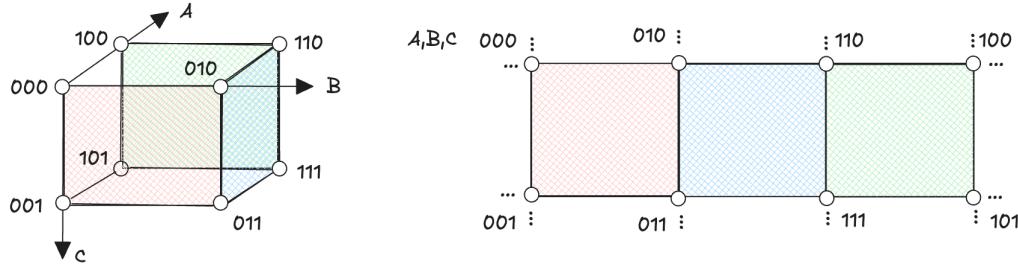
6.1.1 Logical adjacency

When moving along one edge of the cube (from one vertex to its geometrically adjacent vertex) only **one variable changes value**. This reflects a condition of **logical adjacency**, which occurs between two combinations of variables that differ in the value of a single variable; in other words, they are at **distance 1**.

6.1.2 Unfolding

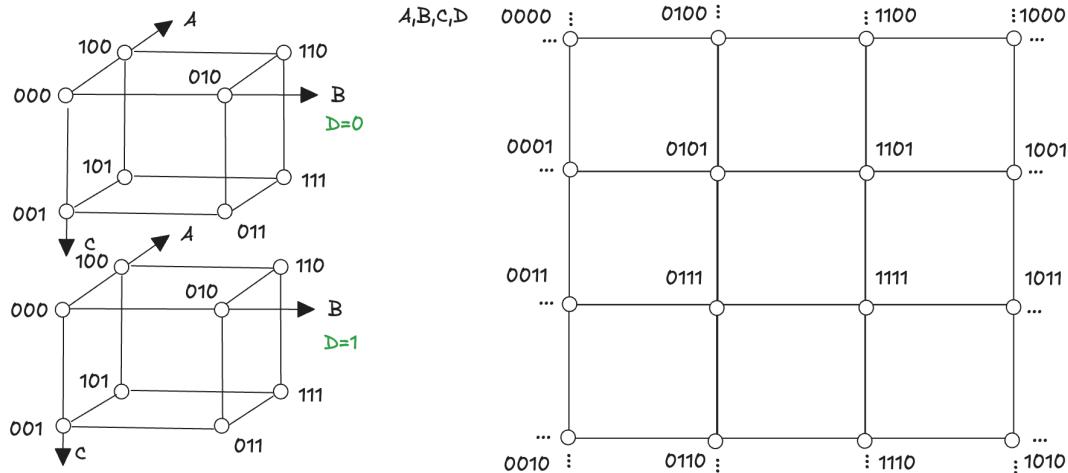
To represent the three-dimensional cube in two dimensions, we can **unfold** it onto a plane. The goal of this operation is to arrange the cube's eight vertices so that their **spatial adjacency is preserved** as much as possible in the planar layout. Each vertex represents one combination

of the three binary variables, and adjacency between vertices corresponds to combinations that differ in only one variable:



To maintain this adjacency, the variables must be ordered according to the **Gray code sequence** rather than the ordinary binary order. In Gray code, successive combinations differ by only a single bit—exactly the property we need to preserve logical adjacency. For example, the order of A and B in the horizontal direction we follows 00, 01, 11, 10, instead of the usual 00, 01, 10, 11. When the cube is unfolded, each pair of adjacent squares in the map corresponds to faces of the cube that share a common edge.

The same approach can be generalized to higher dimensions. Even though we cannot directly visualize a 4-cube, we can represent its structure through unfolded diagrams, where adjacency between vertices is preserved according to the same principle as before. The Gray code ordering ensures that neighboring points in the representation still differ in the value of just one variable:

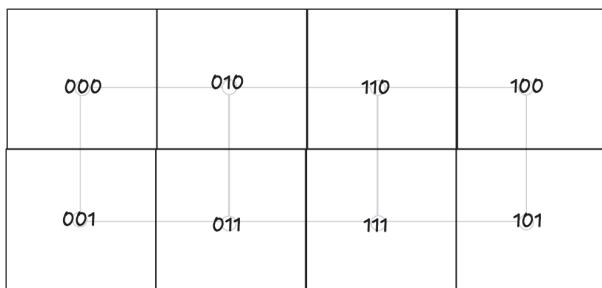


As the number of variables increases, the dimensionality of the cube and thus the number of vertices grows exponentially. However, the geometric relationships and adjacency rules remain consistent: each vertex is connected to n others, corresponding to the n variables that can change value independently. This geometric framework provides a powerful way to reason about log-

ical relationships and adjacency, even in higher dimensions where visualization is no longer possible.

6.2 Karnaugh maps

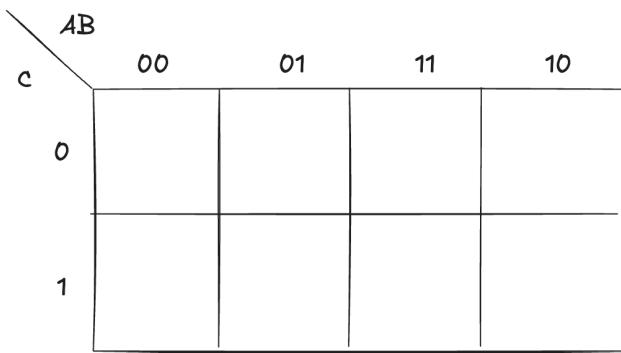
The Karnaugh map is derived from the cube and maintains the same topological relationships as the original three-dimensional figure. Vertices that were adjacent in the cube remain adjacent in the map:



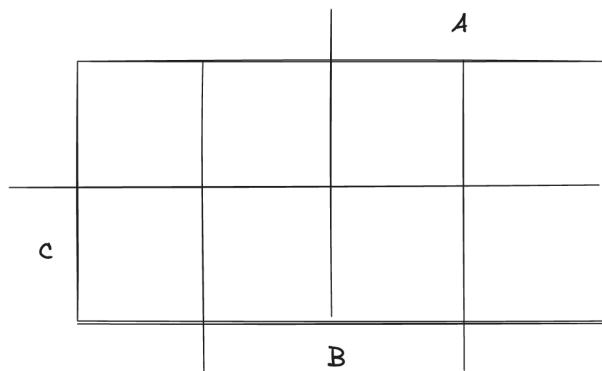
We can observe the following:

- In the **four lower cells**, C=1
- In the **four upper cells**, C=0
- In the **four cells on the right**, A=1
- In the **four cells on the left**, A=0
- In the **four central cells**, B=1
- In the **four outer cells**, B=0

This shows that the **regions of the map** can be represented using the variables as the "coordinates" of the cells. One approach, labels the rows and columns directly with the values of the variables:



A second, more geometric method, divides the map into areas corresponding to the regions where each variable equals 1:



Similar reasoning can be applied to the case of four variables, leading to the construction of a **4-variable Karnaugh map**:

0000	0100	1100	1000
0001	0101	1101	1001
0011	0111	1111	1011
0010	0110	1110	1010

Again, we use the variables as "coordinates" to define the regions of the map. The Gray code ordering is preserved, ensuring that adjacent cells differ by only one variable, which is crucial for simplifying Boolean expressions effectively:

		AB			
		00	01	11	
		00			
		01			
		11			
		10			

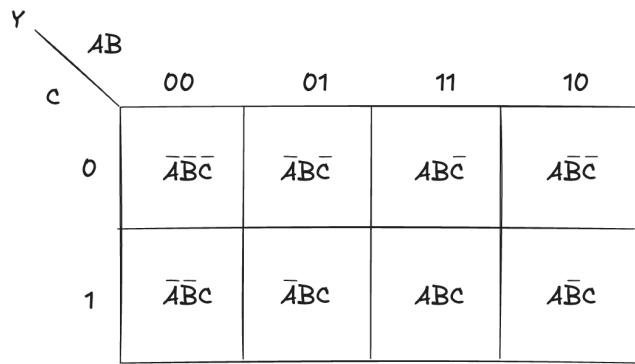
		A			
		00	01	11	
		00			
		01			
		11			
		10			

6.2.1 Construction

Suppose we have a logic function with three inputs, A, B, and C. The corresponding truth table lists all eight possible combinations of these three binary variables (from 000 to 111), together with the output value for each combination:

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

We can use the Karnaugh map as a **graphical reorganization** of this same information. Each cell corresponds to one row of the truth table, that is, to one specific combination of A, B, and C, and contains the value of the output for that combination:



In Boolean algebra, each cell corresponds to a minterm, a product term that is true for that single input combination. For instance, the top-left square in the map corresponds to the case where:

$$A = 0, B = 0, C = 0$$

The corresponding minterm is therefore:

$$\bar{A}\bar{B}\bar{C}$$

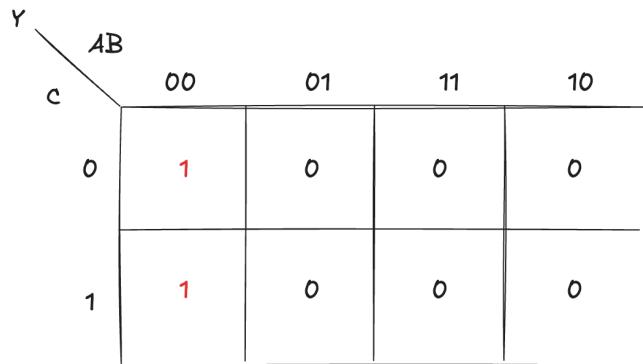
In the example truth table, the output for this case is 1, meaning the function is true when all inputs are zero. There are another 1, corresponding to the input combinations:

$$A = 0, B = 0, C = 1$$

The corresponding minterm is:

$$\bar{A}\bar{B}C$$

These two are placed in **adjacent cells** of the map, making it easy to see possible groupings for simplification:



When observing the structure of a the map, an important property becomes immediately clear: **each square differs from its adjacent squares by the value of only one variable**. This characteristic is what makes simplification so straightforward: two adjacent cells in the map represent minterms that share all but one variable in common. The variable that differs appears in its true form in one cell and in its complemented form in the other. For example, the minterms:

$$\overline{A}\overline{B}\overline{C}$$

and

$$\overline{A}\overline{B}C$$

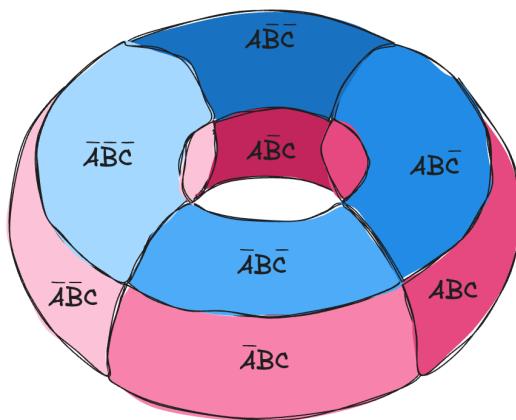
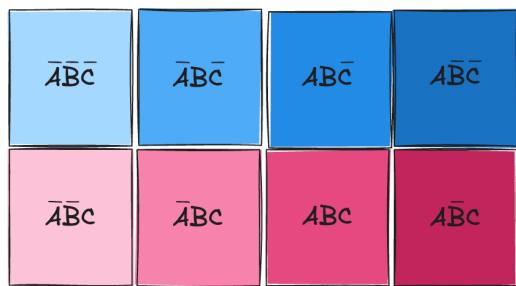
are adjacent because they differ only in variable C. This adjacency allows them to be combined algebraically into a simpler term:

$$\overline{A}\overline{B}$$

effectively eliminating the variable C.

6.2.2 Wrapping around

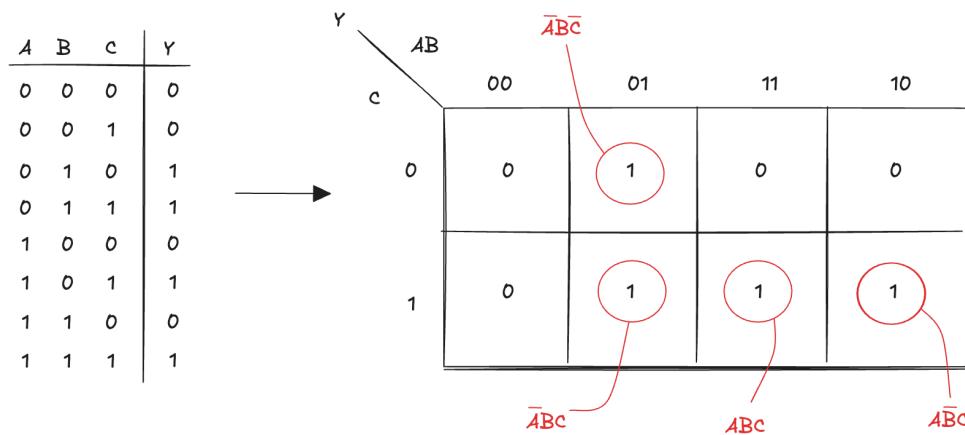
The K-map also has another subtle but crucial feature: it **"wraps around"**. The map is conceptually arranged on the surface of a **torus** (a doughnut shape), so that the first and last columns are considered adjacent, as are the first and last rows. This wrapping maintains the same one-variable difference at the map's edges. For example, in a three-variable K-map, the first column (where A=0, B=0) and the last column (A=1, B=0) are adjacent, because they differ only in A. Similarly, the top row (C=0) and the bottom row (C=1) are also adjacent:



This spatial organization (adjacency by one-variable difference, Gray code ordering, and wrap-around continuity) is what makes the K-map a powerful visualization tool. It allows us to immediately see which terms can be combined, leading to the simplest possible Boolean expressions with minimal effort.

6.2.3 SOP Synthesis

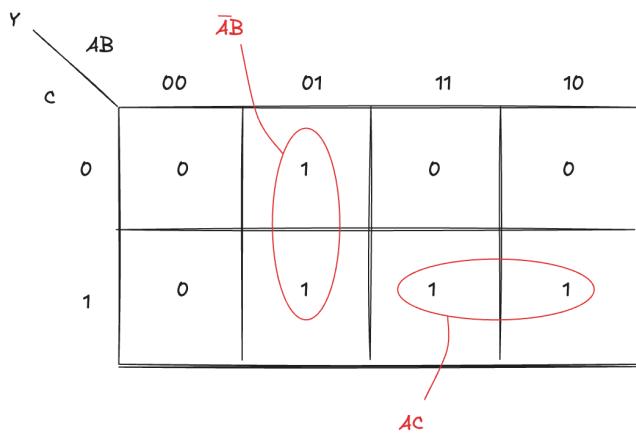
Consider a truth table for a function with three inputs (A, B, C) and copy the function values into the corresponding Karnaugh map cells:



The canonical SOP expression can be derived directly from the map by identifying the minterms (cells with output 1) and writing them as product terms:

$$Y = \overline{A}B\overline{C} + \overline{A}BC + ABC + A\overline{B}C$$

Remember that two cells are adjacent when they are at distance 1 (that is, when their corresponding input combinations differ by the value of only one variable). On the map, this means we can group together adjacent 1s that differ in a single variable, forming one-dimensional subcubes, also known as **groupings**:



In each grouping, **one variable changes** its value while the others remain constant. For every grouping, we can therefore write a term that includes only the variables that do not change, ignoring the one that varies. From the horizontal grouping we obtain the term:

$$AC$$

This corresponds to the region where $A=1$ and $C=1$, while B takes on different values. Similarly, for the vertical grouping, we have:

$$\overline{A}B$$

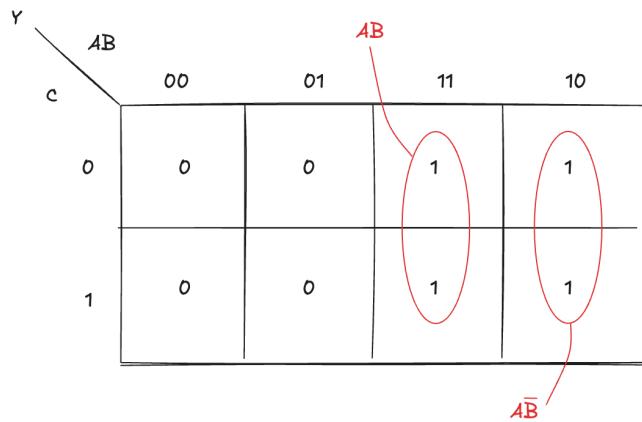
This represents the area where $A=0$ and $B=1$, while C varies. Finally, by summing the two terms, we directly obtain the **minimized expression**:

$$\$ Y = \overline{A}B\overline{C} + AB\overline{C} + ABC + A\overline{B}C \$$$

This is exactly the same result we can derive using algebraic manipulation:

$$\begin{aligned} Y &= \overline{A}B\overline{C} + \overline{A}BC + ABC + A\overline{B}C \\ &= \overline{A}B(\overline{C} + C) + AC(B + \overline{B}) \\ &= \overline{A}B + AC \end{aligned}$$

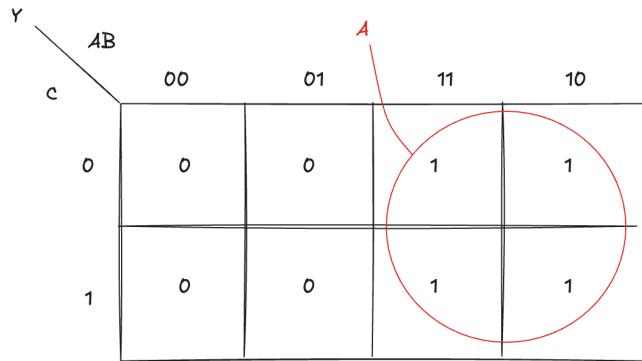
Now, consider the following map:



From the two unidimensional groupings, we can derive the minimized expression:

$$Y = AB + A\bar{B}$$

However, we can also form a **two-dimensional grouping** that includes all four 1s:



In this subcube, the only variable that does not change is A: thus, the function is reduced to the expression:

$$Y = A$$

We could reach the same conclusion using algebraic manipulation:

$$Y = AB\bar{C} + A\bar{B}\bar{C} + ABC + A\bar{B}C$$

$$= A\bar{C}(B + \bar{B}) + AC(B + \bar{B})$$

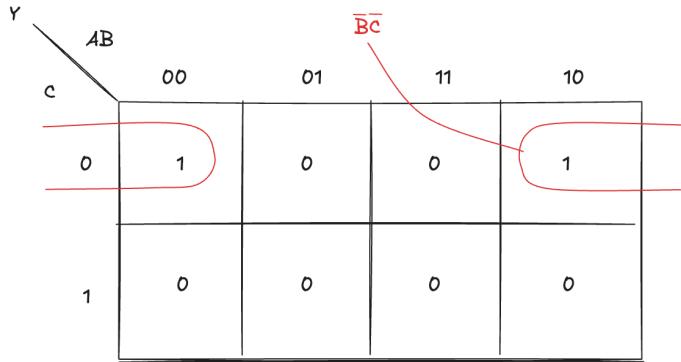
$$= A(\bar{C} + C)$$

$$= A$$

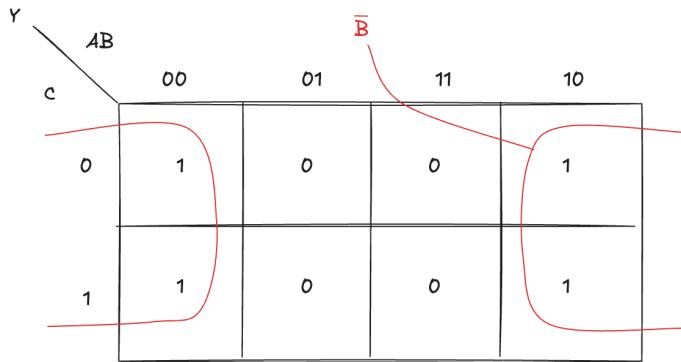
In general, to achieve the most effective simplification, we must **identify the largest possible subcubes**, that is the biggest groupings of adjacent 1s that can be formed on the map. If no

larger subcube exists that completely covers a given grouping, that grouping is called a **prime implicant**.

Remember that the map is **toroidal**, meaning it wraps around both horizontally and vertically. This property allows us to group 1s that are on opposite edges of the map:



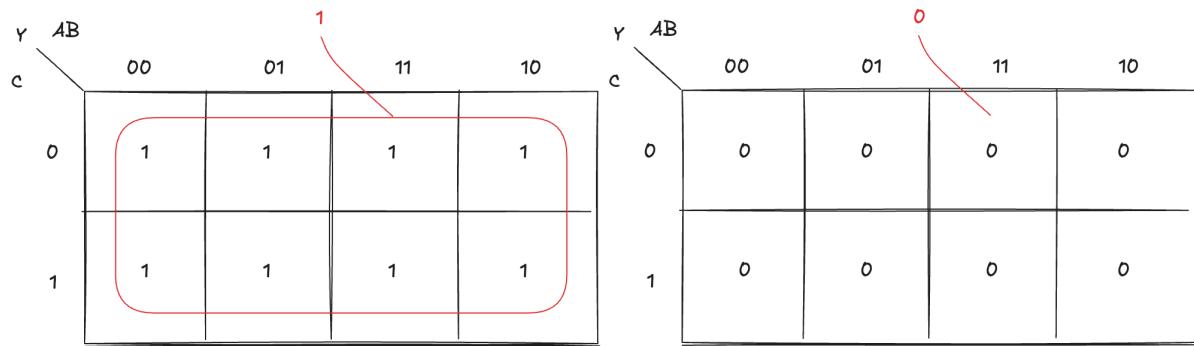
Although it may not seem so at first glance, the two cells in the map are **logically adjacent**. As we have discussed, the **top and bottom edges**, as well as the **left and right edges**, are adjacent. Indeed, if we compare each pair of the highlighted cells, we can see that **only variable A changes value**, confirming that they differ by exactly one variable and are therefore adjacent. Consider another example:



The largest subcube containing the 1s is highlighted:

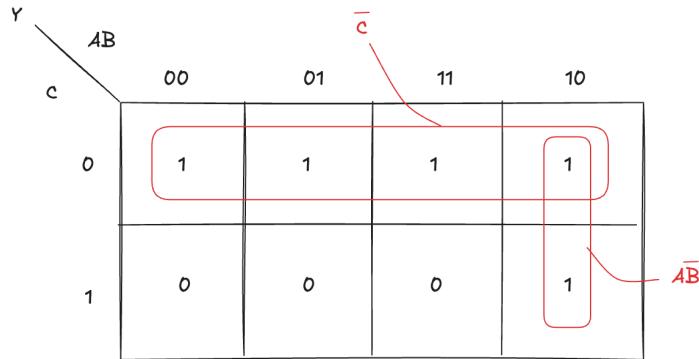
$$Y = B$$

Let us now consider two limiting cases:



In the map on the left, every cell contains a 1, representing the constant function $Y = 1$. Conversely, the map on the right contains only 0s, corresponding to the constant function $Y = 0$.

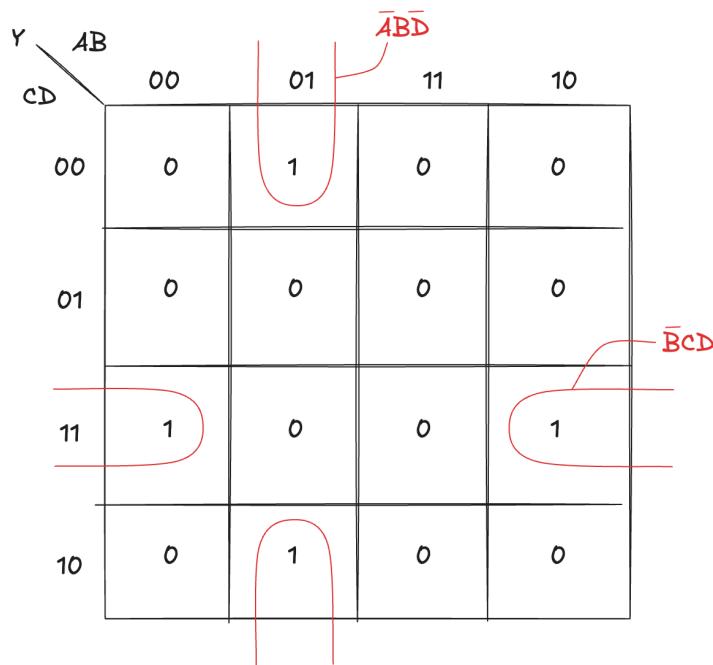
It is important to note that a single 1 on the map **can belong to more than one group**. In other words, the same minterm may be covered by multiple overlapping groupings. This redundancy is intentional: forming larger groups helps eliminate more variables and leads to a simpler overall expression. When different groups share one or more cells, they represent alternative simplifications that all contribute to the final minimized function. The goal is not to avoid overlap but to **ensure that all 1s are covered by at least one group** while achieving the greatest possible simplification:



In the example:

$$Y = A\bar{B} + \bar{C}$$

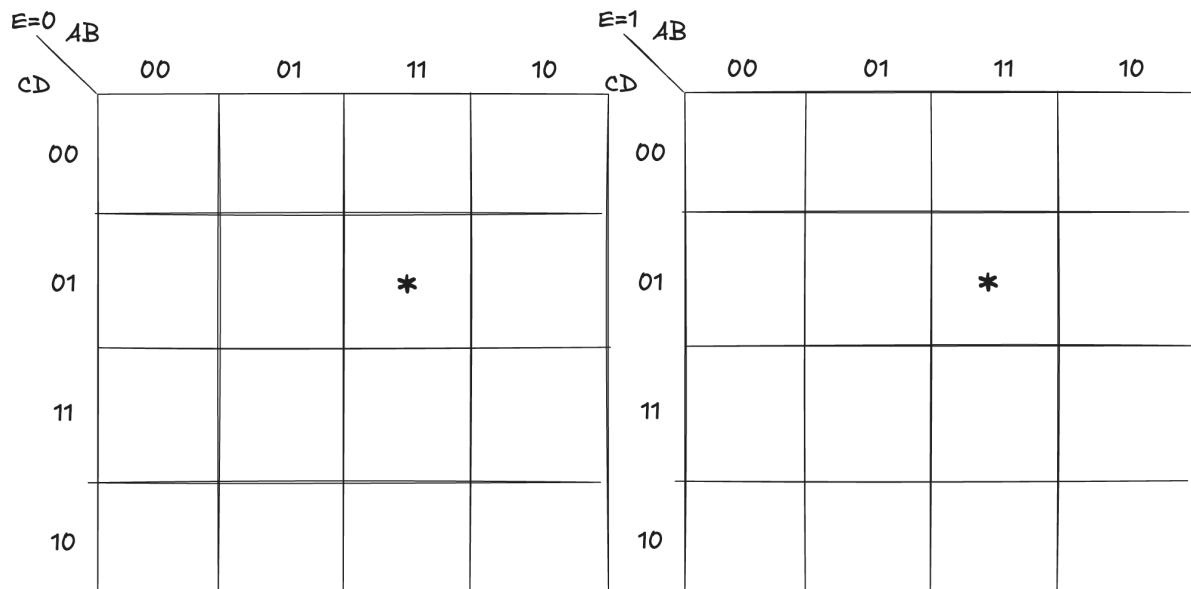
The same principle applies to the four-variable map, where adjacency still corresponds to minterms that differ in the value of a single variable, as illustrated in the following example:



For this map:

$$Y = \overline{AB}\overline{D} + \overline{B}CD$$

Five-variable maps (order-5) can be represented by two order-4 maps, in which cells in the same position in the two maps are adjacent (imagine one map on the top of the other):



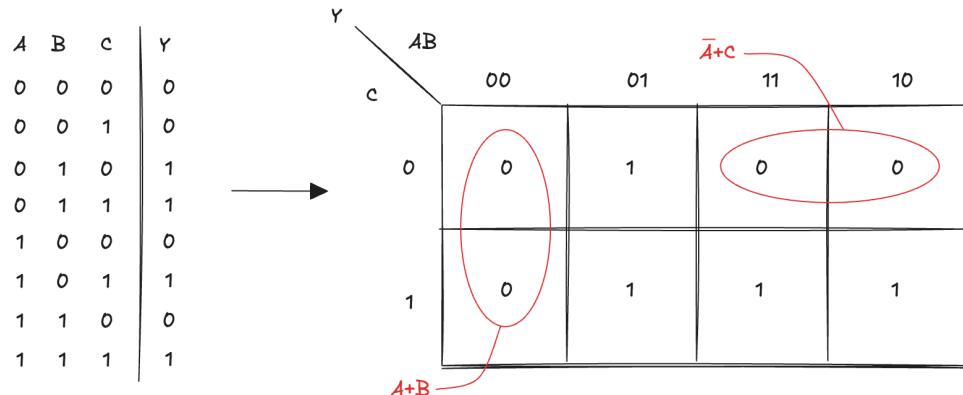
The cells marked with asterisks are adjacent, meaning they differ by only one variable. They can thus be combined into a single group, allowing the variable E to be eliminated from the

resulting simplified expression. Six-variable maps are made with four order-4 maps. However those maps are too complicated to represent and use, so they are rarely employed in practice.

6.2.4 POS Synthesis

In the AND-OR synthesis, we construct the function by focusing on the 1 of its truth table. Each 1 corresponds to a product term (an AND of input variables), where a variable appears in its direct form if its value is 1 and in complemented form if its value is 0. The outputs of all these AND gates are then combined through an OR operation to obtain the final expression.

In contrast, the **OR-AND synthesis** focuses on the 0 of the function. Each 0 is represented as a sum term (an OR of input variables), with each variable appearing direct if its value is 0 and negated if its value is 1. All these OR terms are then combined through an AND operation. We can consider the same example used for AND-OR synthesis:



$$Y = (A + B)(\bar{A} + C)$$

We can show that it is equivalent to the previous expression derived using AND-OR synthesis:

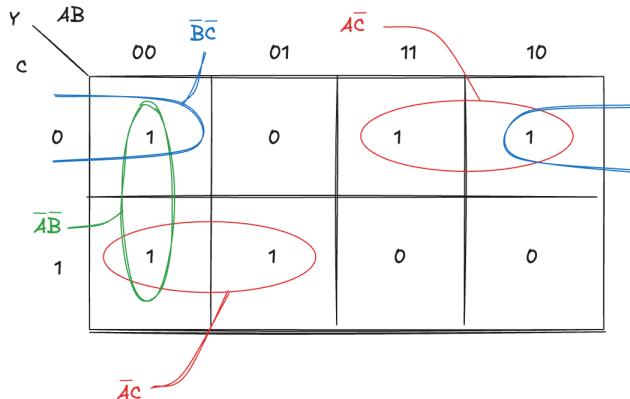
$$Y = \bar{A}B + AC$$

Expanding the OR-AND expression:

$$\begin{aligned} Y &= (A + B)(\bar{A} + C) \\ &= A\bar{A} + AC + B\bar{A} + BC \\ &= 0 + AC + \bar{A}B + BC \\ &= \bar{A}B + AC + BC \text{ (consensus)} \\ &= \bar{A}B + AC \end{aligned}$$

6.2.5 Multiple Minimal Forms

It is important to note that the **minimal set of prime implicants is not necessarily unique**. When simplifying a Boolean function, there may be **multiple valid combinations of prime implicants** that cover all the required 1s on the map. Each of these combinations yields a logically equivalent minimal expression, even though the individual terms may differ:



$$Y = A\bar{C} + \bar{A}C + \bar{A}\bar{B}$$

$$Y = A\bar{C} + \bar{A}C + \bar{B}\bar{C}$$

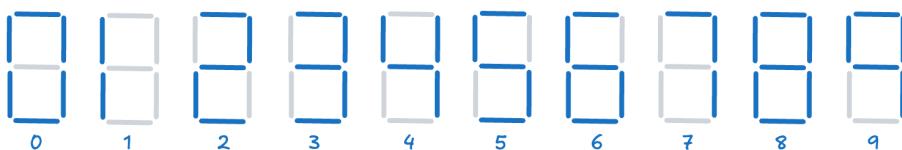
Both forms represent the same Boolean function. The difference lies in which implicants are chosen to cover the overlapping 1s, demonstrating that the **minimal expression is not unique**, since multiple sets of prime implicants can lead to equally simplified results.

This situation typically arises when certain 1s can be covered by more than one prime implicant of the same size. In such cases, the designer can choose among alternative minimal forms based on secondary criteria, such as implementation cost, gate count, or signal availability in a larger circuit.

Thus, **Karnaugh map simplification does not always lead to a single unique solution**, but rather to a **set of equally minimal alternatives**, all of which satisfy the same logical function.

6.3 Seven-segment Display Controller

A **seven-segment display Controller** is a combinational logic circuit designed to drive a display which can show decimal digits from 0 to 9:



The display can represent 10 different decimal digits (from 0 to 9), and we need a unique input combination for each of those digits. The amount of information required to uniquely encode N distinct states using a binary code is given by

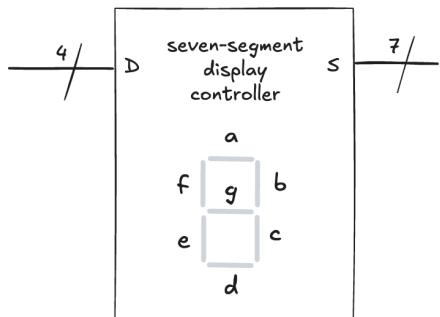
$$D = \log_2 N$$

In our case, $N=10$. Thus,

$$D = \log_2 10 \approx 3.32$$

Since the number of bits must be an integer, we round up to the next whole number, obtaining 4 bits. However, with 4 bits, we can represent up to 16 different combinations, which is more than sufficient to encode the ten decimal digits. So we have six **unused combinations** that do not correspond to any decimal digit.

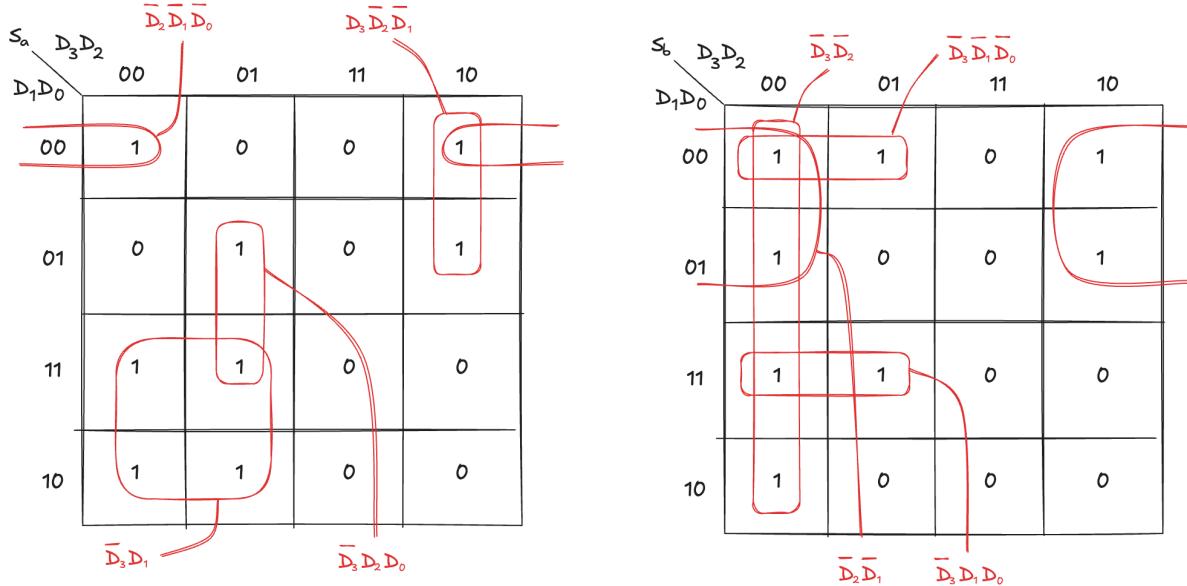
The circuit generates seven output signals S , each corresponding to one of the segments of the display. By controlling which segments are turned on (logic 1) or off (logic 0), the controller forms the visual representation of the desired digit:



For example, to display the digit 0, all segments except the middle one (g) are activated. To display the digit 1, only segments b and c are turned on, and so on. The following truth table summarizes these relationships, showing for each 4-bit input which of the seven outputs should be active:

$D_3\ D_2\ D_1\ D_0$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0 → 0 0 0 0	1	1	1	1	1	1	0
1 → 0 0 0 1	0	1	1	0	0	0	0
2 → 0 0 1 0	1	1	0	1	1	0	1
3 → 0 0 1 1	1	1	1	1	0	0	1
4 → 0 1 0 0	0	1	1	0	0	1	1
5 → 0 1 0 1	1	0	1	1	0	1	1
6 → 0 1 1 0	1	0	1	1	1	1	1
7 → 0 1 1 1	1	1	1	0	0	0	0
8 → 1 0 0 0	1	1	1	1	1	1	1
9 → 1 0 0 1	1	1	1	0	0	1	1
- 1 0 1 0	0	0	0	0	0	0	0
- 1 0 1 1	0	0	0	0	0	0	0
- 1 1 0 0	0	0	0	0	0	0	0
- 1 1 0 1	0	0	0	0	0	0	0
- 1 1 1 0	0	0	0	0	0	0	0
- 1 1 1 1	0	0	0	0	0	0	0

We derive the Karnaugh maps for each of the seven outputs:



$$S_a = \overline{D_2} \overline{D_1} \overline{D_0} + D_3 \overline{D_2} \overline{D_1} + \overline{D_3} D_1 + \overline{D_3} D_2 D_0$$

$$S_b = \overline{D_3} \overline{D_2} + \overline{D_3} \overline{D_1} \overline{D_0} + \overline{D_2} \overline{D_1} + \overline{D_3} D_1 D_0$$

In a similar way, the logic expressions for the remaining segments can also be derived from their corresponding truth tables; however, we can exploit the “don’t-care” conditions to further simplify each Boolean function.

6.4 Don’t Cares

The “don’t-care” condition refers to input combinations or output values that are **irrelevant** to the intended operation of the circuit. They are denoted by the symbol X, meaning that the corresponding entry can be considered either 0 or 1. We can use don’t-cares to better optimize Boolean expressions: when certain input combinations never occur, or when the output is unimportant for specific cases. In a Karnaugh map, these X’s can be included in groups to form larger or fewer loops, further reducing the logic complexity, but they can also be ignored if they do not contribute to simplification.

Considering the seven-segment display, only ten input combinations are needed to represent the decimal digits from 0 to 9, leaving six combinations unused. When the input corresponds to a combination that does not represent a valid decimal digit, there are **two possible design approaches**:

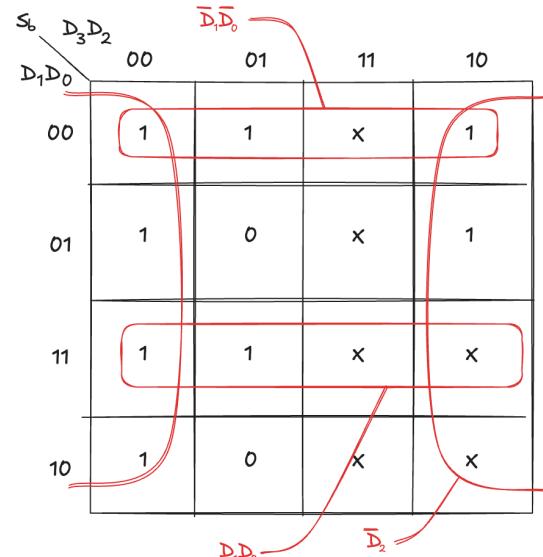
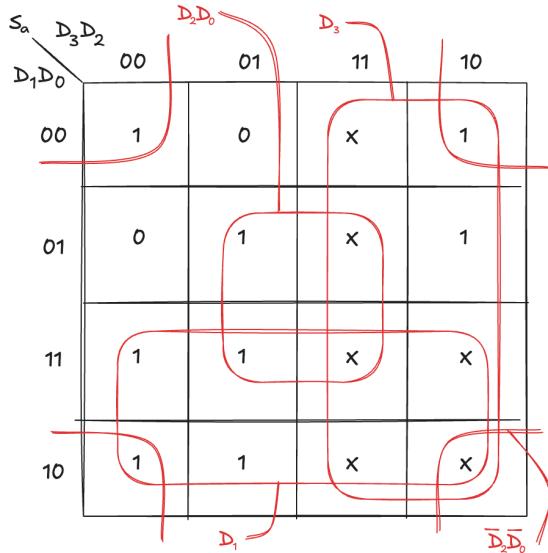
- All segments off: the display remains blank (as we have done in the previous example).

- **Don't-care handling:** the actual output pattern for these unused combinations is **ignored**.

Since the purpose of the device is to decode decimal digits, we assume that the input ABCD will always take values from 0000 to 1001 (that is, from 0 to 9 in decimal). If we are not concerned with what happens for the unused combinations, we can assign them the "don't-care" symbol (X) in the truth table:

$D_3\ D_2\ D_1\ D_0$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0 → 0 0 0 0	1	1	1	1	1	1	0
1 → 0 0 0 1	0	1	1	0	0	0	0
2 → 0 0 1 0	1	1	0	1	1	0	1
3 → 0 0 1 1	1	1	1	1	0	0	1
4 → 0 1 0 0	0	1	1	0	0	1	1
5 → 0 1 0 1	1	0	1	1	0	1	1
6 → 0 1 1 0	1	0	1	1	1	1	1
7 → 0 1 1 1	1	1	1	0	0	0	0
8 → 1 0 0 0	1	1	1	1	1	1	1
9 → 1 0 0 1	1	1	1	0	0	1	1
- 1 0 1 0	x	x	x	x	x	x	x
- 1 0 1 1	x	x	x	x	x	x	x
- 1 1 0 0	x	x	x	x	x	x	x
- 1 1 0 1	x	x	x	x	x	x	x
- 1 1 1 0	x	x	x	x	x	x	x
- 1 1 1 1	x	x	x	x	x	x	x

These don't-care conditions **can then be treated as either 0 or 1, whichever leads to the simplest possible circuit** during logic minimization:



$$S_a = D_3 + D_1 + \overline{D_2} \overline{D_0} + D_2 D_0$$

$$S_b = \overline{D_3} + \overline{D_1} \overline{D_0} + D_1 D_0$$

The presence of don't-cares **allows larger groupings** and thus fewer product terms compared to what would be possible without them. This results in simpler logic, fewer gates, and a more

efficient hardware implementation. However, their use also introduces certain risks and potential drawbacks. When a designer assumes that specific input combinations will never occur, the resulting circuit may **behave unpredictably if such combinations do appear in practice** (for example, due to noise, timing glitches, or unexpected input values from other parts of the system). This can lead to incorrect outputs or unstable display behavior. In **safety-critical** or **high-reliability** applications, these undefined states may propagate through the logic, causing unpredictable or even hazardous system responses. Therefore, although don't-cares simplify the design and save hardware resources, they must be used carefully and only when it is guaranteed that the corresponding input conditions truly cannot occur during normal operation.

6.5 Quine–McCluskey Method

Karnaugh maps, which become impractical for simplifying Boolean expressions involving more than four variables. The **Quine–McCluskey method** can be thought of as the **algorithmic version*** of the manual process used with Karnaugh maps. Its main intuition lies in replacing the visual pattern recognition of adjacent 1 in a K-map with a **systematic, tabular comparison** of binary terms. Conceptually, the method explores the same simplification principle that underlies Boolean minimization: **if two terms differ only by the value of one variable, that variable can be eliminated**. The method is useful because it provides a **automatable process**. Unlike K-maps, which rely on human pattern recognition and are limited to about four variables, Quine–McCluskey can handle many more variables and forms the foundation of **logic minimization algorithms** used in computer-aided design (CAD) tools today. In order to illustrate the method, consider the following function of four variables:

A	B	C	D	f
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

$f(A,B,C,D) = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + A\bar{B}C\bar{D} + ABC\bar{D}$

The algorithm's first phase performs an exhaustive expansion, identifying all **prime implicants** that cannot be simplified further. The second covering step selects the smallest set of these prime implicants that still represents all the original 1 in the truth table.

6.5.1 Expansion Phase

The first step is to **express each minterm with a binary number** (with 1s and 0s representing the true and complemented variables) and **group them according to the number of 1s** they contain:

Minterm	Binary	Number of 1s
m_0	0000	0
m_1	0001	1
m_2	0010	1
m_5	0101	2
m_6	0110	2
m_7	0111	3
m_8	1000	1
m_9	1001	2
m_{10}	1010	2
m_{14}	1110	3

As m_0 has no 1 in its representation it is kept in one group (0). Similarly, m_1, m_2 and m_8 contain one 1 in their representation so they are kept in the next group (1). And so on: m_5, m_6, m_9 , and m_{10} in the next group (2), m_7 and m_{14} in the next group(3):

Group	Minterms	Binary Numbers
0	m_0	0000
1	m_1, m_2, m_8	0001, 0010, 1000
2	m_5, m_6, m_9, m_{10}	0101, 0110, 1001, 1010
3	m_7, m_{14}	0111, 1110

Then we **compare the minterms in adjacent groups** (groups that differ by one in the number of 1s) to **identify pairs that differ by exactly one bit**. When such a pair is found, we combine them into a new term by replacing the differing bit with a dont'care (X), indicating that this variable can take either value (0 or 1). Also, place a check mark (V) to every minterm that has

been combined with at least one minterm. This process effectively eliminates variables from the expression. For example, comparing m_0 (0000) from group 0 with m_1 (0001) from group 1, we see they differ only in the last bit. We combine them to form the new term 000-. This is the table of minterms showing which have been combined in our example:

Minterm	Combined
m_0	V
m_1	V
m_2	V
m_5	V
m_6	V
m_7	V
m_8	V
m_9	V
m_{10}	V
m_{14}	V

And this table summarizes the combinations made:

Group	Pair Minterms	Combined Term
0 & 1	m_0 (0000), m_1 (0001)	000X
0 & 1	m_0 (0000), m_2 (0010)	00X0
0 & 1	m_0 (0000), m_8 (1000)	X000
1 & 2	m_1 (0001), m_5 (0101)	0X01
1 & 2	m_1 (0001), m_9 (1001)	X001
1 & 2	m_2 (0010), m_6 (0110)	0X10
1 & 2	m_2 (0010), m_{10} (1010)	X010
1 & 2	m_8 (1000), m_9 (1001)	100X
1 & 2	m_8 (1000), m_{10} (1010)	10X0
2 & 3	m_5 (0101), m_7 (0111)	01X1

Group	Pair Minterms	Combined Term
2 & 3	m_6 (0110), m_7 (0111)	011X
2 & 3	m_6 (0110), m_{14} (1110)	X110
2 & 3	m_{10} (1010), m_{14} (1110)	1X10

This combination process is **repeated iteratively**, comparing the newly formed terms to identify additional possible merges. In this step, two terms can be combined **only if they differ by a single bit** and have their **don't cares in the same positions**. Terms with don't cares in different positions cannot be combined, as they would differ in more than one variable. The following table summarizes the pairs of terms that can be combined in this iteration:

Minterm	Combined
m_0, m_1 (000X)	V
m_0, m_2 (00X0)	V
m_0, m_8 (X000)	V
m_1, m_5 (0X01)	
m_1, m_9 (X001)	V
m_2, m_6 (0X10)	V
m_2, m_{10} (X010)	V
m_8, m_9 (100X)	V
m_8, m_{10} (10X0)	V
m_5, m_7 (01X1)	
m_6, m_7 (011X)	
m_6, m_{14} (X110)	V
m_{10}, m_{14} (1X10)	V

And this table summarizes the new combinations made:

Term 1	Term 2	Combined Term
m_0, m_1 (000X)	m_8, m_9 (100X)	X00X
m_0, m_2 (00X0)	m_8, m_{10} (10X0)	X0X0
m_0, m_8 (X000)	m_1, m_9 (X001)	X00X
m_0, m_8 (X000)	m_2, m_{10} (X010)	X0X0
m_2, m_6 (0X10)	m_{10}, m_{14} (1X10)	XX10
m_2, m_{10} (X010)	m_6, m_{14} (X110)	XX10

This process continues **until no further combinations are possible**. The terms that have not been combined are identified as **prime implicants**. In this example, the prime implicants are:

- $\bar{A}\bar{C}D$ from (0X01)

- $\bar{A}BD$ from (01X1)

- $\bar{A}BC$ from (011X)

- $\bar{B}\bar{C}$ from (X00X)

- $\bar{B}\bar{D}$ from (X0X0)

- $C\bar{D}$ from (XX10)

6.5.2 Combination Phase

In the second phase of the Quine–McCluskey method, we need to select a minimal set of prime implicants that covers all the original minterms. This is done by constructing a **prime implicant chart**, which maps each minterm to the prime implicants that cover it:

	m_0 (0000)	m_1 (0001)	m_2 (0010)	m_5 (0101)	m_6 (0110)	m_7 (0111)	m_8 (1000)	m_9 (1001)	m_{10} (1010)	m_{14} (1110)
$\bar{A}\bar{C}D$ (0X01)		V		V	V					
$\bar{A}BD$ (01X1)					V			V		
$\bar{A}BC$ (011X)						V		V		
$\bar{B}\bar{C}$ (X00X)	V	V					V	V		

	m_0 (0000)	m_1 (0001)	m_2 (0010)	m_5 (0101)	m_6 (0110)	m_7 (0111)	m_8 (1000)	m_9 (1001)	m_{10} (1010)	m_{14} (1110)
$\bar{B}\bar{D}$ (X0X0)	V		V				V		V	
$C\bar{D}$ (XX10)			V		V			V		V

From the chart, we can identify **essential prime implicants**, those that cover minterms not covered by any other implicant. In this case, the essential prime implicants are:

- $\bar{B}\bar{C}$ (X00X) which is the only one covering m_9 (1001)
- $C\bar{D}$ (XX10) which is the only one covering m_{14} (1110)

By selecting these essential prime implicants, we cover also other minterms:

- $\bar{B}\bar{C}$ (X00X) covers also m_0 (0000), m_1 (0001), and m_8 (1000)
- $C\bar{D}$ (XX10) covers also m_2 (0010), m_6 (0110), and m_{10} (1010)

Next, we look for the remaining minterms that are not yet covered and we need to **select additional prime implicants** to cover them, aiming to minimize the total number of implicants used. In our example:

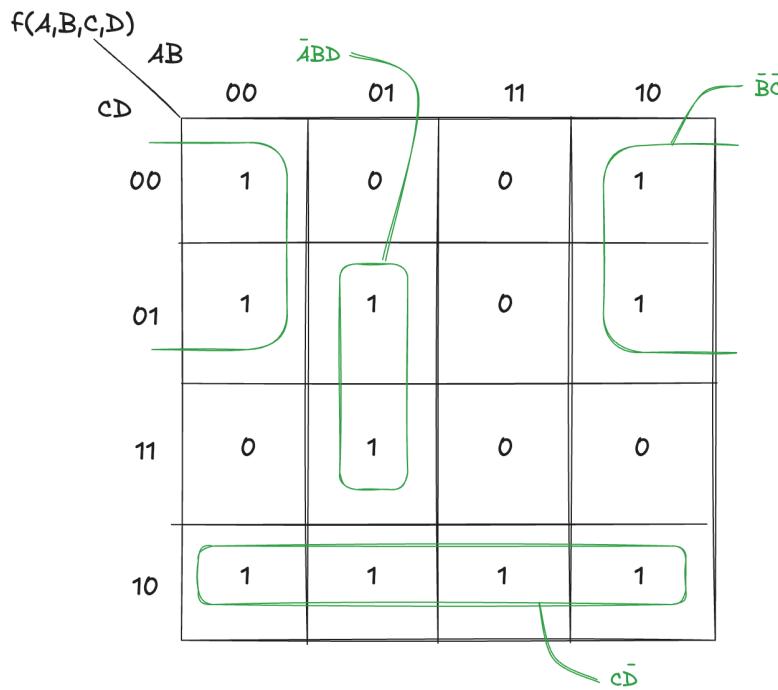
- m_5 (0101) is covered by both $\bar{A}\bar{C}D$ (0X01) and $\bar{A}BD$ (01X1)
- m_7 (0111) is covered by both $\bar{A}BD$ (01X1) and $\bar{A}BC$ (011X)

So we can choose $\bar{A}BD$ (01X1) to cover both m_5 and m_7 .

Finally, we can collect all essential and non-redundant implicants to form the minimized Boolean expression:

$$f(A, B, C, D) = \bar{B}\bar{C} + C\bar{D} + \bar{A}BD$$

We can check the result using a Karnaugh map:



The Quine–McCluskey method provides a systematic and automatable approach to Boolean minimization, making it particularly suitable for **computer-aided design (CAD)** tools used in digital logic synthesis. Its tabular structure translates well into software algorithms that can handle logic reduction without human intervention. However, its **computational complexity grows exponentially** with the number of variables, since the number of possible combinations and prime implicants increases rapidly. For this reason, while Quine–McCluskey is ideal for educational purposes and small-scale circuits, modern CAD systems employ **more advanced heuristic** (such as the **Espresso algorithm**) to find **near-optimal implementations** of logic functions when exact algorithms become computationally infeasible.

7 Hardware Description Languages (HDL)

The process of finding an efficient set of logic gates to perform a given function is labor intensive and error prone, requiring manual simplification of truth tables or Boolean equations.

Designers discovered that they were far more productive if they worked at a **higher level of abstraction**, specifying just the logical function and allowing a **computer-aided design (CAD) tool** to produce the optimized gates. The specifications are generally given in a **Hardware Description Language (HDL)**. Unlike standard programming languages (such as C, Java, or Python), they do not specify the sequence of operations that a processor must execute to per-

form a computation. Instead, they **describe the elements that make up the digital circuit** capable of carrying out the required computation.

An HDL specification, therefore, is not something that can be "executed" in the traditional sense and should be understood as fundamentally different from a program or algorithm. However, an HDL specification can be **simulated** using appropriate tools to verify the behavior of the circuit described.

Once verified, an HDL design can also be **synthesized**, which means that the abstract description is automatically translated into a netlist of physical gates, registers, and connections that can be implemented on actual hardware, such as an FPGA (an integrated circuit that can be programmed and reconfigured by the user after manufacturing to implement any digital logic design) or ASIC (a custom-manufactured chip designed to perform one specific function).

FPGA is the most common device that we can use for our HDL. All major FPGA manufacturers have a set of software and hardware tools that we can use to perform modelling, simulation and synthesis. If we are not interested in proprietary libraries, we can adopt open-source solutions (e.g. GHDL), which will allow us to compile and simulate VHDL code. Synthesis can be accomplished using a free-license version of any major FPGA manufacturer's software tool (e.g. Xilinx Vivado).

Two of the most widely used HDLs are **VHDL** and **Verilog**. Both allow designers to describe complex digital systems at different levels of abstraction, from high-level functional descriptions down to gate-level implementations. In these notes, we will focus primarily on VHDL, which is particularly common in academic and industrial contexts.

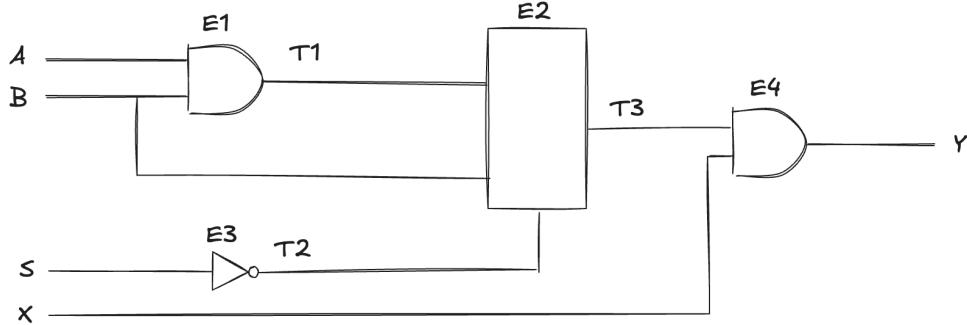
7.1 Levels of Abstraction

HDLs allow us to specify digital circuits at **different levels of abstraction**, from high-level behavioral descriptions to detailed gate-level implementations. This flexibility is crucial because it enables designers to **manage the complexity** of digital systems effectively. At a higher level of abstraction, it's easier to describe and verify overall functionality without worrying about low-level details, while lower levels allow precise control over timing, area, and physical implementation. By moving between these levels, engineers can design, test, and optimize circuits in a systematic and efficient way.

7.1.1 Structural Level (Gate-Level)

At the **lowest level of abstraction**, we can view a circuit as a **graph in which the nodes represent logical elements** (such as logic gates or entire subcircuits), while the edges represent

the connections between these elements. An example of such a view is shown in the following figure:



At this level, VHDL **explicitly describes the elements that make up a circuit and how they are connected**, while the information about the functional transformations that the data undergoes remains implicit. In particular, this type of representation explicitly specifies the following details:

- The names and types of the circuit's primary inputs and outputs (e.g., A, B, S, X, Y)
- The types of logic elements used (e.g., AND, NOT)
- The names of the instances of these logic elements (e.g., E1, E2, E3, E4)
- The names of the internal signals (e.g., T1, T2, T3)
- The connections between signals and the ports of the components

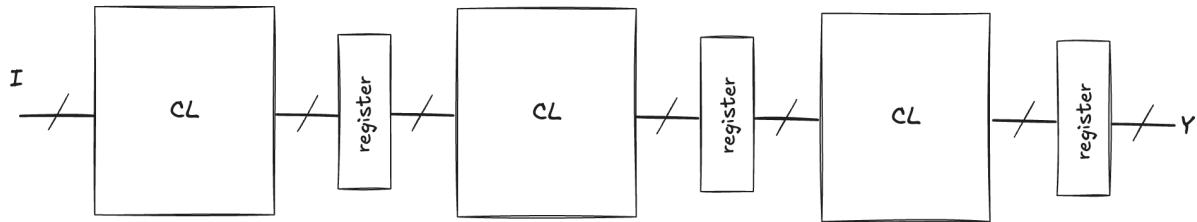
Such a representation is commonly called a **netlist**, and it is **the model that most closely resembles the final physical implementation of the circuit** under design.

7.1.2 RTL Level (Data-Flow)

At a **higher level of abstraction**, we move from the gate-level to the **RTL (Register Transfer Level)**, also known as the data-flow level. In this paradigm, the specification explicitly **describes the transformations that data undergoes as it propagates through the circuit**. Specifically, the circuit is viewed as consisting of two types of elements:

- **Combinational networks**, which define how data is transformed using algebraic expressions, arithmetic operations, and conditions.
- **Memory elements**, which store intermediate results of more complex computations.

This division explains the term "Register Transfer": the specification describes how data is transferred and processed between the registers in the network. Structurally, an RTL description can be seen as a sequence of combinational logic blocks separated by registers:



In this diagram, "I" represents a generic set of inputs, while "Y" represents a generic set of outputs. The rectangles illustrate purely combinational networks of arbitrary complexity, and the registers symbolize general memory elements. From this diagram, it is clear that at the RTL level, a VHDL specification defines both the transformations that signals undergo and the points where these signals are stored. It is worth noting that this organization applies not only to the entire circuit but also to each of its subcomponents, regardless of their size or complexity. For simpler subcircuits, there may be no need to store any data, meaning they may not contain any registers at all. In such cases, the concept of Register Transfer Level still holds, as long as the primary inputs and outputs of the circuit or subcircuit are regarded as equivalent to registers.

In a Structural model, the focus is on how the circuit is physically built. The designer explicitly instantiates all the components and describes how they are connected. In contrast, an RTL model describes how data moves and transforms through the circuit without specifying exactly which physical gates or components will be used. RTL modeling allows designers to manage complex designs more easily and to focus on functional correctness and data flow rather than low-level connectivity.

7.1.3 Algorithmic Level (Behavioral)

The behavioral level represents the **highest level of abstraction**. At this level, the functionality of a circuit is described in terms of one or more algorithms. In this approach, neither the detailed structure of the circuit nor the individual data transformations are explicitly defined. Unlike RTL descriptions, the register-transfer structure is not visible here: it is up to the synthesis tool to determine how to schedule operations across clock cycles, based on constraints provided by the designer. for example, a target minimum clock frequency or a maximum area constraint. In these notes, we will not cover behavioral-level specifications in detail.

7.1.4 Specification

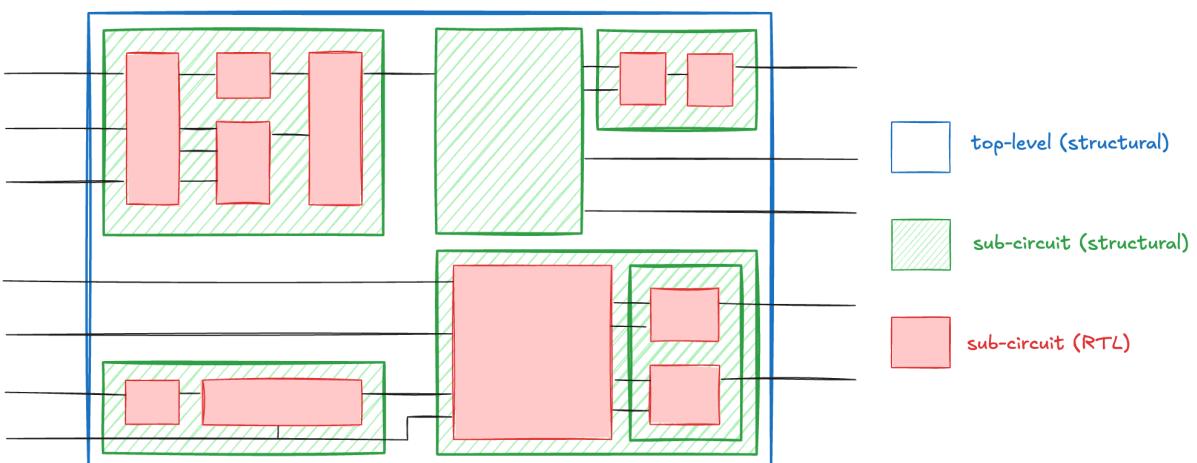
A specification is **the description of a complex digital circuit in which some parts are defined at the structural level, while other parts are expressed at the RTL level**. In practice, this reflects a typical and practical use of different specification styles within a single complex design.

When a circuit reaches a certain level of complexity, it is no longer described as a single, monolithic entity but rather as a **hierarchical structure composed of interconnected subcircuits**. Each subcircuit performs a well-defined function and should be as **isolated** and **independent** from the rest of the system as possible. Simpler subcircuits are then interconnected to form more complex subcircuits, ultimately achieving the complete functionality of the overall circuit.

This naturally leads to two distinct types of subcircuits or modules that need to be specified:

- **Processing modules**, in which data transformations are explicitly described in the code.
- **Integration modules**, in which more complex functionalities are built by interconnecting processing modules.

This situation can be illustrated graphically as follows:



The figure shows both the hierarchical structure of the complete circuit and the different specification styles used for its modules. The entire circuit is itself a module (commonly called the **top-level module**) whose inputs and outputs correspond to the primary inputs and outputs of the design. Its structure is defined by the interconnection of multiple subcircuits. Each submodule also has its own inputs and outputs, which may be connected either to the primary inputs and outputs of the circuit or to internal signals.

As a general rule, the description style for these subcircuits is structural for all modules except for the leaf nodes of the hierarchy, which are instead described at the register transfer level. Why structural is used for higher levels and RTL for leaf nodes? At first glance, it seems counterintuitive: RTL is a higher level of abstraction than structural (gate-level) descriptions, so it seems logical to use it for the big, top-level view. Where structural feels lower-level and closer to the physical hardware, so we might expect to use it at the bottom (leaves). But in digital design, the reason we do the opposite is about modularity and manageability: **top-level modules are about integration, not computation**. Their main job is to connect submodules (processing

blocks, memories, IP cores, interfaces, etc.). So a structural description is perfect: it specifies how the parts are wired together, not how they work internally; leaf modules instead are where the actual data processing happens. Here we want to describe how data moves and transforms within the block: what registers store the results, how combinational logic computes outputs, how everything synchronizes with the clock. For this, the RTL style is ideal: we write the logic and let the synthesis tool decide exactly how to implement it in gates.

7.2 Entities

Every system, from the simplest to the most complex, is composed of well-defined functional units. These units (often called **modules** or **blocks**) are designed to **isolate a specific function** of the overall system. This serves two purposes: it provides a structured view of the design and breaks down a very complex problem (developing an entire system) into a set of smaller, more manageable subproblems.

As an example, consider a system that calculates the roots of a quadratic equation given the three coefficients a , b , and c . Assuming the discriminant is greater than or equal to zero, the solutions are:

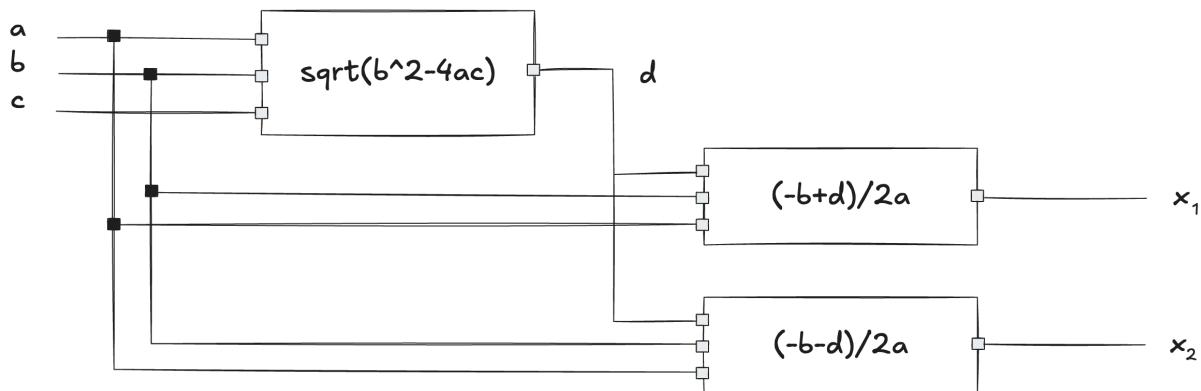
$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

One possible way to break down this problem into simpler subproblems could be:

1. Compute the square root of the discriminant: $d = \sqrt{b^2 - 4ac}$
2. Compute the first solution: $x_1 = \frac{-b + d}{2a}$
3. Compute the second solution: $x_2 = \frac{-b - d}{2a}$

Graphically, this decomposition could be represented as follows:



This decomposition is based on three modules: one for computing the discriminant and two for computing the solutions from d , a , and b . These three modules are relatively complex and sufficiently distinct from each other that they are unlikely to share resources or be easily optimized together. However, this breakdown can be further refined by observing that the solutions can also be derived by distributing the denominator:

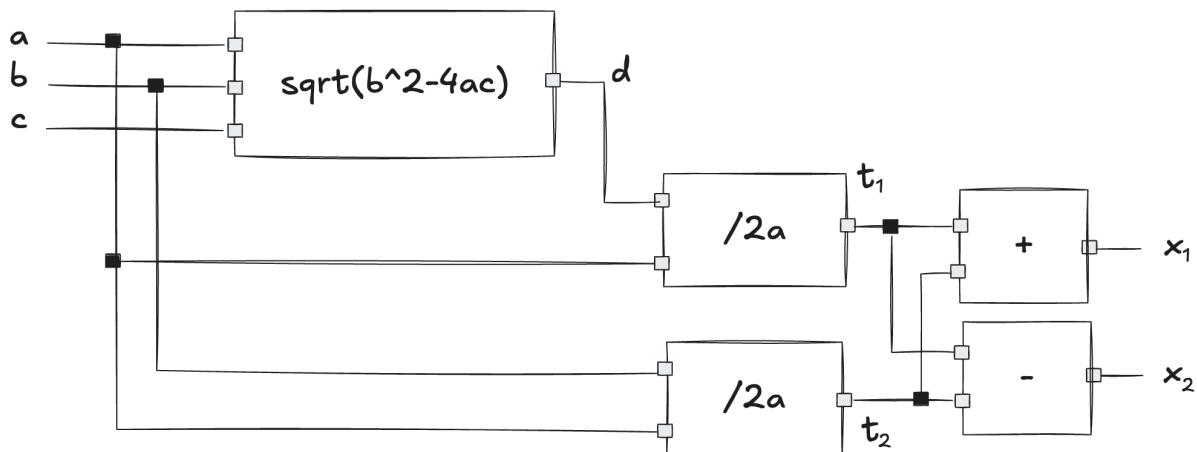
$$x_1 = \frac{-b}{2a} + \frac{d}{2a}$$

$$x_2 = \frac{-b}{2a} - \frac{d}{2a}$$

This new decomposition highlights five distinct operations:

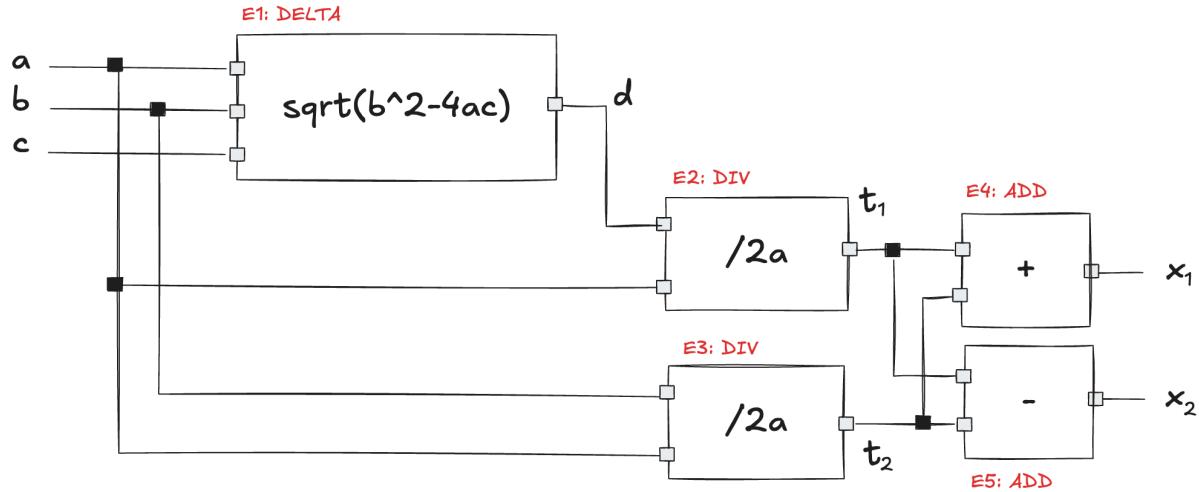
1. Compute the square root of the discriminant: $d = \sqrt{b^2 - 4ac}$
2. Compute the first term of the solutions: $t_1 = \frac{-b}{2a}$
3. Compute the second term of the solutions: $t_2 = \frac{d}{2a}$
4. Compute the first solution: $x_1 = t_1 + t_2$
5. Compute the second solution: $x_2 = t_1 - t_2$

Compared to the previous approach, we can see that the division by $2a$ is performed twice. This means that the same division block can be reused twice. Similarly, the module that performs the addition or subtraction of can be used for two steps. In practice, therefore, although the design is organized into five operations, only three of them are actually unique, the others are reused as needed. The representation of this decomposition is shown below:



We can exploit the reuse of certain modules in the design by introducing the distinction between **module** and **instance**. A module is a **unique entity consisting of an interface and its defined behavior**, whereas an instance represents a **specific use of a module** within the circuit's construction.

The block diagram of the last decomposition, therefore, consists of three unique modules (DELTA, DIV and SUM) and a total of five instances (E1, E2, E3, E4, E5). By labeling each instance with a clear name, we can make this distinction explicit:



Each module is described by clearly **separating** its two complementary aspects: **interface** and **behavior**. The interface defines the block's input and output signals and specifies how the block connects to other blocks (this is done using the **entity construct**). The behavior, on the other hand, describes how the inputs are processed or transformed to produce the outputs (this is defined using the **architecture construct**).

7.2.1 Entity construct

The interface of each module is described by an **entity declaration**. This construct specifies:

- the **name of the module**, which must be unique within a design;
- a **list of port declarations**, that define the signals used as inputs and outputs of the design entity
- if needed, a **set of parameters** (called **generics**) that can be used to adjust its properties.

The general syntax of an entity declaration is as follows:

```
entity entity_name is
  [generic( generic_list );]
  [port( port_list );]
end entity_name;
```

Each port declaration is structured according to the following syntax:

```
port_name[,port_name,...]: {in|out|inout} port_type;
```

The **name of the port** will be used in the architecture to refer to the signal connected to that port. The following keyword indicates the direction of the port and can be **in** for input ports, **out** for output ports, and **inout** for bidirectional ports. Notice that an in port can **only be read** (meaning it may appear only on the right-hand side of an assignment or in a conditional expression), whereas an out port can **only be written** (meaning it may appear only on the left-hand side of an assignment). While inout ports, can be both **read and written**. Finally, the **type of the port** must be specified to indicate the kind of data that can be transmitted through that port (e.g. a single bit, a vector of bits, etc.).

Similarly, the definition of a generic follows the syntax:

```
generic_name[,generic_name,...]: generic_type;
```

Unlike ports, generics do not have any direction and can have complex types, such as floating-point numbers, strings, or records.

As a simple example, consider the declaration of the interface for a basic AND gate with two input signals (A and B), each one bit wide, and one output signal (Y), also one bit wide. Additionally, we want to specify, for simulation purposes, a parameter delay that defines the propagation delay of the gate. This parameter must be assigned a value when the component is instantiated, so that the simulation can take it into account. The complete entity declaration is as follows:

```
entity and_gate is
  generic (delay: real);
  port (A, B: in bit;
        Y : out bit);
end and_gate;
```

Just as with source code for software development, it is good practice to **enrich the specification with comments** that **clarify** its key aspects. In VHDL, comments are introduced using **--** and extend to the end of the line. A commented version of the previous declaration might look like this:

```
entity and_gate is
```

```

-- Generics
generic(
    delay: real -- The delay in ns
);

-- Ports
port(
    -- Inputs
    A: in bit; -- First operand
    B: in bit; -- Second operand

    -- Outputs
    Y: out bit; -- Output
);

end and_gate;

```

Generics can be used, as we will see, to define configurable components intended not only for simulation but also for **synthesis**. Describing a component **parametrically** offers significant advantages in terms of clarity, conciseness, and code reusability. This last benefit is especially valuable when designing highly complex systems.

7.2.2 Architecture construct

The entity declaration defines the interface of a module, but it does not specify the functionality performed by the module or how that functionality is implemented. The behavior is described using the **architecture construct**, which follows the syntax below:

```

architecture architecture_name of entity_name is
    [declarations]
begin
    [implementation]
end architecture_name;

```

The first section consists of a **list of declarations** for elements used to implement the module's behavior, such as **constants**, **signals**, **user-defined types**, and **components**. These declarations are visible only within that specific architecture. Between the begin and end keywords is the **implementation section**, which contains the description of the functionality that the module

must perform. The functionality of an entire circuit is therefore distributed across the architecture declarations of the various entities that make up the system's modules. As an example, we can consider the architecture for the and gate described in the previous example:

```
architecture first of and_gate is
begin
    Y <= A and B;
end first;
```

In this simple example, all the signals used are either the module's inputs or its outputs. For this reason, no additional declaration section is needed. Note that the input signals (A, B) are only read, as required, while the output signals (Y) are only written to.

Each architecture is **associated with exactly one entity**. However, the reverse is not true: it is possible to **define multiple alternative architectures for the same entity** and choose which one to use before synthesis or simulation. This association of a specific architecture with an entity is called a **configuration declaration**. As an example, let's consider a second architecture for the and gate:

```
architecture second of and_gate is
    signal P: bit;
begin
    P <= A and B;
    Y <= P;
end second;
```

In this case, a temporary signal is used to compute an intermediate result. This signal must be **explicitly declared** in the declarative section of the architecture and **must have a type that matches the context** in which it is used. Finally, note that the signal do not have any direction and can therefore be both read from and written to. In projects of moderate complexity, having multiple architectures is usually of limited benefit. However, in larger and complex designs, this approach can be very useful. For example, when we want to maintain separate architectures for different synthesis strategies.

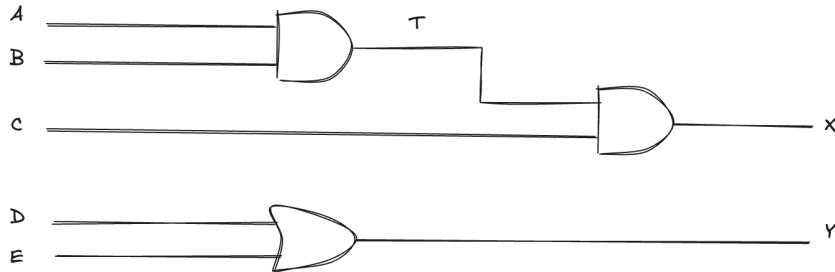
7.3 VHDL Computational Model

The implementation section consists of a set of statements such as assignments, conditional assignments, selection constructs, instantiations, and so on. In a typical programming language like C, these statements are **executed sequentially**, one after the other. In hardware description

languages, however, these statements **are executed concurrently**, their results are computed simultaneously. This distinction is crucial, as it reflects the fact that all elements of a real circuit process their inputs inherently in parallel. A simple example illustrates this concept:

```
architecture par_one of dummy is
    signal T: bit;
begin
    T <= A and B;
    X <= T and C;
    Y <= D or E;
end par_one;
```

The resulting circuit is shown in the following figure:



In digital circuits, logic gates are physical hardware components, they **continuously and simultaneously produce an output** based on their current inputs, they don't "wait their turn". However, from the circuit diagram it is clear that the value of the result X is available only after the first AND gate computes the logical product of A and B and the second AND gate uses that result to produce the final value. However, this is not accurate: in reality, the second AND gate continuously computes the logical product of its inputs, regardless of how or when those inputs were calculated. Therefore, the result X is **always available** and is **continuously updated**. Any change to inputs (A, B, or C) will **propagate through the circuit**, updating X according to the data dependencies implied by the expressions. In other words, the two AND gates and the OR gate process their inputs in parallel, even though the dependency between signals imposes an order on how intermediate results propagate. Based on this observation, **the order of the expressions can be rearranged without changing the behavior of the resulting circuit**. The following architecture is therefore completely equivalent to the previous one:

```
architecture par_two of dummy is
    signal T: bit;
begin
    Y <= D or E;
```

```
X <= T and C;
T <= A and B;
end par_two;
```

VHDL adopts an **event-driven computational model** in which signal changes trigger the re-evaluation of dependent statements, a mechanism that naturally supports the description of circuits. Consider a simple AND gate whose two inputs currently carry a logical 1 and 0. The physical device therefore produces a 0 at its output. If the input signals **remain unchanged**, the output will also **remain stable**. However, the moment one of the inputs changes value, the output may change as well. For instance, if the input that was 0 switches to 1, the output of the AND gate transitions from 0 to 1. In a real circuit this occurs because an input transition modifies the voltage on the gate terminal of one of the transistors inside the AND structure, altering its conduction state and therefore pulling the output to the appropriate logic level. The VHDL computational model is able to represent situations like this:

- when no input changes, a system stays in a stable state
- when an input changes, the system must react by recomputing affected signals

This observation leads to the fundamental principle: **VHDL descriptions are controlled by events, they are event-driven**. An event occurs when the value of a signal changes. Every time an event happens, the simulator determines which parts of the description depend on that signal and need to be re-evaluated. Consider an example fragment of an architecture that includes the following concurrent signal assignments:

```
X <= A or Z;
Y <= C and (not X);
Z <= B and D;
```

At first glance, the textual order of these statements does not match the logical dependencies among the signals. Yet VHDL guarantees that the behavior will be correct. This is precisely because of the event-driven execution model. Let us examine what happens when the signal B changes from 0 to 1:

- the change on B constitutes an event
- physically, this corresponds to a voltage transition at some point in the circuit, which then propagates
- in VHDL, the same phenomenon is simulated: when an event occurs on B, all expressions that read B must be re-evaluated.

In this case, the assignment:

A \leq B and D;

must be recalculated. If D is also 1, then Z changes from 0 to 1. This generates a new event, now on signal Z. Because Z changed, any statement that depends on Z must be re-evaluated. This affects:

X \leq A or Z;

If the new value of X changes, another event is produced on X, which in turn requires re-evaluating:

Y \leq C and (not X);

In this way, **events propagate through the network**, reflecting the propagation of electrical changes in an ideal combinational circuit. In the VHDL simulation model, this chain of re-evaluations happens in **zero physical time**, but with a well-defined ordering. Each re-calculation triggered by an event is scheduled **one delta cycle after the event that triggered it**:

- The initial event on B occurs at time T, then:
- Z is updated at time $T + \Delta T$
- X is updated at time $T + 2\Delta T$
- Y is updated at time $T + 3\Delta T$

Although we write these times as $T + n\Delta T$, the delta cycle duration ΔT represents zero real simulation time. Thus, from the point of view of the designer, the value of B appears to propagate instantaneously to Y. Delta cycles provide VHDL with a **mechanism to simulate the causality** of networks without introducing artificial delays.

7.4 Types

VHDL is a **flexible language**, it allows us to define **many different kinds of data types**, such as single bits, bit vectors, integers, floating-point numbers, enumerated types, records, and even user-defined types. This flexibility is **great for simulation**, where we often want to test a wide range of scenarios and write models that can look more like software.

However, when we **move from simulation to synthesis** (that is, when our design is turned into actual hardware) we have to remember that the hardware must physically implement whatever

types we've used. **Some types map naturally to hardware** elements (for example, a bit can be represented by a wire or a flip-flop that stores a 0 or 1). Other types, like real (floating point) or complex user-defined records, **do not have a straightforward or efficient hardware representation** or may not be supported at all by synthesis tools.

For this reason, in practice, designers usually **stick to a smaller synthesizable subset of data types**, since synthesis tools know how to translate these into real gates, registers, and wires.

7.4.1 Bit type

The **bit type** is the simplest type available and **represents a binary value** that can only take the logical values 0 and 1. Note that the constants 0 and 1 must be enclosed in single quotes ('0' and '1') to distinguish them from the integer numeric values 0 and 1. The operators defined for this type include **assignment operators**, **comparison operators**, and **logical operators**. The following statements are valid:

```
x <= A and B;
Y <= '1';
Z <= A xor (B and not C);
```

In digital circuits, we often have **many signals that are related**, for example, the individual bits of a binary number, the address lines of a bus, or the data lines connecting two components. Instead of handling each bit or line separately, we can group them together under one name, like a bundle of wires tied together. It keeps our design organized, the code more readable and easier to maintain. For this purpose, VHDL provides the **bit_vector type**, a collection of signals that **share a common name** and **can be accessed for reading or writing using an index**. Let's look at the syntax for declaring a signal of type bit_vector:

```
signal_name: bit_vector( index1 {to|downto} index2 );
```

The declaration defines a **composite signal** made up of $(\text{index2} - \text{index1} + 1)$ individual bits. In addition, the vector has an **ordering**, which is determined by the keyword **to** or **downto** used when specifying the index range. This ordering defines which bit is considered the **most significant bit** (MSB). When we use index1 to index2, the leftmost element (index1) is the least significant bit (LSB). Conversely, when we use index1 downto index2, the leftmost element (index1) is the most significant bit (MSB).

```
signal A : bit_vector(0 to 7);      -- A(0) is LSB, A(7) is MSB
signal B : bit_vector(7 downto 0); -- B(7) is MSB, B(0) is LSB
```

To access a specific element in the vector, write its index in parentheses:

```
A(5) -- the element 5 of vector A
```

Accessing an element beyond the vector's defined range constitutes an error.

7.4.2 Integer type

The integer type represents **numbers** using 32 bits and is useful when we need to describe counters, loop indices, or arithmetic operations that go beyond simple bits. However, we should keep in mind that we are describing hardware, not just software and we need to be careful. First, think about what those 32 bits actually mean: should they be treated as signed or unsigned? By default, they're unsigned, but if we want negative numbers, we must make that explicit, otherwise our hardware might not behave as you expect. Second, remember that hardware always has to physically implement whatever we describe. Declaring a signal as an integer means the synthesis tool sees a 32-bit bus, not just a small counter or value. So even if you just need to count from 0 to 7, the tool will build a 32-bit adder and a bank of 32 flip-flops, wasting area and power when only 3 bits would do!

7.4.3 IEEE types

The bit and bit_vector types have some limitations. In particular, they do not allow us to specify **don't care conditions** or **high-impedance states**. It's worth noting that handling high impedance is generally not part of logical synthesis but instead belongs to a lower level, specifically, the electrical level of the design. Nevertheless, it is often necessary to describe components that, under certain conditions, output not just a logical value but a high-impedance state, indicating that the component is temporarily disconnected from the elements it connects. VHDL does not include a type to support this behavior natively, however it provides a **standard IEEE library** that defines additional types which extend basic logic with a **9-value system**, as described below:

- '0' — Logical value 0
- '1' — Logical value 1
- 'Z' — High impedance

- 'X' — Unknown value (could be 0 or 1)
- 'U' — Uninitialized (no value has ever been assigned)
- 'W' — Weak unknown signal (cannot be clearly interpreted as 0 or 1)
- 'L' — Weak signal, interpreted as 0
- 'H' — Weak signal, interpreted as 1
- '-' — Don't care condition

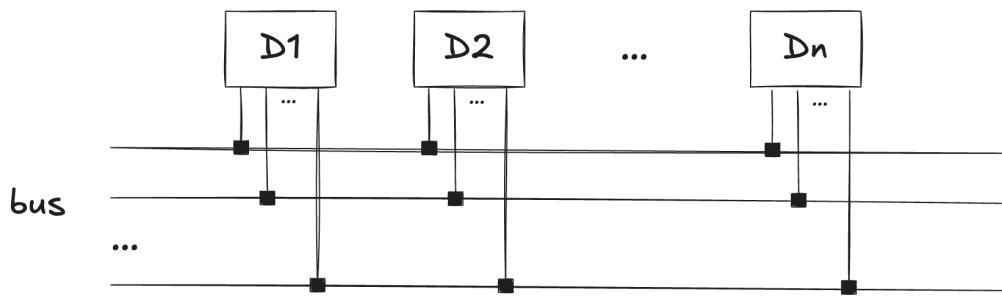
Among these values, the ones that are most relevant and commonly used (besides the logical 0 and 1) are the **high-impedance value 'Z'** and the **two don't care values, 'X' and '-'**. It's important to understand the difference between the two: 'X' is generated by the simulator whenever the actual value of a signal cannot be determined, while '-' must be explicitly assigned by the designer to indicate that a specific bit can take either 0 or 1 during optimization or minimization. In other words, 'X' reflects an unknown state during simulation, whereas '-' is a modeling convenience to express flexibility or indifference. This distinction, however, has no impact during synthesis, both are treated as "don't care" conditions.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

signal A : STD_LOGIC;
signal B : STD_LOGIC_VECTOR(7 downto 0);
```

7.4.4 Resolved types

In well-designed combinational logic, we should not directly connect multiple outputs to the same wire, because this causes contention and it is physically unsafe. However, in real hardware systems, we do sometimes allow multiple drivers on a shared bus. When this happens, we need a way to determine what the actual signal value should be if the drivers conflict. A **resolved type** uses a resolution function to combine these multiple drivers into a single final value according to predefined rules (for instance, 'H' overrides 'L', 'Z' means high impedance, and conflicts can result in 'X'). In contrast, unresolved types (like `bit`) cannot handle multiple drivers. If we connect two sources to a `bit` signal, we'll get an error because VHDL has no way to resolve the conflict on its own. Specifically, the `std_logic_1164 package` defines the `std_logic` and `std_logic_vector` types, which are resolved types commonly used in real designs. To clarify how a resolved type works in practice, let's look at a typical example: a bus connected to multiple devices:



Each line of the bus is therefore connected to multiple devices, which can potentially drive different logic values onto the same wire at the same time, a situation known as a **wired-OR** configuration. Although this setup **does not make sense from a pure logic design perspective**, it is not only valid from an electrical point of view but is actually widely used and sometimes even necessary. Suppose one module drives a logic 1 onto a bus line, while at the same moment another module drives a logic 0 onto that same line. What value will the line actually take? The answer depends on the electrical characteristics of the devices involved, the voltage levels used, whether a positive or negative logic convention is applied, and other physical factors that go beyond pure logical design. Because it is impossible to define the resulting value purely at the logic level, **VHDL prohibits multiple drivers on a signal by default, unless the type of the signal has an associated resolution function**. Resolved types make it possible to model this kind of scenario safely: they use defined rules, based on the nine-value logic system, to determine what value the signal will assume when multiple sources drive it at the same time.

7.4.5 User-defined types

A user-defined type is a **new type** whose characteristics must be explicitly specified by the designer. The designer is also responsible for defining any operators needed to work with signals of this new type. VHDL is a very flexible language and offers several mechanisms for creating custom types. However, for specifying systems of moderate or low complexity, it is usually sufficient to consider just two of these mechanisms.

Subtyping involves defining a new type that is equivalent to an existing type but with a **restricted range** of valid values. This mechanism is mainly useful for numeric types, especially the integer type. The general syntax is as follows:

```
subtype new_type_name is type_name range val1 to val2;
```

This definition means that the type "new_type_name" is equivalent to the existing type "type_name", except that it can represent only the values in the range from "val1" to "val2". In

practice, the most common and useful application of subtyping is for integers, where it helps ensure that the binary representation uses the **minimum number of bits** required. For example, if we want to represent an integer value in natural binary encoding using 5 bits, we can define a subtype of integer with bounds 0 to $2^5 - 1 = 31$, as shown here:

```
subtype small_integer is integer range 0 to 31;
```

Although this subtype definition effectively introduces a type that can be seen as a 5-bit vector, using signals of this subtype is generally not recommended compared to using signals of type std_logic_vector(0 to 4) or bit_vector(0 to 4) instead. The reason is that subtypes like this **rely on the synthesis tool** to correctly infer the minimum bit-width, **which may not always be obvious or consistent across different tools**. In contrast, explicitly declaring a std_logic_vector or bit_vector makes the intended hardware representation clear: we are directly describing a bus of 5 bits. This improves readability, portability, and synthesis efficiency.

The second mechanism for creating types involves defining a new type by **listing all the symbolic values it can assume**. This is known as an **enumerated type**. The syntax for such a definition is:

```
type new_type_name is ( val0, val1, ... , valN );
```

For example, in a simple combinational circuit, we might define an enumerated type to represent basic operations:

```
type operation is (AND_OP, OR_OP, XOR_OP);
```

This makes the logic more readable and self-explanatory when you select operations in your design.

7.5 Slice and Concatenate

Slicing and concatenation are essential operations that let us easily **extract parts of a vector** or **combine smaller signals into larger ones**, making it much simpler to manipulate buses and keep our combinational logic clear and modular.

7.5.1 Slice

A **slice** is a portion of a vector, that is, from a circuit perspective, it represents a **subset of the lines** within a composite signal. To specify a slice, we use the following syntax:

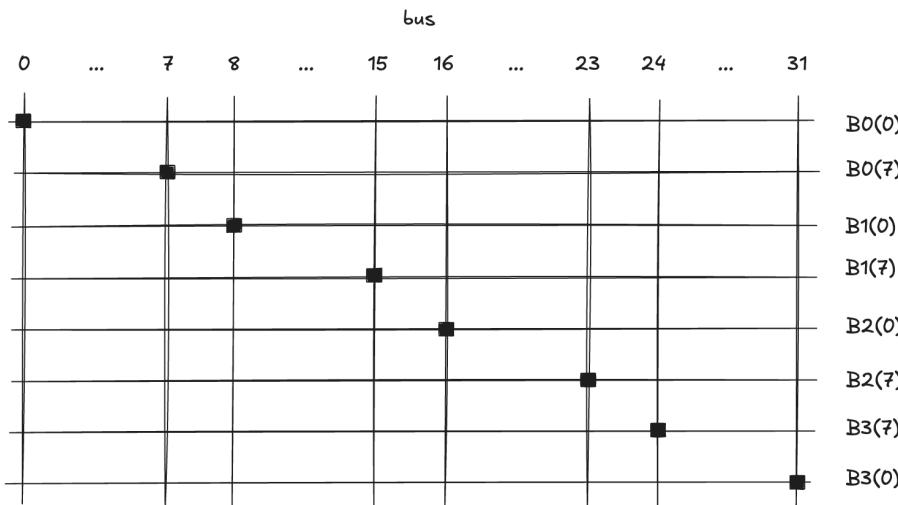
```
signal_name( index1 {to|downto} index2 )
```

where "index1" and "index2" must be valid indices for the vector "signal_name" and must respect the ordering defined by the to or downto clause.

The slice mechanism is often used to **extract certain lines from a bus under a single, more meaningful name**. A typical example is a bus that carries an entire word (32 bits) can be split into the individual bytes (8 bits each) that make it up:

```
architecture rtl of dummy is
    signal BUS: std_logic_vector(0 to 31);
    signal B0, B1, B2, B3: std_logic_vector(0 to 7);
begin
    ...
    B0 <= BUS(0 to 7);
    B1 <= BUS(8 to 15);
    B2 <= BUS(16 to 23);
    B3 <= BUS(31 downto 24);
    ...
end rtl;
```

This corresponds, in circuit terms, to the following situation:



Notice that the elements of vector B3 correspond to elements 24 to 31 of the BUS vector, taken in descending order, as specified by the downto clause.

7.5.2 Concatenate

The concatenation mechanism works in **the opposite way**: it allows us to **combine multiple signals** under a single name. The operator for concatenation is the ampersand (`&`). For example, if we want to reassemble a new bus `bus` from the four bytes of the previous architecture, but with their order swapped so that `B3` is the most significant byte and `B0` is the least significant, we can use an expression like the following:

```
BUS2 <= B3(7 downto 0) & B2 & B1 & B0;
```

In other cases, this operator is used **to group independent signals under a single name**, making it easier to perform faster comparisons. For example, suppose you need to check whether signal "a" equals 0, signal "b" equals 1, and signal "c" equals 0. There are two possible ways to do this. The first is simply to translate the condition directly into VHDL, like this:

```
if ( A = '1' and B = '0' and C = '1' ) then ...
```

This solution is acceptable and convenient if we only need to test this single condition involving the three variables. However, if we want to check multiple conditions on the same signals, it is better to use the concatenation mechanism. First, we declare a temporary signal with the appropriate size:

```
signal temp: std_logic_vector(0 to 2);
```

Then, we build the temporary vector by concatenating the original variables:

```
temp <= A & B & C;
```

And finally, we can write all the conditions in the compact form shown below:

```
if ( temp = "101" ) then ...
```

This style is more readable and concise than the previous approach, and it is often used in practice.

7.6 Logical Expressions and Truth Tables

Logical expressions can be translated into VHDL in a very straightforward way. First, remember that to correctly represent Boolean variables, we typically use the bit or std_logic types. The operators available for constructing logical expressions and their meanings are what you would expect:

Logical Operator	VHDL Operator	Example
AND	and	A and B
OR	or	A or B
NOT	not	not A
XOR	xor	A xor B

A logical expression describes how certain signals should be combined to produce other signals, and this is implemented using the **signal assignment operator** `<=`. Therefore, logical expressions belong to the RTL (Register Transfer Level) style of specification. For example, the logical expression:

$$F = A \bar{C} + D (\bar{A} + \bar{B} C)$$

can be translated with the following assignment:

```
F <= A and (not C) or D and ( (not A) or ((not B) and C) );
```

Some of the parentheses in this expression could be removed, since the evaluation follows the precedence and associativity rules defined for Boolean algebra operators. However, writing the expression with **explicit parentheses** don't change the meaning, but improve clarity and readability.

We can express truth tables very easily using logical expressions and **conditional assignments**, which allow us to define the value of a signal based on the values of other signals:

```
signal_name <= const_1 when cond_1 else
                     const_2 when cond_2 else
                     ...
                     const_N when cond_N else
                     const_default;
```

In this construct, the signal "signal_name" takes on the constant values "const_1", ..., "const_N", or "const_default" depending on whether the conditions "cond_1", ..., "cond_N" are true. Specifically, if none of the conditions is true, the signal will take on the default value const_default. For example, let's consider the following truth table:

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

It can therefore be read as follows: if "a" is 0 and "b" is 0, then "f" is 1; otherwise, if "a" is 0 and "b" is 1, then "f" is 0; and so on. This way of reading the truth table has a straightforward representation in VHDL:

```
F <= '1' when A='0' and B='0' else
      '0' when A='0' and B='1' else
      '1' when A='1' and B='0' else
      '1' when A='1' and B='1';
```

It's important to note that conditional assignment expressions used to represent purely combinational circuits **must explicitly cover all possible cases**. In the previous example, if the variables A and B are of type bit, the assignment is valid, because the bit type only allows the values 0 and 1, so all possible input combinations are covered. In contrast, if A and B were of type std_logic, the assignment would not be valid because std_logic can take on nine possible values. Covering all possible cases would then mean writing out all 81 input combinations, which is impractical. A common solution is to **explicitly specify the main conditions** and then **provide a default value** for the fourth condition to cover any cases not explicitly listed:

```
F <= '1' when A='0' and B='0' else
      '0' when A='0' and B='1' else
      '1' when A='1' and B='0' else
      '1';
```

By following this strategy, it's also possible to simplify the way we write a truth table by **considering only the on-set or the off-set**. In this example, we could write an assignment like the following:

```
F <= '0' when A='0' and B='1' else
      '1';
```

However, this approach is **not always the best solution in terms of clarity**, and it is **only applicable to completely specified functions**, that is, functions for which the don't care set is empty. Let's now consider a truth table for a function that is not fully specified:

A	B	F
0	0	1
0	1	0
1	0	-
1	1	1

In order for the synthesis tool to find the optimal solution for such a function, it is necessary to express the don't care conditions present in the specification. As a first approach, we could explicitly define all the values that the function takes for every possible combination of input values. It's important to note that in this case, you must represent the concept of indifference (don't care), which is not supported by the bit type. Therefore, you need to use signals of type std_logic. One possible solution is as follows:

```
F <= '1' when A='0' and B='0' else
      '0' when A='0' and B='1' else
      '-' when A='1' and B='0' else
      '1' when A='1' and B='1';
```

However, this solution still suffers from the same issue mentioned earlier: it does not explicitly cover all possible cases. A better approach is the following:

```
F <= '1' when A='0' and B='0' else
      '0' when A='0' and B='1' else
      '-' when A='1' and B='0' else
      '1';
```

Finally, if we want to specify the truth table in the most compact form possible, we can use a representation where only two of the three sets (the on-set, off-set, and don't care set) are

explicitly defined. For example, if we choose to specify the off-set and the don't care set, we would write:

```
F <= '0' when A='0' and B='1' else
      '-' when A='1' and B='0' else
      '1';
```

VHDL provides an alternative construct for conditional assignment that allows us to write more concise comparison expressions by using the concatenation operator. Its general syntax is as follows:

```
with signal_test select
  signal_name <= const_1 when case_1,
                const_2 when case_2,
                ...
                const_N when case_N,
                const_default when others;
```

If "signal_test" matches one of the constants "case_1", ..., "case_N", then "signal_name" is assigned the corresponding value "const_1", ..., "const_N". If "signal_test" does not match any of the specified cases, then "signal_name" is assigned the value "const_default". It is clear that the semantics of this construct are equivalent to those of the conditional assignment we just discussed. However, as the following example shows, this form is often more concise and easier to read:

```
architecture rtl of dummy is
  signal temp: std_logic_vector(0 to 1);

begin
  ...
  temp <= A & B;
  ...
  with temp select
    F <= '1' when "00",
      '0' when "01",
      '-' when "10",
      '1' when "11",
      '-' when others;
```

It is clear that this form provides an intuitive way to represent a truth table. It's also worth emphasizing the importance of the final clause in this construct: first, it is necessary because we

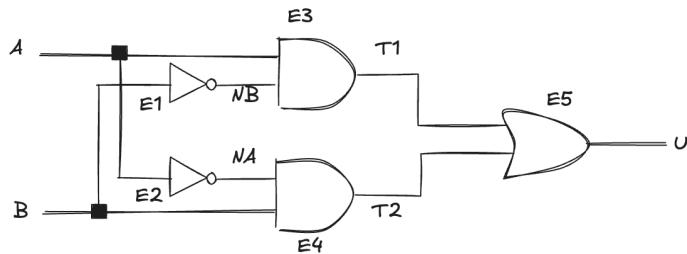
are working with a nine-valued logic signal, so many cases are not explicitly listed. Second, it is good practice not to combine the don't care set with this clausem mainly for clarity.

7.7 Structural Description

What we have covered so far is sufficient for specifying modules of low or moderate complexity, but it is not well-suited for designing highly complex circuits. As in many other areas of science and engineering, a good approach to solving a complex problem is to **break it down into simpler subproblems** and then combine their solutions. In design terms, this means that the development of a very complex circuit should begin by **identifying the system's basic functions, designing components that implement these functions, and then interconnecting these modules appropriately to build the complete system**. To clarify this concept, let's start with a simple example: suppose we want to implement an XOR gate using only the basic gates OR, AND, and NOT. We know that:

$$A \oplus B = (A \bar{B}) + (\bar{A} B)$$

In circuit terms, this corresponds to the following structure:



7.7.1 Component description

To implement the XOR gate starting from the basic components, we first need to create those components. Let's look at the required entities and architectures:

```

-- The NOT gate
entity NOT_GATE is
    port( X: in std_logic;
          Z: out std_logic );
end NOT_GATE;

architecture rtl of NOT_GATE is
begin

```

```

    Z <= not X;
end rtl;

-- The 2-input AND gate
entity AND2_GATE is
  port( X: in std_logic;
        Y: in std_logic;
        Z: out std_logic );
end AND2_GATE;

architecture rtl of AND2_GATE is
begin
  Z <= X and Y;
end rtl;

```

```

-- The 2-input OR gate
entity OR2_GATE is
  port( X: in std_logic;
        Y: in std_logic;
        Z: out std_logic );
end OR2_GATE;

architecture rtl of OR2_GATE is
begin
  Z <= X or Y;
end rtl;

```

7.7.2 Component instantiation

Now that we have the three basic building blocks, we can solve the problem **by connecting these components appropriately**, that is, following the circuit diagram shown earlier. We start by defining the entity for the XOR gate, which will have two inputs and one output:

```

entity XOR2_GATE is
  port( A: in std_logic;
        B: in std_logic;
        U: out std_logic );
end XOR2_GATE;

```

It's important to point out, to avoid any misunderstanding, that there is a fundamental difference between the VHDL and operator "and" the "AND2_GATE" component we just defined. The operator is built into the language and **can only be used inside expressions**, whereas AND2_GATE is a generic component that **cannot be used in an expression, it must be instantiated**, which means inserting it into the design specification (and therefore into the circuit being built) and connecting it properly to other components using signals. Considering this point, the declarative part of the architecture **must specify the names and types of any components it uses**. The declaration of a component generally follows this structure:

```
component component_name is
    [generic( generic_list );]
    port( port_list );
end component;
```

Essentially, a component declaration is identical to its corresponding entity declaration, except that it uses the keyword component instead of entity. At this point, we can start looking at the code for the architecture of our xor module, beginning with its declarative section:

```
architecture structural of XOR2_GATE is

    -- The NOT gate
    component NOT_GATE is
        port( X: in std_logic;
              Z: out std_logic );
    end component;

    -- The 2-input AND gate
    component AND2_GATE is
        port( X: in std_logic;
              Y: in std_logic;
              Z: out std_logic );
    end component;

    -- The 2-input OR gate
    component OR2_GATE is
        port( X: in std_logic;
              Y: in std_logic;
              Z: out std_logic );
    end component;

    -- Internal signals
```

```

signal NA: std_logic;
signal NB: std_logic;
signal T1: std_logic;
signal T2: std_logic;

begin
    ...
end structural;
```

At this point, we have declared everything needed to proceed with building the circuit. To do this, we need to **instantiate the components**. An instantiation has the following general form:

```

instance_name: component_name
    [generic map( generic_assignment_list );]
    port map( port_assignment_list );
```

Each time we use a component, we must **assign it a name**. This name, called the **instance name**, must be **unique within the architecture** and has no connection to the name of the component being instantiated. The name of the component for which "instance_name" is an instance is "component_name", and it must have been declared beforehand in the declarative section of the architecture. For now, we can set aside the optional part of the instantiation construct that deals with generics and focus on the **port map construct**. It specifies **how the ports of the instance are connected to the signals in the architecture**, whether these signals are inputs, outputs, or internal signals. The **port_assignment_list** is a list of these connections and can be written in two forms: positional or named. The positional form has the following syntax:

```
port map( signal_1, ... , signal_N );
```

In this case, **the order of the signals is significant**: the first signal in the list will be connected to the first port listed in the component declaration, the second signal to the second port, and so on. While this form is certainly very concise, it can sometimes be harder to read. The second option is to use the named form, which has the following syntax:

```
port map( port_1 => signal_1, ... , port_N => signal_N );
```

In this form, **the names of the ports of the component being instantiated are explicitly listed**. Thanks to this, the association (the connection of signals) is done by name, so the position within the list no longer matters and can therefore be arbitrary. To clarify the concept of instantiation, let's look in detail at the instantiation of the E3 AND gate using a named port map:

```
E3: AND2_GATE
    port map( X => A, Y => NB, Z => T1 );
```

This means that port X of the AND2_GATE component will be connected to signal A, port Y will be connected to signal NB, and port Z will be connected to signal T1. Notice that the direction of the input and output signals must be respected. This means that an input signal of the architecture (for example, A) cannot be connected to an output port of a component, doing so would imply writing to an input signal, which VHDL prohibits. Likewise, an output signal of the architecture (for example, U) cannot be connected to an input port of a component. However, the arrow operator here does not indicate the direction of data flow; it simply indicates the connection between the port and the signal. The actual data flow is determined by the direction specified in the component declaration.

```
...
begin
    E1: NOT_GATE
        port map( X => B, Z => NB );
    E2: NOT_GATE
        port map( X => A, Z => NA );
    E3: AND2_GATE
        port map( X => A, Y => NB, Z => T1 );
    E4: AND2_GATE
        port map( X => NA, y => b, z => T2 );
    E5: OR2_GATE
        port map( X => T1, Y => T2, Z => U );
end structural;
```

This style of writing VHDL is called **structural VHDL**, because it focuses on describing the structure of the network rather than the transformations that the signals undergo. In practical design, the use of structural VHDL is typically limited to connecting complex components, it is very rare to use it for describing individual elements like basic logic gates or flip-flops. It is also common to use both structural and RTL styles within the same architecture, in order to take advantage of the strengths of each. The resulting VHDL is therefore neither purely structural nor purely RTL, but rather a combination of the two.

7.8 Process

A **process** is a concurrent statement, just like a simple signal assignment, a conditional assignment, or a component instantiation. However, a process is a **compound statement**: it groups

together a set of sequential statements that **execute in order**, but the process itself is evaluated concurrently with other concurrent statements in the architecture. The general syntax of a process is:

```
[process_name]: process ( sensitivity_list )
    [declarations]
begin
    [body]
end process;
```

A process can have an optional name (which helps improve code readability and clarity), an optional declarative region (similar to an architecture's declarative section), a statement body, and a **sensitivity list**, which is a list of signals that can **trigger its execution**. The process is evaluated only when an "event" occurs on one or more signals in the sensitivity list; otherwise, it remains inactive. To clarify this concept, let's consider a general concurrent statement such as:

$$Z \leq (X + Y) T$$

This assignment will automatically update Z whenever there is a change (an **event**) on any of the signals X, Y, or T. In other words, the hardware inferred from this statement continuously monitors these signals in parallel. Likewise, if we want to describe the same logic inside a process, we need to make sure the process knows which signals should trigger its execution. This is done by explicitly listing X, Y, and T in the sensitivity list, so the process "wakes up" whenever any of these signals change. For example:

```
sample: process(X, Y, T)
begin
    Z <= (X or Y) and T;
end process;
```

The **sequential constructs** that can be used inside the body of a process are the same as those found in most programming languages: expressions and assignments, conditional statements, and loops. However, loops require care, as they can lead to issues when synthesizing the design into hardware. In fact, extensive use of loops is typical of a behavioral description style, which goes beyond the scope of our discussion.

7.8.1 If-then-else statement

The basic **conditional construct** is the **if statement**. It works just like in most programming languages: it lets us choose between different actions based on certain conditions being true. In

its most general form, the syntax is:

```
if condition_1 then
    statement_1
[elsif condition_2 then
    statement_2]
...
[else
    statement_N]
end if;
```

As we can see, both the elsif branch and the else branch are optional, and we can have as many elsif branches as needed. It's important to highlight that **all possible cases must be explicitly handled**, as discussed for conditional assignments. In practice, this often means we should include an else branch to cover any remaining conditions. Therefore, the correct minimal form of this construct should be written as follows:

```
if condition then
    statement_then
else
    statement_else
end if;
```

The meaning is exactly what we would expect from most programming languages: if the condition is true, the statement_then block is executed; if the condition is false, the statement_else block is executed instead. Let's look at a simple example. Suppose we want to implement a circuit with three inputs, A, B and S. The output U should be equal to A if S is 0, and equal to B if S is 1 (this is called a **multiplexer**, as we will see later):

```
entity mux is
    port( A: in std_logic;
          B: in std_logic;
          S: in std_logic;
          U: out std_logic );
end mux;
```

The architecture of this circuit can be written using a conditional statement as follows:

```
architecture rtl_concurrent of mux is
begin
    U <= A when s='0' else B;
end rtl_concurrent;
```

If we want to implement the same circuit using a process and sequential statements, we would proceed as follows. First, it's necessary to identify which signals the process should be sensitive to: in this case, any change to A, B, or S should produce an immediate update on the output. Therefore, all three signals must appear in the process's sensitivity list. The condition itself is very simple, it just checks whether S is equal to a specific value:

```
architecture rtl_sequential of mux is
begin
    select: process( A, B, S )
    begin
        if( S = '0' ) then
            U <= A;
        else
            U <= B;
        end process;
    end rtl_sequential;
```

In this case, the sequential is less concise than the concurrent one, however using processes provides greater flexibility, especially when the logic involves complex conditions, nested if statements, or case statements, as we will see considering more complex examples later.

7.8.2 Case statement

Very often, you need to compare a signal against a series of constant values and then perform different actions depending on the result of that comparison. Of course, we can implement this kind of logic using an if-then-elsif-else construct, expressing each condition in a separate branch. However, a deeply nested or overly complex series of if statements quickly becomes hard to read and maintain. To address these issues, VHDL provides the case statement. It is the sequential equivalent of the concurrent with-select construct but offers greater flexibility when describing multiple mutually exclusive choices. The general syntax of the case statement is:

```
case signal_name is
    when range_1 =>
```

```

    statement_1
when range_2 =>
    statement_2
    ...
when range_N =>
    statement_N
when others =>
    statement_default
end case;

```

In this construct, the statement checks the value of signal_name and executes exactly one branch that matches its value; if no explicit match is found, the others clause provides a default action to ensure all possible cases are covered.

7.8.3 For statement

The statement is used to define **loops**. The general syntax is:

```

for identifier in range loop
    statements
end loop;

```

The identifier is the loop's control variable, which takes on each value within the specified range during the loop iterations. It's important to note that **the identifier is not a signal**, so it cannot be used in any construct that requires a signal as an operand. The loop range (the range over which the index varies) **must be constant**, its bounds must be known at synthesis time. The range can take one of the following three forms:

```

first to last
last downto first
array_name'range

```

The first two forms have the same meaning as when specifying the dimensions of an array or describing a slice. The third form uses the **range attribute** of VHDL signals. If array_name is a vector signal, then applying the range attribute returns the index range of that vector. Let's look at a simple example of how to use the for loop to assign the value of a 16-bit vector signal B to another 16-bit vector signal A, copying it bit by bit:

```

signal A, B: std_logic_vector(0 to 15);

...

for I in 0 to 15 loop
    A(I) <= B(I);
end loop;

```

In general, to use loops and still generate a synthesizable description, the values of any expressions involving the loop index must be determinable at synthesis time, either because they are constants, or because they are derived from generic parameters whose values must be set before synthesis begins. Additionally, the statements inside the body of the loop must not modify the value of the index. In other words, the loop variable can only be read, never assigned to or changed within the loop.

7.8.4 While statement

The while loop repeats a block of statements as long as a given condition remains true:

```

while condition loop
    statements
end loop;

```

In synthesizable design, while loops are rarely used, because the number of iterations must be determinable at compile-time for synthesis tools to map it to hardware. However, they can be useful in testing scenarios, where the number of iterations may depend on runtime conditions, as we see in the next chapter.

7.9 Testbench

A **testbench** is a special module used to verify the behavior of another module, known as the **device under test (DUT)**. It generates and applies input signals to the DUT and observes the outputs to ensure they match the **expected results**. The sets of input values and their corresponding expected outputs are called **test vectors**. Consider testing the following module, which computes:

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

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sillyfunction is
    port(A, B, C: in STD_LOGIC;
         Y: out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
    y <= (not A and not B and not C) or
          (A and not B and not C) or
          (A and not B and C);
end;

```

Since this is a simple module, we can use **exhaustive testing**, meaning we apply all eight possible input combinations to fully verify its behavior:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity testbench is
end;

architecture sim of testbench is
    component sillyfunction
        port(A, B, C: in STD_LOGIC;
             Y: out STD_LOGIC);
    end component;

    signal A, B, C, Y: STD_LOGIC;

begin
    -- instantiate device under test
    DUT: sillyfunction port map(B, B, C, Y);

    -- apply inputs one at a time
    -- checking results
    process begin
        A <= '0'; B <= '0'; C <= '0'; wait for 10 ns;
        assert Y = '1' report "000 failed.";

```

```

C <= '1'; wait for 10 ns;
assert Y = '0' report "001 failed.";

B <= '1'; C <= '0'; wait for 10 ns;
assert Y = '0' report "010 failed.";

C <= '1'; wait for 10 ns;
assert Y = '0' report "011 failed.";

A <= '1'; B <= '0'; C <= '0'; wait for 10 ns;
assert Y = '1' report "100 failed.";

C <= '1'; wait for 10 ns;
assert Y = '1' report "101 failed.";

B <= '1'; C <= '0'; wait for 10 ns;
assert Y = '0' report "110 failed.";

C <= '1'; wait for 10 ns;
assert Y = '0' report "111 failed.";

wait;
end process;
end;

```

The test code instantiates the DUT and then applies the input stimuli. The **wait statement** is used to pause the execution of the process. When the specified time elapses (or when a given condition becomes true), the process resumes execution from the point immediately after the wait. This mechanism allows us to apply inputs in the correct sequence and at precise time intervals. After each assignment, we check whether the outputs match the expected values. The **assert statement** checks a condition and prints the message given in the report clause if the condition is not satisfied. Testbenches are simulated in the same way as other HDL modules; however, they are **not synthesizable** and are used purely for verification purposes.

However, writing code for each test vector becomes tedious, especially for modules that require a large number of vectors. A better approach is to **place the test vectors in a separate file**. The testbench simply reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vector, and repeats until reaching the end of the test vectors file. For example, we can have the following vectors.txt file:

```
0001  
0010  
0100  
0110  
1001  
1011  
1100  
1110
```

Then the testbench code can be implemented as follows, using the **TEXTIO** and **STD_LOGIC_TEXTIO** packages to open a file, read one line at a time into a variable of type "line" and then parse the line into individual signal values or vectors. Of course the file I/O is not synthesizable, it's for simulation only.

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_TEXTIO.ALL;  
use STD.TEXTIO.ALL;  
  
entity test_sillyfunction is  
end;  
  
architecture sim of test_sillyfunction is  
    component sillyfunction  
        port (  
            A, B, C : in STD_LOGIC;  
            Y         : out STD_LOGIC  
        );  
    end component;  
  
    signal A, B, C, Y : STD_LOGIC;  
  
begin  
  
    -- Instantiate the DUT  
    DUT: sillyfunction port map(A, B, C, Y);  
  
    -- Apply test vectors from file using while loop  
    process  
        file vec_file : TEXT open READ_MODE is "vectors.txt";  
        variable vec_line : LINE;  
        variable vec_A, vec_B, vec_C, vec_Y : CHARACTER;
```

```

variable expected : STD_LOGIC;
begin
  while not endfile(vec_file) loop
    readline(vec_file, vec_line);
    read(vec_line, vec_A);
    read(vec_line, vec_B);
    read(vec_line, vec_C);
    read(vec_line, vec_Y);

    A <= vec_A;
    B <= vec_B;
    C <= vec_C;

    wait for 10 ns;

    expected := vec_Y;

    assert Y = expected
      report "Test failed for input: " &
        vec_A & vec_B & vec_C &
        ". Expected: " & vec_Y &
        ", Got: " & STD_LOGIC'IMAGE(Y)
      severity ERROR;
  end loop;

  wait; -- Wait forever after all vectors are done
end process;

end sim;

```

This testbench is certainly more elaborate than needed for such a simple circuit. However, its structure makes it easy to adapt for more complex designs: we only need to update the example.txt file with new test vectors, instantiate the new DUT, and adjust a few lines of code to apply the inputs and check the outputs.

7.10 Simulation and Synthesis

Simulation and **synthesis** represent two distinct but complementary phases of the design flow. While simulation verifies the **functional correctness** of the HDL code, synthesis translates the **behavioral description** into a **hardware implementation** mapped onto a specific target technology.

7.10.1 Simulation

Simulation is a **software-based verification step**. It checks whether the written HDL model behaves as intended before any hardware is generated. Simulation tools execute the design code and its testbench using a virtual time model, allowing designers to observe signal transitions, test edge cases, and verify timing relationships at a functional level. In the **Xilinx toolchain**, simulation can be performed using the **Vivado Simulator (XSIM)**, which supports both:

- **Pre-synthesis (behavioral)** simulation uses only the HDL description and ignores physical delays.
- **Post-synthesis (netlist)** simulation includes the synthesized logic structure and propagation delays derived from the target FPGA.

7.10.2 Synthesis

Synthesis is the process that converts a **hardware description** into a **structural representation** made of logic gates and registers that can be physically implemented on a real device. Synthesis bridges the gap between **behavioral description** ("what the circuit must do") and **physical realization** ("how the circuit is built"). The synthesizer analyzes the HDL code, applies Boolean simplifications, and produces an optimized **netlist**, a list of interconnected logic elements available in the chosen **technology library**. Depending on the target platform, synthesis can follow two major paths: **custom (ASIC)** or **FPGA**.

In **Application-Specific Integrated Circuit (ASIC)** design, synthesis is the first step of a fixed, non-reconfigurable hardware implementation. The designer targets a **specific semiconductor process** (for example, 130 nm or 65 nm CMOS) using a **standard-cell library** provided by the foundry. This library contains all the basic logic building blocks (gates (NAND, NOR, XOR, INV), flip-flops, latches, and buffers) each characterized by area, delay, and power consumption parameters. ASIC synthesis tools, such as **Synopsys Design Compiler**, **Cadence Genus**, or the open-source **Yosys**, translate the HDL code into a **gate-level netlist** built exclusively from those standard cells. The output of ASIC synthesis is a **technology-dependent gate-level netlist** and timing/power reports. This netlist is later refined by **place-and-route** tools to produce the **physical layout** and finally the **GDSII mask data** used to fabricate the chip. ASIC synthesis therefore creates a **permanent, high-performance, and power-efficient** circuit, but one that cannot be modified once manufactured.

In contrast, **Field-Programmable Gate Arrays (FPGAs)** are **reconfigurable devices**, consisting of programmable logic blocks, interconnects, and specialized resources such as DSP slices and memory blocks. Synthesis for an FPGA therefore targets a **vendor-specific architecture**, mapping the HDL design onto the primitives available on that device. For example, **Xilinx Vivado**

is the official synthesis and implementation tool for Xilinx FPGAs. When synthesizing a design for a specific FPGA (like the **Artix-7 FPGA** (`xc7a200tfg484-1`), Vivado translates behavioral constructs into logical elements (e.g., adders, multiplexers, state machines) and converts these elements into FPGA primitives. The result is a **technology-specific FPGA netlist**. After implementation, Vivado produces a **bitstream** (`.bit`) that configures the device with the synthesized logic. Unlike ASIC synthesis, FPGA synthesis results in a **reconfigurable design** that can be reprogrammed any number of times.

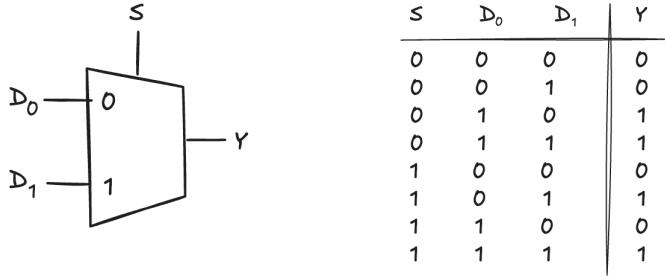
Aspect	ASIC Synthesis	FPGA Synthesis
Target device	Fixed silicon (custom chip)	Reconfigurable FPGA fabric
Technology library	Standard-cell gates (NAND, NOR, INV, DFF, etc.)	Xilinx primitives (LUTs, FFs, BRAMs, DSPs)
Tool examples	Synopsys Design Compiler, Cadence Genus, Yosys	Xilinx Vivado
Output	Gate-level netlist	FPGA netlist + bitstream.
Optimization goal	Area, power, performance	Timing closure, resource utilization
Flexibility	Fixed after fabrication	Fully reprogrammable
Cost and NRE	High (mask cost, fabrication)	Low (single reconfigurable device)

8 Combinational Building Blocks

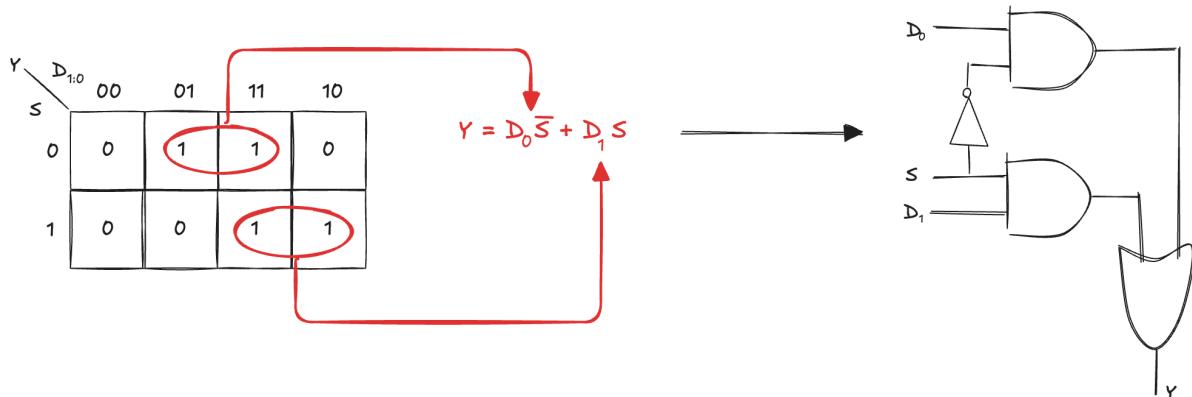
Combinational logic circuits can be grouped into **larger building blocks** to make it easier to design complex systems. By using **abstraction**, we hide the low-level gate details and focus instead on what each block does. **Hierarchy** lets us build big systems by connecting simpler parts together, layer by layer. Thanks to **modularity** and **regularity**, each block has clear, well-defined interface, so we can think of it as a black box and reuse it easily. For example, common building blocks include multiplexers, decoders, arithmetic circuits, and seven-segment display decoders. These blocks perform useful functions that show up repeatedly in digital designs. As we go forward, we'll see how combining these simple blocks helps us build much more powerful systems, like a microprocessor, without getting lost in the details of every single gate.

8.1 Multiplexers

Multiplexers are among the most commonly used combinational circuits. They **choose an output from among several possible inputs, based on the value of a select signal**. A multiplexer is sometimes called a **mux**. The following figure shows the schematic symbol and truth table for a 2:1 mux:



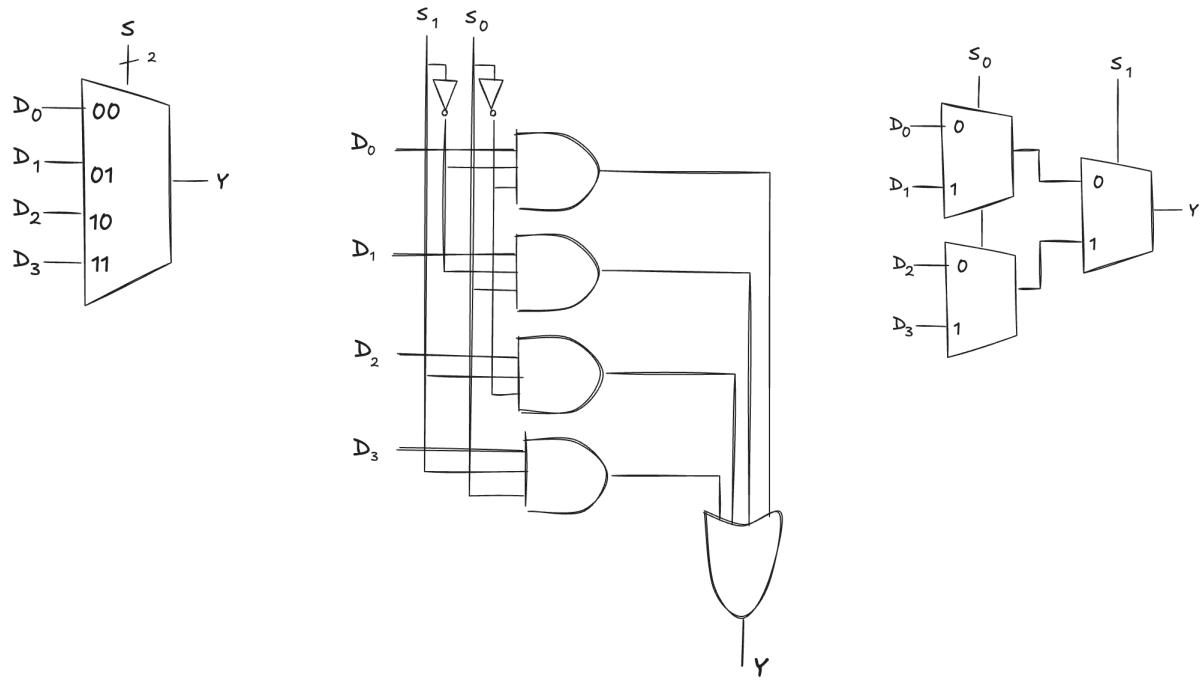
The multiplexer chooses between the two data inputs, based on the select value, which is also called a **control signal** because it controls what the multiplexer does. We can build it from sum-of-products logic and derive its Boolean equation:



$$\text{mux}(D_0, D_1, S) = D_0 \bar{S} + D_1 S$$

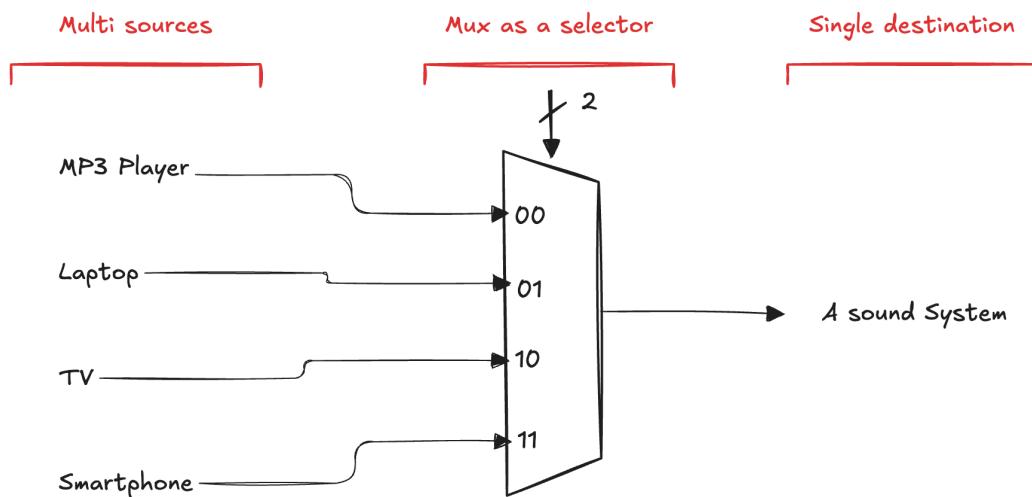
8.1.1 Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output. In that case, two select signals are needed to choose among the four data inputs. It can be built using sum-of-products logic or multiple 2:1 multiplexers:



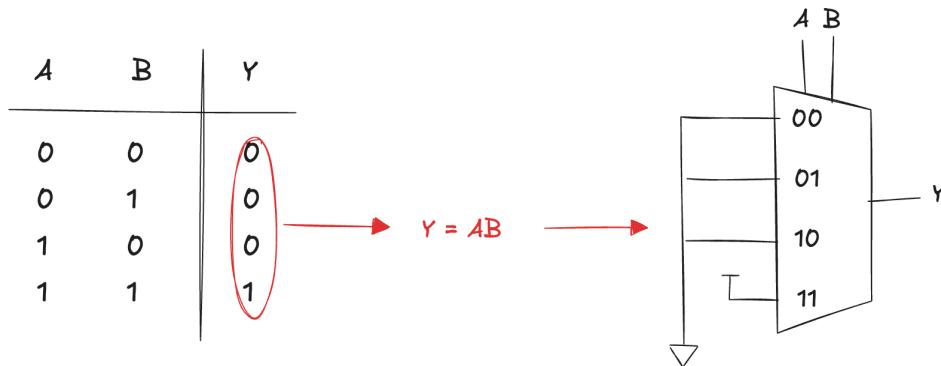
Wider multiplexers, such as 8:1 and 16:1 multiplexers, can be built by expanding the methods. In general, an $N:1$ multiplexer needs $\log_2 N$ select lines.

We can think of mux as a **switchboard** that connects only one input at a time, **allowing different data sources to share the same output line efficiently**. This makes it possible to handle many inputs in a controlled, sequential way without needing separate circuits for each one:



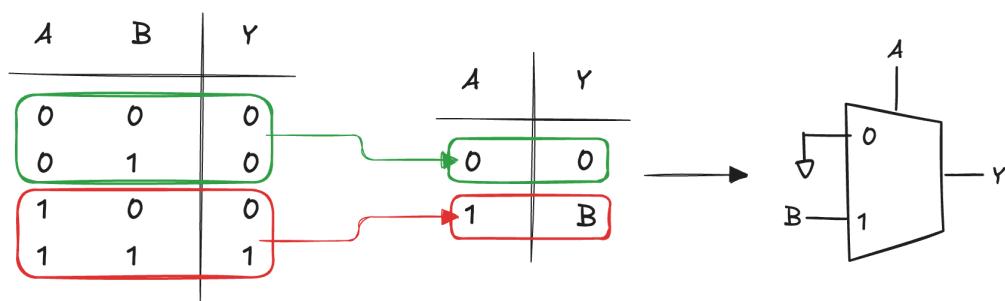
8.1.2 Multiplexer Logic

Multiplexers aren't just for routing data, they can also serve as **lookup tables** for logic operations. For instance, the figure below illustrates how we can use a 4:1 multiplexer to implement the behavior of a two-input AND gate by selecting the right output for each input combination:



The **select lines are used as input** for the multiplexer. The **inputs lines are wired to either 0 or 1, based on the rows of the truth table** for the logic function we want to implement. In general, a multiplexer with 2^N inputs can be configured to perform any logic function of N variables. By simply changing how the data inputs are connected, the **multiplexer can be reprogrammed** to carry out a completely different function.

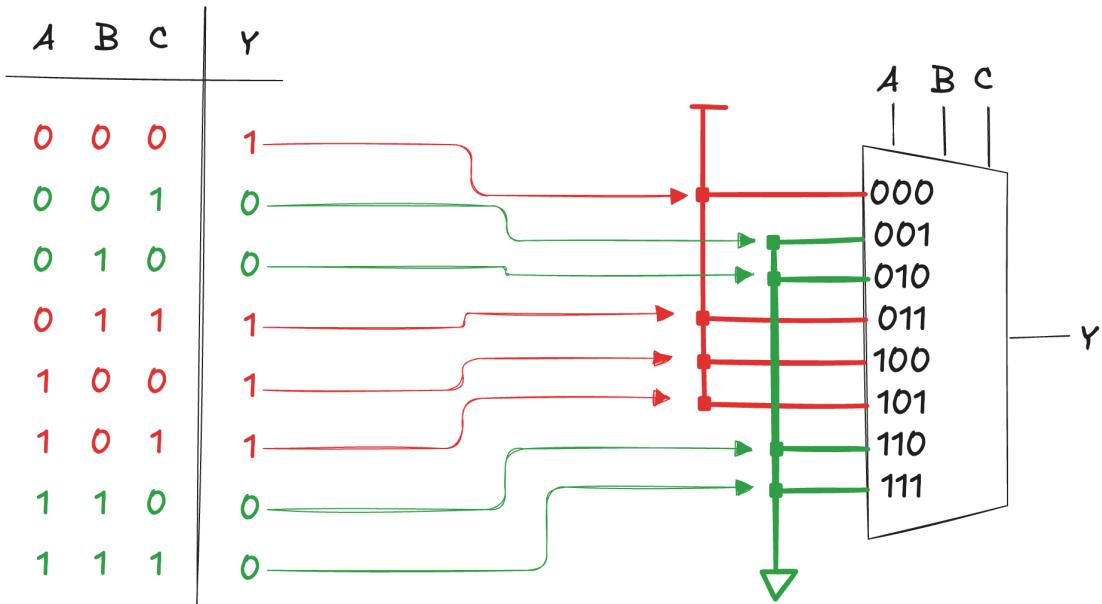
It's possible to reduce the size of the multiplexer by half and still implement any N input logic function using only a $2^{(N-1)}$ input multiplexer. The key idea is to provide one of the input variables directly to the multiplexer's data inputs instead of hard-wiring them to 0 or 1. To see how this works in practice, we can implement a two-input AND gate using a simple 2:1 multiplexer.



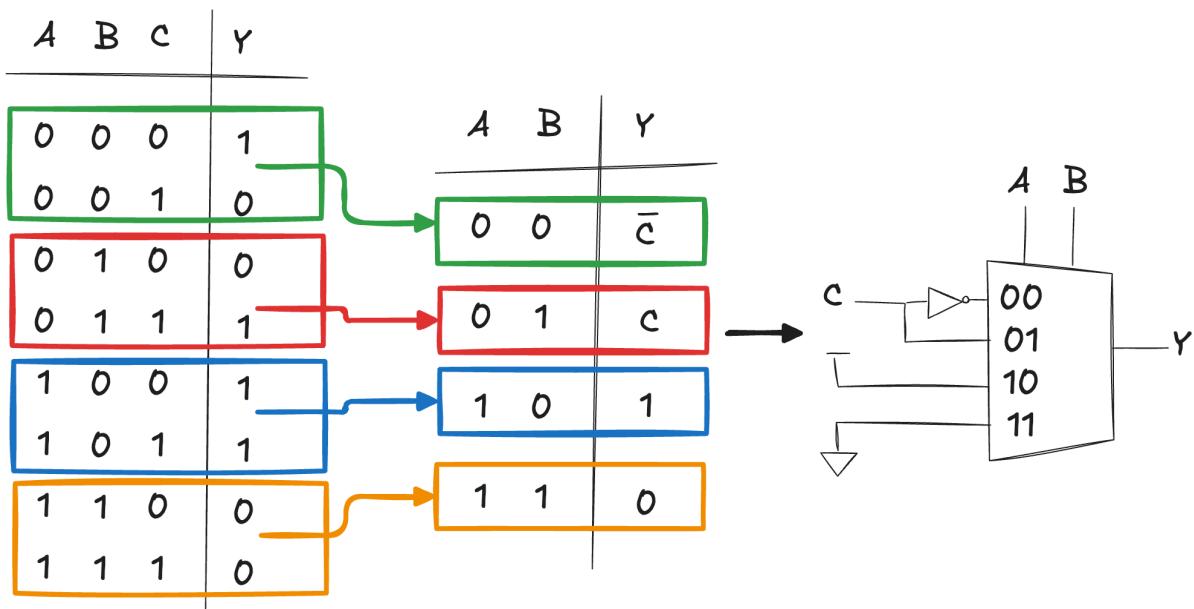
We begin with the complete truth table for the logic function, then combine pairs of rows to eliminate the rightmost input variable. This allows us to rewrite the output in terms of this variable, reducing the number of required data inputs. The multiplexer is then used as a lookup table based on this reduced truth table. To illustrate this approach further, let's consider a more complex example:

$$Y = A \bar{B} + \bar{B} \bar{C} + \bar{A} B C$$

Suppose to have an 8:1 multiplexer for the implementation. How can this function be realized:



Now suppose that the original 8:1 multiplexer is damaged and only a 4:1 multiplexer and an inverter are available as replacement components. Is it possible to implement the function using only these parts?



Vediamo ora la descrizione VHDL di un multiplexer 4:1.

```

entity mux_4_1 is
    port( s0, s1: in std_logic;
          a, b, c, d: in std_logic;
          y: out std_logic
    );
end mux_4_1;

architecture rtl of mux_4_1 is
begin
    y <= a when s0='0' and s1='0' else
              b when s0='0' and s1='1' else
              c when s0='1' and s1='0' else
              d when s0='1' and s1='1' else
              '-';
end rtl;

```

Il costrutto "with ... select" offre la possibilità di rendere la specifica più sintetica e leggibile:

```

architecture rtl of mux_4_1 is
    signal sel: std_logic_vector(0 to 1);
begin
    with sel select
        y <= a when "00",
                  b when "01",
                  c when "10",
                  d when "11",
                  '-' when others;
end rtl;

```

Supponiamo ora che i segnali di ingresso, e di conseguenza il segnale di uscita, non siano di un bit bensì siano segnali a 8 bit. La nuova entity per il multiplexer in esame diventa la seguente:

```

entity mux_4_1_8bit is
    port( s0, s1: in std_logic;
          a, b, c, d: in std_logic_vector(0 to 7);
          y: out std_logic_vector(0 to 7)
    );
end mux_4_1_8bit;

architecture rtl of mux_4_1_8bit is
    signal sel: std_logic_vector(0 to 1);

```

```

begin
  with sel select
    y <= a when "00",
      b when "01",
      c when "10",
      d when "11",
      "-----" when others;
end rtl;

```

Vediamo ora, a titolo di esempio, come è possibile specificare un multiplexer 4-a-1 generico per segnali di dimensione variabile, grazie alla parametrizzazione che si realizza mediante il costrutto generic:

```

entity mux_4_1_Nbit is
  generic( N: integer );
  port( sel: in std_logic_vector(0 to 1);
        a, b, c, d: in std_logic_vector(0 to N-1);
        y: out std_logic_vector(0 to N-1)
      );
end mux_4_1_Nbit;

```

La dichiarazione di generic introduce nell'entity un parametro, N, che indica la dimensione dei segnali di ingresso e del segnale di uscita. Quando questo componente sarà utilizzato nel progetto, sarà necessario assegnare un valore specifico e costante al parametro N. Ciò fatto, lo strumento di sintesi sarà in grado di realizzare il componente in modo consistente con la dimensione assegnata ai segnali. Consideriamo ora l'architectura, l'unica parte che mostra una dipendenza esplicita dalla dimensione dei segnali è l'assegnamento del valore costante costituito da tutti don't care. Non conoscendo a priori la dimensione del segnale di uscita, non è possibile utilizzare una costante predefinita per tale assegnamento. A tal fine il VHDL dispone di un costrutto che permette di assegnare un valore costante ad ogni elemento di un vettore, sia quando la dimensione è nota a priori, sia quando non lo è:

```

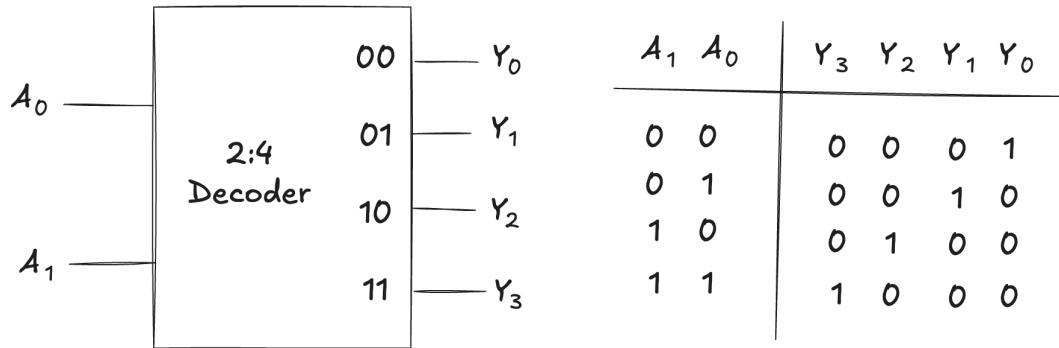
architecture rtl of mux_4_1_Nbit is
begin
  with sel select
    y <= a when "00",
      b when "01",
      c when "10",
      d when "11",

```

```
(others  $\Rightarrow$  '-'') when others;
end rtl;
```

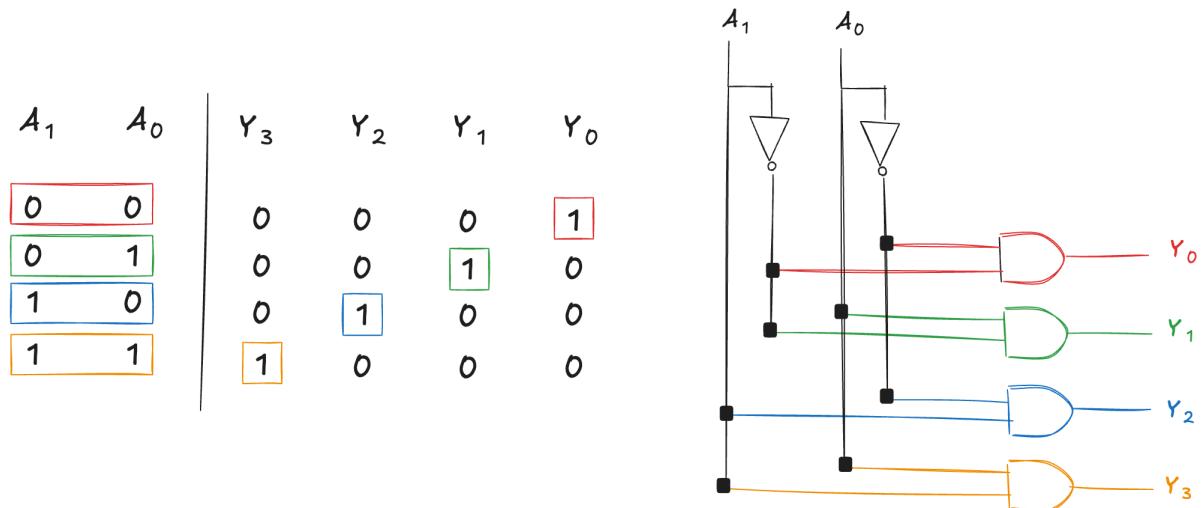
8.2 Decoders

A decoder is a circuit with N inputs and 2^N outputs. For each possible input combination, **exactly one output line is asserted** (set to 1), while all others remain low. The figure below illustrates a 2-to-4 decoder as an example:



The outputs are called **one-hot**, because exactly one is "hot" (HIGH) at a given time.

A 2-to-4 decoder can be implemented using four AND gates and two NOT gates. Each AND gate generates one output by combining the inputs in either their true or complemented form, according to the required minterm.



Starting from the truth table, it is straightforward to write the corresponding VHDL description:

```

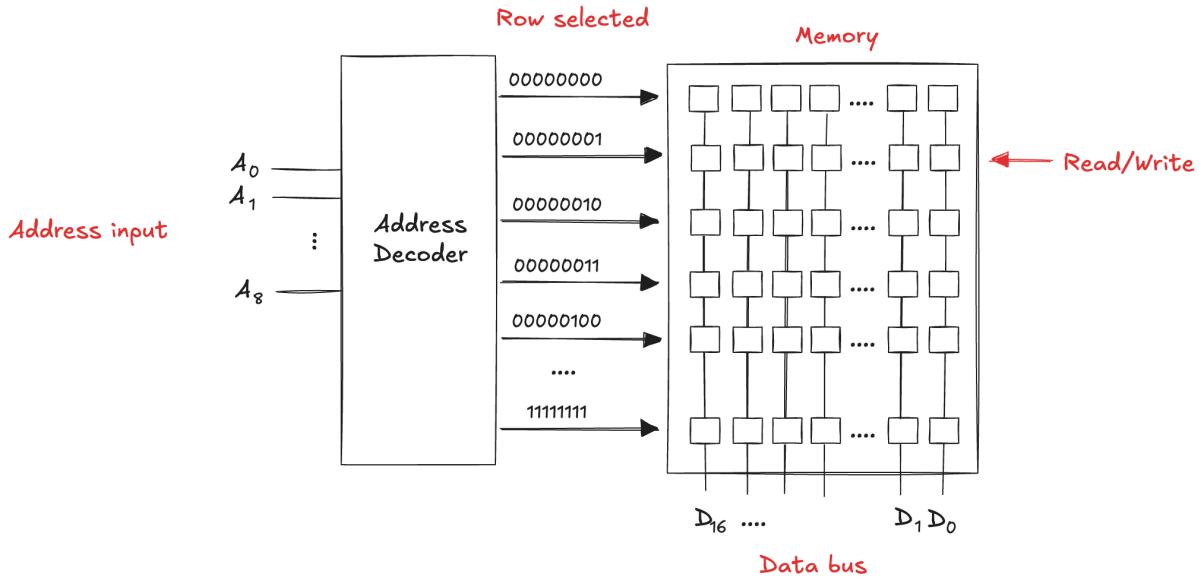
entity decoder_2_4 is
    port( dec_in: in std_logic_vector(0 to 1);
          dec_out: out std_logic_vector(0 to 3)
    );
end decoder_2_4;

architecture rtl of decoder_2_4 is
begin
    with dec_in select
        dec_out <= "0001" when "00",
                    "0010" when "01",
                    "0100" when "10",
                    "1000" when "11",
                    "----" when others;
end rtl;

```

In general, an N-to- 2^N decoder can be constructed using 2^N AND gates, each with N inputs. Each AND gate generates one possible combination of the input variables in their true or complemented form, so that each output corresponds to a single minterm of the input variables.

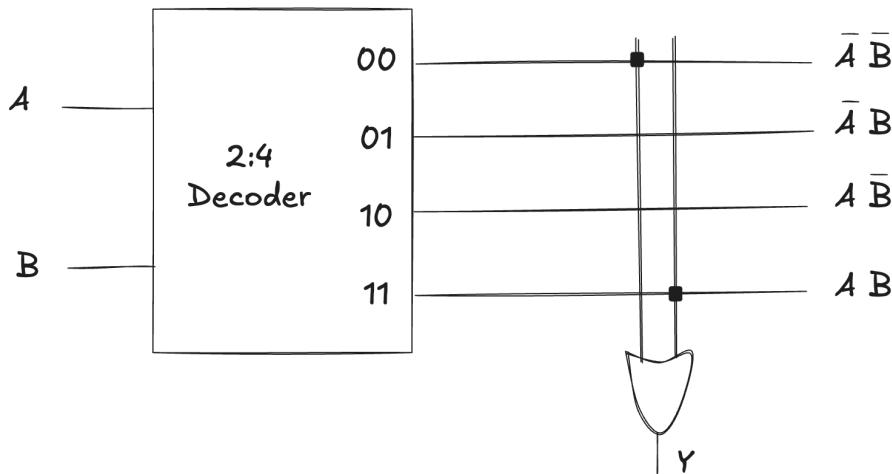
This type of circuit is often used to **decode a memory address**. The inputs represent the memory address in standard binary form, while the output activates exactly one of the memory lines that needs to be accessed:



8.2.1 Decoder Logic

Decoders can be combined with OR gates to **build logic functions**. The following figure shows the two-input XNOR function implemented using a 2:4 decoder and a single OR gate. Because each output of a decoder represents a single minterm, the function is built as the OR of all of the minterms in the function:

$$Y = \overline{A \oplus B} = \overline{A \bar{B}} + A \bar{B}$$



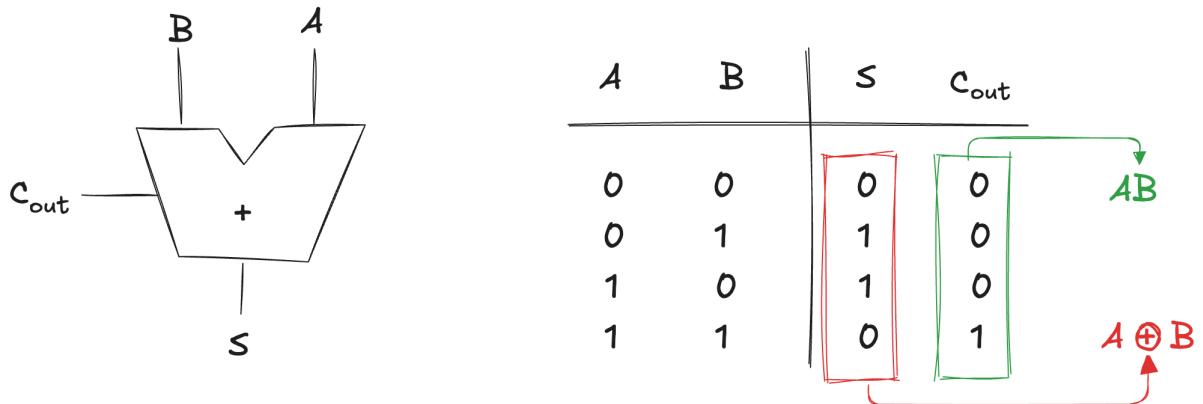
An N-input function with M 1 in the truth table can be built with an N:2^N decoder and an M-input OR gate attached to all of the minterms containing 1 in the truth table. This concept will be applied to the building of read-only memories (ROMs).

8.3 Adder

Addition is one of the most common operations in digital systems. We start considering how to add two 1-bit binary numbers. We then extend to N-bit binary numbers. Adders also illustrate trade-offs between speed and complexity.

8.3.1 Half Adder

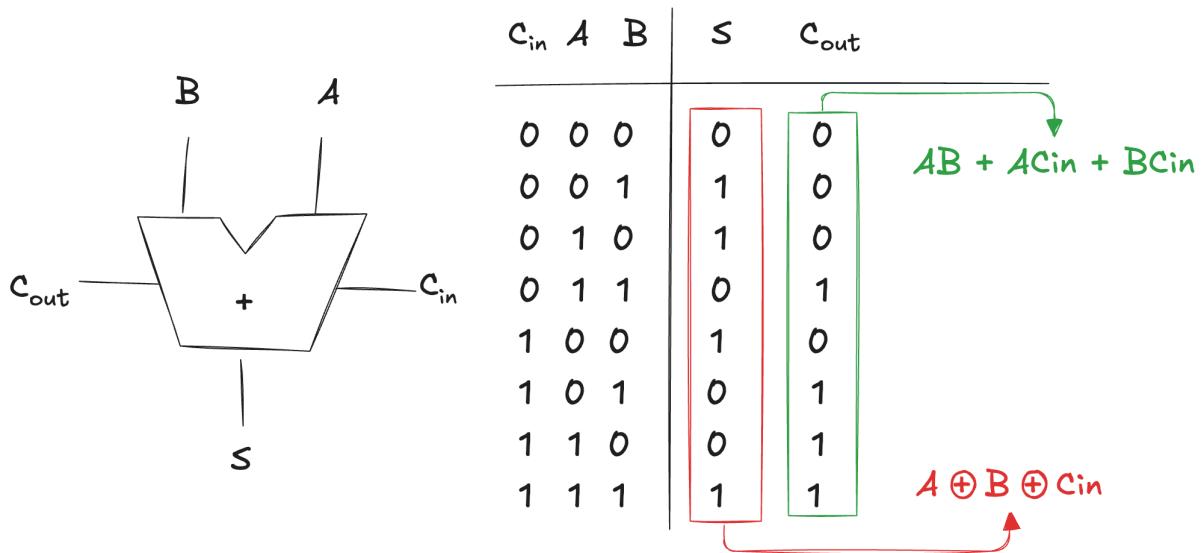
A half adder has two inputs, A and B, and two outputs: S (sum) and Cout (carry out). The sum output represents the addition of A and B. When both inputs are 1, their sum is 2, which cannot be expressed with a single binary digit. Instead, the sum is split: S is 0 and the overflow is indicated by Cout. A half adder can be implemented using an XOR gate for the sum and an AND gate for the carry:



In a multi-bit adder, the carry-out from one bit position must be passed as a carry-in to the next more significant bit. However, a half adder does not have a C_{in} input, so it cannot receive the carry from the previous stage.

8.3.2 Full Adder

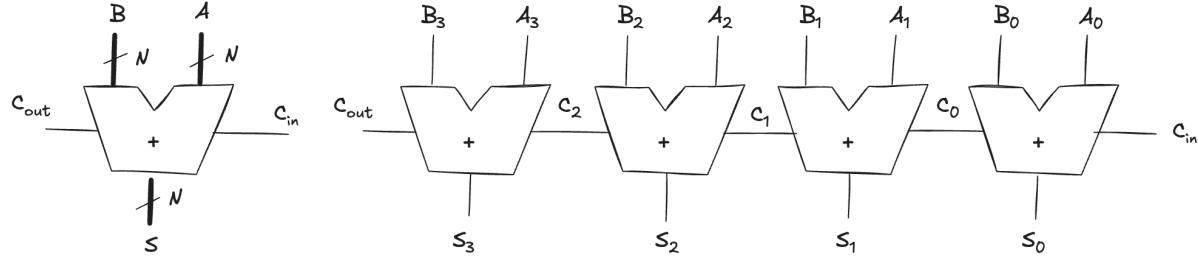
A full adder accepts the carry in C_{in} :



8.3.3 Ripple-carry adder

An N -bit adder sums two N -bit inputs, A and B , along with a carry-in (C_{in}), to produce an N -bit output sum (S) and a carry-out (C_{out}). This type of adder is commonly referred to as a **carry propagate adder (CPA)** because the carry-out from each bit position propagates to the

next higher bit. The simplest implementation **chains together N full adders**, where the Cout of one stage serves as the Cin for the next:



This is known as a **ripple-carry adder**. It is a good example of modularity and regularity: the full adder module is reused multiple times to build a larger system. However, it has the drawback of being **slow for large values of N**. For example, in a 32-bit operation, S₃₁ depends on C₃₀, which depends on C₂₉, and so on, all the way back to the initial Cin. As a result, the adder's delay increases linearly with the number of bits:

$$T_{\text{ripple}} = N \cdot T_{FA}$$

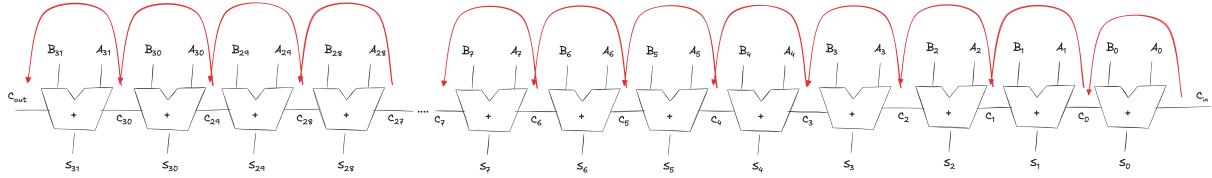
The next VHDL code describes a CPA with carries in and out:

```
entity adder is
    generic(N: integer := 8);
    port(A, B: in std_logic_vector(N-1 downto 0);
         C_in: in std_logic;
         S: out std_logic_vector(N-1 downto 0);
         C_out: out std_logic);
end;

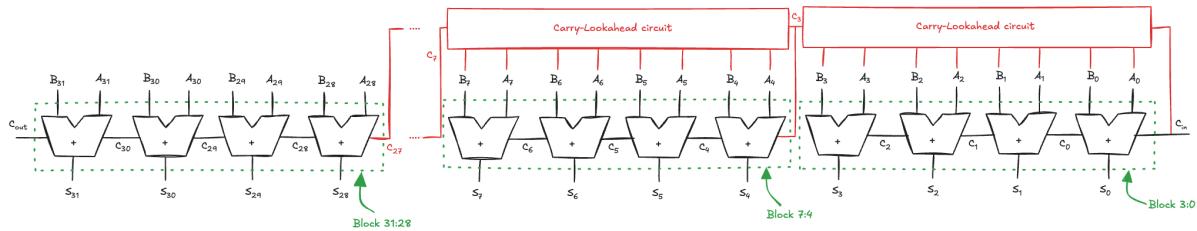
architecture synth of adder is
    signal result: std_logic_vector(N downto 0);
begin
    result <= ("0" & A) + ("0" & B) + C_in;
    S <= result(N-1 downto 0);
    Cout <= result(N);
end;
```

8.3.4 Carry-Lookahead Adder

The fundamental reason why large ripple-carry adders are slow is that the carry signal must propagate through every bit in the adder.



A **carry-lookahead adder (CLA)** addresses this problem by dividing the adder into blocks and adding circuitry that can quickly determine the carry-out of each block as soon as its carry-in is known. In this way, the adder "looks ahead" across blocks instead of waiting for the carry to ripple through every full adder within a block:

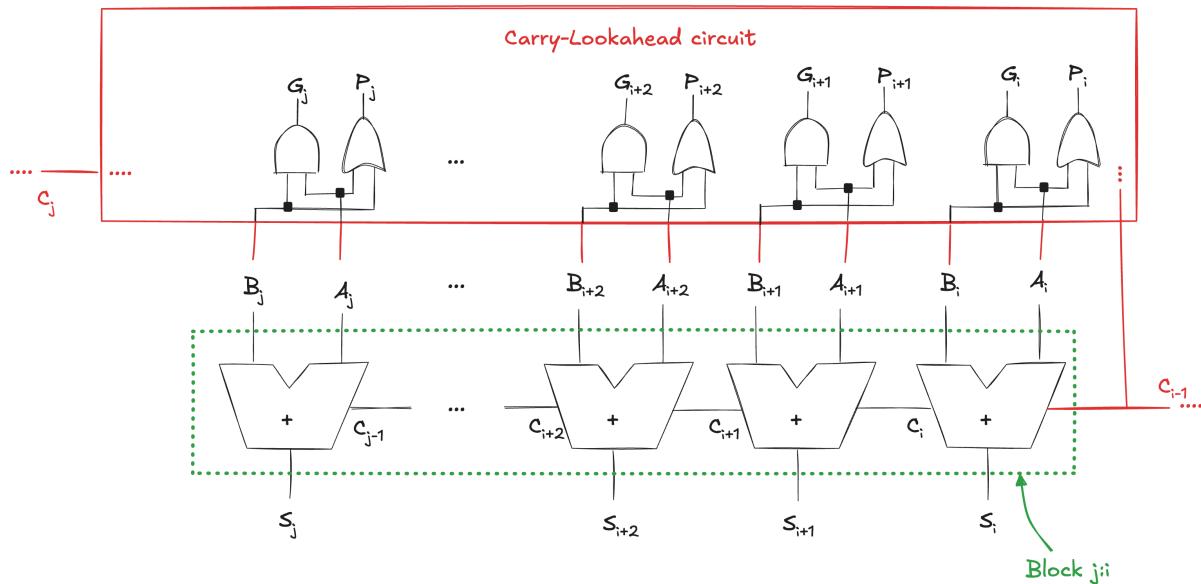


CLAs use **generate** and **propagate** signals to describe how each block determines its carry-out. The i -th bit position of an adder is said to generate a carry if it produces a carry-out regardless of the carry-in, this happens when both inputs are 1:

$$G_i = A_i \cdot B_i$$

The i -th bit position is said to propagate a carry if it produces a carry-out whenever there is a carry-in, this is guaranteed when at least one of its inputs is 1:

$$P_i = A_i + B_i$$



Using these definitions, we can rewrite the carry logic for a given bit position. A bit position will **produce a carry-out if it either generates a carry on its own or propagates an incoming carry**:

$$C_i = G_i + P_i C_{i-1}$$

The generate and propagate definitions **extend to multiple-bit blocks**. A block is said to generate a carry if it produces a carry out independent of the carry in to the block, and this happens if the most significant column generates a carry, or if the previous column generated a carry and the most significant column propagates it, and so forth:

$$G_{j:i} = G_j + P_j (G_{j-1} + P_{j-1} (G_{j-2} + P_{j-2} (\dots (G_{i+2} + P_{i+2} (G_{i+1} + P_{i+1} G_i) \dots))))$$

For example, the generate signal for the second 4-bit block is:

$$G_{7:4} = G_7 + P_7 (G_6 + P_6 (G_5 + P_5 G_4))$$

A block is said to propagate a carry if it produces a carry-out whenever there is a carry-in to the block. This condition holds when all the individual bit positions within the block propagate the carry:

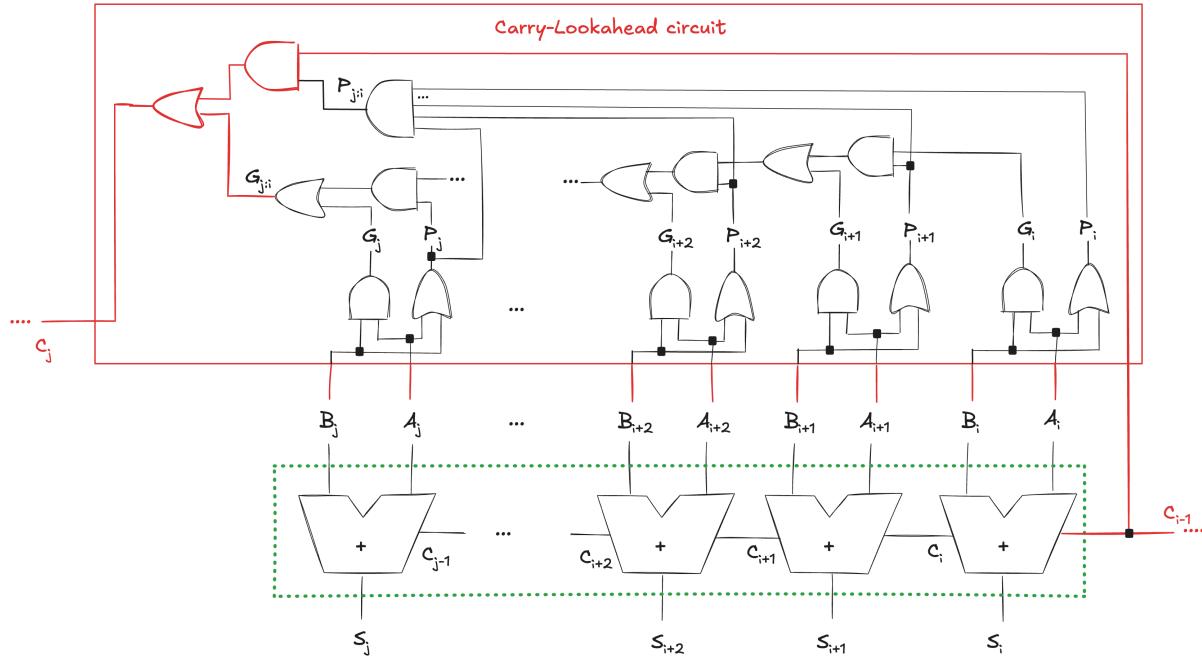
$$P_{j:i} = P_j P_{j-1} P_{j-2} \dots P_{i+2} P_{i+1} P_i$$

For example, the propagate signal for the second 4-bit block is:

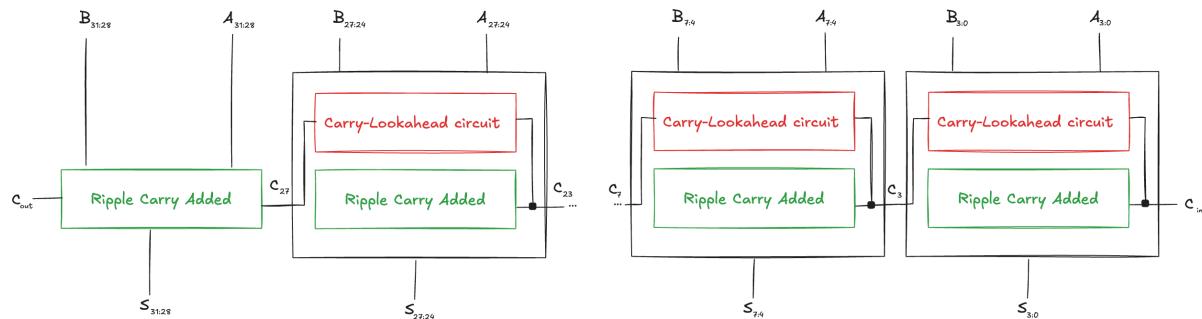
$$P_{7:4} = P_7 P_6 P_5 P_4$$

Using the block generate and propagate signals, we can compute the carry out of the block:

$$C_j = G_{j:i} + P_{j:i} C_{i-1}$$



For example, a 32-bit carry-lookahead adder can be built from eight 4-bit blocks. Each block includes a 4-bit ripple-carry adder along with lookahead logic that computes the block's carry-out based on its carry-in. The final block simply uses a ripple-carry adder, since no additional lookahead logic is needed beyond that point:



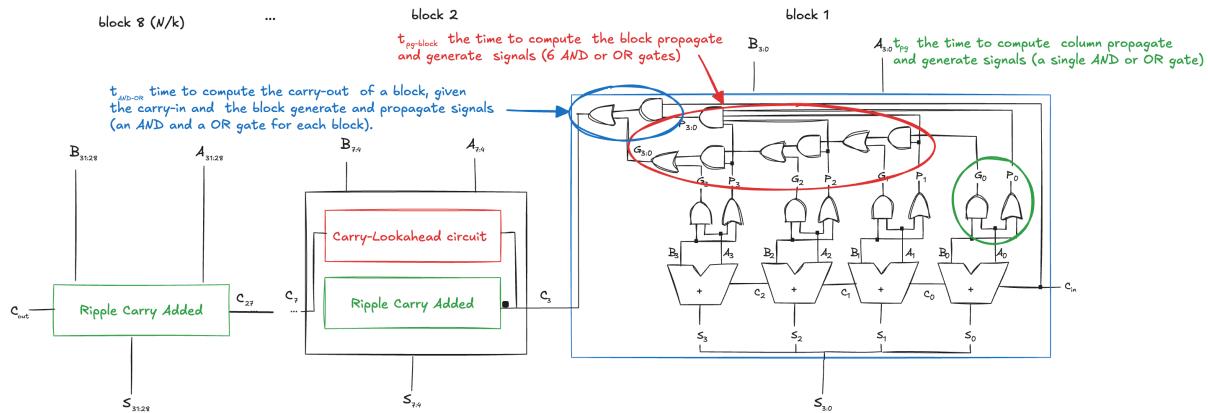
All CLA blocks compute their column-level and block-level generate and propagate signals in parallel. Once these signals are available, the carry-in propagates through the lookahead logic of each block to compute the carry-out for the next stage. Specifically, C_{in} is used to generate C_3 in the first block, C_3 is used to compute C_7 in the second block, then C_7 is used to generate C_{11} , and so on, up to C_{27} in the final stage. The total delay is determined by the number of logic levels required to compute the carry signals across the blocks, rather than by the number of bits:

$$t_{CLA} = t_{pg} + t_{pg-block} + \left(\frac{N}{k} - 1\right) t_{AND-OR} + k t_{FA}$$

where:

- t_{pg} is the time to compute column propagate and generate signals (a single AND or OR gate)
- $t_{pg-block}$ is the time to compute the block propagate and generate signals (this depends on the number of bits in the block, for a 4-bit block it is 6 AND or OR gates)
- $(N/k - 1)$ is the number of blocks that include lookahead logic, which equals the total number of bits divided by the number of bits per block, minus one (since the final block is a standard ripple-carry adder)
- $t_{AND-OR+k}$ is the time to compute the carry-out of a block, given the carry-in and the block generate and propagate signals (an AND and a OR gate for each block). This computation is done in sequence for each block, since it depends on the carry-in from the previous block.
- t_{FA} is the time of a one bit full adder, which is used k times in the last CRA block.

For $N > 16$, the carry-lookahead adder is generally much faster than the ripple-carry adder. However, the adder delay still increases linearly with N . For example, we compare the delay of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. We assume that each two-input gate delay is 100ps and that the full adder delay is 300ps:



According to the previous equations:

$$t_{ripple} = 32 * 300\text{ps} = 9.6\text{ns}$$

$$t_{CLA} = 100\text{ps} + 6 * 100\text{ps} + (32/4 - 1) * 2 * 100\text{ps} + 4 * 300\text{ps} = 3.3\text{ns}$$

In those conditions, the carry-lookahead adder is more three times faster than the ripple-carry adder. However, faster adders **require more hardware** and therefore are **more expensive** and **power-hungry**. These trade-offs must be considered when choosing an appropriate adder for a design. Modern synthesis tools select among many possible implementations, choosing the cheapest (smallest) design that meets the speed requirements. This greatly simplifies the de-

signer's job. We can write in VHDL the code for a 4-bit CLA module that can be used as a building block for larger carry-lookahead adders:

```

entity CLA4 is
    port (A, B      : in  std_logic_vector(3 downto 0);
          C_in     : in  std_logic;
          S        : out std_logic_vector(3 downto 0);
          C_out    : out std_logic);
end CLA4;

architecture behavioral of CLA4 is
    signal G, P : std_logic_vector(3 downto 0);
    signal C    : std_logic_vector(4 downto 0);

begin
    -- Initial carry
    C(0) <= C_in;

    -- Generate and Propagate
    G <= A and B;
    P <= A xor B;

    -- Carry Lookahead Logic
    C(1) <= G(0) or (P(0) and C(0));
    C(2) <= G(1) or (P(1) and G(0)) or (P(1) and P(0) and C(0));
    C(3) <= G(2) or (P(2) and G(1)) or (P(2) and P(1) and G(0)) or
              (P(2) and P(1) and P(0) and C(0));
    C(4) <= G(3) or (P(3) and G(2)) or (P(3) and P(2) and G(1)) or
              (P(3) and P(2) and P(1) and G(0)) or
              (P(3) and P(2) and P(1) and P(0) and C(0));

    -- Sum
    S <= P xor C(3 downto 0);

    -- Final carry-out
    C_out <= C(4);
end behavioral;

```

8.3.5 Subtractor

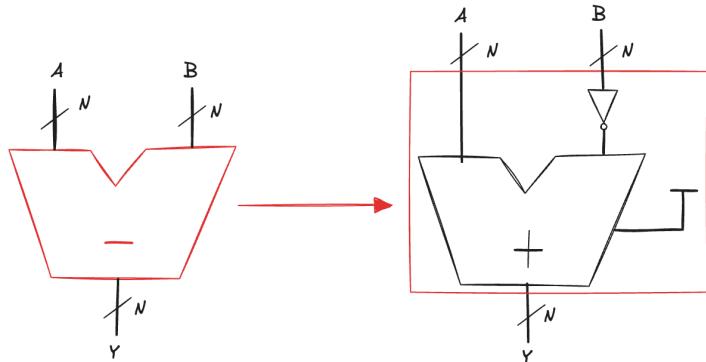
Recall that adders can add positive and negative numbers using two's complement number representation. Subtraction is almost as easy, to compute:

$$Y = A - B$$

we simply create the two's complement of B (inverting all bits of B and adding 1) and the add A:

$$Y = A + \overline{B} + 1$$

This sum can be performed with a single CPA by adding A + B with Cin = 1:

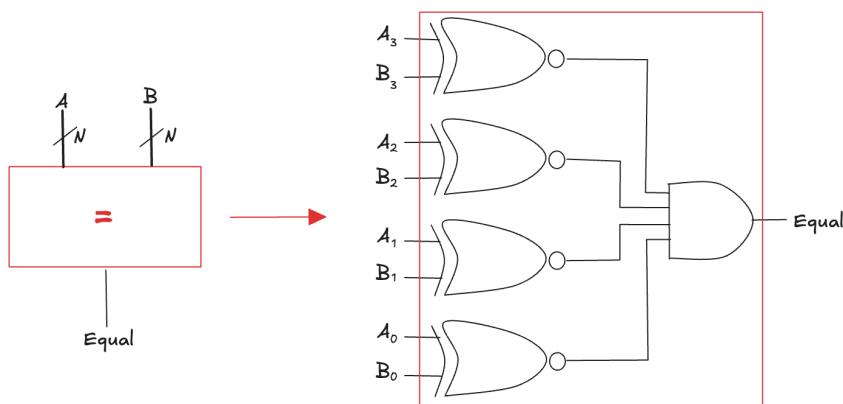


8.4 Comparators

A comparator is a combinational circuit that determines whether two binary numbers are equal (**equality comparator**) or whether one is greater or less than the other (**magnitude comparator**). The equality comparator is simpler in hardware.

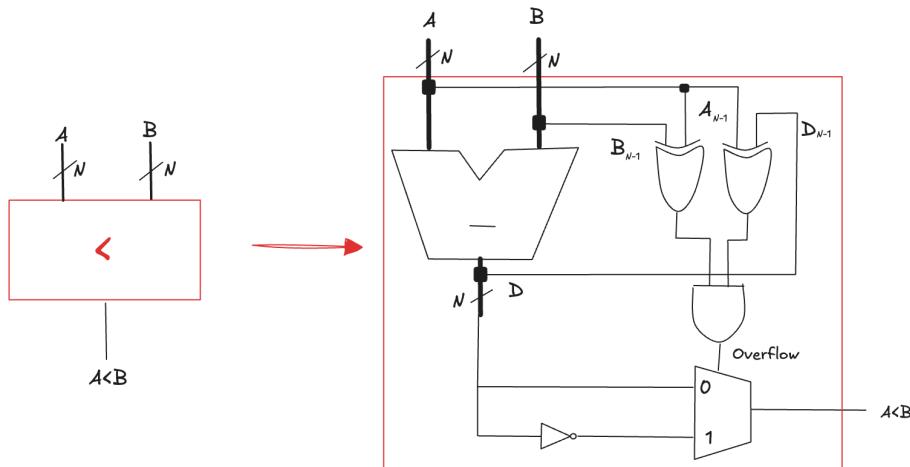
8.4.1 Equality comparator

An equality comparator compares each corresponding bit of the two input numbers, A and B, using XNOR gates. If all bits in every column match, the output indicates that the numbers are equal:



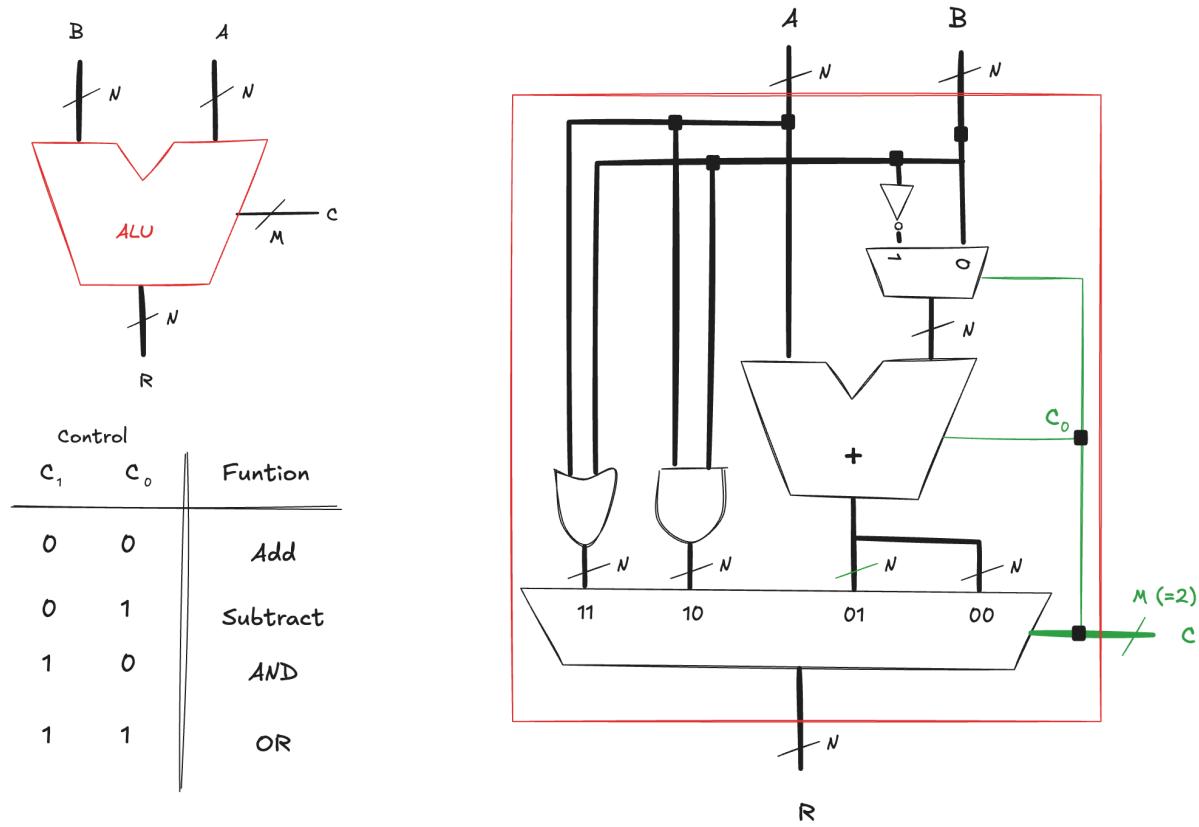
8.4.2 Magnitude comparator

Magnitude comparison of signed numbers is typically performed by computing $(A - B)$ and examining the sign bit (the most significant bit) of the result. If the result is negative (i.e., the sign bit is 1), then $A < B$; otherwise, $A \geq B$. However, this method can give incorrect results in the presence of **overflow**. Overflow occurs when the sign of the result does not match the expected outcome based on the input signs, specifically when A and B have opposite signs and the sign bit of the result differs from the sign of A . To handle this correctly, we must detect this overflow condition and adjust the comparison accordingly:



8.5 Arithmetic/Logical Unit (ALU)

An **Arithmetic/Logic Unit (ALU)** integrates a range of arithmetic and logical operations into a single hardware block. A typical ALU can perform operations such as **addition**, **subtraction**, **bitwise AND**, and **bitwise OR**. The ALU is a central component of most computer architectures, responsible for executing the core computational tasks. To control which operation is executed, the ALU receives a **control signal** that selects the desired function. In diagrams, control signals are typically shown in a different color to distinguish them from **data signals**, which carry the actual operands and results.

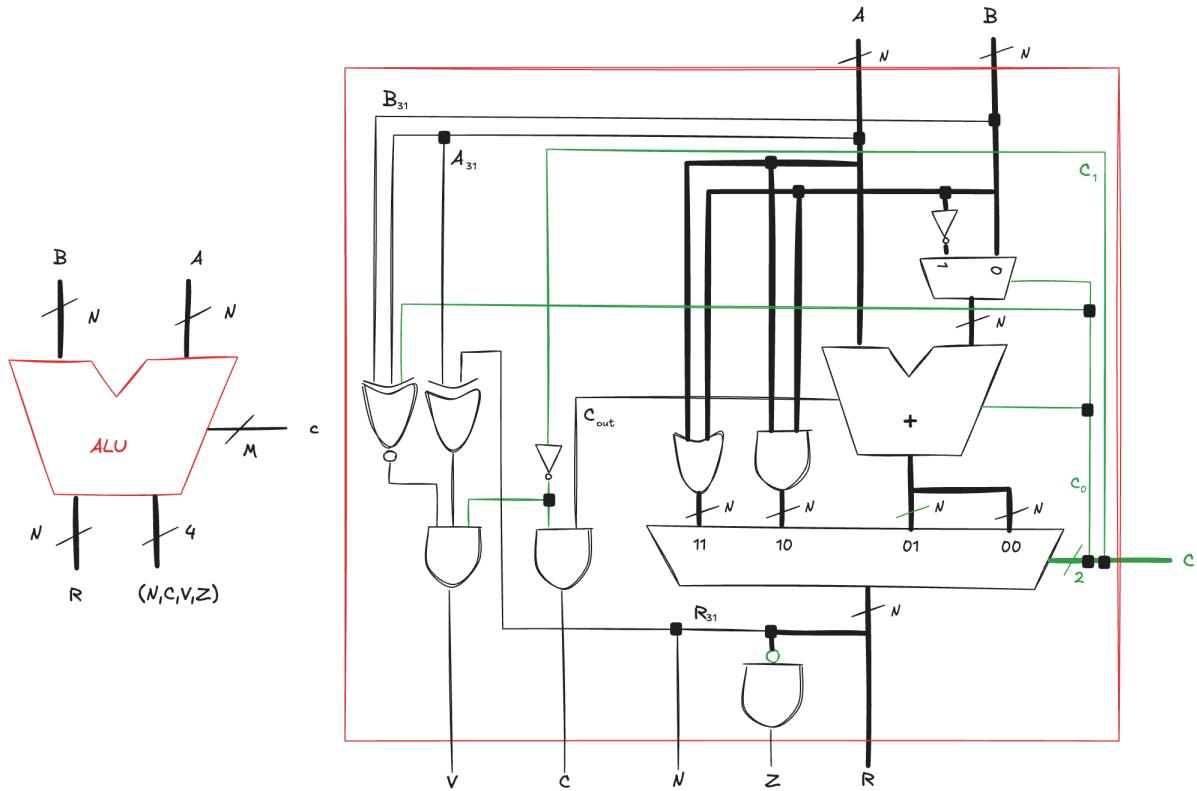


The ALU implementation includes an N-bit adder, along with N 2-input AND gates and N 2-input OR gates. It also incorporates inverters and a multiplexer to conditionally invert input B when the control signal C_0 is asserted, enabling two's complement subtraction. A 4:1 multiplexer selects the final output based on a 2-bit control signal C:

- $C = 00$: the multiplexer selects the sum output, computing the sum
- $C = 01$: the ALU performs subtraction, this is achieved by inverting B and setting the carry-in using C_0 (which is 1), so the adder computes the two complement subtraction
- $C = 10$: the ALU outputs the bitwise AND
- $C = 11$: the ALU outputs the bitwise OR

8.5.1 Flags

Some ALUs produce extra outputs, called **flags**, that indicate information about the ALU output:



As shown in the schematic, the flags output is composed of the **N**, **Z**, **C**, and **V** flags.

The most significant bit of a two's complement number is 1 if it is negative and 0 otherwise. Thus, the **N** (**Negative**) flag is connected to the most significant bit of the ALU output.

The **Z** (**Zero**) flag is asserted when all of the bits of the result are 0, as detected by the N-bit NOR gate.

The **C** (**Carry out**) flag is asserted when the adder produces a carry out and the ALU is performing addition or subtraction (indicated $C_1 = 0$).

Overflow detection is trickier, as it depends on the operation being performed. So, the **V** (**oVerflow**) flag is asserted when all three of the following conditions are true:

- the ALU is performing addition or subtraction ($C_1 = 0$),
- A and sum have opposite signs, as detected by the XOR gate,
- overflow is possible as detected by the XNOR gate: either A and B have the same sign and the adder is performing addition ($C_0 = 0$) or A and B have opposite signs and the adder is performing subtraction ($C_0 = 1$). The 3-input AND gate detects when all three conditions are true and asserts V.

8.5.2 Comparisons

The ALU does not only perform arithmetic operations, but it can also be used for **comparisons** between two operands. This is typically done by subtracting the two operand and examining the resulting flags to determine the relationship between the two numbers:

Condition	Expression	Meaning
$A = B$	$Z = 1$	The subtraction result is zero
$A \neq B$	$Z = 0$	The subtraction result is nonzero
$A < B$	$N \text{ xor } V = 1$	Negative and overflow flags differ
$A \leq B$	$Z \text{ or } (N \text{ xor } V) = 1$	Either equal or less than
$A > B$	$Z \text{ and } (N \text{ xor } V) = 0$	Not equal and not less than
$A \geq B$	$N \text{ xor } V = 0$	Negative and overflow flags are the same

If the **Zero flag (Z)** is set, the subtraction result is zero, which means that **A equals B**. If the Zero flag is not set, the result is different from zero, and consequently **A is not equal to B**.

Magnitude comparisons require a bit more attention. When the subtraction $A - B$ yields a negative result, the **Negative flag (N)** is set, which normally indicates that A is smaller than B. However, this simple interpretation can be misleading when an **overflow occurs**. In such cases, the subtraction result cannot be represented correctly within the available number of bits, and the Negative flag alone no longer reflects the true sign of the difference. To correct this, we need also consider the **Overflow flag (V)**. By combining both flags, we can determine the correct relationship between the operands: A is less than B precisely **when the sign of the result and the overflow indicator do not agree**. This condition ensures that the comparison remains valid even in the presence of signed arithmetic overflow.

Using these conditions, more complex comparisons can be derived. For instance, $A \leq B$ holds if either $A=B$ ($Z = 1$) or $A < B$ ($N \text{ xor } V = 1$). The condition $A > B$ is the negation of $A \leq B$, meaning that both the zero flag must be clear and the combination $N \text{ xor } V$ must be false. Finally, $A \geq B$ is simply the negation of the $A < B$ condition, hence true when $(N \text{ XOR } V)$ is false.

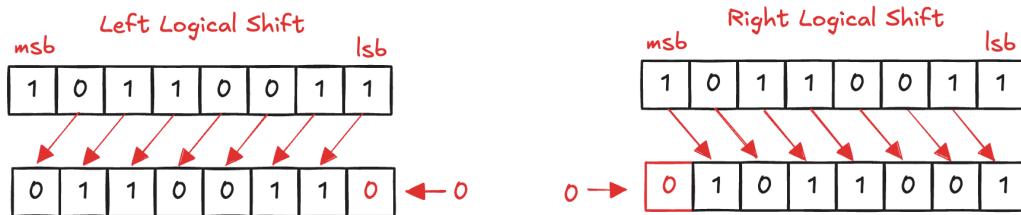
These logical relationships among the ALU flags allow processors to perform **conditional branching**, such as checking whether a value is less than zero or whether it lies within specific bounds. Modern architectures often extend this basic principle, adding new combinations of status bits or defining additional flags, but they all rely on the same fundamental mechanism: interpreting the results of arithmetic operations to guide program flow.

8.6 Shifters

Shifters are digital circuits that **move binary data**, a fundamental operation in arithmetic, logic, and data manipulation. They can shift the bits of a binary word to the left or to the right, changing the position of each bit and often introducing or discarding values at the ends. Depending on the type of shift, the behavior at the boundaries differs, and the interpretation of the result can vary between unsigned and signed numbers.

8.6.1 Logical shifters

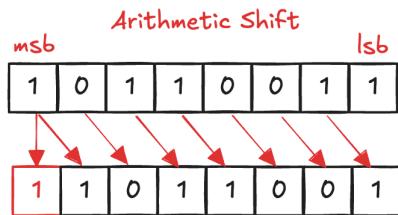
The first and simplest kind of shifter is the **logical shifter**. In this case, all bits are shifted either left or right, and zeros are inserted into the newly vacated positions:



For a left logical shift, the bits move toward the most significant end, and a zero is introduced at the least significant position. Conversely, for a right logical shift, the bits move toward the least significant end, and a zero enters the most significant position. Logical shifts are typically used with **unsigned binary numbers**, where no sign bit needs to be preserved.

8.6.2 Arithmetic shifters

The **arithmetic shifter** behaves similarly, but it preserves the **sign bit** (the most significant bit) when shifting to the right:



This distinction is crucial for signed numbers represented in **two's complement**, where the most significant bit encodes the sign of the number. When an arithmetic right shift is performed, the sign bit is replicated instead of being replaced by a zero. This keeps the sign consistent and effectively divides the number by powers of two, rounding toward negative infinity.

Shifts are not just data manipulation tools, they have direct **arithmetic meaning**. A **left shift** by N positions is equivalent to multiplying the number by 2^N . For instance, shifting the binary value 3

000011

left by four bits results in

110000

which corresponds to multiplying 3 by 2^4 , giving 48.

A **right shift**, on the other hand, divides the number by 2^N .

For signed numbers, an arithmetic right shift ensures that negative values remain negative. For example, shifting -4

11100

right by two bits gives

11111

corresponding to $-4/(2^2) = -1$.

In summary, shifters are essential components, supporting a range of operations from simple scaling to efficient implementation of multiplication, division, and bitwise manipulation.

In the following VHDL code, we describe an N-bit shifter that can perform both logical and arithmetic shifts, in either direction, based on control signals:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Shifter is
    generic (
        N : integer := 8  -- width of the data word
    );
    port (
        A      : in  std_logic_vector(N-1 downto 0);  -- input data
        SHIFT  : in  integer range 0 to N-1;           -- shift amount
        DIR    : in  std_logic;                         -- 0 = left, 1 = right
        ARITH  : in  std_logic;                         -- 0 = logical, 1 = arithmetic
        Y      : out std_logic_vector(N-1 downto 0)    -- output data
    );
end Shifter;
```

```

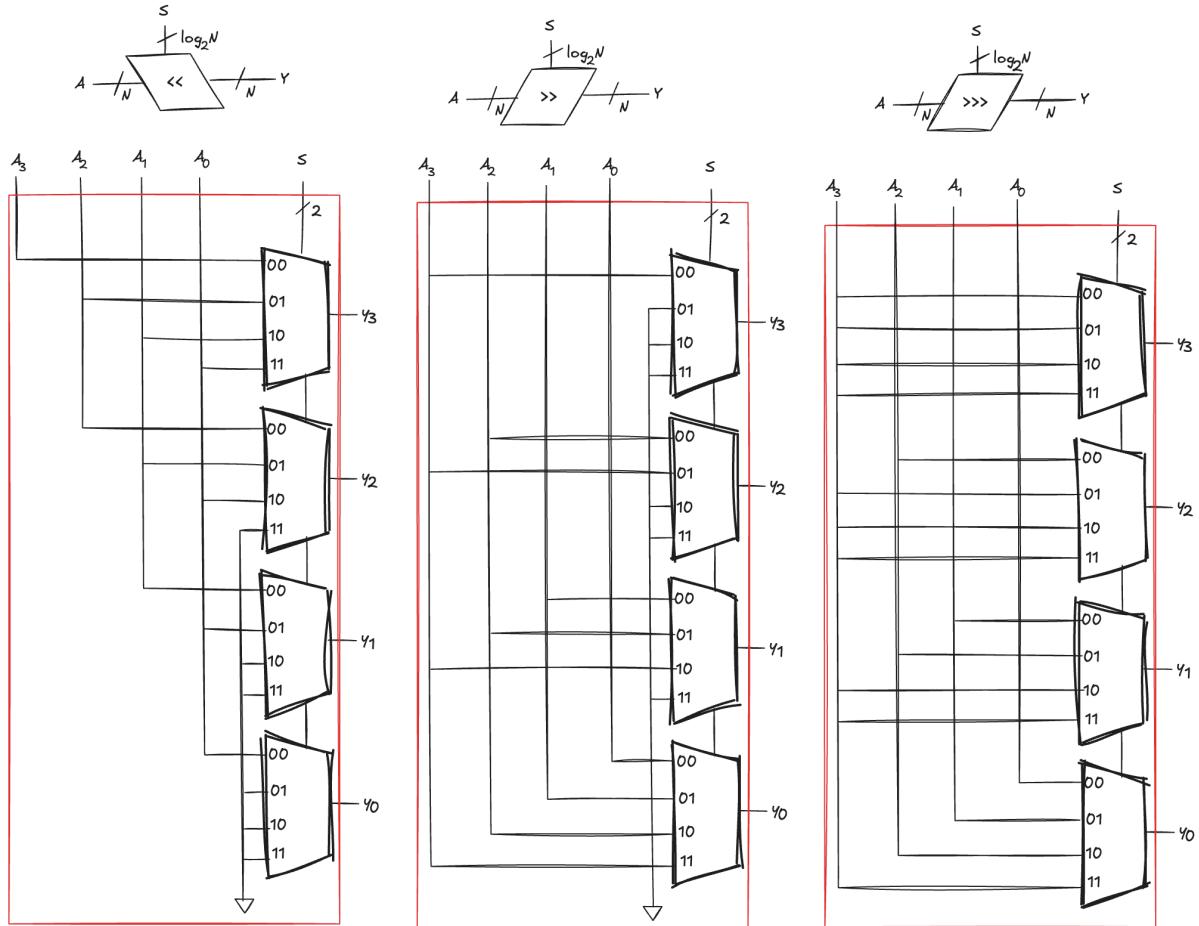
architecture Behavioral of Shifter is
begin
    process (A, SHIFT, DIR, ARITH)
        variable temp : std_logic_vector(N-1 downto 0);
    begin
        if DIR = '0' then -- LEFT SHIFT
            if SHIFT < N then
                temp := std_logic_vector(shift_left(unsigned(A), SHIFT));
            else
                temp := (others => '0');
            end if;
        else -- RIGHT SHIFT
            if ARITH = '1' then -- Arithmetic right shift (preserve sign bit)
                temp := std_logic_vector(shift_right(signed(A), SHIFT));
            else -- Logical right shift (fill with zeros)
                temp := std_logic_vector(shift_right(unsigned(A), SHIFT));
            end if;
        end if;
        Y <= temp;
    end process;
end Behavioral;

```

In this VHDL description, the **shift amount** is declared as an **integer** rather than a **std_logic_vector** because the language's built-in functions **shift_left()** and **shift_right()** expect an integer to specify how many positions the bits must move. This makes the code simpler and more readable, avoiding type conversions. The use of a **process** block ensures that every time the input, direction, or shift amount changes, the output is recomputed. Overall, the design expresses the shift operation behaviorally, using numeric operations (**shift_left**, **shift_right**) controlled by integer parameters, while the synthesizer will later translate it into equivalent hardware.

8.6.3 Shifters as multiplexers

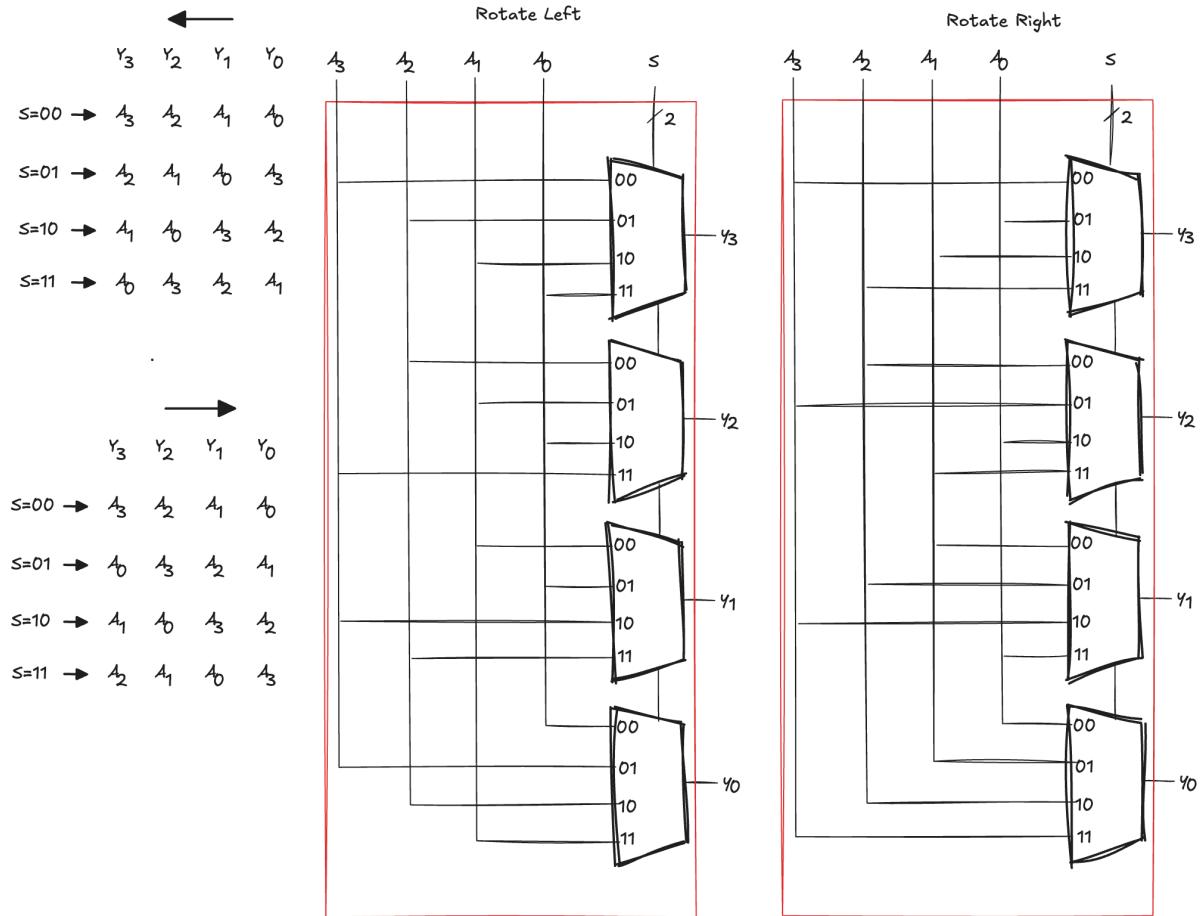
An **N-bit shifter** can be systematically built using **N multiplexers**, each with **N inputs**. Every multiplexer selects which input bit will appear at its output, depending on a control signal that represents the desired shift amount. The control signal has a width of **log₂(N)** bits, allowing the circuit to select any shift amount from **0** to **N-1** positions:



To understand this structure, imagine each output bit being driven by one multiplexer that can choose among all input bits. For instance, in a left shifter, the least significant output can select between all input bits shifted left by varying amounts, while higher-order outputs will select correspondingly higher input bits. If the shift amount is 0, the output equals the input; if it is 1, all bits move one position to the left, and so on.

8.6.4 Rotators

A **rotator** is a circuit that moves the bits of a binary word in a **circular fashion**, unlike a shifter that discards bits at one end and inserts zeros. In a rotation, the bits that "fall off" one side are **reintroduced on the opposite side**, so the pattern of bits is preserved, just rearranged. This operation can be performed in either direction: **rotate left** or **rotate right**:



For example, in a 4-bit word, a left rotation by one position moves each bit toward the more significant end, and the most significant bit reappears at the least significant position. Similarly, a right rotation moves each bit toward the least significant end, and the least significant bit reappears at the most significant position.

This behavior makes rotation different from shifting because no information is lost. Instead, the bit pattern is **cycled** through the positions, making rotators particularly useful in certain arithmetic operations, bitwise manipulations, and cryptographic algorithms, where circular movement of bits can produce predictable but non-destructive transformations.

Hardware-wise, a rotator can be implemented similarly to a shifter, using **multiplexers** to select which input bit will appear at each output position. The control signal, typically encoded on $\log_2(N)$ bits, determines the rotation amount (for example, from 0 to $N-1$). By modifying the multiplexer connections, the same structure can support left or right rotations.

The following VHDL code describes an N-bit rotator that can perform both left and right rotations based on a control signal:

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

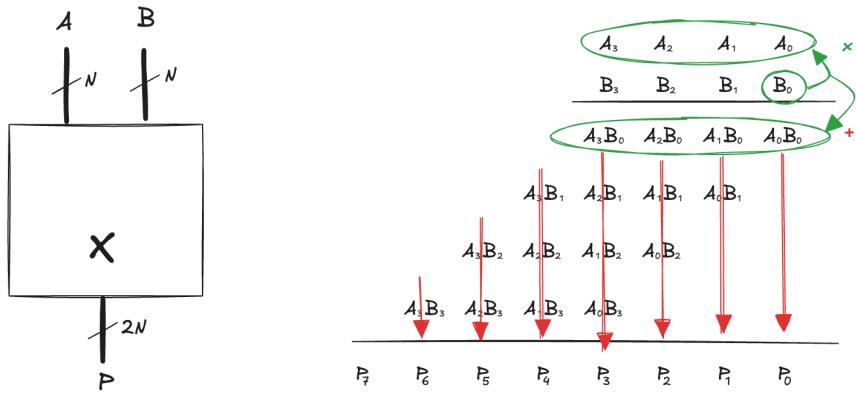
entity Rotator is
    generic (
        N : integer := 8 -- width of the data word
    );
    port (
        A      : in  std_logic_vector(N-1 downto 0); -- input data
        SHIFT : in  integer range 0 to N-1;           -- rotation amount
        DIR   : in  std_logic;                         -- 0 = left, 1 = right
        Y      : out std_logic_vector(N-1 downto 0)    -- output data
    );
end Rotator;

architecture Behavioral of Rotator is
begin
    process (A, SHIFT, DIR)
        variable temp : std_logic_vector(N-1 downto 0);
    begin
        if DIR = '0' then
            -- Rotate left
            temp := A(N-1-SHIFT downto 0) & A(N-1 downto N-SHIFT);
        else
            -- Rotate right
            temp := A(SHIFT-1 downto 0) & A(N-1 downto SHIFT);
        end if;
        Y <= temp;
    end process;
end Behavioral;

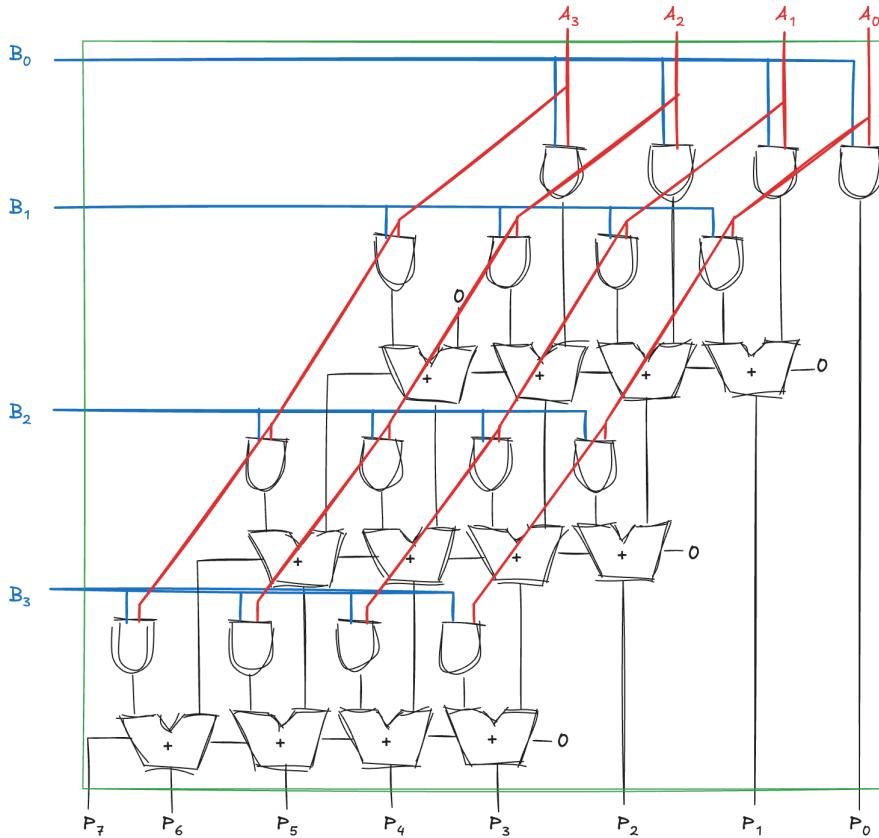
```

8.7 Multipliers

Binary multiplication relies on the same principle as decimal multiplication, except that each bit of the multiplier can only be 0 or 1. The multiplication of two single bits is **equivalent to the AND operation**: the result is 1 only when both inputs are 1. A general **$N \times N$ multiplier** produces a result that is **$2N$ bits wide**, because it is obtained by generating and summing a series of **partial products**:



Each **partial product** is formed by taking one bit of the multiplier and performing a bitwise AND with all bits of the multiplicand. This operation produces an N -bit value. Then, each partial product is **shifted left** according to the position of the multiplier bit and added to the previous partial products. In a 4×4 unsigned multiplier, this results in four partial products, each shifted and then added together. The final product is an 8-bit number:



Each bit of the multiplier controls a row of AND gates, generating the corresponding partial product by multiplying that bit with every bit of the multiplicand. These partial products are

then summed together using a network of half full adders (FA). This **array multiplier** architecture is a simple and regular hardware implementation, scalable to any bit width (N). Although this design is efficient for small operands, larger multipliers often use more advanced architectures (e.g., **carry-save**, **Wallace tree**, or **Booth multiplier**) to reduce delay and hardware complexity.

We can consider the VHDL implementation, using the built-in multiplication operator from numeric_std library, which synthesis tools implement as a network of AND gates and adders, similar to our diagram):

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Multiplier is
    generic (
        N : integer := 4  -- width of operands
    );
    port (
        A : in std_logic_vector(N-1 downto 0);
        B : in std_logic_vector(N-1 downto 0);
        P : out std_logic_vector(2*N-1 downto 0)
    );
end Multiplier;

architecture Behavioral of Multiplier is
begin
    process (A, B)
        variable A_u : unsigned(N-1 downto 0);
        variable B_u : unsigned(N-1 downto 0);
        variable P_u : unsigned(2*N-1 downto 0);
    begin
        -- convert to unsigned
        A_u := unsigned(A);
        B_u := unsigned(B);

        -- perform multiplication
        P_u := A_u * B_u;

        -- assign result
        P <= std_logic_vector(P_u);
    end process;
end Behavioral;

```

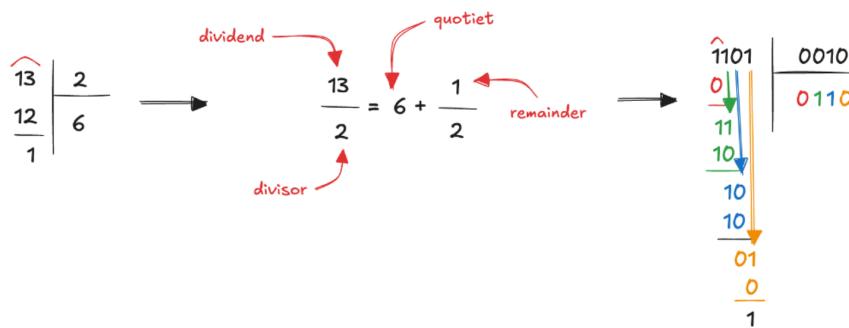
The VHDL code uses the "unsigned" function from the numeric_std library to give numerical meaning to bit vectors, allowing standard arithmetic operations such as multiplication. The * operator performs binary multiplication between two unsigned operands, producing a result twice as wide as the inputs. Without the unsigned conversion, the multiplication will be treated as a bitwise operation (and AND bit by bit), which is not the intended behavior for arithmetic multiplication.

8.8 Dividers

Division is **the most complex of all basic arithmetic operations** to implement in hardware, as it involves repeated comparison, subtraction, and shifting. Conceptually, binary division follows the same logic as **long division in decimal**. If we divide a number A (the dividend) by another number B (the divisor), we obtain a **quotient** (Q) and a **remainder** (R), according to the relation:

$$\frac{A}{B} = Q + \frac{R}{B}$$

The goal is to determine how many times B can fit into A without exceeding it. Each time the divisor fits, we append a 1 to the quotient; when it does not fit, we append a 0:



The process is iterative:

```
R ← 0                                -- Initialize remainder
For i from N-1 downto 0 loop          -- Process each bit from MSB to LSB
    R' ← (R << 1) + A[i]            -- Shift left and bring down next bit of A
    D ← R' - B                      -- Subtract divisor
    if D < 0 then                   -- If subtraction negative
        Q[i] ← 0                     -- Quotient bit = 0
        R ← R'                      -- Restore previous remainder (no change)
    else
        Q[i] ← 1                     -- Quotient bit = 1
        R ← R' - B                  -- Perform subtraction
```

```

else
    Q[i] ← 1           -- Quotient bit = 1
    R ← D             -- Keep new remainder
end if
end loop

```

At the beginning, the remainder R is initialized to 0. In each iteration, the algorithm shifts the remainder left by one bit and brings down the next bit of the dividend A[i]. This new value represents the partial remainder R'. The divisor B is then subtracted from this partial remainder to test whether it fits. If the result D is negative, the subtraction exceeded the value that can be divided, so the quotient bit Q[i] is set to 0, and the remainder is restored (kept unchanged). If the result is non-negative, the quotient bit is set to 1, and the remainder becomes the new difference D. The final remainder R and quotient Q are obtained after all bits have been processed.

This procedure is known as the **restoring division algorithm**, since the remainder is "restored" to its previous value whenever the subtraction overshoots. Later variations, such as **non-restoring** and **SRT division**, were developed to improve efficiency, but the restoring algorithm remains fundamental for understanding division in digital circuits.

Let's consider a step-by-step example:

$$A = 1101 \text{ (13)}$$

$$B = 0010 \text{ (2)}$$

$$R = 0000 \text{ (0)}$$

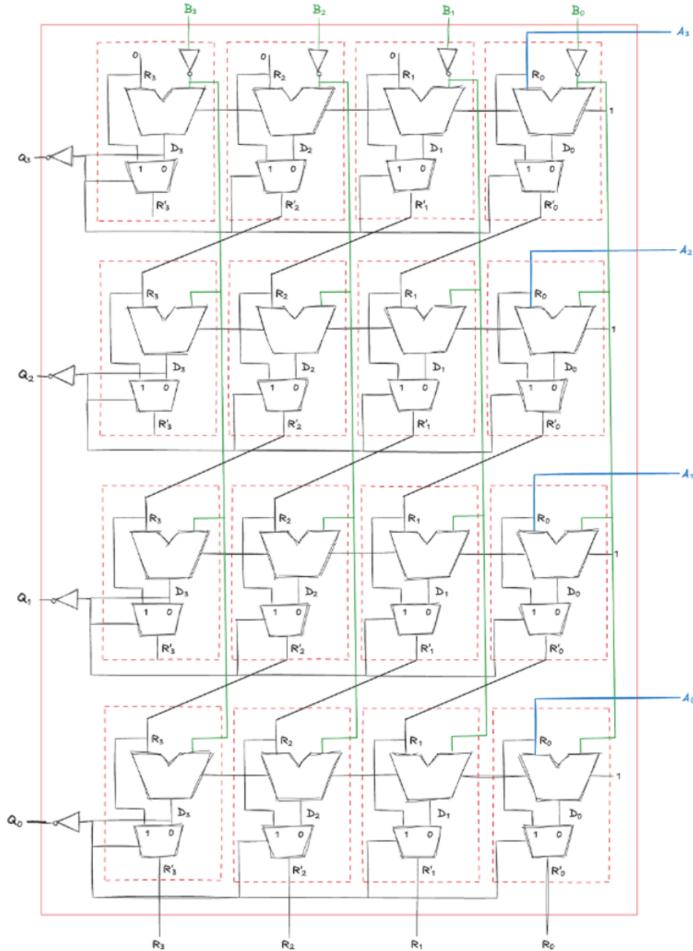
Step	i	Operation	R'	D (R' - B)	Q[i]	R
1	3	$(0000 \ll 1) + 1$	0001	Negative	0	0001
2	2	$(0001 \ll 1) + 1$	0011	0001	1	0001
3	1	$(0001 \ll 1) + 0$	0010	0000	1	0000
4	0	$(0000 \ll 1) + 1$	0001	Negative	0	0001

Final Result:

$$Q = 0110 \text{ (6)}$$

$$R = 0001 \text{ (1)}$$

The following diagram represents a **4×4 restoring division array**, built from a grid of basic **division cells**. Each cell performs a small part of the restoring division algorithm, combining **subtraction**, **restoration**, and **shift** logic:



Each row corresponds to one bit of the quotient and each column corresponds to one bit of the divisor. The **dividend bits** enter along the right edge, and the **remainder bits** propagate downward through the array. Each cell performs the following local operation:

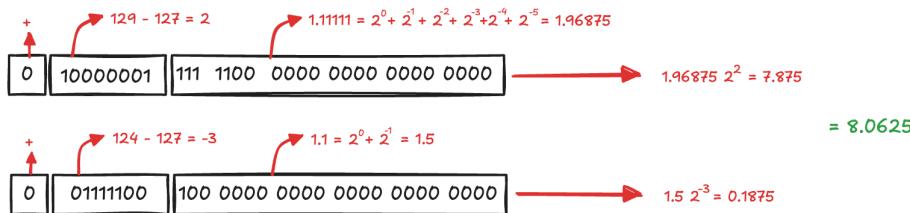
1. Subtracts the divisor bit from the partial remainder.
2. If the result is negative, it restores the previous remainder (undoing the subtraction).
3. Passes the updated remainder to the next lower row, and the shifted divisor to the next column.

This **array-based organization** allows all partial remainder updates and restorations to proceed **in parallel** across the hardware structure, while the control logic shifts through each bit of the dividend sequentially. This design directly maps the restoring division algorithm into a regular,

modular hardware structure, making it easy to scale to higher bit widths.

8.9 Floating-Point Adder

Finally, we can consider also operations involving **floating-point numbers**, which are used to represent a wide range of real numbers in scientific and engineering applications. Floating-point arithmetic is more complex than integer arithmetic due to the need to handle varying exponents and normalization. As described by the IEEE 754 standard, Floating-point numbers are represented using three fields: the **sign**, the **exponent**, and the **fraction (mantissa)**. To add two floating-point values, the hardware must perform a series of alignment and normalization steps to ensure the numbers share the same exponent before performing the actual addition. Let's consider the example shown:



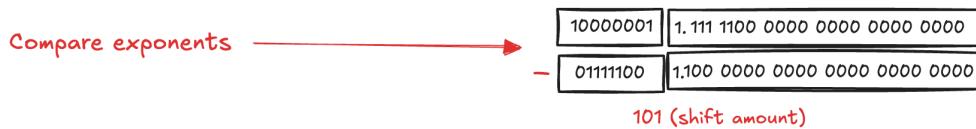
The first step is to separate the sign, exponent, and fraction from each operand. For normalized numbers, the exponent is stored in **biased form** (bias = 127 for single precision):



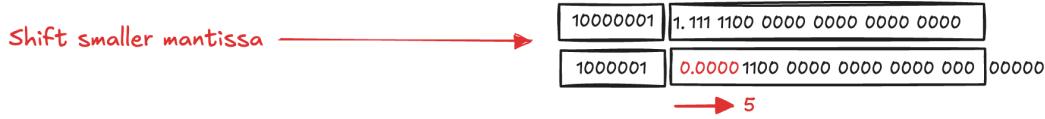
Normalized floating-point numbers always have an **implicit leading 1** before the binary point (because the mantissa is in the form 1.xxxxx). So, the fraction field becomes:



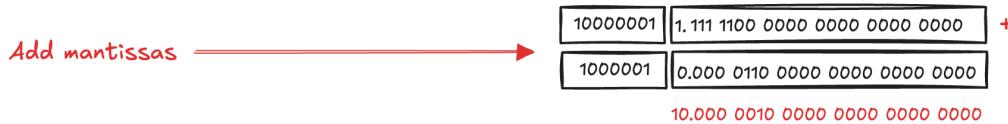
To align the two numbers, the smaller exponent must be increased by shifting its mantissa right:



The mantissa of the number with the smaller exponent is right-shifted by the difference between exponents. This ensures both numbers are scaled to the same power of two before addition:



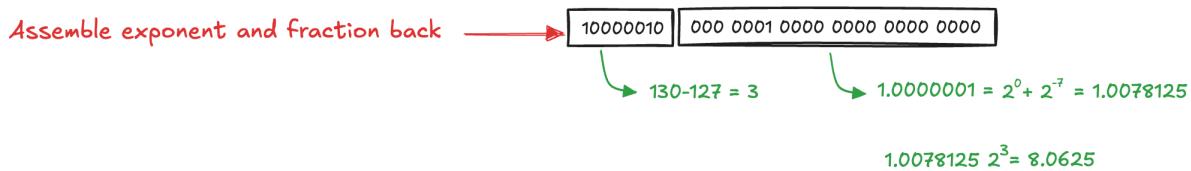
Once aligned, the mantissas are added just like integers:



The result must be adjusted so that the mantissa is again in normalized form (i.e., only one '1' before the binary point):



Finally, the mantissa (minus the implicit 1) and the new exponent are re-encoded into IEEE 754 format:



Floating-point arithmetic is one of the most demanding operations to implement in hardware because it requires handling normalization, rounding, overflow, and precision. For this reason, it is typically carried out by a dedicated hardware block known as the **Floating-Point Unit (FPU)**. The FPU operates alongside the **Central Processing Unit (CPU)** but is distinct from it, allowing arithmetic on floating-point numbers to be performed much faster and more accurately than if implemented in software.

One of the most famous hardware bugs in computing history is the **Intel Pentium floating-point division bug**, discovered in 1994:



The issue arose from a flaw in the lookup table used by the Pentium's FPU to compute division.

Specifically, several entries were missing from a table that stored precomputed reciprocal values, which the hardware used during the division process. When certain rare combinations of input operands accessed one of these missing entries, the FPU would return **slightly incorrect results**, errors typically appearing in the 4th or 5th decimal place. Although the probability of such an error occurring was extremely small, the potential for incorrect scientific and financial calculations made the issue unacceptable. In some specific cases, depending on operand values, the errors could accumulate or occur repeatedly, producing visibly incorrect results.

In **December 1994**, Intel publicly acknowledged the defect and issued a large-scale **recall of all affected Pentium processors**, marking the first full recall of a computer chip in history. The total cost of replacing processors and repairing Intel's reputation was estimated at **\$475 million**. This incident became a classic example of how **tiny hardware design errors in arithmetic units** can have enormous financial and technological consequences, emphasizing the importance of **verification and exhaustive testing** in hardware design.

9 Sequential Circuits

In digital design, it is important to distinguish between **combinational** and **sequential** logic. A combinational circuit produces outputs that depend only on the **current values** of its inputs. If we know the truth table or the Boolean equations describing the circuit, we can fully predict its behaviour at any instant. Nothing about the past matters, because the circuit has **no memory** of previous inputs.

Sequential logic behaves differently. Its outputs depend not only on the inputs that are present right now, but also on **inputs that occurred earlier**. In other words, a sequential circuit has **memory**. However, we do not necessarily mean that it remembers *all* the past inputs exactly as they occurred. Instead, a sequential circuit remembers only the information that is necessary to determine what it should do next. This essential information is what we call the **state** of the system: a collection of bits (called **state variables**) that summarizes everything about the past that is **relevant** for predicting the future behaviour of the circuit. Think of the state as the circuit **internal snapshot** of its history. It does **not** store the entire history; it stores only what matters. For example, a washing machine controller can be in phases like "wash", "rinse", "spin". Each phase determines what the system does and which phase comes next. The controller does not need to remember every button press or water-level measurement. It remembers only its current phase, its state.

Sequential logic is important because many real-world tasks require memory of past events. For example: counting events over time, coordinating the phases of a protocol or saving intermediate results while processing data. A **stateful system** can perform these tasks because the state

acts as a structured memory of past events.

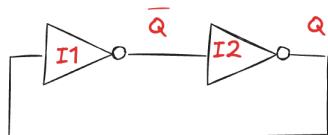
Because the behaviour of sequential circuits **evolves over time and depends on past events**, analysing them can be **significantly more challenging** than analysing combinational circuits. A powerful way to simplify this complexity is to **discipline ourselves** to design only **synchronous** sequential circuits. In a synchronous system, all memory elements update their state in a co-ordinated way controlled by a common signal, called the **clock**. This allows us to view the circuit as consisting of two well-defined parts:

- the **combinational logic**, that **computes the next values of the state**
- a bank of memory elements to store the current state and update it at each clock edge.

This disciplined structure makes it much easier to design, analyse, and verify sequential digital systems.

9.1 Bistable Element

The simplest building block of memory in digital systems is a circuit that can rest in **two different stable configurations**. Such a circuit is called **bistable**. It can store one bit of information because each of its two stable states naturally corresponds to a logical 0 or a logical 1. The most fundamental example of a bistable circuit is formed by **two inverters connected in a loop**, also called **cross-coupled inverters**:



The output of the first inverter feeds the input of the second, and the output of the second feeds back into the first. Although this circuit has no external inputs, it has two outputs which always hold opposite logical values. Because the inverters feed each other, the circuit forms a **cycle**: whatever value one inverter produces becomes the input for the other. What makes this arrangement interesting is that the feedback **reinforces the stored value**. Let us look at the two possible situations:

Case 1: assume $Q=0$

- This means that inverter I1 receives a TRUE input, because it is connected to \bar{Q} .
- As an inverter, I1 outputs FALSE, confirming $Q=0$
- Meanwhile, inverter I2 receives a FALSE input, and therefore outputs TRUE, confirming $\bar{Q}=1$

The two values reinforce each other perfectly. Every signal is consistent with the assumption, so nothing tries to change. The circuit naturally stays in this configuration, it is **stable**.

Case 2: assume Q=1

- Inverter I1 now receives FALSE and outputs TRUE, maintaining Q=1.
- Inverter I2 receives TRUE and outputs FALSE, maintaining not-Q=0.

Again, the circuit is stable: no signal contradicts our initial assumption.

The crucial point is that the circuit has **two** self-consistent configurations. Both are stable and can persist indefinitely as long as power is supplied. The circuit will stay in one of these states until something external forces it to switch to the other. This ability to hold one of two values makes the cross-coupled inverter pair the **fundamental memory cell** used in digital electronics.

Because it can rest in **exactly two stable states**, we call the circuit **bistable** ("bi" meaning two, "stable" meaning self-sustaining). More generally, a circuit with **N stable states** can store **log₂(N) bits** of information. Since a bistable circuit has N=2 stable states, it stores 1 bit.

The pair of inverters uses feedback to reinforce the current value indefinitely. This means that the present value of Q contains all the relevant information about the past that determines what the circuit will do in the future. If the circuit is in the state Q=0, it will stay in that state; if it is in the state Q=1, it will remain there unless some external event forces it to change.

9.2 Latches

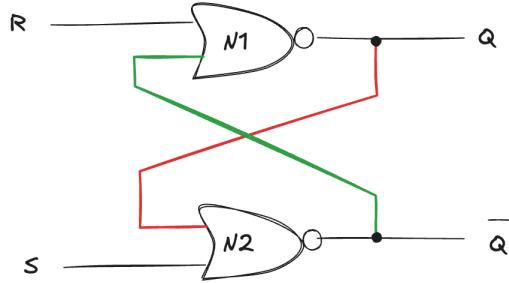
An important limitation of the simple bistable circuit is that it has **no inputs**. It can store a value, but the user cannot directly force it into either state. As a result, when the power supply is first applied, the circuit's initial state is **unknown and unpredictable**. Small variations in manufacturing, temperature, or electrical noise may push it toward either stable state. Consequently, each time the circuit is powered on, it may spontaneously settle into 0 or 1, and the user has no way to control the outcome.

This lack of control means that, although the cross-coupled inverter pair behaves correctly as a memory element, it is not **practical** for real digital systems. A useful memory cell must allow us not only to **store** information but also to **set** or **update** that information in a controlled and deterministic way.

This limitation is what motivates the introduction of more sophisticated memory element called **latch**, which extend the bistable structure with inputs that allow us to change or load the stored bit predictably.

9.2.1 SR Latch

Unlike the simple pair of cross-coupled inverters, the SR latch adds **inputs** that allow us to force the stored value to 0 or 1. It is built from **two cross-coupled NOR gates**:



Each NOR gate feeds its output back into the other gate, forming the same kind of feedback loop that provides bistability. But now each gate also receives an external input:

- **S** (Set), which tends to drive the output Q to 1
- **R** (Reset), which tends to drive the output Q to 0

Because of these new inputs, we can deliberately control the internal state of the latch. Recall that a NOR gate outputs 1 only when **all** of its inputs are 0. If **any** input is 1, the output becomes 0. This property is the key to understanding how the Set and Reset inputs control the latch.

Case 1: ($R=1, S=0$), reset the latch

- N1 receives $R=1$. Since one of its inputs is TRUE, the NOR output must be FALSE, so $Q=0$.
- N2 receives both $S=0$ and $Q=0$. Both inputs are FALSE, so the NOR output becomes TRUE, meaning $\bar{Q}=1$.

This result is consistent and stable. The latch ends in the state:

$$Q = 0, \quad \bar{Q} = 1$$

The latch has been **reset**.

Case 2: ($R=0, S=1$), set the latch

- N1 receives $R=0$ and \bar{Q} , which is still unknown at this point, so we cannot determine Q yet. However, we can first look at gate N2.
- N2 receives $S=1$. Since at least one input to a NOR gate is TRUE, its output becomes FALSE, so $\bar{Q}=0$.
- With $\bar{Q}=0$ now known, we revisit N1, which receives inputs $R=0$ and $\bar{Q}=0$. Both inputs are FALSE, so the NOR output becomes TRUE.

Again, the circuit reaches a consistent and stable state:

$$Q = 1, \quad \overline{Q} = 0$$

The latch has been **set**.

Now that we have seen how the SR latch behaves when either Set or Reset is asserted, we can examine the remaining input combinations.

Case 3: (R=0, S=0), hold state

- N1 receives input R=0 and not-Q, that we do not yet know.
- N2 receives inputs S=0 and Q, that we also do not yet know.

At first glance, the circuit looks **ambiguous**. However, we know that the latch must already be in one of its two stable states, either Q=0 or Q=1. So let us analyse both possibilities.

Assume Q=0

- N2 receives two FALSE inputs, so its output must be TRUE, meaning not-Q=1
- N1 receives one TRUE input, so its output must be FALSE, confirming Q=0

Assume Q=1

- N2 receives a TRUE input, so its output must be FALSE, giving not-Q=0.
- N1 receives two FALSE inputs, so its output must be TRUE, confirming Q=1

In both cases, the circuit remains stable and consistent. Therefore, when both Set and Reset are 0, the latch **retains its previous state**.

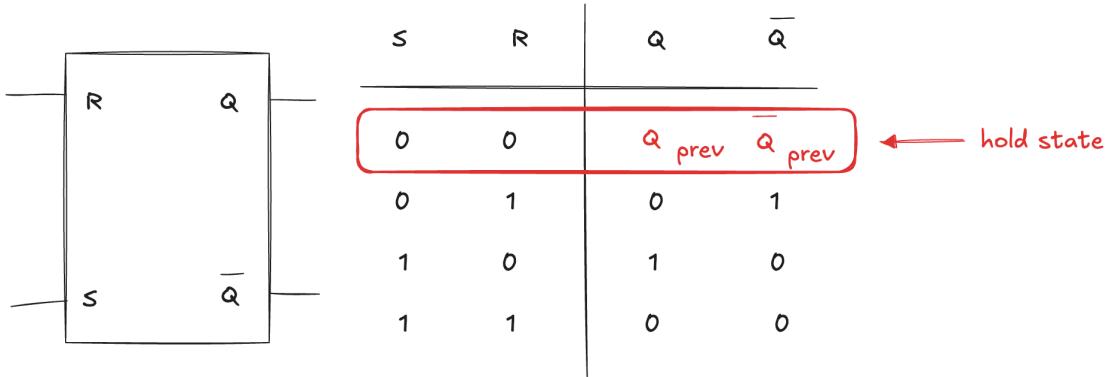
Case 4: (R=1, S=1), forbidden condition

Since N1 and N2 both see a 1, their outputs must go to 0. This means the circuit produces:

$$Q = 0, \quad \overline{Q} = 0$$

But this is **not valid**, because we expect the two outputs to be complements of each other. Moreover, when both R and S return to 0, the latch may unpredictably fall into either final state depending on tiny differences or noise. For this reason, it is considered an illegal or forbidden input combination, and real systems are designed to avoid it.

The behaviour of the SR latch can be neatly summarized using a truth table. This table captures the four combinations of the inputs S and R, and shows the resulting outputs Q. The table also reflects the latch's ability to remember its previous value:



Putting these observations together:

- When R=1 or S=1, the latch is forced into a known state.
- When R=0 and S=0, the latch simply remains in whatever state it was in before

This means the latch **stores one bit of information**. The combination R=0, S=0 is the "hold" condition: the latch **remembers** the last value that was written into it. This is the **fundamental idea behind memory** in digital systems.

We can use VHDL processes to describe the behaviour of the SR latch as follows:

```
entity SR_LATCH is
  port( S: in std_logic;
        R: in std_logic;
        Q: out std_logic
      );
end SR_LATCH;
```

The output may change in response to an event on either signal S or signal R. For this reason, both signals must appear in the sensitivity list:

```
architecture rtl of SR_LATCH is
begin
  latch: process(S, R)
  begin
    if( S='1' and R='0' ) then
      Q <= '1';
    elsif( S='0' and R='1' ) then
      Q <= '0';
    elsif( S='0' and R='0' ) then
      null;
```

```

    elsif( S='1' and R='1' ) then
        Q <= 'X';
    end if;
end process;
end rtl;

```

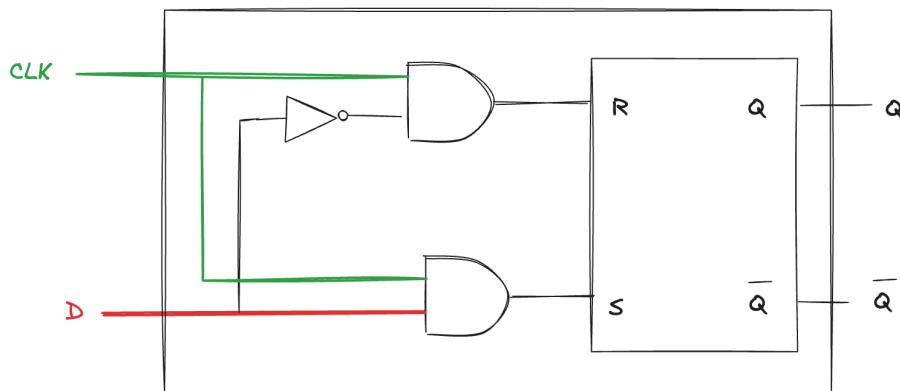
The behaviour of the SR latch depends on the values of the control signals S and R, and it is immediately sensitive to any event on either of them; otherwise, the latch preserves its current state. It is important to highlight a key aspect of the specification: when both control signals are zero, the latch, although enabled, must hold its previous value. To express this behaviour in VHDL we use the **null statement**, which explicitly denotes that no action is to be taken.

9.2.2 D Latch

The SR latch works, but it has two important drawbacks. First, it **behaves unpredictably** when both S and R are asserted at the same time, producing an invalid state. Second, the meanings of S and R **mix together two different ideas**:

- **What** value the latch should take (set or reset)
- **When** the latch should update its value

Because these two concepts are tied to the same pair of inputs, the SR latch is awkward to use in more complex designs. Digital systems are much easier to design when we separate the **data** we want to store and the **moment** when the storage should occur. This separation is exactly what the **D latch** provides:



The D latch avoids the invalid S=1, R=1 condition entirely by using a single **data input (D)** to tell the latch what the next value of (Q) should be, and a **clock input (CLK)** to tell the latch when it is allowed to change its state. Internally, the D latch transforms the single data input into appropriate S and R signals using two AND gates controlled by the clock:

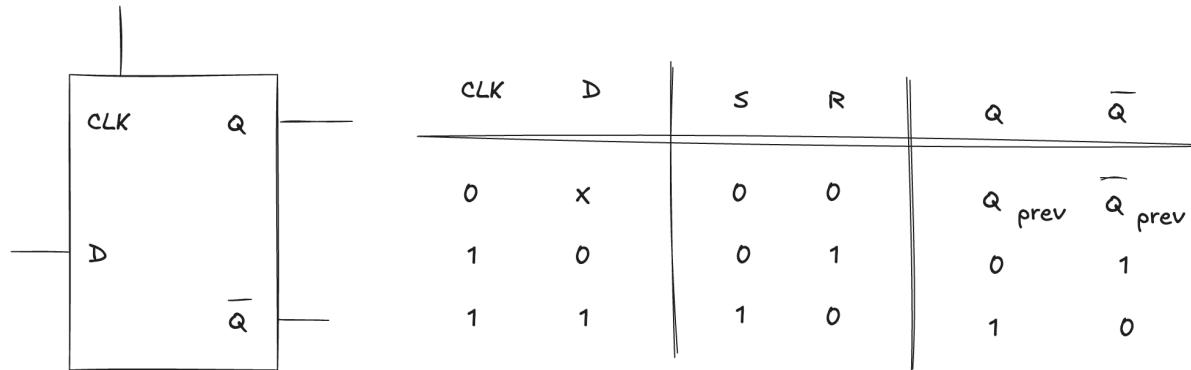
- When **CLK=0**:

- Both AND gates are disabled
- So both S and R are 0
- The latch is in the **memory mode**
- Q simply holds its previous value, regardless of D
- The circuit is effectively **closed** to new data

- When **CLK=1**:

- The value of D determines which AND gate becomes active
- If D=1 the upper AND produces S=1, R=0 and the latch is **set**
- If D=0 the lower AND produces S=0, R=1 and the latch is **reset**
- In either case, Q=D and the latch becomes **transparent** to the data

The truth table summarizing the D latch behaviour is:



When the **clock is low**, the latch **remembers** its previous value. When the **clock is high**, the latch **copies** whatever value is present on D directly to Q. This design completely eliminates the ambiguity of the SR latch, because the D latch never attempts to set and reset at the same time.

The entity declaration for the D latch is the following:

```
entity D_LATCH is
  port( D: in std_logic;
        CLK: in std_logic;
        Q: out std_logic
      );
end D_LATCH;
```

The output may change in response to an event on either signal D or signal CLK. For this reason, both signals must appear in the sensitivity list. We also note that the output is updated only

when the enable signal is high; when it is low, the output must remain unchanged. The resulting architecture is therefore the following:

```
architecture rtl of D_LATCH is
begin
    latch: process(D, CLK)
    begin
        if( CLK = '1' ) then
            Q <= D;
        end if;
    end process;
end rtl;
```

The first thing to notice is the absence of an else branch. This omission is **an error when the intention is to describe purely combinational behaviour**. The intuitive reason is the following: since the description does not explicitly state what should happen when the value of CLK is not equal to 1, the synthesis tool assumes that any signal assigned within the process (i.e., any signal appearing on the left-hand side of an assignment) must retain its previous value. This, in turn, **requires the presence of a memory element** capable of holding that value.

Summarizing, the D latch is the first memory element that cleanly separates **what to store** from **when to store it**, which is crucial for synchronous digital design.

9.3 Flip-Flops

In a D latch, the clock signal controls the moment when new data is allowed to flow into the storage element. When the clock is high ($CLK=1$), the latch becomes **transparent**: the value present on the D input passes directly to the output Q, almost as though the latch were just a buffer. Any changes in D immediately appear at Q for as long as the clock remains high. When the clock is low ($CLK = 0$), the latch becomes **opaque**: new data can no longer reach Q, and the latch instead holds and preserves the value it had at the instant the clock transitioned from high to low.

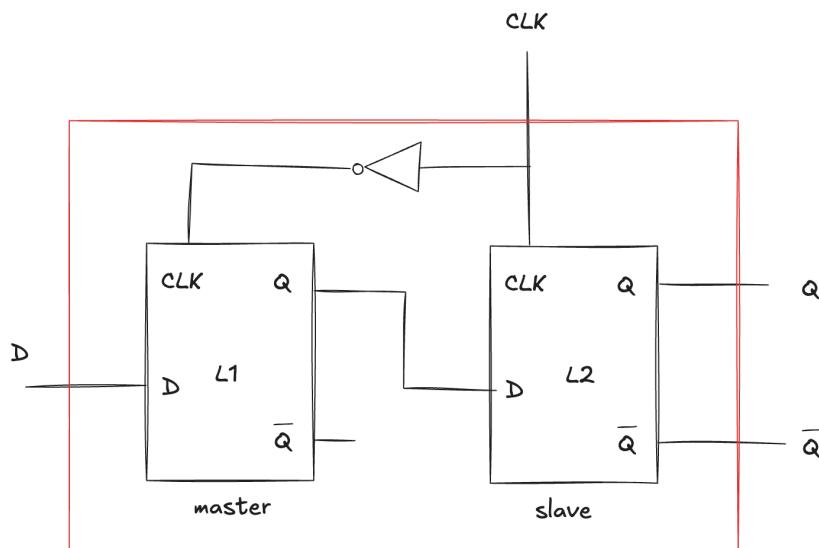
This simple mechanism gives the designer control over **when information is allowed to enter the latch**. However, this behaviour also introduces a **subtle timing problem** that is not obvious from the truth table. Real signals do not change instantaneously; every transition takes a finite amount of time to propagate through gates and wires. As long as the clock remains at 1, the latch is continuously transparent, meaning that **any small disturbance or glitch on the D input can propagate through the combinational logic and appear directly at Q**. If D changes near the moment when the clock transitions from high to low, the latch may briefly enter an unstable

region before finally settling. Depending on the exact timing, the latch might capture the "old" value, the "new" value, or even oscillate momentarily before stabilizing.

This characteristic is known as the **level-sensitive** nature of a latch: it responds to the **level of the clock**, not to a **precise instant**. This timing difficulty motivate the next refinement in sequential design: the **edge-triggered device** called **flip-flop**, which restricts the moment of state change to a **single, well-defined event** (the **rising or falling edge of the clock**).

9.3.1 D Flip Flop

The D flip-flop solves the timing problems that arise with level-sensitive latches by ensuring that the stored value changes **only at a single, well-defined instant**: the clock edge. Instead of responding throughout an entire interval when the clock is high (as the D latch does), the flip-flop updates its state only at the moment the clock transitions. This creates a clean boundary between one state and the next and prevents race-through effects. The flip-flop is built from **two D latches connected in series**, commonly referred to as the **master** and the **slave**:

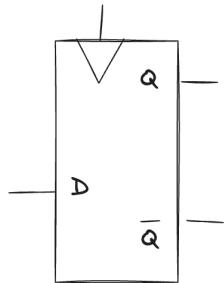


These two latches are controlled by **complementary clock signals**. When the clock is low, the master latch is transparent while the slave latch is opaque. This means that during the low phase, the master is free to track the D input, but the slave remains locked, holding the previous output value. When the clock switches to high, the situation reverses: the master becomes opaque, freezing whatever value it captured just before the transition, while the slave becomes transparent and copies the master's stored value to the output Q.

This arrangement has a crucial consequence: **the final output Q changes only at the moment the clock rises**. While the clock is low, the master may still be following D, but the slave cannot

see those changes. When the clock rises, the master closes instantly (blocking any further input changes), and exactly at that same moment the slave opens and captures the master's stable value. Because the slave sees only the value that was already settled inside the master at the clock edge, the output cannot be influenced by any glitches or delays that may occur after the transition. Between edges, the slave remains opaque, and the output stays constant.

This edge-sensitive behaviour is what defines the D flip-flop: **D determines what the next state will be, and the clock edge determines precisely when that new state becomes active.** The flip-flop therefore cleanly separates "what to store" from "when to store it", solving the ambiguity inherent in level-sensitive latches and making it possible to build synchronous circuits in which every memory element in the system updates in perfect lockstep. The schematic symbol for the D flip-flop emphasizes this event-driven nature:



The small triangle on the clock input marks it as an **edge-triggered device**. In a rising-edge D flip-flop—the most common variety—the triangle indicates **sensitivity to the low-to-high transition**. Regardless of what happens on D before or after that instant, only the value that is present exactly at the rising edge is captured and stored.

We now begin with the description of the entity for the D flip-flop

```
entity D_FF is
    port( CLK: in std_logic;
          D: in std_logic;
          Q: out std_logic
    );
end D_FF;
```

In the case of latches, the condition controlling the update of the output depended on the level of the CLK signal and could therefore be expressed simply by comparing that signal to the desired logic value (0 or 1). For flip-flops, the situation is more complex because the update must occur only in correspondence with a signal transition. To make this concrete, assume that the device is sensitive to the rising edge of the clock. In this case, two conditions must be checked:

- an event has occurred on the clock signal

- the value of the clock after the event is '1'

To express the occurrence of an event, however, we must use a specific VHDL construct based on the notion of a **signal attribute**. In this case, we need to verify that the event attribute of the clock signal is true. The combined condition is therefore written as follows:

```
if(CLK'event and CLK = '1') then
  ...
end if;
```

Since the device must preserve its state in the absence of a rising edge on the clock signal, the else branch of the if statement must again be omitted. It is also clear that the data signal D should not appear in the sensitivity list of the process, because the flip-flop's state is not affected by events on that signal. Based on these considerations, we can now proceed to specify the architecture:

```
architecture rtl_rising_edge of D_FF is
begin
  ff: process(CLK)
  begin
    if(CLK'event and CLK='1') then
      Q ≤ D;
    end if;
  end process;
end rtl_rising_edge;
```

If we wish to describe a D flip-flop that is sensitive to the **falling edge**, it is sufficient to check that the value of the clock signal after the event is '0' rather than '1', that is:"

```
architecture rtl_falling_edge of D_FF is
begin
  ff: process(CLK)
  begin
    if(CLK'event and CLK='0') then
      Q ≤ D;
    end if;
  end process;
end rtl_falling_edge;
```

A better way to check the rising edge of the clock is to use the **rising_edge function** provided by the ieee.std_logic_1164 package:

```

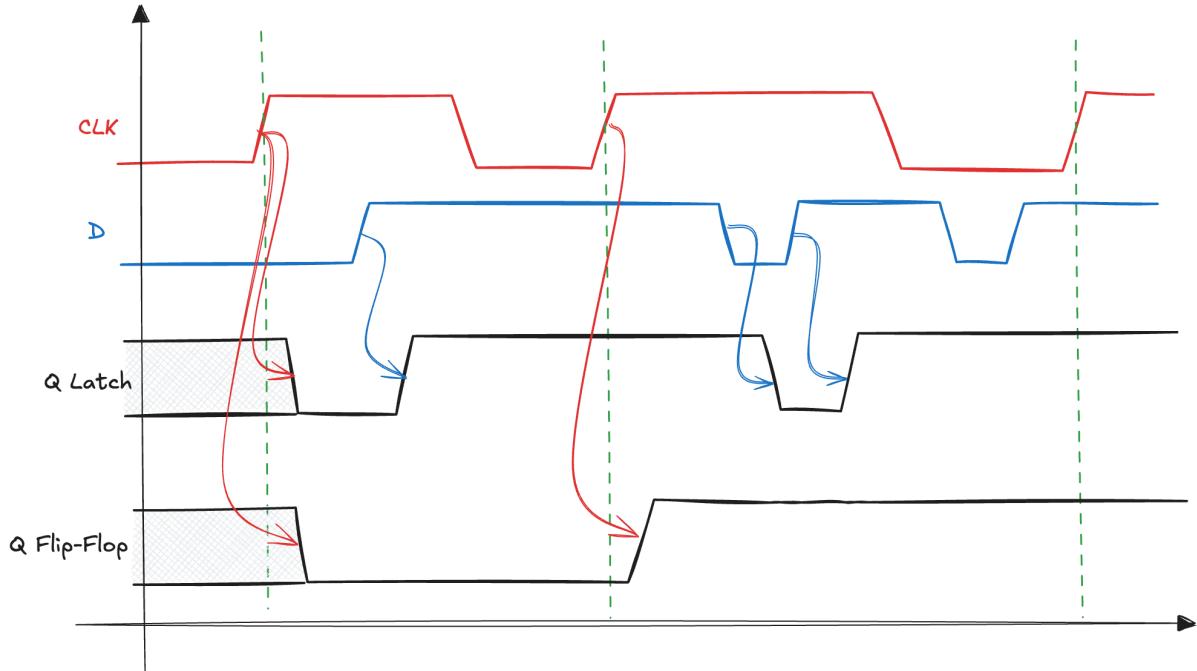
state_register : process(CLK)
begin
  if rising_edge(CLK) then
    Q <= Q_next;
  end if;
end process;

```

The use of `rising_edge(CLK)` is preferred today because it is the IEEE-standard, safer, and more robust way of detecting a clock edge. It avoids glitches, handles `std_logic` correctly, improves synthesis compatibility, and produces clearer and more portable code. All major tools (Synopsys, Xilinx, Intel, Mentor, Cadence) detect it immediately as a clock and generate better timing reports.

9.3.2 Flip-Flops and Latches comparison

The behaviour of a D latch and a D flip-flop becomes especially clear when we apply the same D and CLK waveforms to both devices and observe how their outputs evolve over time:



At the beginning of the sequence, the output Q is unknown for both elements, since we have not yet applied a clock event or an initial condition. What happens next, however, depends entirely on whether the storage element is level-sensitive or edge-triggered.

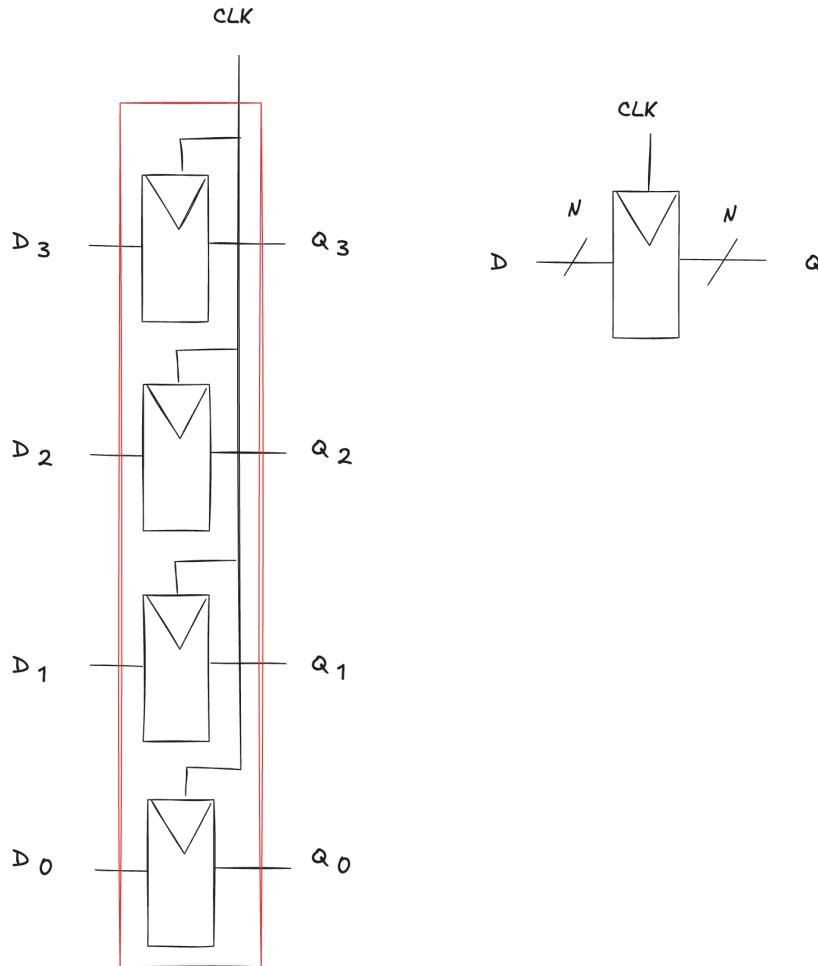
The D latch responds whenever the clock is high. At the first rising transition of CLK, the latch becomes transparent, and because D is low at that moment, Q immediately drops to zero. As long as CLK remains high, the latch continues to follow any changes on D. If D wiggles or produces glitches while the clock is high, those same variations appear at the latch output. When the clock falls back to zero, the latch becomes opaque and stops responding to input changes; whatever value Q held at that moment is preserved. If D changes while CLK is low, the latch ignores it completely.

The flip-flop behaves very differently. Being edge-triggered, it samples the input only at discrete instants: specifically, on the rising edge of the clock. At each of these clock edges, the value present on D is captured and transferred to Q. Between edges, the stored value remains stable, even if D changes multiple times or glitches. The output therefore changes only at well-defined moments, giving a clean, staircase-like waveform with no intermediate transitions.

By comparing the two signals, we can see the essential difference. The latch is controlled by the **level** of the clock and is therefore transparent whenever the clock is high; this makes it sensitive to the exact timing of D changes during that interval. The flip-flop, by contrast, compresses the entire clock-high interval into a **single event** (the rising edge) which eliminates uncertainty and prevents unintended propagation of input transitions. This contrast explains why latches can lead to timing hazards in large synchronous systems, whereas flip-flops provide the predictable, edge-defined behaviour needed for reliable digital design.

9.3.3 Registers

Once we have a reliable edge-triggered flip-flop, we can begin to assemble **more powerful storage structures**. The simplest and most fundamental of these is the **register**. An N-bit register is nothing more than a collection (or **bank**) of N flip-flops placed side by side. Each flip-flop stores one bit, and all of them share the **same clock signal**. This shared clock is what gives the register its defining property: all bits update **simultaneously** on the same clock edge.



This arrangement allows a register to capture an entire multi-bit value in one atomic operation. When the clock edge arrives, every flip-flop in the bank samples its corresponding input bit and updates its output. Between clock edges, the outputs remain stable, holding the stored N -bit word. In this way, the register acts as a discrete-time storage element: it updates only at well-defined instants and remains unchanged at all other times.

The VHDL description of an N -bit register is straightforward. We define an entity with N -bit input and output ports, and inside the architecture we instantiate N flip-flops, all driven by the same clock signal. The process that describes the behaviour of the register looks like this:

```
entity Register is
    generic(N : integer := 8);
    port(CLK : in std_logic;
         D   : in std_logic_vector(N-1 downto 0);
         Q   : out std_logic_vector(N-1 downto 0)
    );

```

```

end Register;

architecture structural of Register is
    component D_FF is
        port(CLK : in std_logic;
             D   : in std_logic;
             Q   : out std_logic
        );
    end component;
begin
    GEN_FF : for i in 0 to N-1 generate
        FF_i : D_FF
            port map(CLK => CLK, D => D(i), Q => Q(i));
    end generate GEN_FF;
end structural;

```

Observe that the **for-generate loop** is synthesizable because it does not describe an iterative behavior in time. Instead, it describes multiple copies of hardware that the synthesizer can fully expand at compile time.

Registers play a central role in almost every sequential digital system. They hold counters, intermediate results in arithmetic units, configuration values, memory addresses, instructions, pipeline state, and much more. In fact, the internal state of a processor at any moment can largely be described as the contents of its various registers. Because they update synchronously and predictably, registers make it possible to design complex systems in which data moves from one stage to the next in a coordinated rhythm dictated by the global clock.

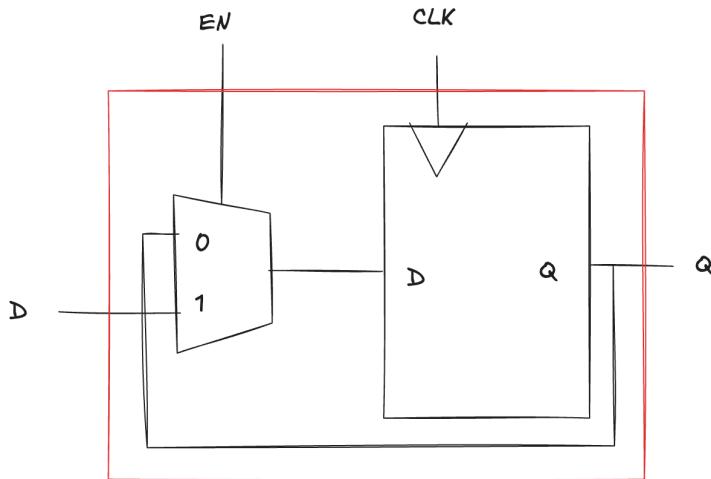
9.3.4 Enabled Flip-Flop

In many designs, it is unnecessary (or even undesirable) for a flip-flop to update its stored value at **every single clock edge**. For example, a counter should update only when a count-enable signal is active, a configuration register should change only when software writes to it; and pipeline registers should only accept new data during certain phases of execution. To support this behaviour, we add an additional input to the flip-flop: **Enable**.

The Enable signal determines whether the flip-flop should load a new data value on the next clock edge. When Enable is true, the flip-flop behaves just like an ordinary D flip-flop: the value on D is sampled at the rising edge of the clock and becomes the new output Q. When Enable is false, the flip-flop simply ignores the clock edge and retains its old state indefinitely. This

gives fine-grained control over when the stored value is allowed to change, while preserving the clean, edge-triggered timing behaviour of the underlying flip-flop.

There are two main ways to build such an enabled device. The most common and safest method inserts a **multiplexer** in front of the D input. The multiplexer chooses between two values: the external input D, or the flip-flop's own previous output Q. When Enable is true, the multiplexer forwards the external input so the flip-flop can update. When Enable is false, the multiplexer instead selects the old Q value, effectively "recirculating" the stored bit. Even though the clock still toggles, the flip-flop sees no change at its input and therefore maintains its previous state

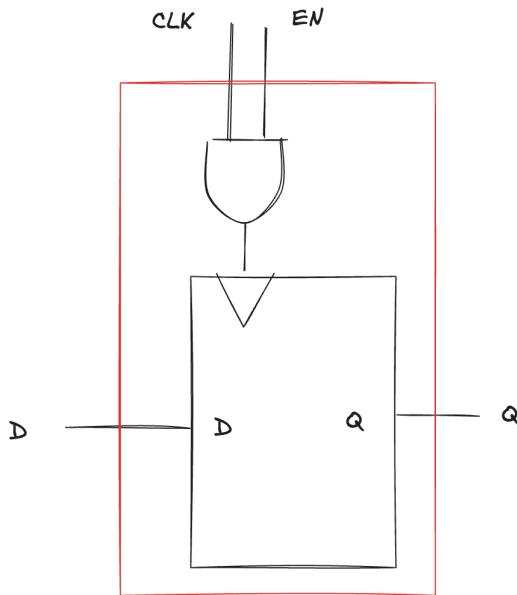


The VHDL description is:

```
entity DFF_en is
    port(CLK : in std_logic;
         EN  : in std_logic;
         D   : in std_logic;
         Q   : out std_logic
    );
end entity;
```

```
architecture rtl of DFF_en is
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            if EN='1' then
                Q <= D;
            end if;
        end if;
    end process;
end rtl;
```

A second method modifies the clock itself. In this approach the clock signal is AND-ed with the Enable signal. If Enable is true, the clock passes through normally; if Enable is false, the clock is forcibly held low, preventing any edge from reaching the flip-flop.



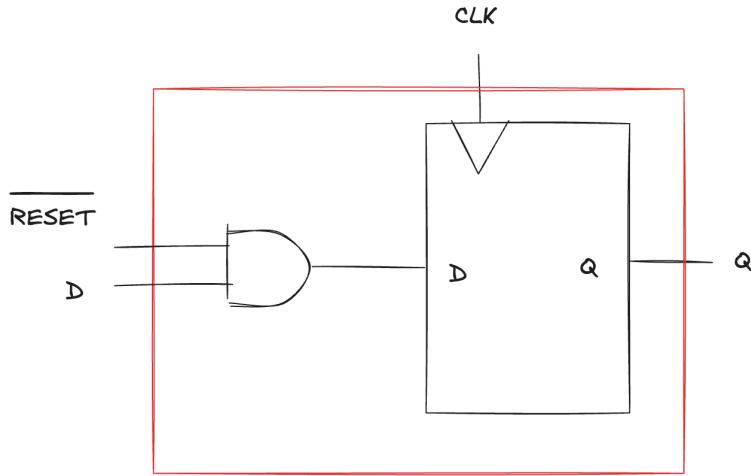
Although this approach seems elegant, it is **extremely delicate**: if Enable changes while the clock is already high, the AND gate can produce a **glitch**, a tiny unintended pulse that the flip-flop interprets as an additional clock edge. This can cause the device to capture data at the wrong moment, leading to subtle and difficult-to-debug timing errors. For this reason, performing any kind of logic directly on the clock signal is widely considered a bad practice in synchronous design.

The multiplexer-based implementation avoids these hazards entirely, because the clock remains clean and untouched. The enable logic influences only the input data path, not the timing reference of the entire system. This preserves the integrity of the clock domain and ensures predictable, reliable behaviour even in large systems. For these reasons, essentially all modern digital design tools and hardware description languages generate **mux-enabled flip-flops** by default when the designer describes an "if enable then load else keep" behaviour.

9.3.5 Resettable Flip-Flop

In a real digital system, it is often essential to start from a **known state**. When a processor powers on, all its registers and sequential elements are in random, unpredictable conditions, since each flip-flop may have settled into either 0 or 1 depending on tiny variations in the manufacturing process and electrical noise. Before the system can operate reliably, we need a way to force every flip-flop into a defined value regardless of the data that would normally flow into it. This is the purpose of adding a **Reset** input to a flip-flop. It behaves like a normal D flip-flop as long as Reset is inactive. But when Reset is asserted, the flip-flop ignores its data input entirely and

forces its output to 0. As soon as Reset is released, the device returns to ordinary operation, again sampling D only at the active clock edge.



To implement this idea, we can begin with a standard D flip-flop and simply AND the data input with the Reset signal. Because the Reset in this example is **active low**, it performs its reset function when it is 0. When Reset is high (inactive), the AND gate passes the original D input unchanged, and the flip-flop behaves normally. But when Reset is pulled low, the AND gate outputs a forced 0 regardless of the value of D, ensuring that the flip-flop will load a zero at the next clock edge. The choice of an active-low reset is purely a design convention. If preferred, we could just as easily add an inverter and create an **active-high** reset. What matters is that the designer and the system agree on the polarity and use it consistently.

This type of flip-flop is particularly useful at system start-up. By asserting Reset for a short time, we can guarantee that every sequential element begins in a clean and predictable state. Complex digital systems (CPUs, controllers, memory interfaces) universally rely on this mechanism to **avoid undefined behaviour on power-up**. We can describe the behaviour of a resettable D flip-flop in VHDL as follows:

```
entity DFF_reset_low is
    port(CLK      : in  std_logic;
         RESET   : in  std_logic;  -- active low
         D       : in  std_logic;
         Q       : out std_logic
    );
end entity DFF_reset_low;
```

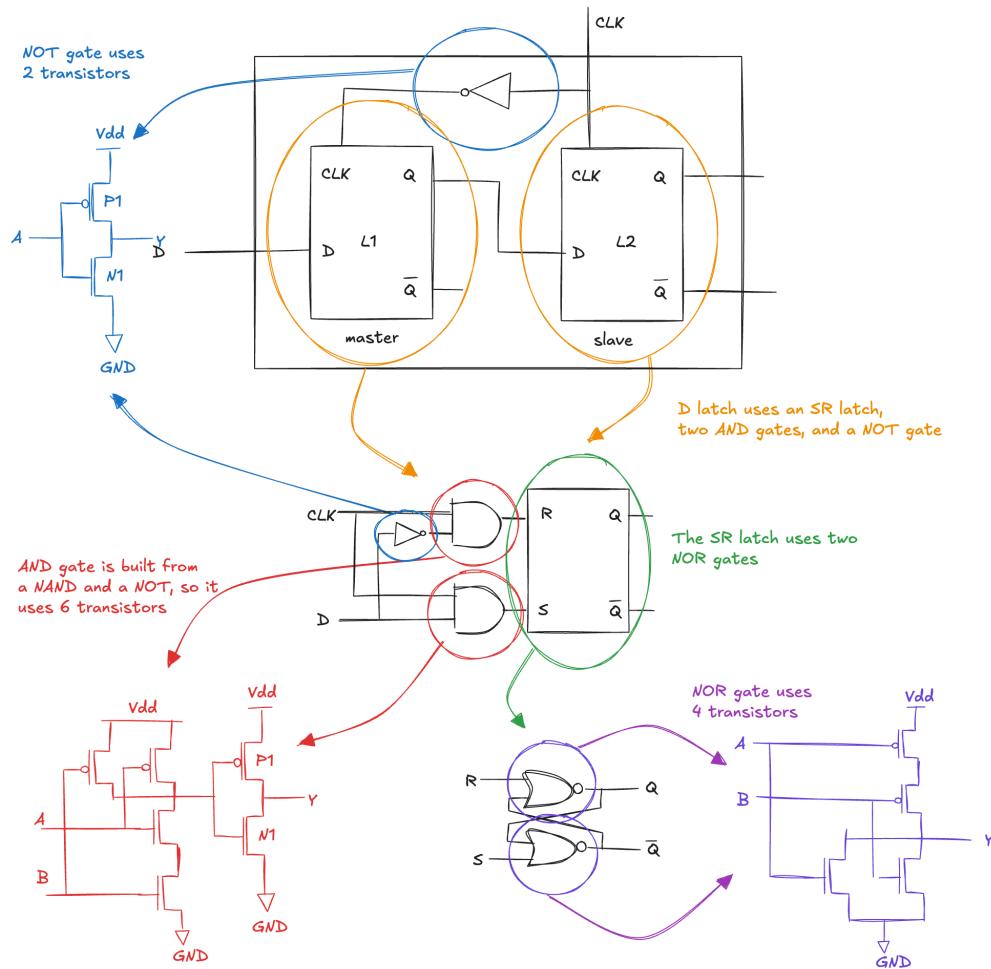
```
architecture rtl of DFF_reset_low is
    signal D_int : std_logic;
begin
    -- Active-low reset implemented via AND gate
    D_int <= D and RESET;

    process(CLK)
    begin
        if rising_edge(CLK) then
            Q <= D_int;
        end if;
    end process;

end architecture rtl;
```

9.4 Transistor level implementation

Considering the D Flip-Flop impplementation using logic gate, we can try to estimate the number of transistors required for its construction, recalling the transistor implementation of the basic logic gates.

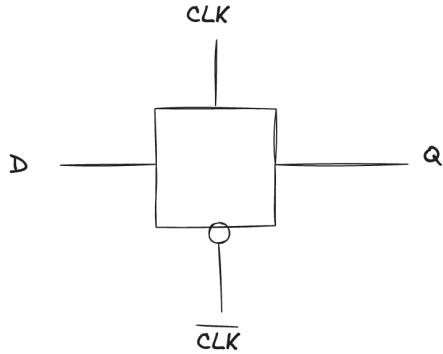


The D flip-flop can be decomposed into two D latches arranged in a master–slave configuration, together with an inverter that generates the complementary clock. Each D latch consists of an SR latch, two AND gates, and a NOT gate. The SR latch itself is built from two NOR gates, each implemented with four transistors, giving eight transistors in total. Each AND gate is realised by combining a NAND gate and an inverter, requiring six transistors per AND gate, for a total of twelve. The NOT gate contributes two more. A single D latch therefore uses twenty-two transistors. Since the flip-flop contains two such latches, the pair contributes forty-four transistors, and the additional clock inverter adds two more. Altogether, the complete D flip-flop uses **46 transistors**.

This number is extremely high and the flip-flop can be considered a **basic building block**, so it is needed to find more efficient implementations, **going beyond the digital abstraction** and exploiting the electrical characteristics of transistors to reduce the overall count.

Looking at the full CMOS implementation of a D flip-flop, it becomes clear that a conventional gate-level design requires many transistors. Yet the essential behaviour of a latch is conceptually

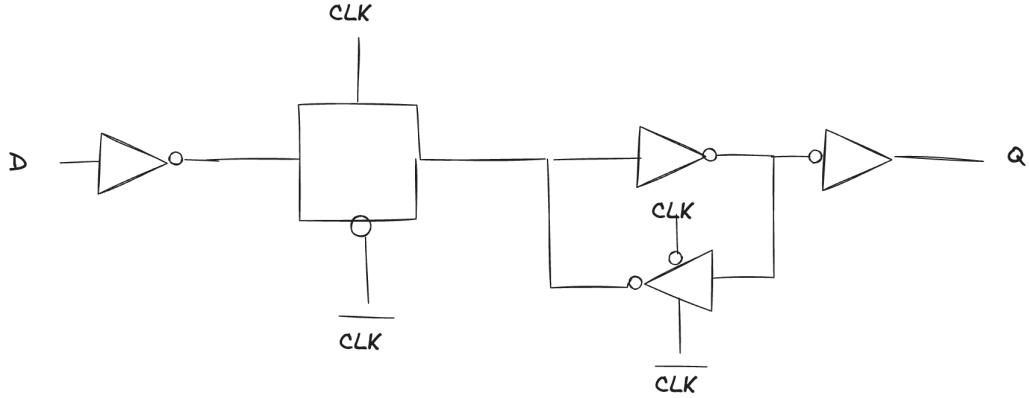
quite simple: it needs to be **transparent** when the clock allows data to flow in, and **opaque** the rest of the time, so that the stored value is preserved. In spirit, it behaves like a controlled switch. This observation leads naturally to a **much more compact transistor-level design**. Instead of constructing the latch from NOR gates, AND gates, and inverters, we can build a minimal D latch using a **single transmission gate**:



It is essentially a bidirectional switch implemented with one nMOS and one pMOS transistor in parallel, controlled by complementary clock signals. When the clock is high, the switch is on, allowing the input D to pass directly to the internal storage node Q, making the latch transparent. When the clock is low, the switch turns off, isolating Q from D and leaving the latch opaque. Although elegant and efficient, this minimal design suffers from two important limitations. The first is the presence of a **floating node**. When the transmission gate is off, the internal storage node is not actively driven by any gate; it merely holds whatever charge was left on its capacitance. Over time, leakage currents and noise can disturb this charge, gradually corrupting the stored value. In a practical system, this makes the latch unreliable unless it is frequently refreshed. The second limitation is the lack of any **buffering** around the storage node. Without buffers, the input and the internal node are directly connected (when transparent), and noise can easily force the transistors into conduction even when the clock says they should be off. A small negative glitch on D, for example, can accidentally turn on the nMOS device, briefly making the latch transparent even though the clock is low. Similarly, a voltage spike above VDD can activate the pMOS device. Because of this sensitivity, neither the internal state node nor the transmission-gate input should ever be exposed to external circuitry where uncontrolled noise might appear.

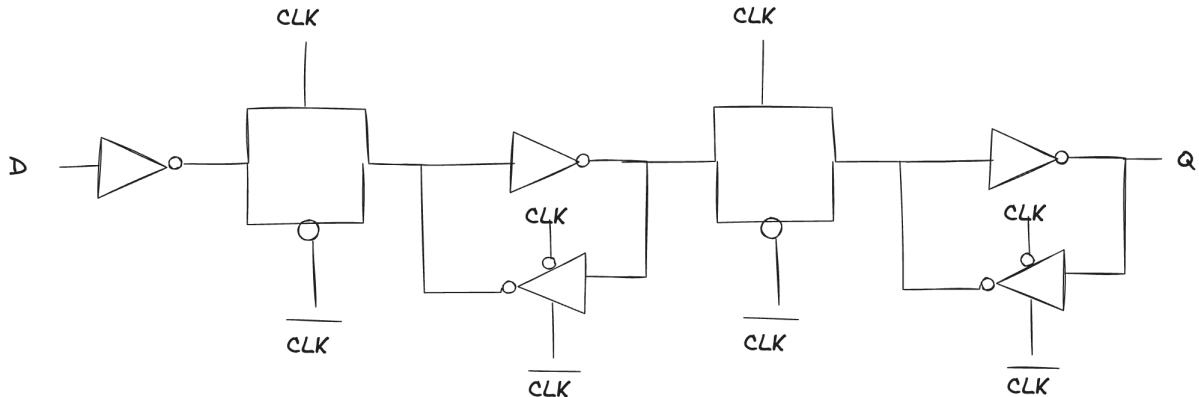
These issues explain why real digital systems almost never use this ultra-compact transmission-gate latch on its own. Instead, practical latches and flip-flops add buffers, feedback, and well-controlled gating logic to ensure that the stored value remains stable and immune to noise, even in the presence of long clock intervals or electrical disturbances. The transmission-gate concept remains useful, but only as one component inside a more robust sequential element.

A real implementation of a reliable D latch that overcomes these limitations combines the transmission gate with a pair of inverters connected in a feedback loop:



Although the internal details of this optimized latch involve subtle transistor-level techniques that fall outside the scope of an introductory course, the essential idea is straightforward: the transmission gate performs the switching action, while the inverters ensure that the stored value is actively driven and protected from leakage or floating-node disturbances. The latch therefore remains compact without sacrificing reliability.

When we place two of these improved D latches in the familiar master–slave arrangement and include the small amount of additional gating needed to generate the complementary clock, the complete D flip-flop ends up requiring only **twenty transistors**:



This is a dramatic reduction compared to the forty-six transistors needed when the flip-flop is built from logic gates alone. It also illustrates how transistor-level circuit design can achieve significant efficiency improvements beyond what is visible in gate-level diagrams.

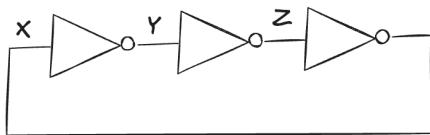
In short, commercial flip-flops rely on clever transistor-level constructions that are far more compact than their gate-based counterparts, reducing area, power, and delay—while preserving the essential behaviour of a D flip-flop used throughout synchronous digital systems.

9.5 Problematic sequential circuits

While the disciplined use of synchronous sequential circuits greatly simplifies design and analysis, certain configurations can still lead to unexpected or undesirable behaviour. Two common problematic

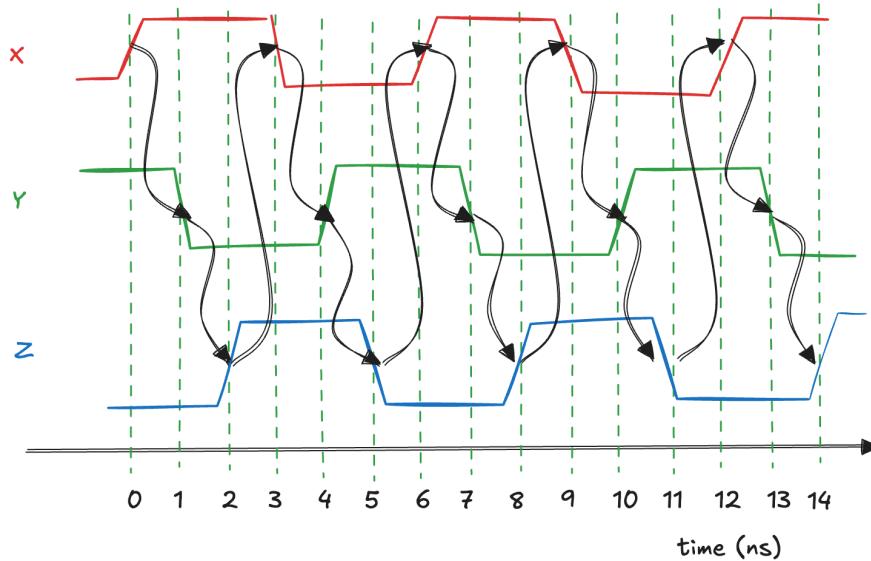
9.5.1 Astable circuits

A particularly interesting behaviour emerges when we connect an odd number of inverters in a loop:



At first glance, the circuit seems trivial: each inverter simply produces the complement of its input. But once placed in a ring, the three inverters begin to interact in a way that prevents the system from ever reaching a consistent, stable configuration. If we assume that the first node, X, starts at zero, the next inverter immediately forces Y to one, the third sets Z to zero, and the final inverter loops back to drive X to one, contradicting our initial assumption. No matter which starting value we pick, the logic "chases itself" and overturns the previous value. The circuit therefore has **no stable states**, and is called **astable**.

What actually happens in practice is that the signals begin to oscillate. Each inverter introduces a small propagation delay (e.g. one nanosecond), so a change at X takes time to ripple through Y and Z before coming back to X. If X rises at time zero, Y will fall one nanosecond later, Z will rise at two nanoseconds, and X will fall at three nanoseconds. This falling transition then propagates in the same fashion: Y rises at four nanoseconds, Z falls at five nanoseconds, and X rises again at six nanoseconds. Once the first cycle completes, the pattern simply repeats.

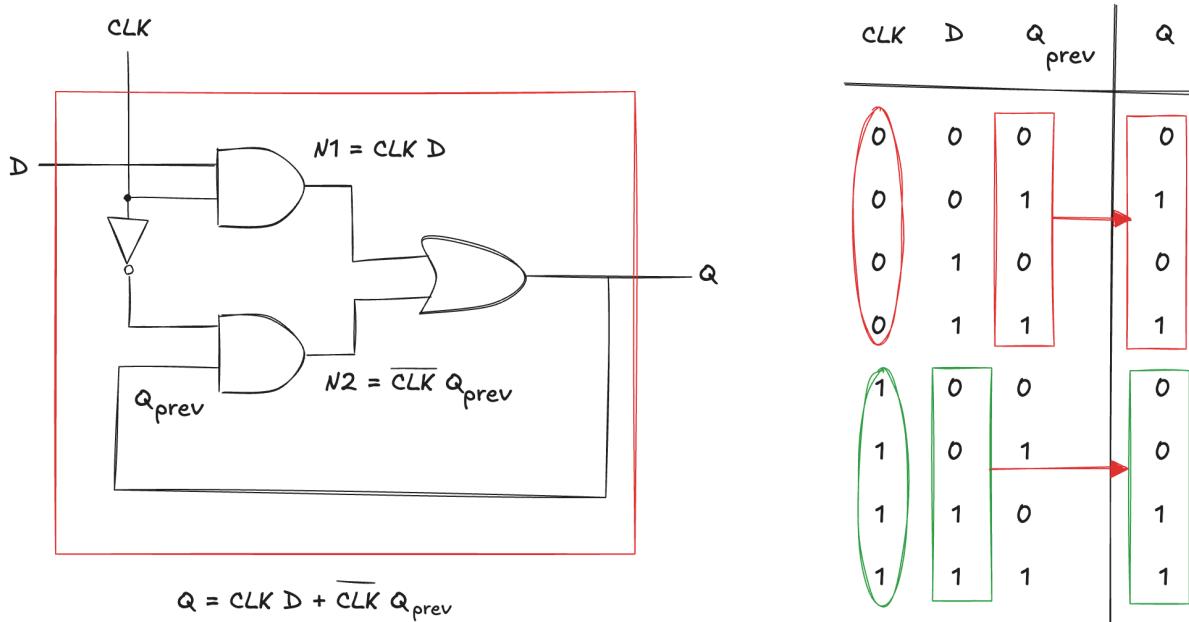


Every node in the loop alternates periodically between zero and one, forming what is known as a **ring oscillator**. In this three-stage example, the period is twice the total delay through the loop, giving a six-nanosecond oscillation.

Although elegant in its simplicity, the behaviour of this oscillator depends entirely on the actual propagation delays of the inverters, which vary with supply voltage, manufacturing conditions, and temperature. Even tiny changes in the physical environment can stretch or shrink the delay of each gate, and therefore alter the oscillation frequency. For this reason, the exact period of a ring oscillator is extremely difficult to predict accurately. Despite this unpredictability, ring oscillators are widely used as clock sources, randomness generators, and test structures, precisely because their behaviour arises from the intrinsic delays of the underlying technology.

9.5.2 Race conditions

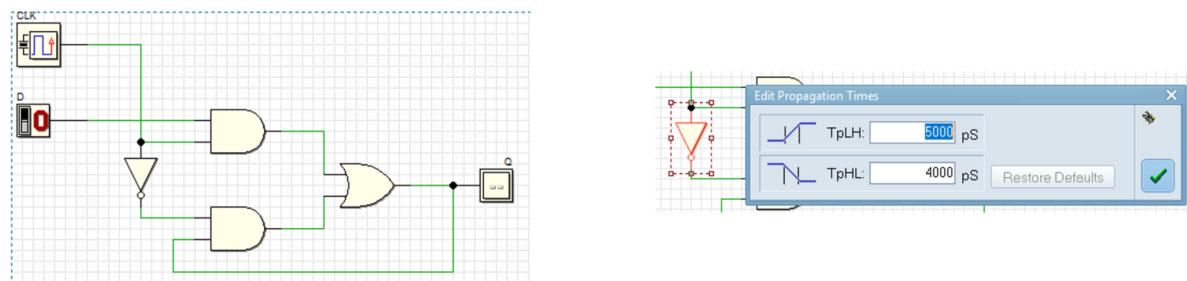
Another important timing hazard in digital systems is the **race condition**. This occurs when the output of a combinational circuit depends not only on the logical values of its inputs, but also on the precise timing of when those inputs change. If two or more input signals change nearly simultaneously, small differences in their arrival times can cause the output to take different paths through the logic, leading to unpredictable or incorrect results. Consider the following example, at first glance, the circuit appears to be a very compact and elegant implementation of a D latch:

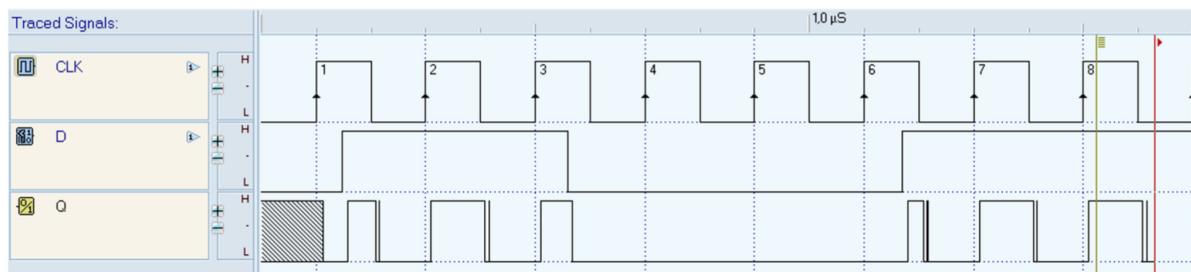


It uses fewer gates than the conventional design, and its logic expression suggests exactly the behaviour we expect: when the clock is high, Q should follow D, and when the clock is low, Q should retain its previous value. The truth table reinforces this impression, matching the intended functionality for all combinations of CLK, D, and the previously stored output. However, the apparent simplicity hides a subtle and dangerous problem: the circuit contains a **race condition** that makes its behaviour unreliable when different gates experience different propagation delays:

- the inverter in the CLK path delays the transition to the lower AND gate,
- when the clock switches from 1 to 0, both AND momentarily become 0
- this creates a temporary drop of Q, which can propagate into the feedback and lock the latch into 0 permanently, or create glitches.

To see it we can simulate the circuit in DEES:





This behaviour is the hallmark of a race condition: **two signals that must arrive in a precise temporal order instead arrive out of sequence, allowing a transient glitch to determine the long-term state of the circuit**. Such timing hazards are intolerable. They make the circuit sensitive to minute variations in temperature, voltage, or manufacturing, and therefore fundamentally unreliable.

9.5.3 Asynchronous circuits

The sequential circuits we have examined so far share an important structural feature: they contain loops, or **cyclic paths**, in which the output of a logic gate is fed back (directly or indirectly) to its own input. Such feedback is **essential for building memory elements**, but it also introduces behaviour that depends sensitively on the precise timing of signals as they propagate through the gates. In a purely **asynchronous circuit**, the final outcome is determined by whichever path happens to respond first. If one gate switches slightly faster than the others, the circuit may settle into one behaviour; if another path happens to be faster (perhaps because of manufacturing variation, supply voltage fluctuations, or even temperature changes) the circuit may settle differently. A design that "works" on the designer's desk may fail in another batch of chips, or at a different temperature, or under a different supply voltage. These **failures are notoriously difficult to diagnose**, because the circuit does not violate any logical rule: it is simply relying on unpredictable analog timing effects.

9.6 Synchronous digital design

To avoid the fragility of sequential circuits, modern digital design follows a **disciplined approach**: we **break the cyclic paths by inserting registers**, creating a system that alternates cleanly between combinational logic and well-defined storage elements. Each register holds the system's state, and that state is updated only on the active edge of the clock. Once a value has been captured, it does not change until the next clock event, no matter how long or short the delays inside the combinational logic might be.

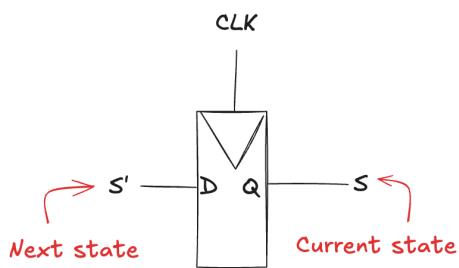
This transformation has a **profound stabilising effect**. Because the system only observes changes at clock edges, the detailed **ordering of internal transitions no longer matters**. As long as **the clock is slow enough to allow all combinational paths to settle before the next edge arrives**, every register in the system captures correct and consistent values. Races that previously depended on microscopic delay differences disappear, replaced by a predictable, repeatable behaviour driven by a single global timing reference.

This is the essence of **synchronous digital design**: by structuring a circuit as a sequence of combinational logic blocks separated by registers, we eliminate the chaotic and hard-to-debug behaviour inherent to asynchronous circuits, and replace it with a system whose behaviour is both dependable and scalable.

A synchronous sequential circuit is distinguished by **the presence of a clock input**, a signal whose rising edges define specific **instants at which the circuit is permitted to change its internal state**. Instead of reacting immediately to every change in its inputs, a synchronous system updates its state in a disciplined and predictable rhythm. The notions of current state and next state capture this behaviour: the current state describes the values stored in the circuit's registers at the present moment, while the next state is the set of values that will replace them at the next rising edge of the clock.

This structure leads to a set of powerful design rules. In a synchronous circuit, every element must be either a **combinational block**, which performs pure logic without memory, or a **register**, which holds part of the system's state. Crucially, there must be at least one register in any feedback loop within the design, ensuring that the circuit cannot respond instantaneously to its own output. All registers receive the same clock signal, so the entire system updates in lockstep. The clock thus acts as a **global coordinator**, slicing time into discrete steps during which combinational logic computes new values and registers simultaneously capture them.

The flip-flop represents the simplest possible synchronous sequential circuit. It has a single data input, a clock input, and one stored output:

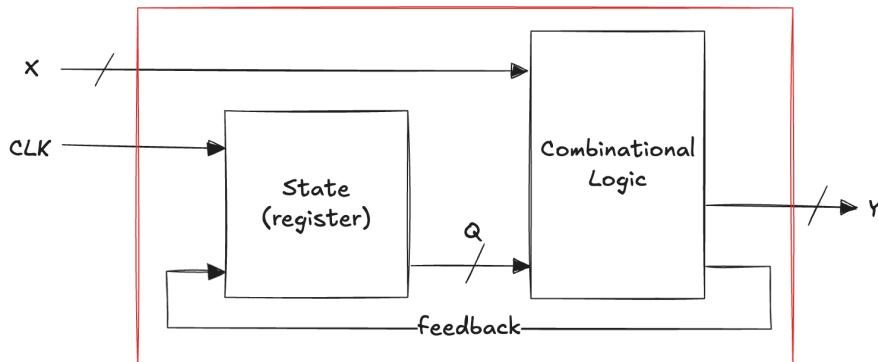


At each rising edge of the clock, the flip-flop samples the value on D and stores it as the new output Q, which becomes the circuit's current state until the next edge arrives. In this sense, the flip-flop embodies the fundamental synchronous operation: compute a next state and commit

to it only at a well-defined moment in time.

This disciplined approach is what makes synchronous circuits predictable, analyzable, and robust. By separating computation from state updates and tying all transitions to the clock, the designer gains control over timing, eliminates races, and ensures that the circuit behaves consistently despite the intrinsic delays of its individual gates.

The following diagram illustrates the **canonical structure** of a synchronous sequential circuit:



At its core lies a state register, a collection of flip-flops that collectively store the current state of the system. These flip-flops all share the same clock signal, ensuring that the stored state changes only at well-defined instants—the rising edges of the clock.

The behaviour of the circuit between clock edges is governed entirely by the combinational logic. This logic block receives two kinds of inputs: the external input signals and the current state coming from the register. Using these two sources of information, the combinational logic computes both the circuit's output and the next state, which is the value that will be loaded into the state register at the next clock edge.

Below is a general-purpose VHDL template for implementing the synchronous circuit:

```
entity SyncCircuit is
    generic(N : integer := 4    -- number of state bits);
    port(CLK : in std_logic;
         X  : in std_logic_vector(N-1 downto 0);  -- external inputs
         Y  : out std_logic_vector(N-1 downto 0)   -- outputs
    );
end SyncCircuit
```

architecture Behavioral of SyncCircuit is

```
-- State register
```

```

signal Q      : std_logic_vector(N-1 downto 0); -- current state
signal Q_next : std_logic_vector(N-1 downto 0); -- next state

begin

-- 1. COMBINATIONAL LOGIC:
comb_logic : process(X, Q)
begin
    -- default assignments (mandatory for combinational logic)
    Q_next <= Q;
    Y      <= (others => '0');

    -- INSERT COMBINATIONAL BEHAVIOR HERE
    -- Q_next <= some_function_of(X, Q);
    -- Y      <= some_other_function_of(X, Q);
end process;

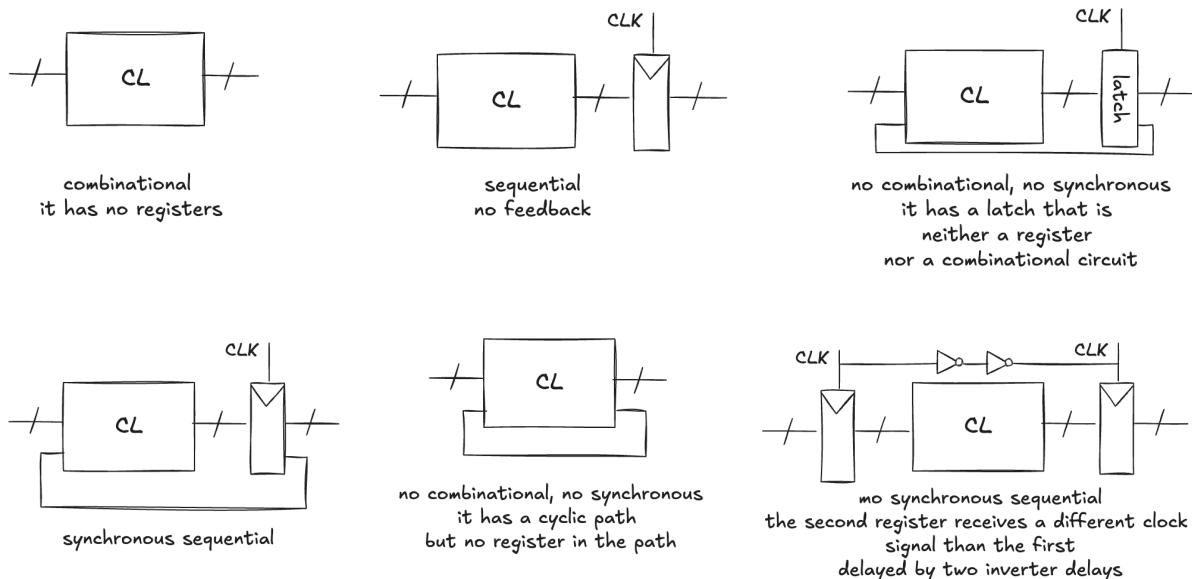
-- 2. SEQUENTIAL PROCESS:
state_register : process(CLK)
begin
    if rising_edge(CLK) then
        Q <= Q_next;
    end if;\

end process;

end Behavioral;

```

Summarizing, a crucial feature of synchronous design is the **feedback path** from the outputs of the combinational logic back to the state register. Although this looks like a cycle, the presence of the register breaks the path in time: the feedback does not take effect immediately, but only after the next clock event. This prevents the kind of uncontrolled oscillations and race conditions seen in asynchronous circuits. The combinational logic sees a stable state while it computes, and the register captures the newly computed values only when the clock permits it. The following examples illustrate how the presence or absence of registers and feedback determines whether a circuit is combinational, sequential, or something invalid from a synchronous-design perspective:



Although **synchronous design dominates modern digital systems**, asynchronous circuits remain an **active and fascinating area of research**. Their defining feature is the absence of a global clock: instead of waiting for periodic timing signals, asynchronous circuits move data forward whenever the necessary conditions are met. This simple conceptual shift of replacing **time-driven control** with **event-driven coordination** opens both significant opportunities and formidable challenges.

From a design perspective, asynchronous circuits promise several appealing advantages. Without a clock that must toggle continuously, they **consume power only when actual computation occurs**, making them inherently attractive for **ultra-low-power** applications. Their **speed is not constrained by a fixed clock period** but dictated by the **natural propagation delays** of the circuit's components. As a result, asynchronous systems can **adapt their performance to environmental conditions**: they speed up at high voltages and low temperatures, and slow down when conditions worsen, all without violating correctness.

Yet designing robust asynchronous circuits is far **more complex** than assembling their synchronous counterparts. The lack of a clock means that correctness depends on subtle timing relationships within the circuit. **Designers must reason about hazards, races, and the precise ordering of signal transitions**. Several design styles attempt to manage this complexity. **Delay-insensitive** approach, for example, use handshake protocols and special encoding schemes to guarantee correctness regardless of individual gate delays.

In recent decades, research has explored sophisticated asynchronous pipelines, advanced handshake circuits, and new forms of state encoding that reduce ambiguity in signal transitions. Tools, languages and various models have been developed to support asynchronous design

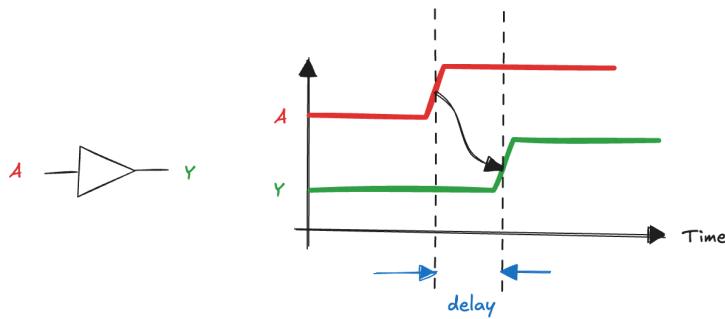
flows, though none has reached the industrial maturity of synchronous design tools. Nonetheless, asynchronous circuits have found practical use in niche but important domains: ultra-low-power sensor nodes, cryptographic hardware that resists attacks, energy-harvesting devices, and systems where variations in manufacturing or environment make a fixed clock unreliable.

Research continues to push the boundaries, exploring **hybrid systems** that combine local asynchronous regions within an otherwise synchronous framework, a compromise that captures many benefits of asynchrony while remaining compatible with mainstream tools.

10 Timing

One of the fundamental challenges in digital circuit design is **understanding and managing timing**. Even though we often describe logic gates as if they respond **instantaneously** to changes at their inputs, **real circuits always require a finite amount of time to react**. Making a circuit run fast is therefore not simply a matter of writing efficient logic equations; it ultimately depends on **how quickly every gate and wire in the system can respond to changing signals**. When the input of a gate changes, the output does not switch immediately. Instead, it takes a certain amount of time for the internal transistors of the circuit to react and settle to the new logical level. This interval is called **delay**, and it is an **inherent physical property of the technology** used to implement the circuit. In modern digital circuits, delays are extremely small, typically ranging from a few picoseconds (10^{-12} seconds) to several nanoseconds (10^{-9} seconds), but even these tiny times **accumulate** across many levels of logic and can **limit how quickly a digital system can operate**.

To visualize this behaviour, it is useful to look at a **timing diagram**. A timing diagram shows how signals vary over time, capturing the transient response of a circuit when the input changes. If we consider a simple buffer, the diagram will show the moment when the input begins to rise from a low voltage to a high voltage, a moment known as the **rising edge**:



The output also performs a rising transition, but noticeably later, demonstrating the presence of delay. Similarly, when the input transitions from high to low, the circuit exhibits a **falling edge**,

again followed by a delayed falling transition at the output. The arrows in the diagram highlight the dependency of the output transitions on the corresponding transitions at the input.

Delay is measured in a consistent and technology-independent way. Rather than measuring from the exact moment the voltage begins to change (which is often difficult to define precisely) we use standardized reference points. A rising edge is considered to occur at the moment the signal crosses the 50% voltage level between its low and high values. The output delay is therefore defined as **the time difference between the 50% point of the input transition and the 50% point of the corresponding transition at the output**. This convention allows engineers to compare delays across different circuits and technologies with clarity and precision.

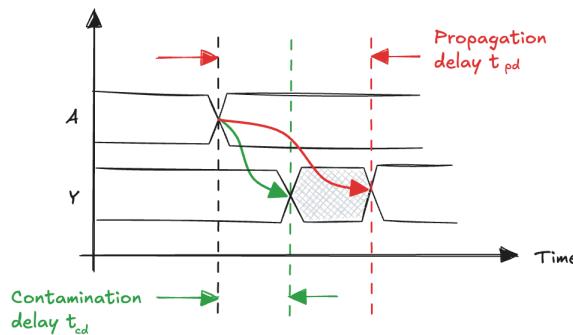
Understanding delay is the first step toward analysing more complex timing concepts such as propagation delay, contamination delay, setup and hold times, and ultimately the maximum clock frequency at which a synchronous digital system can safely operate.

10.1 Combinational Circuits Timing

In combinational circuits, timing plays a fundamental role because even the simplest logic operations require finite, non-negligible delays to propagate through gates, and understanding these delays is essential for predicting how quickly and reliably a digital system can respond to its inputs.

10.1.1 Propagation and Contamination Delays

When analysing how signals travel through digital circuits, it is not enough to say that a gate "has a delay". In practice, two different kinds of delay are important. The first is called **propagation delay** and it represents the **maximum** amount of time a gate may take to produce a valid output after its input has changed. In other words, if an input transitions at some instant, the output is guaranteed to have reached its final, stable value no later than a propagation delay after that moment. The second parameter is the **contamination delay** and it is the **minimum** amount of time between a change at the input and the moment the output begins to respond. Even the fastest internal transistor paths cannot cause the output to start changing before a contamination delay has elapsed. To understand these quantities more concretely, imagine that the input A of a gate initially holds a stable value, either HIGH or LOW. At a particular moment, A switches to the opposite logic level. For our purposes, we do not care whether the transition is rising or falling; what matters is simply that a transition occurs:



The output Y will react to this change, but not immediately. It will remain unchanged for at least the contamination delay. This ensures that any unintended early glitches cannot occur sooner than this minimum delay. After this initial interval, the output begins to move toward its new logical level. However, it does not reach that final level instantly. It takes some amount of time to complete the transition, and we know with certainty that it will have fully settled within the propagation delay. Thus, the output is guaranteed to be correct by the end of this maximum delay window.

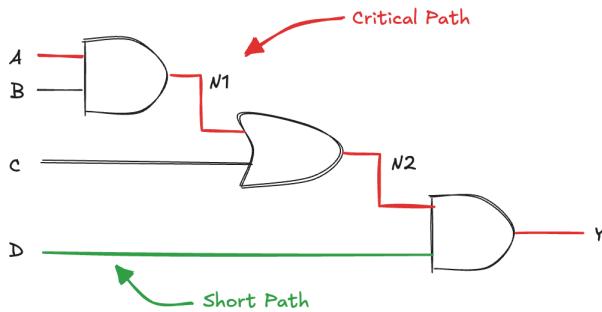
The timing diagram typically used to illustrate these ideas shows the input switching at a given time, followed by a shaded region representing the uncertainty interval during which the output may or may not have begun to change. The earliest possible change corresponds to the contamination delay, while the latest time at which the output must be stable corresponds to the propagation delay. This region highlights the fact that while we know the output cannot react too early, we also know it will not react too late.

The origin of these delays lies in the **physical behaviour of electronic circuits**. Every gate contains transistors and small parasitic capacitors, and these capacitances must charge or discharge before voltages can change appreciably. This charging process takes time, and the speed at which electric signals propagate through materials is ultimately limited by fundamental physical principles, including the speed of light in the medium. As a result, even the simplest logic gate exhibits both minimum and maximum timing characteristics.

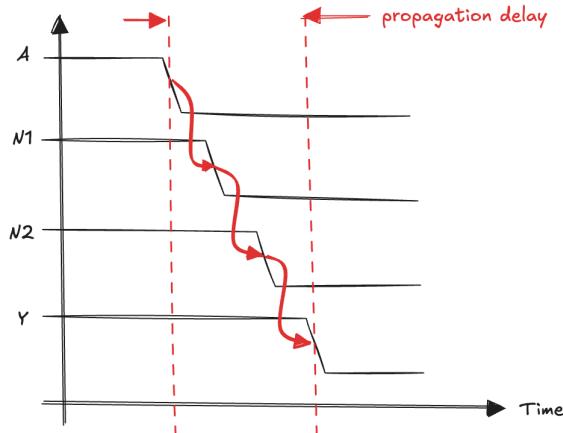
Accurately determining the values of propagation and contamination delays requires studying the internal transistor-level implementation of the gate, which is far below the level of abstraction we use when designing complex digital systems. Fortunately, we rarely need to perform these low-level analyses ourselves. Semiconductor manufacturers provide **data sheets** that specify the timing characteristics of each gate type under various operating conditions. By relying on these published parameters, digital designers can predict and verify the timing behaviour of much larger systems without ever needing to analyse their circuits at the transistor level.

10.1.2 Critical and Short Paths

Up to now we have considered delay as a property of individual gates, but real digital circuits are built by connecting many gates together. As a result, the total delay from an input to an output depends not only on the characteristics of each gate, but also on the specific **path** that a signal takes as it travels through the circuit. Some paths contain only a few gates, while others force the signal to propagate through several stages of logic. Since every gate introduces some delay, the overall timing behaviour of the circuit is governed by the lengths of these paths. For example, consider the following combinational circuit:



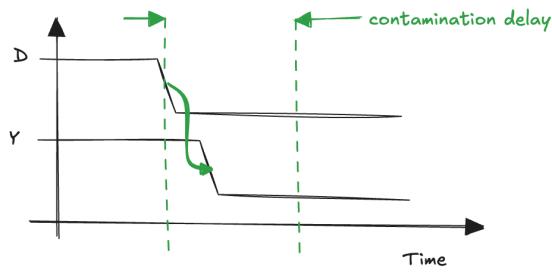
The most important of these paths is called the **critical path**. The critical path is simply the **longest or slowest** path from any input to any output. A signal that enters the circuit and must traverse this path will go through more gates, and therefore experience a larger total propagation delay than signals travelling along shorter paths. The length of the critical path plays a fundamental role in determining how fast the circuit can operate. Specifically, it sets an **upper bound on the clock frequency** of synchronous systems or, in combinational circuits, on how quickly valid outputs can appear after valid inputs are applied. The propagation delay of the entire circuit is obtained by taking the sum of the propagation delays of all the gates along the critical path. In the example, the critical path passes through two AND gates and an OR gate, the total propagation delay is simply the sum of the delays of these three components:



$$t_{pd} = t_{pd,AND} + t_{pd,OR} + t_{pd,AND}$$

This cumulative delay represents the worst-case situation: the longest time we must wait to guarantee that the output reflects a change at the input.

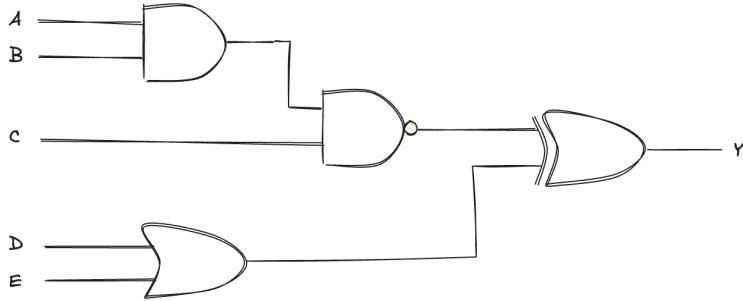
In contrast, the **short path** in a circuit is the **shortest** or **fastest** path from input to output. A signal travelling along this path encounters the minimum number of gates and therefore reaches the output more quickly than through any other route. The short path is important because it determines the **contamination delay** of the circuit. Just as with the propagation delay, the contamination delay of the circuit is computed as the sum of the contamination delays of the elements along the shortest path. For instance, in the example the short path includes only a single AND gate:



$$t_{cd} = t_{cd,AND}$$

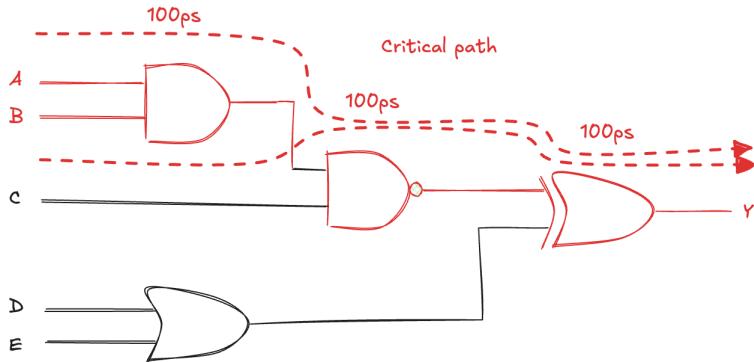
The timing diagrams associated with these paths help visualize how signals propagate through multi-stage logic. In the diagram illustrating the critical path, the input transition triggers a series of delayed transitions through intermediate nodes (N1 and N2 in the example), each step adding its own delay, until the final output settles. The total span of this chain corresponds to the propagation delay of the circuit. Conversely, the diagram for the short path shows a much quicker response: the output begins to change only after the minimum contamination delay has passed, reflecting the behaviour of the fastest possible route through the network.

As an example, consider the following circuit:



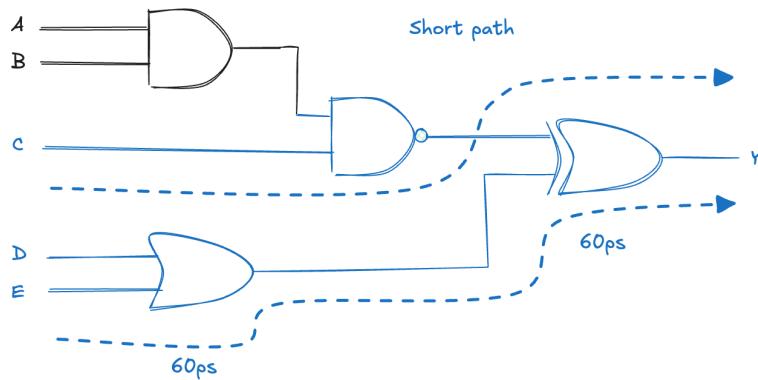
According to the datasheets, every gate in the circuit exhibits a propagation delay of 100 picoseconds and a contamination delay of 60 picoseconds. These values apply uniformly to all gates, regardless of their type. Our goal is therefore to trace the possible signal paths from each input to the output and sum the appropriate delays along those paths.

The longest route is taken by changes on inputs A or B. A transition on either of these inputs passes first through an AND gate, then through a second gate, and finally through the OR gate that produces the output Y. Since this path includes three gates in sequence, the total propagation delay is simply the sum of the propagation delays of these three stages:



$$t_{pd} = t_{pd,AND} + t_{pd,NAND} + t_{pd,XOR} = 100 + 100 + 100 = 300 \text{ picoseconds}$$

A change on inputs C, D, or E reaches the output through only two gates. For example, a transition on C travels through the middle gate and then the final OR gate, and similarly a transition on D or E goes through the lower OR gate followed by the output stage. In all cases this route consists of exactly two gates:



$$t_{cd} = t_{cd,OR} + t_{cd,XOR} = 60 + 60 = 120 \text{ picoseconds}$$

This example illustrates the general method for timing analysis in combinational circuits: identify the longest path to compute the propagation delay, identify the shortest path to compute the contamination delay, and use the individual gate delays supplied by the manufacturer to determine the overall timing behaviour. Be careful, because when individual gates have different propagation and contamination delays, the critical path and the short path **do not necessarily correspond to the paths with the greatest or smallest number of gates**. A path with fewer stages may still be slower if it contains gates with larger delays, and similarly a longer path may turn out to be the fastest if its gates are quicker.

Understanding both the critical path and the short path is essential for designing reliable digital systems. **The longest path limits the maximum operating speed, while the shortest path affects hazards, glitches, and certain timing checks in synchronous circuits.** Together, they form the foundation for timing analysis in combinational and sequential digital design.

10.1.3 Control-critical and Data-critical circuits

When analysing timing in combinational circuits, it is important to recognize that **not all critical paths are of the same nature**. In many designs, the signal that determines the overall speed constraint is not merely a data signal flowing through the datapath; it can also be a **control signal** that determines which data is selected, enabled, or combined. For this reason, we distinguish between **control-critical** and **data-critical** circuits.

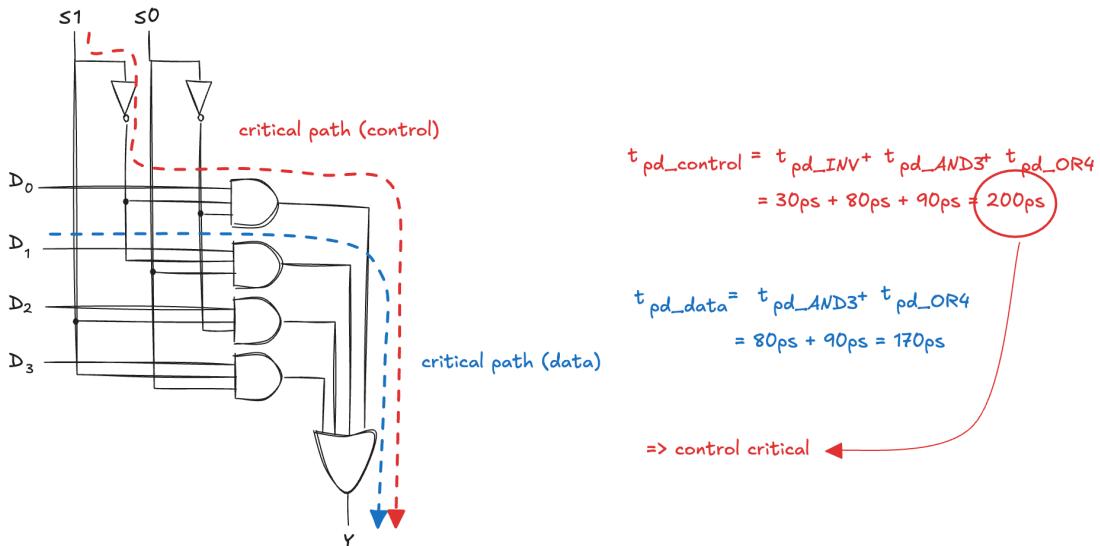
A circuit is said to be **control-critical** when the slowest path from any input to the output originates from the control signals. Control inputs such as select lines, enables, or mode bits may pass through several levels of logic before they influence the output. If control signals arrive later than the associated data signals, the output must wait for the control path to settle. In these situations, the design that minimizes the control delay is preferable, because speeding up the datapath provides no benefit if the output is still limited by a slow control transition.

Conversely, a circuit is **data-critical** when the slowest path begins at the data inputs. Here, the critical path runs through the portion of the circuit that processes or combines the actual data values. If the control signals arrive earlier, the controlling logic becomes ready quickly, and the output timing is determined entirely by how long the data takes to propagate through its gates. In such a case, reducing the data delay is the key priority, since optimizing the control path would not improve overall performance.

To illustrate these ideas, consider the two alternative implementations of a four-input multiplexer using a technology with known gate delays:

Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
Tristate (A to Y)	50
Tristate (Enable to Y)	35

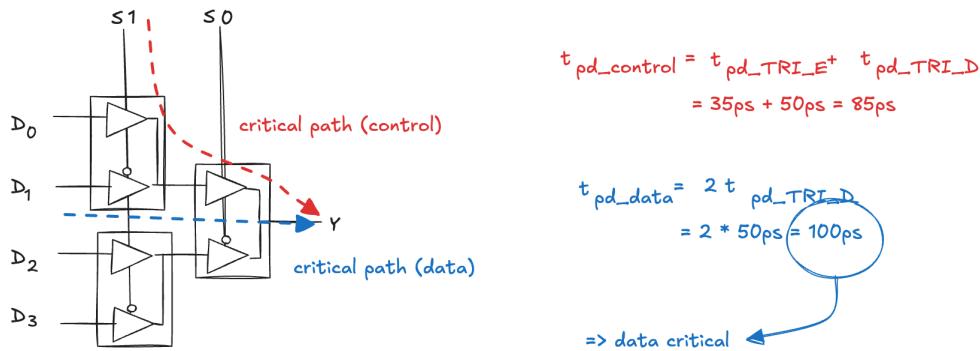
The first design is a two-level logic implementation built from AND, OR, and inverter gates.



The control lines S1 and S0 must propagate through an inverter, then through a three-input AND gate, and finally through a four-input OR gate before influencing the output. Summing the individual delays provided in the datasheet gives a control-path delay of about 200 ps. The

data inputs, by contrast, pass through fewer gates—only a three-input AND followed by the four-input OR—resulting in a data delay of roughly 170 ps. Since the control path is slower, this design is **control-critical**.

The second design uses multiplexer logic built from **tri-state buffers**. A tri-state buffer is a digital device that outputs a 0 or 1 when enabled, but enters a high-impedance state when disabled, effectively disconnecting itself from the circuit so multiple components can safely share the same signal line.



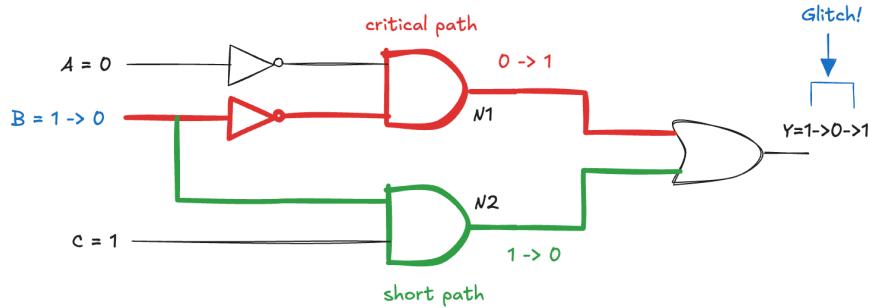
In this structure, the control signals enable the appropriate tri-state driver and then pass through an additional tri-state stage before reaching the output. The total control delay in this implementation is around 85ps, while the data path includes two tri-state elements in series, giving a longer data delay of about 100ps. Here, the slowest signal originates from the data inputs, not the control signals, meaning the design is **data-critical**.

These two examples demonstrate that the choice between logic-level implementations and multiplexer-based versions is not determined solely by correctness but by which timing behaviour better matches the requirements of the system. Depending on whether control or data signals are expected to arrive first, one implementation may offer a significant advantage. Ultimately, the best design is the one that balances timing with power consumption, cost, and the availability of components.

10.1.4 Hazards and Glitches

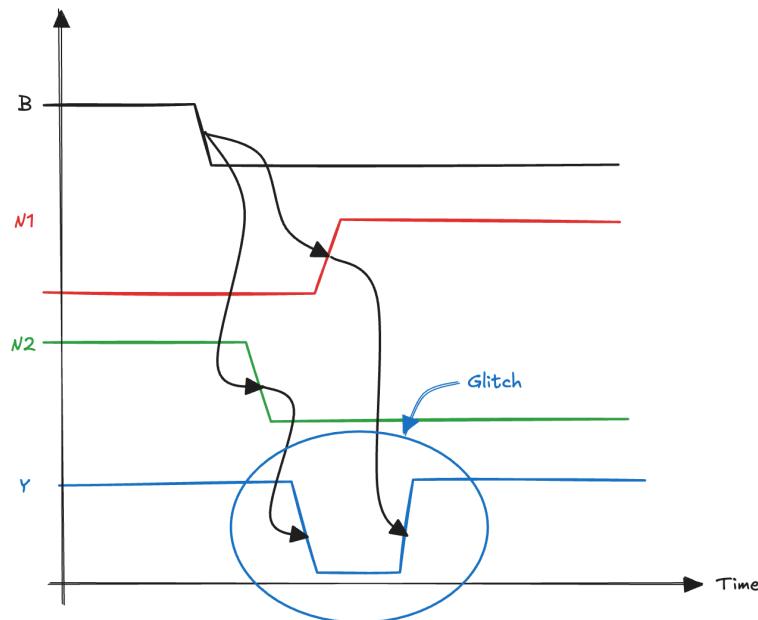
Even when a logic function is correct and its Boolean expression is perfectly minimized, the physical circuit may still behave unexpectedly during input transitions. This happens because the implementation is subject to real propagation delays, and these delays can differ along the various paths that connect the inputs to the output. When such differences exist, the circuit may be susceptible to a **potential timing problem** called a **hazard**: it indicates that, during an input change, the internal signals might not switch in a coordinated way, creating the conditions for an

incorrect output. If this potential problem actually manifests, we observe a **physical, temporary incorrect output pulse** called **glitch** produced when the circuit momentarily evaluates to the wrong value before settling to the correct one. In other words, the **hazard is the risk**, and the **glitch is the observable effect** of that risk materializing. Consider the following example:



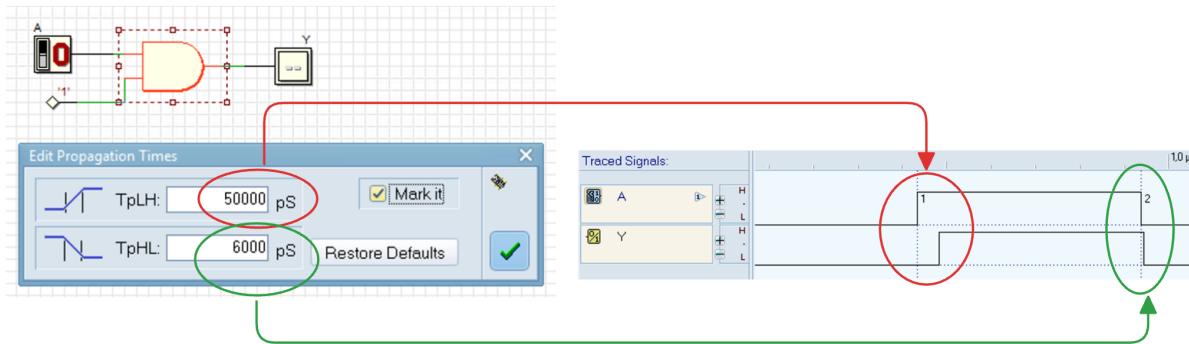
When input B transitions from 1 to 0, the circuit contains two distinct paths to the output: one long and slow, the other short and fast. Because the short path responds first, the output briefly drops to 0. Only later does the slower signal arrive and restore the correct value of 1. This unwanted 1-0-1 pulse is the glitch, and it occurs because the mismatch in path delays created an hazard condition.

The timing diagram highlights how the internal nodes N1 and N2 respond at different times, and how their mismatch briefly forces the output into an incorrect state:

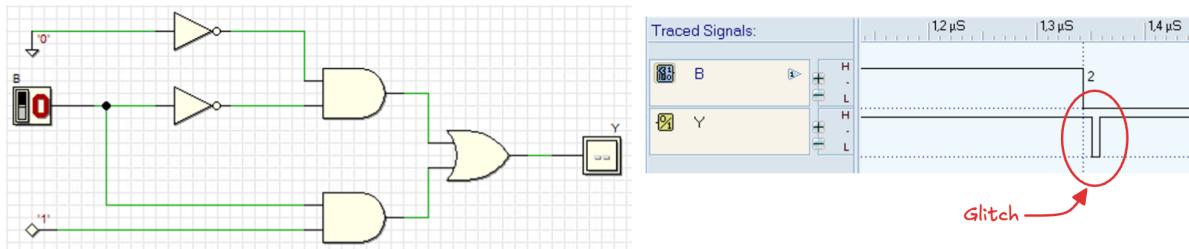


The presence of glitches is not just a theoretical possibility; they can be directly observed when using a simulator that models propagation delays. The DEEDS simulator, for example, assigns

a finite propagation delay to each logic gate (nanoseconds by default). This means that when an input changes, the output of the gate does not update immediately but only after the specified delay has elapsed.

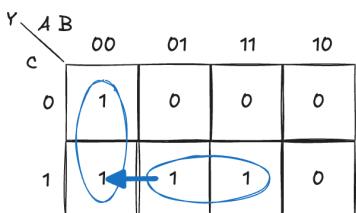


As soon as we introduce these realistic timing characteristics into our circuit, hazards in the logic can turn into visible glitches on the waveform display. We can observe demonstrates the subtle glitch phenomenon discussed in the previous circuit by simulating it in DEEDS:



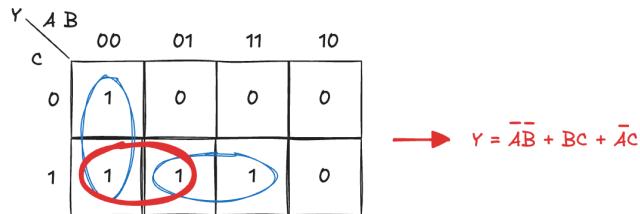
If we zoom into the waveform, DEEDS clearly shows this temporary incorrect pulse appearing at the output. Notice that **glitches are not artifacts of simulation but inherent consequences of unequal path delays in real hardware**. A simulator that faithfully models propagation delays will expose these effects exactly as they would occur in an actual digital circuit.

Hazards can also be predicted from a Boolean perspective. In a Karnaugh map, a transition that crosses the boundary between two prime implicants may leave the output momentarily uncovered:

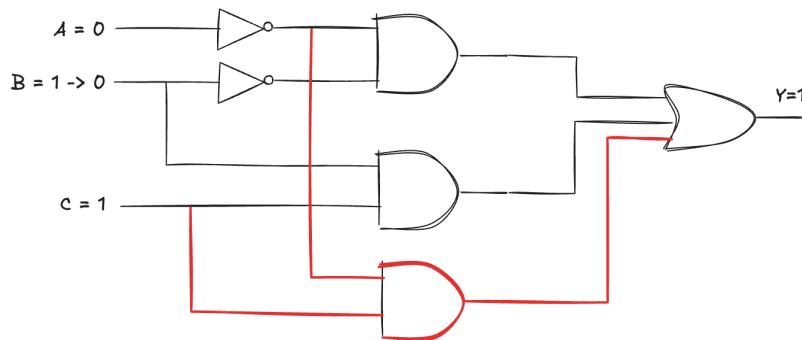


This uncovered region corresponds to a **static hazard**: the output is intended to remain constant during the transition, but the circuit may briefly produce a wrong value, creating a glitch.

A common technique for removing hazards, and therefore preventing glitches, is to add **redundant implicants** to the Boolean expression. By adding an extra grouping in the Karnaugh map that covers the boundary between two prime implicants, we ensure that the output remains covered during the input transition:



This eliminates the hazard and guarantees that no incorrect momentary value is produced. The cost, however, is additional hardware: more logic gates are required to implement the redundant term, increasing area and possibly power consumption.



Even though adding redundant implicants can eliminate **static hazards**, this technique has a fundamental limitation: it cannot prevent glitches that occur when **more than one input changes at nearly the same time**. These are known as **dynamic hazards**, and they arise from the physical reality that signals in a real circuit never change perfectly simultaneously. To understand why, imagine two inputs that are supposed to change at the same instant. On a whiteboard or in a Boolean equation, we treat both transitions as happening together. But in an actual electronic circuit, one input will always change slightly earlier than the other (maybe only a few picoseconds earlier, but still earlier). During that tiny interval, the circuit briefly sees a combination of input values that "should not" exist according to the ideal Boolean abstraction. The circuit reacts to this temporary, unintended combination of inputs by producing a momentary wrong output value (a glitch). And because this glitch comes from two or more inputs moving at once, no amount of adding redundant implicants can guarantee eliminating it.

This leads to a pragmatic engineering observation: glitches are not rare anomalies, they are an inherent part of combinational logic. Designers must therefore accept that glitches are a **fact of life**. The point is not to remove every possible glitch, which is often impractical, but rather to

be aware that they exist and to design systems that do not depend on glitch-free signals in situations where hazards matter. This awareness becomes particularly important when examining timing diagrams on a simulator or oscilloscope. What may appear to be an incorrect output is often just the natural behavior of a real combinational circuit responding to input changes.

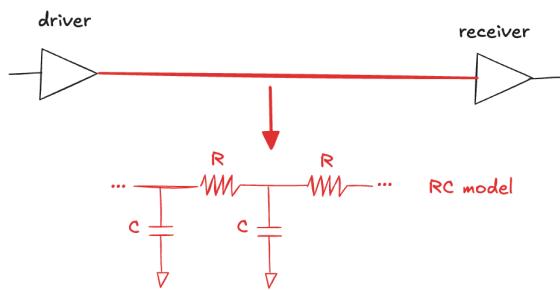
Fortunately, glitches do not always cause functional problems. In synchronous systems, the outputs of combinational blocks are typically **sampled only after adequate time has passed for all signals to settle**. As long as no other component reads the output during the short unstable interval, **the glitch is harmless**. Issues arise only when another circuit is sensitive to the transient behaviour, such as in asynchronous designs or when outputs directly control enable signals, resets, or other timing-critical operations.

10.1.5 Wire delay

Up to this point, we have implicitly assumed that **wires behave as ideal, instantaneous connections**, simple equipotential conductors in which every point along the wire shares exactly the same voltage at any moment in time. This assumption works well for small circuits, short interconnects, and relatively slow signals, because in those cases the electrical behaviour of the wire does not significantly distort or delay the signal. However, **real wires do not transmit information instantaneously**. Instead, **signals propagate along them at a finite speed**, carried by electromagnetic waves that travel at a fraction of the speed of light.

In many practical digital systems, this propagation delay is small enough to be **safely ignored**. If a wire is **short compared to the physical scale of the circuit**, or if the signal **changes slowly enough** that its rise and fall times dominate the timing behaviour, then the wire delay contributes only a negligible amount. A common engineering rule-of-thumb is that if the propagation delay of the wire is less than **about 20%** of the signal's rise or fall time, its effect is considered insignificant.

However, as the length of the wire increases, the assumptions of equipotential behaviour begin to break down. Long wires exhibit non-negligible **resistance** along their length and **capacitance** to surrounding conductors. Together, these effects slow down the propagation of a signal and smear out its edges. To capture this behaviour more accurately, we often replace the idealized wire with an **RC model** consisting of a chain of small resistors and capacitors distributed along the length of the line.



This distributed RC network acts as a low-pass filter, gradually attenuating and delaying fast transitions. The longer the wire, the larger the total resistance and capacitance, and the more pronounced the delay and distortion.

In extreme cases—when wires are very long or when signals switch extremely rapidly, the RC approximation is no longer sufficient. Under these conditions, the signal propagates as a **wave**, and the interconnect behaves like a **transmission line** rather than a simple conductor. Transmission-line behaviour introduces new phenomena: the voltage does not change simultaneously along the line but instead travels as a wavefront. If the line is not properly terminated, this wave can **reflect** at the ends, creating echoes and oscillations that cause noise, overshoot, undershoot, and other undesirable effects. These problems become especially important in high-speed designs such as microprocessors, FPGAs, and fast serial links, where even a few centimetres of wire may act as a significant transmission line.

Understanding these distinctions is crucial as digital design scales up in complexity and speed. While short wires can be treated as ideal connections, long or fast-switching wires demand more sophisticated models to ensure reliable signal integrity and correct timing behaviour across the entire circuit.

10.2 Timing of Sequential Logic

Sequential circuits behave very differently from purely combinational ones because they rely on memory elements (typically flip-flops) to store information from one clock cycle to the next. A flip-flop samples its input D at a specific moment in time, usually on the rising edge of the clock, and transfers that value to its output Q. If the input is stable (clearly a 0 or clearly a 1) at the instant the clock rises, the flip-flop's behaviour is straightforward and predictable.

The difficulty arises when the input signal is changing at the exact moment the clock edge occurs. In such a case, the flip-flop cannot immediately determine whether the input should be interpreted as a 0 or a 1. The situation is similar to that faced by a camera attempting to take a snapshot of a moving object. For the picture to be sharp, the object must remain still for a short interval surrounding the moment the shutter opens. If the object moves during this interval,

the captured image becomes blurred or undefined. Flip-flops have an analogous requirement: the input must remain steady for a certain period around the sampling edge for the device to produce a well-defined output. This interval is known as the **aperture time**. Just as a camera shutter has a brief open window, a flip-flop has a small time window centred on the clock edge during which the input must not change. The input must be stable for some minimum time **before** the clock edge, called the **setup time**, and must remain stable for some minimum time **after** the clock edge, called the **hold time**. Only when both constraints are satisfied can the flip-flop reliably capture the input and update its output correctly.

10.2.1 Dynamic Discipline

Sequential circuits operate under a simple but powerful abstraction: we assume that the value of a signal A at the end of the n-th clock cycle $A[n]$ is **all that matters**, rather than the precise, continuous-time behaviour $A(t)$. This timing-based view of operation leads naturally to the abstraction of time into discrete units called **clock cycles**. We treat all activity within a clock cycle analogously to how we treat voltages in combinational logic: although signals may oscillate, glitch, or pass through intermediate values, what matters is only their final, stable value at the moment when the next clock edge arrives and sampling occurs. As long as the combinational logic between flip-flops settles before the next sampling instant, and the input signals obey setup and hold constraints, the sequential system behaves in a clean and predictable manner despite the underlying analog complexities.

As the **static discipline** guided our use of logic levels, ensuring that voltages always stayed within safe ranges representing clear 0s and 1s, sequential circuits introduce a complementary rule, the **dynamic discipline**, which governs **when** signals are allowed to change. The dynamic discipline requires that all inputs to sequential elements must change **outside** the aperture time, leaving the flip-flop undisturbed around the sampling moment. As a result, the clock period must be long enough to accommodate all propagation delays in the datapath. This requirement places **a fundamental limit on how fast a sequential system can run**: no matter how efficient the logic is, the clock cannot tick faster than the slowest signal path allows.

In an idealized world, every flip-flop in the system would receive the clock edge at precisely the same instant. In real hardware, however, the clock signal travels along wires of different lengths and through buffers of differing delays. This leads to **clock skew**, a phenomenon in which some flip-flops see the clock edge slightly earlier or later than others. Clock skew effectively reduces the amount of time available for signals to propagate within a clock cycle, meaning the designer must choose **a clock period that is even longer than what would be required by pure combinational delay alone**.

There are situations, especially when interacting with unpredictable external inputs, where it

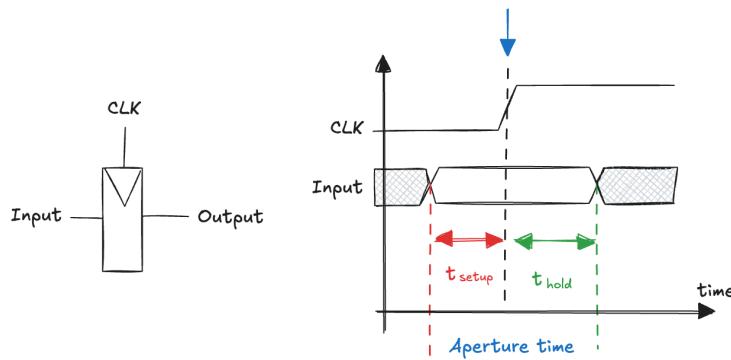
becomes impossible to fully satisfy the **dynamic discipline**. Consider a circuit whose input comes from a mechanical pushbutton. A user pressing the button might do so at an arbitrary moment, and this moment may coincide almost perfectly with a rising clock edge. In such a case, the flip-flop is forced to sample an input that is actively changing. Because it cannot cleanly interpret the voltage as either a valid 0 or a valid 1, the flip-flop may enter a special, problematic state known as **metastability**, a condition in which the internal circuits of the flip-flop fail to resolve the input to a valid logic level. Instead of settling quickly to 0 or 1, the output may hover at an intermediate voltage for an unpredictable amount of time before finally deciding. In theory, this resolution time can be arbitrarily long; in practice, it may last long enough to cause downstream logic to behave incorrectly. Metastability is not a design flaw, it is a direct consequence of the physics of bistable circuits when they are asked to make a decision under ambiguous conditions.

The dynamic discipline therefore tells us not only how sequential circuits should behave in the abstract model, but also reminds us of the physical constraints that limit their reliability. Managing propagation delays, accounting for clock skew, and designing proper synchronization circuits for external inputs are all essential strategies for ensuring that the system stays within the bounds of this discipline and operates predictably from one clock cycle to the next.

10.2.2 Aperture Time

In order for a flip-flop to operate reliably, both its input and its output must respect certain timing constraints. These constraints are part of what manufacturers specify for each sequential element, and they define how the circuit must behave for the model of synchronous digital design to remain valid. The timing requirements concern what happens immediately around the active clock edge, the moment when the flip-flop samples its input and potentially updates its output.

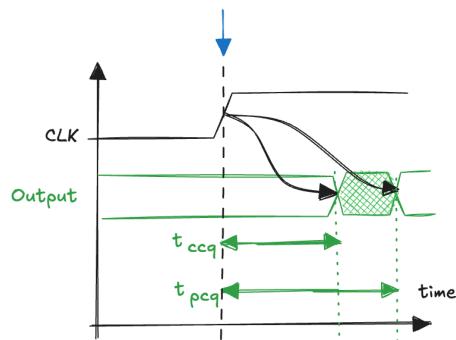
Let us begin with the input. A flip-flop cannot correctly capture a value that is changing at the exact moment the clock edge occurs. Instead, the input must be stable for a short period of time before the rising edge of the clock; this interval is called the **setup time**. During setup time, the input must already have reached its intended value, ensuring that the flip-flop sees a clean and unambiguous 0 or 1 as sampling approaches. The input must then remain stable for a further window of time after the clock edge. This second interval is known as the **hold time**. Together, the setup and hold intervals form the **aperture time**, the total time during which the input must not change. If the input transitions anywhere within this aperture time, the flip-flop may behave unpredictably or even become metastable.



The diagrams illustrate this concept: the clock edge defines the centre of the aperture, while the input must remain flat across the entire region. Any transition within that region risks violating the dynamic discipline and corrupting the sampled value.

10.2.3 Clock-to-Q Delays

Now consider the output of the flip-flop. The output does not update instantaneously at the moment of the clock edge. Instead, there are two important timing parameters that define how the output changes. The first is the **clock-to-Q contamination delay**, which is the earliest possible moment when the output might begin to move in response to the clock. This delay corresponds to the fastest internal paths within the flip-flop and represents the soonest time another circuit might see a change on Q. The second is the **clock-to-Q propagation delay**, which describes the latest time by which the output is guaranteed to have reached its final, stable value. This corresponds to the slowest internal paths that the flip-flop may take while updating the output.



Just as contamination and propagation delays matter in combinational circuits, here they define the timing envelope within which the output transition occurs. The output may begin to rise or fall shortly after the clock, and it may take a short interval before settling, but the timing specifications guarantee that the correct value will appear no later than the propagation-delay bound. All downstream logic must be designed to tolerate this range.

Together, the input timing (setup and hold) and output timing (clock-to-Q delays) form the essential timing specification for a flip-flop. These parameters ensure that despite the analog nature of real circuits, we can treat sequential behaviour as a sequence of precisely sampled digital states (one stable state per clock cycle) maintaining the abstraction on which synchronous digital design is built.

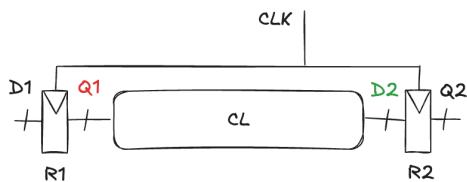
10.3 Synchronous System Timing

In a synchronous digital system, time is divided into discrete, regular intervals defined by the clock. The **clock period** is the amount of time between two consecutive rising edges of the clock signal. Its reciprocal is the **clock frequency**, measured in hertz, meaning cycles per second.

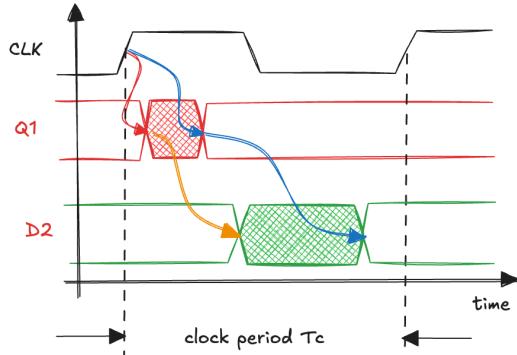
$$f_{clock} = \frac{1}{T_{clock}} [\text{Hz}]$$

Increasing the clock frequency (or equivalently, decreasing the clock period) allows the system to perform more operations per second and therefore increases its overall computational throughput. However, the clock period cannot be made arbitrarily small. Once a value is captured by a flip-flop at the start of a cycle, it must propagate through the intervening combinational logic before reaching the next flip-flop in time for the following clock edge. This propagation takes a finite amount of time due to the logic delays, the wiring delays, and the clock-to-Q delay of the launching register. If the next rising edge arrives before the signal has settled to a stable 0 or 1, the receiving flip-flop may sample an incorrect or metastable value.

To understand this constraint, consider a simple sequential path consisting of two flip-flops, R1 and R2, with some combinational logic CL between them:



At the rising edge of the clock, R1 updates its output Q1, which then begins to propagate through the logic. Because the logic delays are not uniform, some internal nodes may begin to transition quickly (after the contamination delay) while the final output of the logic may only reach its correct, stable value after the propagation delay. These two bounds define the earliest and latest times at which the signal can appear at the input D2 of the receiving flip-flop R2. The following timing diagram illustrates this behaviour:



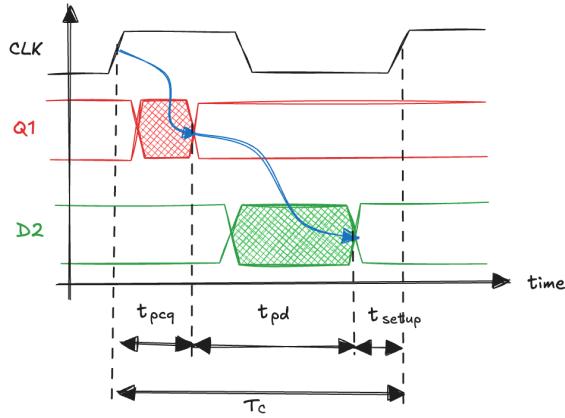
Shortly after the clock edge, the output Q1 starts to change, prompting the signals inside the combinational block and eventually the input D2 of the next flip-flop to begin transitioning. The earliest possible change is shown by the orange arrows, representing the contamination-delay limit. The signal then continues to evolve before ultimately settling to its final value, shown by the blue arrows, representing the propagation-delay bound.

For the system to operate correctly, **the clock period must be long enough to allow all of this activity to complete safely**. The receiving flip-flop must see a stable input for its required setup time before the next rising edge. If the logic requires more time to settle than the clock provides, the system risks **violating the dynamic discipline and capturing incorrect values**. Therefore, the clock period imposes a fundamental limit on the speed of a synchronous design: it must be at least as long as the longest register-to-register path, including all relevant delays. This relationship between the clock frequency, the combinational logic delay, and the flip-flop timing parameters is central to the performance of digital systems, and it guides how we structure, pipeline, and optimize circuits to achieve higher speeds.

10.3.1 Setup Time Constraint

To understand how fast a synchronous digital system can run, we must examine the timing constraints imposed by the receiving register in a register-to-register path. Specifically, we analyse the behaviour of the signal with respect to the **setup time** of the second flip-flop. The signal produced by the first register R1 must propagate through the combinational logic and arrive at the input D2 of the second register R2 **early enough** to satisfy its setup requirement. This journey involves several distinct delays. First, the updated value at Q1 does not appear the moment the clock rises. Instead, the output of R1 transitions only after the clock-to-Q propagation delay. Once the new value is present, it begins travelling through the combinational logic, which takes an additional amount of time equal to the logic's propagation delay**. Only after both of these delays have elapsed will D2 begin settling toward the correct value. However, this is not sufficient. For R2 to capture the correct input at the next rising edge, the value at D2 must be stable

at least one setup time before that edge. In other words, the logic's output cannot be allowed to arrive at the very last moment; it must arrive early enough to satisfy the setup constraint.



Combining all of these requirements, we obtain a simple and fundamental inequality for the minimum allowable clock period:

$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

This inequality tells us that the maximum clock frequency of a synchronous system is limited by three factors: the **delay of the launching register**, the **delay of the combinational logic**, and the **setup time of the receiving register**. In practice, only one of these terms is under the designer's control. The clock period is often dictated by performance targets or marketing requirements, and the flip-flop timing parameters are fixed characteristics of the technology chosen for the design. The **only adjustable quantity** is therefore the propagation delay of the combinational logic. Designers work hard to meet the constraint:

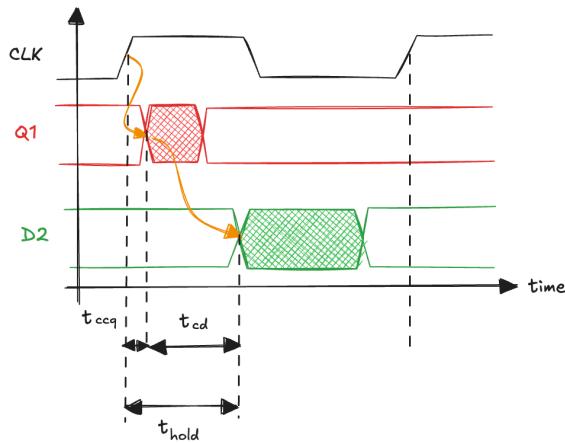
$$t_{pd} < T_c - (t_{pcq} + t_{setup})$$

which essentially states that **the logic must be fast enough to fit within the limited time budget** left over after accounting for flip-flop overhead.

Ideally, the entire clock period would be available for useful computation in the combinational logic. However, the flip-flop timing requirements consume part of this interval. The sum of the flip-flop delays represents a **sequencing overhead** that cannot be avoided: time spent not computing but instead managing the boundaries between clock cycles. As we push systems to run faster, this overhead becomes increasingly significant, reducing the time available for meaningful logic operations and making timing closure more challenging.

10.3.2 Hold Time Constraint

Just as we analysed the setup requirement of the receiving register to determine how long the clock period must be, we now analyse its hold time to determine **how soon** the input is allowed to change after a clock edge. While the setup constraint concerns the latest time at which the signal can arrive, the hold constraint concerns the earliest time at which the signal is allowed to leave its old value. Immediately after the rising edge of the clock, the flip-flop R2 must continue to see a stable input D2 for a minimum interval (hold time). If the signal coming from R1 (and passing through the combinational logic) changes too quickly, R2 may inadvertently capture the new value instead of the old one, violating the design's intended behaviour. In other words, we must ensure that the new value launched by R1 takes at least the duration of the hold time before it can reach R2's input. Two delays contribute to this "arrival" of the new value. First, R1 does not update its output instantly, it responds only after the clock-to-Q contamination delay, the earliest moment at which its output might start to change. Second, the output then travels through the combinational logic, which has its own contamination delay. Only after both delays have elapsed can the new value begin to appear at D2:



To satisfy the hold requirement, the sum of these earliest delays must exceed the hold time:

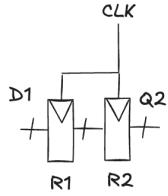
$$t_{ccq} + t_{cd} > t_{hold}$$

Rearranging this inequality yields a requirement on the combinational logic: it must introduce **at least** a certain amount of contamination delay:

$$t_{cd} > t_{hold} - t_{ccq}$$

This may seem surprising, since in setup analysis we were concerned about maximum delays, but in hold analysis we must ensure that delays are **not too small***. This is why we sometimes describe hold-time analysis as making sure the logic is "slow enough".

In many systems, we expect that two flip-flops may be placed directly one after the other with no combinational gates between them:



For this to work safely, the contamination delay of the first flip-flop must naturally exceed the hold time of the second. Reliable flip-flops are therefore designed so that:

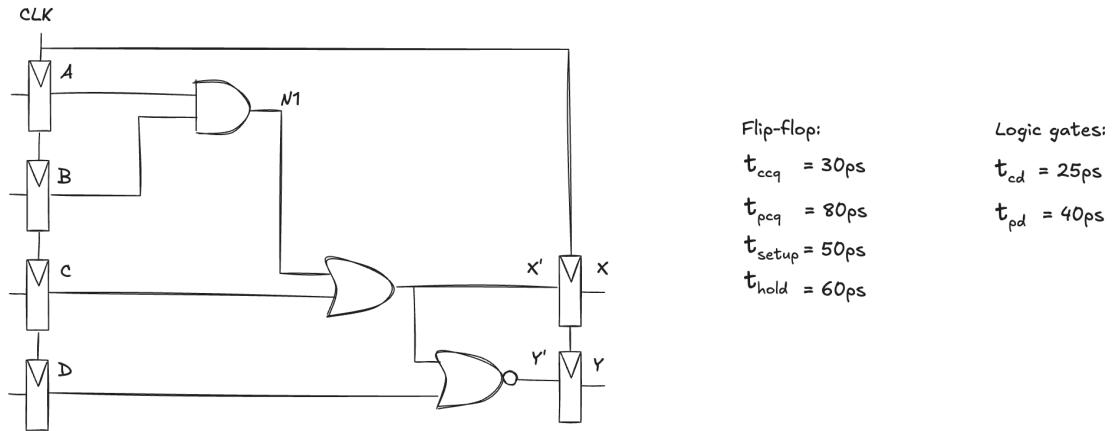
$$t_{\text{hold}} < t_{\text{ccq}}$$

Often, manufacturers go even further and design flip-flops with **zero hold time**, effectively guaranteeing that direct cascading is safe. This dramatically simplifies timing closure, as it eliminates the need to artificially insert delay between registers. Despite this, the hold-time constraint is often overlooked in early designs because setup analysis tends to dominate discussions of performance and clock frequency. However, the hold constraint is **critically important** precisely because violating it cannot be fixed simply by adjusting the clock period. While a setup-time violation can be resolved by lowering the clock frequency, a hold-time violation requires **adding delay** to the logic, modifying the circuit topology itself. This can be difficult, and in some cases disruptive, since artificial delay elements or longer routing paths must be introduced to slow the signal down.

Thus, while setup time determines how fast a circuit can be clocked, the hold constraint ensures that signals do not arrive too quickly. Both constraints work together to guarantee correct sequential behaviour, and a robust design must obey them simultaneously.

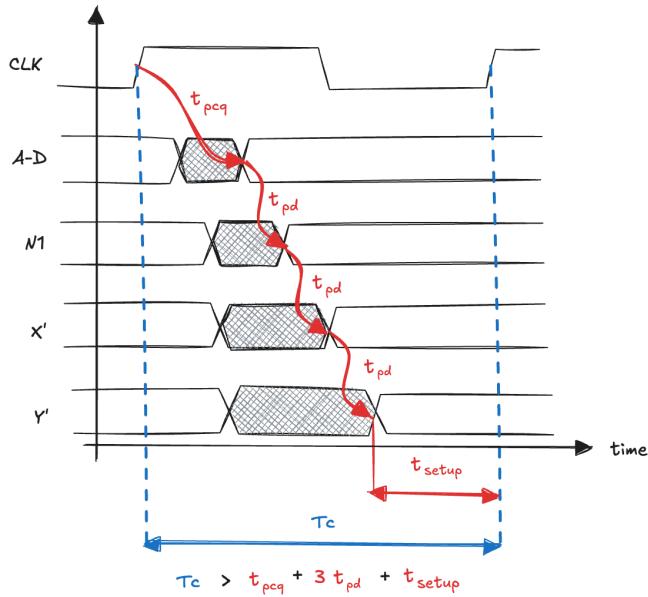
10.3.3 Timing Analysis

Sequential circuits must satisfy both setup-time and hold-time constraints, and these constraints together define the allowable range of delays for the combinational logic that sits between flip-flops. The **setup constraint** imposes a limit on the **maximum** propagation delay of the logic, while the **hold constraint** imposes a limit on the **minimum** propagation delay. In practice, this means that the logic must be neither too slow nor too fast. If the critical path becomes too long, the system cannot operate at the desired clock frequency. If the shortest path becomes too short, the receiving flip-flop may experience a hold-time violation. To illustrate this problem, consider the following sequential circuit:



Several flip-flops launch registered signals A, B, C and D, which then pass through a small combinational network consisting of AND and OR gates. The outputs of this network X' and Y' are captured by the next pair of flip-flops producing the outputs X and Y. The goal is to analyse the timing of this circuit, determine the **maximum safe clock frequency**, and check whether the design suffers from either setup-time or hold-time violations. We are given the timing parameters for both flip-flops and combinational gates. Using these values, we can now identify the relevant timing paths in the circuit.

For the **setup-time analysis**, we trace the **longest** path from any launching flip-flop through the combinational logic into the receiving flip-flop. Along this path, the delays add up: the launching flip-flop contributes its clock-to-Q propagation delay, each gate adds its propagation delay, and the receiving flip-flop requires its setup time margin. The sum of these must be strictly less than the clock period. Violating this constraint would mean that the next rising clock edge arrives before the signal has stabilised at the input of the receiving flip-flop. In the example, all the logic gates have identical propagation delay, so the length of the critical path is determined entirely by the number of gates the signal passes through. Tracing the path from inputs A or B through the AND gate, then through the OR gate, and finally through the last gate that produces Y' , we identify a chain of three logic stages:



Because Y' feeds the receiving register, its value must settle early enough to satisfy the setup requirement:

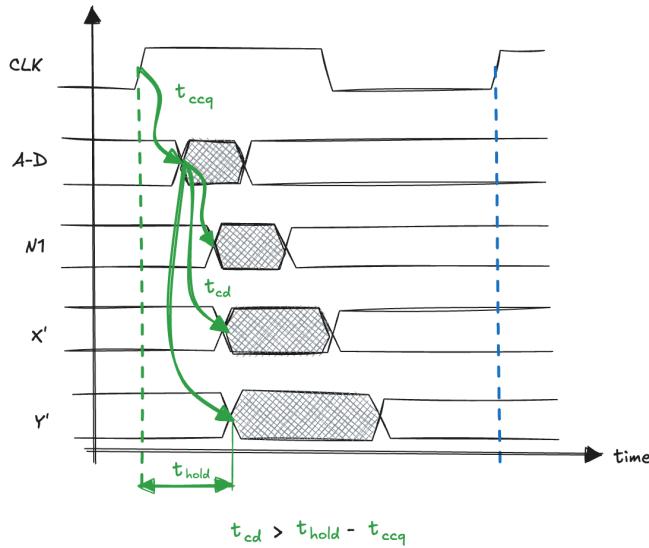
$$T_c > 80\text{ps} + 3 \cdot 40\text{ps} + 50\text{ps} = 250\text{ps}$$

This corresponds to a maximum safe clock frequency of:

$$f_c = \frac{1}{T_c} = 4\text{GHz}$$

This is the theoretical upper limit. Any faster clock would violate the setup constraint, causing the receiving flip-flop to sample a value that has not yet settled.

For the **hold-time analysis**, we trace the **shortest** path from the launching flip-flop to the receiving one. Here, we use contamination delays since they tell us how soon the signal might begin to change. The new value produced by the launching flip-flop must not reach the receiving flip-flop too quickly, or it might overwrite the old value before the hold time has elapsed. This condition gives us a minimum required contamination delay. The shortest path in the example runs from input C directly into the gate producing X' , or from input D directly into the gate that produces Y' , both paths contain only a single logic gate:



For the hold constraint to be satisfied, this sum must exceed the hold time of the receiving flip-flop:

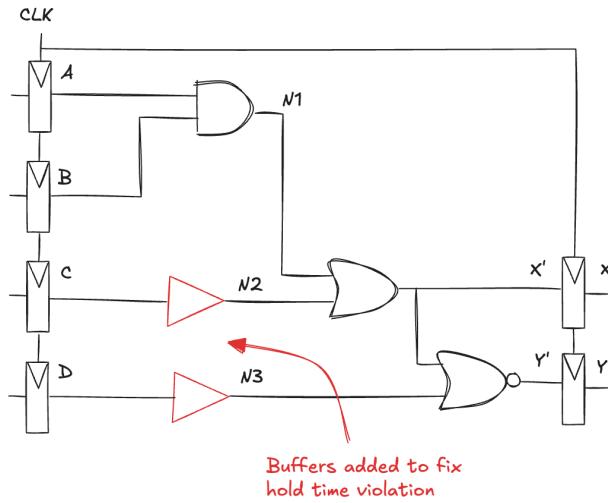
$$t_{ccq} + t_{cd} = 30\text{ps} + 25\text{ps} = 55\text{ps}$$

but we have an hold time requirement of:

$$t_{hold} = 60\text{ps}$$

This indicates a **hold-time violation**: the signal can arrive **too quickly**, potentially corrupting the stored value in the receiving flip-flop. Unlike setup violations, where slowing down the clock can often bring the circuit back into a safe operating zone, hold violations cannot be corrected by adjusting the clock period. This is because the hold constraint concerns what happens immediately after the sampling edge, a time window unaffected by the frequency of the clock. The only way to resolve the issue is to ensure that the signal takes longer to reach the receiving flip-flop, giving its input enough time to satisfy the hold requirement.

A common and practical solution is to **intentionally increase the contamination delay** of the shortest path by adding extra gates, typically simple buffers or inverters, that introduce predictable delay without affecting the logical function of the circuit. In the example, buffers can be inserted on the short path originating from C and D:



These buffers do not change the logic values, but each contributes additional contamination delay. As a result, the earliest possible moment at which the new value can arrive at the receiving register is pushed further away from the clock edge. With two such buffers inserted, the earliest arrival time becomes:

$$t_{ccq} + 2 \cdot t_{cd} = 30\text{ps} + 2 \cdot 25\text{ps} = 80\text{ps}$$

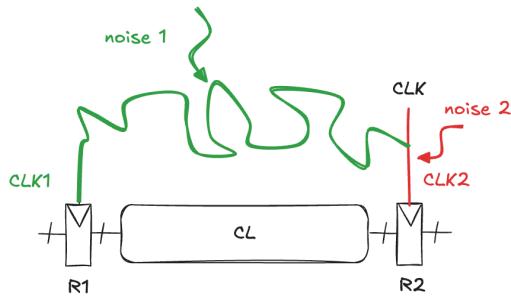
which now comfortably exceeds the hold time requirement. The critical detail here is that the added buffers lie only on the shortest path, not on the critical (longest) path. Because the buffers add only contamination delay (and not propagation delay) to the long path, the maximum clock frequency remains unchanged. The circuit still supports the same 4 GHz upper limit determined by the setup-time analysis. In many designs, simply inserting buffers is a clean and controlled way to achieve this, and automated design tools routinely use this technique during physical synthesis and timing closure.

The example shown has an unusually large hold time, chosen for educational purposes to make the violation more obvious. In commercial hardware, flip-flops are typically designed with hold times smaller than their contamination delays, ensuring that two flip-flops can safely be cascaded without additional buffering. This significantly reduces the likelihood of hold violations, though they can still arise in certain corner cases.

Once both timing constraints are checked, we can determine whether the circuit is safe as designed, and if so, what the maximum possible clock frequency is. This is the essence of timing analysis in synchronous systems: by examining the delays along all possible paths, we can guarantee that the circuit behaves deterministically from one clock cycle to the next.

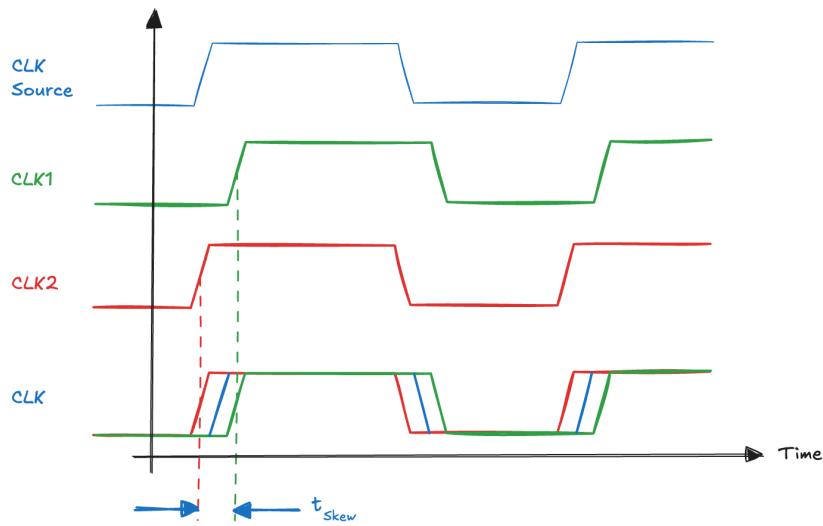
10.3.4 Clock Skew

Throughout our timing discussions, we have implicitly assumed that the clock signal reaches every flip-flop in the system at exactly the same moment. This idealised assumption allows us to treat the entire synchronous circuit as if all registers are triggered simultaneously by **a perfectly aligned global clock edge**. In reality, however, this is rarely the case. Physical constraints, such as differences in wire length, routing congestion, buffering stages, and electrical noise, mean that the clock edge arrives at different registers at slightly different times. This phenomenon is known as **clock skew**.



Clock skew arises because the clock signal must physically travel from the clock source to each flip-flop, often following paths of different lengths through a complex distribution network. A longer wire introduces greater delay than a shorter one, and additional clock buffers inserted along the way contribute their own delays. Moreover, different sections of the clock tree may experience different amounts of noise or interference, causing slight variations in the timing of the clock edge as seen locally by each register.

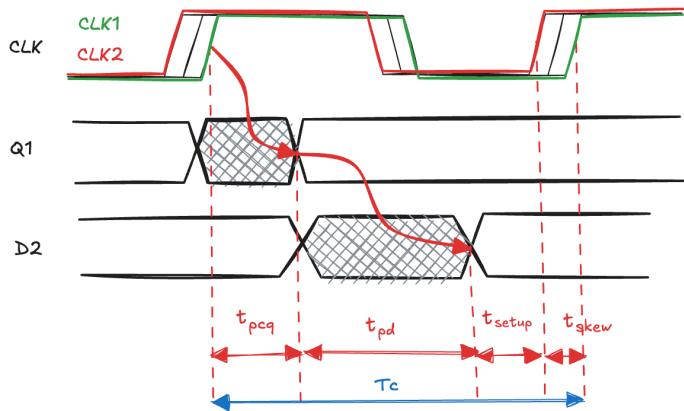
In the picture, for example, the two registers R1 and R2 receive the clock through different routes CLK1 and CLK2. Although both are driven from the same global clock source, the actual arrival time of the rising edge differs: one edge is shifted to the left, the other to the right. A timing diagram shows that the clock edges seen by the registers do not align perfectly: the red and green waveforms represent the real local clocks, offset relative to the idealised global clock (blue):



The time difference between these local clock edges is the **clock skew**. It affects timing analysis in important ways. A skew that delays the clock edge at the receiving register effectively gives more time for signals to propagate, helping to satisfy setup constraints but making hold constraints more difficult to meet. Conversely, a skew that delays the launching register tightens setup margins. Because skew can work for or against the designer depending on its direction, we cannot rely on favourable skew to improve performance. Instead, when doing timing analysis, we must always consider the **worst-case scenario**, assuming the skew values that make both setup and hold constraints as difficult as possible to satisfy.

This conservative approach ensures that the digital system will function correctly **under all circumstances**, even when manufacturing variations, temperature fluctuations, or noisy environments cause slight changes in clock arrival times. Clock skew is therefore an essential part of real-world timing analysis, and understanding its effects is a critical step toward designing robust and reliable synchronous circuits.

Setup time concerns how **late** a signal is allowed to arrive at the receiving flip-flop. Therefore, the worst case occurs when the **launching flip-flop receives the clock late**, while the **receiving flip-flop receives the clock early**. In this situation, the data starts its journey as late as possible while the receiving register asks for it as early as possible. Clock skew therefore **reduces** the effective time available for data to propagate:



To express this mathematically, we add the skew term to the existing setup constraint. The new inequality becomes:

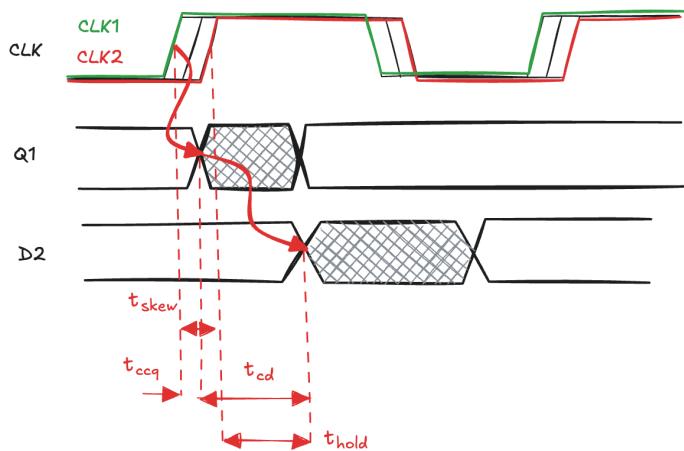
$$T_c > t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$

This directly reduces the maximum allowable propagation delay of the combinational logic:

$$t_{pd} < T_c - (t_{pcq} + t_{setup} + t_{skew})$$

In other words, **clock skew tightens the setup constraint**, leaving less time for useful computation within each clock cycle. If skew becomes large, the designer may be forced to shorten logic paths, add pipeline stages, or lower the clock frequency.

Next, we consider the hold-time constraint. Hold time concerns how **soon** the input to the receiving flip-flop may change after the clock edge. The worst case now is the opposite of the setup scenario: the **launching flip-flop receives an early clock**, while the **receiving flip-flop receives a late clock**. The signal may begin changing very quickly at the launch side, but the receiving register does not capture the clock edge until some time later. This increases the duration during which the receiving flip-flop must continue seeing the old value.



To guarantee safety, the earliest possible change at the receiving flip-flop must be delayed by at least the skew plus the hold time:

$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$

Rearranging yields:

$$t_{cd} > t_{hold} + t_{skew} - t_{ccq}$$

This shows how **clock skew worsens hold-time constraints as well**, requiring greater minimum contamination delay. If the skew is large enough, then even back-to-back flip-flops—normally safe because may suddenly violate hold time. This can happen even when the hold time of the flip-flop is zero. A skew greater than the flip-flop's contamination delay means that the receiving register sees the new value too soon, destroying the stored value.

Here is the **discursive lecture-notes explanation** for *Clock Skew (3)*, written in the same clear and coherent style as the previous sections.

We can revisit the earlier timing-analysis example, but this time we assume that the clock distribution network introduces a 50ps clock skew between the launching flip-flop and the receiving flip-flop. Using the numbers from the earlier example, the new setup constraint becomes:

$$T_c > t_{pcq} + t_{pd} + t_{setup} + t_{skew} = 80\text{ps} + 3 \cdot 40\text{ps} + 50\text{ps} + 50\text{ps} = 300\text{ps}$$

This increases the minimum clock period from the original 250ps to 300ps, reducing the maximum safe clock frequency to:

$$f_c = \frac{1}{T_c} = 3.33 \text{ GHz}$$

Thus, even though the logic did not get slower, the **system** did—simply because skew reduced the amount of usable time per clock cycle.

Next, let us examine the **shortest path**, which determines the hold-time constraint. In the previous analysis, this circuit was already violating hold time. With skew added, the required minimum delay becomes:

$$t_{hold} + t_{skew} = 60\text{ps} + 50\text{ps} = 110\text{ps}$$

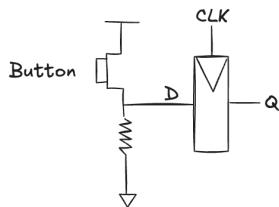
Clock skew makes the hold-time violation significantly worse. The fast path was already too fast; with skew, it becomes even farther from satisfying the timing requirement.

Clock skew therefore extends both ends of the timing window. It reduces the time available for combinational logic to perform useful work (worsening the setup margin), and it increases the minimum delay required through the logic (worsening the hold margin). For this reason, designers must always consider the **worst-case skew** when analysing synchronous systems. Even small amounts of clock skew can dramatically alter the allowable timing range and must be accounted for to ensure correct operation under all conditions.

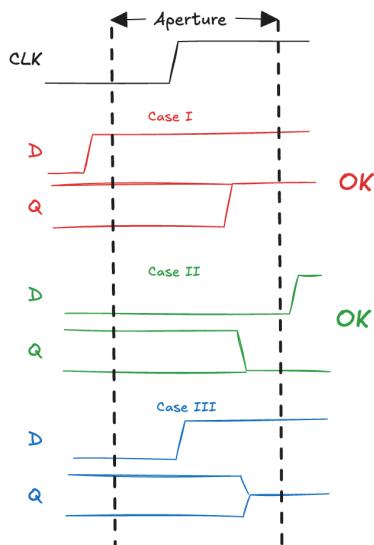
10.4 Timing violations

Up to this point, we have assumed that the input to a flip-flop remains stable throughout the entire aperture time surrounding the active clock edge. In carefully designed synchronous systems this assumption usually holds, because all signals originate from other registers and therefore follow the rules of the dynamic discipline. However, when a signal comes from **outside** the synchronous system (for example, from a mechanical button, a sensor, or an external device) we cannot longer guarantee that it will respect the setup and hold times of the receiving flip-flop. In these cases, it is entirely possible for the input to change at precisely the wrong moment, violating the dynamic discipline and leading to behaviour that cannot be predicted by Boolean logic alone.

To see this, imagine a button connected directly to the D input of a flip-flop:



If the button is pressed well **before** the rising edge of the clock, the input has plenty of time to settle, and the flip-flop correctly samples a logical 1. If the button is not pressed until well **after** the clock edge, the flip-flop sees a 0 and behaves correctly again. But when the button changes state **during** the aperture interval (between the setup time before the clock edge and the hold time after it) the flip-flop sees an input that is neither reliably 0 nor reliably 1. The input violates the dynamic discipline, and in this case the output becomes **fundamentally undefined**.

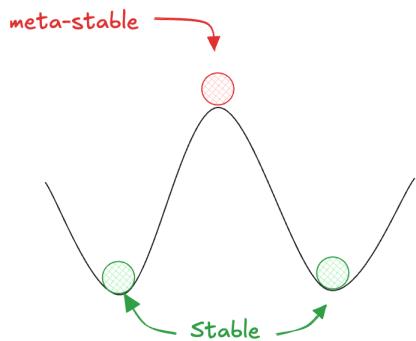


When a flip-flop samples an input that is changing during the aperture window, the internal circuitry cannot cleanly decide between the two stable logic levels. Instead, it may enter a peculiar condition known as **metastability**. In the metastable state, the output voltage hovers in the forbidden zone—neither a valid 0 nor a valid 1. It may oscillate, drift slowly, or remain stuck near the switching threshold for a surprisingly long time.

This phenomenon is not a design flaw, nor can it be eliminated: metastability is an inherent consequence of attempting to sample an arbitrarily timed signal with a synchronous circuit. Therefore, rather than trying to avoid metastability entirely, digital designers employ special techniques to **control** it and ensure that its effects do not propagate unpredictably through the system.

10.4.1 Metastability

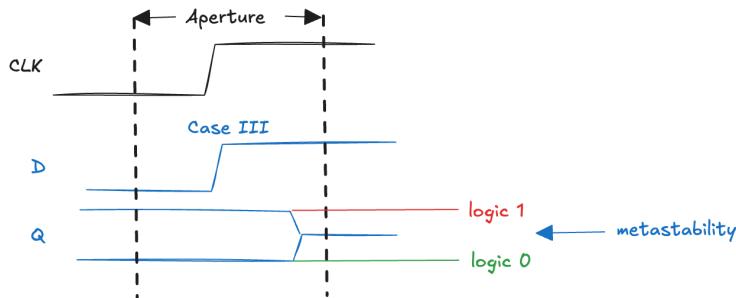
The metastable state of a flip-flop is similar to a ball placed exactly at the top of a hill between two valleys.



The valleys represent **stable states** (logic 0 and logic 1). If the ball is in a valley, it stays there (it is stable). The top of the hill represents the **metastable state**. If the ball were perfectly balanced, it could remain there, but in reality even the tiniest disturbance (noise, imperfections, microscopic asymmetries) will eventually make it roll down one side or the other. This model captures two essential properties of metastability:

- **Unpredictability:** we cannot know in advance which valley (0 or 1) the ball will end up in.
- **Unbounded resolution time:** if the ball is placed very close to the summit, it may take a long time before it starts rolling, meaning the flip-flop may take an arbitrarily long time to resolve to a valid logic level.

Since every bistable element has such a metastable point, **metastability is unavoidable** in sequential circuits:



If a flip-flop enters a metastable state, that uncertain value can **propagate** to other parts of the system, causing unpredictable behaviour unless properly contained.

10.4.2 Resolution Time

When a flip-flop becomes metastable due to a timing violation, it does not remain in that state indefinitely. Instead, the internal circuitry eventually resolves the ambiguity and settles to either a logical 0 or a logical 1. The time it takes for the flip-flop to exit the metastable condition and produce a valid output is known as the **resolution time**.

A key aspect of resolution time is that it **behaves like a random variable** rather than a deterministic delay. Even if we repeat exactly the same experiment (same input transition instant, same clock edge, same circuit) the time required for the output to settle can vary dramatically. Sometimes the flip-flop resolves quickly, in just a few tens of picoseconds; other times, the internal nodes can linger near the metastable threshold for a much longer interval. This **statistical nature** arises from the microscopic physical processes: thermal noise, small transistor mismatches, and the analog behaviour of the internal latch structure.

Empirical and theoretical studies show that the **probability** that the resolution time exceeds some duration decays **exponentially** with the duration. This behaviour is typically described by the relation:

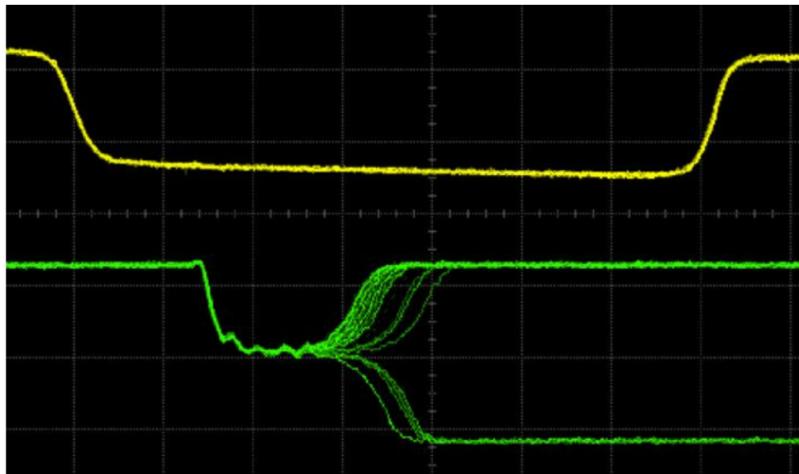
$$P(t_{\text{res}} > t) = \frac{T_0}{T_c} \cdot e^{-t/\tau}$$

Where:

- T_c is the clock period: the shorter the clock, the more often we "sample" the input and the more opportunities we have to hit the dangerous aperture window.
- T_0 and τ are physical constants of the specific flip-flop technology.

The important consequence of this equation is that the **resolution time is theoretically unbounded**. In other words, no matter how good the technology is, there is always a **non-zero** probability that the flip-flop will take an arbitrarily long time to settle. In the following figure,

we see oscilloscope traces of a flip-flop output that has entered a metastable state. The input was changed during the aperture window, causing the output to hover in the forbidden voltage region before eventually settling to a valid logic level:



The probability becomes extremely small for long times, but it never becomes zero. This is a crucial insight: **the digital world cannot completely escape the analog behaviour of its physical implementation.**

However, from a practical perspective, if the system waits long enough (typically one or two clock cycles) the probability that the metastable state remains unresolved becomes negligibly small. In engineering terms, "waiting long enough" means using additional stages or giving the flip-flop sufficient time before its output is used by subsequent logic.

10.4.3 Synchronizers

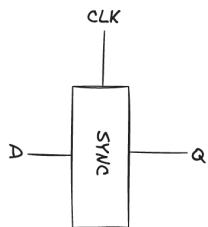
In any real digital system, sooner or later we must deal with signals that do not obey our timing discipline. Up to this point, we have assumed that every signal in the system is synchronous: it changes only in well-defined intervals, always respecting the aperture window around a clock edge. Unfortunately, this assumption breaks down as soon as our circuit interacts with the external world. Inputs generated by physical actions (pressing a button, moving a sensor, receiving a signal from another device with a different clock) are inherently **asynchronous**. They can change at arbitrary times, completely unrelated to the internal clock of the system. As a result, they very easily violate the setup or hold constraints of the receiving flip-flop, producing metastable outputs and potentially leading to erratic failures.

These asynchronous inputs are not rare exceptions; they are **inevitable**. A human pushing a button certainly does not coordinate the motion of their finger with the rising edge of a clock. Even two subsystems on the same board, operating with clocks of similar frequencies, will drift

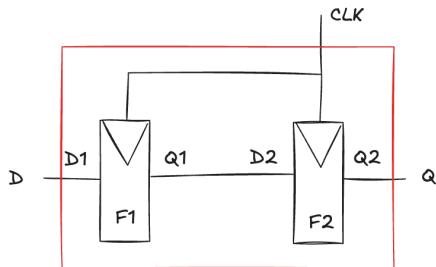
relative to each other. The consequence is that any digital system that interfaces with reality **must be prepared to handle asynchronous transitions safely**.

Since metastability cannot be avoided completely, designers work to ensure that the **probability** of observing a metastable voltage at a point where it might cause damage is **extremely small**. The acceptable probability depends on the application. For a consumer device such as a smartphone, a single metastability-related failure in ten years may be tolerable: the user might experience a rare glitch but can simply restart the device. For a safety-critical application, such as a medical implant or a system controlling industrial machinery, the acceptable probability may be so small that the expected number of failures over the lifetime of the universe is still less than one. Designing for such **extreme reliability** requires careful control of where metastability can occur, and how it is contained.

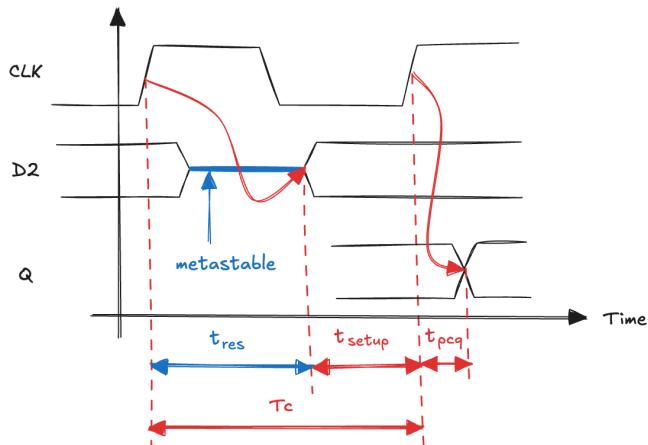
To achieve this, we can introduce a building block called a **synchronizer**. It is a circuit that receives an asynchronous input signal D and a system clock CLK, and produces an output Q that is safe for the rest of the synchronous system to use. The key idea is that the synchronizer allows **metastability to occur only in a controlled location** (inside its own internals) and gives time for this metastability to decay before the output is observed elsewhere.



The behaviour of a synchronizer is simple to describe. If the asynchronous input D happens to be stable during the aperture window, the synchronizer behaves like an ordinary register: on the rising edge of the clock it copies D to Q. If instead D changes during the aperture, the synchronizer internals may go metastable. However, the design ensures that the metastability will almost certainly resolve before the next clock edge, so that the final output Q is always a valid logic level, even if the value does not necessarily match the most recent value of D. In other words, the synchronizer protects the system from metastability by containing it and giving it enough time to settle, thereby turning an unpredictable input into a clean, stable signal with high probability. A very common and effective way to build a synchronizer is to place **two flip-flops in series**, both clocked by the same system clock. This simple structure provides a surprisingly powerful mechanism for containing metastability and preventing it from affecting the rest of the circuit.



The first flip-flop, F1, directly samples the asynchronous input D on the rising edge of the clock. Because D is not synchronized to the clock, it might change at a dangerous time (in the middle of the aperture window). If that happens, the output Q1 (which is the input D2 of the second flip-flop F2) may briefly enter a metastable state. The key idea is that **F1 is the only flip-flop allowed to go metastable**, and its output is not used anywhere else except as input to F2. Now comes the crucial part: metastability does not last forever. If the **clock period is sufficiently long**, then F1 has most of the entire cycle to resolve its metastable condition. By the time the next rising edge arrives, the voltage at D2 has (**with high probability**) settled to a valid logic level. At that point, F2 samples this now-stable value and produces a clean, valid output Q for the rest of the system.



In other words, the first flip-flop isolates the metastability, and the second flip-flop ensures that only a stable signal escapes into the synchronous logic. The two-flip-flop synchronizer is the standard building block used in virtually every modern digital design, from microcontrollers to FPGAs to large SoCs. This approach has limits: the synchronizer will fail if the metastable state lasts longer than the time available before the second flip-flop samples the signal. More precisely, F2 will receive an invalid or metastable input if:

$$t_{\text{res}} > T_C - t_{\text{setup}}$$

meaning that the resolution time of F1 metastability exceeds the time budget between the clock

edge and when F2 setup constraint must be satisfied. Fortunately, from the theory of metastability we know that the probability that the resolution time exceeds a certain duration decreases exponentially with that duration. Applying the earlier formula, the probability that the synchronizer fails on any given sampling event is:

$$P(\text{failure}) = \frac{T_0}{T_C} e^{-\frac{T_C - t_{\text{setup}}}{\tau}}$$

This expression makes several important points clear. First, the failure probability decreases exponentially as the clock period increases. A longer cycle gives F1 more time to recover. Second, the constants determined by the physical characteristics of the technology play a fundamental role: they represent the inherent speed at which the flip-flop escapes its metastable state.

Finally, since systems rarely encounter only a single asynchronous event, if the **asynchronous input changes N times per second**, each transition presents a fresh chance for metastability. The **probability of a failure occurring per second** is therefore:

$$P(\text{failure/sec}) = N \cdot P(\text{failure})$$

This gives designers a **quantitative way to evaluate the reliability of their synchronizers**. By choosing appropriate flip-flops, adjusting the clock period, or adding additional stages, they can ensure that the probability of a failure can be small as required for the application at hand.

10.4.4 Reliability

We have seen that metastability cannot be entirely eliminated; at best, we can reduce the probability that a metastable condition propagates into the rest of the system. To evaluate **how safe a design is**, engineers use a quantitative **measure of reliability** known as the **Mean Time Between Failures (MTBF)**. This value tells us, on average, how long the system will operate before a metastability-induced failure occurs. The larger the MTBF, the more reliable the system.

The MTBF is simply the reciprocal of the probability that the synchronizer fails per second. If the chance of failure in any given second is extremely small, then the MTBF will be extremely large, perhaps measured in years or centuries. Using the earlier expression for the probability of failure, we can write:

$$\text{MTBF} = \frac{1}{P(\text{failure/sec})} = \frac{T_C}{NT_0} \exp\left(\frac{T_C - t_{\text{setup}}}{\tau}\right)$$

This equation illustrates a powerful insight: **the MTBF increases exponentially as the synchronizer is given more time to resolve the metastability**. Waiting even one extra clock cycle can increase the MTBF by several orders of magnitude. This is why, in typical digital systems, a two-flip-flop synchronizer is sufficient: it provides enough time that the probability of failure

becomes astronomically small. However, in high-speed systems (where the clock period is extremely short) the amount of time available for resolution may be so limited that several extra synchronizer stages may be required to maintain an acceptable MTBF.

As an example, consider a system that receives asynchronous updates from a sensor. On average, this sensor changes its output only 0.2 times per second, meaning there are relatively few "dangerous" input transitions. We want the system to have an MTBF of at least one year, meaning the expected time between metastability-induced failures should exceed:

$$\text{MTBF} = 1 \text{ year} \approx 3.14 \times 10^7 \text{ seconds}$$

The flip-flops used in the synchronizer have the following characteristics:

- $\tau = 200\text{ps}$
- $T_0 = 150\text{ps}$
- $t_{\text{setup}} = 500\text{ps}$

Substituting these values into the MTBF equation, we get:

$$3.14 \times 10^7 = \frac{T_C}{0.2 \times 150 \times 10^{-12}} \exp\left(\frac{T_C - 500 \times 10^{-12}}{200 \times 10^{-12}}\right)$$

This expression has no closed-form algebraic solution, but it is easy to solve numerically:

$$T_C \approx 3.036 \text{ ns}$$

Thus, for the MTBF to exceed one year, the system clock must have a period of at least 3.036ns, corresponding to a maximum clock frequency of approximately 330 MHz. If the clock were faster than this, the synchronizer would not have enough time to resolve metastability with sufficiently high probability, and the expected failure rate would exceed our requirement.

11 Sequential Building Blocks

When we move from basic sequential logic elements (such as latches and flip-flops) to larger systems like counters or memories, it becomes useful to group circuits into **building blocks**. This philosophy mirrors what we have already seen with combinational logic: rather than dealing with individual gates, we organize logic into functional modules that can be reused and combined.

Sequential logic benefits even more from this approach, because these circuits involve state, and reasoning about many interconnected states at the transistor or gate level would quickly become unmanageable.

Designers rely on the same core principles that guide the digital abstraction:

- **Abstraction:** each building block hides its internal implementation. For example, a register might internally use a chain of flip-flops, but from the outside we simply see a component that stores an n-bit value.
- **Hierarchy:** small, simple blocks are composed to form larger and more complex ones. A counter is made of flip-flops; an ALU may contain several counters, registers, and multiplexers; a microprocessor contains many of these subsystems.
- **Modularity:** each block has a **well-defined interface**—its inputs, outputs, and timing behavior are clearly defined. This allows us to replace one implementation with another as long as the interface remains the same.
- **Regularity:** many blocks follow patterns that repeat. A register file contains many similar registers; a memory array contains many identical cells. Repetition simplifies both design and verification.

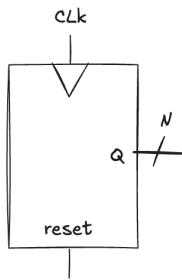
By hiding unnecessary gate-level detail, we keep our focus on the **function** of the block rather than on its internal wiring. In many cases, the building block can be treated as a **black box**: we only need to know what it does, not how it does it.

These building blocks are the foundation of every digital system you will study. As the course progresses, we will frequently use them—individually and in combination—to build increasingly complex structures. Ultimately, these blocks allow us to assemble the architecture of a **microprocessor**, demonstrating how high-level computing behavior arises from systematic organization of simpler sequential and combinational modules.

11.1 Counters

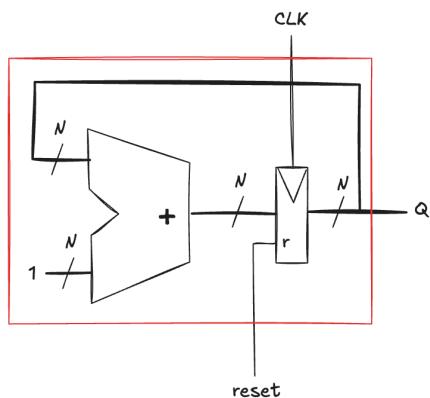
A **counter** is one of the simplest and most widely used sequential building blocks. Conceptually, it is a device that progresses through a well-defined sequence of binary values. If the counter is N bits wide, it moves through all the 2^N possible patterns, starting from 0 and incrementing by 1 at every clock cycle until it wraps around and starts again.

The counter advances on the **rising edge** of the clock. Each rising edge triggers a state update: the stored value is increased by one. To ensure predictable behavior, counters usually include a **reset** input. When reset is asserted, the counter ignores the clock and forces its output to 0. This is useful during system initialization, so that all parts of a digital design begin from a known state.



11.1.1 Counter by Addition

One way to implement a counter is to use a combination of an adder and a register. The register holds the current value of the count, and the adder computes the next value by adding one to whatever is stored in the register:



On every clock cycle, the result of the addition is written back into the register, overwriting the previous value. This simple feedback loop produces the familiar counting behavior. When viewed at the bit level, each bit of the counter toggles at a different rate: lower-order bits toggle quickly, while higher-order bits toggle only after many cycles. In fact, the most significant bit changes state only once every 2^N clock cycles.

11.1.2 Frequency Divider

Counters are often used to **reduce the frequency of a fast clock signal**. Because each bit inherently divides the clock frequency by a factor of two, an N-bit counter acts as a frequency divider by:

$$2^N$$

This property is especially useful when designers need to slow down fast, internal signals for observation or for external interfaces. As an example, consider a system clocked at 50 MHz. By feeding this clock into a 24-bit counter and observing the most significant bit, we obtain a signal that toggles at approximately 2.98Hz, slow enough to drive an LED that blinks at a rate visible to the human eye.

11.1.3 Digitally Controlled Oscillator

A digitally controlled oscillator (DCO) is a natural extension of the counter concept. Instead of simply adding one at every clock cycle, we allow the counter to add an arbitrary integer value P. This small change gives us the ability to generate a wide range of output frequencies starting from a single reference clock. The idea is that if an N-bit counter accumulates the value P at each rising edge of the clock, the most significant bit no longer toggles once every 2^N cycles, but rather at a rate proportional to

$$\frac{P}{2^N}$$

The output frequency taken from the most significant bit becomes

$$f_{\text{out}} = f_{\text{clk}} \frac{P}{2^N}$$

This formula shows immediately how the designer can shape the output frequency. By choosing the counter width N and the increment value P, it becomes possible to obtain nearly any desired output rate:

$$\frac{P}{2^N} = \frac{f_{\text{out}}}{f_{\text{clk}}}$$

A larger N provides finer frequency resolution because the step becomes smaller, though the cost is a wider adder and register.

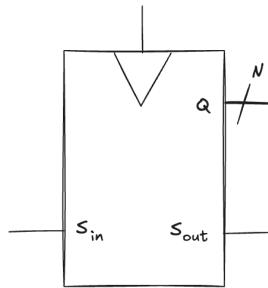
As an example, consider a system driven by a 50MHz clock where the goal is to generate an output signal of approximately 500Hz. The target ratio is

$$\frac{P}{2^N} = \frac{500\text{Hz}}{50 \times 10^6\text{Hz}} = 0.00001$$

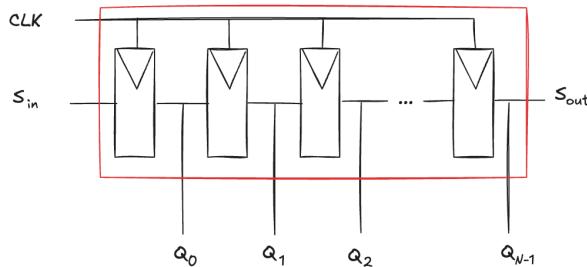
One option is to use a 24-bit register. In this case, selecting P=168 gives an output of about 500.68Hz, which is already quite close to the target. If an even more accurate value is needed, increasing the counter to 32 bits allows a better approximation. With N=32, choosing P=42.950 produces an output of roughly 500.038Hz, essentially reaching the desired frequency with much finer precision. This illustrates how the DCO technique provides a simple, flexible, and fully digital way to synthesize arbitrary frequencies without requiring analog oscillators.

11.2 Shift Registers

A shift register is a sequential circuit specifically designed to **move data through a chain of storage elements**. At every rising edge of the clock, the register accepts a new bit from its serial input S_{in} and simultaneously shifts all previously stored bits one position forward. The newest bit enters the first position, the previous value of that position moves into the second, and so on, until the oldest value emerges at the serial output S_{out} . After each clock cycle, the entire content effectively moves one position to the right, behaving very much like a conveyor belt for bits.



The simplest way to build a shift register is to connect N flip-flops in series, all driven by the same clock. Each flip-flop stores one bit of the register, and its output feeds the input of the next flip-flop in the chain:



Because they all update together on the clock edge, the data moves through the system in a controlled and synchronous manner.

It is important not to confuse shift registers with shifters. Shifters are combinational circuits that shift a multi-bit input left or right by a specified amount without requiring a clock. They produce an instantaneous output as soon as the input changes. Shift registers, in contrast, rely on the clock and move data gradually, one bit per cycle.

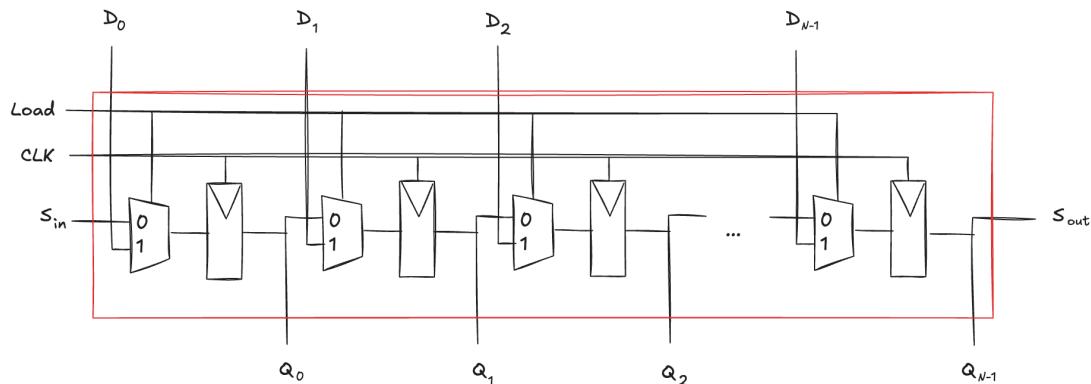
11.2.1 Serial-to-Parallel Conversion

A useful way to interpret a shift register is as a serial-to-parallel converter. The input arrives one bit at a time through S_{in} . After N clock cycles, the register contains the last N input bits, now available simultaneously at the parallel outputs Q . This makes shift registers extremely valuable when interfacing serial data streams with circuits that expect parallel data, such as memory modules, processors, or digital communication blocks.

11.2.2 Parallel-to-Serial Conversion

Shift registers are not limited to serial-to-parallel conversion. A closely related structure is the **parallel-to-serial converter**, a circuit that performs the opposite transformation. Instead of receiving data one bit at a time, it loads an entire N -bit word simultaneously through a set of parallel inputs. Once the data is stored, the circuit shifts the bits out sequentially on each clock cycle. This method is commonly used in communication interfaces where data is processed internally in parallel form but must be transmitted serially to save wiring or bandwidth.

A standard shift register can be extended so that it supports both directions of conversion—serial-to-parallel and parallel-to-serial—with the same hardware. To achieve this dual functionality, each flip-flop in the chain receives not just the usual serial input from its neighbor but also an additional parallel input line.



A control signal, often named **load**, determines the operating mode. When the signal is asserted, all flip-flops simultaneously capture the values present on their parallel inputs D . In this mode, the shift register acts like an array, loading an entire word in a single clock cycle. When load is not asserted, the circuit returns to its normal shifting behavior, moving data step by step from the serial input S_{in} toward the serial output S_{out} .

This flexible design makes the shift register a very versatile building block in digital systems. It can convert serial streams into parallel data for internal processing, or prepare parallel data

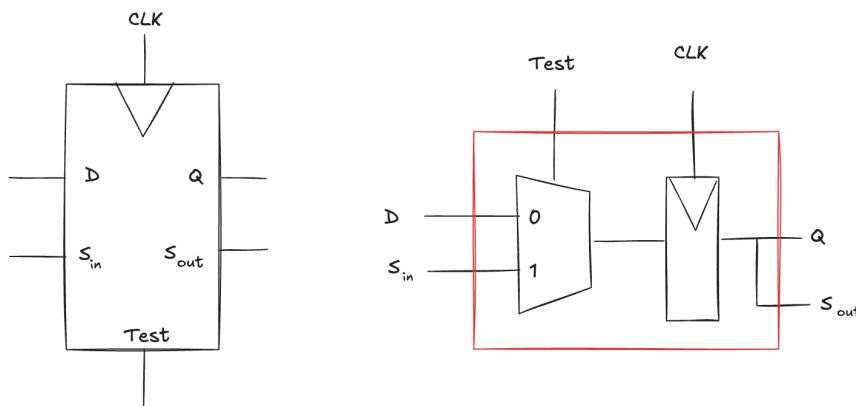
for serial transmission. It is widely used in digital communication peripherals, input expansion modules, display drivers, and many other subsystems that must interface between different forms of data representation.

11.2.3 Scan Chains

Testing digital circuits is an essential part of ensuring correct operation, but the difficulty of testing depends heavily on whether the circuit is combinational or sequential. Combinational circuits are relatively easy to test because they have no memory: every output depends only on the current inputs. To verify that such a circuit behaves correctly, the designer simply applies a set of known input patterns, called **test vectors** and checks that the resulting outputs match the expected ones.

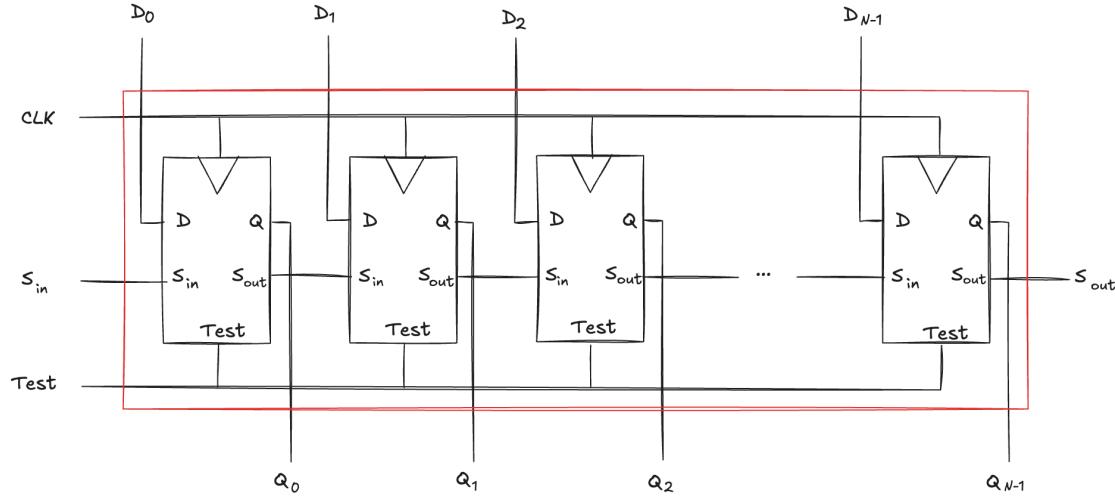
Sequential circuits, however, are much more challenging to test because they contain internal state stored in flip-flops. The output at any moment depends not only on the present inputs but also on a long history of previous inputs. As a consequence, the **circuit may need to be stimulated with many cycles of test vectors before it reaches a specific internal state**. In some cases this becomes **impractical**. For example, verifying the correct behavior of the most significant bit of a 32-bit counter may require 2^{31} clock cycles just to reach the point where that particular state is exercised. **Such long tests are completely unrealistic in real hardware testing environments.**

To overcome this problem, designers introduce a technique known as **scan chains** that exploit shift register to facilitate reaching a desired internal state quickly. The idea is to give the tester direct access to the flip-flops' internal state. During normal operation, each flip-flop behaves as usual, loading data from its functional input D. But in test mode, the flip-flop gets input from a different source S_in:



All the flip-flops are configured so that they form **one long shift register**. Their S inputs and

outputs are chained together serially using, allowing the tester to **shift arbitrary values into all flip-flops** and, later, shift their contents back out for inspection. A control signal selects whether the circuit is running normally or in scan mode:



This mechanism dramatically simplifies sequential circuit testing. Instead of having to drive the circuit through a long sequence of cycles to reach a desired state, the tester can simply shift the desired bit pattern directly into the flip-flops in only N cycles, where N is the number of flip-flops. After shifting in the test state, the underlying combinational logic can be exercised for one or more clock cycles, and then the resulting state can be shifted out and examined.

Consider the example of a 32-bit counter. To test it, one could shift the pattern 0 1 1 1 1 ... 1 1 into the chain while the circuit is in test mode. After one normal counting cycle, the counter should increment this pattern and produce 1 0 0 0 0 ... 0 0. By shifting this result back out, the tester can verify that the counter increments correctly. The full test requires 32 cycles to shift in the initial pattern, one functional cycle to perform the increment, and 32 more cycles to shift out the resulting pattern, for a total of 65 cycles, far fewer than the billions of cycles that would be required without scan techniques.

Scan chains therefore transform the problem of testing complex sequential systems into something almost as straightforward as testing combinational circuits. They are now a standard feature in modern digital integrated circuits, often inserted automatically by design tools to ensure testability.

11.3 Memories

Digital systems rely heavily on memory to store both the data they manipulate and the intermediate values produced during computation. Even the simplest circuits use some form of

storage: for example, registers are a small but essential form of memory that hold temporary values inside processors, counters, or control units. While registers handle only a few bits at a time, memory arrays allow digital systems to store extremely large amounts of information efficiently.

Memories are traditionally classified according to **how they store and preserve data**. The most familiar category is **Random Access Memory** (RAM). RAM is a **volatile** form of memory, meaning that it loses its contents when power is removed. When the system is switched off, everything stored in RAM disappears. In contrast, **Read Only Memory** (ROM) is generally **non-volatile*** and it retains information even in the absence of power, making it suitable for storing firmware, boot code, or data that must survive across power cycles.

The common names RAM and ROM, however, stem from **historical distinctions** that no longer perfectly match modern technology. RAM was originally introduced to highlight the fact that it provides uniform access time to any location, unlike sequential access devices such as tape recorders, where reaching nearby data was faster than reaching distant data. ROM, on the other hand, was once genuinely "read-only" because it was manufactured with fixed contents that could not be overwritten. Today, both of these ideas are outdated. Modern ROM types not only allow random access but can also be rewritten under specific electrical conditions, while RAM technologies come in many varieties with different behaviors.

Because of this evolution, the terminology has become somewhat confusing. A ROM is not really "read-only", and a RAM is not the only memory that provides random access. For practical engineering purposes, the most meaningful distinction that remains is whether the memory is volatile or non-volatile, whether it forgets or remembers its contents when the power is turned off. This single difference strongly influences how a memory is used in digital systems and how it is integrated into larger architectures.

To appreciate how far digital memory technology has come, it is worth looking back at one of the earliest commercial computer storage systems: the IBM 305 RAMAC, introduced in 1956.



This machine was a landmark in computing history because it was the **first system to feature a hard disk drive** rather than relying on magnetic drums or punched cards. The RAMAC system consisted of several large cabinets, including a processing unit, magnetic process drum, magnetic core registers, and the electronic logic and arithmetic circuits that made up the computing engine. By today's standards, these components appear enormous, occupying an entire room and requiring specialized operators.

The most striking detail is its storage device. The RAMAC's hard disk held just **5 megabytes** of data, an amount so small that modern smartphones handle it without a second thought. Yet in 1956, those 5MB represented a technological breakthrough. The drive itself was physically massive as seen in historical photographs of installations:

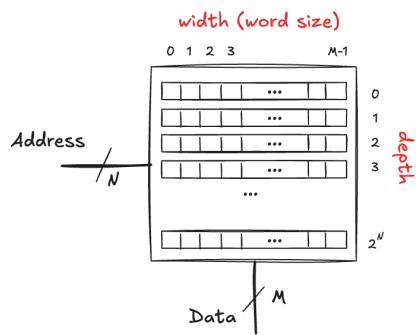


Instead of being purchased outright, the RAMAC system was typically leased to customers. The price was around 3.200 US dollars per month, which corresponds to roughly 30.000 US dollars per month in today's currency, after adjusting for inflation. Despite this cost, companies adopted the RAMAC because it fundamentally changed the way data could be accessed and stored, offering random access to records instead of the strictly sequential access of earlier technologies.

Looking at these examples highlights the extraordinary progress in memory technology. What once required an entire room, a team of technicians, and a large financial investment is now integrated onto tiny chips costing only a few cents, while offering storage capacities millions of times larger.

11.3.1 Memory Arrays

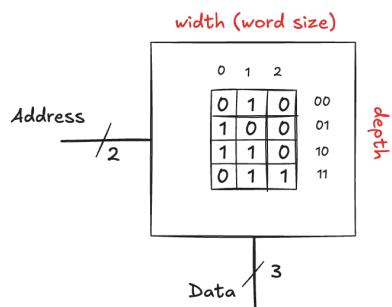
A memory device can be understood as a **two-dimensional grid of storage cells** arranged in rows and columns. Each row represents a **location** where a word of data is stored, and each column corresponds to a single bit within that word. To access a specific row, the circuit uses an **address**: a binary number that selects exactly one of the available rows. The total number of rows is referred to as the **depth** of the memory. Once the row has been chosen by the address, the bits stored in that row form the data word that is read or written. The number of bits in each word, which corresponds to the number of columns in the array, is called the **width** or **word size** of the memory.



If the memory uses an N -bit address, then it contains 2^N rows, because N bits can uniquely encode that many distinct positions. If each row holds M bits of data, the memory is M columns wide. The total size of the memory is therefore the product of its depth and its width: $(2^N \times M)$ bits

$$2^N M \text{ bits}$$

To make these ideas concrete, imagine a memory with a 2-bit address and 3-bit data:



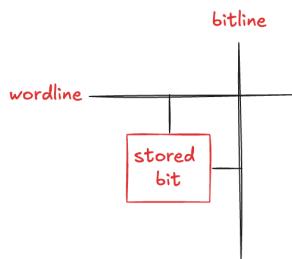
Because the address is 2 bits wide, it can specify exactly four rows. Each of these rows stores a 3-bit data word. Thus, this device is a 4-by-3 memory array: four words of memory, each three bits wide. When an address such as 01 or 10 is presented, the memory selects the corresponding row and either outputs the 3-bit word stored there or writes new data into that location.

This simple model of rows and columns provides the foundation for all forms of memory, whether tiny register files inside a processor or massive RAM modules used in modern computers. Regardless of scale, the conceptual structure remains the same: memory stores words at discrete addresses, and each access selects one row and retrieves or updates the bits contained in that word.

11.3.2 Bit Cell

At the lowest level, every memory array is composed of **tiny storage elements** called **bit cells**. Each bit cell stores exactly one bit of information, and millions or billions of these cells are arranged in a regular grid to form modern memory chips. Although different memory technologies use different circuit designs for the bit cell, they all share the same architectural idea: each cell must be individually **selectable**, **readable**, and **writable**.

To make this possible, every bit cell is connected to two special control lines: a **wordline** and a **bitline**:



The wordline runs horizontally across a row of cells, while the bitline runs vertically down a column. The **memory's address decoder activates exactly one wordline** at a time based on the incoming address. When a particular wordline is asserted, it effectively opens a small switch inside each bit cell of that row, **connecting the internal storage element to the corresponding bitline**. If the wordline is not active, the bit cell remains electrically isolated; its stored value is preserved, but it cannot interact with the outside circuitry. The exact mechanism used to store the bit depends on the **memory technology**.

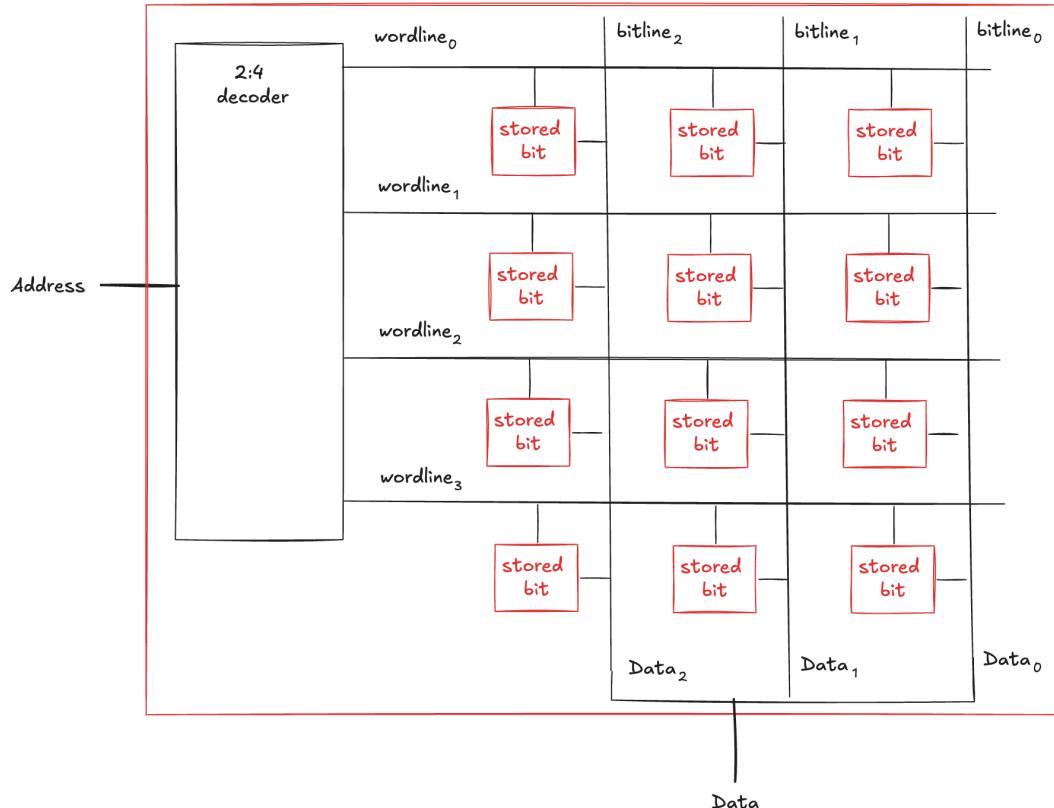
Reading the content of a bit cell begins with preparing the bitline in a neutral state, it is left floating or precharged so that it does not impose a value onto the cell. When the wordline is then activated, the tiny internal storage element of the bit cell influences the bitline, either pulling it slightly toward a logical 1 or allowing it to fall toward a logical 0. Sense amplifiers connected to the bitline detect this small change and convert it into a clean, full-swing digital signal representing the stored bit.

Writing to a bit cell reverses the roles. Before activating the wordline, the memory controller drives the bitline to the value that must be written, either a strong 0 or a strong 1. Once the

wordline turns on, this driven bitline overwhelms whatever value was previously stored inside the bit cell. The cell's storage circuit is forced into the new state, and when the wordline is turned off again, the new bit remains stored inside the cell.

This interplay between the wordline selecting the row and the bitline carrying the data enables the memory array to read and write individual bits efficiently while scaling to extremely large storage capacities.

Individual bit cells come together to form a complete memory array:

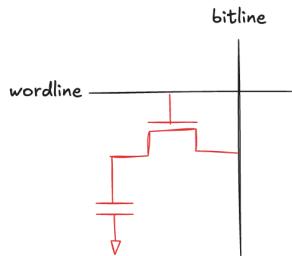


The address enters a decoder, which activates exactly one of the wordlines. All bit cells along that selected row become connected to their respective bitlines, while the cells in the remaining rows stay isolated. Each column corresponds to a bitline that carries the data to or from the bit cells. When a wordline is activated, the bit cells in that row interact with their bitlines. If the memory is being read, the stored bits in that row drive the bitlines, producing a data word at the output. If the memory is being written, the bitlines are driven with the new value, and activating the wordline forces the selected bit cells to store those values.

This layout helps visualize how the array implements the logical view of memory: selecting one row with the address corresponds physically to asserting exactly one wordline, and the parallel data word corresponds to the bitlines connected to the chosen row.

11.3.3 Dynamic Random Access Memory (DRAM)

Dynamic RAM, or DRAM, stores each bit of information using a **tiny capacitor**. This capacitor holds electrical charge, and a single transistor acts as a switch that either connects the capacitor to the bitline or isolates it from the rest of the circuit:



When the capacitor is charged up to the supply voltage, the memory cell represents a logical 1; when it is discharged to ground, it represents a logical 0. This extremely compact structure (just one transistor and one capacitor per bit) makes DRAM the most space-efficient memory technology available. As a result, DRAM can store vastly larger amounts of data in the same silicon area, which explains why it is used for main memory in virtually all computers.

Writing a value into a DRAM cell is straightforward. The memory controller first drives the bitline to the desired value. Then it activates the appropriate wordline, turning on the access transistor. The capacitor becomes connected to the bitline, and the strong driver forces the charge on the capacitor to match the bitline level. When the wordline is de-asserted, the transistor turns off and the capacitor is left holding the stored charge.

Reading from a DRAM cell is more subtle because the stored charge is extremely small. Before a read operation, the bitline is precharged to a neutral midpoint. When the wordline is activated, the tiny charge on the capacitor slightly perturbs this bitline voltage. A sensitive amplifier detects the small deviation and amplifies it into a clean digital 0 or 1. However, **the act of reading removes charge from the capacitor**, disturbing the very bit that is being read. In other words, reading a DRAM cell **destroys** its stored value, so **the memory must immediately rewrite** the bit to restore the charge after every read. This is known as the "read-and-restore" process.

Even if the memory is not being accessed, the charge on the capacitors **gradually leaks away** due to tiny imperfections in the transistors and dielectric materials. For that reason, DRAM requires **refreshing**: every cell must be periodically read and rewritten to prevent data loss. Refresh cycles typically occur every few milliseconds, managed automatically by the DRAM controller. This periodic maintenance is what makes DRAM "dynamic": the data must be actively maintained over time!

The modern DRAM cell can be traced back to the invention of Robert Dennard at IBM in 1966.



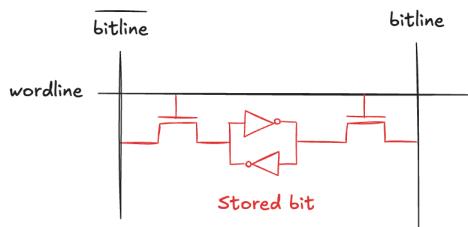
Robert Dennard
(1932, 2024)

He invented DRAM (1966) at IBM. Many were skeptical, but by mid-70 DRAM was in all computers!
He received 35 patent in semiconductors and microelectronics.

Although initially met with skepticism, Dennard's concept proved transformative. By the mid-1970s, DRAM had become the dominant memory technology in computers.

11.3.4 Static Random Access Memory (SRAM)

Static RAM, or SRAM, stores each bit using a pair of cross-coupled inverters, essentially a **tiny bistable circuit** that can hold its state indefinitely as long as power is supplied.



This bistable structure has two stable configurations: one corresponding to a logical 0 and the other to a logical 1. Because the inverters continuously reinforce each other, the stored bit does not leak away or degrade over time, which means that SRAM **does not require periodic refresh cycles** like DRAM.

To interact with the stored value, each SRAM bit cell uses two access transistors controlled by a wordline. When the wordline is inactive, the cross-coupled inverters are isolated from the outside world, maintaining their internal state without disturbance. When the wordline is asserted, both access transistors turn on, connecting the internal storage node to a pair of complementary bitlines. These bitlines carry both the value and its inverse, which improves noise immunity and allows fast, reliable sensing during read operations.

Reading from an SRAM cell involves letting the stored bit influence the bitlines. Before the read, the bitlines are usually precharged to an intermediate level. When the wordline activates the access transistors, the stronger of the two inverter states pulls one bitline slightly down while the opposite node pushes the complementary bitline slightly up. A sense amplifier detects this differential change and amplifies it into a clean digital result. The important point is that this read operation does not disturb the stored data, the inverters are strong enough to preserve the original value even while driving the bitlines.

Writing, on the other hand, requires overpowering the existing state of the cell. To accomplish this, the memory controller drives the bitlines with the new value and its complement. When

the wordline is asserted, these externally driven voltages override the inverters long enough to flip the cell into the new state. Once the wordline is lowered again, the cross-coupled inverters resume reinforcing the updated bit value.

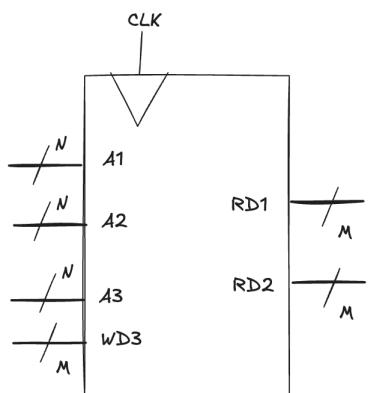
Because SRAM cells maintain their state without needing refresh and can be **accessed very quickly**, they are ideal for caches, register files, and other high-speed memory structures. However, the drawback is that an SRAM cell requires **significantly more transistors** than a DRAM cell (typically six instead of one) which makes it more expensive in terms of silicon area. As a result, SRAM is used sparingly where speed is critical, while DRAM is used for larger, slower main memory.

11.3.5 Register Files

A register file is a **small, fast storage structure** composed of a **set of registers** used to hold temporary values during computation. Unlike main memory, which is large but relatively slow, register files are designed for **rapid access** and are therefore placed very close to the processing core. The values stored here typically represent intermediate results, loop counters, addresses, operands, and other short-lived data items that the CPU needs to access frequently.

Rather than being built from individual flip-flops, register files are usually implemented as **compact, multi-ported SRAM arrays**. This approach is far more space-efficient, since SRAM cells require fewer transistors per bit than flip-flops. The term "multi-ported" means that the register file supports multiple simultaneous accesses. A processor often needs to read two operands and write back a result in every clock cycle, so the register file must be able to handle several independent addresses at the same time without conflicts.

Consider a common example: a register file containing 32 registers, each 32 bits wide.



Because the file holds 32 entries, the address size is 5 bits. A typical three-ported design includes two read ports and one write port. The addresses A1 and A2 select the two registers to be

read, producing the read data RD1 and RD2. Meanwhile, A3 selects the register to be written, and WD3 provides the data to store. With this organization, the register file can deliver two operands and accept one result within the same cycle, which matches the requirements of many instruction set architectures.

In some designs, one register is **assigned a special purpose**: it is hardwired to always return zero when read. This is useful because zero is one of the most frequently used constants in programs. Hardwiring a dedicated zero register avoids the need to store or compute zero explicitly, simplifying certain instructions and saving both time and hardware resources.

Overall, register files play a crucial role in processor performance. By combining fast access, controlled size, and multiple ports, they provide the high-speed data stream required to keep the execution units busy every cycle.

11.3.6 Volatile memories comparison

Although flip-flops, SRAM, and DRAM are all forms of volatile memory (they lose their contents when power is removed) they differ dramatically in how much hardware they require and how quickly they can respond. A simple comparison highlights these differences:

Memory Type	Transistors per Bit Cell	Latency
Flip-Flop	20	Fast
SRAM	6	Medium
DRAM	1	Slow

Flip-flops are the **fastest option**. Because they are built from robust logical elements, the stored value is immediately available at the output without waiting for any sensing or amplification. However, this speed comes at the price of **significant silicon area**, which makes flip-flops practical only for **tiny memories** such as pipeline registers or control-state storage inside a processor.

SRAM sits in the middle ground. It is **not as fast as a flip-flop**, because reading or writing requires the access transistors and bitlines to interact with the cross-coupled inverters. Still, SRAM provides **excellent performance** with much lower area cost, making it the natural choice for **caches and register files** where both speed and capacity matter.

DRAM is **slower** still. Because the stored charge in a capacitor is tiny, reading a DRAM cell involves waiting for that charge to influence the bitline slightly before a sense amplifier can

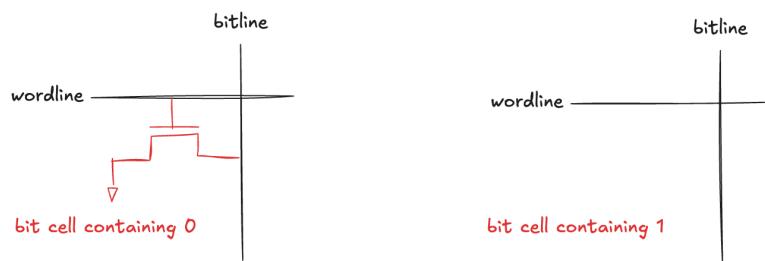
detect it. Writing also requires time for the bitline driver to charge or discharge the capacitor. On top of this, DRAM must periodically refresh its contents so that the stored charge does not leak away. These refresh cycles add overhead and contribute to latency. Despite this, DRAM's **incredibly compact cell structure** makes it ideal for **very large memories** such as the main memory of computers.

Modern DRAM technologies have evolved significantly. Synchronous DRAM, and especially double-data-rate (DDR) variants, use both rising and falling edges of the clock to transfer data, effectively doubling throughput. The first DDR standard appeared around the year 2000 with effective data rates near 100 MHz. Today, DDR5 achieves multi-gigahertz transfer rates (exceeding 5 GHz) thanks to sophisticated signaling and parallelism. DDR6 technology is already in development and announced for 2027, promising even higher speeds and efficiency improvements.

Choosing the right memory technology for a particular system **depends on the requirements** of that design. If maximum speed is critical, flip-flops or SRAM may be the best option. If large capacity is needed and slower access is acceptable, DRAM is the natural choice. **Designers must always balance speed, cost, power consumption, and area** to select the most appropriate memory structure for their application.

11.3.7 Read Only Memory (ROM)

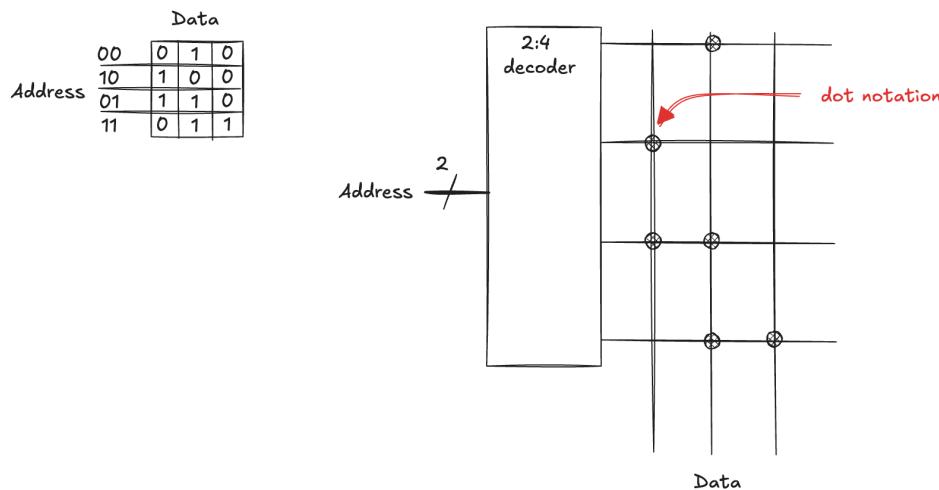
Read Only Memory, or ROM, is one of the simplest forms of digital storage. Unlike RAM, which repeatedly rewrites or refreshes its contents, a ROM stores information permanently using the physical structure of the circuit. Each bit in a ROM cell is defined by **whether a particular transistor is present or absent in the layout**. This structural encoding makes ROM inherently non-volatile, its data persists even when power is removed.



Reading a ROM cell relies on very simple behavior. Each bitline is initially weakly pulled high, meaning it sits at a logical 1 unless something actively pulls it down. When a read occurs, the memory activates one wordline corresponding to the selected row. Turning on this wordline connects the bit cell to the bitline, but only if a transistor is actually present at the crossing point. If the transistor exists, it forms a conducting path to ground and pulls the bitline low, producing

a logical 0. If the transistor is absent, no conduction occurs; the bitline stays high, and the read value is interpreted as a logical 1. Thus, the physical presence of the device encodes a 0, while its absence encodes a 1.

Because ROM bits are **fixed during fabrication**, the pattern of transistors in the array fully defines its contents. Designers often describe the programmed values using **dot notation***, where a dot represents the presence of a transistor at the intersection of a wordline and a bitline.



The visual grid of dots provides a direct representation of how the ROM is wired internally. This direct physical encoding makes ROM extremely compact and reliable, and it is **widely used for storing constants, lookup tables, microcode, and any data that must remain unchanged over the lifetime of a system**.

11.3.8 Programmable ROM

A programmable ROM, or PROM, is a refinement of the simple mask-programmed ROM. Instead of deciding during fabrication whether a transistor is present or absent in each bit cell, a PROM includes a transistor in every location by default. What makes a bit a logical 0 or 1 is no longer the presence of the device, but whether that transistor is electrically connected to ground. The key idea is to give the user a way to **permanently connect or disconnect this path after manufacturing**.

Programming a PROM involves applying a high voltage to specific locations in order to "blow" **tiny fuses**. When a fuse is intact, the transistor in that bit cell remains connected to ground.



During a read, activating the wordline allows this transistor to pull the bitline low, representing a stored 0. If the fuse is blown, the connection to ground is physically broken, leaving the transistor isolated. With no discharge path, the bitline remains high when the wordline is enabled, and the cell therefore represents a 1. In this way, destroying a fuse encodes a permanent logical 1.

Because each fuse can be blown only once, a traditional PROM is a **one-time programmable device**. Once a cell has been programmed, its contents cannot be changed or repaired. This irreversible step is often described as "burning" the ROM, a name that reflects both the physical destruction of the fuse and the permanence of the resulting data pattern. PROMs are well suited for **storing configuration data, firmware, or lookup tables that must never change after deployment**.

11.3.9 Reprogrammable ROM

Classic PROMs rely on irreversible fuses, later technologies introduced **reversible mechanisms** that allow connections to be made or broken multiple times. These reprogrammable devices (such as EPROM, EEPROM, and flash memory) extend the core idea of PROM by providing a way to modify stored data without replacing the chip, while preserving the compact layout and simple read mechanism of ROM-based structures.

While traditional PROMs can be written only once, later technologies introduced mechanisms that allow ROM contents to be **erased and rewritten**. The first major step was the development of **EPROMs**, or Erasable Programmable Read-Only Memories. Instead of using a fuse to store a bit, an EPROM relies on a special type of transistor called a floating-gate transistor. This device has an extra gate that is electrically isolated by an insulator, meaning it is not physically connected to any other wires in the circuit. Electrons can be placed onto or removed from this floating gate, and their presence or absence determines whether the transistor conducts.

To program an EPROM, a high voltage is applied so that electrons tunnel through the insulator and become trapped on the floating gate. When enough charge accumulates, the transistor

switches on, allowing the bitline to be pulled toward the selected wordline during a read—representing a programmed state. Erasing an EPROM requires exposing the chip to intense ultraviolet light for an extended period, often around thirty minutes. The high-energy photons dislodge the trapped electrons, emptying the floating gate and turning the transistor off again. Because these two operations (injecting charge and removing it) change the stored value, they are referred to as programming and erasing. This mechanism made EPROMs reusable, but the need for UV exposure also made the process slow and inconvenient.

The next evolution was EEPROMs, Electrically Erasable PROMs, and eventually **Flash memory**. These devices use the same underlying principle of floating-gate transistors but add circuitry that allows both programming and erasing to occur electrically, without any need for UV light. This capability made reprogramming far more practical, enabling updates to be made directly within a system. Flash memory, in particular, became extremely widespread because it provides high density, low cost, and good durability. By 2024, Flash memory could be purchased for only a few cents per gigabyte, making it the dominant storage technology in portable devices, solid-state drives, embedded systems, and countless consumer electronics.



Fujio Masuoka
(1943–)

Invented Flash memory at Toshiba during the late 1970s as an unauthorized project he pursued during nights and weekends. The name "Flash" was inspired by the way the memory is erased, resembling the quick flash of a camera. Although Toshiba developed the technology, they were slow to commercialize it, allowing Intel to bring the first Flash memory product to market in 1988.

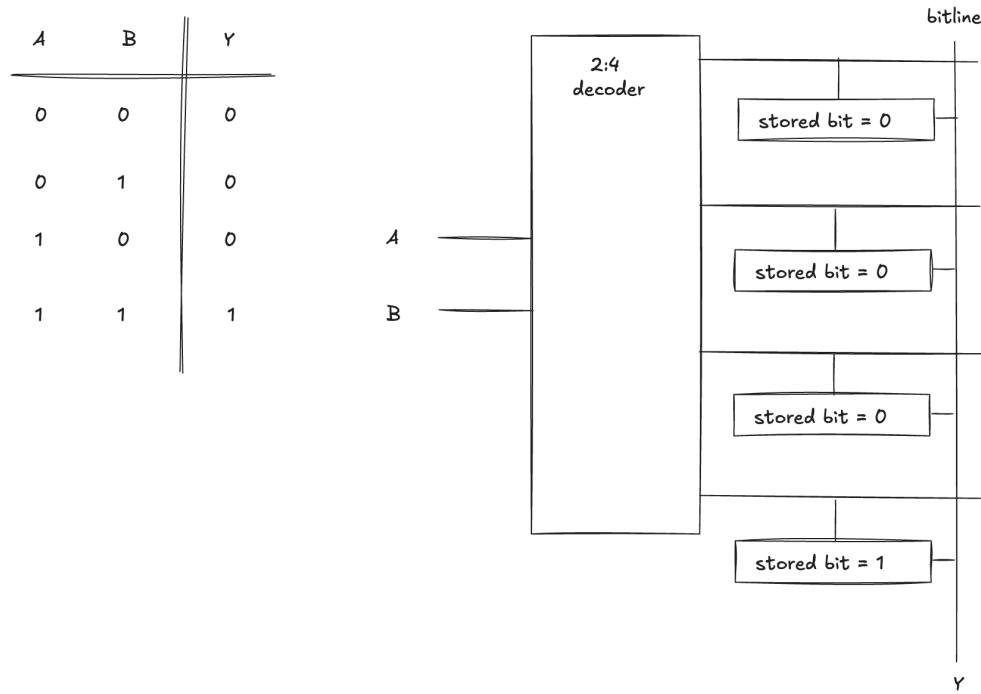
Although the term "Read-Only Memory" suggests a technology that cannot be changed, modern ROM variants clearly blur that boundary. What truly differentiates ROM from RAM is not the ability to write but the nature of how writing occurs. ROM-based technologies take considerably longer to program or erase, but they retain their contents without power. This non-volatility is what makes them essential for storing firmware, configuration data, and any information that must persist reliably from one power cycle to the next.

11.4 Programmable Hardware

Programmable hardware allows digital systems to implement new logic functions simply by **changing configuration** data rather than **altering the physical circuit**. By storing patterns that describe a desired function, devices such as Lookup Tables, Programmable Logic Arrays, and Field-Programmable Gate Arrays can adapt their behavior to a wide range of applications, offering the flexibility of software combined with the performance of dedicated hardware.

11.4.1 Lookup Tables (LUT)

Although we usually think of memories as devices for storing data, a memory array can just as easily be used to **implement combinational logic**. The key observation is that a **truth table is itself a small memory**: each combination of input variables corresponds to a particular row, and each row contains the output values that the logic function should produce. If we treat the **inputs of the logic function as the address lines of a memory**, and **we store the desired output bits as the data stored in that memory**, then **reading the memory is equivalent to evaluating the logic function**. A memory with 2^N words and M bits per word can therefore implement any combinational function with N inputs and M outputs. The decoder chooses one of the 2^N rows based on the input pattern, and the bits stored in that row become the output of the function. This arrangement is known as a **lookup table**, or LUT, because the memory "looks up" the output value associated with the given input. To make this idea concrete, consider a simple example with two inputs, A and B, and one output Y. The function we want to implement is $Y=AB$ (a logical AND):

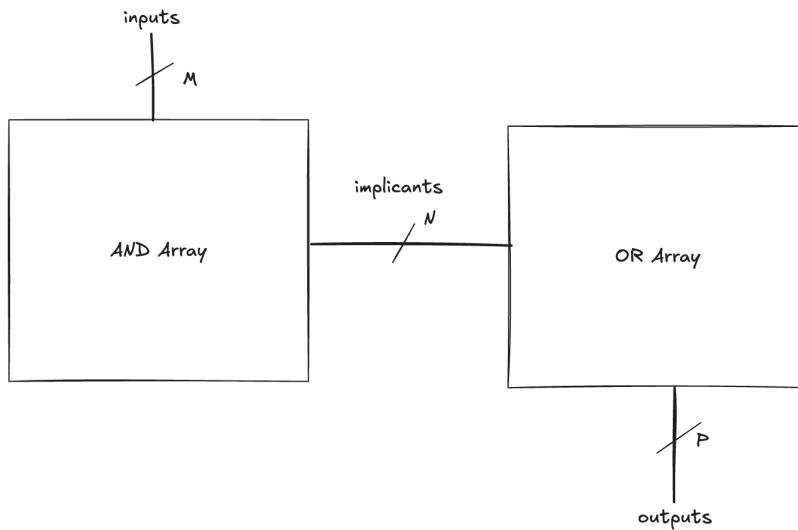


This technique generalizes to more complex functions and larger numbers of inputs and outputs. **Rather than wiring together many logic gates, designers can simply program the memory contents.** For this reason, LUTs are the **fundamental building blocks of FPGAs**, where each configurable logic block contains a small memory array used to implement arbitrary logic functions specified by the user.

11.4.2 Programmable Logic Array (PLA)

A Programmable Logic Array, or PLA, is built on the idea that logic gates can be arranged in regular, grid-like structures, much like memory cells. If the connections inside these grids are made programmable, the array can be configured after fabrication to implement a wide range of logic functions. This approach provides a structured alternative to wiring individual gates by hand, while still allowing considerable flexibility.

PLAs are designed specifically to implement **two-level combinational logic** in the classic sum-of-products form. The circuit begins with the set of input signals, provided both in their true and complemented versions. These signals feed into a programmable array of AND gates. By choosing which connections are present and which are absent, the AND array generates a collection of product terms—also known as implicants—that correspond to the logical conditions relevant to the function being implemented. The outputs of the AND array then flow into a second programmable array made of OR gates. Here, each output is formed by selecting which implicants to combine. Because the AND array can generate any desired set of minterms or simplified implicants, and the OR array can combine them in any pattern, the PLA can realize any two-level combinational logic function.



This organization offers a powerful and compact way to implement logic functions that would otherwise require many separate gates and hand-designed interconnections. It also highlights a recurring theme in digital design: by organizing components into predictable arrays and making connections programmable, complex behavior can be achieved through configuration rather than detailed custom wiring.

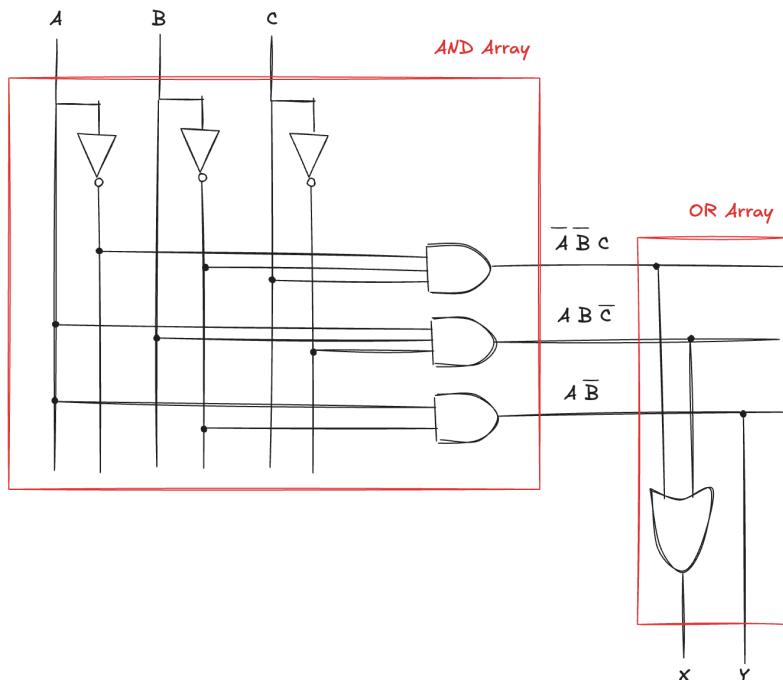
A PLA can be understood most clearly through a concrete example. Suppose we want to implement two output functions of three inputs, such as

$$X = \overline{A} \overline{B} C + A B \overline{C}$$

and

$$Y = A \overline{B}$$

These expressions illustrate the essence of a PLA: each function is written in sum-of-products form, and each product term is generated by the programmable AND array. The inputs, provided in both true and complemented form, feed rows of potential connections. Wherever a connection is present, that input contributes to a particular implicant. Each implicant then becomes an input to the OR array, which selects which product terms participate in producing each output. By programming these two planes of connections, the PLA generates exactly the logic required:

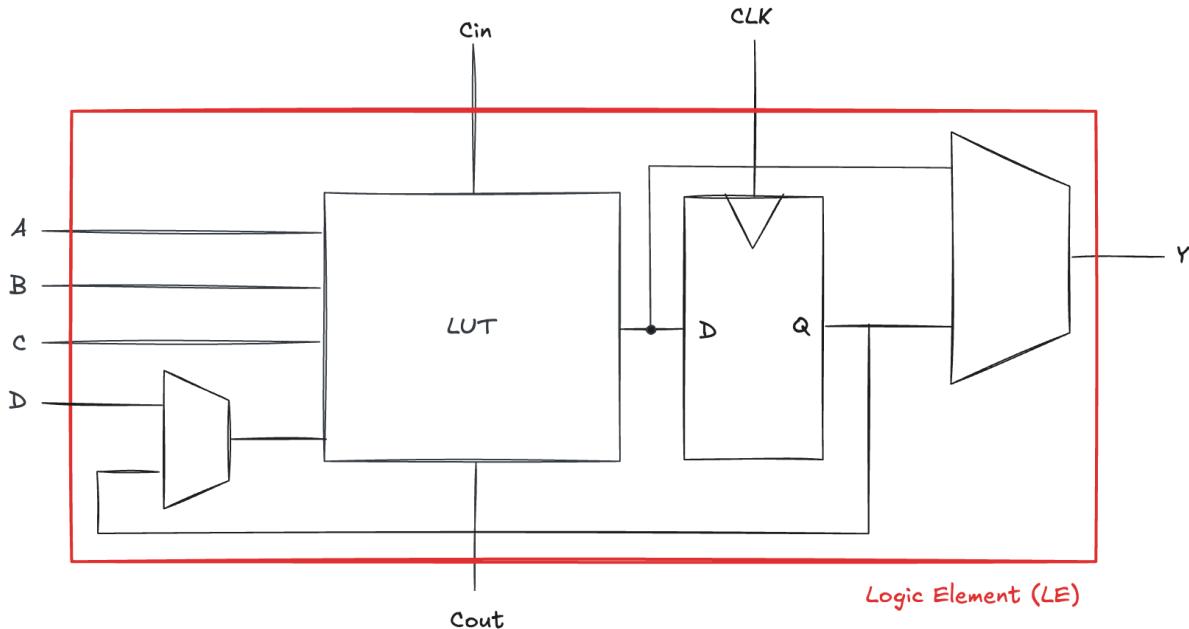


Although both LUTs and PLAs can implement arbitrary combinational logic, LUTs are preferred in modern FPGAs because they act as small memory blocks that map any truth table with uniform speed and structure, making them far more flexible, easier to configure, and more scalable than the two-level, connection-intensive architecture of PLAs.

11.4.3 Field-Programmable Gate Array (FPGA)

An FPGA is built around a **large array of reconfigurable Logic Elements (LE)** that the designer can program to perform virtually any digital function. Instead of hardwiring gates during fabrication, the chip provides a pool of flexible resources that can be configured after manufacturing

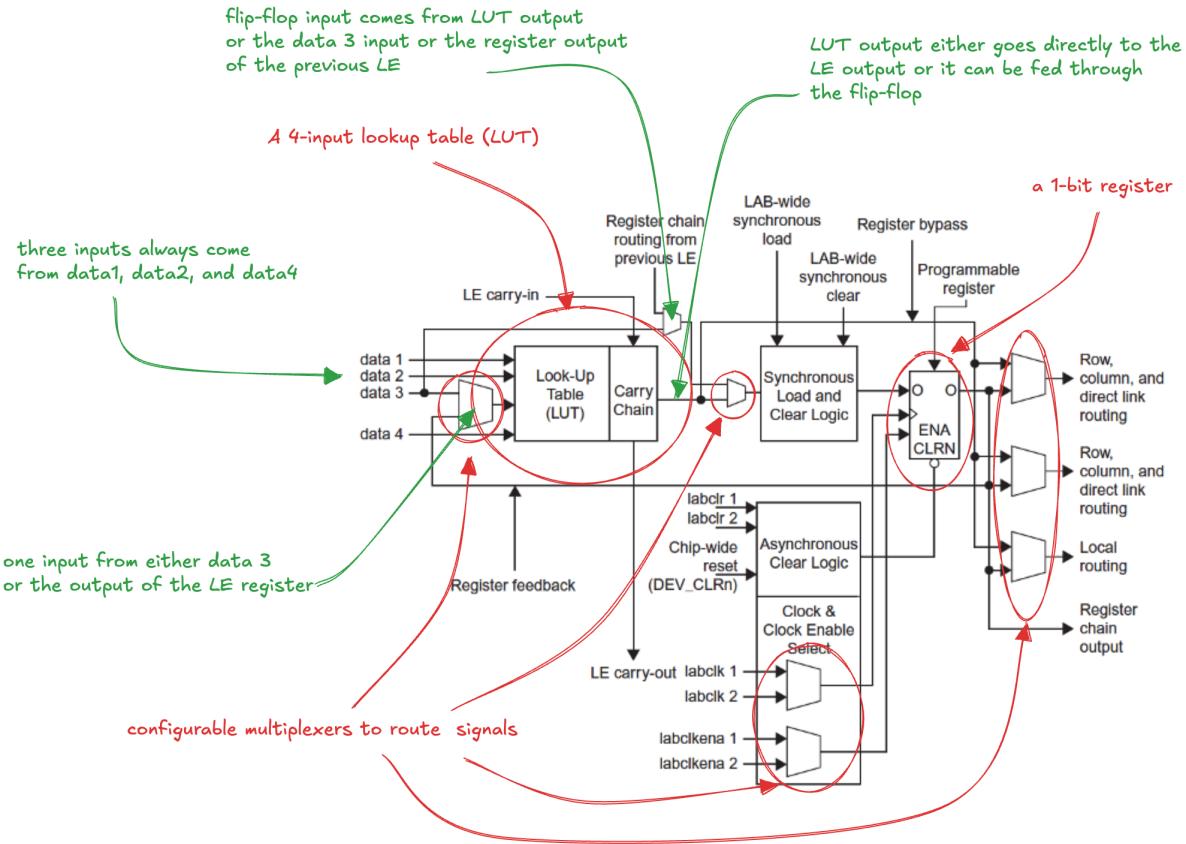
using a hardware description language or schematic. Each LE can implement combinational logic (typically using a LUT) and can also include flip-flops or other circuitry to support sequential behavior. Below we can consider a highly simplified diagram of a typical LE:



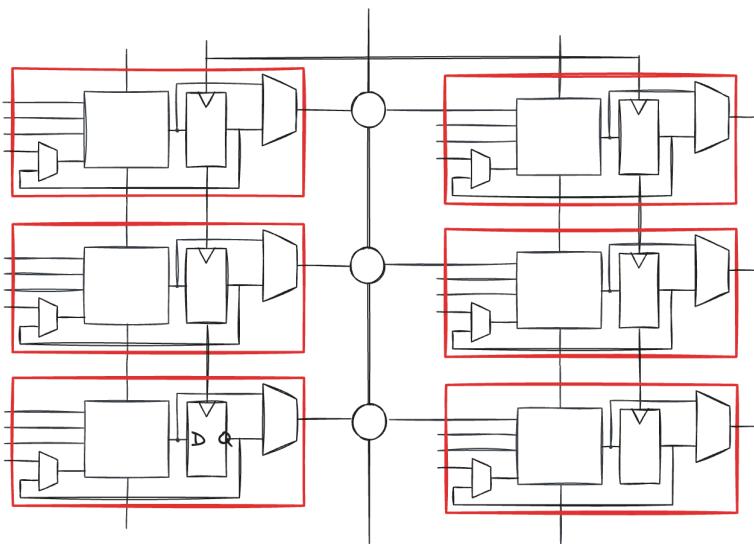
Four input signals A , B , C and D enter the logic element and feed a lookup table. The LUT has been configured to implement a specific logic function: for every possible combination of input bits, it stores the output value that the circuit designer intends. This allows the LUT to act like any Boolean gate or combination of gates. One of the inputs (D) first passes through a multiplexer. This selection gives the logic element flexibility to choose between different internal or external signals before they reach the LUT. In addition to these data inputs, the block includes a dedicated carry-in signal, C_{in} , which connects to the LUT and is part of the fast arithmetic chain used for operations such as addition. The corresponding carry-out, C_{out} , leaves the logic element so that neighbouring elements can be linked together efficiently. The output of the LUT can go in two different directions. It may be sent straight to the output multiplexer, providing a purely combinational function whose output changes immediately in response to changes at the inputs. Alternatively, the LUT output may be routed into a D-type flip-flop. This flip-flop captures the value of the LUT on the rising edge of the clock and holds it stable until the next clock event. The presence of the flip-flop enables the FPGA to build synchronous sequential circuits where timing and state play an essential role. The output of the logic element is chosen by a multiplexer that selects either the direct LUT value or the stored value from the flip-flop. The result Y becomes the observable output of the element and can be routed to other logic elements or off-chip pins. Even though this representation is simplified, it shows the key idea: a logic element combines **flexible combinational logic, optional state storage, and fast arithmetic**.

metic support, and can be connected with many other elements to construct complex digital systems inside an FPGA.

We can consider the **internal structure of a real logic element**, the following diagram shows an LE from Intel's Cyclone IV architecture:



A single logic cell is a compact and elegant building block, but **the real power of an FPGA emerges from scale**. Each cell can be connected (flexibly and dynamically) to many others, allowing thousands of these simple elements to cooperate in implementing complex digital systems:

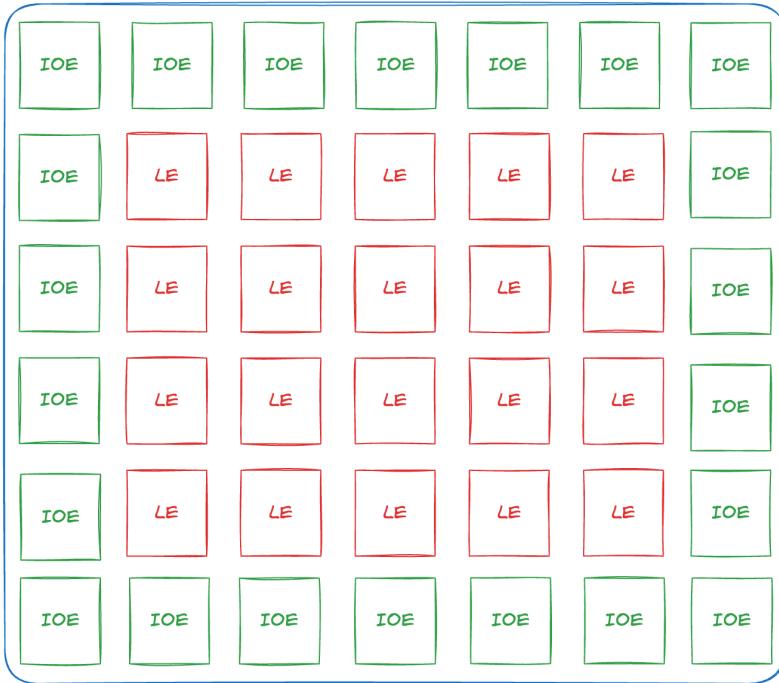


The large circles in the middle of the diagram represent **programmable routing switches** that allow the output of a logic cell to be directed almost anywhere on the chip. This drawing is, of course, highly simplified; real FPGAs contain a much denser web of interconnects, with many more wires and switches than we could reasonably show here.

Carry-in and carry-out pins become important when implementing arithmetic. Although each LUT typically contains only a one-bit adder, the carry chain allows these one-bit slices to be linked together to form wider adders and counters. Because these carry paths are dedicated, fast, and bypass much of the general routing fabric, they give FPGAs a significant performance edge over other forms of programmable logic.

Digital circuits depend heavily on clock signals, so robust clocking resources are essential. Every flip-flop in every logic cell must receive a clean, low-skew clock edge, regardless of where it is located. To achieve this, FPGAs include a network of global clock lines engineered to distribute precise, low-distortion clock signals across the entire device.

Surrounding this sea of logic elements are **input/output elements** that connect the internal circuitry to the pins of the chip package, allowing the FPGA to communicate with the outside world:



Modern FPGAs go far beyond this. By integrating **additional hardware blocks** directly on the chip, the device gains substantial functionality that cannot be matched by assembling equivalent components purely from logic cells. Dedicated circuits are almost always faster, more power-efficient, and significantly smaller in silicon area than their soft logic equivalents. One of the most important built-in resources is **memory**. Nearly all FPGAs include dedicated SRAM banks distributed across the chip. Another major enhancement in modern devices is the inclusion of **DSP slices**. These are **dedicated arithmetic units designed for high-performance digital signal processing**. A typical DSP slice includes a hardware multiplier and several adders. Many FPGAs can integrate also a variety of specialized peripherals, like analog-to-digital and digital-to-analog converters. Together, these hard peripherals expand the FPGA's capabilities far beyond what could be achieved with logic cells alone, making the device a flexible and powerful platform for building complex digital systems. Although different manufacturers organize their FPGAs in slightly different ways, the fundamental ideas remain the same across devices. For example, Intel's Cyclone IV uses four-input LUTs inside each logic element, while Xilinx's 7-series families use six-input LUTs grouped into slices. Despite these architectural variations, the role of the LUTs, multiplexers, registers, and routing fabric is essentially identical: together, they form a reconfigurable platform capable of implementing a vast range of digital circuits.

The behavior of an FPGA is described using a hardware description language. Designers typically begin by writing a behavioral model and then thoroughly simulating it to uncover and correct functional errors before any hardware is configured. Once the design behaves correctly in simulation, the next step is to map it onto the actual FPGA. The toolchain first **synthesizes** the

HDL into a **netlist**, a detailed list of the logic elements, signals, and interconnections required to implement the design. A **place-and-route** algorithm then determines where on the FPGA each logic element should reside and how the interconnect network should be used to connect them. This stage must balance multiple goals (performance, power efficiency, and resource utilization) making it a **complex optimization problem**. After placement and routing are complete, the design can be **programmed** into the FPGA. The **configuration bitstream** sets the contents of all LUTs, programs the multiplexers and routing switches, and initializes every I/O and clocking resource so that the device behaves exactly as specified. Once the synthesis and place-and-route stages are complete, the resulting configuration information is downloaded to the FPGA. Many devices are **volatile**: when power is removed, the configuration disappears, and the device must be reprogrammed at the next startup. To address this, systems typically include an external flash memory that stores the configuration bitstream and automatically reloads it into the FPGA upon power-up. Most development boards provide this capability out of the box. The DEEDS simulator can generate HDL descriptions from schematics, enabling students to transition seamlessly from graphical design to deployable FPGA implementations. As you will see in the next semester, this workflow makes it possible to take a circuit you have drawn, synthesize it automatically, and run it directly on a real FPGA development board.

11.4.4 Comparison: FPGA vs. ASIC

Application-Specific Integrated Circuits (ASICs) are chips fabricated with a fixed, unchangeable hardware design. Because they are custom-built for a particular task, ASICs generally outperform FPGAs in terms of speed, power efficiency, and silicon area. FPGAs introduce overhead due to their programmable routing and configurable logic, which makes them slower and more power-hungry in most cases. However, the gap has narrowed: modern high-end FPGAs can approach ASIC performance for certain applications.

Despite this, FPGAs offer two major advantages. The first is **cost**. Producing an ASIC requires not only the behavioral design but also a full physical layout, mask creation, and wafer fabrication: an extremely expensive process. For low or moderate production volumes, FPGA-based systems are dramatically cheaper. The second advantage is **flexibility**. An FPGA can be reprogrammed any number of times, whereas an ASIC is permanent once manufactured. Even companies that ultimately plan to build an ASIC often **use FPGAs during prototyping** because the behavioral design is nearly identical, and debugging is vastly easier when the hardware can be reconfigured instantly.

The combination of reconfigurable logic, abundant routing resources, and versatile I/O makes FPGAs a powerful and adaptable device. They support both combinational and sequential designs, can be reprogrammed repeatedly, and scale well to large, complex applications. These

advantages explain why FPGAs have become common in many modern products. For instance, a Mercedes-Benz S-Class contains more than a dozen Xilinx FPGAs, taking advantage of their flexibility, performance, and rapid time-to-market. Their ability to be reconfigured after deployment also makes them ideal for prototyping, updating functionality, and building systems that may need to evolve over time.