Computer Science Fundamentals:

Intro to Algorithms, Systems, & Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

Big thanks to **Christine Papadakis-Kanaris**
for teaching Intro. to Computer Science II,
Dora Erdos and **Adam Smith**
for teaching BU CS330: Introduction to Analysis of Algorithms
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne,
at Boston University

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

Theorem 0.1: Common Derivatives

Power Rule: For $n \neq 0$	$\frac{d}{dx}(x^n) = n \cdot x^{n-1}$. E.g., $\frac{d}{dx}(x^2) = 2x$
Derivative of a Constant:	$\frac{d}{dx}(c) = 0$. E.g., $\frac{d}{dx}(5) = 0$
Derivative of ln:	$\frac{d}{dx}(\ln x) = \frac{1}{x}$
Derivative of $\log_a$:	$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$
Derivative of $\sqrt{x}$:	$\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}}$
Derivative of function $f(x)$:	$\frac{d}{dx}(x) = 1$. E.g., $\frac{d}{dx}(5x) = 5$
Derivative of the Exponential Function:	$\frac{d}{dx}(e^x) = e^x$

Theorem 0.2: L'Hopital's Rule

Let $f(x)$ and $g(x)$ be two functions. If $\lim_{x\to a} f(x) = 0$ and $\lim_{x\to a} g(x) = 0$, or $\lim_{x\to a} f(x) = \pm\infty$ and $\lim_{x\to a} g(x) = \pm\infty$, then:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Where $f'(x)$ and $g'(x)$ are the derivatives of $f(x)$ and $g(x)$ respectively.

Theorem 0.3: Exponents Rules

For $a, b, x \in \mathbb{R}$, we have:

$$x^a \cdot x^b = x^{a+b} \text{ and } (x^a)^b = x^{ab}$$

$$x^a \cdot y^a = (xy)^a \text{ and } \frac{x^a}{y^a} = \left(\frac{x}{y}\right)^a$$

Note: The $:=$ symbol is short for "is defined as." For example, $x := y$ means x is defined as y.

Definition 0.1: Logarithm

Let $a, x \in \mathbb{R}$, $a > 0$, $a \neq 1$. Logarithm x base a is denoted as $\log_a(x)$, and is defined as:

$$\log_a(x) = y \iff a^y = x$$

Meaning *log* is inverse of the exponential function, i.e., $\log_a(x) := (a^y)^{-1}$.

Tip: To remember the order $log_a(x) = a^y$, think, "base a," as a is the base of our *log* and y.

Theorem 0.4: Logarithm Rules

For $a, b, x \in \mathbb{R}$, we have:

$$\log_a(x) + \log_a(y) = \log_a(xy) \text{ and } \log_a(x) - \log_a(y) = \log_a\left(\frac{x}{y}\right)$$

$$\log_a(x^b) = b \log_a(x) \text{ and } \log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

Definition 0.2: Permutations

Let $n \in \mathbb{Z}^+$. Then the number of distinct ways to arrange n objects in order is $n! := n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. When we choose r objects from n objects, it's Denoted:

$$^nP_r := \frac{n!}{(n-r)!}$$

Where $P(n, r)$ is read as "n permute r."

Definition 0.3: Combinations

Let n and k be positive integers. Where order doesn't matter, the number of distinct ways to choose k objects from n objects is it's *combination*. Denoted:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Where $\binom{n}{k}$ is read as "n choose k.", and $\binom{\cdot}{\cdot}$, the *binomial coefficient*.

Theorem 0.5: Binomial Theorem

Let a and b be real numbers, and n a non-negative integer. The binomial expansion of $(a+b)^n$ is given by:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

which expands explicitly as:

$$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{n-1}ab^{n-1} + \binom{n}{n}b^n$$

where $\binom{n}{k}$ represents the binomial coefficient, defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

for $0 \leq k \leq n$.

Theorem 0.6: Binomial Expansion of 2^n

For any non-negative integer n, the following identity holds:

$$2^n = \sum_{i=0}^{n} \binom{n}{i} = (1+1)^n.$$

Definition 0.4: Well-Ordering Principle

Every non-empty set of positive integers has a least element.

Definition 0.5: "Without Loss of Generality"

A phrase that indicates that the proceeding logic also applies to the other cases. i.e., For a proposition not to lose the assumption that it works other ways as well.

Theorem 0.7: Pigeon Hole Principle

Let $n, m \in \mathbb{Z}^+$ with $n < m$. Then if we distribute m pigeons into n pigeonholes, there must be at least one pigeonhole with more than one pigeon.

Theorem 0.8: Growth Rate Comparisons

Let n be a positive integer. The following inequalities show the growth rate of some common functions in increasing order:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

These inequalities indicate that as n grows larger, each function on the right-hand side grows faster than the ones to its left.

— 1 —

Circuits and Logic

1.0.1 Building Transistors: The Chemistry of Silicon

To even begin to manage currents and voltages, we will need a way to control the flow of electricity:

Definition 0.1: Transistor

A **transistor** is a small electronic semiconductor device. A **semiconductor** (e.g., silicon) is a material with electrical conductivity between that of a **conductor** (great electricity conductor) and an **insulator** (inhibits electric flow). Transistors fall into two broad families:

- **Bipolar Junction Transistor (BJT):** a current-controlled device with three terminals (pins),
  - **Emitter (E):** current flows *out*.
  - **Base (B):** controls operation.
  - **Collector (C):** current flows *in*.
- **Field-Effect Transistor (FET):** a voltage-controlled device with three terminals,
  - **Source (S):** current flows *in*.
  - **Gate (G):** controls operation.
  - **Drain (D):** current flows *out*.

Low-power transistors are molded in an epoxy (resin) package. Higher-power transistors often use a metal tab or "can" that you bolt to a **heat sink** (a metal object that dissipates heat).

Pin order and package style vary by model; check the **part number** and manufacturer's **datasheet** for exact details [1, 7].

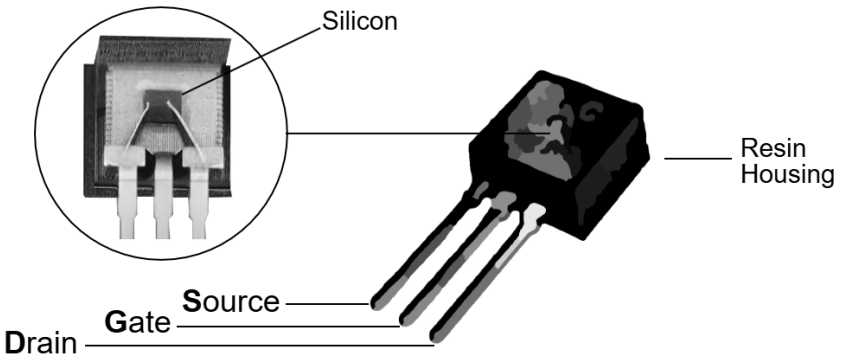


Figure 1.1: Cross-section of a discrete transistor: a silicon die (center) is bonded to three metal leads, all encased in an epoxy package. A metal tab (not shown) may be added for heatsinking.

Theorem 0.1: FETs over BJTs

A BJT needs continuous base current, which wastes energy. A MOSFET only requires its gate to be charged or discharged (i.e., voltage applied or removed), which is more efficient.

Now we briefly step into chemistry for completeness sake to understand differing silicon charges:

Definition 0.2: Anotomy of an Atom

An **atom** is the smallest unit of matter that retains the properties of an element. It consists of three main subatomic particles:

- **Protons:** Positively charged particles found in the nucleus.
- **Neutrons:** Neutral particles also found in the nucleus (same size as protons).
- **Electrons:** About the same charge as proton, but negative, and about 1800x smaller and lighter than a proton.

Protons and neutrons are tightly packed together in a space called the **nucleus**, gaining the name **nucleons**; Electrons orbit the nucleus at discrete distances called **shells** or **energy levels**. The number of protons in the nucleus defines the element (i.e., specifications). E.g., 79 protons will always be gold.

Opposite charges attract, causing an **orbital space**, in which subatomic particles never collide (i.e., alike orbiting planets). Neutrons act as a buffer between protons (e.g., Silver is stable with 60 or 62 neutrons, but unstable with 61). Atoms with different number of neutrons are called **isotopes**, latin for "same place". Electrons may jump between shells and atoms. If there is a greater number of electrons to protons, the atom is **negatively charged** (anions), otherwise it is **positively charged** (cations) [2].

Definition 0.3: Periodic Table

The **Periodic Table of Elements** organizes all known elements by the number of protons in their nuclei. This is called an **atomic number** (e.g., gold's atomic number is 79). Elements are abbreviated from their latin translations (e.g., gold is **aurum**, AU, which means "shining dawn"). There are 118 elements, with 80 being stable and the rest being unstable isotopes. Anything past 82 protons (lead) is unstable, undergoing radioactive decay.

Tip: The periodic table is complete, hence movies that claim "we discovered a new element!" truly deserve science-fiction as their defining genre.

We'll stop with the chemistry dive after these next two critical definitions

Definition 0.4: Shell Capacities & Valence Electrons

The first shell of any atom can hold up to 2 electrons, and the second 8. From 1-20 periodic elements, the third and fourth shells can hold 8 and 2 respectively. A *full* shell is considered **stable**, otherwise it is **unstable**. This arrangement of electrons within the shells is called the **electron configuration** (EC) of the atom. An EC is written as a n-tuple, starting with the inner-most shell (e.g., 2, 8, 8, 2 for calcium).

The outer most shell is called the **valence shell**. An atom's **valency** (the number of electrons in the valence shell) determines whether a chemical reaction will occur. If an atom is stable (i.e., full valence shell), it will not react with other atoms. Unstable atoms *strive* to become stable by either gaining, losing, or sharing electrons with other atoms [6].

Definition 0.5: Chemical Bonds – Molecules & Compounds

The act of atoms joining together (e.g., sharing electrons, which is called a **covalent bond**), forms a **molecule**. Concretely, a molecule is a merger of two or more elements. We use subscripts to denote the number of atoms in a molecule (e.g., H_2O is water, with two hydrogen atoms and one oxygen atom). **Compounds** are a subset class of molecules that consists only of two more more **different** elements (e.g., H_2O is a compound, but O_2 is not, as it only has one element, oxygen) [9].

Now to what we've been waiting for:

Definition 0.6: Doping – N-type & P-type Silicon

Silicon has 14 atoms, with an EC of (2, 8, 4); Hence silicon is unstable. If we view silicon (Si) as a 3D lattice (a string of Si atoms in 3D grid), each Si atom will share its four valence electrons with it neighbors to become stable (covalent bonding). This creates a **silicon crystal**.

Adding another element to the silicon lattice is called **doping**. We are interested in two types of doping [7]:

- **N-type:** When adding an element like phosphorus (P), EC of (2, 8, 5), is added to the silicon lattice, one electron goes unused after the covalent bonding. This free electron creates a **negative charge carrier** (hence N-type).
- **P-type:** Conversely, adding boron (B), EC of (2, 8, 3), creates a **positive charge carrier** (hence P-type). This is because boron won't have enough to share with its neighbors, causing **holes** (absence of electrons), overall lowering the density of electrons.

Let's visualize what we've learned so far:

Figure 1.2: An image of a silicon crystal [4].

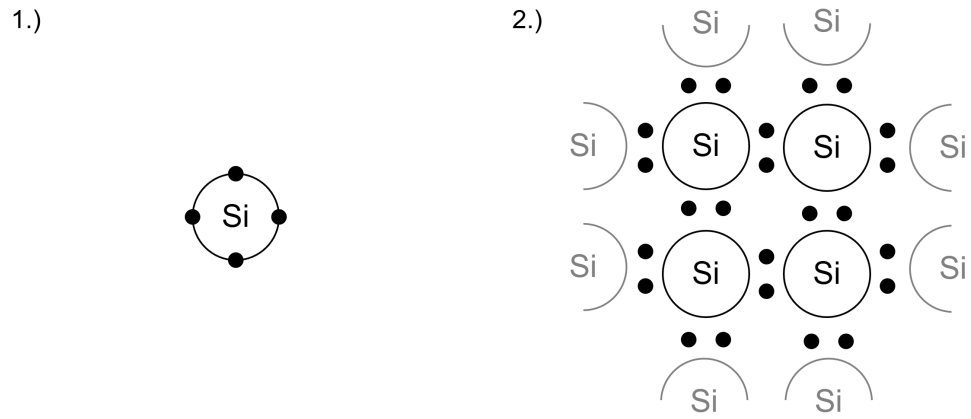


Figure 1.3: (1) Shows a single silicon atom (Si) and its valence electrons (4 black dots). (2) Shows a flattened silicon lattice where neighboring Si atoms share their electrons to become stable. This creates an electron configuration of (2, 8, 8) for surrounded Si atoms.

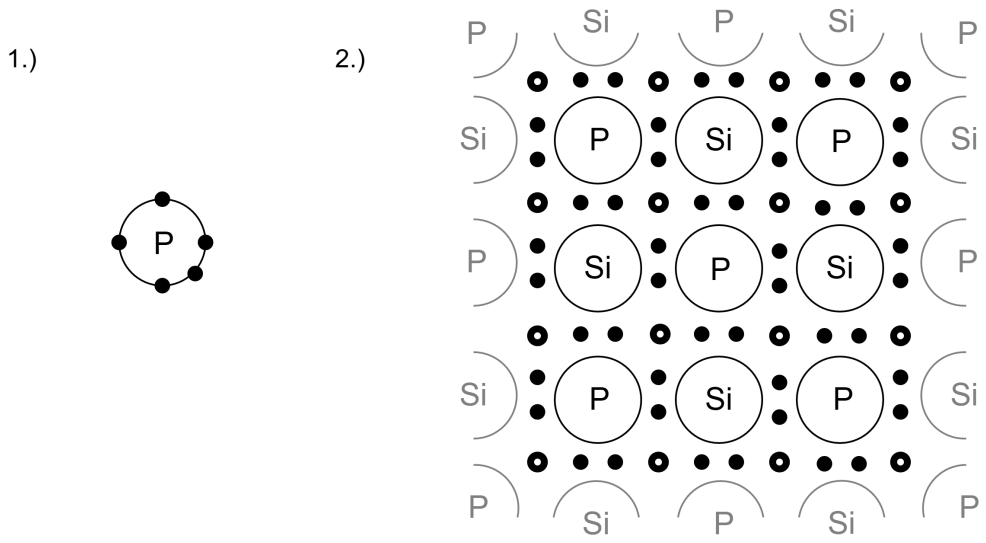


Figure 1.4: (1) Shows a single phosphorus atom (P) and its valence electrons (5 black dots). (2) A silicon lattice where phosphorus is doped, creating free electrons (thick dots with holes).

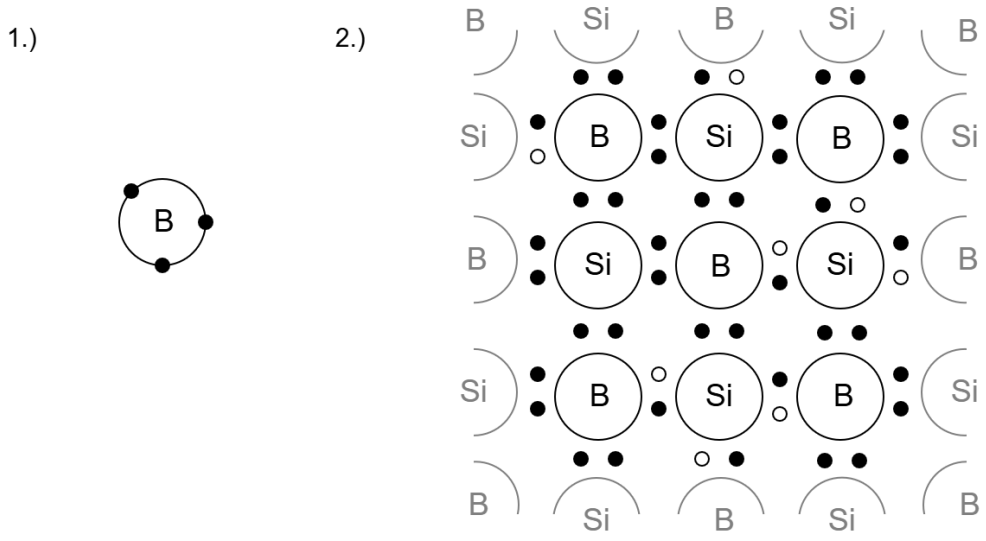


Figure 1.5: (1) Shows a single boron atom (B) and its valence electrons (3 black dots). (2) Shows a flattened silicon lattice where boron is doped, creating holes (halo dots).

Definition 0.7: N-type & P-type Junctions

When a N-type and P-type material are placed next to each other creates a **PN-junction**. At this junction, we get a **depletion region**, where N-type electrons and P-type holes cross; This leaves a slightly positive region on the N-type side and a slightly negative region on the P-type side. This manifests an electric field, creating a barrier, preventing further flow across the junction [7].

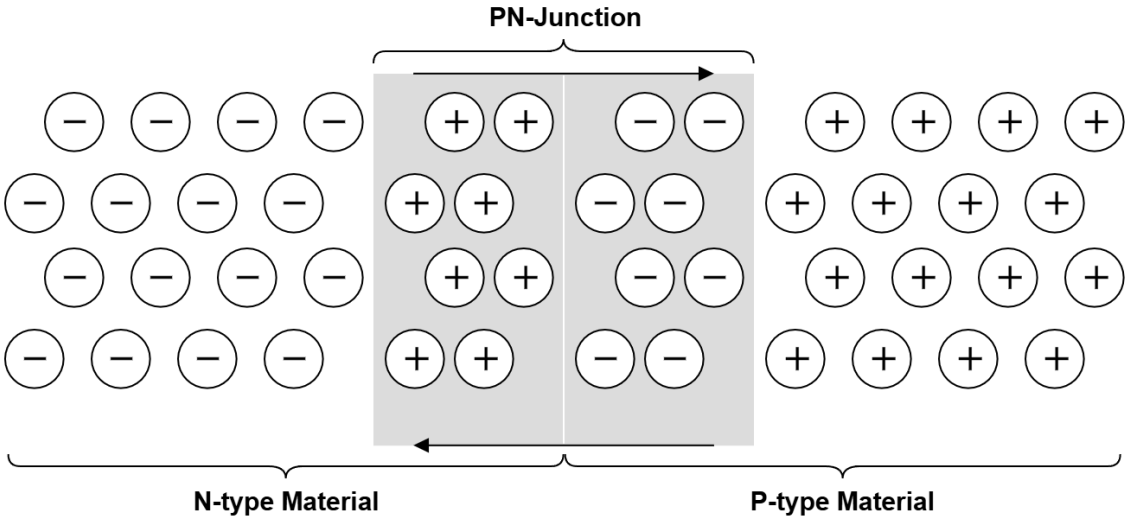


Figure 1.6: A PN-junction, where the depletion region is shown in gray.

Definition 0.8: MOSFET – High-Level Overview

A **MOSFET** (Metal-Oxide-Semiconductor Field-Effect Transistor) is a type of FET that uses its gate to control the flow of current. It consists of two default starting states:

- **Enhancement:** Normally **off** (i.e., no current flows), until such voltage is applied:
  - **N-Channel Enhancement:** A positive voltage.
  - **P-Channel Enhancement:** A negative voltage.
- **Depletion:** Normally **on** (i.e., current flows), until such voltage is applied:
  - **N-Channel Depletion:** A negative voltage.
  - **P-Channel Depletion:** A positive voltage.

Definition 0.9: MOSFET – Anatomy of N-Channel

A MOSFET **N-Channel Enhancement** is constructed as follows:

- **Substrate:** A base-layer of P-type material from which all parts will build upon.
- **Source & Drain:** Two motes are dug at either ends of the substrate and filled with N-type material; One for our **source** and the other for our **drain**. Two metal contacts are placed on these motes (our terminals); A body of metal is connected to the bottom of the substrate (**base/body terminal**), which connects to the source terminal.
- **Gate:** A metal contact pad is placed between the motes on top of the SiO_2 layer, forming the **gate terminal**. A layer of silicon dioxide (SiO_2) is sandwiched between the gate and the substrate. Since SiO_2 is a superb insulator, it prevents the gate terminal from touching/interacting with the substrate.
- **Channeling:** SiO_2 is a **dielectric** material, meaning that when a charge is applied to one side, the opposite charge builds on the other side, creating an electric field. When a positive charge is applied, it attracts negative electrons from the other side, creating a **channel** (i.e., bridge) between the two motes (source to drain), allowing current.

The **M**etal from gates, the **O**xide from the SiO_2 layer, the **S**emiconductor from the substrate and motes, and the **FET** from the field-effect, gives us the name **MOSFET**.

For **N-Channel Depletion**, A *thin* N-type substrate-channel is already present, bridging the two motes (source and drain) together. Once a negative charge is applied to the gate, positive holes are attracted, weakening the channel; This effectively stops the flow of current.

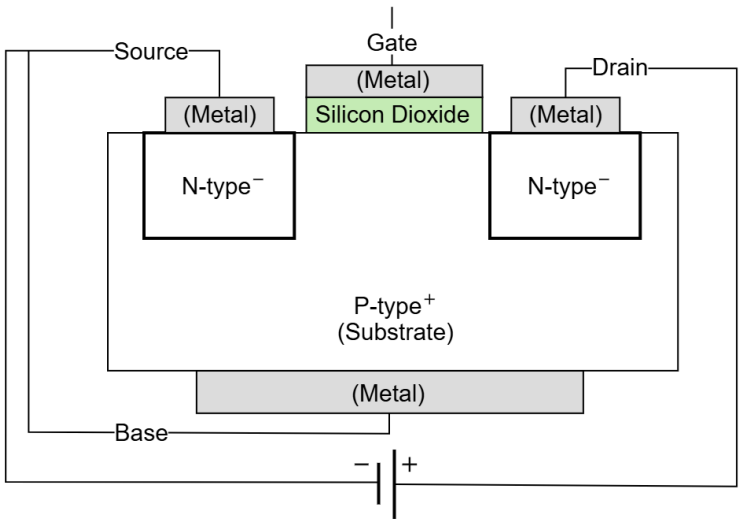


Figure 1.7: N-Channel Enhancement (off). Negative battery side to source, positive to drain.

Theorem 0.2: Flow of Electrons

Recall that a body of electrons that are negatively charged (low potential), have a surplus of electrons. A positive charge (high potential) reflects a deficit of electrons. Therefore, when given the chance (alike water), electrons will flow to fill the void.

Tip: An empty stomach has a high potential for food, while a full stomach has a low potential, as there's not much more room left to stuff food into.

Definition 0.10: MOSFET – N-Channel Battery Configuration

One battery is used to power the MOSFET, and another to control the gate. The source connects to the negative, and the drain to the positive side of the battery.

The gate is connected to a second battery, which can be either positive or negative, depending on the MOSFET type. The **other** end of this battery is connected to the source terminal.

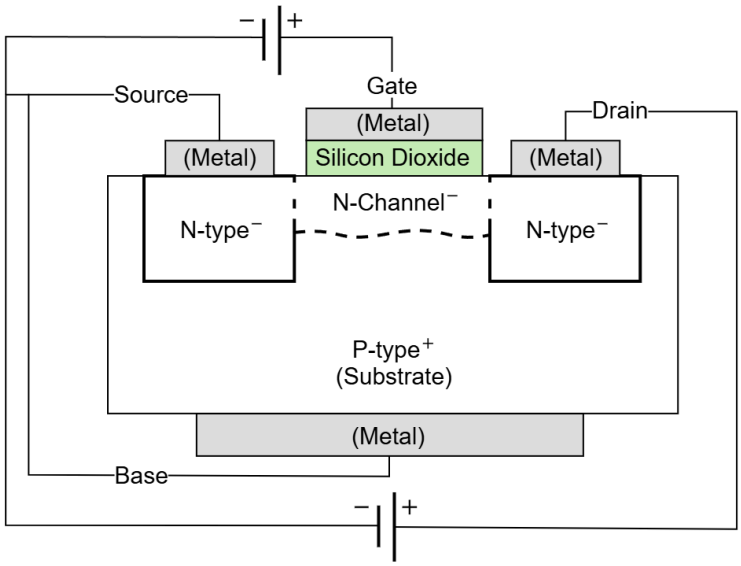


Figure 1.8: N-Channel Enhancement (on). Negative battery side to gate, positive to source. Positive charge given to the gate attracts negative electrons on the other side of the SiO_2 layer, creating a channel between the source and drain.

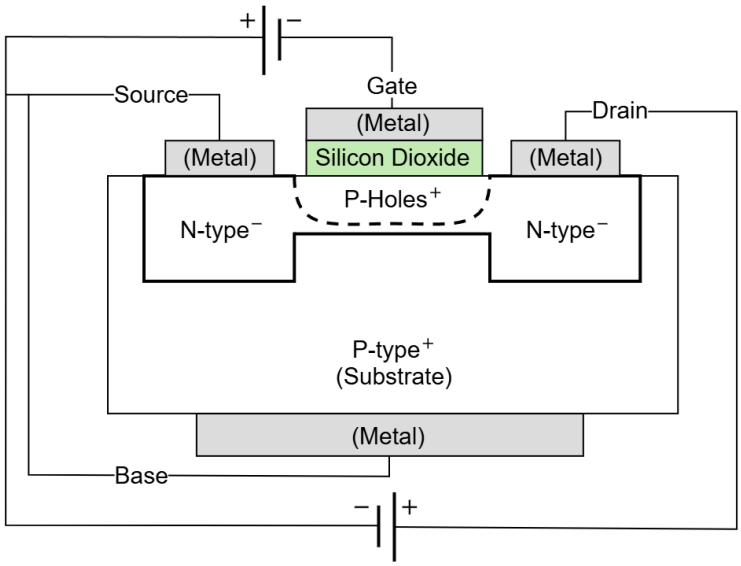


Figure 1.9: N-Channel Depletion (on). Negative charge to gate, creates holes into the channel.

Definition 0.11: MOSFET – Anatomy of P-Channel

The P-Channel variation follows the same logic as the N-Channel Definition (0.9); Instead, we swap N-type for P-type materials, and vice versa. Then apply negative for Enhancement and positive for Depletion on the gate to open or close the channel respectively (0.8). **In particular**, source now connects to a high potential and drain to a low potential.

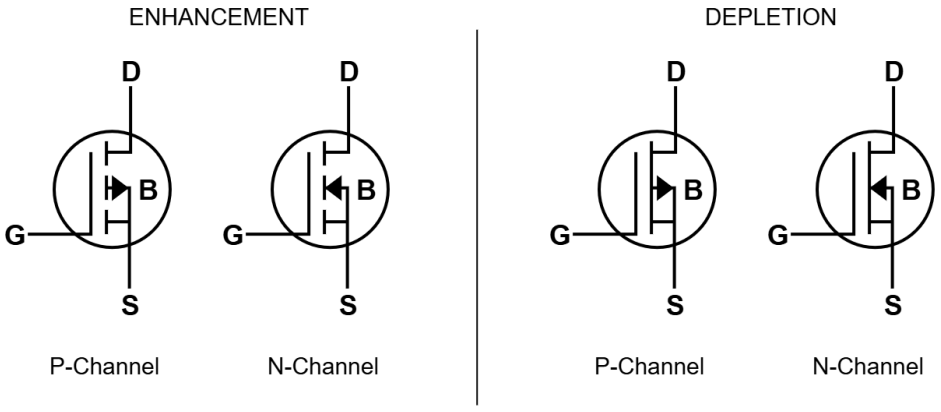


Figure 1.10: MOSFET symbols, **G** (gate), **S** (source), **D** (drain), and **B** (body) terminals.

1.0.2 Logic Gates & Functional Completeness

This section will cover how we take MOSFETs and use them to build logic gates.

Definition 0.12: Gate-Source Voltage V_{GS}

Recall Definition (0.10), the gate battery's opposite end is connected to the source terminal. This serves as a zero-volt reference for the gate terminal. The difference in potential between the gate and source terminals is called the **gate-source voltage** (V_{GS}):

$$V_{GS} = V_G - V_S,$$

Once V_{GS} exceeds the threshold voltage V_{TN} (for N-channel) or is below the threshold voltage V_{TP} (for P-channel), the MOSFET turns on, allowing current to flow from source to drain.

Tip: Notice that source takes from gate, i.e., for N-channel, the gate must overcome the source's negative charge. Hence we must exceed a threshold voltage to turn on the MOSFET. For P-channel, the same logic applies, but in reverse, as the components are implemented in a complementary manner.

Definition 0.13: Pull-Up & Pull-Down Switches

Let V_{DD} be the positive supply ("logical 1") and ground (0 V) be "logical 0." A CMOS logic gate uses:

- **Pull-Down Switch** (Off: 0, On: 1): N-channel enhancement. If $V_{GS} > V_{TN}$, source connects to drain, producing a logical 1.
- **Pull-Up Switch** (Off: 1, On: 0): P-channel enhancement. If $V_{GS} < V_{TP}$, source connects to V_{DD}, producing a logical 1.

Tip: Think of pull-down as "pulling down to the ground" to allow electrons to escape.

Additionally, the "DD" in V_{DD} does not stand for anything; it was made not to be confused with V_D, the voltage at the drain terminal. Though, unimaginative, it is simply convention.

Definition 0.14: CMOS Logic Gate

A **CMOS logic gate** is a circuit that uses **C**omplementary **MOS**FETs to perform logical operations. It consists of:

- **Pull-Down Network (PDN):** N-channel MOSFETs (NFET) connected to ground.
- **Pull-Up Network (PUN):** P-channel MOSFETs (PFET) connected to V_{DD}.

The output is high when the PUN is active and the PDN is inactive, and vice versa.

Pull-up	Pull-down	Digital
on	off	"1"
off	on	"0"
on	on	undefined
off	off	disconn.

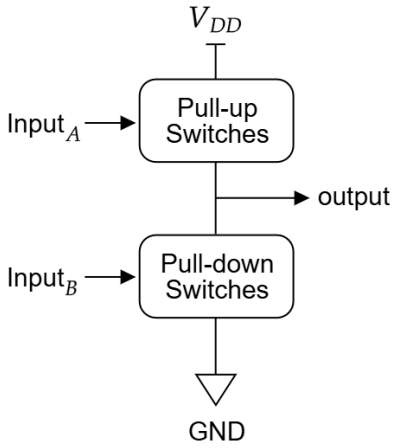
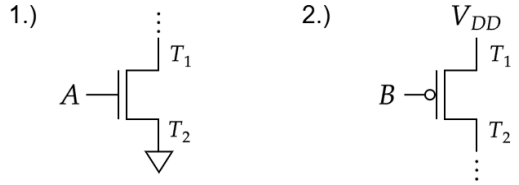


Figure 1.11: Simple CMOS logic gate, where **GND** stands for ground, V_{DD} for positive supply. **Note:** That even if the circuit is disconnected, the output may still *"remember"* its last state for some time until the charge dissipates.

Definition 0.15: Simplified NFET & PFET Symbols

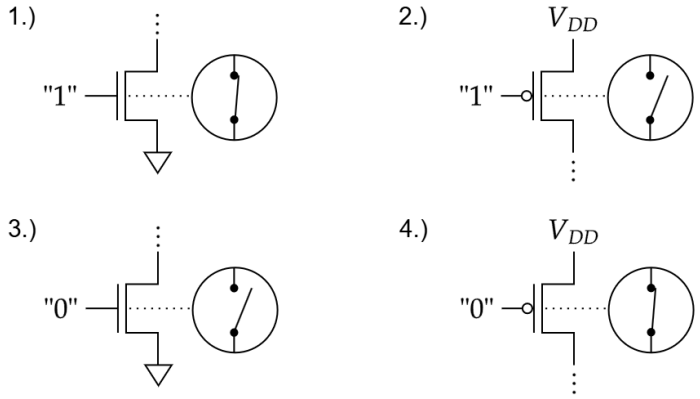
Below are input gates A and B, with two other terminals T_1 and T_2, simplifying Figure(1.11):



(1) NFET, T_1 output, and T_2 ground; (2) PFET, T_1 is V_{DD} (typically 1V), and T_2 output [3].

Definition 0.16: Open & Closed Circuits – NFET & PFET Logic

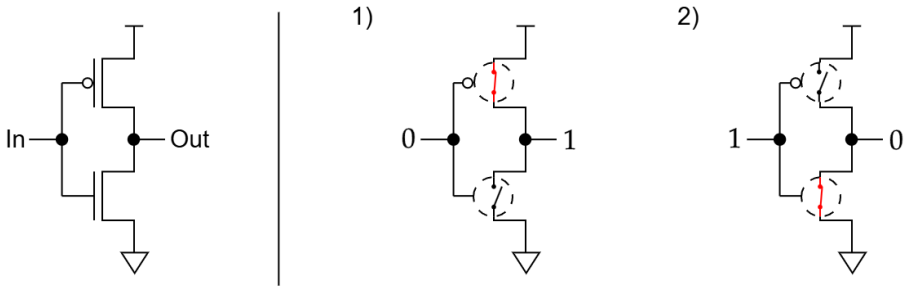
A **closed circuit** is a complete path for current to flow, while an **open circuit** is an incomplete path:



(1) An NFET is closed when given a digital 1 (high voltage), while (2) a PFET, is open; (3) NFET, open when given 0 (low voltage), while (4) a PFET is closed.

Definition 0.17: MOSFET Logic Gate – Not

Below shows a logic gate, NOT, using MOSFETs (NFET and PFET):



Both the NFET and PFET share the same input. The top line represents V_{DD} (positive supply), while the bottom line is ground (triangle). (1) input low, NFET is open, PFET is closed, output is high from V_{DD}; (2) input high, NFET is closed, PFET is open, output is low from ground. **Important:** A black dot connecting two or more lines represents a connection.

Definition 0.18: Functional Completeness

A function is **functionally complete** if it can expresses all possible Boolean functions. For example, the set of operators {AND, OR, NOT} is functionally complete. The NAND (NOT AND) or NOR (NOT OR) operators by themselves are functionally complete.

Example 0.1: Functionally Complete Sets of Operators

1. {NAND} alone

$$\begin{aligned} \neg A &= A \,\text{NAND}\, A, \\ A \wedge B &= \neg\big(A \,\text{NAND}\, B\big) = \big(A \,\text{NAND}\, B\big) \,\text{NAND}\, \big(A \,\text{NAND}\, B\big), \\ A \vee B &= (A \,\text{NAND}\, A) \,\text{NAND}\, (B \,\text{NAND}\, B). \end{aligned}$$

2. {NOR} alone

$$\begin{aligned} \neg A &= A \,\text{NOR}\, A, \\ A \vee B &= \neg\big(A \,\text{NOR}\, B\big) = \big(A \,\text{NOR}\, B\big) \,\text{NOR}\, \big(A \,\text{NOR}\, B\big), \\ A \wedge B &= (A \,\text{NOR}\, A) \,\text{NOR}\, (B \,\text{NOR}\, B). \end{aligned}$$

3. $\{\wedge, \neg\}$ alone

$$\begin{aligned} \neg A &= \neg A, \\ A \wedge B &= A \wedge B, \\ A \vee B &= \neg\big(\neg A \wedge \neg B\big). \end{aligned}$$

4. $\{\vee, \neg\}$ alone

$$\begin{aligned} \neg A &= \neg A, \\ A \vee B &= A \vee B, \\ A \wedge B &= \neg\big(\neg A \vee \neg B\big). \end{aligned}$$

5. $\{\rightarrow, \neg\}$ alone

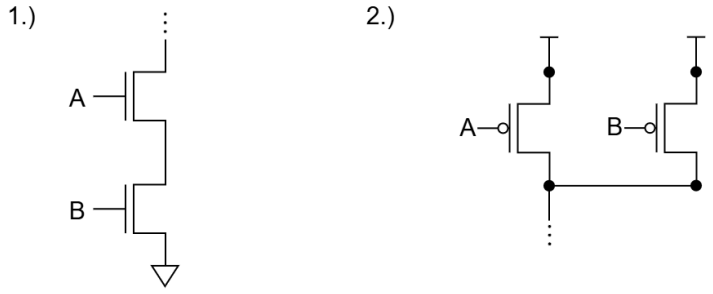
$$\begin{aligned} \neg A &= \neg A, \\ A \vee B &= \neg A \rightarrow B, \\ A \wedge B &= \neg\big(A \rightarrow \neg B\big). \end{aligned}$$

Recall, $A \rightarrow B$ is logically equivalent to $\neg A \vee B$. ■

Let's brainstorm possibilities for an AND and NAND, and elaborate on Defintition (0.14):

Theorem 0.3: Balancing Series & Parallel Connections

It's important in a CMOS circuit that the PUN and PDN are in fact complements of each other. Below illustrates two types of connections:



(1) Shows a NFET **series** connection (i.e., sequentially connected). Theoretically in isolation, this represents an AND gate ($A \cdot B$), meaning both A and B must be high to close the circuit.

(2) Shows a PFET in **parallel** connection (i.e., side-by-side connected). As per complementarity, this represents the NAND gate ($\overline{A} + \overline{B} = \overline{A \cdot B}$), by De Morgan's Law; Either A or B must be low to close the circuit.

Hence to **balance**, between the PUN and PDN, we must ensure that:

- each NFET series requires a PFET parallel.
- each PFET series requires a NFET parallel.

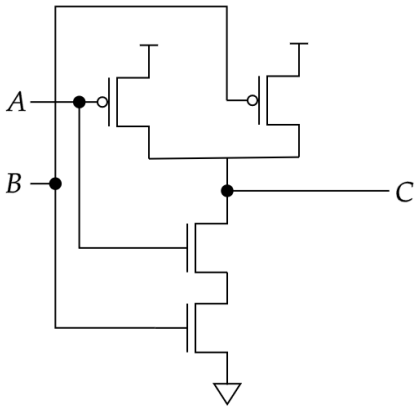
This keeps our networks complementary, ensuring that when one is closed, the other is open.

Tip: Think about how to create this before moving to the next page. understand that the placement of the output determines the logic of the circuit. Since PUNs default to high, think about how that might affect the output.

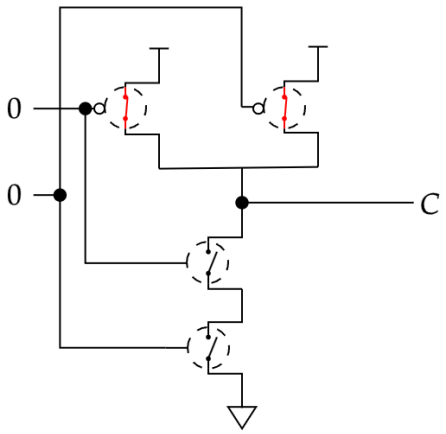
In CMOS, there must be both a PUN and PDN, think about how to balance the (2), from the above Theorem (0.3), to create a NAND gate.

Theorem 0.4: CMOS Logic Gate – NAND

Combining both networks in Theorem (0.3) yields a NAND gate in the following configuration:

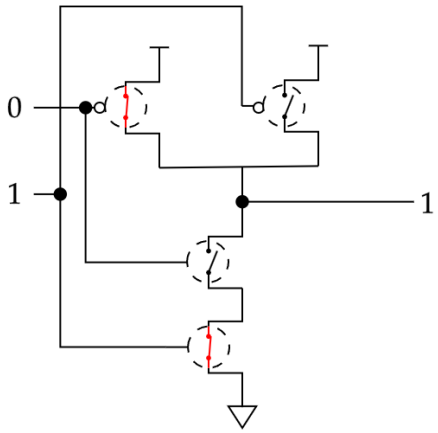


A	B	C
0	0	1
0	1	1
1	0	1
1	1	0



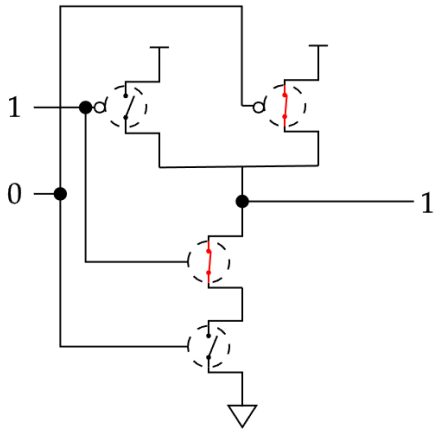
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Figure 1.12: Both inputs are (0,0), PUN closed, PDN open, hence the output is high (1).



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

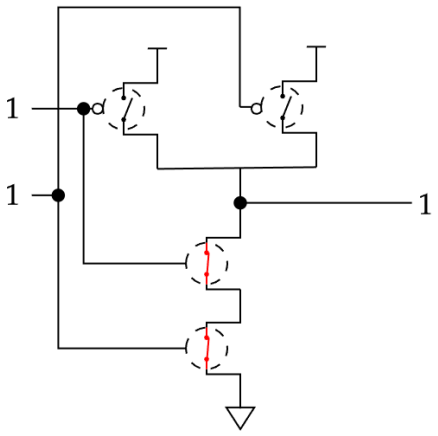
Figure 1.13: Both inputs are (0,1), PUN half-closed reaching the output, PDN half-closed, but can't reach the output; Hence the output is high (1).



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

Figure 1.14: Both inputs are (1,0), PUN half-closed reaching the output, PDN half-closed, but doesn't affect the output; Hence the output is high (1).

To continue with the last case:



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

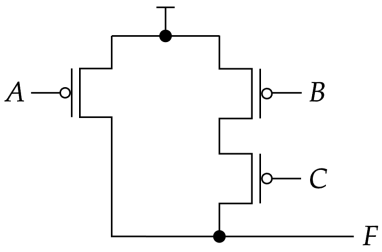
Figure 1.15: Both inputs are (1,1), PUN open, PDN closed reaching the output; Hence the output is low (0).

Theorem 0.5: Modeling Functions via PUNs

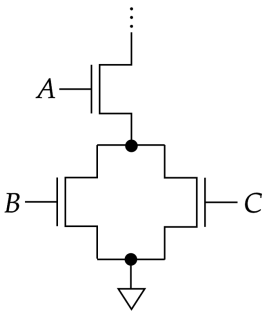
FETs in Series yields an AND, and in parallel yields an OR. Below we have the function $F = \overline{A} + \overline{B} \cdot \overline{C}$: (1) shows the PUN network, and then we design a complementary PDN network in (2) to balance the circuit:

$$F = \overline{A} + \overline{B} \cdot \overline{C}$$

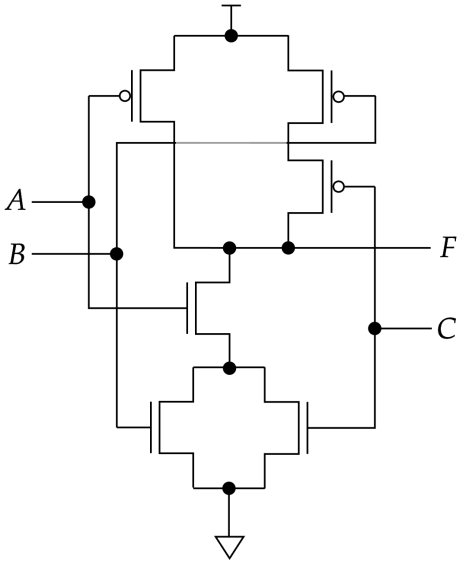
1.)



2.)



Note: Since PUNs are by default high, they naturally invert logic, requiring us to invert again or adjust logic for other functions which do not require inversion.



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

Figure 1.16: The complete CMOS logic gate for $F = \overline{A} + \overline{B} \cdot \overline{C}$. Here, the PUN is shown on top, and the PDN on the bottom.

1.0.3 CMOS Timing Specifications:

We touched briefly on timing specifications in Def (**??**). We fullen the picture with the following definition:

Definition 0.19: Contamination Delay (t_{CD}) & Propagation Delay (t_{PD})

In CMOS circuits, there are two main timing specifications to consider:

- **Propagation Delay** (t_{PD}): The maximum time for an input to stabilize at the output.
- **Contamination Delay** (t_{CD}): The minimum time delay an input remains unreflected at the output. I.e., how long the old value lingers before change is detected at the output.

Manufacturers often call Contamination delay the **Minimum Propagation Delay**. These specifications help manufacturers to verify signal coherence when designing circuits.

Let's take a look at an example on the next page.

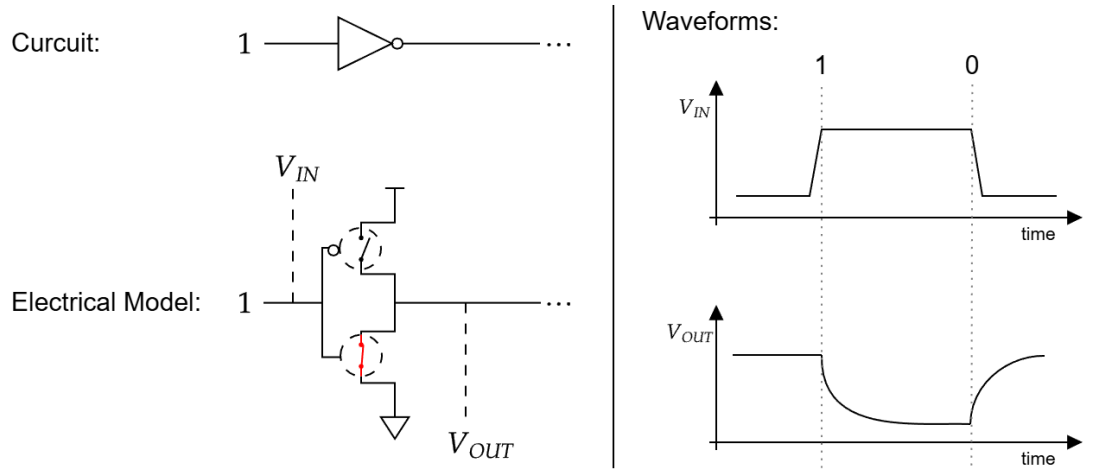


Figure 1.17: Observe the following inverter circuit receiving a high voltage (1) at the top-left of the diagram. Below strips away the symbol, showing the CMOS implementation; Here, the PUN is open, while the PDN is closed, relaying a low voltage (0) to the output. Now observe the waveforms on the right; Here, the y-axis represents voltage-level, and the x-axis represents time. Assuming we began with no voltage applied (0) at the input, when we apply a high voltage (1), it takes short time for V_{IN} to reach a readable high voltage (1). At the same time, the output V_{OUT} begins to drop from high (1) to low (0). Vice-versa when we take away power again from the input, shown as the 0 mark.

Now to calculate an overall t_{PD} and t_{CD} for a circuit:

Definition 0.20: Total Propagation & Contamination Delay

For a circuit with multiple logic gates, the overall timing specifications are calculated as follows:

- **Overall Propagation Delay** (t_{PD}): The sum of the individual propagation delays of each gate along the longest path from input to output.
- **Overall Contamination Delay** (t_{CD}): The sum of the individual contamination delays of each gate along the shortest path from input to output.

To clarify: Longest/shortest path refers to time spent, not the number of gates.

Note: In the next Diagram (1.18) t_{PD} and t_{CD} are illustrated apart from each other. However, this is just for clarity; In reality, t_{CD} is a subset of t_{PD}, both starting at the same time. I.e., $t_{CD} \leq t_{PD}$.

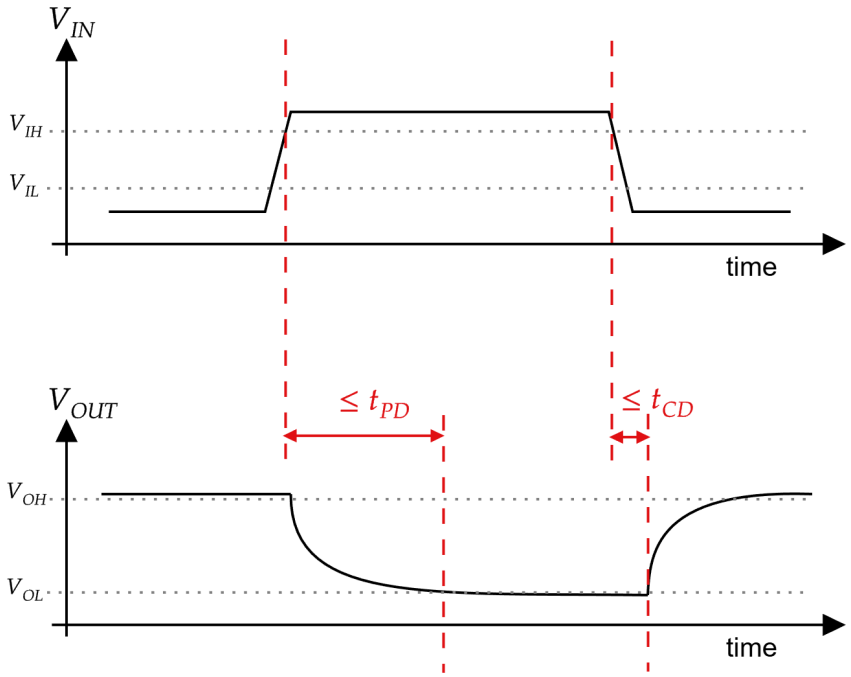


Figure 1.18: Zooming into the transition period, we can observe the timing specifications. The **Propagation Delay** (t_{PD}) is measured from when V_{IN} reaches a readable high voltage (1) to when V_{OUT} reaches a readable low voltage (0). The **Contamination Delay** (t_{CD}) is measured from when V_{IN} begins to rise from low (0) to when V_{OUT} begins to drop from high (1). Here we include both V_{IH}, V_{IL}, V_{OH}, and V_{OL} as described in Definition (**??**).

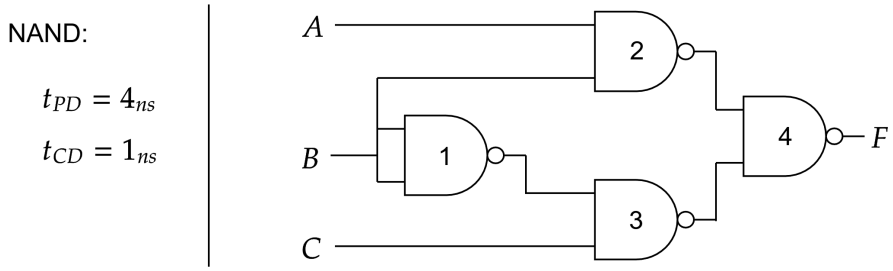


Figure 1.19: This circuit only consists of NAND gates, all of which have $t_{PD} = 4ns$ and $t_{CD} = 1ns$. The longest path from input to output is the ordered set $1, 3, 4$, hence the overall propagation delay is $t_{PD} = 4ns + 4ns + 4ns = 12ns$. The shortest path from input to output is the ordered set $2, 4$, hence the overall contamination delay is $t_{CD} = 1ns + 1ns = 2ns$.

1.0.4 Glitches, Hazards & Lenient Combinational Devices

These contamination periods introduce a new problem:

Definition 0.21: Glitches & Hazards

A **glitch** is a brief, unwanted change in a signal's state, often caused by differences in propagation delays within a circuit. A **hazard** is a condition that can lead to glitches, typically occurring when multiple signal paths with different delays converge at a single point. Hazards can be classified into:

- **Static Hazard:** Occurs when the output is supposed to remain constant (0 or 1) but temporarily changes due to differing path delays.
- **Dynamic Hazard:** Occurs when the output is expected to change from one state to another (0 to 1 or vice versa) but oscillates multiple times before settling. [5]

We can fix this with the following strategy:

Definition 0.22: Lenient Combinational Devices

A **lenient combinational device** is a circuit designed to safeguard against glitches and hazards. This is often achieved through redundancy or specific design techniques.

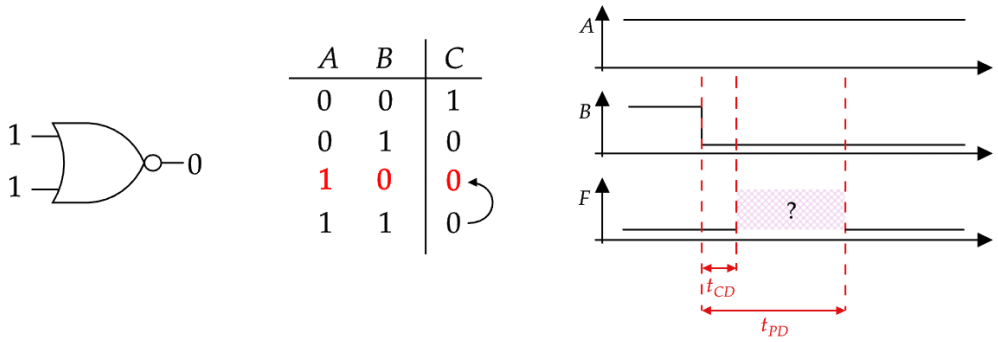
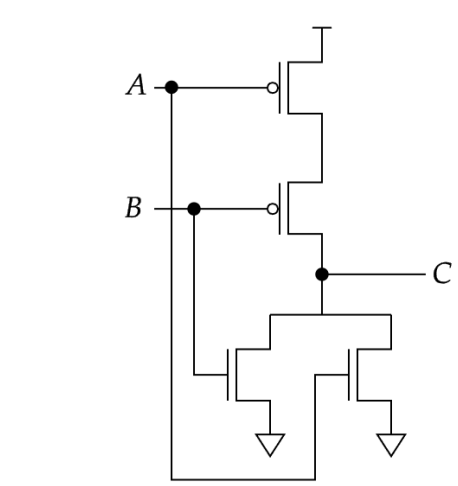


Figure 1.20: An example of a static hazard. Here, the output is supposed to remain low (0) when inputs (A, B) transition from (1,1) to (1,0). However, the time after t_{CD} before the end of t_{PD}, the output is undefined, potentially fluctuating between high (1) and low (0).

Next we'll show an example of a lenient combinational device that avoids hazards.

To show leniency, consider the following lenient NOR gate:



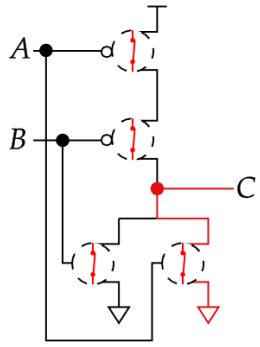
1)

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

2)

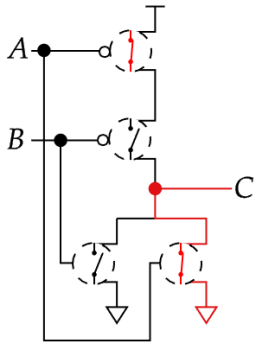
A	B	C
0	0	1
X	1	0
1	X	0

3)



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

4)

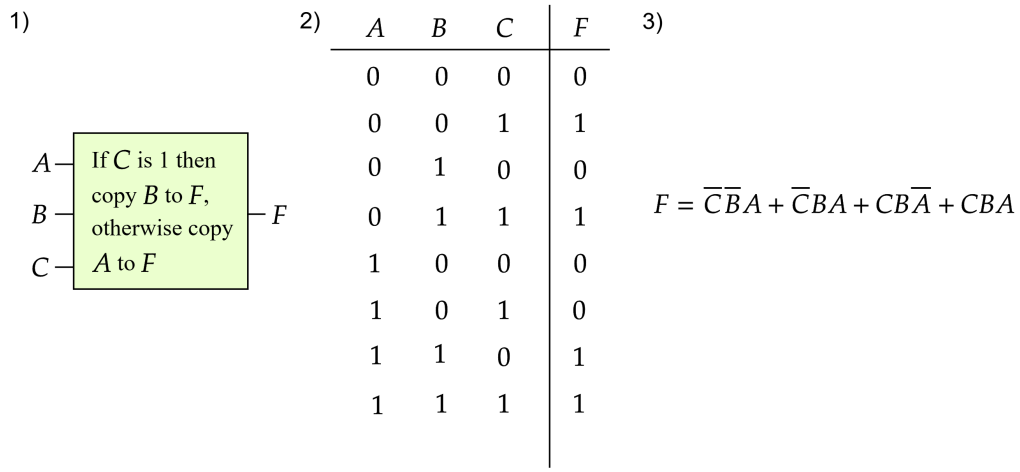


A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Figure 1.21: The following is a lenient NOR gate. 1) Shows the standard truth table for NOR. 2) Shows the lenient NOR truth table, where X represents a don't-care condition. I.e., when B is 1, no matter what A is, the output is 0. Vice-versa when A is 1. 3) Shows the the circuit when both A and B are 1. We see when transitioning to 4) where B flips to 0, A's complement retains its ground connection on the output; Hence the don't-care condition on B.

1.0.5 Combinational Logic

Truth tables quickly grow as n inputs lead to 2^n rows in the truth table. So instead, we model functions using boolean algebra (e.g., $F = A \cdot B + \overline{C}$), and then implement them using logic gates.

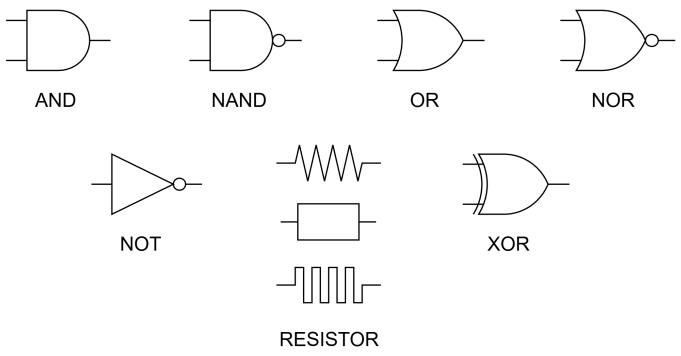


A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 1.22: 1) Shows the desired circuit function, 2) the corresponding truth table, and 3) the boolean logic. From the truth table, we can pick a row, substituting the inputs A, B, and C into the boolean expression, we expect to see the output corresponding to F. For example, row 3, $F = \overline{C}\,\overline{B}A + \overline{C}BA + CB\overline{A} + CBA$, is, $0 = (\overline{0} \cdot 1 \cdot 0) + (\overline{0} \cdot 1 \cdot 0) + (0 \cdot 1 \cdot 0) + (0 \cdot 1 \cdot 0)$.

Definition 0.23: Logic Gates

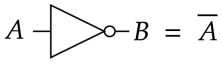
Below are the common logic gates with the addition of the resistor which limits current:



The resistor has many symbols; However, we focus purely on logic gates in this text, and how we can make more complex logical systems from these basic building blocks.

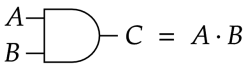
For example, lets consider just NOT, AND, and OR gates and their truth tables:

1) NOT



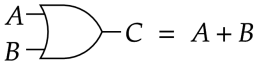
A	B
0	1
1	0

2) AND



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

3) OR



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

Figure 1.23: 1) NOT gate 2) AND gate 3) OR gate. Recall that inside these gates is a combination of PUNs and PDNs to achieve the desired logic, which here we have abstracted away.

$$F = \overline{C}\,\overline{B}A + \overline{C}BA + CB\overline{A} + CBA$$

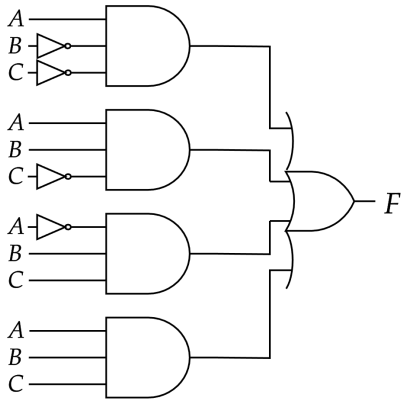
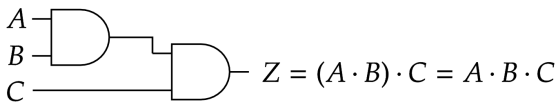


Figure 1.24: Modeling $F = \overline{C}\,\overline{B}A + \overline{C}BA + CB\overline{A} + CBA$ with logic gates. Note, we see that the ANDS and ORS have more than 2 inputs; They operate the same way as their 2-input counterparts, just with more inputs.

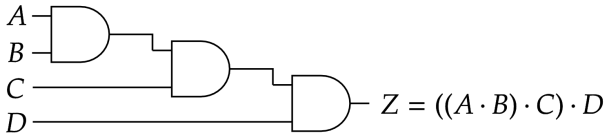
Theorem 0.6: Increasing Gate Input Size

Above in Figure 1.24, we see AND and OR gates with more than 2 inputs. There are multiple ways to implement these larger gates, observe below:

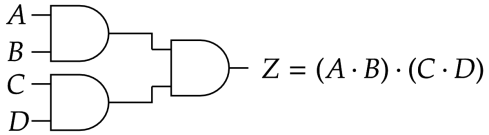
1)



2)



3)



1) is a 3 input gate, and 2) is a 4 input gate; This method of attaching the gates in sequential order is called **chaining**. 3) takes a different approach and instead uses a **tree method**.

In terms of t_{PD}, assuming all gates have the same delay, **chaining grows linearly** with the number of inputs, and the **tree method grows logarithmically** with the number of inputs.

However, if the gates were to have **different delays**, introduces **bottlenecks**; In the case of the tree method, getting an input like D introduces a longer path, while in the sequential method, D has less impact on the overall delay.

Theorem 0.7: NAND AND NOR - Gate Increase Problem & Efficiency

NAND and NOR are **not associative** like AND and OR. This means that we cannot use the chaining or tree method to increase the number of inputs for NAND and NOR gates.

Additionally, NAND and NOR gates are more efficient in CMOS logic, as CMOS is naturally inverting with PUNs and PDNs. Thus, their implementations are simpler than AND and OR gates.

To further illustrate that NAND and NOR are functionally complete, observe the below diagram:

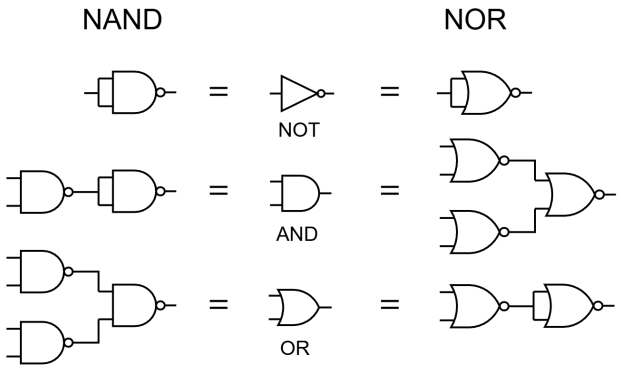
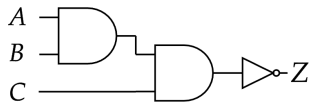


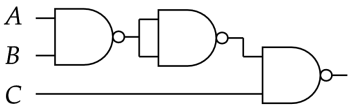
Figure 1.25: This diagram shows that NOT, AND, and OR gates can be constructed purely from NAND (left) or NOR (right) gates.

Theorem 0.8: Increasing NAND Gate Input Size with 2-input NANDs

Start with the desired function, e.g., $\overline{A \cdot B \cdot C}$, for clarity $\texttt{NAND}(A, B, C)$. We attempt to group inputs into pairs of 2: $\texttt{NAND}(A, B, C) = \texttt{NOT}(\texttt{AND}(A, B, C))$. Now we group: $\texttt{NOT}(\texttt{AND}(A, B, C)) = \texttt{NOT}(\texttt{AND}(\texttt{AND}(A, B), C))$, each AND has 2 inputs:



To only use NAND gates, we bring the NOT and AND back together: $\texttt{NAND}(\texttt{AND}(A, B), C)$. To replace A and B's AND gate, we substitute with NANDs using Figure (1.25):



Note: As we've mentioned before, chaining vs. tree methods vary differently on t_{PD}. Additionally notice that the size of our circuit can differ based on how we decide to break down our boolean expression. In the above Theorem (0.8), if we assume AND gates are slower than NAND gates in this system, the simplified NAND version is faster, and smaller, assuming AND gates are composed of multiple NANDs.

1.0.6 Karnaugh Maps

To help us find simpler boolean expressions, we can change how we represent our truth table data:

Definition 0.24: Karnaugh Maps

A Karnaugh map (K-map) represents truth table data with a format called **Gray Code**. In Gray Code, columns and rows differ by only one bit of information:

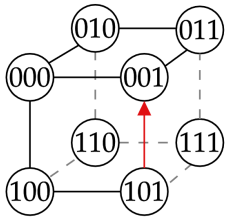
1)

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

2)

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	0	1	1	0

3)



1) Shows the truth table for inputs A, B, and C with output F. 2) The corresponding K-map (left-most column, C, top row, AB respectively). **Note:** in 2) columns and rows are cyclical, meaning the leftmost and rightmost columns are adjacent, as are the top and bottom rows. Hence with 3), we can represent our truth table as a 3D cube (Formatted ABC). The red arrow in both 2) and 3) indicate adjacency. We extend this to 4 inputs:

$CD \setminus AB$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

For inputs 5 and 6, we need an additional dimension, building a $4 \times 4 \times 4$ k-map. Anything beyond 6 inputs becomes difficult to visualize, so algorithms become more integral in the simplification process.

Now to actually use this k-map to simplify we introduce the following method:

Definition 0.25: Simplifying with Implicants & Prime Implicants

An **implicant** is a set of adjacent 1s in a k-map, whose width and length are a power of 2 (i.e., 1, 2, 4, 8, ...). An implicant that is not a proper subset of another implicant (i.e., not contained within another implicant) is called a **prime implicant**.

Name of the game: Find the smallest set of prime implicants that cover all 1s in the k-map. Then translate each member of such set into a product term (e.g., top row $AB := 01 \to \overline{A}B$). Then sum all found product terms to yield the simplified boolean expression:

1)

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	1	0	0	0

2)

$C \setminus AB$	00	01	11	10
0	1	0	0	1
1	1	1	0	1

1) The red singleton implicant translates to $\overline{A}\overline{B}C$. The blue pair translates to $A\overline{C}$, yielding the simplified expression $F = \overline{A}\overline{B}C + A\overline{C}$. **Note:** In the second term we dropped B, as despite its value switching, it did not affect the output.

2) Red group: $\overline{A}C$, blue group: $\overline{B}$, yielding $F = \overline{A}C + \overline{B}$, which demonstrates that finding **larger groups** lead to smaller terms, and thus, a simpler expression.

Theorem 0.9: Prime Implicants & Uniqueness

Though we look for the smallest set of prime implicants, there may be multiple such sets (i.e., same size, different members) that cover all 1s in the k-map.

Tip: Think of our 1 sets as capturing the truth table in its truth states (i.e,. when A is this, and B is this, the output is true!). We combine all such states to reflect the overall behavior (i.e., if this state or this state is true, the output is true!).

Recall from discrete math, this is called the disjunctive normal form (DNF) of a boolean function. We can do the same with 0s, called the conjunctive normal form (CNF) of a boolean function, where we specify which states we don't want; Though we won't explore CNF here.

Let us explore an example with 4 inputs:

1)

$CD \setminus AB$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

2)

$CD \setminus AB$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

Figure 1.26: 1) and 2) A 4-input k-map with 4 prime implicants highlighted. 1) Yields $F = \overline{B}C + D + B\overline{C} + \overline{C}A$. 2) Yields $F = \overline{B}C + D + B\overline{C} + A\overline{B}$.

Theorem 0.10: Glitches from Implicant Hopping

Moving between two prime implicants (product terms) which do not overlap causes glitches; Because each is solely driving the output. So switching one off one and the other on, briefly exhibits undefined behavior, due to the t_{PD} (uncertainty period).

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	0	1	1	0

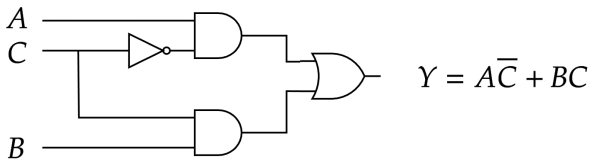


Figure 1.27: To the left shows the k-map of a function consisting of two prime implicants (circled red). The right shows the logic gate implementation and its resulting sum of products expression.

We deviate from our game of smallest sets, adding redundancy:

Theorem 0.11: Resolving Implicant Glitches with Redundancy

To resolve glitches caused by gaps between prime implicants, we can add another prime implicant that fills that gap to help hold the output steady during transitions.

To further illustrate the glitch we visualize the voltage over time:

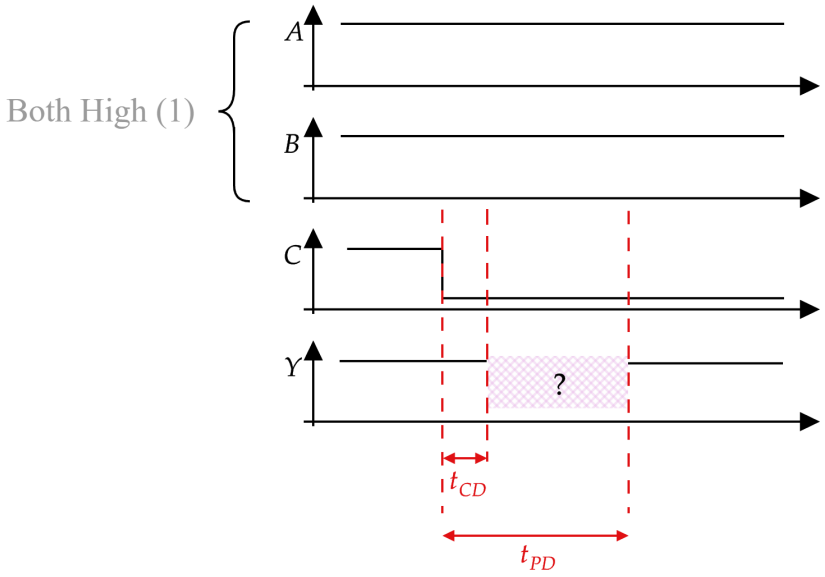
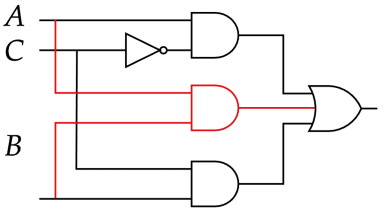


Figure 1.28: A and B inputs remain steady high (1), while C transitions from high (1) to low (0). Based on Figure 1.27, the output Y should remain high (1); However, due to both products relying on C, the output is briefly undefined for t_{PD}.

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	0	1	1	0



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

Figure 1.29: By adding the redundant prime implicant (in red), we ensure steady output during transitions. In particular, the output isn't solely reliant on C during its transition, thus avoiding the glitch.

1.0.7 Multiplexers & Read-only Memory

Below observe the table we've used in our k-map Definition (0.24):

1)

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2)

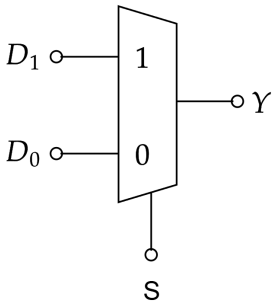


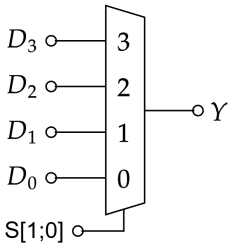
Figure 1.30: 1) A three input truth table for when S is 0 D_0 is the output, and when S is 1, D_1 is the output. 2) A diagram simplification of the truth table: S has two controls 1 and 0, which chooses between data inputs D_1 and D_0 respectively.

This new combination device we've created has a name:

Definition 0.26: Multiplexer

A **multiplexer** (MUX) is a combinational logic device that selects one of many data inputs based on control inputs, outputting the selected data input. In Figure (1.30), S the control input and D_0, D_1 the data inputs.

More generally, a MUX with n control inputs can select from 2^n data inputs. Vice-versa a MUX with m data inputs, has $\log_2(m)$ control inputs.



E.g., a 4-to-1 MUX: 2 control inputs S_1 and S_0 selecting from 4 data inputs.

In the above Definition (0.26), we simply showed a 4-1 MUX; However, they are much more complicated devices:

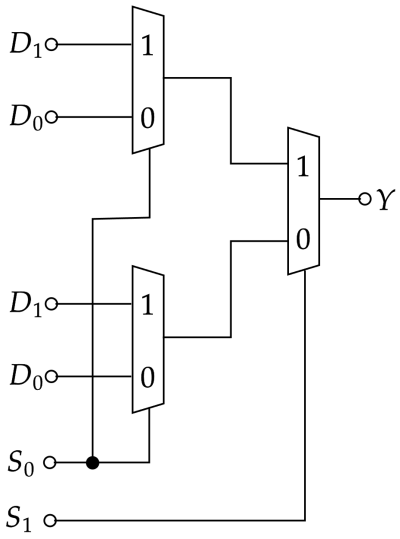


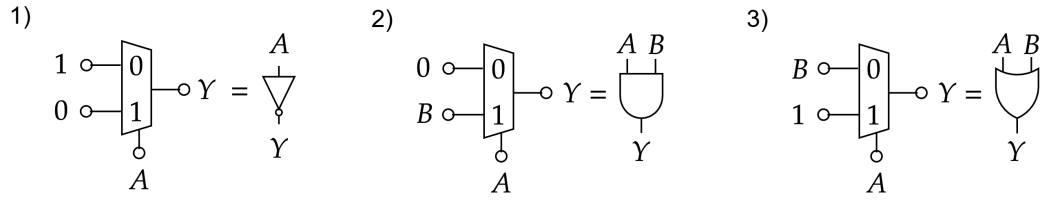
Figure 1.31: This shows that under the hood is composed of a tree of 2-to-1 MUXes.

Theorem 0.12: Building Larger MUXes from 2-to-1 MUXes

Any larger MUX can be built from a tree of 2-to-1 MUXes. In particular, an n-to-1 MUX requires $n-1$ of 2-to-1 MUXes to build. [8]

Theorem 0.13: MUX as Boolean Function Implementers

MUXes are functionally complete, i.e., can implement any boolean function.



E.g., 1) shows a NOT, 2) an AND, and 3) an OR gate implemented with a 2-to-1 MUX.

To be clear, Logic gates and MUXes may both be used to implement boolean functions, it's a matter of which use-case is more appropriate:

Theorem 0.14: MUX vs. Logic Gates

Logic gates help with basic/dedicated functions, such as needing a AND, OR, NOT gate, or a simple routine (algorithm). MUXes help with selecting between multiple data inputs, i.e., large if-else/branching logical statements.

Depending on the function implementation's t_{PD} and size requirements, one may be more appropriate than the other based on the schematics at hand.

Note: In real world applications, MUXes carry significant overhead compared to logic gates. They are complex devices for simple selection use cases, meaning they are not as flexible as logic gates. This is an important consideration when it comes to the design costs/efficiency of a circuit.

We discussed a device (MUX) that selects the row of a truth table where a specific piece of data lives. This is good for one output; But to allow for n outputs, we'll add another set of n lines (columns) to each row. Each column leads to an output:

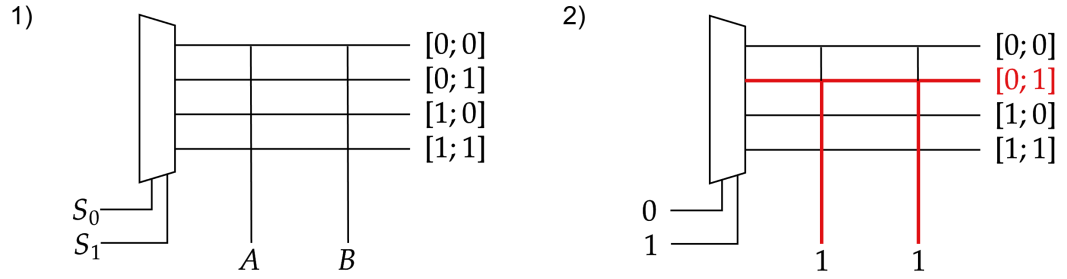


Figure 1.32: 1) Shows an altered 4-to-1 MUX from Definition (0.26) with the addition of 2 columns (outputs) A and B per row. 2) Shows $S_0 = 0$ and $S_1 = 1$, thus selecting row 2, outputting $A = 1$ and $B = 1$.

This new device also has a name:

Definition 0.27: Decoder

A **decoder** is a combinational logic device that takes n control inputs and decodes them to 2^n possible unique outputs (explained on the next page). E.g., Figure (1.32) shows a 2-to-4 decoder, 2 inputs leading to 4 outputs.

This differs from the MUX naming convention where n select inputs lead to 1 output (2^n-to-1 MUX) vs. (n-to-2^n Decoder).

In Figure (1.32), we see that we'll always get 1 from each input, which isn't very interesting;

Theorem 0.15: Decoder PDN-line Manipulation

To achieve 2^n output on a decoder, PDNs are attached to output-lines driving high (1) lines to low (0). Each row mix-and-matches PDNs to each intersection just how a truth table would with 0s and 1s for unique combinations.
Note: PDNs only effect their intersection, not the entire column.

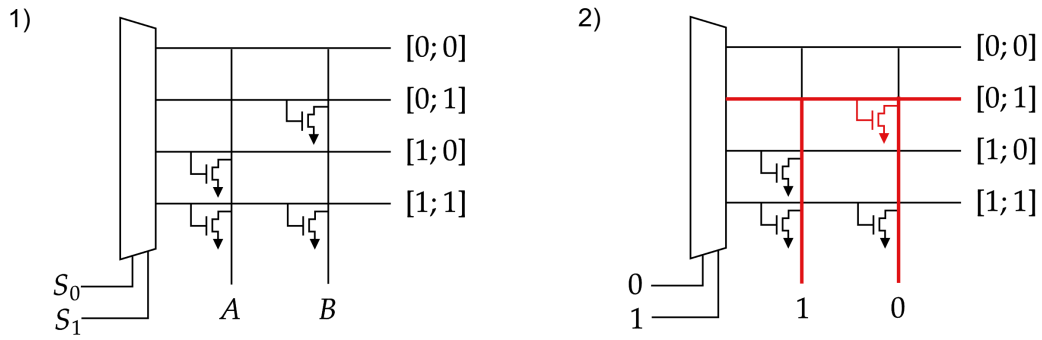


Figure 1.33: 1) Shows the 2-to-4 decoder from Figure (1.32) now with PDNs: $[0;0]$ has 0 PDNs, $[0;1]$ and $[1;0]$ have 1 PDN each, and $[1;1]$ has 2 PDNs. 2) Shows $S_0 = 0$ and $S_1 = 1$, thus selecting row 2, outputting $A = 1$ and $B = 0$ because of it's PDN. **Notice:** That A's line isn't effected by the other PDNs below, thus it's output remains high.

1.0.8 32-bit Full Adder (Addition)

We now know enough to create a combinational logic circuit that performs addition. Recall the information provided in Theorem (**??**) about binary addition. We now build intuition:

Theorem 0.16: 32-bit Full Adder Strategy (Logic Gates)

We approach the problem truth-table first. To add two binary numbers A and B, we need at least 2 inputs; However, we need to account for carry-in bits C_{in} from previous additions (3 inputs). We also need to pick between sum bit S and carry-out bit C_{out} (2 outputs).

Logically, we only have a carry if at least 2 of the 3 inputs are 1; Hence, $C_{out} = (A \wedge B) \vee (A \wedge C_{in}) \vee (B \wedge C_{in})$. For the sum bit, at most 1 input can be 1; Hence, $S = A \oplus B \oplus C_{in}$ (XOR).

This only adds 1 bit; To extend to 32-bits, we chain 32 of these full adders (FA) together: Break A and B into 32 individual bits, A_0 to A_{31} and B_0 to B_{31}. The carry-out of each FA becomes the carry-in of the next FA. We may use the last carry-out C_{out} as an overflow bit.

Let's implement the above strategy:

A	B	C_i	$C_o = (A \wedge B) \vee (A \wedge C_i) \vee (B \wedge C_i)$	$S = A \oplus B \oplus C$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 1.34: Full Adder (FA) truth table, A, B, and C_{in}, with outputs C_{out} and S.

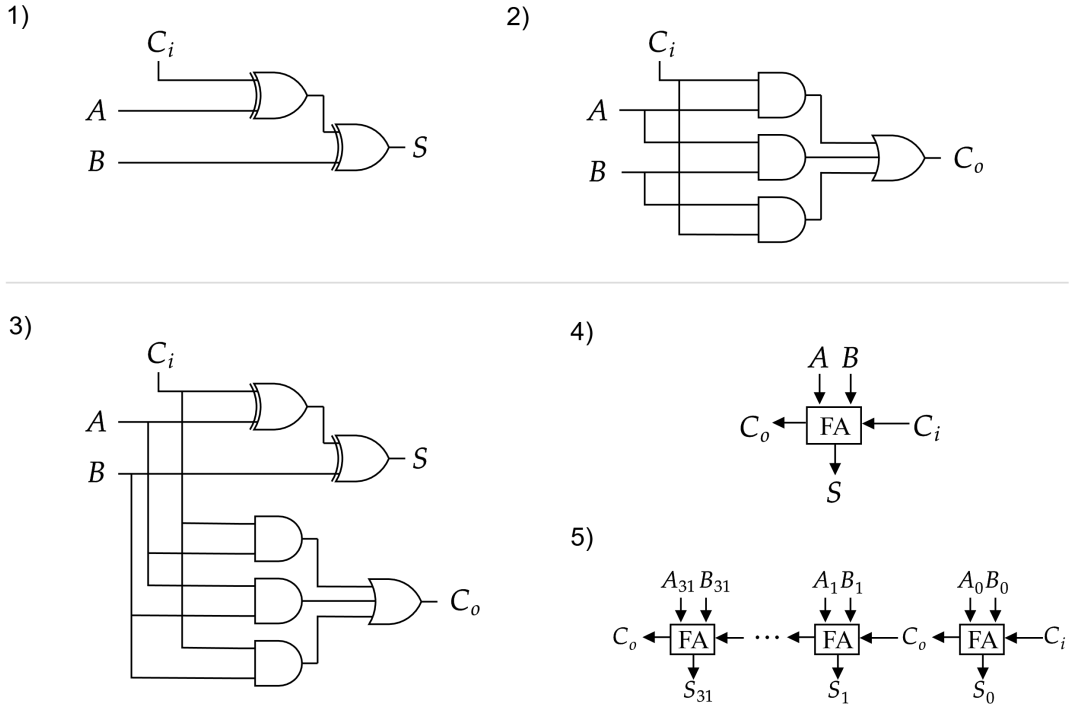


Figure 1.35: 1) S logic gates, 2) C_{out} logic gates, 3) combining S and C_{out} logic gates to form a full adder. 4) diagram representation of FA, 5) chaining 32 FAs to create a 32-bit adder.

Again, remember, we can always make a MUX/Decoder version of whatever logic gate version we create:

1)

A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

2)

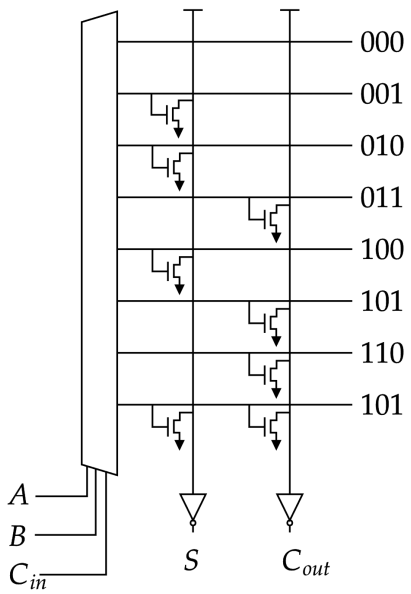


Figure 1.36: 1) Shows our desired logic (truth-table) 2) Shows a 3-to-8 Decoder, with select lines A, B, and C_{in}, driving 8 outputs, leading to outputs S and C_{out} via OR gates. **Note:** The inverters just before the outputs; This flips our PDNs low outputs to high for our desired logic.

Bibliography

[1] What is a transistor? YouTube video, https://youtu.be/AwXp6jVaTV4?si=s4-UwlgglmiCBPso, 2022.

[2] CrashCourse. The nucleus: Crash course chemistry #1. YouTube video, https://youtu.be/FSyAehMdpyI?si=h5ngW3IvcCTiOoL_, 2013. Published February 12, 2013.

[3] EngMicroLectures. Building logic gates from MOSFET transistors. https://youtu.be/1rZyGL1K5QI?si=ODgxP84kuHqVq6Bi, ?? 2013. [Online; accessed 2025-07-13].

[4] Getty Images. Getty Images Stock Photo 700832601. https://www.thoughtco.com/thmb/TTC7l9oab0l_A2_xRPryeoHSvXc=/2092x1433/filters:no_upscale():max_bytes(150000):strip_icc()/GettyImages-700832601-5bb602c0c9e77c002609fe08.jpg. Thumbnail image hosted on ThoughtCo via Getty Images. Accessed: 2025-07-07.

[5] Harvard University. Hazards and glitches. CSCI E-93: Computer Architecture, Lecture Slides, 2024. URL: https://cscie93.dce.harvard.edu/fall2024/slides/Hazards%20and%20Glitches.pdf [Accessed: 2025-12-13].

[6] Infinity Learn. Concept of valency - introduction | atoms and molecules. YouTube video, August 2018. Accessed: 6 July 2025.

[7] The Engineering Mindset. Mosfet explained - how mosfet works. YouTube video, https://youtu.be/AwRJsze_9m4?si=whrgrmMmzrychH2c, 2024.

[8] Quora. How do you design a 12-to-1 multiplexer from 2-to-1 multiplexers only? https://www.quora.com/How-do-you-design-a-12-to-1-multiplexer-from-2-to-1-multiplexers-only, 2025. Accessed: 2025-12-23.

[9] Wayne Breslyn (Dr. B.). Molecule vs compound: Examples and practice. YouTube video, 2013. Accessed: 6 July 2025.