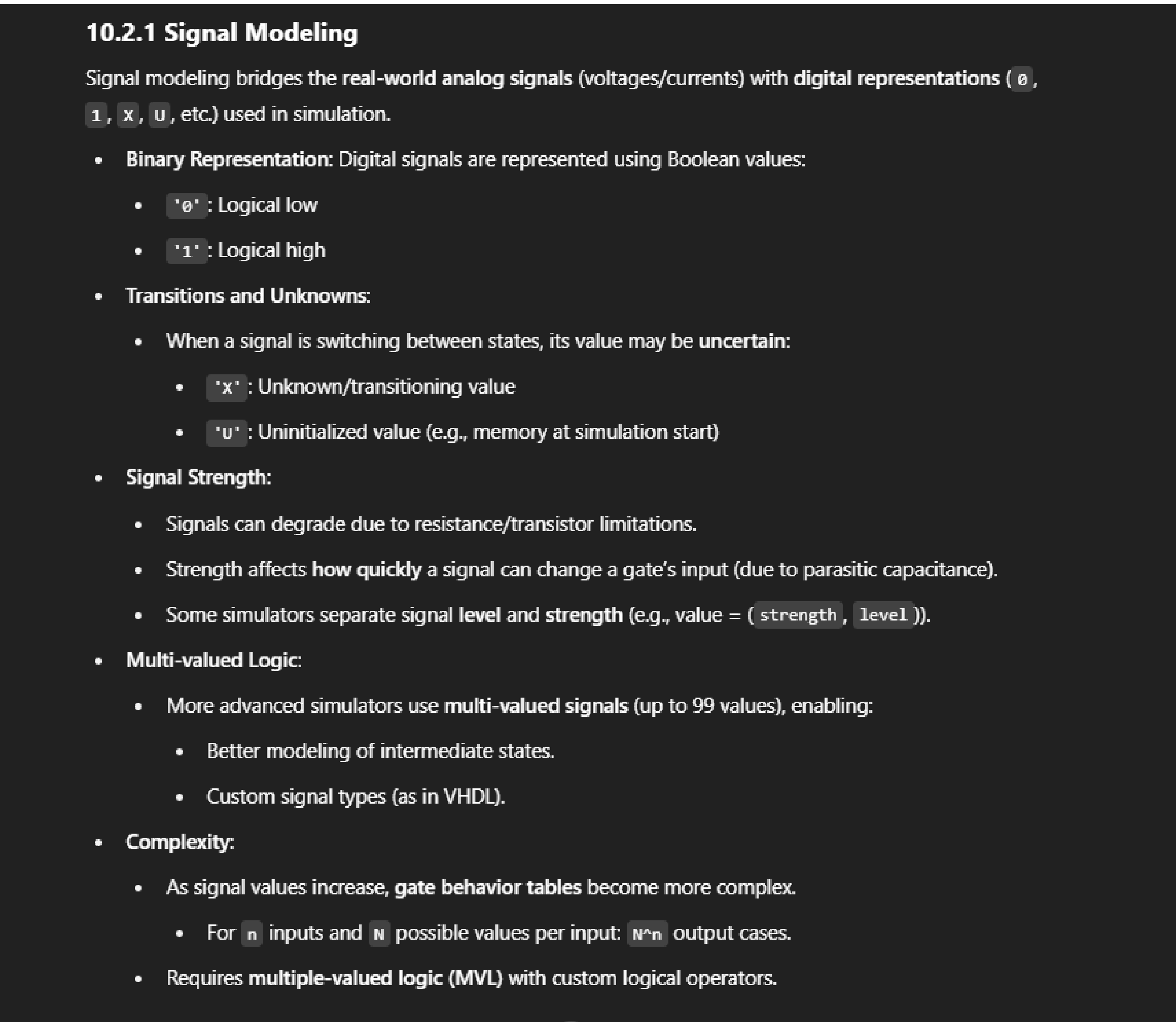
Gate-Level Modeling and Simulation

Gate-level simulation involves mimicking the behavior of digital logic gates (AND, OR, NAND, etc.) and their interconnections in a circuit. This simulation type models how binary signals propagate through gates over time, accounting for signal values, gate behavior, delay, and circuit connectivity.



10.2.1 Signal Modeling

Signal modeling bridges the **real-world analog signals** (voltages/currents) with **digital representations** (`0`, `1`, `X`, `U`, etc.) used in simulation.

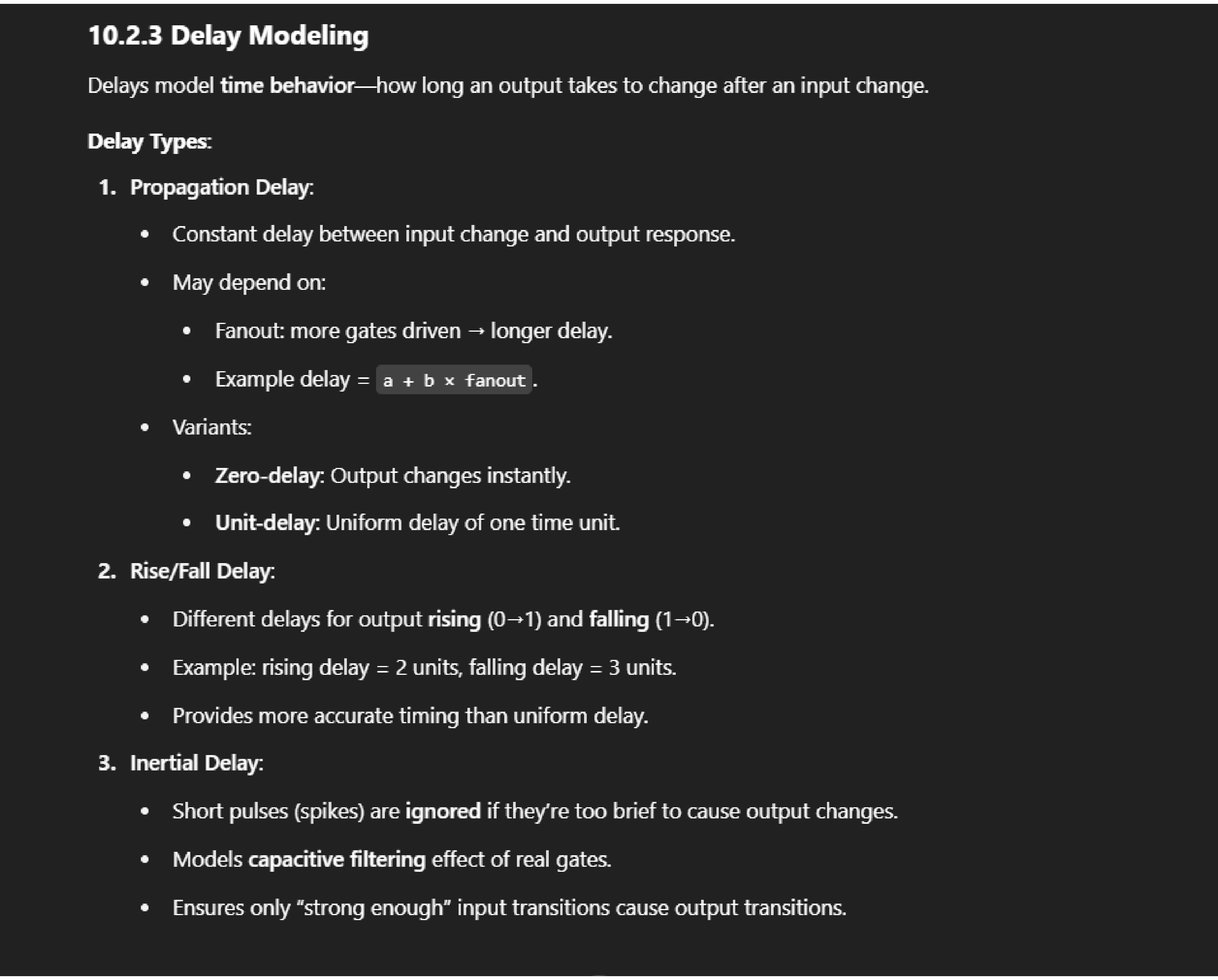
- **Binary Representation:** Digital signals are represented using Boolean values:
  - `'0'`: Logical low
  - `'1'`: Logical high
- **Transitions and Unknowns:**
  - When a signal is switching between states, its value may be **uncertain**:
    - `'X'`: Unknown/transitioning value
    - `'U'`: Uninitialized value (e.g., memory at simulation start)
- **Signal Strength:**
  - Signals can degrade due to resistance/transistor limitations.
  - Strength affects **how quickly** a signal can change a gate's input (due to parasitic capacitance).
  - Some simulators separate signal **level** and **strength** (e.g., value = (`strength`, `level`)).
- **Multi-valued Logic:**
  - More advanced simulators use **multi-valued signals** (up to 99 values), enabling:
    - Better modeling of intermediate states.
    - Custom signal types (as in VHDL).
- **Complexity:**
  - As signal values increase, **gate behavior tables** become more complex.
    - For `n` inputs and `N` possible values per input: `N^n` output cases.
  - Requires **multiple-valued logic (MVL)** with custom logical operators.

10.2.2 Gate Modeling

Modeling a gate means defining how output signals depend on input signals.

Techniques:

1. **Truth Table Representation**:
   - Explicit table of input → output combinations.
   - Example: Table 10.1 shows a **2-input NAND gate** truth table for inputs `0`, `1`, and `X`.
2. **Subroutine (or Logic Function) Representation**:
   - More efficient, especially with only binary values.
   - Uses **machine-level instructions** for logical operations.
   - Can be extended to support MVL with more complex mappings.



10.2.3 Delay Modeling

Delays model **time behavior**—how long an output takes to change after an input change.

Delay Types:

1. **Propagation Delay**:
   - Constant delay between input change and output response.
   - May depend on:
     - Fanout: more gates driven → longer delay.
     - Example delay = `a + b × fanout`.
   - Variants:
     - **Zero-delay**: Output changes instantly.
     - **Unit-delay**: Uniform delay of one time unit.
2. **Rise/Fall Delay**:
   - Different delays for output **rising** (0→1) and **falling** (1→0).
   - Example: rising delay = 2 units, falling delay = 3 units.
   - Provides more accurate timing than uniform delay.
3. **Inertial Delay**:
   - Short pulses (spikes) are **ignored** if they're too brief to cause output changes.
   - Models **capacitive filtering** effect of real gates.
   - Ensures only "strong enough" input transitions cause output transitions.

10.2.5 Compiler-Driven Simulation

Compiler-driven simulation is one of the two main simulation methods (other is **event-driven**).

Best Use Case:

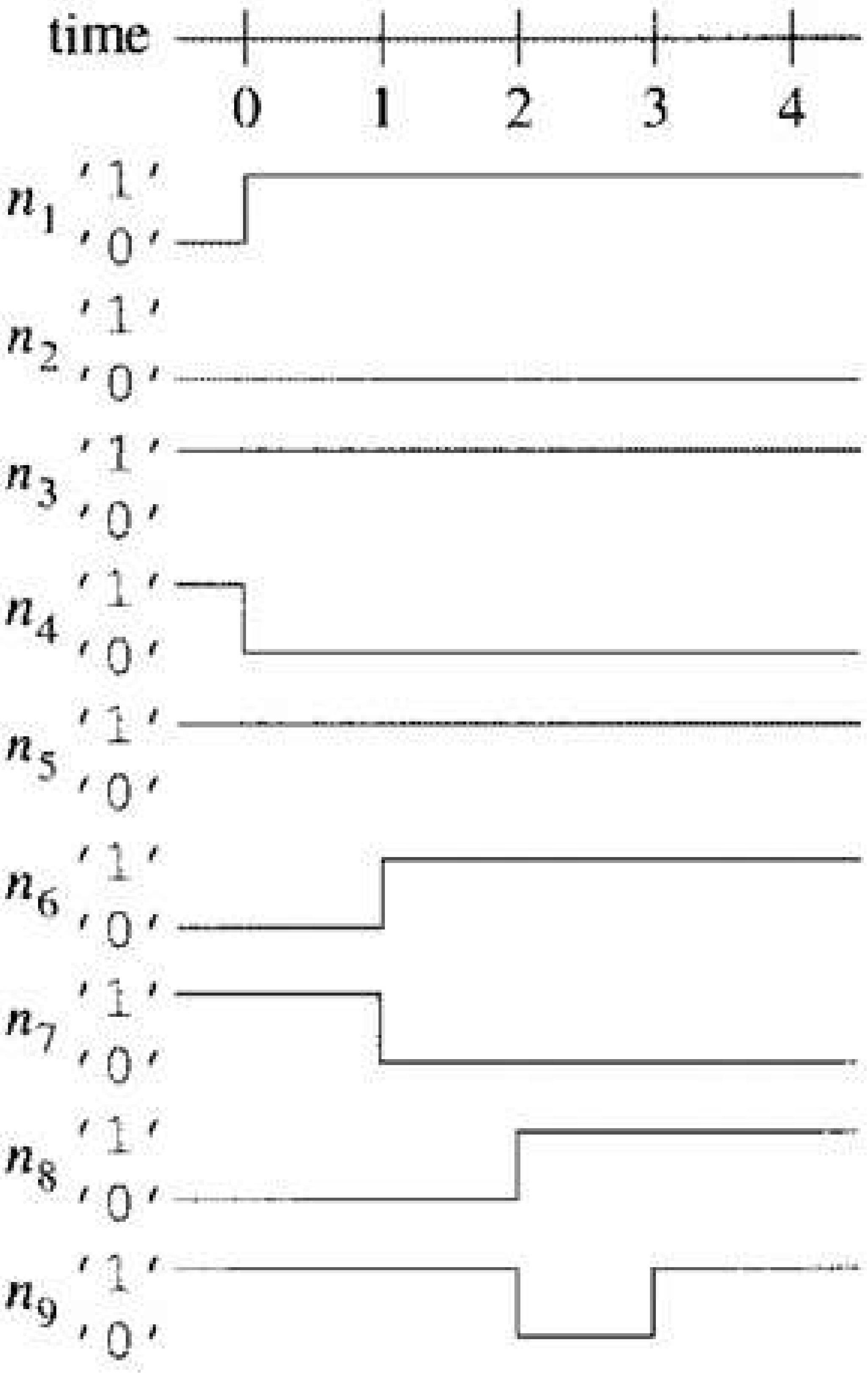
- **Synchronous circuits**, where simulation **ignores timing delays** (i.e., assumes **zero-delay**).

Steps in Compiler-Driven Simulation:

1. **Leveling:**
    - Assign levels to nets:
        - Inputs: level 0
        - Gates: level = max(input levels) + 1
    - Determines **execution order** for evaluating gates.
2. **Code Generation:**
    - Pseudo-code generated per net based on level order.
    - Logical operations translated into **inline machine code**.
    - Avoids function calls for speed.
3. **Simulation Execution:**
    - At each simulation step, input changes are applied.
    - Corresponding gate evaluations are executed.
    - Output is passed to the result display module.

Example:

- Simple 5-input combinational circuit (Figure 10.3) → translated into simulation code (Figure 10.4).
- Shows signal computation based on gate logic and dependencies.



time
0
1
2
3
4
n_1
n_2
n_3
n_4
n_5
n_6
n_7
n_8
n_9
'1'
'0'

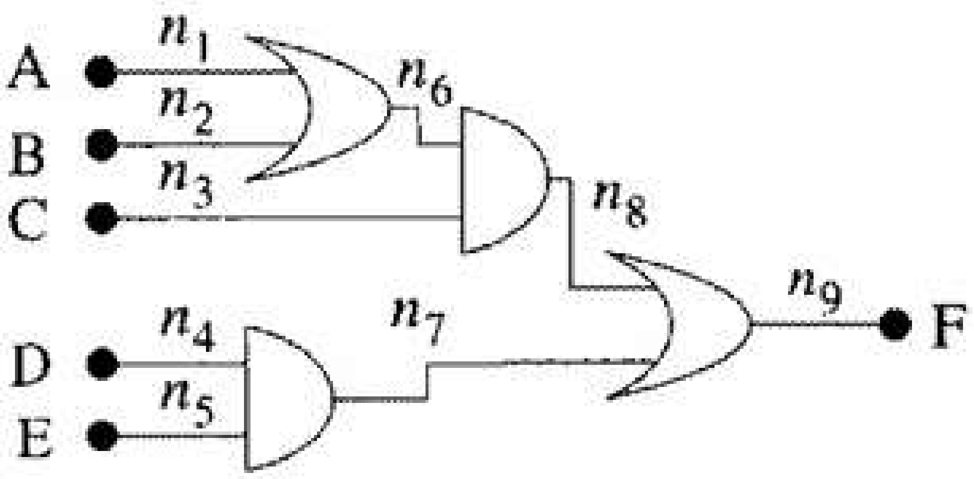
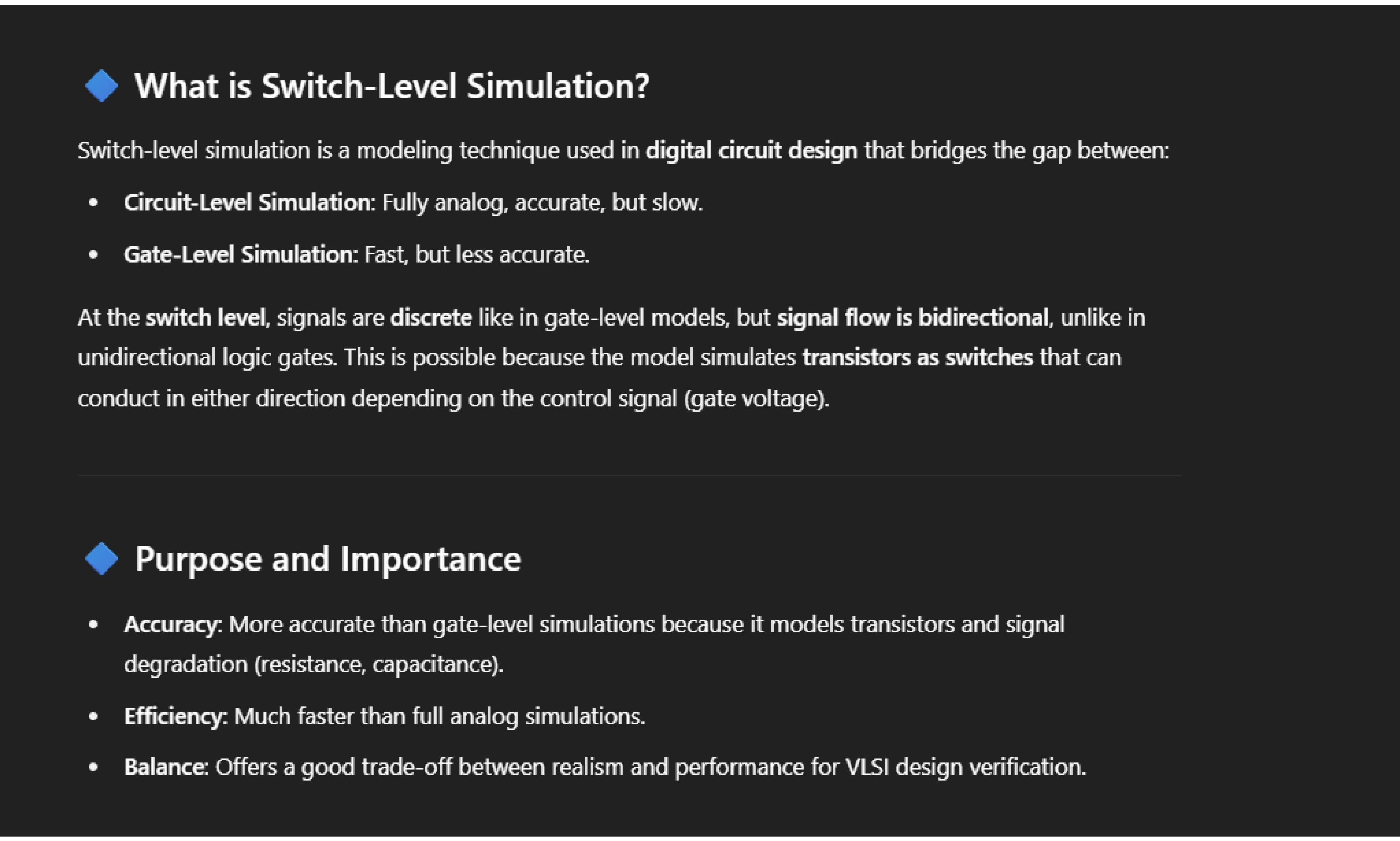


Figure 10.3 A simple circuit composed of logic gates.

```
$n_1 \leftarrow$ A;
$n_2 \leftarrow$ B;
$n_3 \leftarrow$ C;
$n_4 \leftarrow$ D;
$n_5 \leftarrow$ E;
$n_6 \leftarrow$ OR($n_1, n_2$);
$n_7 \leftarrow$ AND($n_4, n_5$);
$n_8 \leftarrow$ AND($n_6, n_3$);
$n_9 \leftarrow$ OR($n_7, n_8$);
F $\leftarrow n_9$;
```



🔷 What is Switch-Level Simulation?

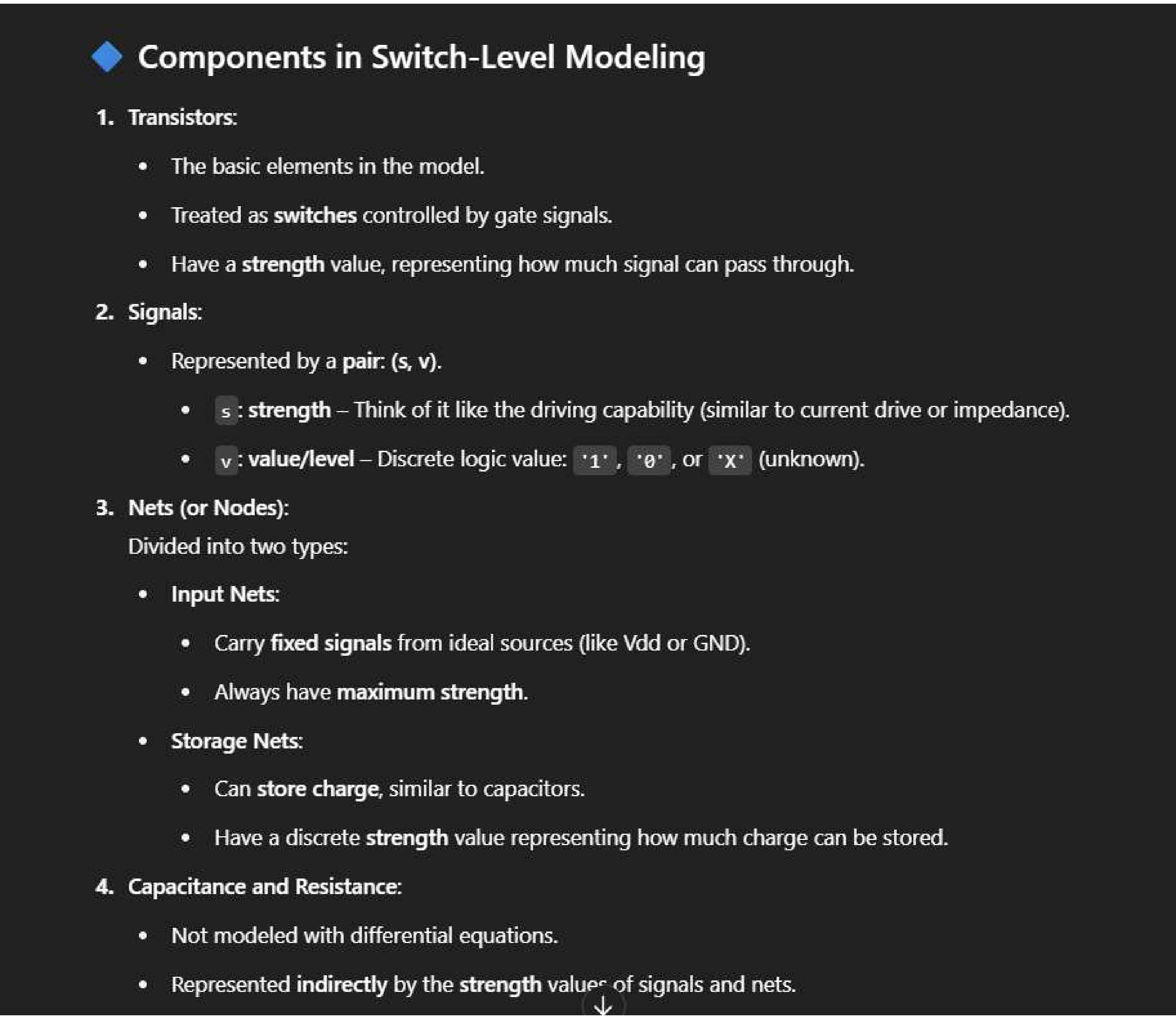
Switch-level simulation is a modeling technique used in **digital circuit design** that bridges the gap between:

- **Circuit-Level Simulation**: Fully analog, accurate, but slow.
- **Gate-Level Simulation**: Fast, but less accurate.

At the **switch level**, signals are **discrete** like in gate-level models, but **signal flow is bidirectional**, unlike in unidirectional logic gates. This is possible because the model simulates **transistors as switches** that can conduct in either direction depending on the control signal (gate voltage).

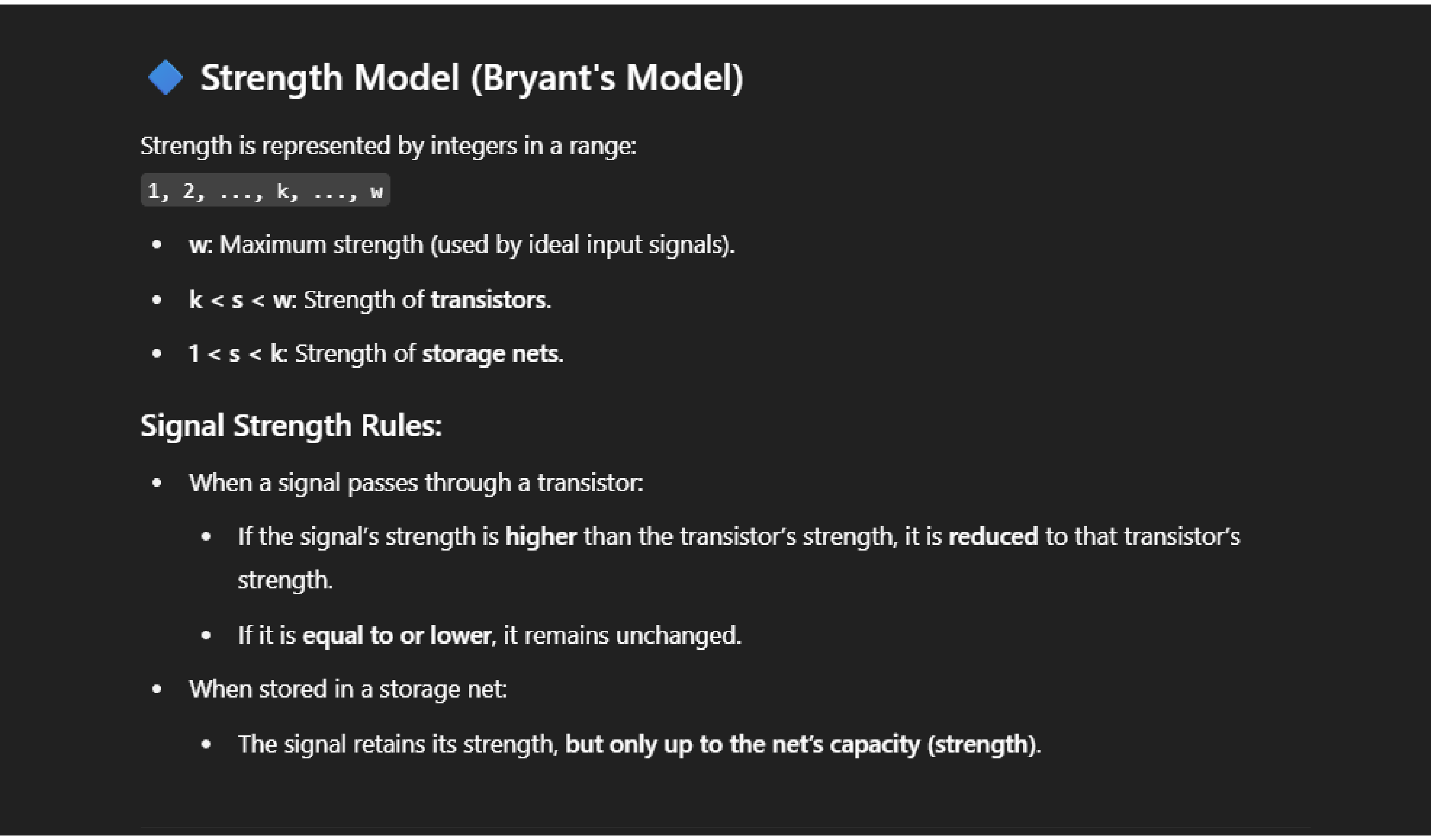
🔷 Purpose and Importance

- **Accuracy**: More accurate than gate-level simulations because it models transistors and signal degradation (resistance, capacitance).
- **Efficiency**: Much faster than full analog simulations.
- **Balance**: Offers a good trade-off between realism and performance for VLSI design verification.



🔷 Components in Switch-Level Modeling

1. **Transistors:**
   - The basic elements in the model.
   - Treated as **switches** controlled by gate signals.
   - Have a **strength** value, representing how much signal can pass through.
2. **Signals:**
   - Represented by a **pair: (s, v)**.
     - `s` : **strength** – Think of it like the driving capability (similar to current drive or impedance).
     - `v` : **value/level** – Discrete logic value: `'1'`, `'0'`, or `'X'` (unknown).
3. **Nets (or Nodes):**
   Divided into two types:
   - **Input Nets:**
     - Carry **fixed signals** from ideal sources (like Vdd or GND).
     - Always have **maximum strength**.
   - **Storage Nets:**
     - Can **store charge**, similar to capacitors.
     - Have a discrete **strength** value representing how much charge can be stored.
4. **Capacitance and Resistance:**
   - Not modeled with differential equations.
   - Represented **indirectly** by the **strength** values of signals and nets.



Strength Model (Bryant's Model)

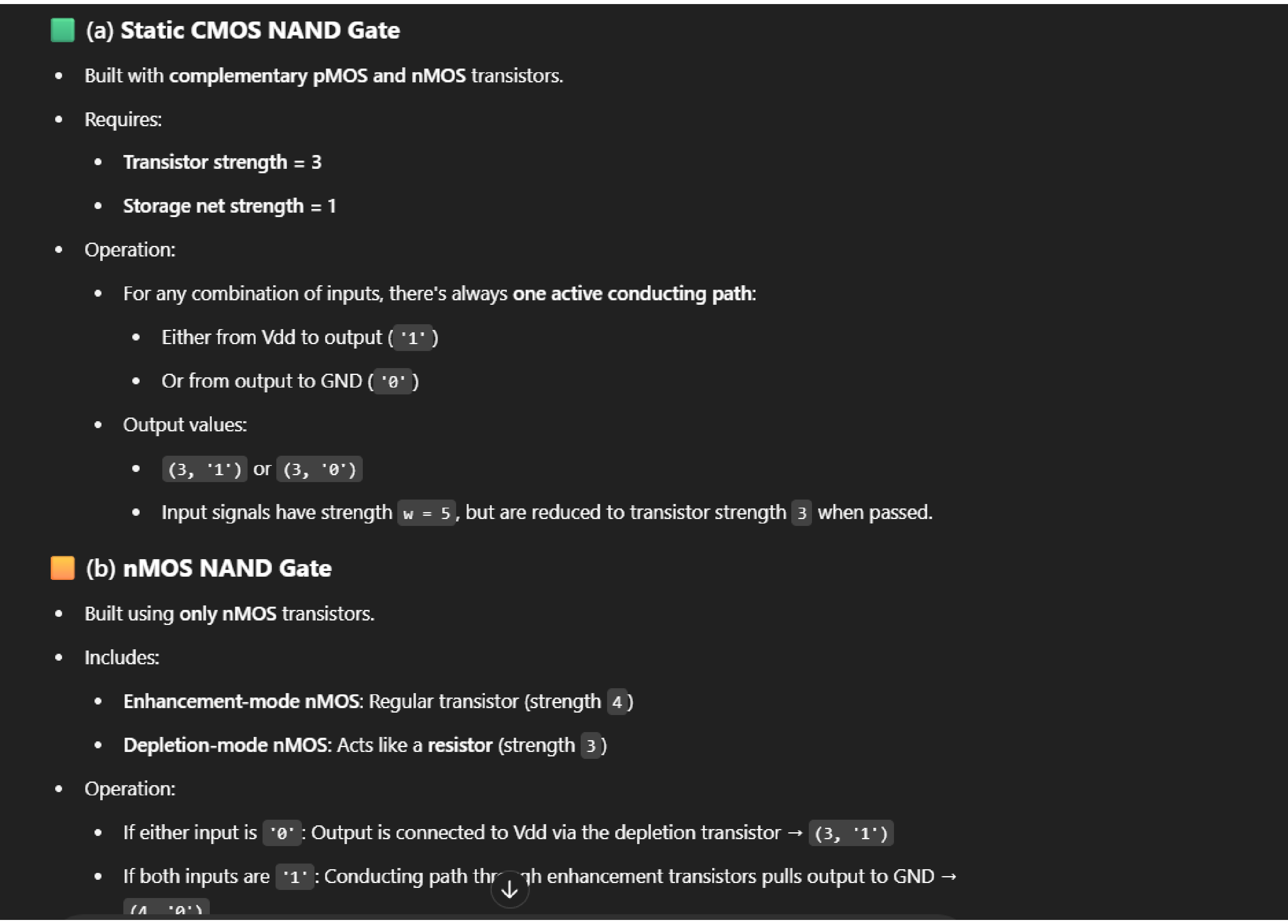
Strength is represented by integers in a range:

`1, 2, ..., k, ..., w`

- **w**: Maximum strength (used by ideal input signals).
- **k < s < w**: Strength of **transistors**.
- **1 < s < k**: Strength of **storage nets**.

Signal Strength Rules:

- When a signal passes through a transistor:
  - If the signal's strength is **higher** than the transistor's strength, it is **reduced** to that transistor's strength.
  - If it is **equal to or lower**, it remains unchanged.
- When stored in a storage net:
  - The signal retains its strength, **but only up to the net's capacity (strength)**.



🟩 (a) **Static CMOS NAND Gate**

- Built with **complementary pMOS and nMOS** transistors.
- Requires:
  - **Transistor strength** = 3
  - **Storage net strength** = 1
- Operation:
  - For any combination of inputs, there's always **one active conducting path**:
    - Either from Vdd to output (`'1'`)
    - Or from output to GND (`'0'`)
  - Output values:
    - `(3, '1')` or `(3, '0')`
    - Input signals have strength `w = 5`, but are reduced to transistor strength `3` when passed.

🟧 (b) **nMOS NAND Gate**

- Built using **only nMOS** transistors.
- Includes:
  - **Enhancement-mode nMOS**: Regular transistor (strength `4`)
  - **Depletion-mode nMOS**: Acts like a **resistor** (strength `3`)
- Operation:
  - If either input is `'0'`: Output is connected to Vdd via the depletion transistor → `(3, '1')`
  - If both inputs are `'1'`: Conducting path thr[illegible]gh enhancement transistors pulls output to GND → `(4, '0')`

Two Level Implementation of Logic Gates

Last Updated : 11 Sep, 2024

The term "*two-level logic*" refers to a logic design that uses no more than two logic gates between input and output. This does not mean that the entire design will only have two logic gates, but it does mean that the single path from input to output will only have two logic gates.

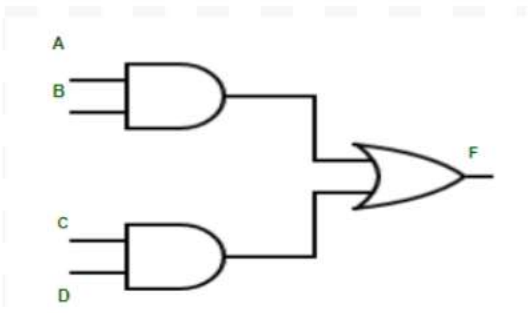
In two-level logic, irrespective of the total number of logic gates, the maximum number of logic gates that can be cascaded between any input and output is two. The outputs of first-level logic gates are connected to the inputs of second-level logic gates in this configuration.

What is Logic Gate?

A Logic gates is a basic component of digital circuits which perform basic logical operations. Just like algebraic structures or mathematical systems they execute basic logical operations that are so critical in designing intricate digital networks. A gate can operate applying Boolean algebra with input that can be given to it and yields outputs as a result. Here’s a quick overview of the primary logic gates used in two-level logic design:

- AND Gate: Outputs true (1) only if all inputs are true.
- OR Gate: Outputs true (1) if at least one input is true.
- NAND Gate: Outputs true (1) unless all inputs are true.
- NOR Gate: Outputs true (1) only if all inputs are false.

Example of two-level logic implementation



An Example of Two-level implementation

We explore four logic gates in two-level logic implementation: AND Gate, OR Gate, NAND Gate, and NOR Gate. There are a total of 16 two-level logic combinations if we choose one of these four gates at the first level and one at the second level. These are

```
AND-AND,  AND-OR,  AND-NAND,  AND-NOR,
OR-AND,   OR-OR,   OR-NAND,   OR-NOR,
NAND-AND, NAND-OR, NAND-NAND, NAND-NOR,
NOR-AND,  NOR-OR,  NOR-NAND,  NOR-NOR.
```

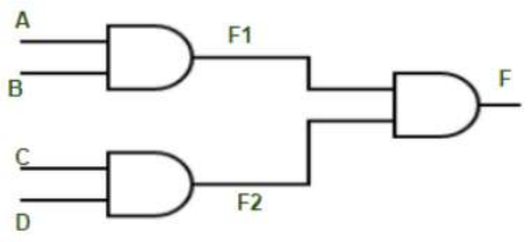
Degenerative form

Degenerative form occurs when the output of a two-level logic realization can be achieved with only one logic gate. The advantage of degenerative form is that the number of inputs of single Logic gate increases which results in the increment of fan-in of logic gates.

In those 16 combinations, there are 8 degenerate forms. Below are instances of each of these degenerate types.

AND-AND Implementation

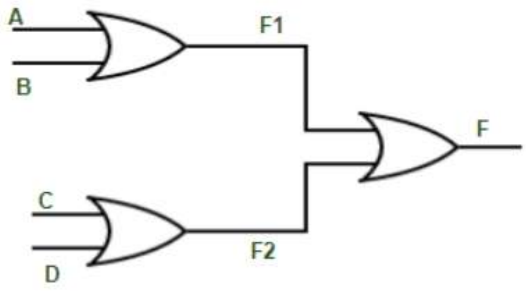
Because the entire function results in an AND function of all the inputs, this AND-AND gate combination is a degenerate form.



AND-AND Implementation

OR-OR Implementation

The output of an OR-OR gate combination is the Logic Function OR. With this combination, the OR function can be implemented with several inputs.

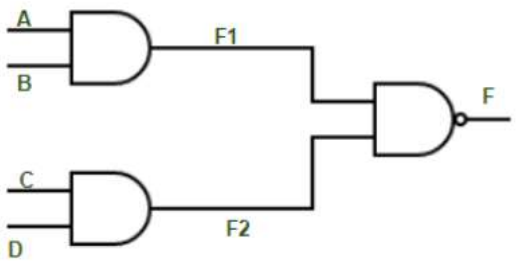


OR-OR implementation

The outputs of first-level logic gates: F1=A+B and F2=C+D. These outputs are applied as inputs of the second level, so the output of the second level is F=F1+F2 which means F=A+B+C+D.

AND-NAND Implementation

AND gate are present in the first level of this logic implementation, while NAND gates are present in the second level. An example of AND-NAND logic realization is shown in the diagram below.



AND-NAND Implementation

The outputs of first-level logic gates: F1=AB and F2=CD. These outputs are applied as inputs of the second level, so the output of the second level is F= (F1F2)' which means F=(ABCD)'.