EC311: Logic Design

Lecture notes for Logic Design

Giacomo Cappelletto

Published: September 22, 2025
Last updated: September 22, 2025

Contents

List of Figures

* * *

Chapter 1: Introduction to Logic Design

Digital logic design is the foundation of modern computing systems, from simple embedded controllers to complex processors. This course covers the systematic approach to designing digital circuits using Boolean algebra, logic gates, and systematic design methodologies.

1.1. Design Flow Overview

Digital System Design Flow — Definition 1.1.1

The modern digital design process follows a structured approach: Analog Input $\longrightarrow$ ADC $\longrightarrow$ Device $\longrightarrow$ Digitized Data $\longrightarrow$ Processing

This flow transforms real-world analog signals into digital representations that can be processed by digital logic circuits.

1.2. System Hierarchy

Digital systems are organized in a hierarchical structure for manageable design:

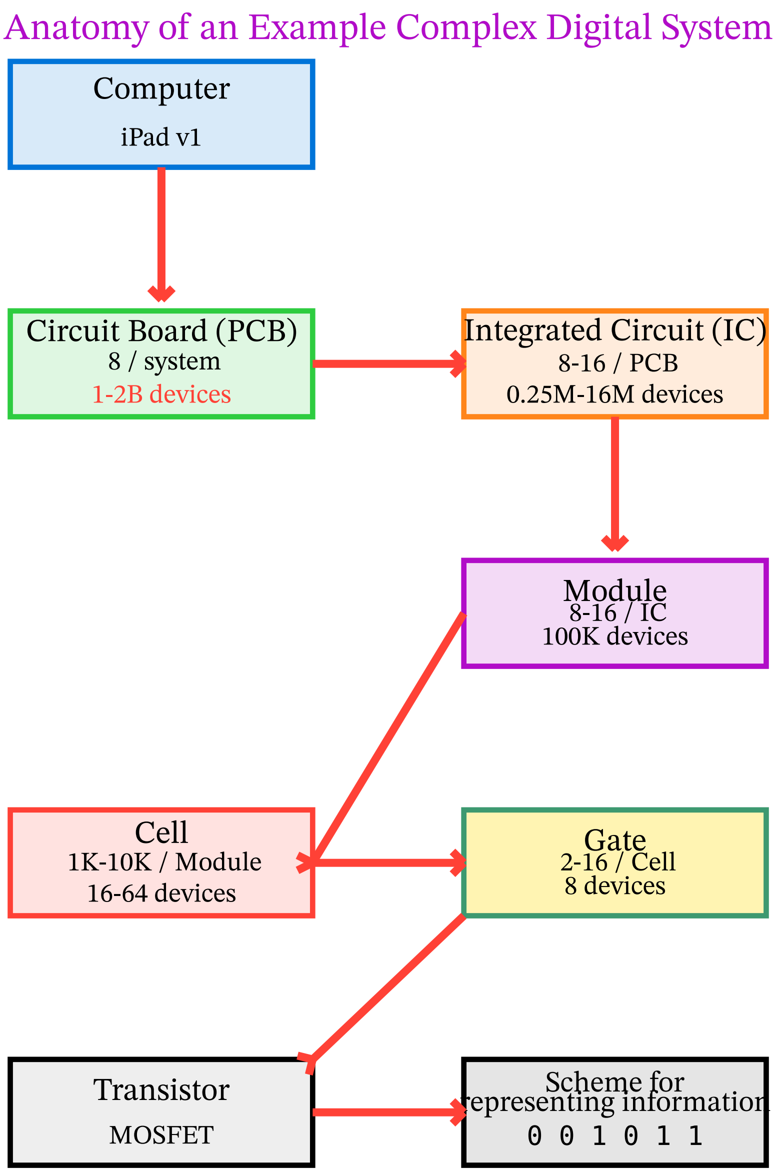


Figure 1: Complete anatomy of a complex digital system showing hierarchy from computer to transistor level

Claude Shannon — Note 1.2.1

Claude Shannon's work in the 1940s established the mathematical foundation for digital logic design, showing how Boolean algebra could be used to analyze and synthesize switching circuits.

Chapter 2: Digital Logic Fundamentals

2.1. Basic Logic Elements

Digital circuits are built from fundamental logic gates that perform Boolean operations on binary inputs.

Logic Gate — Definition 2.1.1

A logic gate is a digital circuit that implements a Boolean function. It has one or more binary inputs and produces a single binary output based on the logical relationship defined by the gate type.

2.1.1. Truth Tables and Boolean Functions

For 2 input variables (X, Y), there are $2^{2^2} = 16$ possible Boolean functions:

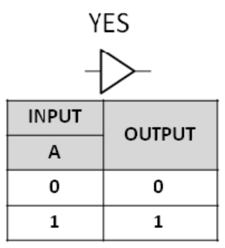
Table 1: Complete truth table showing all 16 possible Boolean functions (F_0-F_{15}) for inputs X and Y

X	Y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

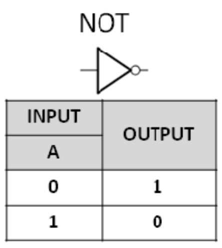
Table 2: Complete identification of all 16 Boolean functions with their logical expressions

$F_0 =$	0	(Always FALSE)	$F_8 =$	$\overline{X+Y}$	(X NOR Y)
$F_1 =$	$X \cdot Y$	(X AND Y)	$F_9 =$	$\overline{X \oplus Y}$	(X XNOR Y)
$F_2 =$	$X \cdot \overline{Y}$	(X AND NOT Y)	$F_{10} =$	$\overline{Y}$	(NOT Y)
$F_3 =$	X	(Copy X)	$F_{11} =$	$X + \overline{Y}$	(X OR NOT Y)
$F_4 =$	$\overline{X} \cdot Y$	(NOT X AND Y)	$F_{12} =$	$\overline{X}$	(NOT X)
$F_5 =$	Y	(Copy Y)	$F_{13} =$	$\overline{X} + Y$	(NOT X OR Y)
$F_6 =$	$X \oplus Y$	(X XOR Y)	$F_{14} =$	$\overline{X \cdot Y}$	(X NAND Y)
$F_7 =$	$X + Y$	(X OR Y)	$F_{15} =$	1	(Always TRUE)

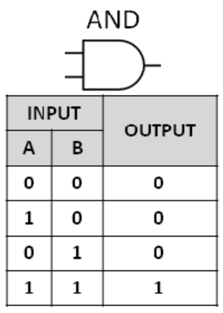
2.1.2. Standard Logic Gates



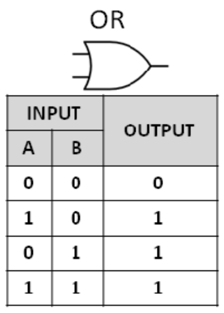
INPUT	OUTPUT
A	
0	0
1	1



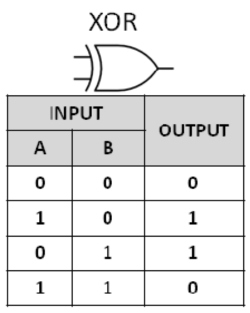
INPUT	OUTPUT
A	
0	1
1	0



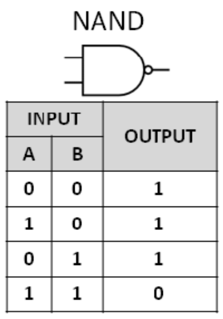
INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1



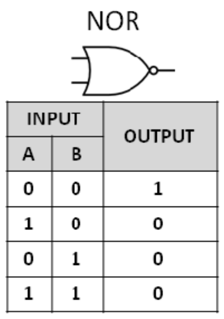
INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1



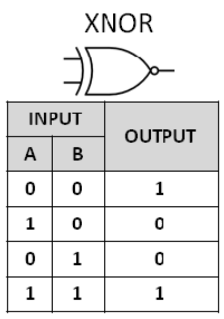
INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0



INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0



INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	0



INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	1

2.1.3. Boolean Expressions

XOR and XNOR Expressions Example 2.1.3.1

The XOR and XNOR gates have expanded Boolean forms:

- XOR: $Z = X \oplus Y = X \cdot \overline{Y} + \overline{X} \cdot Y$
- XNOR: $Z = \overline{X \oplus Y} = X \cdot Y + \overline{X} \cdot \overline{Y}$

These expressions show that XOR outputs 1 when inputs differ, while XNOR outputs 1 when inputs are the same.

2.1.4. Decimal to BCD and Binary to BCD (Double-Dabble)

Binary-Coded Decimal (BCD) Definition 2.1.4.1

BCD encodes each decimal digit (0–9) in 4 bits. For example, 2 → 0010, 4 → 0100, 3 → 0011.

Decimal → BCD Example 2.1.4.1

Encode each decimal digit independently: 243 → 2|4|3 → 0010 0100 0011.

Binary → BCD with double-dabble Example 2.1.4.2

Convert an n-bit binary number to BCD by repeating for each bit (MSB→LSB): 1) If any BCD nibble ≥ 5, add 3 to that nibble. 2) Shift the entire BCD register left by 1 and shift in the next input bit. After all shifts, the BCD nibbles are the decimal digits.

Tiny example for $243_{10} = 11110011_2$ (8 bits):

- Ones nibble hits 7 → add 3 → 10 before shifting
- Later ones hits 5 → add 3 → 8
- Tens hits 6 → add 3 → 9

After 8 shifts: BCD = 0010 0100 0011 → digits 2 4 3.

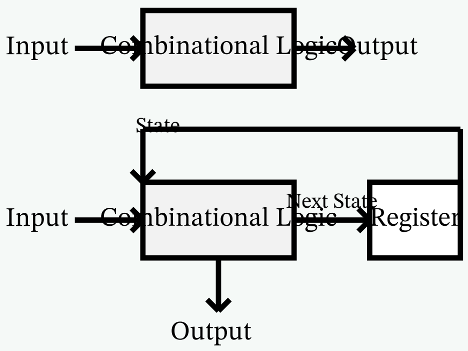
Table 3: Double-dabble run for 243_{10} (11110011_2). Left: BCD register; Right: original register. Transparent grid mimics textbook layout. Result: 0010 0100 0011 → digits 2 4 3.

`0000 0000 0000`	`11110011`	Initialization
`0000 0000 0001`	`11100110`	Shift
`0000 0000 0011`	`11001100`	Shift
`0000 0000 0111`	`10011000`	Shift
`0000 0000 1010`	`10011000`	Add 3 to ONES (was 7)
`0000 0001 0101`	`00110000`	Shift
`0000 0001 1000`	`00110000`	Add 3 to ONES (was 5)
`0000 0011 0000`	`01100000`	Shift
`0000 0110 0000`	`11000000`	Shift
`0000 1001 0000`	`11000000`	Add 3 to TENS (was 6)
`0001 0010 0001`	`10000000`	Shift
`0010 0100 0011`	`00000000`	Shift

2.2. Digital Logic Systems

Types of Digital Logic Systems Definition 2.2.1

- Combinational Logic: Output depends only on the current inputs, memoryless
- Sequential Logic: Output depends on current inputs and previous states, has memory



2.2.1. Combinational Logic Circuits

Boolean Expression Basics Definition 2.2.1.1

A Boolean function combines binary variables using logical operations:

- a, b, c are binary inputs
- Product (e.g., ab) denotes AND
- Sum (e.g., $a + b$) denotes OR
- Inversion (e.g., a') denotes NOT

Example function: $F(a, b, c) = a'bc + ab'c'$

Canonical Terms Definition 2.2.1.2

Fundamental terms appearing in Boolean expressions:

- A variable or its complement is a *literal*
- abc is a cube (product term) with 3 literals
- Minterms are products of all variables (or their complements), e.g., abc, $a'bc$, $ab'c$, $a'b'c$
- Maxterms are sums of all variables (or their complements), e.g., $a + b + c'$, $a + b' + c'$, $a' + b + c'$, $a' + b' + c'$

Standard Forms Definition 2.2.1.3

Two common normal forms for Boolean functions:

- Product of sums (POS): $F(a, b, c) = (a + b + c')(a + b' + c')$
- Sum of products (SOP): $F(a, b, c) = abc + a'bc + ab'c'$

2.2.2. Binary Cubes (Hypercube View)

Binary n-cube Definition 2.2.2.1

The binary hypercube Q_n is the graph of all n-bit vectors:

- Vertices: all bitstrings of length n (minterms)
- Edges: connect vertices that differ in exactly one bit (Hamming distance 1)
- Adjacency drives implicant merging in Karnaugh maps and algebraic minimization

Cubes as implicants Note 2.2.2.1

A product term with don't-cares (dashes) corresponds to an axis-aligned sub-cube of Q_n.

- dimension: number of don't-cares = k
- size: 2^k minterms covered
- Example: term c in 3 variables covers 4 minterms — a 2D face where $c = 1$

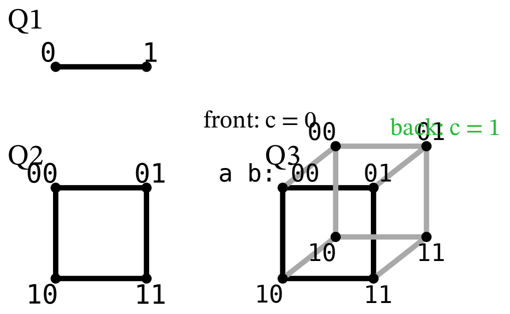
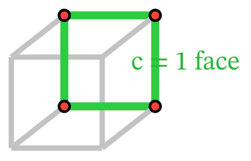


Figure 3: Binary 1-, 2-, and 3-cubes Q_1, Q_2, Q_3. Edges connect minterms that differ in one bit (Hamming distance 1).

Implicant as a sub-cube Example 2.2.2.1

Consider $F(a, b, c)$ with minterms $m(1, 3, 5, 7)$. The four minterms lie on the face where $c = 1$, forming a 2D cube (size 4). The corresponding prime implicant is simply c.



Implicant: c (covers 4 minterms)

Figure 4: Face $c = 1$ (green) is a 2D cube covering 4 minterms: $001, 011, 101, 111$ — implicant c.

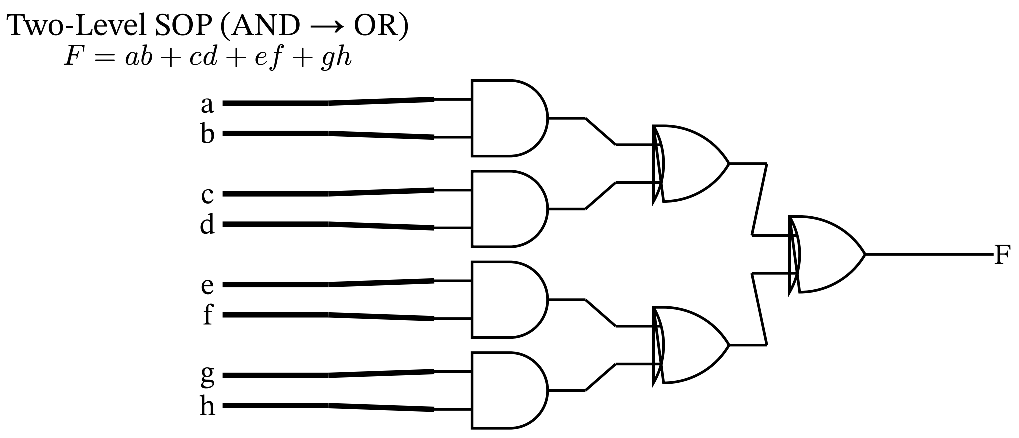


Figure 5: SOP implementation using only 2-input gates: $F = ab + cd + ef + gh$

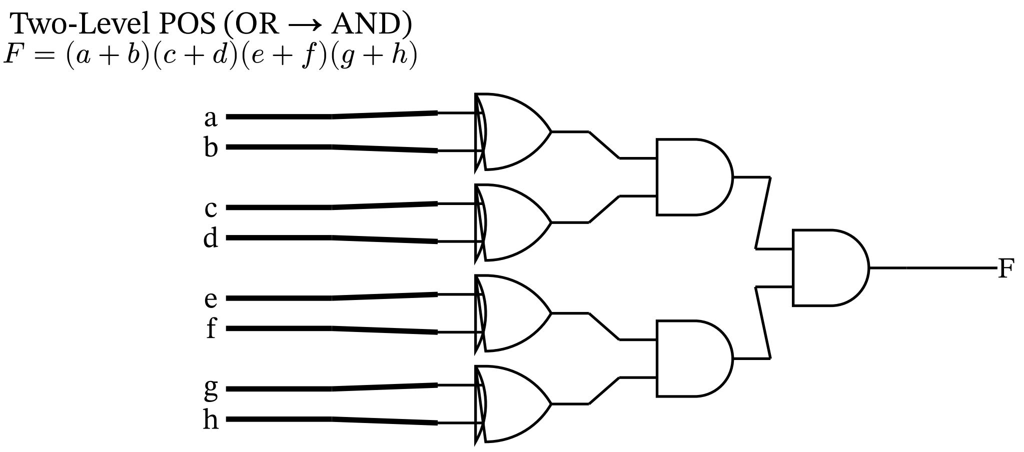

Two-Level POS (OR $\longrightarrow$ AND)
$F = (a+b)(c+d)(e+f)(g+h)$
a
b
c
d
e
f
g
h
F

Figure 6: POS implementation using only 2-input gates: $F = (a+b)(c+d)(e+f)(g+h)$

2.2.3. Decimal to BCD and Binary to BCD (Double-Dabble)

Binary-Coded Decimal (BCD) Definition 2.2.3.1

BCD encodes each decimal digit (0–9) in 4 bits. For example, 2 $\longrightarrow$ 0010, 4 $\longrightarrow$ 0100, 3 $\longrightarrow$ 0011.

Decimal $\longrightarrow$ BCD Example 2.2.3.1

Encode each decimal digit independently: 243 $\longrightarrow$ 2|4|3 $\longrightarrow$ 0010 0100 0011.

Binary $\longrightarrow$ BCD with double-dabble Example 2.2.3.2

Convert an n-bit binary number to BCD by repeating for each bit (MSB$\longrightarrow$LSB): 1) If any BCD nibble $\geq$ 5, add 3 to that nibble. 2) Shift the entire BCD register left by 1 and shift in the next input bit. After all shifts, the BCD nibbles are the decimal digits.

Tiny example for $243_{10} = 11110011_2$ (8 bits):

- Ones nibble hits 7 $\longrightarrow$ add 3 $\longrightarrow$ 10 before shifting
- Later ones hits 5 $\longrightarrow$ add 3 $\longrightarrow$ 8
- Tens hits 6 $\longrightarrow$ add 3 $\longrightarrow$ 9

After 8 shifts: BCD = 0010 0100 0011 $\longrightarrow$ digits 2 4 3.

Table 4: Double-dabble run for 243_{10} (11110011_2). Left: BCD register; Right: original register. Transparent grid mimics textbook layout. Result: 0010 0100 0011 $\longrightarrow$ digits 2 4 3.

0000 0000 0000	11110011	Initialization
0000 0000 0001	11100110	Shift
0000 0000 0011	11001100	Shift
0000 0000 0111	10011000	Shift
0000 0000 1010	10011000	Add 3 to ONES (was 7)
0000 0001 0101	00110000	Shift
0000 0001 1000	00110000	Add 3 to ONES (was 5)
0000 0011 0000	01100000	Shift
0000 0110 0000	11000000	Shift
0000 1001 0000	11000000	Add 3 to TENS (was 6)
0001 0010 0001	10000000	Shift
0010 0100 0011	00000000	Shift

Chapter 3: Verilog HDL

3.1. Module, ports, and declarations

Module, ports, and declarations Definition 3.1.1

Minimal Verilog module structure:

- `module <name> (port_list);` begins a module; `endmodule` closes it (no semicolon).
- Port directions and internal nets are declared after the module header:
  - `input`, `output`, `inout` declarations end with semicolons.
  - `wire` for combinational nets; `reg` for variables assigned in procedural blocks.
- Gate/module instances end with semicolons. Instance names are optional.
- Comments: `//` single-line, `/* ... */` multi-line.

Gate-level example: Simple_Circuit Example 3.1.1

```verilog
// Verilog model: Simple_Circuit
module Simple_Circuit (A, B, C, D, E);
  output D, E;       // outputs (semicolon)
  input  A, B, C;    // inputs  (semicolon)
  wire   w1;         // internal net (semicolon)

  and G1 (w1, A, B);    // instance name optional
  not G2 (E, C);        // inverter
  or  G3 (D, w1, E);    // 2-input OR
endmodule               // no semicolon here

/* Notes:
   - Port order for primitive gates is (output, inputs...)
   - Replace primitives with module names to instantiate submodules
*/
```

Common syntax reminders Note 3.1.1

- Every declaration/instance statement must end with `;`.
- Use commas to separate names in a declaration: `output D, E;`
- Avoid mixing `reg` and `wire` on the same identifier.
- Prefer named port connections for large modules: `.port(signal)`.

3.2. Values in Verilog

Four-state logic values Definition 3.2.1

Verilog signals are four-state by default:

- `0`: logic zero, false
- `1`: logic one, true
- `X`: unknown (could be 0, 1, Z, or in transition). Useful for don't-cares, uninitialized regs, and catching contention in simulation.
- `Z`: high-impedance (floating). Common on tri-state buses and bidirectional pins.

Table 5: Verilog four-state logic values

Value	Meaning
`0`	logic zero, false
`1`	logic one, true
`X`	unknown; may be 0, 1, Z, or in transition
`Z`	high-impedance; floating node used for tri-state/bidirectional pins

3.3. Number representation in Verilog

Sized literals: <size>'<base><digits> — Definition 3.3.1

- `<size>`: number of bits in the literal (e.g., `8`, `16`).
- `<base>`: `b` binary, `d` decimal, `o` octal, `h` hex.
- `<digits>`: the value; underscores are allowed for readability (e.g., `8'b1010_1101`).
- Digits may include `x` and `z` to encode unknowns/high-Z: `4'b1xz0`.

Literal examples — Example 3.3.1

```verilog
4'b0000   // 4-bit binary 0000
4'b0      // also 4-bit 0000
2'b11     // two-bit value (decimal 3)
16'hABBA  // hexadecimal number
```

3.4. Data types and vectors

Nets vs variables — Definition 3.4.1

- `wire`: physical connection; driven by continuous assignments or outputs of instances.
- `reg`: holds a value assigned in procedural blocks (`always`, `initial`). Not necessarily a latch/flip-flop; storage only if the procedure implies it.
- `integer`, `time`: 32-bit signed and simulation time types (for testbenches).

Vectors, indexing, concatenation — Definition 3.4.2

- Vectors: `wire [MSB:LSB] a;` (e.g., `wire [7:0] a;`).
- Bit-select: `a[i]`; part-select: `a[MSB:LSB]`.
- Concatenation: `{a, b, 2'b11}`; replication: `{4{1'b0}}`.

Vector examples — Example 3.4.1

```verilog
wire  [7:0] data;
reg   [3:0] nibble;
assign data = {nibble, 4'b0000};
wire  last_bit = data[0];
wire  upper    = data[7:4];
```

3.5. Operators and comparisons

Table 6: Common Verilog operators. Use parentheses to avoid precedence pitfalls. Case equality (`===`, `!==`) treats X/Z as distinct.

Kind	Operators	Example
Bitwise	`~ & | ^ ~^ ^~`	`a & b`, `~a`
Logical	`! && ||`	`a && b`
Reduction	`& | ^ ~& ~| ^~ ~^`	`&vec`
Shift	`<< >>`	`a << 1`
Relational	`== != > < >= <=`	`a == b`
Case equality	`=== !==`	`a === 1'bx`
Arithmetic	`+ - * / %`	`a + b`

3.6. Continuous vs procedural assignments

Assignment styles — Definition 3.6.1

- Continuous: `assign y = a & b;` (drives a net continuously).
- Procedural: inside `always` / `initial`. Produces combinational or sequential hardware depending on style.

Combinational vs sequential — Example 3.6.1

```verilog
// Combinational: all RHS read, LHS written for every path
always @(*) begin
  y = (a & b) | c;   // blocking OK in combinational
end

// Sequential (flop with async active-low reset)
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) q <= 1'b0;     // non-blocking
  else        q <= d;
end
```

3.7. Blocking vs non-blocking

Guidelines — Note 3.7.1

- Use blocking `=` in combinational `always @(*)` blocks.
- Use non-blocking `<=` in clocked `always @(posedge ...)` blocks.
- Mixing in the same always block can cause race-like behavior in simulation.

Pitfall and fix Example 3.7.1

verilog
```verilog
// Pitfall (uses blocking in sequential logic)
always @(posedge clk) begin
  a = b;
  b = a;  // b gets old a? No, both become old b in sim
end

// Correct
always @(posedge clk) begin
  a <= b;
  b <= a;
end
```

3.8. Parameters and module parameterization

Parameters and named port/param Example 3.8.1

verilog
```verilog
module adder #(parameter WIDTH = 8) (
  input  [WIDTH-1:0] a, b,
  output [WIDTH:0]   sum
);
  assign sum = a + b;
endmodule

// Instantiation with named parameter and ports
adder #(.WIDTH(16)) u_add (
  .a(a16), .b(b16), .sum(sum16)
);
```

3.9. Generate constructs (arrays of hardware)

Generate for-loop Example 3.9.1

verilog
```verilog
module and_array #(parameter N = 8) (
  input  [N-1:0] a, b,
  output [N-1:0] y
);
  genvar i;
  generate
    for (i = 0; i < N; i = i + 1) begin : gen
      assign y[i] = a[i] & b[i];
    end
  endgenerate
endmodule
```

3.10. Combinational and sequential templates

Always block templates Example 3.10.1

```verilog
// Combinational template (avoids latches)
always @(*) begin
  // default assignments
  y = '0;
  // logic
  y = (a & b) | c;
end

// Sequential template with sync reset
always @(posedge clk) begin
  if (rst) q <= '0; else q <= d;
end
```

3.11. Modern Technology: MOS and CMOS

MOSFET Technology Definition 3.11.1

Modern digital circuits primarily use MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor) technology:

- NMOS: N-channel transistors that conduct when gate is HIGH
- PMOS: P-channel transistors that conduct when gate is LOW
- CMOS: Complementary MOS using both NMOS and PMOS for low power consumption

The CMOS inverter we studied earlier demonstrates how these transistors work together to create efficient digital switches with minimal power consumption except during transitions.

Chapter 4: Circuit Analysis and Abstraction

4.1. Abstraction Levels

Design Abstraction Note 4.1.1

Digital design uses multiple levels of abstraction:

1. Behavioral Description: Specification of what the circuit should do
2. Circuit Schematic: Gate-level implementation
3. Hardware Implementation: Physical realization in silicon

Each level abstracts away lower-level details while maintaining functionality.

4.2. CMOS NOT Gate Implementation

The CMOS (Complementary MOS) NOT gate demonstrates the fundamental principle of modern digital logic design using both NMOS and PMOS transistors.

4.2.1. Transistor Operation as Switches

MOS Transistor Switch Model — Definition 4.2.1.1

MOSFET transistors can be modeled as voltage-controlled switches:

- NMOS: Acts like a switch between drain and source, controlled by gate voltage
  - Gate HIGH (VDD) $\longrightarrow$ Switch CLOSED (conducts)
  - Gate LOW (GND) $\longrightarrow$ Switch OPEN (does not conduct)
- PMOS: Acts like an inverted switch (note the bubble on gate symbol)
  - Gate LOW (GND) $\longrightarrow$ Switch CLOSED (conducts)
  - Gate HIGH (VDD) $\longrightarrow$ Switch OPEN (does not conduct)

4.2.2. Complete CMOS Inverter Circuit

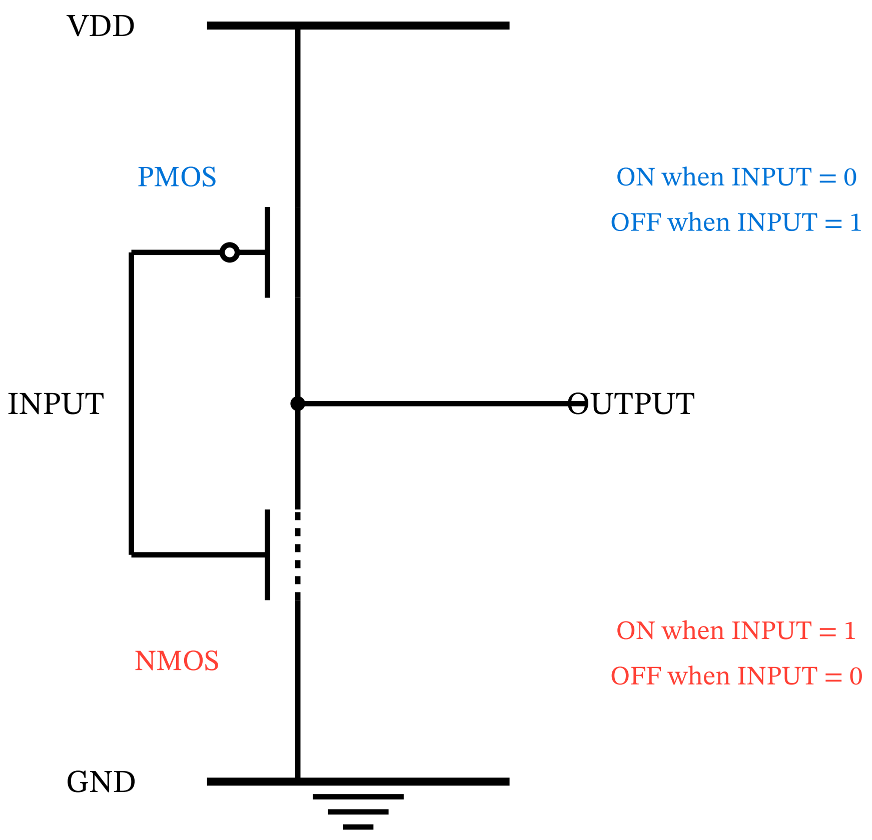


Figure 7: CMOS NOT gate schematic showing complementary operation

4.2.3. Switch Model Abstraction

To understand why we need both NMOS and PMOS, consider the resistor abstraction:

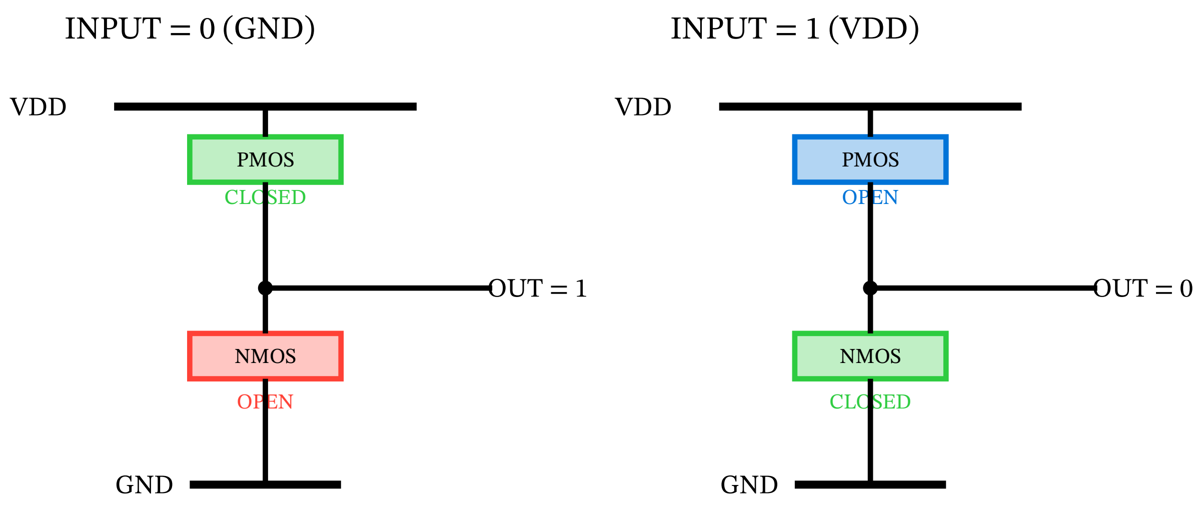


Figure 8: Switch model showing why both NMOS and PMOS are necessary

4.2.4. Why Both Transistor Types Are Essential

Necessity of Complementary Transistors — Example 4.2.4.1

Each transistor type serves a specific role:

PMOS (Pull-up network):

- Connects output to VDD when input is LOW
- Good at "pulling up" to logic 1
- Poor at "pulling down" to logic 0

NMOS (Pull-down network):

- Connects output to GND when input is HIGH
- Good at "pulling down" to logic 0
- Poor at "pulling up" to logic 1

Together they provide:

- Strong drive in both directions (full rail-to-rail output)
- No static current path (one is always OFF)
- Fast switching with minimal power consumption

4.2.5. Operation Analysis

Table 7: CMOS inverter truth table and current paths

INPUT	PMOS	NMOS	OUTPUT	Current Path
0 (GND)	ON	OFF	1 (VDD)	VDD ⟶ PMOS ⟶ Output
1 (VDD)	OFF	ON	0 (GND)	Output ⟶ NMOS ⟶ GND

Power Consumption Advantage — Note 4.2.5.1

The complementary nature ensures that in steady state, one transistor is always OFF, preventing any direct current path from VDD to GND. Power is only consumed during switching transitions, making CMOS extremely power-efficient compared to other logic families.

4.3. Alternative Single-Transistor Approaches (Why They Don't Work)

NMOS-only Inverter Problems Example 4.3.1

If we tried to build an inverter with only NMOS:

- Could pull output LOW when input is HIGH
- Cannot pull output HIGH when input is LOW (would need a resistor)
- Resistor would cause static power consumption
- Weak HIGH output level (degraded logic levels)

This is why early logic families like NMOS required large pull-up resistors and consumed significant power.

The CMOS approach solves all these problems by using the PMOS as an "active pull-up" device that strongly drives the output HIGH while consuming no static power.