Digital Logic Design

Ryan Hou

2024-12-21

Table of contents

Preface

This is my notes on digital logic design.

Resources

Some relevant resources:

- EECS 270 - Logic Design (University of Michigan)
- Digital Design and Computer Architecture (ETH Zurich)

Textbooks:

- J. F. Wakerly, Digital Design: Principles and Practices, 4th ed., Prentice-Hall.
- J. P. Hayes, Introduction to Digital Logic Design, Addison-Wesley.
- C. H. Roth, Jr., Fundamentals of Logic Design.
- R. H. Katz, Contemporary Logic Design, Prentice-Hall.
- D. Thomas, P. Moorby, The Verilog Hardware Description Language.

1 Introduction

1.1 Perspective

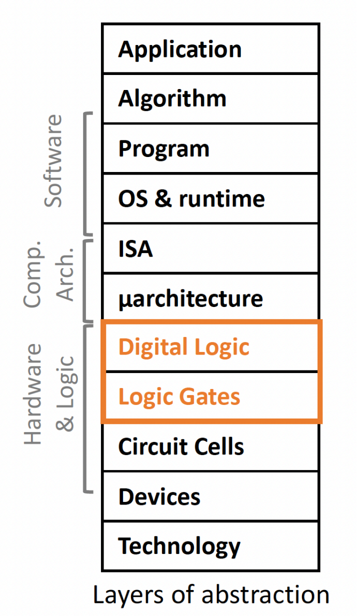


Figure 1.1: Digital Logic Design in the Computing Stack (figure from EECS270-W24)

> Note 1: Definition - Digital
>
> **Digital** signals represent information as *discrete* values, typically binary values where two valid states exist: 0 (low, false) or 1 (high, true).

This notebook focuses on the design of **digital circuits**. We study both the logic/math used to build digital systems (Boolean algebra), as well as the circuit design implications (transistors, timing, etc).

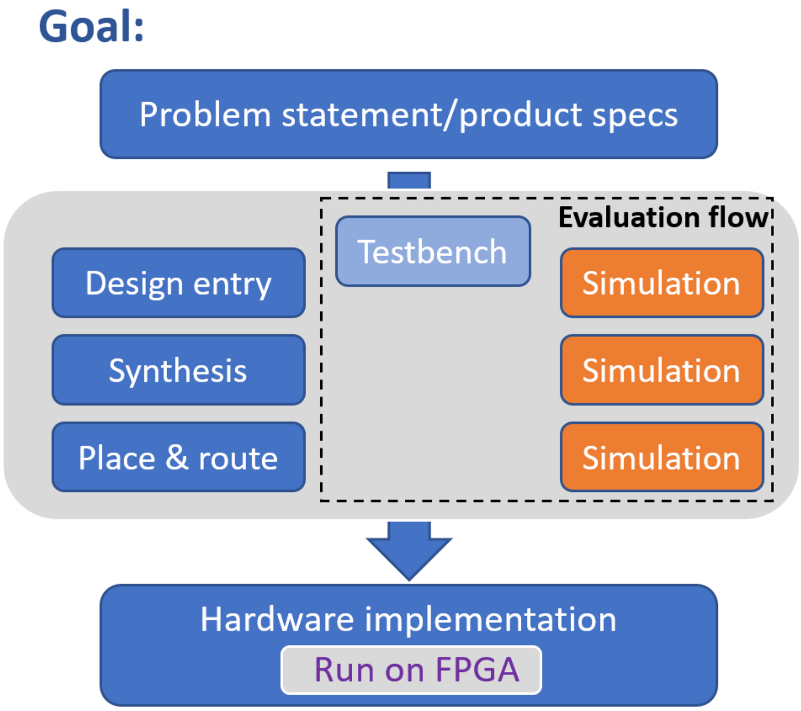


Figure 1.2: High-level design flow of a digital circuit to be run on FPGA

1.2 (Very) High-Level Digital Circuit Design Flow

The design flow of a digital circuit starts off with a problem statement or design specification. Digital circuits are then described by the designer in a **Hardware Description Language (HDL)**, most commonly **Verilog/SystemVerilog** or **VHDL**. The design is then simulated with a **testbench**, which feeds the design with test inputs. During **simulation**, we can use tools to inspect the state of the signals in the circuit to analyze, debug, and evaluate the design. At this point, such a **behavioral** description of the design merely describes the functionality and not yet its physical implementation. **Synthesis** maps the behavioral description of the design into **netlist** of standard cells, which indicates the physical mapping to circuit components. The **place and route** process then physically places the netlist of cells and routes the wires to connect the components, generating a hardware implementation.

This notebook will cover basic Verilog. For more in-depth notes on Verilog/SystemVerilog, please refer to my other notebook.

2 Binary Basics

2.1 Analog vs Digital

In contrast to the *discrete* **digital** signals, **analog** signals are *continuous.* Signals from the physical world are inherently analog (e.g. sound, light, temperature, voltage). However, modern computing systems are primarily digital because of several key advantages:

- Reliability: Provides more noise resistance since it operates at low or high levels
- Digitized signals can represent analog values with good precision given enough digits
- Ease of data storage, transmission, and compression
- Digital circuit components are more cost-effective and scalable compared to analog components

2.2 Why Binary?

- Storing/transmitting binary values is much easier than three or more values
- Binary switches are easier, more robust, and more noise tolerant in circuit implementation

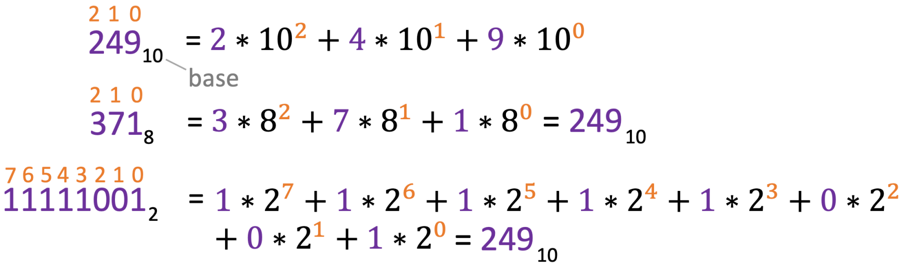
Note that digital   binary!

2.3 Data Encoding

Numbers are encoded in a system using digits and powers of a base number. In simpler terms, each position of a number represents a quantity. And the digit in each position indicates how many of that quantity there are in the number.

A **bit** is a binary digit. The total number of integers that can be represented with n bits is 2^n.

The maximum (unsigned) decimal number that can be represented with n bits is Max Value $= 2^n - 1$. This range can be generalized to other bases:



$$249_{10} = 2 * 10^2 + 4 * 10^1 + 9 * 10^0$$

$$371_8 = 3 * 8^2 + 7 * 8^1 + 1 * 8^0 = 249_{10}$$

$$11111001_2 = 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 249_{10}$$

Figure 2.1: Data encoding in base 10, 8, and 2. Figure from EECS270-W24

ℹ Note 2: Equation - Max Unsigned Decimal Value with n Digits

$$\text{Max Value} = base^n - 1$$

Common number systems:

- Base-16: **Hexadecimal**
- Base-10: **Decimal**
- Base-8: **Octal**
- Base-2: **Binary**

The number of bits n needed to represent an unsigned decimal number x is given below:

ℹ Note 3: Equation - Number of Bits to Represent Unsigned Decimal Number

$$n = ceil(log_2(x + 1))$$

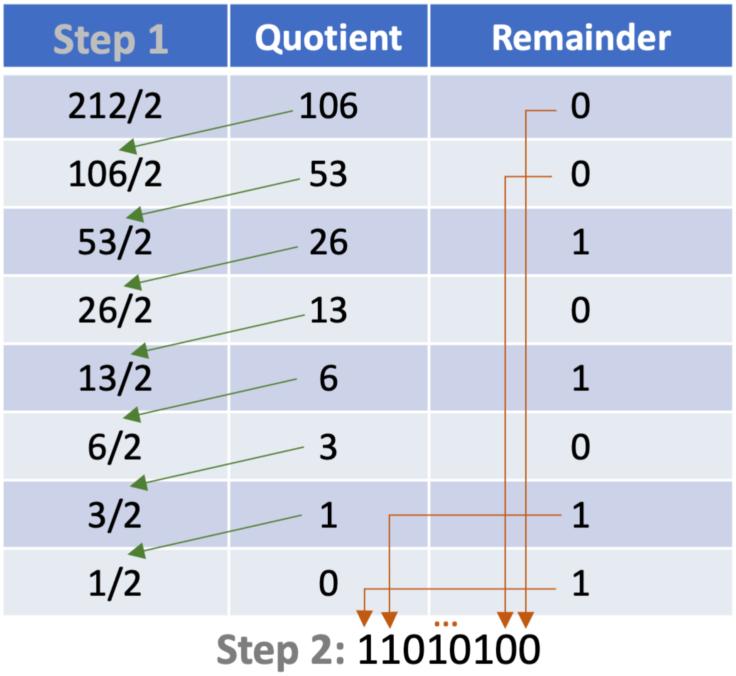
where the ceil() function is a ceiling function that rounds up to the nearest integer.

2.3.1 Conversions

2.3.1.1 Decimal - Binary

To convert from decimal to binary:

- Step 1: Divide the given number repeatedly by 2 until you get 0 as the quotient.
- Step 2: Write the remainders in reverse order.



Step 1	Quotient	Remainder
212/2	106	0
106/2	53	0
53/2	26	1
26/2	13	0
13/2	6	1
6/2	3	0
3/2	1	1
1/2	0	1

Step 2: 11010100

Figure 2.2: Example decimal to binary conversion. Figure source: EECS 270

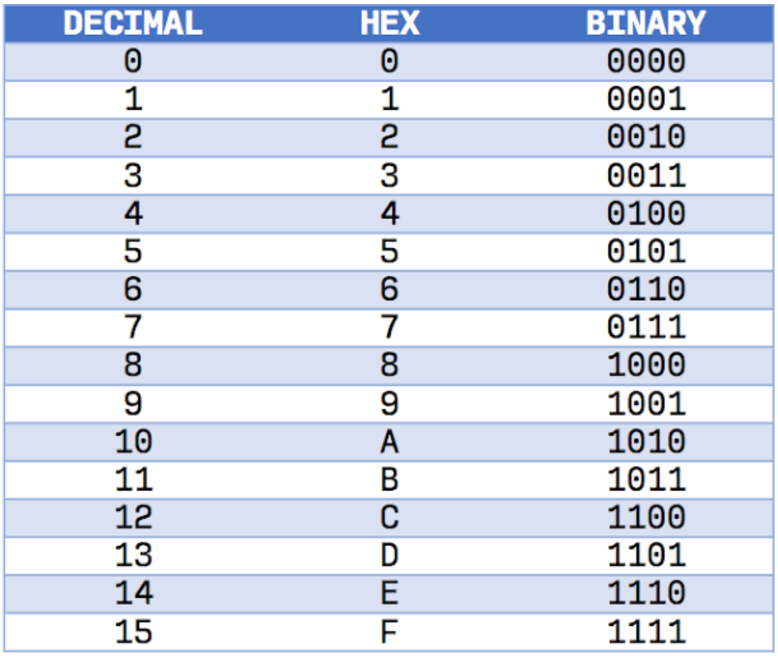
2.3.2 Hex - Octal - Binary

Hexadecimal and octal have bases that are powers of 2, which makes conversion much simpler. Since hex is base 16, which is 2^4, we can split each **hex digit into 4 bits** when converting to binary, conversely group every 4 bits into 1 hex digit. Similarly, an **octal digit corresponds to 3 bits**.

2.3.3 ASCII

Text can also be encoded by numbers. ASCII is a common character encoding standard that represents a character in 8 bits.

An ASCII conversion chart can be found here.



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

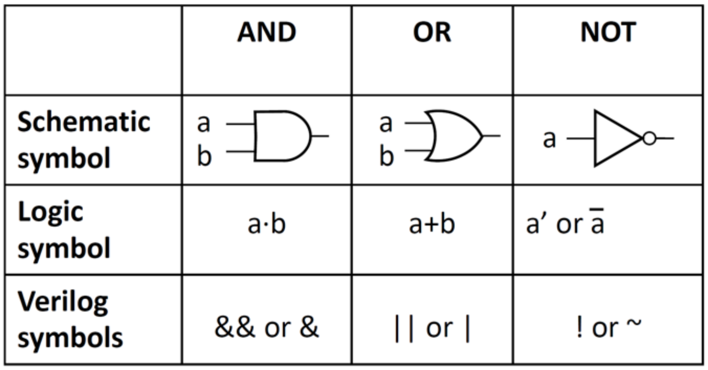
Figure 2.3: Hexadecimal Conversion Chart. Source

3 Boolean Algebra

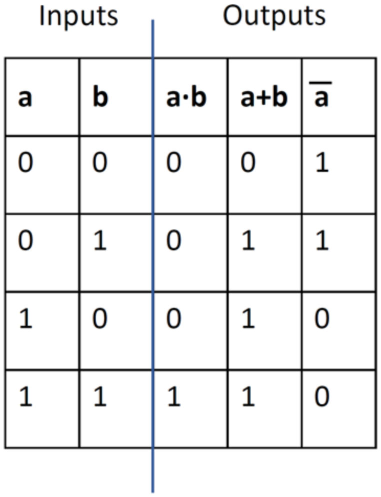
3.1 Simple Equations

Operator Precedence Rules: 1. NOT (highest priority) 2. AND 3. OR

A **truth table** relates the inputs to a combinational logic circuit to its outputs, showing output for every possible combination of inputs.



	AND	OR	NOT
Schematic symbol	a b	a b	a
Logic symbol	a·b	a+b	a’ or $\overline{a}$
Verilog symbols	&& or &	|| or |	! or ~



Inputs		Outputs		
a	**b**	**a·b**	**a+b**	$\overline{\mathbf{a}}$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

To generalize, for a Boolean function with N binary inputs:

- There are 2^N possible input combinations, i.e. number of rows in the truth table
- There are 2^{2^N} “semantically” different Boolean functions
  - Derivation: There are 2^N entries in the truth table. The output of each truth table entry takes on 2 possible values, thus there are 2^{2^N} different ways you can pick a combination of outputs.

3.2 Boolean Algebra Basics

Boolean Algebra is an algebra manipulating Boolean values of 0s and 1s.

We start with:

- **Axioms**: basic things about objects and operations that we assume to be true as the start
- Then using axioms we derive:
  - **Laws and theorems**: which allow us to manipulate Boolean expressions and perform simplifications
- Then we derive more sophisticated **properties** for manipulating Boolean equations

3.3 Axioms

ℹ Note 4: Axiom - Binary

[A1] $a = 0$ if $a \neq 1$
[A1'] $a = 1$ if $a \neq 0$

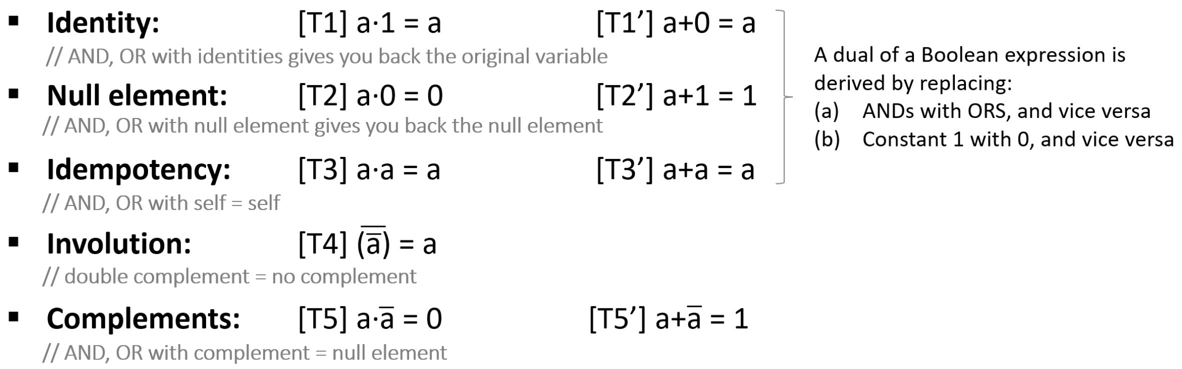
ℹ Note 5: Axiom - Complement

[A2] if $a = 0$, then $\bar{a} = 1$
[A2'] if $a = 1$, then $\bar{a} = 0$

ℹ Note 6: Axiom - AND and OR

[A3] $0 \cdot 0 = 0$
[A3'] $1 + 1 = 1$
[A4] $1 \cdot 1 = 1$
[A4'] $0 + 0 = 0$
[A5] $0 \cdot 1 = 1 \cdot 0 = 0$
[A5'] $1 + 0 = 0 + 1 = 1$

3.4 Single Variable Theorems



- **Identity:** [T1] $a \cdot 1 = a$ [T1’] $a+0 = a$
  // AND, OR with identities gives you back the original variable
- **Null element:** [T2] $a \cdot 0 = 0$ [T2’] $a+1 = 1$
  // AND, OR with null element gives you back the null element
- **Idempotency:** [T3] $a \cdot a = a$ [T3’] $a+a = a$
  // AND, OR with self = self
- **Involution:** [T4] $(\overline{\overline{a}}) = a$
  // double complement = no complement
- **Complements:** [T5] $a \cdot \overline{a} = 0$ [T5’] $a+\overline{a} = 1$
  // AND, OR with complement = null element

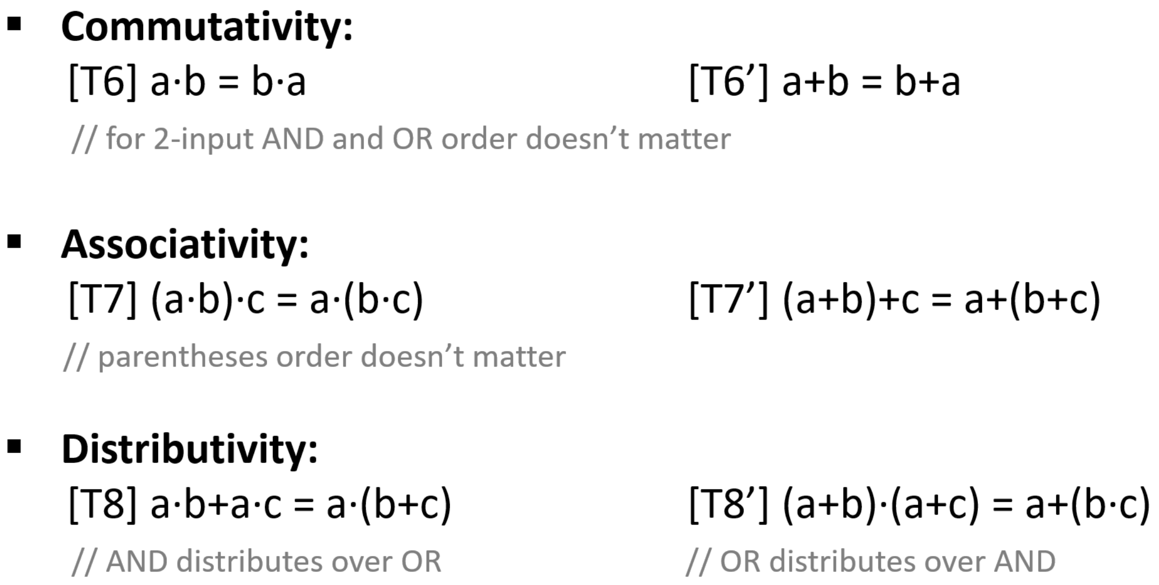
A dual of a Boolean expression is derived by replacing:
(a) ANDs with ORS, and vice versa
(b) Constant 1 with 0, and vice versa

Figure 3.1: A list of single variable theorems. Source: EECS270-W24

To prove the idempotency theorems [T3] and [T3’]:

TODO:

3.5 Two and Three Variable Theorems



- **Commutativity:**
  [T6] $a \cdot b = b \cdot a$ [T6’] $a+b = b+a$
  // for 2-input AND and OR order doesn’t matter
- **Associativity:**
  [T7] $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ [T7’] $(a+b)+c = a+(b+c)$
  // parentheses order doesn’t matter
- **Distributivity:**
  [T8] $a \cdot b+a \cdot c = a \cdot (b+c)$ [T8’] $(a+b) \cdot (a+c) = a+(b \cdot c)$
  // AND distributes over OR // OR distributes over AND

- **Covering:**

  [T9] $a\cdot(a+b) = a$ [T9’] $a+a\cdot b = a$

  // a covers b

- **Combining:**

  [T10] $a\cdot b+a\cdot\overline{b} = a$ [T10’] $(a+b)\cdot(a+\overline{b}) = a$

  // combine two terms to one

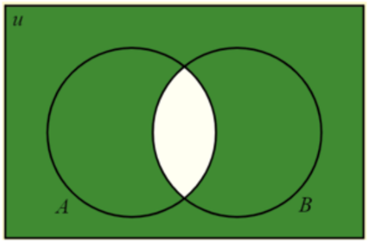
- **Consensus:**

  [T11] $a\cdot b+\overline{a}\cdot c+b\cdot c = a\cdot b+\overline{a}\cdot c$ [T11’] $(a+b)\cdot(\overline{a}+c)\cdot(b+c) = (a+b)\cdot(\overline{a}+c)$

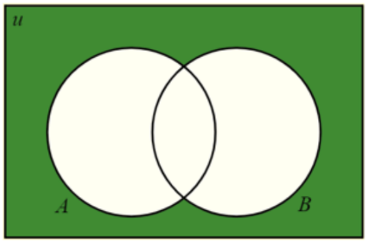
  //conjunction of all the unique literals of the terms

3.6 De Morgan’s Law

[T12] $(\overline{a\cdot b\cdot c\cdot\ldots}) = \overline{a}+\overline{b}+\overline{c}+\ldots$ [T12’] $(\overline{a+b+c+\ldots}) = \overline{a}\cdot\overline{b}\cdot\overline{c}\cdot\ldots$



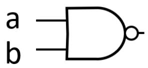
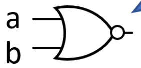
$(A\cap B)' = A'\cup B'$

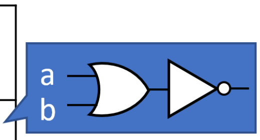


$(A\cup B)' = A'\cap B'$

De Morgan’s law essentially says that **negation distributes over AND and OR by inverting operators and complementing terms**. In simpler words, De Morgan’s swaps ANDs and ORs while pushing the complements inside the variables.

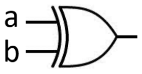
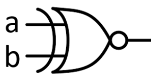
3.7 More Boolean Gates

	NAND	**NOR**
Schematic symbol		
Logic symbol	$\overline{a \cdot b}$	$\overline{a+b}$
Verilog expression	~(a&b)	~(a|b)



- NAND returns 1 when AND return 0, and vice versa
- NOR returns 1 when OR returns 0, and vice versa

Inputs		Outputs	
a	**b**	$\overline{\mathbf{a \cdot b}}$	$\overline{\mathbf{a+b}}$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

	XOR	**XNOR**
Schematic symbol		
Logic symbol	$a \oplus b$	$\overline{a \oplus b}$
Verilog expression	a^b	~(a^b)

Inputs		Outputs	
a	**b**	$a \oplus b$	$\overline{a \oplus b}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Note that XNOR can also be expressed by the symbol $\odot$.

3.8 Functional Completeness

A set of operations is **functionally-complete (or universal)** if and only if all possible truth tables can be expressed entirely by means of operations from this set.

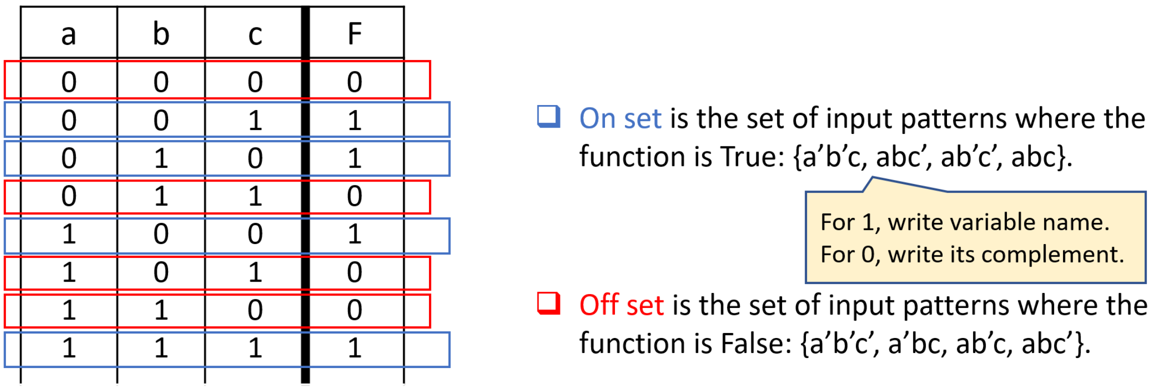
Basic functionally-complete operation sets:

- $\{+, \cdot, '\}$ - by definition

- $\{+,'\}$ - by De Morgan's
- $\{\cdot,'\}$ - by De Morgan's
- $\{\uparrow\}$ - NAND can implement AND, OR, and NOT
- $\{\downarrow\}$ - NOR can implement AND, OR and NOT

Note that XOR and XNOR by themselves are not functionally complete.

3.9 Canonical Forms



a	b	c	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

❑ On set is the set of input patterns where the function is True: {a'b'c, abc', ab'c', abc}.

For 1, write variable name.
For 0, write its complement.

❑ Off set is the set of input patterns where the function is False: {a'b'c', a'bc, ab'c, abc'}.

i Note 7: Definition - Literal

Literal: a single variable or its complement
E.g.: a, a'

i Note 8: Definition - Product Term

Product Term: AND of (more than one) literals
E.g.: $abc, a'bc'$

i Note 9: Definition - Sum Term

Sum Term: OR of (more than one) literals
E.g.: $a+b+c, a+b'+c'$

i Note 10: Definition - Sum of Products (SOP)

Sum of Products (SoP): sum of On Set input patterns, i.e. the OR of minterms (product terms)

E.g.: $F = a'b'c + a'bc' + ab'c' + abc$ is the SOP of the example truth table above (blue rows).

Note 11: Definition - Product of Sums (POS)

Product of Sums (PoS): product of Off Set input patterns, i.e. the AND of maxterms (sum terms)
E.g.: $F = (a+b+c)(a+b'+c')(a'+b+c')(a'+b'+c)$ is the POS of the example truth table above (red rows).

Notice that the POS and SOP can be derived from each other via De Morgan's: $F_{POS} = \bar{F}_{SOP}$. Due to this De Morgan's derivation, notice how essentially the product and sum terms have the variables in complement.

3.10 Minterms & Maxterms

Note 12: Definition - Normal Term

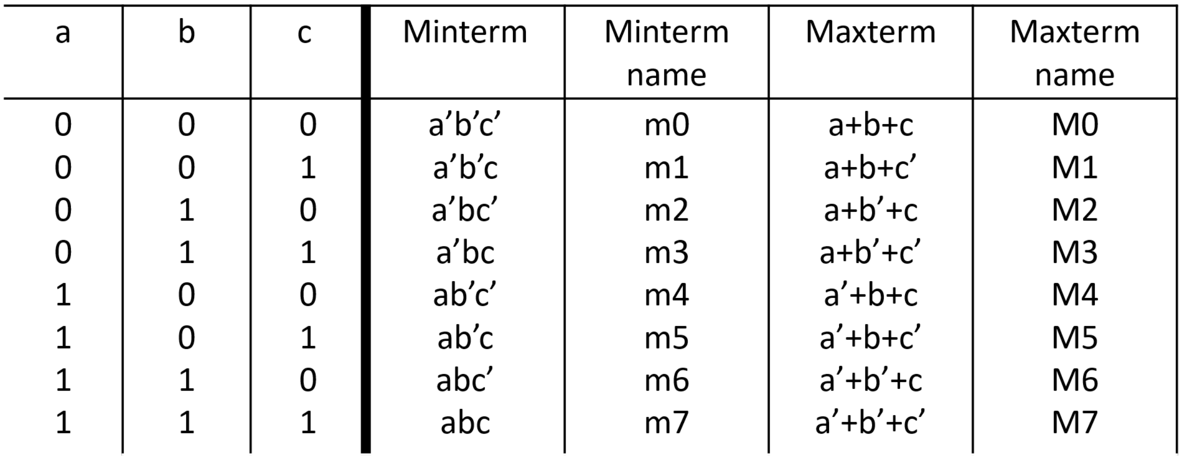
Normal Term: product or sum term in which every variable appears once
E.g.: For function $F(a, b, c, d)$, terms $ab'cd', a + b + c + d'$ are normal terms

Note 13: Definition - Minterm

Minterm: Normal product
E.g.: For function $F(a, b, c)$, $ab'c, a'b'c'$ are minterms.

Note 14: Definition - Maxterm

Maxterm: Normal sum
E.g.: For function $F(a, b, c)$, $(a + b' + c), (a' + b' + c')$ are maxterms.

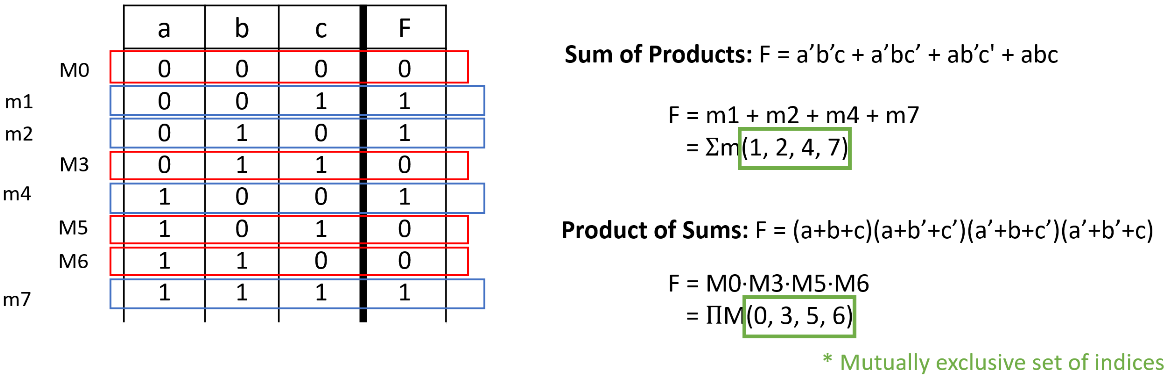


a	b	c	Minterm	Minterm name	Maxterm	Maxterm name
0	0	0	a’b’c’	m0	a+b+c	M0
0	0	1	a’b’c	m1	a+b+c’	M1
0	1	0	a’bc’	m2	a+b’+c	M2
0	1	1	a’bc	m3	a+b’+c’	M3
1	0	0	ab’c’	m4	a’+b+c	M4
1	0	1	ab’c	m5	a’+b+c’	M5
1	1	0	abc’	m6	a’+b’+c	M6
1	1	1	abc	m7	a’+b’+c’	M7

Each input combination has a corresponding minterm and maxterm.

3.11 Canonical Form

A **canonical form** is a representation such that every object has a unique representation. Do note that canonical form ≠ minimal form.



	a	b	c	F
M0	0	0	0	0
m1	0	0	1	1
m2	0	1	0	1
M3	0	1	1	0
m4	1	0	0	1
M5	1	0	1	0
M6	1	1	0	0
m7	1	1	1	1

Sum of Products: F = a’b’c + a’bc’ + ab’c' + abc

F = m1 + m2 + m4 + m7
= Σm(1, 2, 4, 7)

Product of Sums: F = (a+b+c)(a+b’+c’)(a’+b+c’)(a’+b’+c)

F = M0·M3·M5·M6
= ΠM(0, 3, 5, 6)

* Mutually exclusive set of indices

Notice how the SOP and POS have **mutually exclusive** set of indices!

3.12 Shannon’s Expansion Theorem

A Boolean function may be expanded with respect to any of its variables.

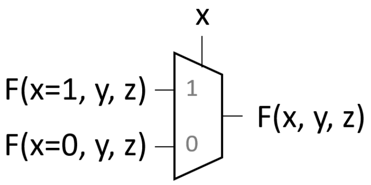
Note 15: Theorem - Shannon's Expansion Theorem

Shannon's Expansion Theorem, a.k.a. Boole's Expansion Theorem, Shannon Decomposition:

$$F(X_1, X_2, \dots, X_n) = X_1 \cdot F(1, X_2, \dots, X_n) + X_1' \cdot F(0, X_2, \dots, X_n)$$

This can help us with MUX-based logic function implementations:

$$F(x, y, z) = x\cdot F(x=1, y, z) + x'\cdot F(x=0, y, z) \Rightarrow$$

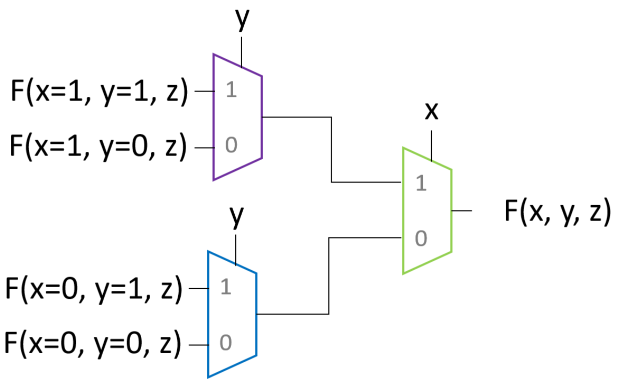


$$F(x, y, z)= x\cdot(y\cdot F(x=1, y=1, z) + y'\cdot F(x=1, y=0, z)) + x'\cdot(y\cdot F(x=0, y=1, z) + y'\cdot F(x=0, y=0, z))$$

$$\Downarrow$$

?

$$F(x, y, z)= \boxed{x\cdot(\boxed{y\cdot F(x=1, y=1, z) + y'\cdot F(x=1, y=0, z)}) + x'\cdot(\boxed{y\cdot F(x=0, y=1, z) + y'\cdot F(x=0, y=0, z)})}$$



4 Combinational Logic

Note 16: Definition - Combinational Logic

Combinational Logic: output is a pure function of the present input only.

4.1

4.2 Transistors

Logic gates are built from switches that consist of three parts: input, output, and control.

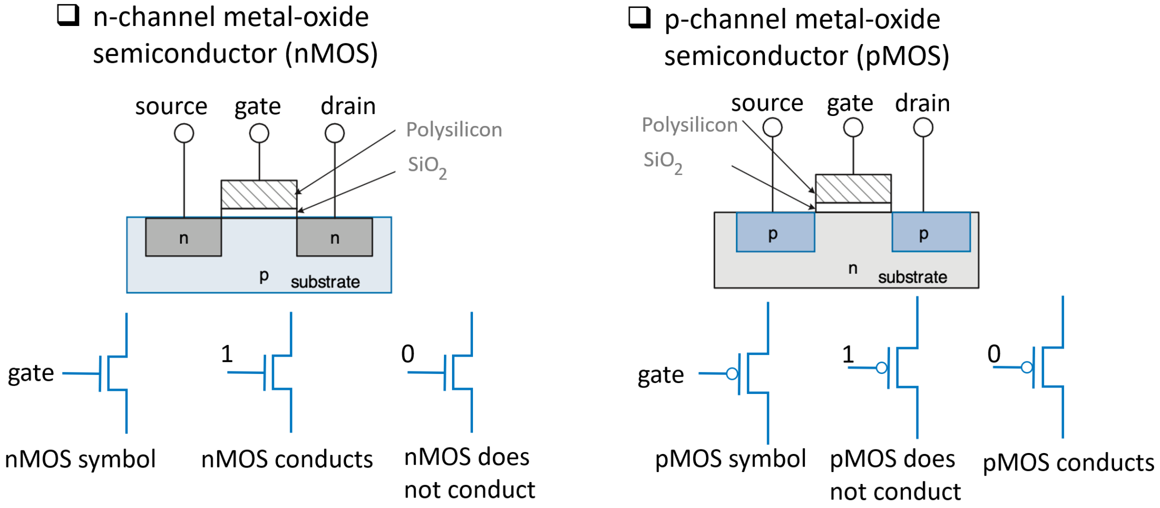


Figure 4.1: Operation of NMOS and PMOS transistors. Source: EECS270-W24

Complementary Metal-Oxide Semiconductor (CMOS) is the predominant technology used as today's switches. CMOS switches consist of NMOS and PMOS transistors. The gate terminal serves as the control, and carriers (electrons or holes) flow from the source to the drain. For more detailed notes, please see the CMOS & VLSI notebook.

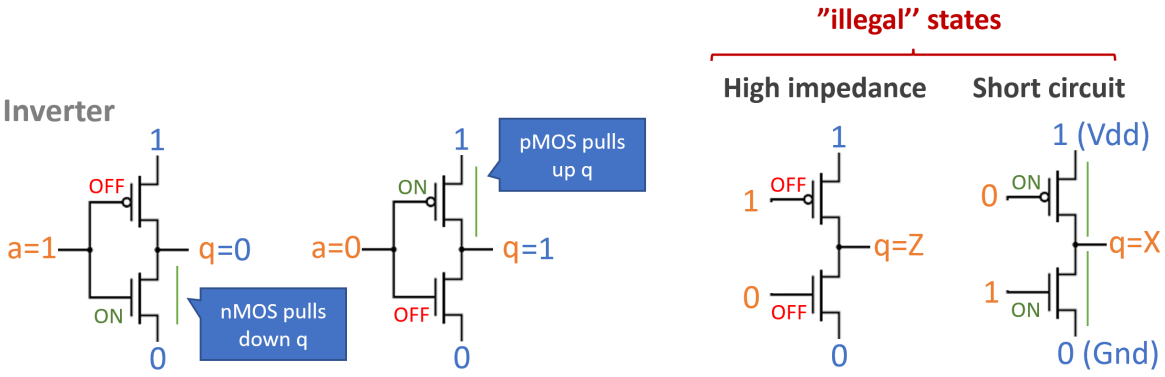


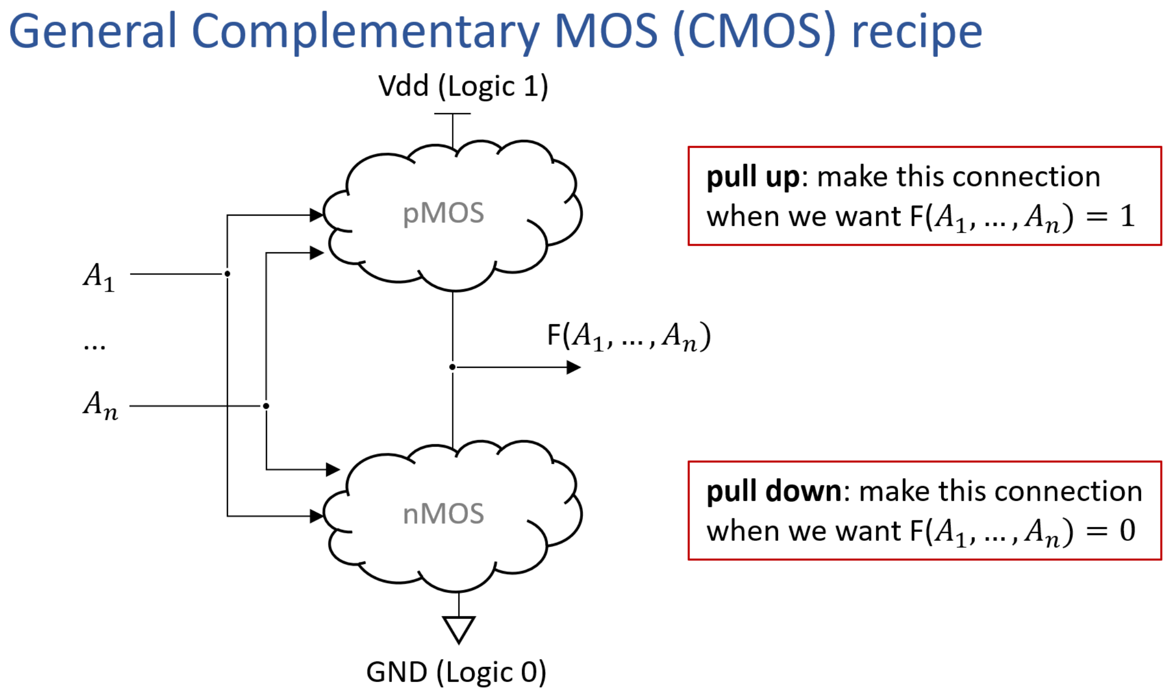
Figure 4.2: Pull-up and Pull-Down of Transistors, and illegal states. Source: EECS270-W24

Why does CMOS use both NMOS and PMOS? - Transistors are not ideal switches - NMOS is good at propagating 0s (pull-down) - PMOS is good at propagating 1s (pull-up)

Asides from propagating 1s and 0s, transistors could also be in illegal states:

The symbol X indicates that the circuit node has an unknown or illegal value. This commonly happens if it is being driven to both 0 and 1 at the same time.

The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be floating, high impedance, or high Z. A floating node does not always mean there is an error in the circuit, as long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.



General Complementary MOS (CMOS) recipe

pull up: make this connection when we want $F(A_1, \dots, A_n) = 1$

pull down: make this connection when we want $F(A_1, \dots, A_n) = 0$

Figure 4.3: CMOS uses pull-up and pull-down as "complement". Source: EECS270-W24

4.3 Transistor Scaling

Moore's law: The number of transistors in a dense integrated circuit (IC) doubles about every 18-24 months.

Dennard scaling: As the dimensions of a device go down, so does power consumption.

5 Timing

5.1

6 Sequential Logic

6.1

7 Finite State Machines

7.1

8 Binary Arithmetic

8.1

9 Memories

10 Summary

In summary...

References