alex-levesque.com

ELEC 271 - Lecture Notes

Digital Systems

Prof. Kleber Cabral • Fall 2025 • Queen's University

Contents

1 Preface

Grading Scheme:

Midterm: 30%

Labs (4 total): 30%

Final: 40%

Textbook: Fundamentals of Digital Logic with Verilog Design

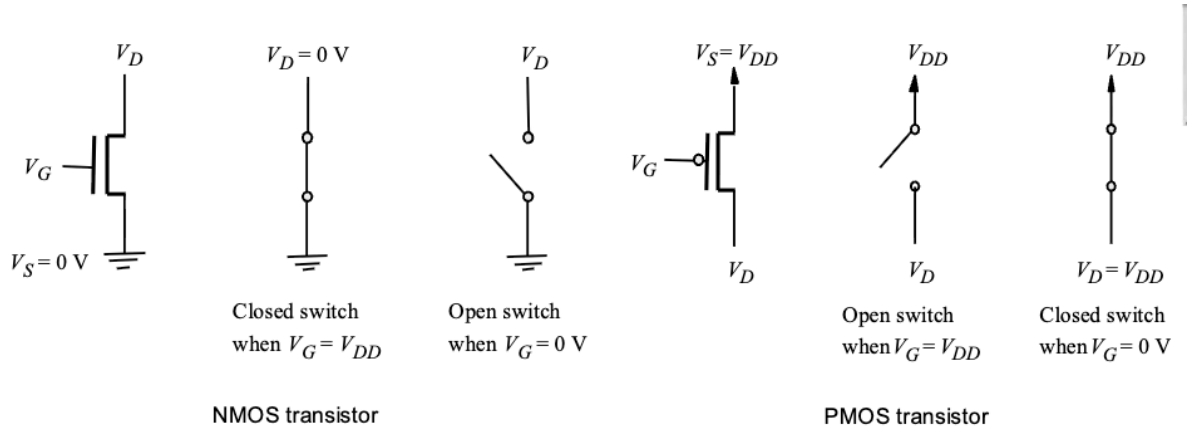
Comments:

- the textbook is fairly useful
- the labs are relatively easy and are a big help towards the final grade
- the midterm for this course is typically easy, however the final is not
- knowing how to implement what we learn in class in VHDL is a nice to have

2 Digital Hardware and Binary Numbers

Digital hardware, Moore's law, technology trends, chip types, layers of abstraction, design process, design flow for logic circuits, fixed point numbers and positional number representation

Analog signal is not discrete values, ex: radio signal **Digital** signal relies on discrete values, such as 1 or 0, on or off. A switch is basic element in implementing a digital system



2.1 Transistors

An NMOS (N-channel Metal-Oxide-Semiconductor) is a type of transistor used as a switching element in digital systems

When the gate voltage V_G is high, or equal to the supply voltage V_{DD}, the NMOS acts like a closed switch. When V_G is low (0V), the NMOS acts as an open switch

A PMOS is the opposite. When the gate voltage is high, the PMOS acts like an open switch. When the gate voltage is low, the PMOS acts as a closed switch.

Moore's Law and Chips

This law states that the number of transistors on a chip is doubling every 18 months. Some may predict the end of Moore's Law, and yet it keeps going.

2.2 Binary Numbers

The general form for determining the decimal value of a number in base k is given by: $Value = \Sigma_{i=-m}^{n-1} b_i k^i$

n is the highest power of the base m is the number of fractional digits after the decimal point

Conversion:

Each digit in a binary number represents a power of 2, depending on its position

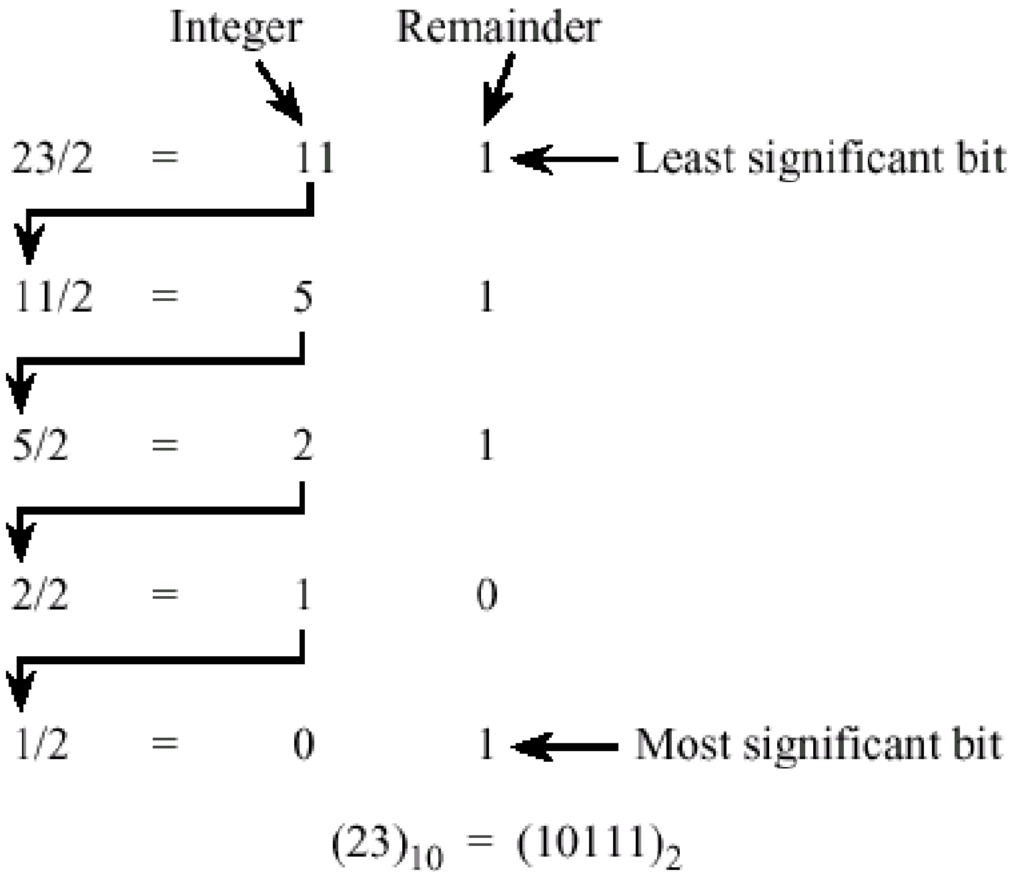
To convert a binary number to decimal, multiply each digit b_i by 2^i, where i is the position

Example: $(1010.01)_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0\ 8 + 0 + 2 + 0 + 0 + 0.25 = (10.25)_2$

Converting decimal to binary, we divide the number by 2 as many times as possible

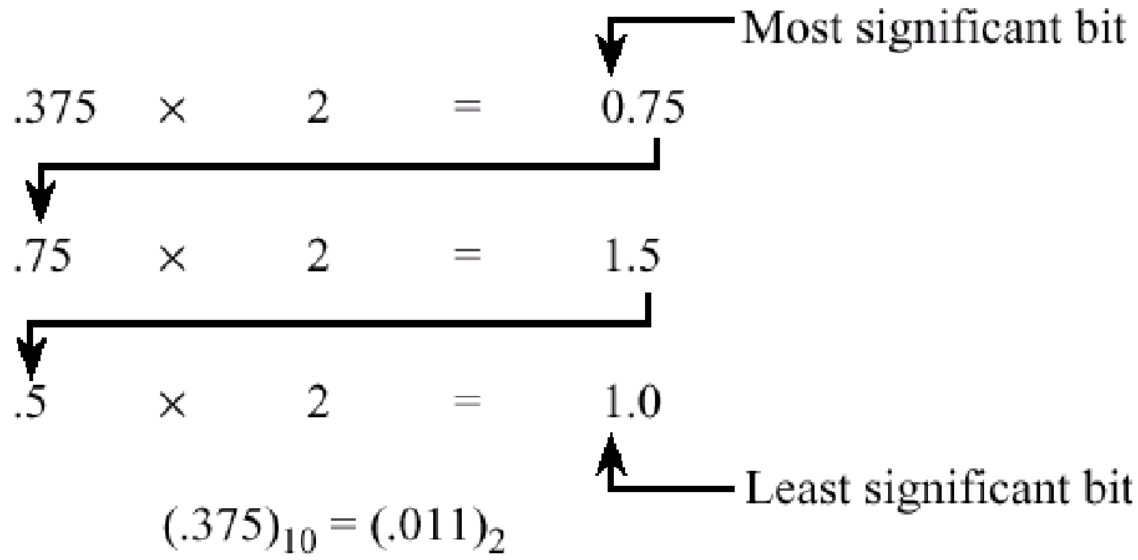
Example: Convert 23.375 to base 2:

Start with the integer part:



$$(23)_{10} = (10111)_2$$

Then do the decimal part:



$$(.375)_{10} = (.011)_2$$

Beyond base 2

The hexadecimal value 0xFF converts to: $0xFF = (15 * 16^1) + (15 * 16^0) = 240 + 15 = 255$

The $0x$ in front of the number indicates that the number is written in hexadecimal notation

3 Boolean Algebra and Minterm/Maxterm

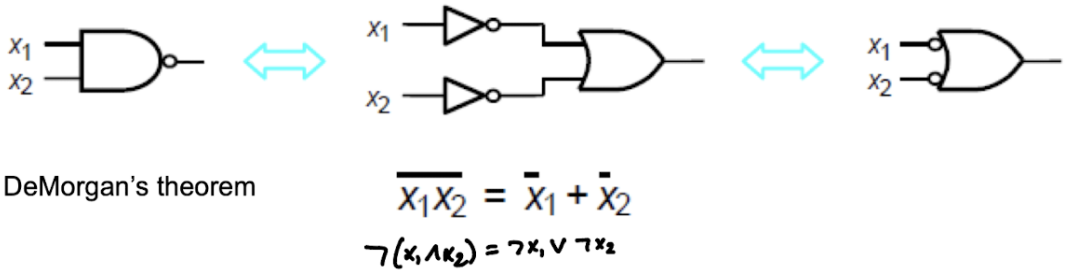
This is very similar to MTHE 217:

$$1 + 1 = 1$$
$$x + x = x$$
$$x + \bar{x} = 1$$
$$\bar{\bar{x}} = x$$

Duality: a dual of a Boolean expression is obtained by replacing all "+" operators with "." operators

Example: De Morgan's theorem states that $(\bar{x + y}) = \bar{x} * \bar{y}$

We can also showcase [[DeMorgan]]'s theorem as follows:



Use DeMorgan's theorem

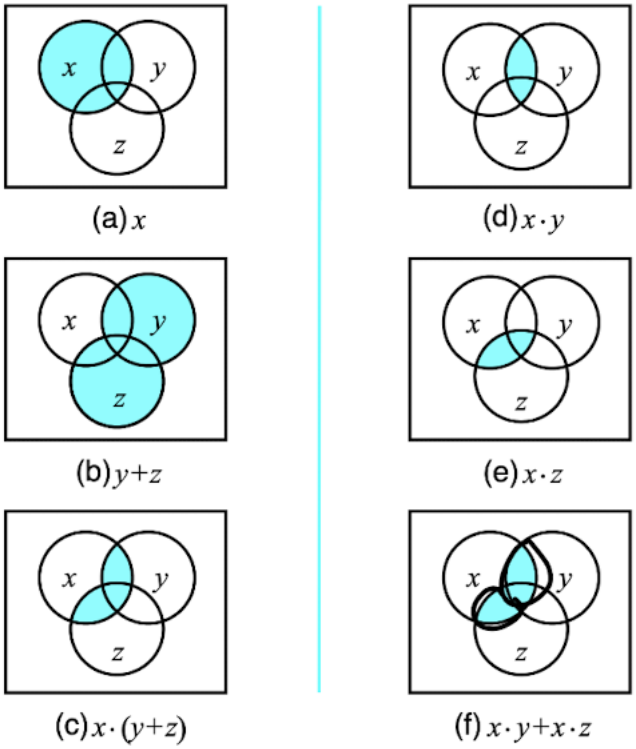
$$\overline{X_1 X_2} = \bar{X}_1 + \bar{X}_2$$

$$\neg(x_1 \wedge x_2) = \neg x_1 \vee \neg x_2$$

An important result when simplifying a logic equation is $x * \bar{x} = 0$

3.1 Venn Diagrams

We can intuitively how two expressions may be equivalent with Venn diagrams



3.2 Synthesis using operators

Minterm (m): a minterm is a product (AND) term in which each of the n variables appears once (ANDed product of literals)

Maxterm (M): a maxterm is the complement of minterm, a sum (OR) term in which each of the n variables appears once (ORed sum of literals)

$\bar{m_i} = M_i$

In a three literal circuit, examples: $m_0 = \bar{x_1} * \bar{x_2} * \bar{x_3}$ $M_4 = \bar{x_1} + x_2 + x_3$ $m_6 = x_1 * x_2 * \bar{x_3}$ $M_6 = \bar{x_1} + \bar{x_2} + x_3$

If each product term is a minterm, the expression is called a canonical sum-of-products (SOP)

If each sum term is a maxterm, the expression is called canonical product-of-sums (POS)

We can simplify a canonical SOP to a minimal-cost realization by reducing the number of gates and reducing number of input variables

POS example:

$$f(x_1, x_2) = m_0 + m_1 + m_3$$
$$f(\bar{x_1}, x_2) = m_2 = x_1\bar{x_2}$$
$$f(\bar{\bar{x_1}}, x_2) = f = (x_1 \bar{x_2}) = \bar{x_1} + x_2 = M_2$$

Simplification Tricks

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

If every case of two propositions is covered, let it = 1 Example: $(x_1x_2 + \bar{x_1}x_2 + x_1\bar{x_2} + \bar{x_1}\bar{x_2}) = 1$

4 Multiplexers and VHDL

4.1 Multiplexer Circuit

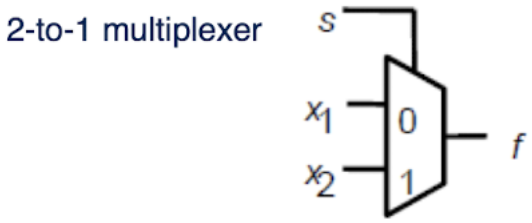
This is a circuit that chooses data from exactly one of the number of possible input sources

If we are given two sources of data x_1, x_2, an output f, and a select input control signal s, the result of f depends on s

Example:

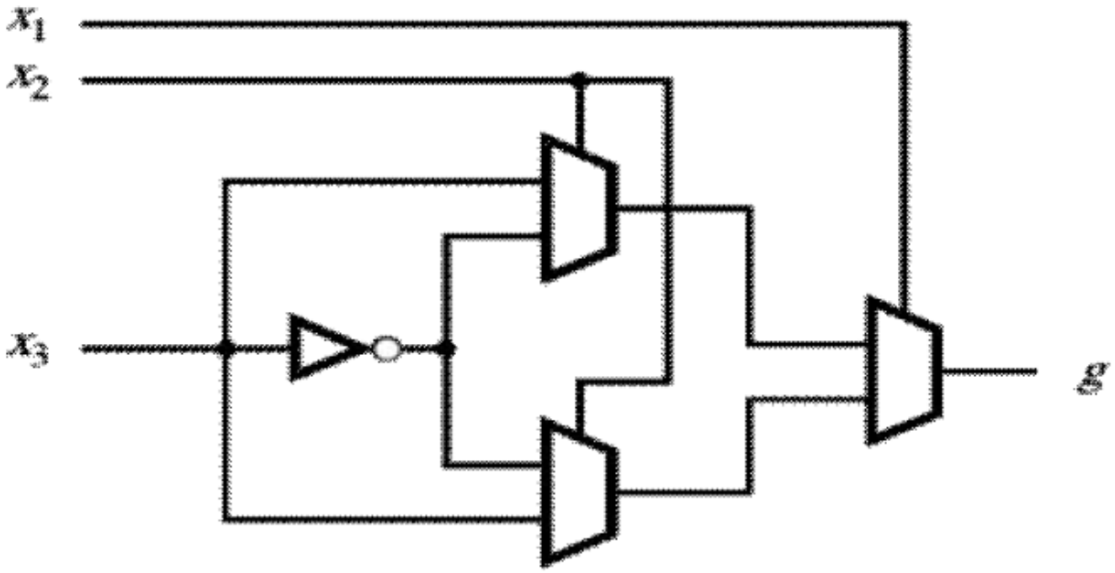
$f = (I_0 * \bar{S}) + (I_1 * S)$, or formally, $MUX(S; A, B) = \overline{S}A + SB$

If $S = 0$, $f = I_0$ If $S = 1$, $f = I$



The numbers written inside the block represent which input is connected to the output depending on the value of s

Example:



$MUX_1 = (x_2; x_3, \overline{x_3}) = \overline{x_2}x_3 + x_2\overline{x_3} = x_2 \oplus x_3$ $MUX_2 = (x_2; \overline{x_3}, x_3) = \overline{x_2 x_3} + x_2 x_3 = x_2 \odot x_3$
$MUX_3 = (x_1; x_2 \oplus x_3, x_2 \odot x_3) = x_1 \oplus x_2 \oplus x_3$

4.2 Introduction to VHDL

Entity: Defines the interface of a circuit block (its input and output ports)

Architecture: Describes the internal implementation or behaviour of the entity

Signal assignment: Defines how outputs relate to inputs, often using logic equations

```
entity example-2 IS
    port (x1, x2, x3, x4: in bit;
          f, g: out bit);
end entity example-2;

architecture LogicFunc OF example-2 IS
begin
    f <= (x1 AND x3) OR (x2 AND x4);
    g <= (x1 OR NOT x3) AND (NOT x2 OR x4);
end LogicFunc;
```

Example-2 is a circuit block with four input signals and two output signals

architecture: describes how the outputs f and g are logically derived from the inputs. The assignments to f and g are called concurrent signal assignments (they happen simultaneously in hardware)

5 Logic Circuits

A **logic function** $L(x)$ is a collection of signals $x_1, \ldots, x_n$

5.1 Variables and Functions

The logical AND ($\wedge$) function serves for a series connection, where the function $L(x_1, x_2) = x_1 * x_2$

The logical OR ($\vee$) function serves for a parallel connection, where the function $L(x_1, x_2) = x_1 + x_2$

The logical XOR ($\oplus$) outputs 1 if exactly one of the inputs is 1

XOR: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$

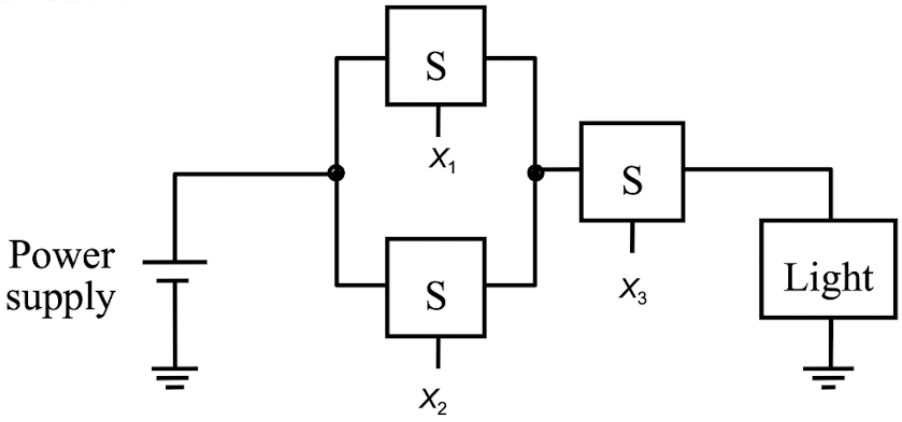
The logical XNOR ($\odot$) outputs 1 if both inputs are the same, this is the complement of XOR

XNOR: $A \odot B = (A \wedge B) \vee (\neg A \wedge \neg B)$

XOR/XNOR identity: $\overline{x_1}(x_2 \odot x_3) + x_1(x_2 \oplus x_3) = x_1 \oplus x_2 \oplus x_3$

Example:

- A series-parallel connection:

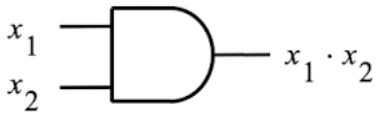


$L(x_1, x_2, x_3) = (x_1 + x_2) * x_3$

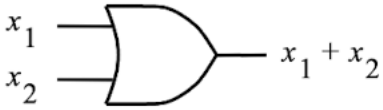
5.2 Logic Gates and Circuits

We can use several gates to represent connectors

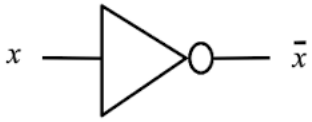
AND gates

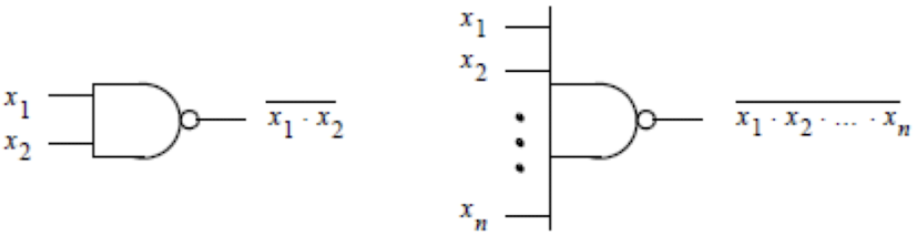


OR gates



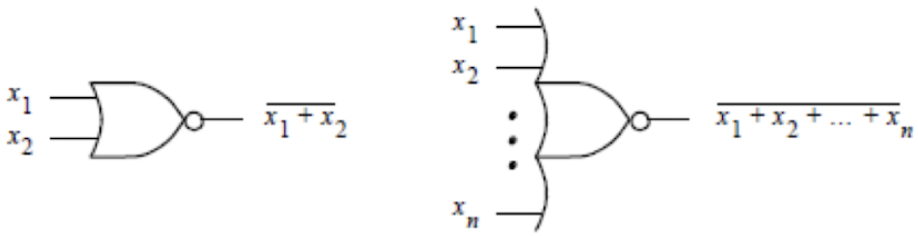
NOT gate





Truth table?

NAND gates



Truth table?

NOR gates

So, we can draw a logic network into an equivalent logic circuit using these symbols.

6 Transistor Switches

6.1 Logic Values and Voltage Levels

Digital systems use binary values: 0 (low voltage, ~0V) and 1 (high voltage, e.g., 5 V)

Positive logic: 0 = low, 1 = high. This is the usual system.

The **noise margin** is the amount of "buffer" a logic signal has against electric noise before it risks being misinterpreted as the wrong logic level.

- High-state margin: $NM_H = V_{IL_{max}} - V_{OL_{max}}$

NM_H tells us how much noise a high-level signal can tolerate before being misread as low. It is the difference between the "maximum voltage guaranteed to be recognized as logic 0" and "maximum output voltage when driving logic 0"

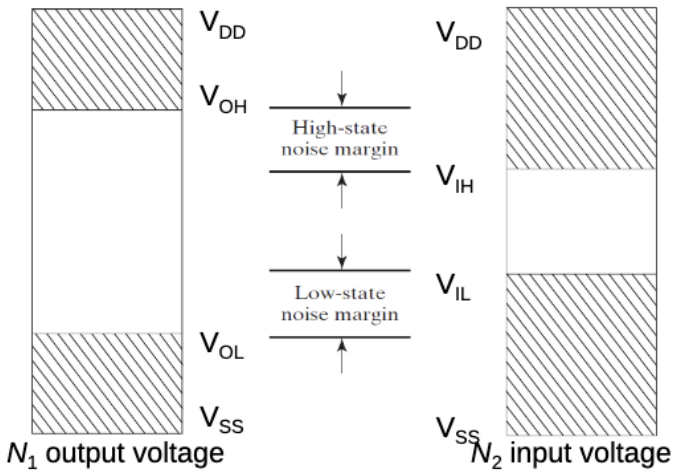
- Low-state margin: $NM_H = V_{OH_{min}} - V_{IH_{min}}$

NM_L tells us how much noise a low-level signal can tolerate before being misread as high. It is the difference between the "minimum output voltage when driving logic 1" and "minimum voltage guaranteed to be recognized as logic 1"

Below: V_{SS} is the lowest voltage V_{DD} is the highest voltage

High state noise margin:
$NM_H = V_{OH(min)} - V_{IH\ (min)}$

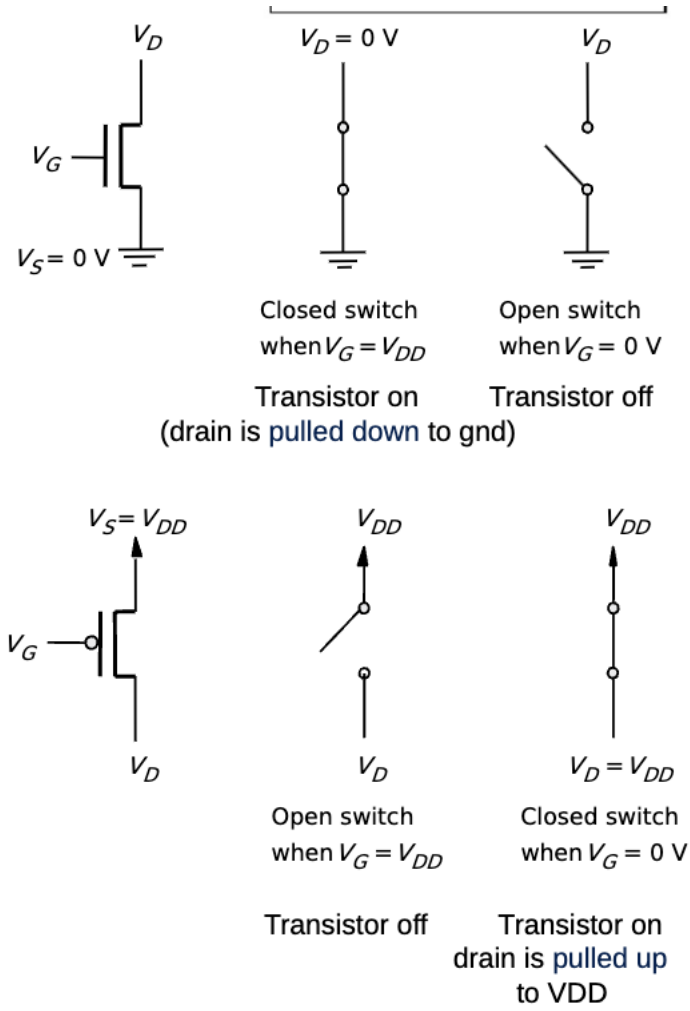
Low state noise margin
$NM_L = V_{IL(max)} - V_{OL(max)}$



6.2 Transistor Switches

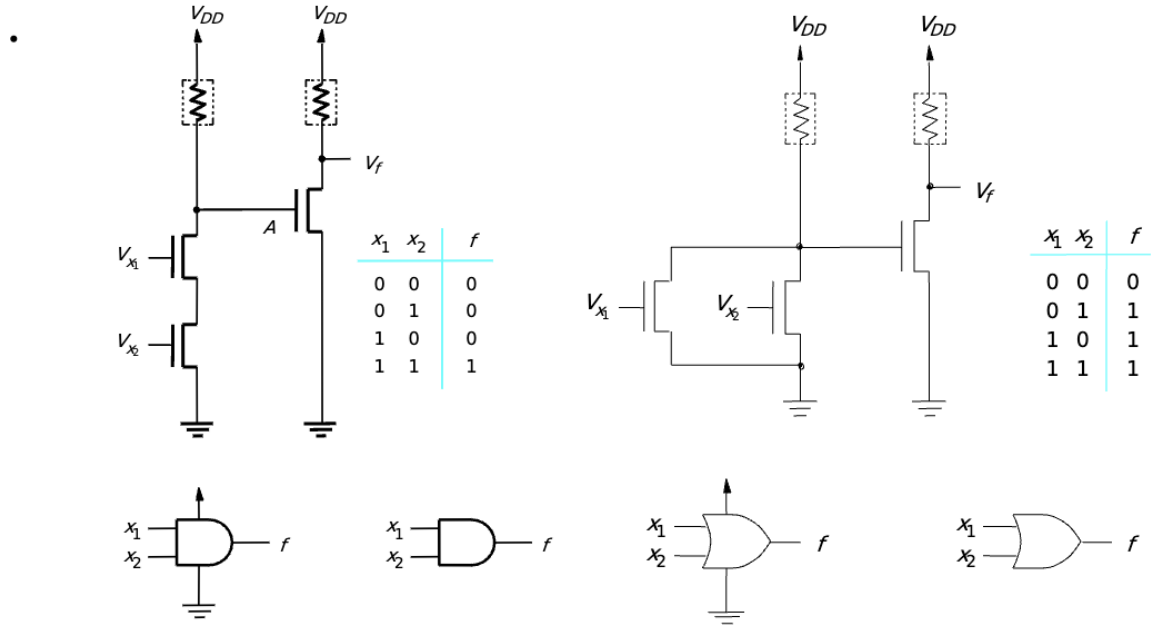
Logic gates are built from metal oxide semiconductor field-effect transistor (MOSFET), with a N-channel (NMOS) and P-channel (PMOS)

V_S is the source voltage, where charge carriers enter the channel V_G is the gate voltage, applies a voltage to control whether current flows V_D is the drain voltage, where charge carriers leave the channel V_{DD} is the positive supply voltage



AND and OR gate using NMOS

- An AND and an OR gate using NMOS technology:
-



7 CMOS Logic Gates

CMOS: Complementary MOS (both NMOS and PMOS)

CMOS is popular because no power is dissipated under steady state conditions (no current flows when the input is either low or high)

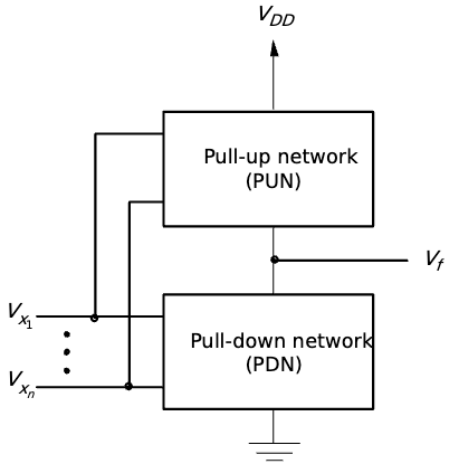
In the image below, the **pull-up network (PUN)** is made of PMOS transistors and connects the output node to V_{DD} when the logic function requires a `1`

The **pull-down network (PDN)** is made of NMOS transistors and connects the output node to Ground (0V) when the logic function requires a `0`

General formula:

PUN = PMOS transistors -> Vdd PDN = NMOS transistors -> GND

- Pull-up device is replaced with a pull-up network (PUN) using PMOS transistors.
- PDN and PUN networks have complementary functions, being dual of each other, one having transistors in series, the other in parallel, and vice versa.



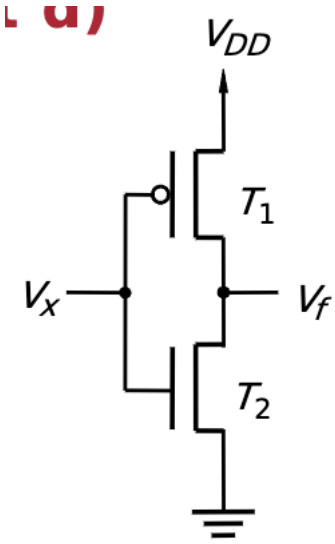
Structure of an CMOS circuit

Notice that when two signals are stacked, they are OR'ed, and when they are in series, they are AND'ed. Furthermore, the complement of the NMOS logic function is the PMOS logic function, and vice-versa.

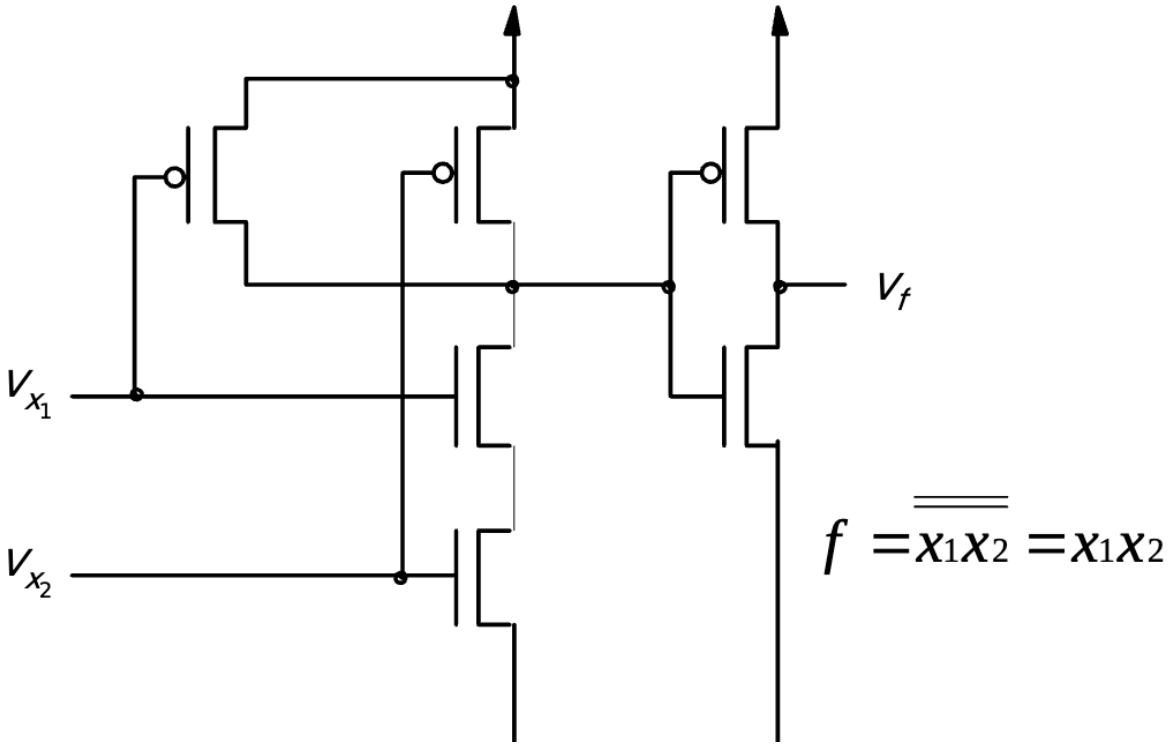
CMOS Examples

A **NOT gate** reverses the input logic state.

When the input $V_x = 0$, T_1 is ON and connects V_{DD} to V_f When the input $V_x = 1, T_2$ is OFF and disconnects



An **AND** gate is mote complex than one may think. If you try to directly wire PMOS/N-MOS to match AND truth table, you get something messy. It is better to treat AND as *NAND + NOT*

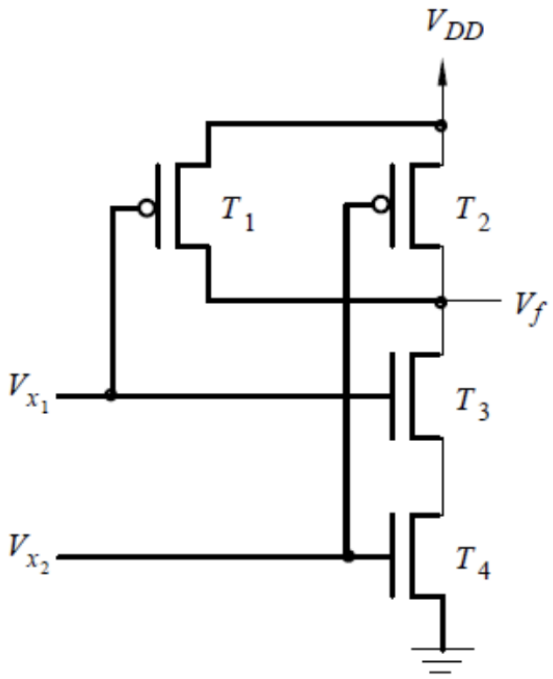


Similarly, an OR can be obtained from a NOR followed by a NOT

7.1 NAND/NOR Implementation

A **NAND** gate outputs the opposite of an AND gate, $f = \overline{x_1 x_2}$

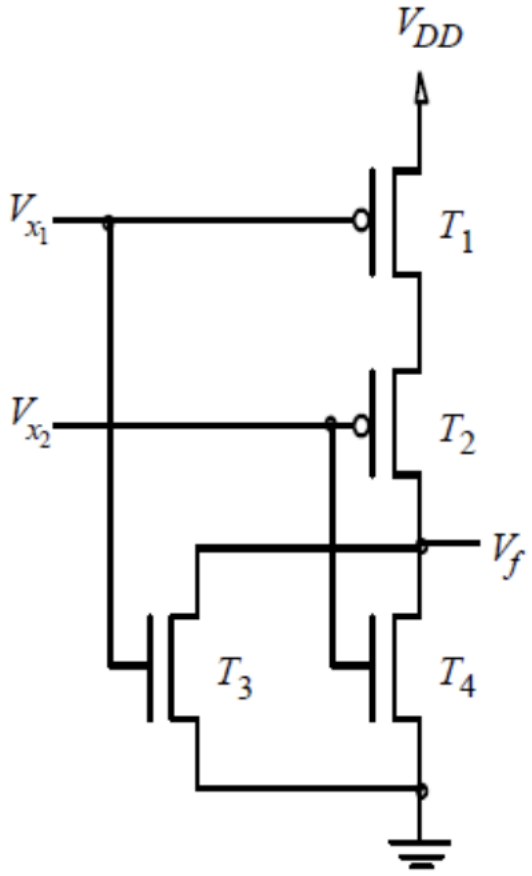
Here, when x_1 and x_2 are 0, the PUN is on and **the signal is being pulled up to** V_{DD}, not the ground



$$f = \overline{x_1 x_2}$$

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

A **NOR** gate outputs the opposite of an OR gate, $f = \overline{x_1 + x_2}$, where the PMOS transistors are placed in series and the NMOS transistors are placed in parallel



$$f = \overline{x_1 + x_2}$$

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

Circuit Truth table and transistor states

7.2 NAND and NOR Conversion

NAND is the negation of AND NOR is the negation of OR

A NAND or NOR conversion takes a boolean expression and implements it using NAND or NOR gates directly. Sometimes this leaves some of the original AND/OR structure embedded in a NAND or NOR form.

A NAND/NAND conversion is a specific procedure:

1. Convert the expression to all-NAND form of the SOP, using double negation if needed
2. Make every AND and OR replaced by NAND combinations

NAND

- $\overline{a} + \overline{b} = a \text{ NAND } b$
- $\overline{ab} = a \text{ NAND } b$
- $a \cdot b = \overline{a \text{ NAND } b} = (a \text{ NAND } b) \text{ NAND } (a \text{ NAND } b)$
- $a + b = \overline{\overline{a}\overline{b}} = \overline{a} \text{ NAND } \overline{b} = (a \text{ NAND } a) \text{ NAND } (b \text{ NAND } b)$

A NOR/NOR conversion is similar:

1. Convert the expression to all-NOR form of the SOP, using De Morgan's laws, or using double negation if needed

NOR

- $\overline{a} \cdot \overline{b} = a \text{ NOR } b$
- $\overline{a + b} = a \text{ NOR } b$
- $a + b = \overline{a \text{ NOR } b} = (a \text{ NOR } b) \text{ NOR } (a \text{ NOR } b)$
- $a \cdot b = \overline{\overline{a} + \overline{b}} = \overline{a} \text{ NOR } \overline{b} = (a \text{ NOR } a) \text{ NOR } b(\text{ NOR } b)$

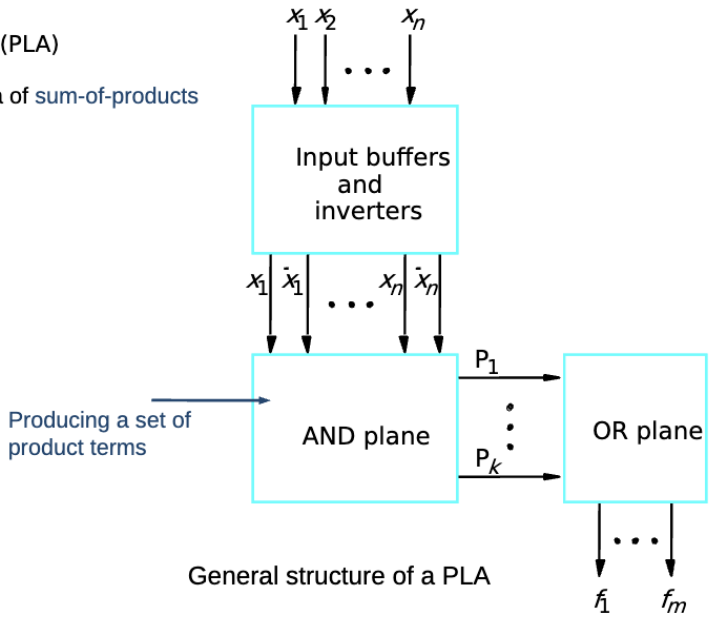
8 Programmable Logic Devices

Programmable Logic Devices (PLDs) is a general-purpose chip for implementing logic circuits, where the input is logic variables and the output is logic functions

A **Programmable Logic Array** (PLA) is a type of logic device with an AND plane followed by an OR plane

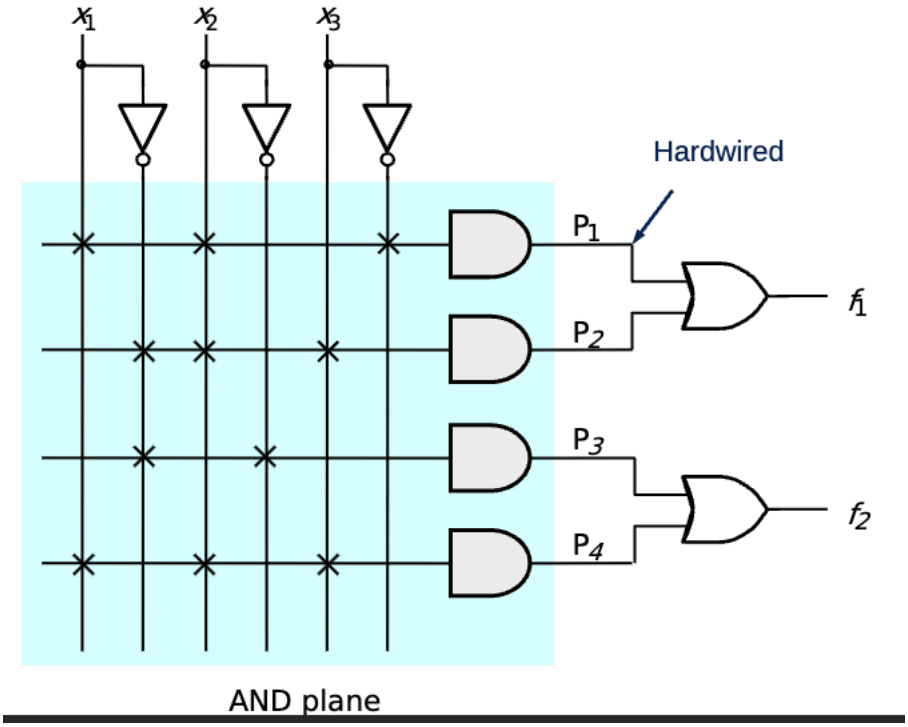
Inputs can be fed into the AND plane to form any desired set of product terms, and then fed into the OR place, where they can be combined to form SOP expressions for the outputs

- Programmable Logic Array (PLA)
- Based on the idea of sum-of-products



General structure of a PLA

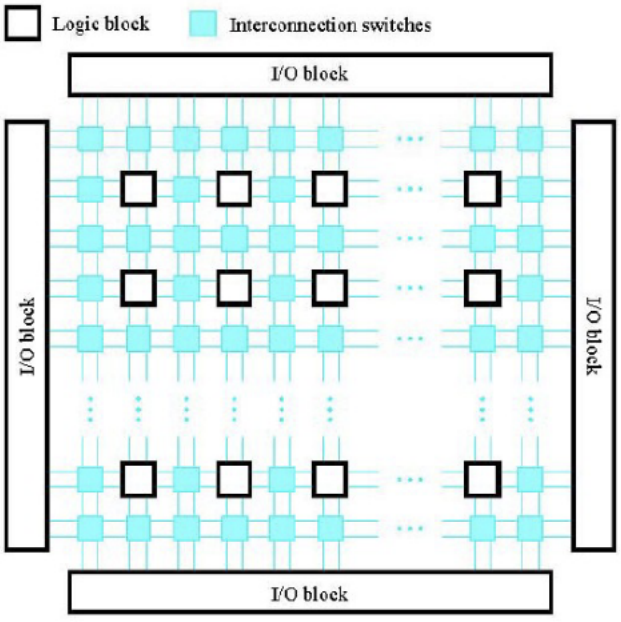
A **Programmable Array Logic** (PAL): only programmable AND array, for cost reason and better performance, but less flexible



A **Field-Programmable-Gate-Array** (FPGA) is for larger logic circuits, and do not contain AND or OR planes

FPGA chips contain three main type of resources: logic blocks, I/O blocks, and interconnection wires and switches

- General structure of an FPGA



Each logic block has a small number of inputs and outputs, with lookup tables (LUT) being the most common implementation. Commercial FPGAs typically use LUTs with up to five inputs.

8.1 Lookup Tables (LUTs)

A lookup table is a memory-based structure used in FPGAs to implement any Boolean function of a few inputs. A LUT can be programmed by the user to implement any logic function of its input. Thus, if a LUT has three inputs, then it can implement any logic function of these three inputs

It has n inputs, stores 2^n entries, and outputs 1 output bit, which is looked up from memory

Example: $f = x_1x_2x_4 + x_2x_3\overline{x_4} + \overline{x_1x_2x_3}$. Show how this function can be implemented in an FPGA that has three-input LUTs

We can use four 3-input LUTs, three of these for the three-input AND operations, and the final LUT implements the OR operation. A 3-input LUT stores $2^3 = 9$ bits

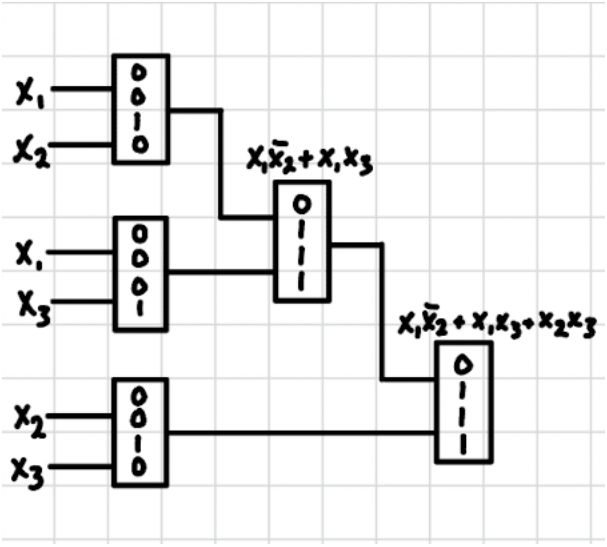
LUT Circuit Example: Consider $f = x_1\overline{x_2} + x_1x_3 + x_2\overline{x_3}$. Show a circuit using 5 two-input LUTs to implement this expression.

x_1	x_2	$f_1 = x_1\overline{x_2}$	x_1	x_3	$f_2 = x_1x_3$	x_2	x_3	$f_3 = x_2\overline{x_3}$
0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	1	0
1	0	1	1	0	0	1	0	1
1	1	0	1	1	1	1	1	0

Even though we calculated f_1 as 0 0 1 0, we do not use those values as the input later (i.e. when we do $f_4 = f_1 + f_2$).

$f_1, x_1 = 1$	$f_2, x_1 = 1$	$f_4 = f_1 + f_2 = x_1\overline{x_2} + x_1x_3$
0	0	0
0	1	1
1	0	1
1	1	1

f_3	f_4	$f_5 = f_3 + f_4 = x_2\overline{x_3} + x_1\overline{x_2} + x_1x_3$
0	0	0
1	0	1
0	1	1
1	1	1



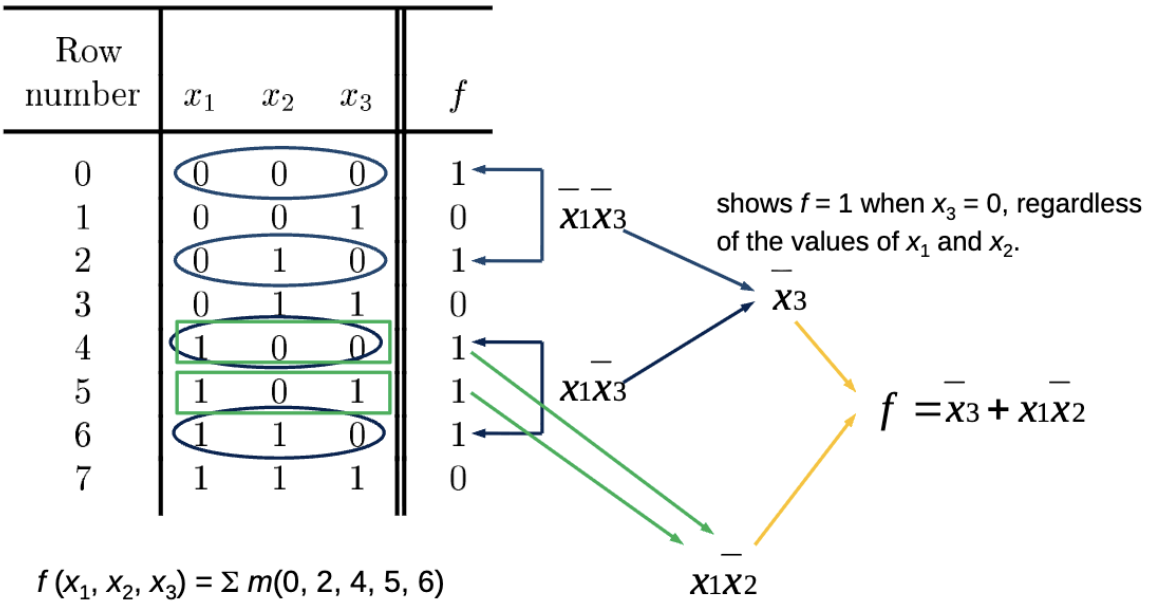
Python analogy example:

```
# XOR function implemented as a LUT
lut = {
    (0, 0): 0,
    (0, 1): 1,
    (1, 0): 1,
    (1, 1): 0
}

print(lut[(1, 0)])  # Output: 1
```

9 Karnaugh Maps (K-maps)

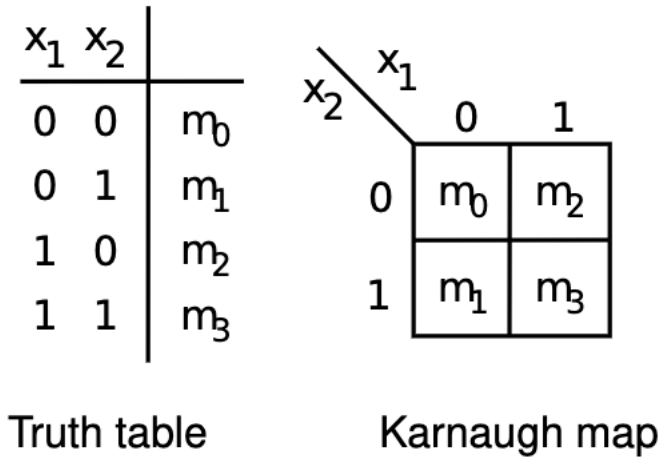
A **Karnaugh Map** provides a systematic way of optimizing logic functions. It is an alternative to the truth table and allows easy discover of groups of *minterms* for which $f = 1$ can be combined into single terms

Key point: Allows to replace **two** *minterms* that differ in the value of one variable with a single product term that does not include that variable



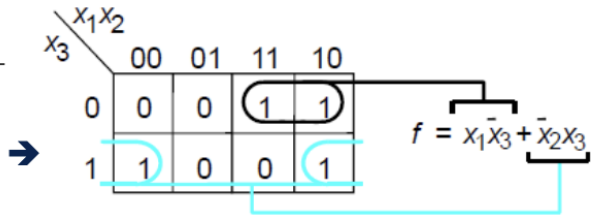
Building a Karnaugh Map:

Two variables:

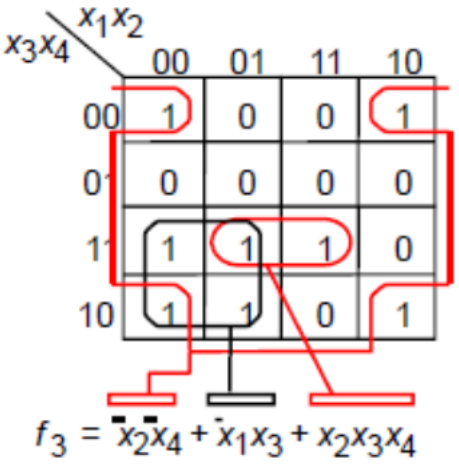


Three variables:

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0



Four variables:



9.1 Solving using a K-map

Strategy: To find as *few as possible* and as *large as possible* groups of 1s that cover all cases where the function has a value of 1

Cost: The number of gates + the number of inputs to the gates

$$f = x_1\bar{x}_2 + \bar{x}_3\bar{x}_4 \qquad \text{Cost} = 9$$

4 literals + 2 AND gates + 1 OR gate + 2 NOT gates

Literal: each appearance of variable (uncomplemented or complemented) in a product term

Implicant: A product term that indicates the input valuations for which a given input is equal to 1 (essentially a group of 1's)

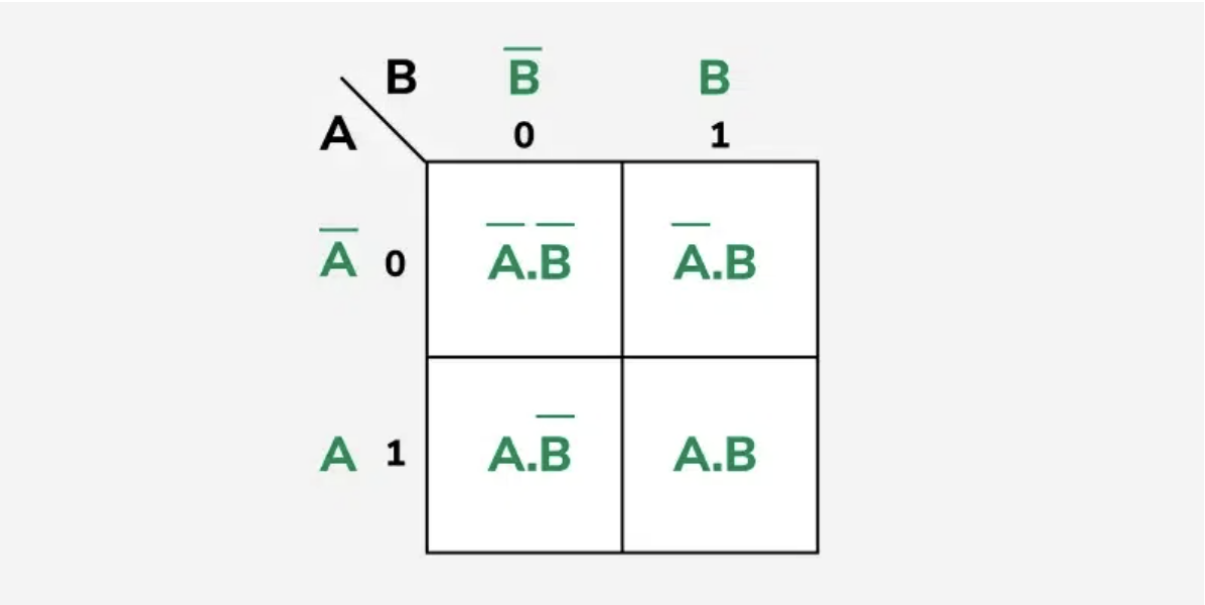
Prime Implicant: A group of 1's that cannot be combined into a larger group of 1's (maximal rectangle of 1's)

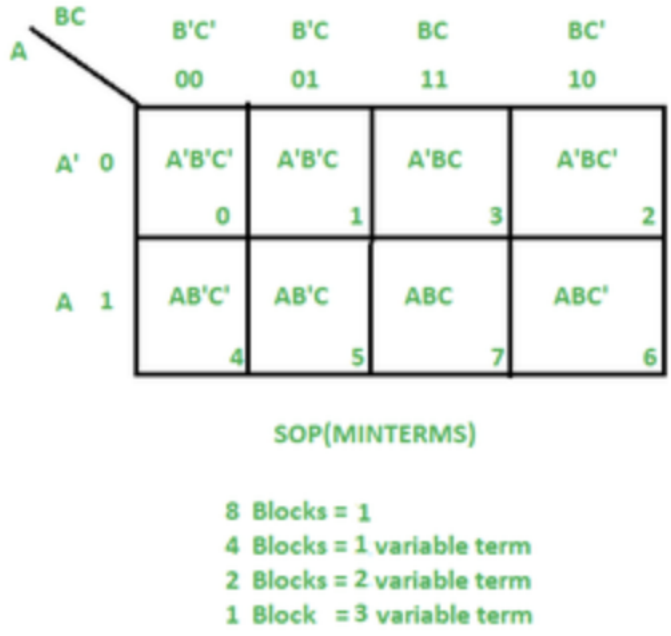
Essential Prime Implicant (EPI): A prime implicant that covers at least one 1 in the K-map that no other prime implicant covers. These are must-have groups in your final simplified function.

Cover: A collection of implicants that account for all valuations for which a given function is equal to.

9.1.1 SOP Solution

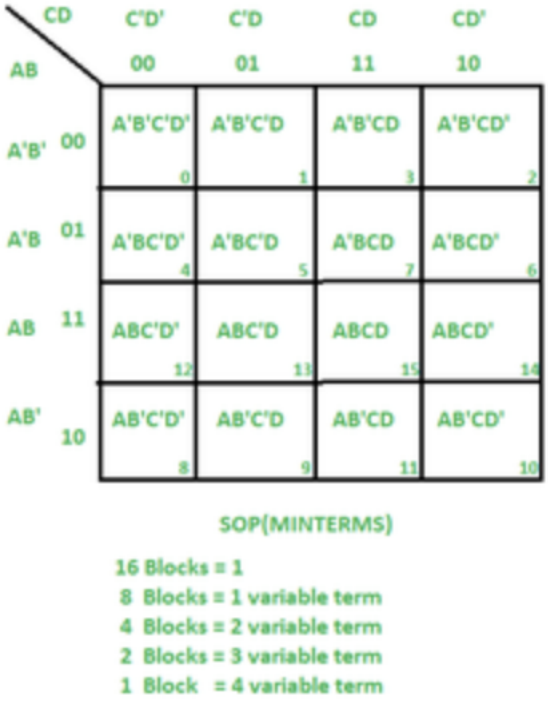
Select K-map according to number of variables





SOP(MINTERMS)

8 Blocks = 1
4 Blocks = 1 variable term
2 Blocks = 2 variable term
1 Block = 3 variable term



Identify minterms or maxterms as given in the problem and assign them to the corresponding box

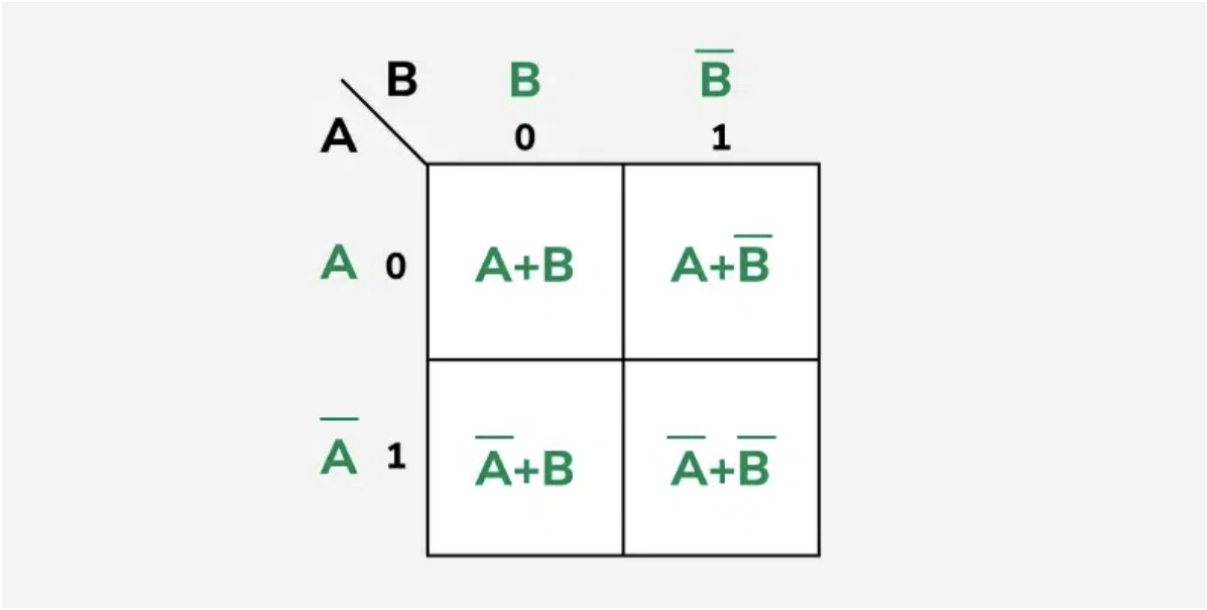
Put 1's in blocks of K-map respective to the minterms (0's elsewhere)

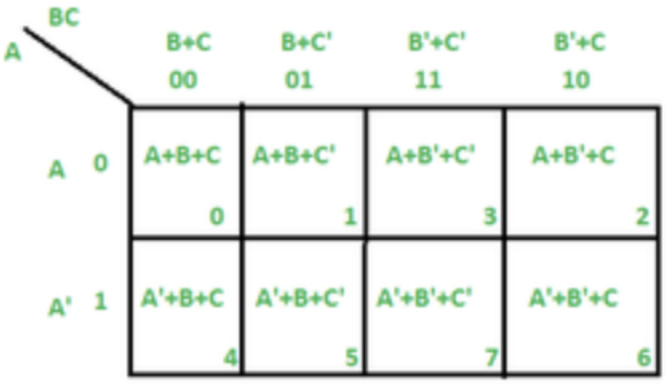
Make rectangular groups of 1 blocks containing total terms in power of two like 2,4,8 and try to cover as many elements as you can in one group

From the groups made in step 5 find the product terms and sum them up for SOP form

9.1.2 POS Solution

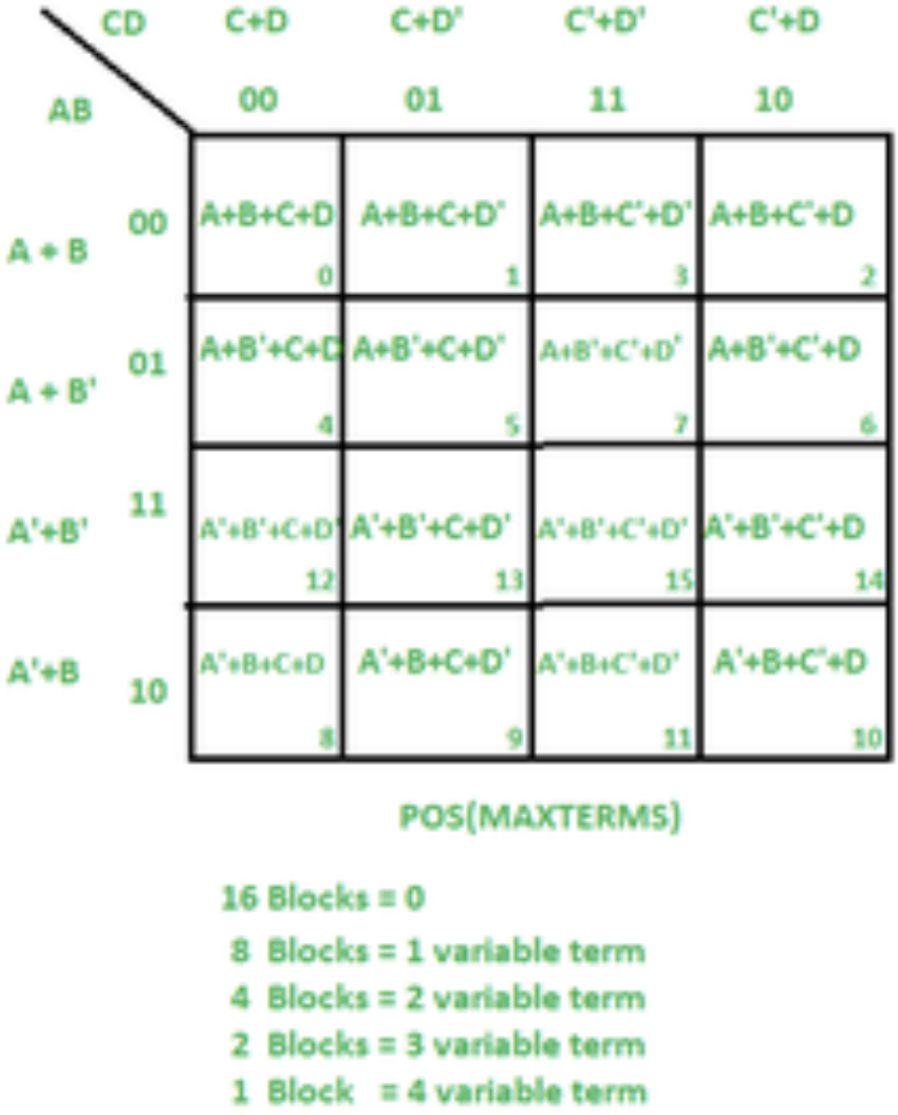
Select K-map according to number of variables





POS (MAXTERMS)

8 Blocks = 0
4 Blocks = 1 variable term
2 Blocks = 2 variable term
1 Block = 3 variable term



Identify minterms or maxterms as given in the problem and assign them to the corresponding box

For POS put 0's in blocks of K-map respective to the maxterms (1's elsewhere)

Make rectangular groups of 0 blocks containing total terms in power of two like 2,4,8 and try to cover as many elements as you can in one group

Take the complement of the groups and sum the literals. Ex: $\overline{C'DB} = (C + D' + B')$

From the groups made in step 5 find the product terms and sum them up for POS form

10 Karnaugh Maps Related Topics

10.1 Incompletely Specified Functions

When an input combination can't ever happen, it's called a don't-care condition A function with one or more don't-care conditions is called incompletely specified

Ex: Allowed inputs for x_1 and x_2 are 00, 01, 10. $(x_1, x_2) = 11$ is a don't care

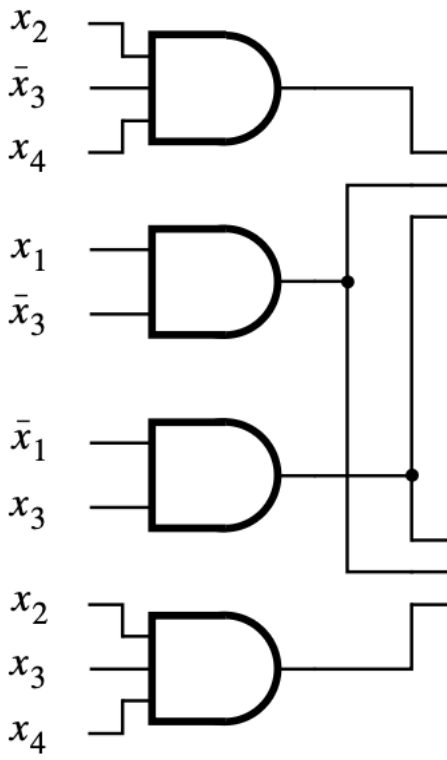
We can treat "d" in a k-map as either 0 or 1, which is useful when trying to find a minimum-cost function

10.2 Multiple-Output Circuits

A circuit can have multiple outputs $f_1, f_2, \ldots, f_n$

Example: $f_1 = x_1\overline{x_3} + \overline{x_1}x_3 + x_2\overline{x_3}x_4$ and $f_2 = x_1\overline{x_3} + \overline{x_1}x_3 + x_2x_3x_4$

The cost of each is 14, and would cost 28 in two circuits. A less-expensive realization is possible if the two circuits are combined into a single circuit with two outputs



We can utilize that the first two product terms are identical to build:

The combined circuit above shows a cost of 22, which is less than the cost of two separate circuits (28)

10.3 The Fan-In Problem

Fan-in refers to the number of inputs a logic gate can handle. For example, a 2-input AND gate has a fan-in of 2.

When the fan-in is too large, several issues arise: - Increased propagation delay: each additional input increases the input capacitance, slowing the gate's switching speed - Higher power consumption: More transistors are required to support additional inputs, raising static and dynamic power usage - Reduced noise margin: Larger fan-in increases signal degradation and reduces voltage margins for reliable logic-level recognition - Design limitations: Many fabrication technologies (e.g., CMOS, FPGA LUTs) have strict maximum fan-in limits, typically 4 to 6 inputs

To solve this, you can split the inputs over multiple gates, called gate decomposition. If not, you can insert buffer circuits (repeaters or drivers) between the original gate and the loads. Buffers restore signal strength, minimize loading efforts, and ensurer reliable voltage levels for all fan-out connections.

10.4 Multilevel Synthesis

In some cases, multilevel circuits may reduce the **cost** of implementation, even if fan-in is not a problem at the cost of longer propagation delay

Before, logic functions were implemented in SOP or POS form. These are two-level circuits: - SOP -> first-level AND gates feeding a second-level OR - POS -> first level OR gates feeding a second-level AND

As the number of inputs grows, two-level circuits can create gates with very large fan-in (too many inputs)

Multilevel synthesis fixes this issue with smaller fan-in gates per stage, and several more stages to decompose the function into

Example:

$f = x_1x_3x_6 + x_1x_4x_5x_6 + x_2x_3x_7 + x_2x_4x_5x_7$ is in SOP form. This needs gates with high fan-in (3 or 4 input ANDs, a 4-input OR)

We can factorize into $f = (x_1x_6 + x_2x_7)(x_3 + x_4x_5)$, which is now 2 LUTs (multilevel)

This is multilevel because we have a three levels (ANDs in each parentheses, one OR in each parentheses, and one AND to sum them all up).

11 VHSIC Hardware Description Language (VHDL)

11.1 Step-by-step approach

1. Library and Use Clauses These lines bring in the standard logic definitions required for most digital projects

```
library ieee;
USE ieee.std_logic_1164.all;
```

2. ENTITY declaration This block describes the interface - inputs/outputs of your module

```
entity <entname> is
    port (
        x1, x2, x3, x4, x5 ... : in std_logic;
        output1, output2, f, ... : out std_logic;
    );
end entity <entname>;
```

3. Architecture Block This is the circuit's internal logic. Declare any needed signals (internal wires), and write logic using VHDL operators

```
architecture <arcname> of <entname> is
    signal temp1, temp2: std_logic; -- internal signals
begin
    -- assign values using boolean logic equations, examples
    temp1 <= x1 OR x2 OR x3;
    temp2 <= (NOT x3 AND x4) OR (x3 AND NOT x4);
    f <= (k AND g) OR (NOT k AND NOT g);
end architecture <archname>;
```

11.2 Template

```
library ieee;
use ieee.std_logic_1664.all;

entity Q1 is
    port (
        x1, x2, x3: in std_logic;
        f: out std_logic;
    );
end entity Q1;

architecture logicfunc of Q1 is
    signal k: std_logic; -- signals are not always necessary
begin
    k <= x1 AND x2;
    f <= k AND NOT x1;
end architecture logicfunc;
```

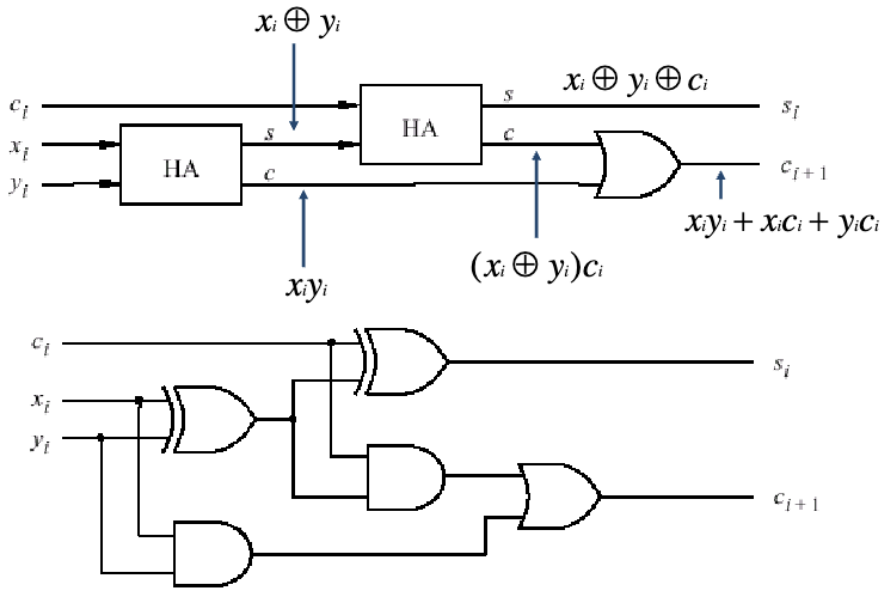
11.3 Examples

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY func1 IS
    PORT (
        x1, x2, x3 : IN STD_LOGIC;
        f : OUT STD_LOGIC
    );
END func1;

ARCHITECTURE LogicFunc OF func1 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND NOT x3) OR
         (NOT x1 AND x2 AND NOT x3) OR
         (x1 AND NOT x2 AND NOT x3) OR
         (x1 AND NOT x2 AND x3) OR
         (x1 AND x2 AND NOT x3);
END LogicFunc;
```

Use the IEEE 1164 value system, for values like `'U'` (uninitialized), `'X'` (unknown), `'0'` (logic 0), `'1'` (logic 1), `'Z'` (high impedance), `'W'`, `'L'`, `'H'` for weak/strong drive states.

11.4 Full-Adder VHDL Code

- Decomposed implementation of full-adder circuit




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fulladd IS
    PORT (Cin, x, y : IN STD_LOGIC;
            s, Cout: OUT STD_LOGIC;)

END fulladd;

ARCHITECTURE LogicFunc OF full add IS
BEGIN
    s <= x XOR y XOR Cin;
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y);
END LogicFunc;
```

Exams focus on declaring an ENTITY, and the LogicFunc, and the instructors expect proper syntax

12 Number Representation in Digital Systems

12.1 Addition of Unsigned Numbers

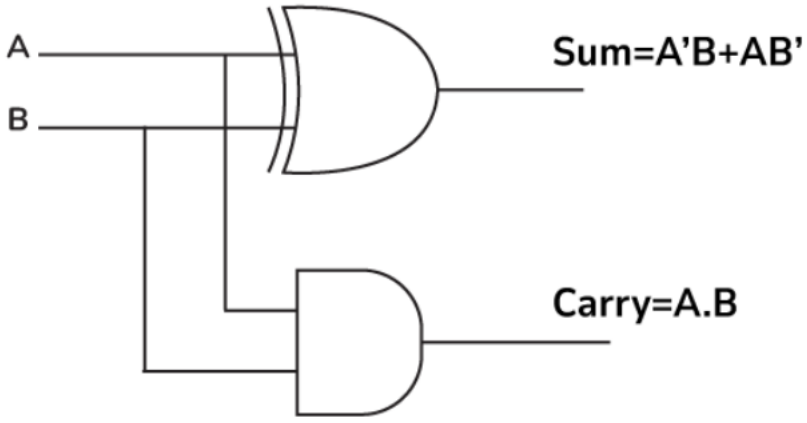
Unsigned numbers: only positive values

When adding binary numbers, four possible combinations of 0 and 1 occur. Therefore, two bits are needed to represent the result of the addition.

The rightmost bit is called the **sum, s**. The leftmost bit, which is produced as the **carry-out** when both bits added equal to 1, is called the **carry, c**.

The sum bit s is the XOR function $s = x \oplus y$, and the carry c is the AND function $c = x + y$. This circuit, which implements the addition of only two bits, is called a **half-adder**.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



When larger numbers that have multiple bits are involved, it is necessary to consider the carry from the previous bit position.

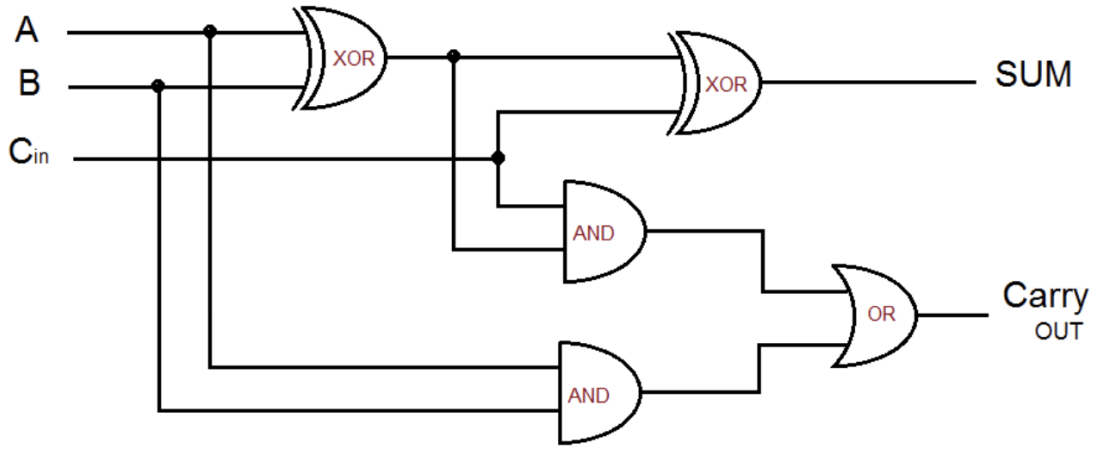
A **full-adder** allows to add two input bits and carry-in from a previous column:

Input: $(x, y), c_{in}$

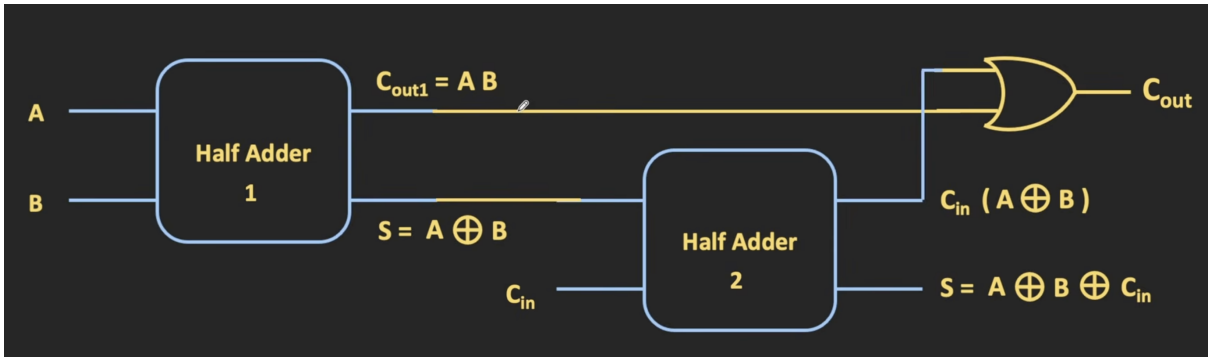
$s = x \oplus y \oplus c_{in}$

$c_{out} = (x \wedge y) \vee [(x \oplus y) \wedge c_{in}]$

Input			Output	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Full-adder using Half-adders

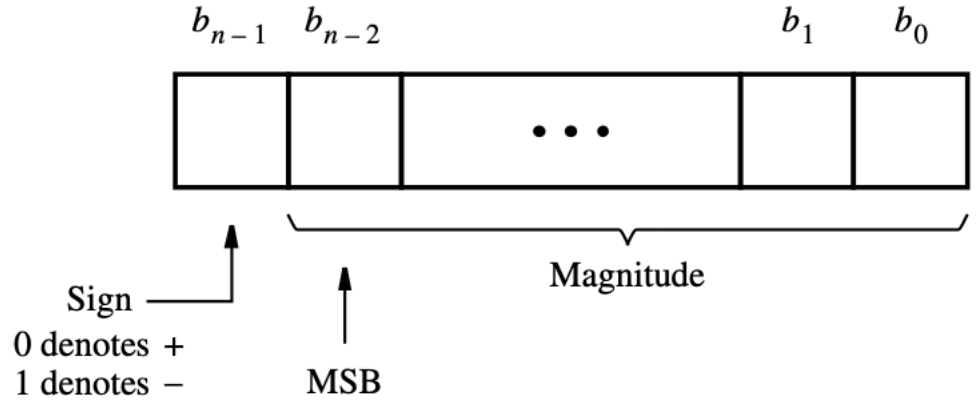


When we add multi-bit binary numbers, we need to combine several full-adders. A **ripple-carry adder** connects a series of full-adders together:

- The carry-out from each less significant stage becomes the carry-in for the next higher bit
- The process starts at the rightmost bit, where carry-in is usually set to 0

12.2 Signed Numbers and Representation

The sign of a number is indicated by a + or a − symbol. In the binary system, the sign of a number is denoted by the leftmost bit. The most significant bit (MSB) is the leftmost bit b_{n-2} and the sign is at b_{n-1}



The notation above is the **sign-and-magnitude** representation. **Example:** $+5 = 0101$ and $-5 = 1101$

In the **1's complement** scheme, a **negative number** K is obtained by simply complementing each bit of the number, including the sign bit. This results in the number K_1

In the **2's complement** scheme, a **negative number** K is obtained by taking $K_2 = K_1 + 1$

12.3 Addition and Subtraction

Unsigned addition is the same in all number systems, just add the bits and propagate carries.

Sign-and-magnitude If both numbers have the same sign: add the magnitudes and assign sign. If different signs, subtract the smaller magnitude from the larger, and the

sign of the larger wins.

1's complement Add both numbers with their sign bit, if there is a carry from the leftmost bit, add it back to the rightmost bit (end-around carry)

2's complement Add as if unsigned numbers including sign bits and discard any carry out of the sign bit. This process is standard in computers.

Example: +5 + (-3):

Decimal	Sign-mag	1's comp	2's comp
+5	0 101	0 101	0101
-3	1 011	1 100	1101
+5+(-3)	0 010 = +2	(1)0001 = 0010 = +2	(1)0010 = +2

Overflow can occur if the result is too large to represent with the available bits.

In 2's complement, overflow is detected when the carries into and out of the sign bit are different.

Example: $7 + 3 = 0111 + 0011 = 1010$

```
Carry in: 0 1 1 1
          | | | |
          0 1 1 1  (7)
        + 0 0 1 1  (3)
        -----------
          1 0 1 0  (sum)
```

The carry *into sign bit* is 1, but the carry *out from sign bit* is 0. So, since these carries are different, overflow occurred.

13 Shannon's Expansion Theorem

Shannon's Expansion Theorem: Any boolean function $f(w_1, w_2, \ldots w_n)$ can be written in the form: $f(w_1, w_2, \ldots, w_n) = \bar{w_1} \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$

$f = \bar{w_1} f_{\bar{w_1}} + w_1 f_{w_1}$

This decomposes a function into two cofactors, which helps break large expressions into smaller pierces so they're easier to analyze or minimize

Shannon Example: $f = \bar{w_1} w_3 + w_2 \bar{w_3}$

Decompose using $w_1 \Rightarrow f = \bar{w_1}(w_3 + w_2 \bar{w_3}) + w_1(w_2 \bar{w_3})$

Decompose using $w_2 \Rightarrow f = \bar{w_2}(\bar{w_1} w_3) + w_2(\bar{w_1} w_3 + \bar{w_3})$

Decompose using $w_3 \Rightarrow f = \bar{w_3}(w_2) + w_3(\bar{w_1})$, optimal solution, lowest cost

13.1 Algorithm (w_1)

1. Take $\bar{w_1}$, check each term, if $\exists \bar{w_1}$, take it out, then write OR (+) in the expansion.
2. If $\nexists \bar{w_1}$, then include the full term in the OR function, iterate
3. If $\exists \bar{\bar{w}_1}$, or $\exists w_1$, ignore it
4. Iterate
5. Then, take w_1, and iterate through the same process
6. Do for each w_n and find the lowest cost solution (or look for which w_n) appears in the most terms

13.1.1 Recursive example

We can apply this algorithm recursively, by separating the function into stages:

$$f = w_1 w_2 + w_1 w_3 + w_2 w_3$$

$$f = w_1(w_2 + w_3) + \overline{w_1}(w_2 w_3) \text{ with respect to } w_1$$

Let $g = w_2 w_3$ and $h = w_2 + w_3$

$$g = \overline{w_2}(0) + w_2(w_3) \text{ and } h = \overline{w_2} w_3 + w_2(1)$$

13.1.2 4-to-1 example

$$f = \bar{w_1} \bar{w_3} + w_1 w_2 + w_1 w_3$$

Try to express f in terms of the minterms selected by (w_1, w_2). For each combination of w_1, w_2, substitute their values into f

w_1	w_2	Selected case	f with values substituted
0	0	$\overline{w_1 w_2}$	$f = \overline{w_3}$

w_1	w_2	Selected case	f with values substituted
0	1	$\overline{w_1}w_2$	$f = \overline{w_3}$
1	0	$w_1\overline{w_2}$	$f = w_3$
1	1	w_1w_2	$f = 1$

So, the decomposition becomes:

$$f = \overline{w_1}\,\overline{w_2}(\overline{w_3}) + \overline{w_1}w_2(\overline{w_3}) + w_1\overline{w_2}(w_3) + w_1w_2(1)$$

This is useful because there are only natural inputs, and no complex gates before the multiplexer. Therefore, it is easier to program.

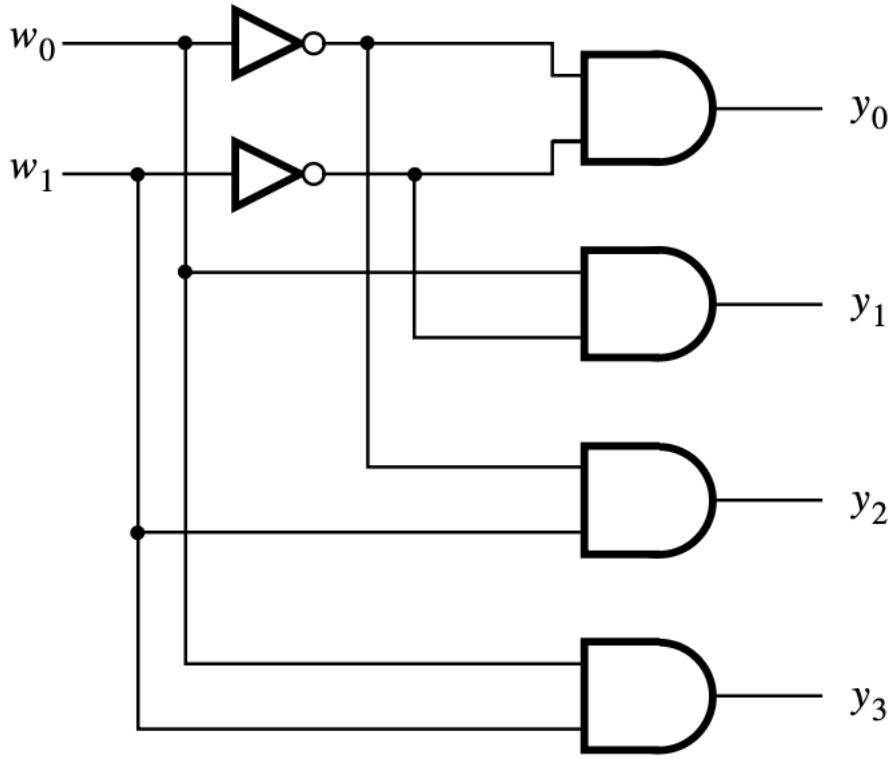
14 Decoders and Encoders

Encoders and Decoders are ways to take one set of information and change them into another type/form of information. These are both examples of code converters.

14.1 Decoders

Decoders convert binary-encoded input into a one-hot output. One-hot encoding is when one output is asserted (or hot) at a time.

As seen in the logic circuit, each combination of (w_1, w_2) outputs a different y_n value. Therefore, from a couple inputs, the decoder outputs many more values. Formally, for n inputs, the decoder produces 2^n outputs.

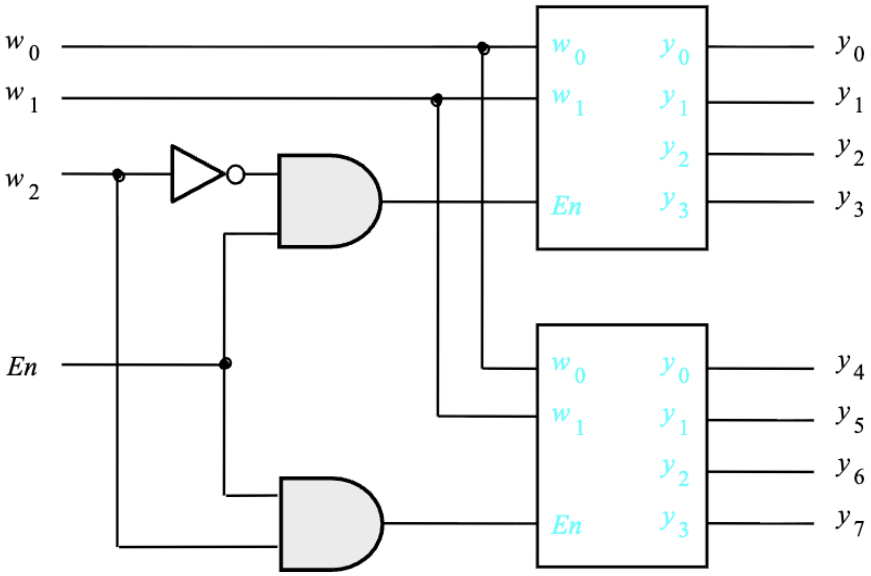


(c) Logic circuit

It is useful to include an *enable* input, *En*, in a decoder circuit. When enabled by setting $En = 1$, the decoder behaves normally. But, if it is disabled by setting $En = 0$, then none of the outputs are asserted.

For example, in the picture below, when $w_2 = 0$, the `Enable` of Encoder 2 is 0, therefore Encoder 2 is off. When $w_2 = 1$, by the not gate, the `Enable` of Encoder 1 is 0, therefore Encoder 1 is off while Encoder 2 is on.

- A 3-to-8 decoder using two 2-to-4 decoders:



14.2 Demultiplexers

The purpose of the multiplexer is to *multiplex* the n data inputs onto the single data output under control of the select inputs.

A circuit that performs the opposite function, that is pacing the value of a single data input onto multiple data outputs, is called a *demultiplexer*.

Below, we see how to use decoders as a demultiplexer.

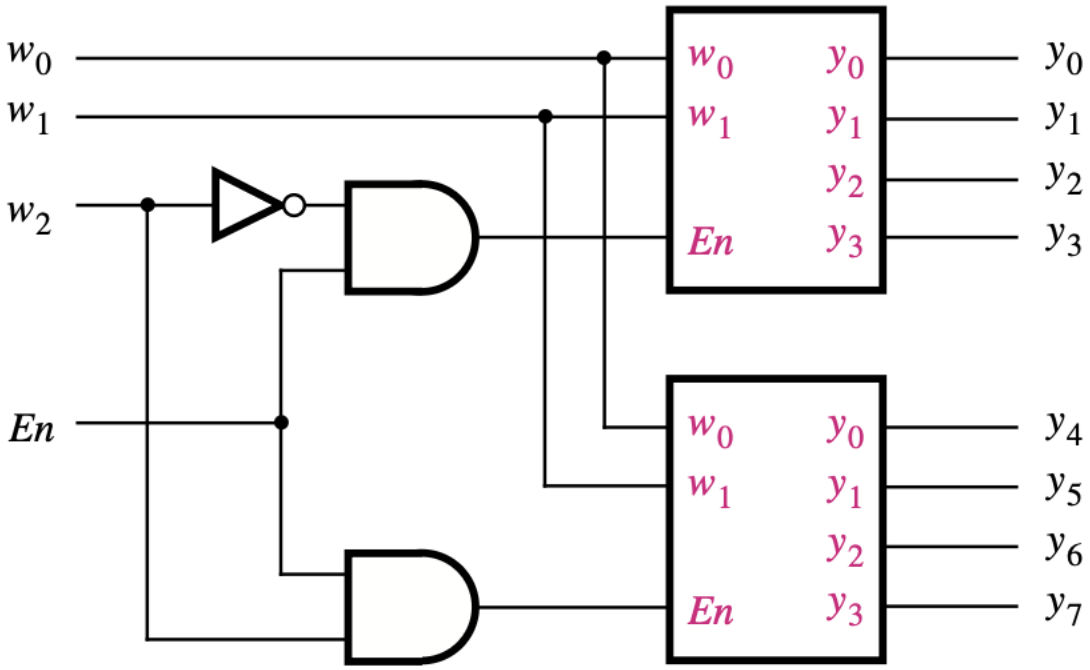


Figure 4.15 A 3-to-8 decoder using two 2-to-4 decoders.

14.3 Encoders

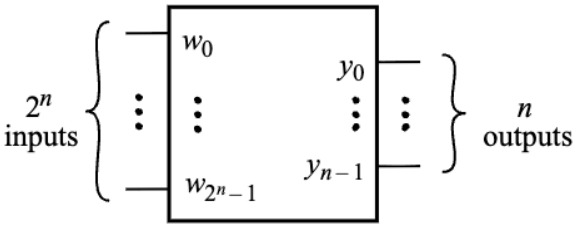
Definition: Encoders are used to encode given information into a more compact form. They convert a one-hot input into a binary code.

For example, a 4-to-2 encoder inputs 4 lines and outputs 2 bits. In general, they convert a 2^n input into a n-bit code.

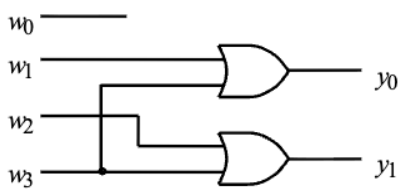
Below, notice that w_0 is not connected, because the output (y_0, y_1) is $(0, 0)$

One-hot encoded inputs,
don't-cares are not shown

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



A 2^n-to-n binary encoder



A 4-to-2 binary encoder

Priority encoders are encoders where a priority is associated with every input signal. The priority encoder outputs indicate the active input that has the highest priority.

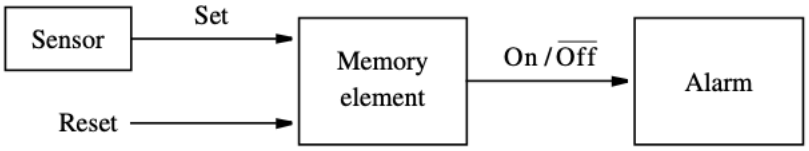
15 Flip-Flops

This chapter introduces **sequential circuits**, circuits where outputs depend not only on present inputs but also on the circuit's past behaviour. Unlike combination circuits, sequential circuits include storage elements that maintain state over time.

The need for storage elements can be shown by example: Consider an alarm system that must:

- Turn on when a sensor detects an event
- **Remain on** even after the sensor returns to normal
- Turn off only when manually reset

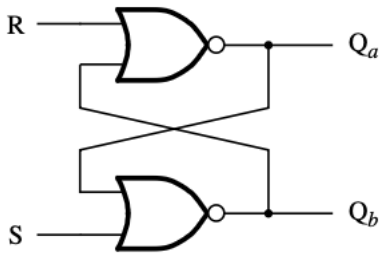
This requires memory because the circuit must remember that the alarm was triggered. Sequential circuits use storage elements to implement such memory.



15.1 Latches

15.1.1 Basic Latch and Gated Latches

The simplest storage element is the basic latch, formed by cross-coupling two NOR gates, with the following truth table:



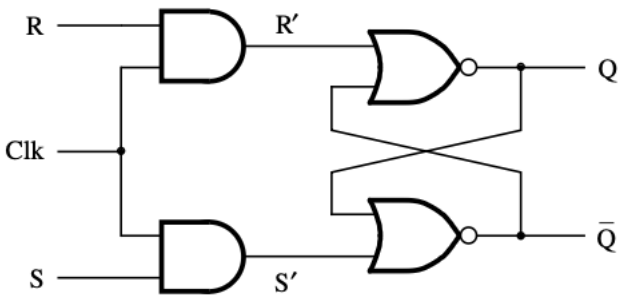
S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

Basic latches respond immediately to input changes. **Gated latches** add a control signal (Clock) to determine when the latch can change state

15.1.2 Gated SR Latch and Gated D Latch

A Gated SR Latch has AND gates control when S and R reach the basic latch

Notice how the last case $S = 1, R = 1$ is not allowed, because it leads to an undefined state (both outputs 0 temporarily, breaking the feedback logic). This is called the **forbidden state**

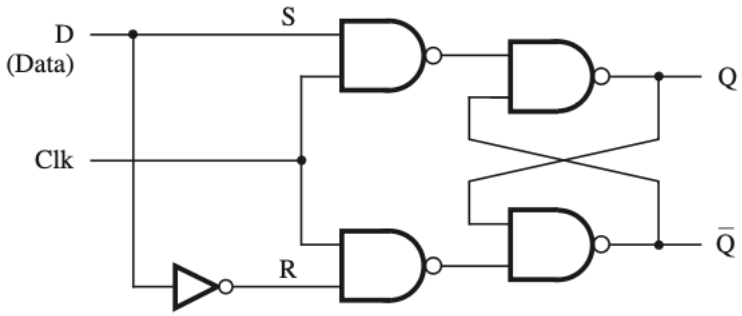


Clk	S	R	Q(t + 1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

The D latch simplifies the SR latch by eliminating the forbidden state: $S = D, \quad R = \overline{D}$. That is:

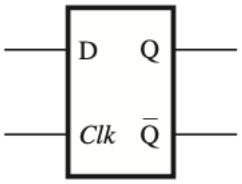
$$\text{When } D = 1 : S = 1, R = 0 \rightarrow \text{output Set } (Q = 1)$$
$$\text{When } D = 0 : S = 0, R = 1 \rightarrow \text{output Set } (Q = 0)$$

Now, you can never have $S = R = 1$, because one is always the complement of the other



(a) Circuit

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1



15.2 Edge-Triggered D Flip-Flops

Flip-flops solve the gated latch problem by responding only to clock edges (transitions from $0 \rightarrow 1$), not levels.

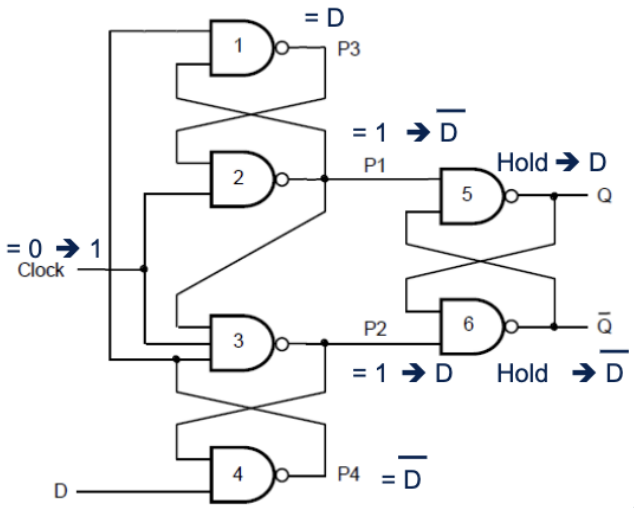
Leader-follower D flip-flop: In some circuits, it is desirable for storage elements to change their states no more than once during one clock cycle, and only at the end of Clock signal

Design: Two D latches in series:

- leader latch: loads when $Clock = 1$
- follower latch: loads when $Clock = 0$ (on falling edge)

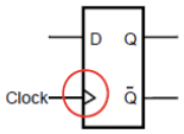
Behaviour: Output changes only on negative edge of clock

More commonly used are positive edge-triggered flip-flops, where state changes on rising edge ($0 \rightarrow 1$) of the clock



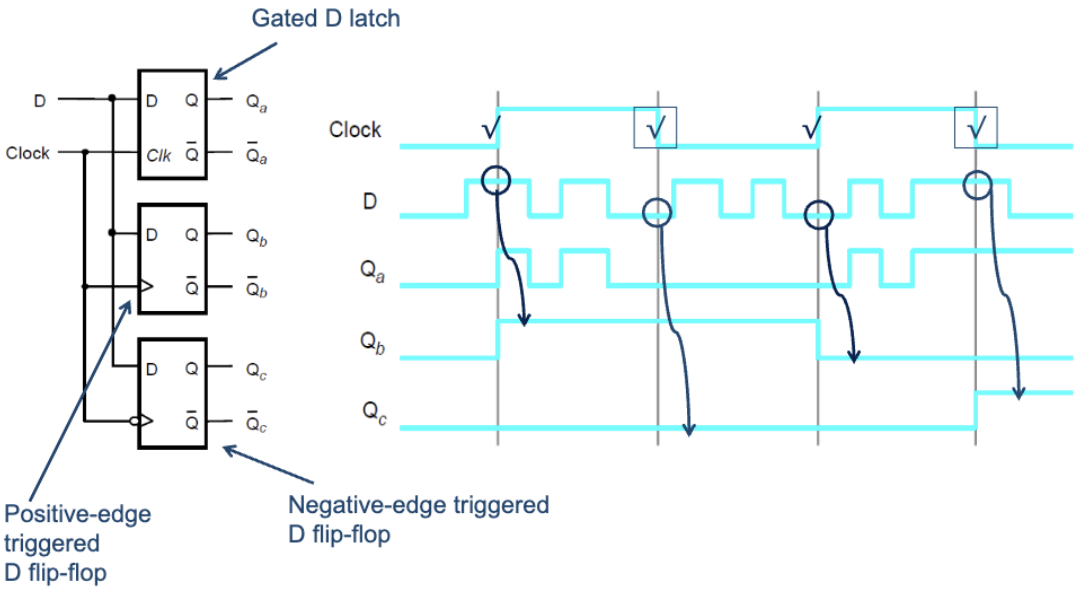
-Clock = 0 ➔ output of gate 2 and 3 are high ➔ P1 and P2 high, and the output is maintained.
-Clock = 1 ➔ P3 and P4 are transmitted through gate 2 and 3 to cause P1 = $\overline{D}$ and P2 = D. This sets Q = D and $\overline{Q}$ = $\overline{D}$

-P3 and P4 must be stable when the Clock goes from 0 to 1. After that, the changes in D have no effect.



A positive-edge-triggered D flip-flop

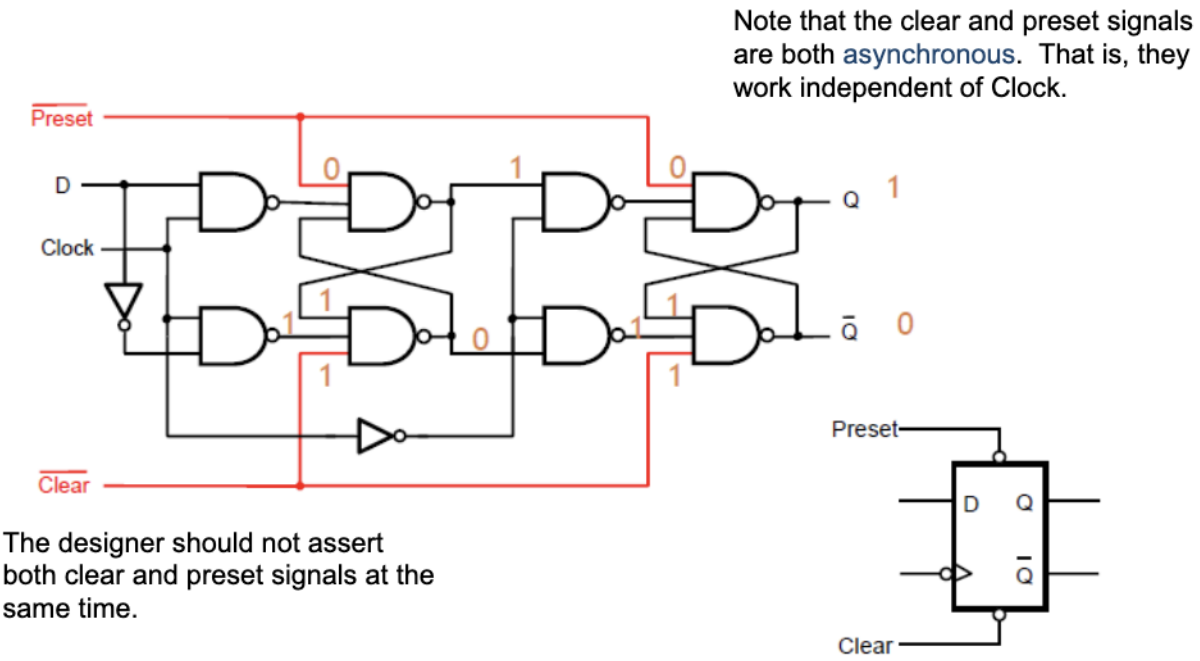
- Level-sensitive vs. edge-triggered storage elements:



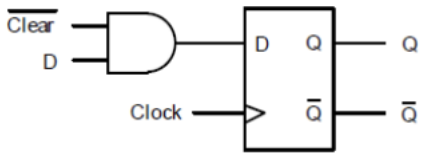
15.3 D Flip-Flops with Clear and Preset

Preset and Clear are asynchronous (independent of Clock) signals are a both `NOT`.

When we set Preset, Q will go to one. When we set Clear, Q will go to zero.



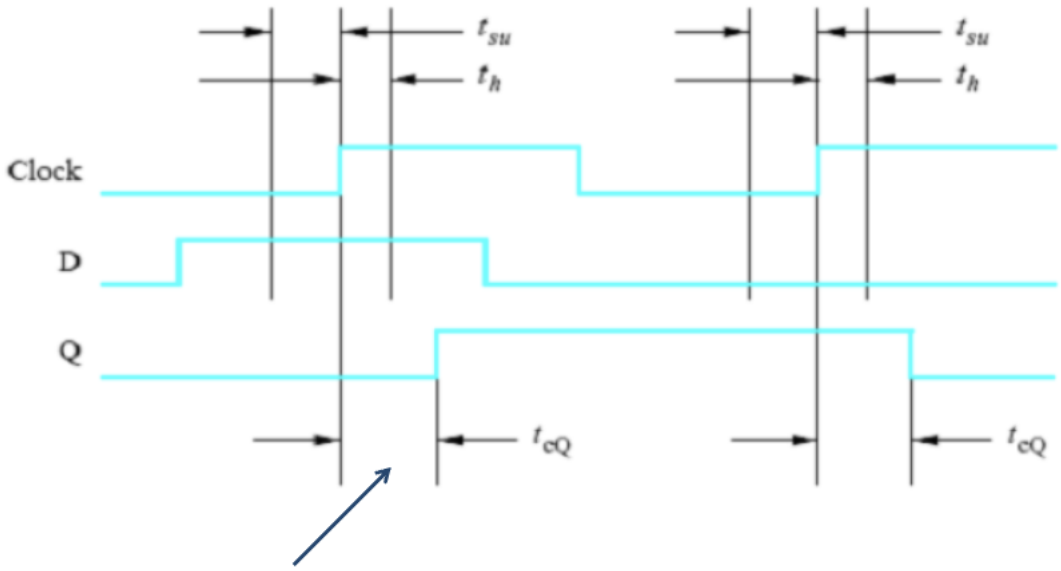
Alternatively, we could make this synchronous with an `AND` gate with `D`



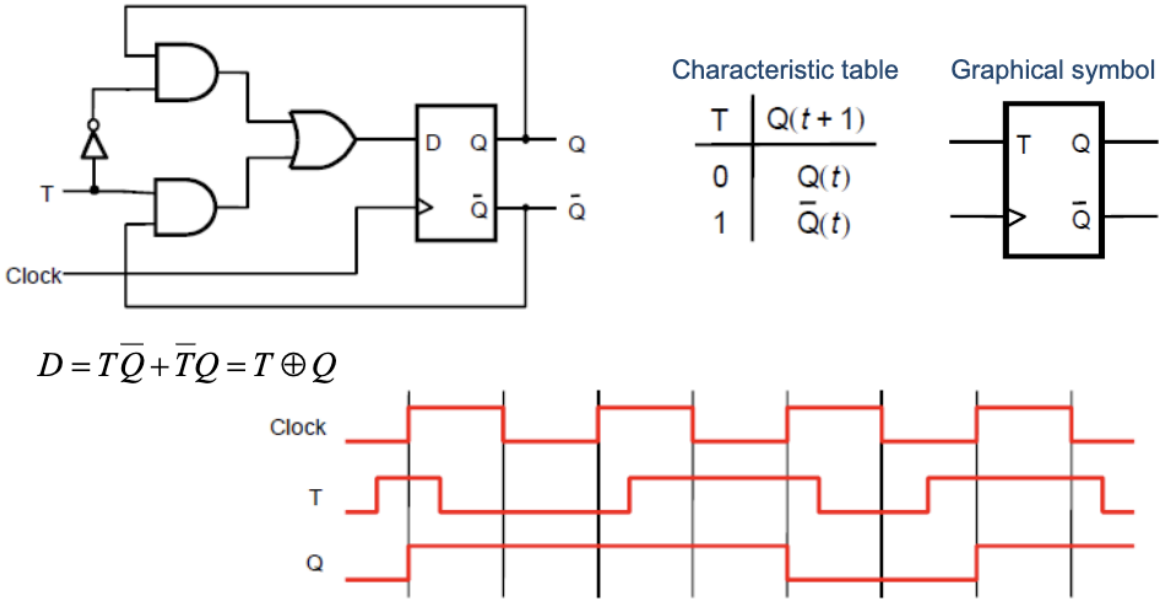
Adding a synchronous clear

15.4 T Flip-Flop

Timing issues:

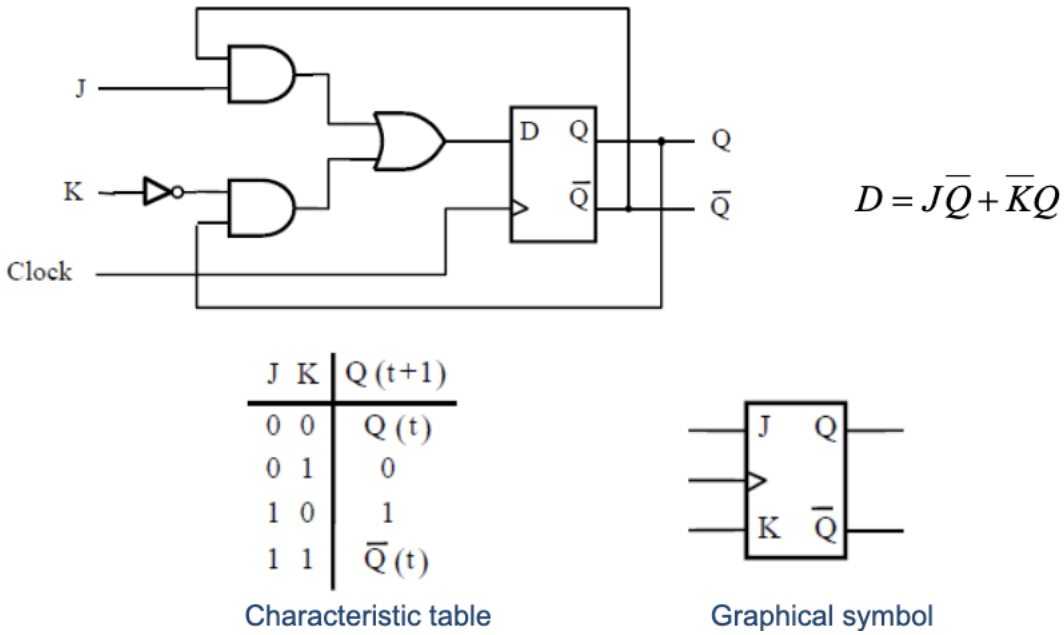


T Flip-flops are generally not used because of JK Flip-Flops. However, the behaviour is important. T Flip-Flops **toggles** its value when `T=1`.



15.5 JK Flip-Flops

JK combines the behaviours of SR and T flip-flops. It behaves like an SR for all input values except `J=K=1`, for which the JK toggles its state like T.



15.6 Timing Analysis of Flip-Flop Circuits

When we calculate t_{min}, which is the minimum clock period, it is important to know that t_{min} must allow:

1. Data to propagate
2. Satisfy the flip-flop's setup time at the next stage

We assume that $t_{su} = 0.6ns$, $t_h = 0.4ns$, $0.8ns \leq t_{cQ} \leq 1.0ns$, and the delay through any logic gate can be calculated as $1 + 0.1k$, where k is the number of inputs to the gates

For example:

$$t_{min} = t_{cQ} + t_{NOT} + t_{su} = 1.0 + 1.1 + 0.6 = 2.7ns$$

Also, note that $T_{min} = \frac{1}{F_{max}}$. Therefore,

$$F_{max} = \frac{1}{2.7ns} \approx 370.37MHz$$

To avoid data errors, we must ensure that the shortest path after a clock edge is longer than the hold time:

$$t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9ns > t_h = 0.4ns$$

15.6.1 Choosing a Flip-Flop

Gated SR latch: - Fundamental building block of other flip-flops, not recommended

D Flip-Flop: - Simplest design technique - Best choice for storage registers, preferred in VLSI technologies

T Flip-Flop: - Not really used, usually best choice for implementing counters

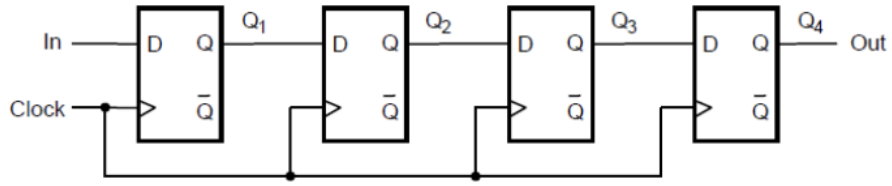
JK Flip-Flop: - Versatile building block, can be used to implement D and T flip-flops - Has two inputs with increased wiring complexity

Clear and Preset inputs highly desirable

16 Registers and Counters

16.1 Registers

Register: a set of n flip-flops with a common clock **Shift register:** a register that provides the ability to shift its contents



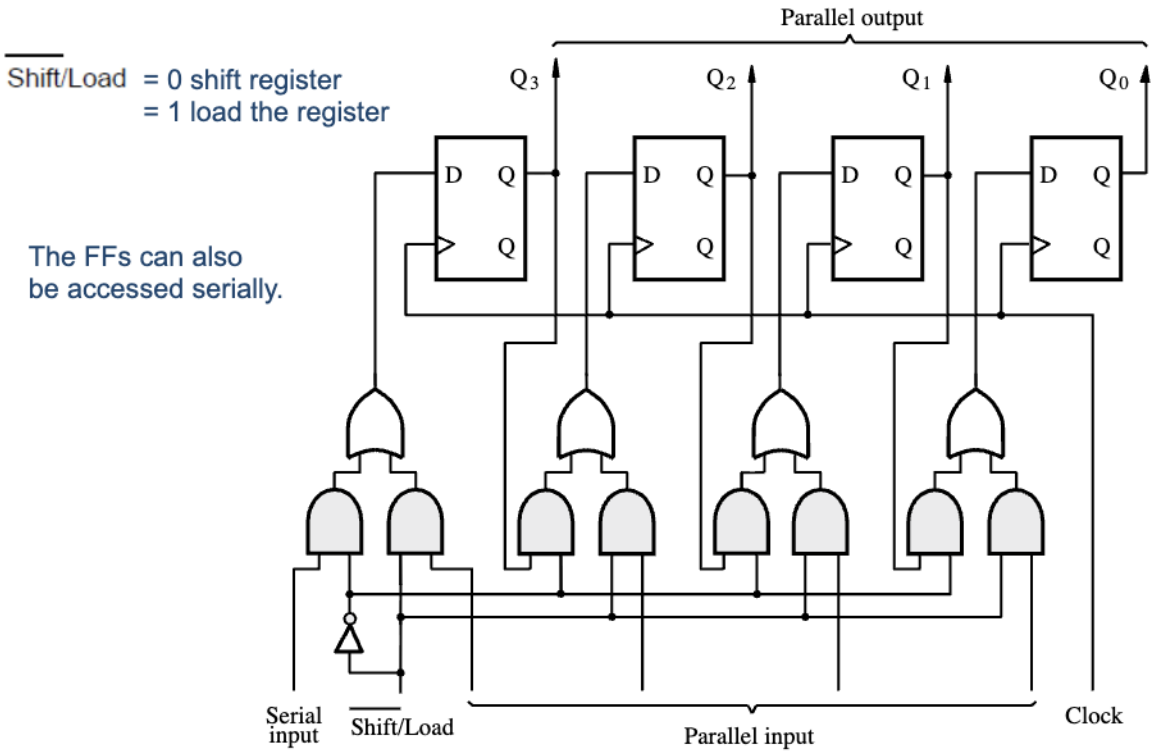
A four-bit serial shift register

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

A sample sequence

Parallel-Access Shift Register:

In the image below, we can access the output at any Q, and decide whether to shift or load the register

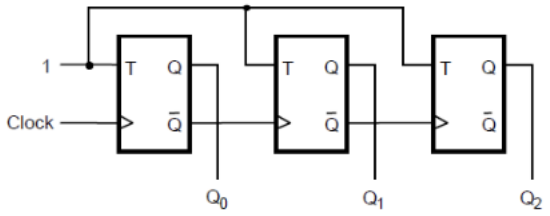


16.2 Counters

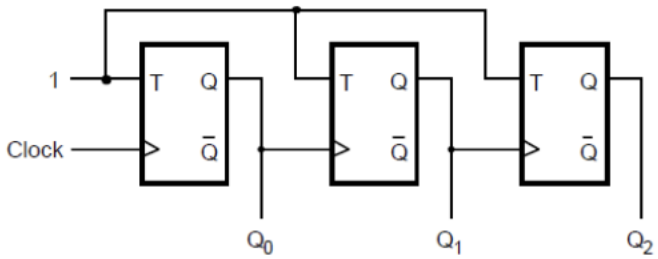
Counters: used to count the number of occurrences of certain events, generate timing intervals for control of various tasks in a system, and keep track of time elapsed between specific events.

Sometimes, we may want to inhibit counting, so that the count remains in its present state. An Enable input can do this.

3-bit up-counter, taking $\overline{Q}$ as the Clock signal, and counts upward from $0000 \rightarrow 0001 \ldots$



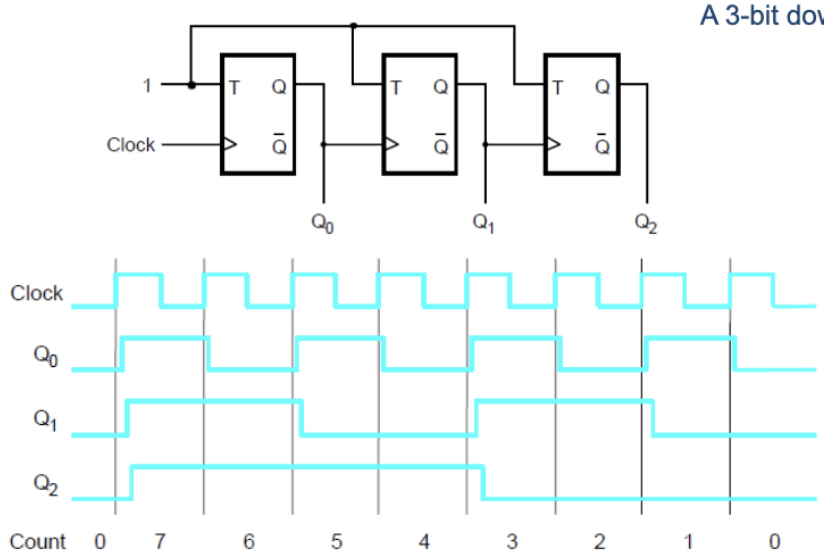
3-bit down-counter, taking Q as the Clock signal, and counts downward from $1111 \rightarrow 1110 \ldots$



Asynchronous: each subsequent FF clock input is trigged by the output of the previous FF (Q or $\overline{Q}$)

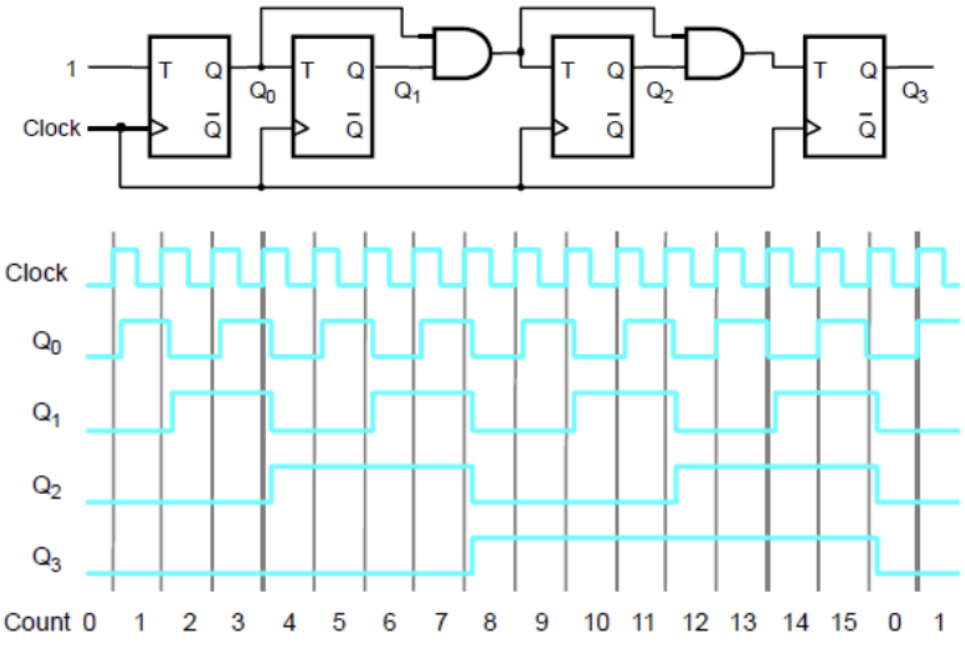
Asynchronous down-counter with T flip-flops:

A 3-bit down-counter



Synchronous: All FFs share the same clock input. Because of this, bits change at the same time and offset is reduced compared to asynchronous

A four-bit synchronous up-counter with T flip-flops :



17 Basic Design Steps and the Mealy State Model

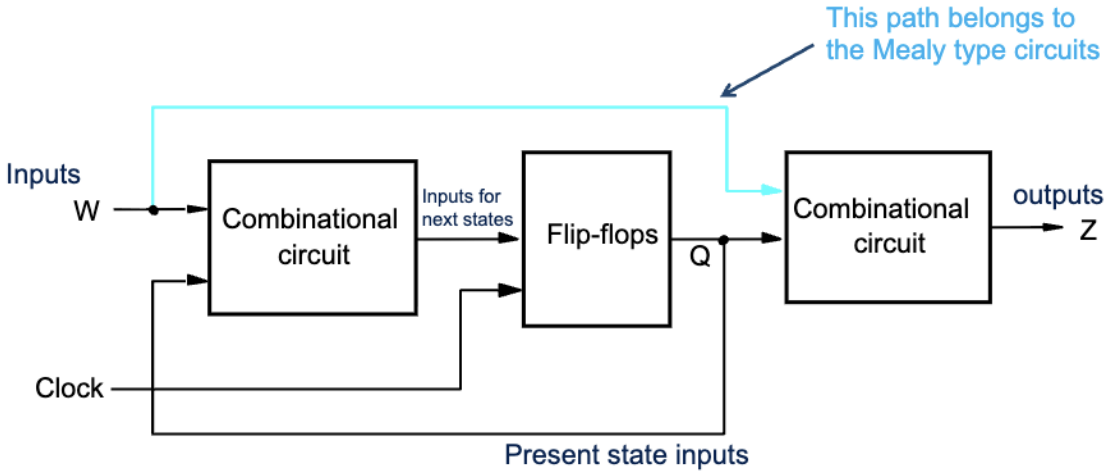
Definition: Sequential circuits are digital circuits where the outputs depend on preset values of inputs and past behaviour of the circuit (in memory).

This is different than combinational circuits, where outputs depend only on current inputs.

Sequential circuits have key characteristics:

- contain feedback paths that create memory
- have a state, a configuration that represents past history
- change state in response to inputs and clock signals

Moore type synchronous sequential circuits are where the output only depends on the state of the circuits. A **Mealy** type have outputs depending on both the state and the primary inputs, hence the cyan path below.



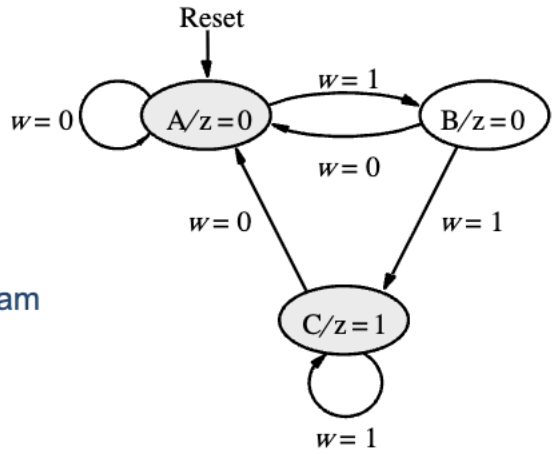
Basic Design Steps:

1. Obtain circuit specification - understand what the circuit must do
2. Derive state diagram - graphical representation of states and transitions
3. Create state table - tabular form of state diagram
4. Minimize states - reduce to minimum number of states (if possible)
5. Perform state assignment - assign binary codes to states
6. Choose flip-flop type - D, T, or JK
7. Derive next-state logic - Boolean expressions for flip-flop inputs
8. Derive output logic - Boolean expressions for circuit outputs
9. Implement circuit - draw final schematic

17.1 Sequence Detector Example

Design a circuit that detects two or more consecutive 1s on its input:

Step 1 - state diagram:



The state diagram

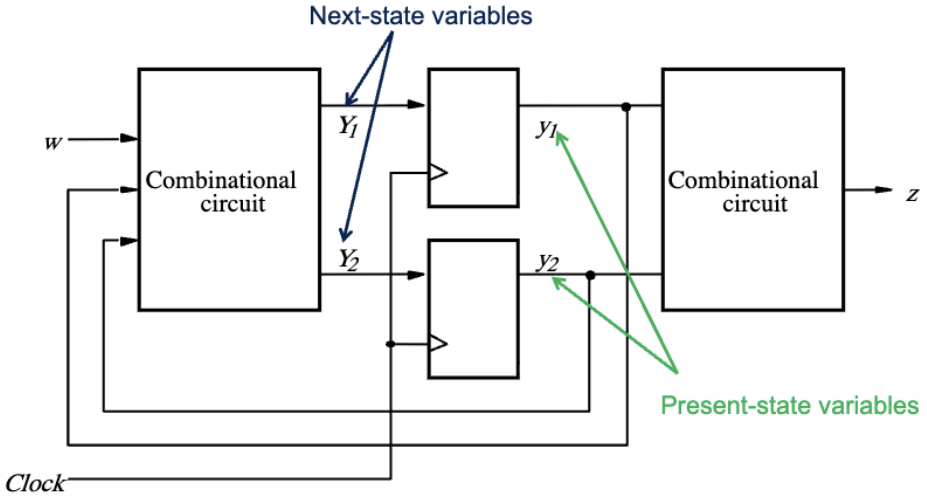
Step 2 - state table:

Present state	Next state		Output z
	w = 0	w = 1	
A	A	B	0
B	A	C	0
C	A	C	1

The associated state table

Step 3 - state assignment:

- There are three states. ➔ we need two state variables, one flip-flop per state variable.



A general sequential circuit with input *w*, output *z*, and two state flip-flops

- We need to design the combinational circuit with inputs w, y1, and y2 to produce the outputs (next states) Y1 and Y2. ➔ create the truth table.
- We also need to design the other combinational circuit with inputs y1 and y2 to produce the output, z.

  ➔ create the truth table for this Moore type FSM.

- To create the truth table, we assign a specific valuation of variables y1 and y2 to each state. There are many possibilities, including:

Present state	Next state $w=0$	Next state $w=1$	Output z
A	A	B	0
B	A	C	0
C	A	C	1

	Present state y_2y_1	Next state $w=0$: Y_2Y_1	Next state $w=1$: Y_2Y_1	Output z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

State-assigned table

Step 4 - k-maps, choose flip flops:

Present state y_2y_1	Next state $w=0$: Y_2Y_1	Next state $w=1$: Y_2Y_1	Output z
A 00	00	01	0
B 01	00	10	0
C 10	00	10	1
11	*dd*	*dd*	*d*

w \ y_2y_1	00	01	11	10
0	0	0	d	0
1	1	0	d	0

w \ y_2y_1	00	01	11	10
0	0	0	d	0
1	0	1	d	1

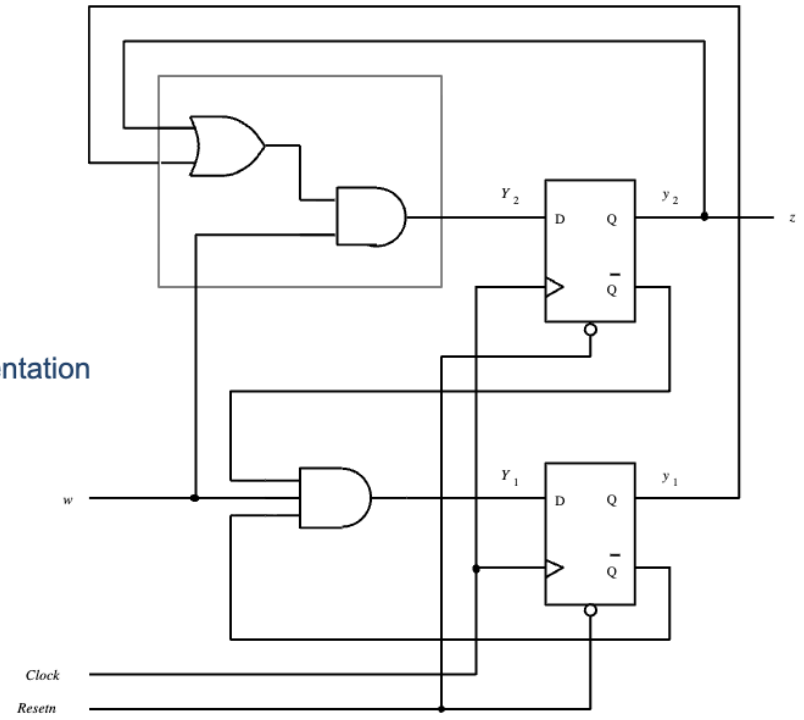
Using D flip-flops here, the easiest one

y_2 \ y_1	0	1
0	0	0
1	1	d

	Ignoring don't cares	Using don't cares
Y_1	$Y_1 = w\bar{y}_1\bar{y}_2$	$Y_1 = w\bar{y}_1\bar{y}_2$
Y_2	$Y_2 = wy_1\bar{y}_2 + w\bar{y}_1y_2$	$Y_2 = wy_1 + wy_2 = w(y_1 + y_2)$
z	$z = \bar{y}_1y_2$	$z = y_2$

- Step 4 (cont'd):

Final implementation



#ECE

18 ELEC 271 Digital Systems Cheatsheet

18.1 Digital Hardware & Binary Numbers

- **NMOS/PMOS:**
  - NMOS: Closed switch if $V_G = V_{DD}$, open if $V_G = 0$
  - PMOS: Opposite of NMOS
- **Moore's Law:** Number of transistors doubles every 18 months
- **Binary Numbers:**
  - Decimal value in base k:

$$\text{Value} = \sum_{i=0}^{n} b_i k^i$$

- Conversion: Binary to Decimal and vice versa by position and repeated division

18.2 Logic Circuits

- **Basic Gates:** AND ($\wedge$), OR ($\vee$), XOR, XNOR
- **Logic Gate Symbols:** Standardized, used to represent logical functions as circuits

18.3 Boolean Algebra & Minterm/Maxterm

- **Duality:** Interchange AND/OR and 1/0
- **DeMorgan's Laws:**
  - $\overline{A + B} = \overline{A} \cdot \overline{B}$
  - $\overline{A \cdot B} = \overline{A} + \overline{B}$
- **Minterm:** Product (AND) of all variables
- **Maxterm:** Sum (OR) of all variables
- **Sum-of-products (SOP):** Canonical form, minimum-cost by reducing gates and variables
- **Product-of-sums (POS):** Canonical form, uses maxterms

18.4 Multiplexer & VHDL Basics

- **Multiplexer (MUX):** Circuit selecting one of multiple inputs
  - $MUX(S; A, B) = SA + \overline{S}B$ (2-to-1 example)
- **VHDL:** Hardware description language for digital circuits

18.5 Transistors & Voltage Levels

- **Logic Values:** Positive logic: 0 (low, 0V), 1 (high, e.g. 5V)
- **Noise Margins:**
  - High-state margin (NM_H), Low-state margin (NM_L), differences between voltage levels

18.6 CMOS Logic Gates

- **CMOS:** Uses both NMOS and PMOS transistors, low power
- **AND/NAND/NOR:**
  - AND = NAND + NOT
  - OR = NOR + NOT

Notice that when two signals are stacked, they are OR'ed, and when they are in series, they are AND'ed. Furthermore, the complement of the NMOS logic function is the PMOS logic function, and vice-versa.

Gate Type (n-input)	CMOS Transistors Needed
NOT	2 (1 PMOS, 1 NMOS)
NAND	$2n$ (2 PMOS, 2 NMOS)
NOR	2n (2 PMOS, 2 NMOS)
AND	$2n$+2 (NAND + NOT)
OR	$2n$+2 (NOR + NOT)
XOR	~8
XNOR	~10

18.7 Programmable Logic Devices

- **PLA:** Programmable AND plane, followed by OR plane
- **PAL:** Only programmable AND array
- **FPGA:** Contains logic blocks (LUTs), IO blocks, and interconnection wires

18.8 Karnaugh Maps (K-Maps)

- **Purpose:** Simplify logic functions
- **Groups:** Combine minterms (SOP) or maxterms (POS) into rectangles representing implicants
- **Prime Implicant:** Maximal group
- **Essential Prime Implicant:** Covers at least one minterm not covered elsewhere
- **Don't-care condition:** Useful for minimization

18.9 Multiple Output Circuits & Multilevel Synthesis

- **Multiple Outputs:** Share product terms to reduce total cost
- **Multilevel Synthesis:** More than 2 logic levels (AND/OR), can reduce fan-in

18.10 VHDL Template/Example

```
library ieee;
use ieee.std_logic_1664.all;

entity Q1 is
    port (
        x1, x2, x3: in std_logic;
        f: out std_logic;
    );
end entity Q1;

architecture logicfunc of Q1 is
    signal k: std_logic; -- signals are not always necessary
begin
    k <= x1 AND x2;
    f <= k AND NOT x1;
end architecture logicfunc;
```

18.11 Number Representation

- **Unsigned:** Only positive values
- **Signed:** Positive and negative (Two's complement)
- **Addition:**
  - Half-adder: $s = x \oplus y,\ c = x \wedge y$
  - Full-adder combines two half-adders

Tip: Visualize gate and K-map diagrams as you study, and practice transforming truth tables to canonical SOP/POS forms for minimization.