alex-levesque.com

ELEC 271 - Lecture Notes

DIGITAL SYSTEMS

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

Contents

#ECE

1 Digital Systems Overview

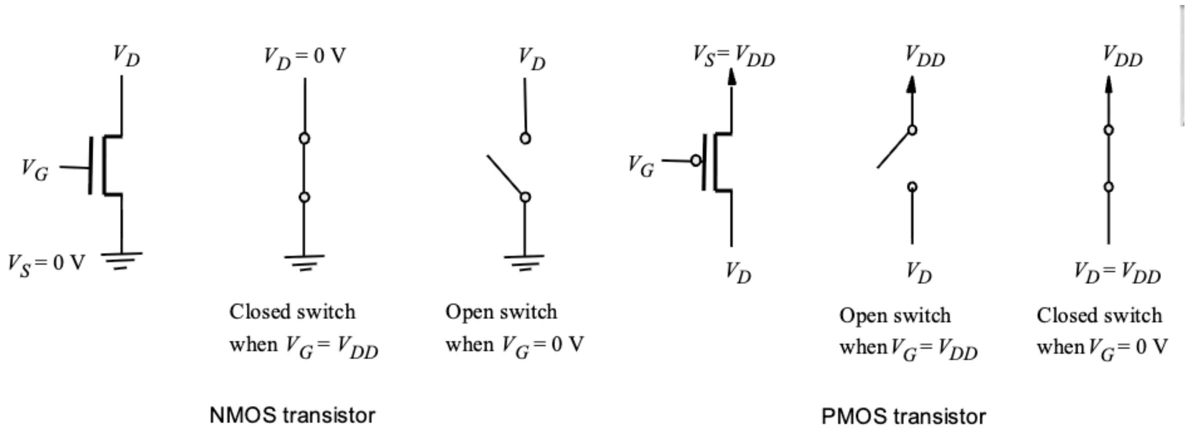
[[09-03 Digital Hardware and Binary Numbers]] [[09-04 Boolean Algebra and Min-Maxterms]] [[09-08 Multiplexers and VHDL]] [[09-10 Logic Circuits]] [[09-15 Transistor Switches]] [[09-17 CMOS Logic Gates]] [[09-18 Programmable Logic Devices]] [[09-22 Karnaugh Maps]] [[09-24 Karnaugh Maps Related Topics]] [[10-06 Number Representation in Digital Systems]] [[09-25 VHDL Examples]] [[ELEC 271 Cheat Sheet]]

#ECE

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

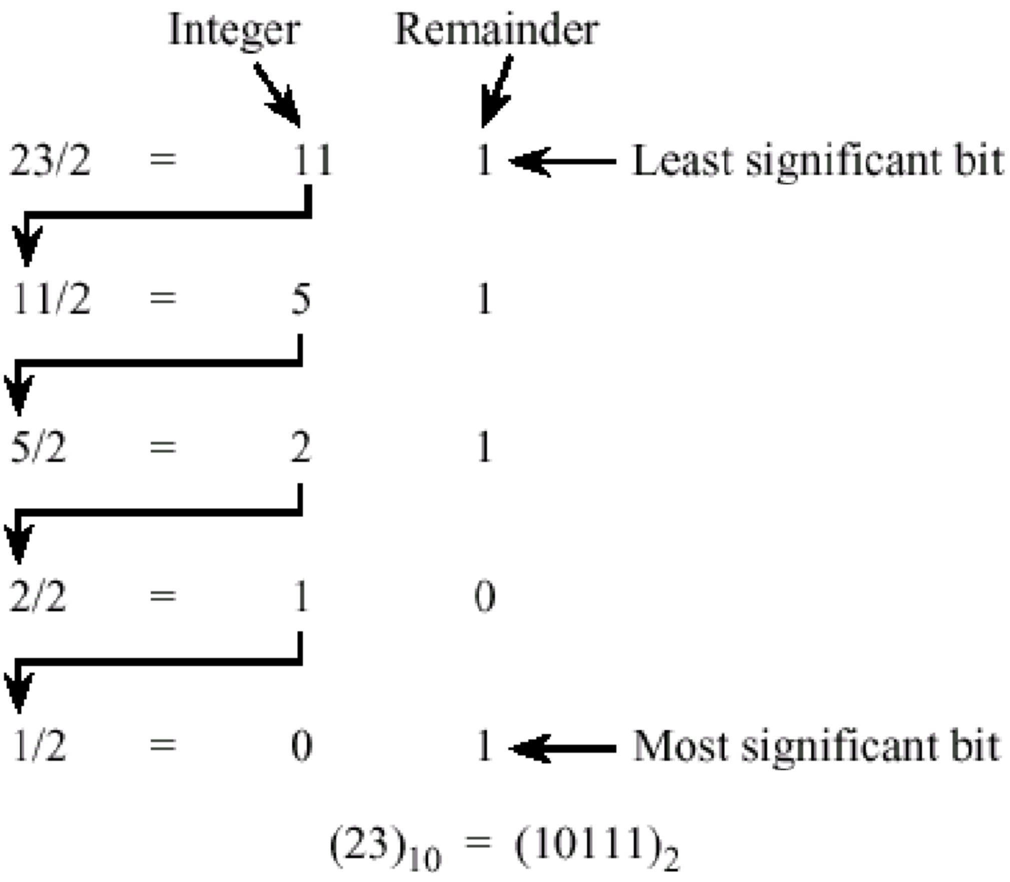
To convert a binary number to decimal, multiply each digit b_i by 2^i, where i is the position

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

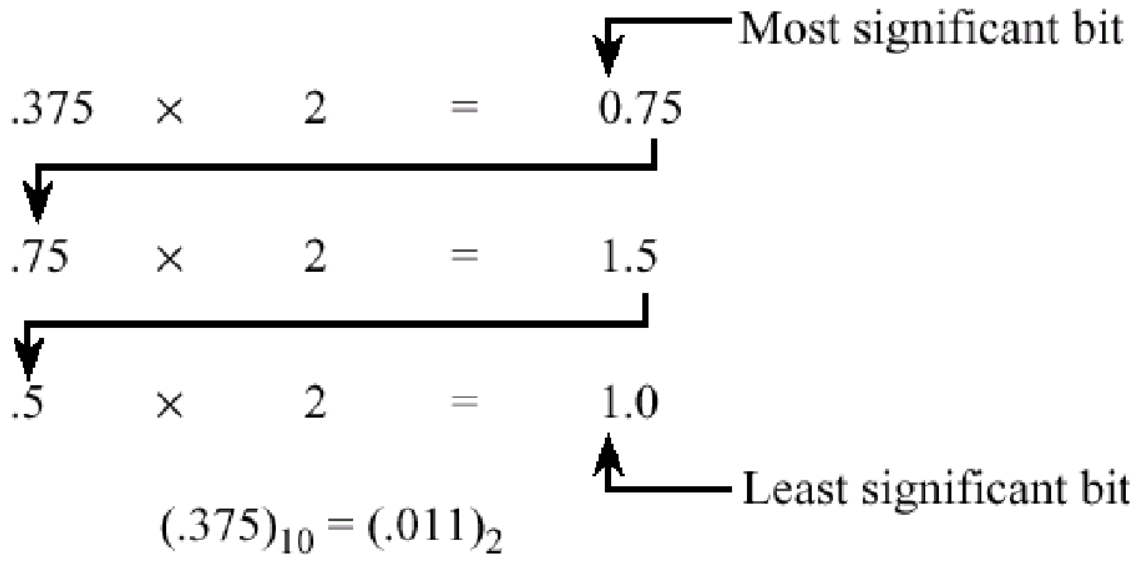
Converting decimal to binary, we divide the number by 2 as many times as possible

Example: Convert 23.375 to base 2:

Start with the integer part:



Then do the decimal part:



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

#ECE

3 Boolean Algebra and Minterm/Maxterm

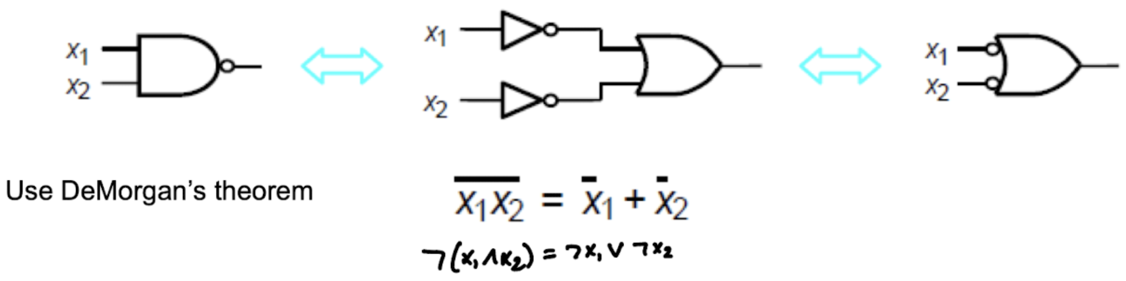
This is very similar to MTHE 217:

$$\begin{aligned} 1+1&=1 \\ x+x&=x \\ x+\bar{x}&=1 \\ \bar{\bar{x}}&=x \end{aligned}$$

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

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

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

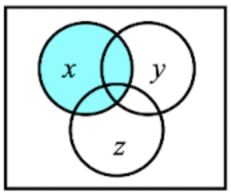


x_1
x_2
Use DeMorgan's theorem
$\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$
$\neg(x_1 \wedge x_2) = \neg x_1 \vee \neg x_2$

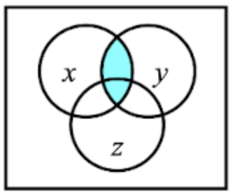
An important result when simplifying a logic equation is $x * \bar{x} = 0$

3.1 Venn Diagrams

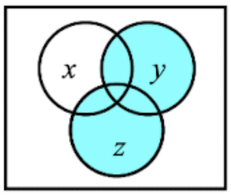
We can intuitively how two expressions may be equivalent with Venn diagrams



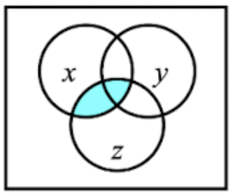
(a) x



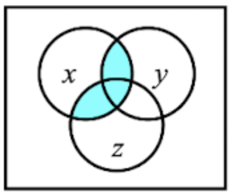
(d) $x \cdot y$



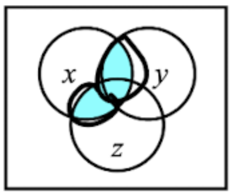
(b) $y+z$



(e) $x \cdot z$



(c) $x \cdot (y+z)$



(f) $x \cdot y + x \cdot z$

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: $(x_1x_2+\bar{x_1}x_2+x_1\bar{x_2}+\bar{x_1}\bar{x_2}) = 1$

#ECE

4 Multiplexers and VHDL

4.1 Multiplexer Circuit

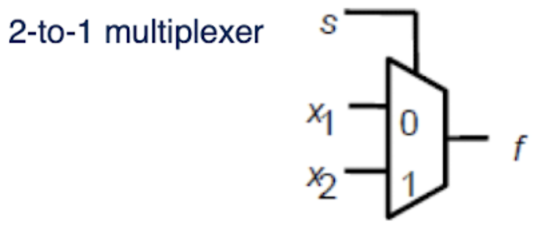
This is a circuit that chooses data from exactly one of the number of possible input sources

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:

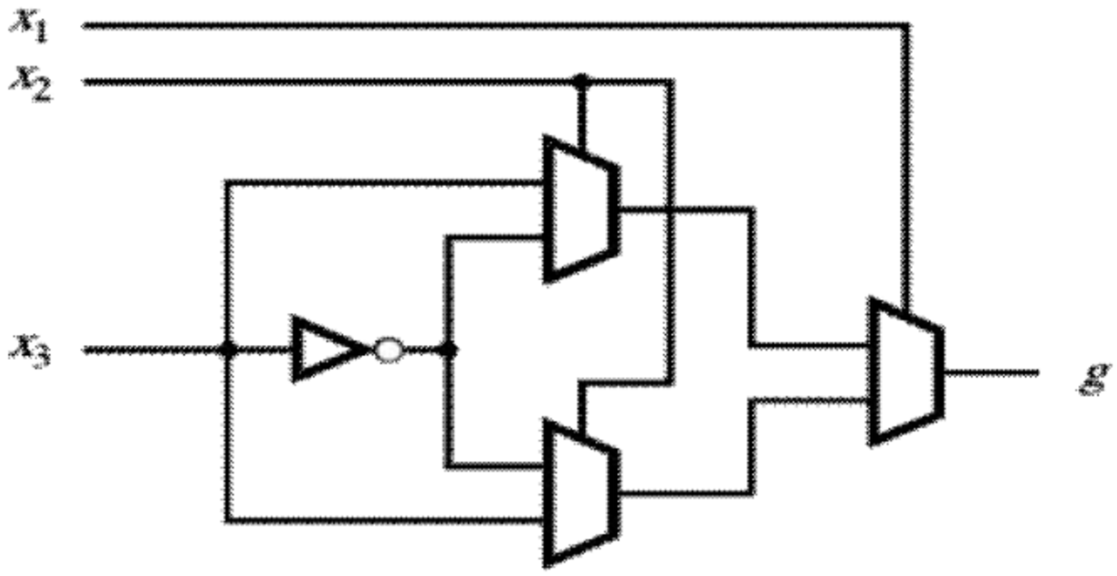
$f = (I_0 * \bar{S}) + (I_1 * S)$, or formally, $MUX(S; A, B) = \overline{S}A + SB$

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:



$MUX_1 = (x_2; x_3, \overline{x_3}) = \overline{x_2}x_3 + x_2\overline{x_3} = x_2 \oplus x_3$ $MUX_2 = (x_2; \overline{x_3}, x_3) = \overline{x_2}\overline{x_3} + x_2x_3 = x_2 \odot x_3$ $MUX_3 = (x_1; x_2 \oplus x_3, x_2 \odot x_3) = x_1 \oplus x_2 \oplus x_3$

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)

#ECE

5 Logic Circuits

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

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

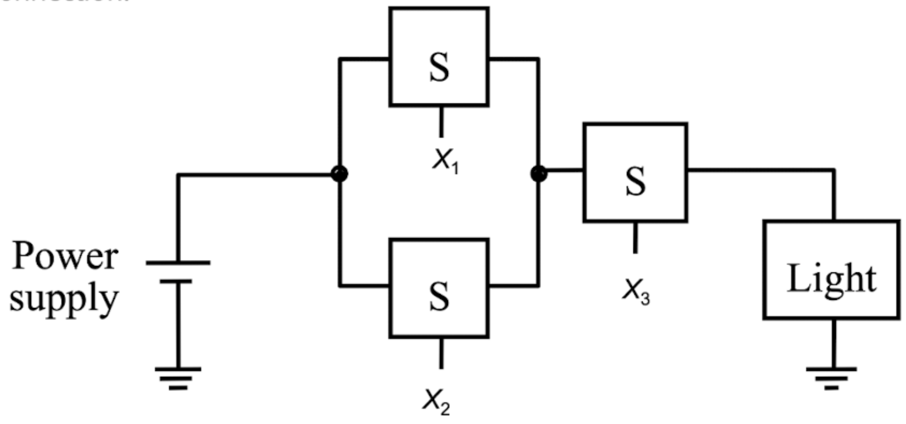
The logical XNOR ($\odot$) outputs 1 if both inputs are the same, this is the complement of XOR

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:

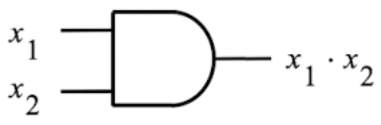


$L(x_1, x_2, x_3) = (x_1 + x_2) * x_3$

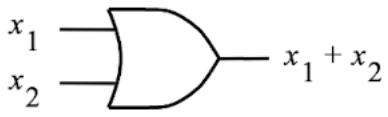
5.2 Logic Gates and Circuits

We can use several gates to represent connectors

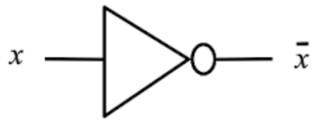
AND gates

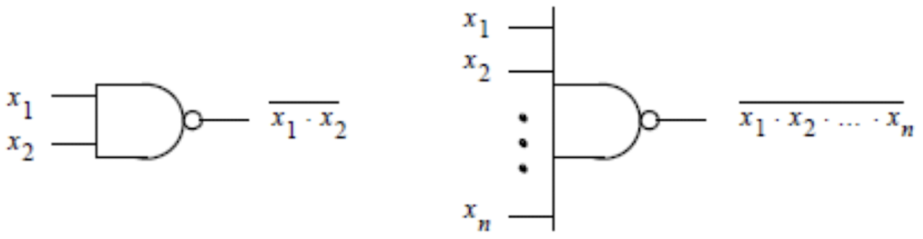


OR gates



NOT gate

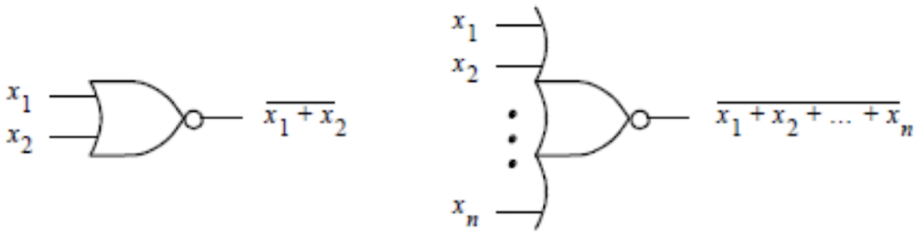



x_1
x_2
$\overline{x_1 \cdot x_2}$
x_1
x_2
x_n
$\overline{x_1 \cdot x_2 \cdot \ldots \cdot x_n}$

Truth table?

NAND gates

Truth table?



NOR gates

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

#ECE

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.

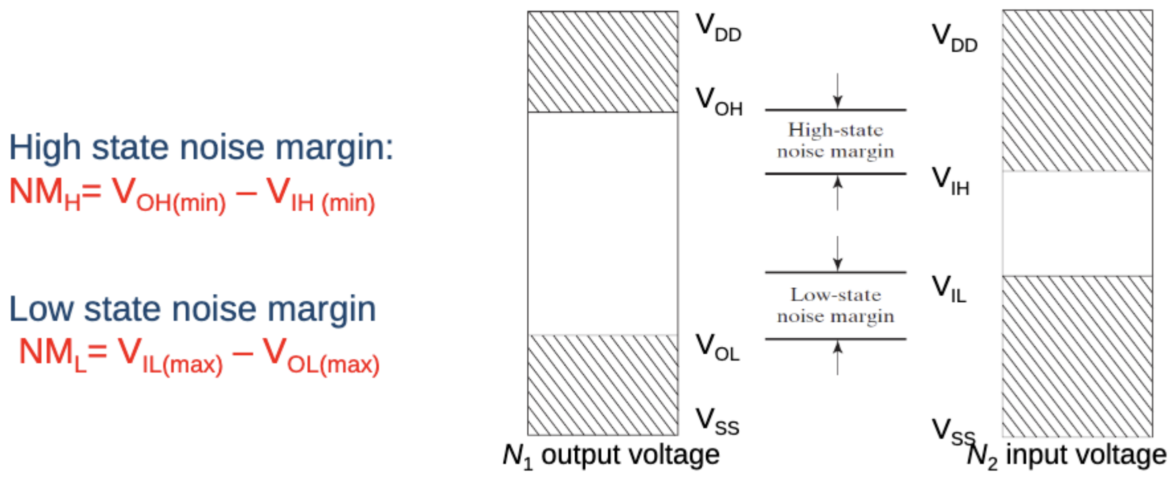
- High-state margin: $NM_H = V_{IL_{max}} - V_{OL_{max}}$

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"

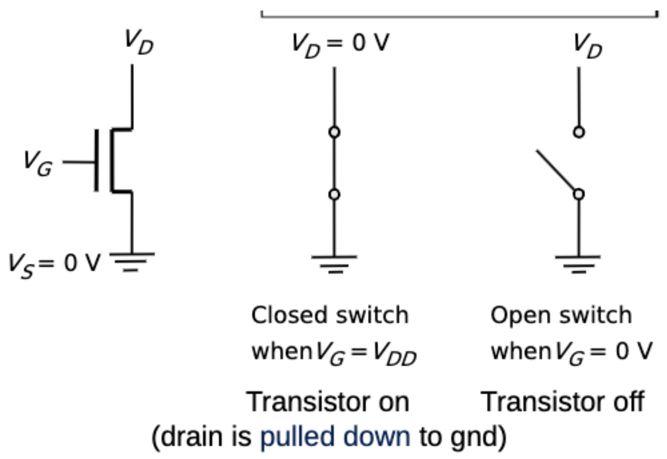
Below: V_{SS} is the lowest voltage V_{DD} is the highest voltage



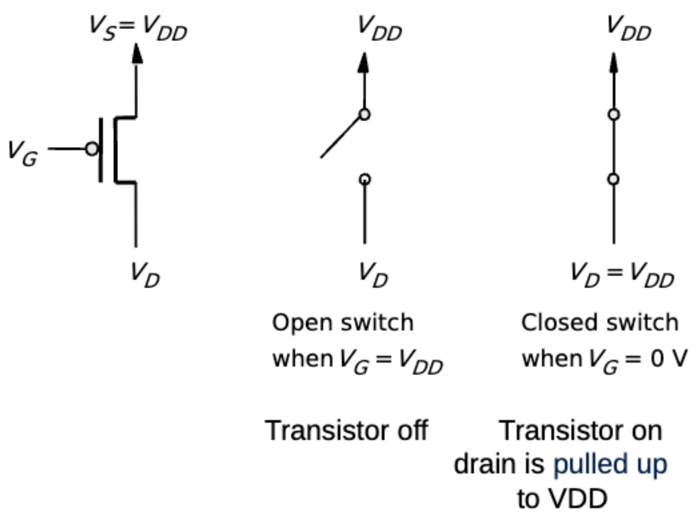
6.2 Transistor Switches

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



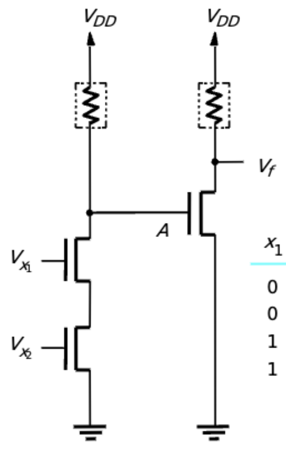
V_D
V_G
$V_S = 0$ V
$V_D = 0$ V
V_D
Closed switch when $V_G = V_{DD}$
Open switch when $V_G = 0$ V
Transistor on
(drain is pulled down to gnd)
Transistor off



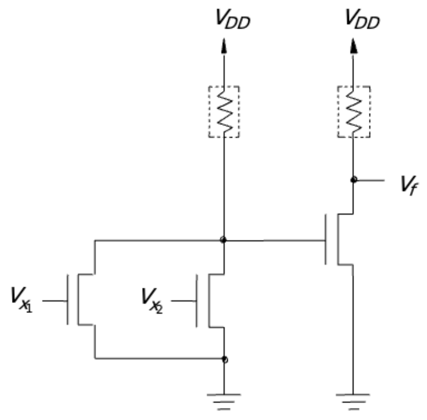
$V_S = V_{DD}$
V_G
V_D
V_{DD}
V_D
V_{DD}
$V_D = V_{DD}$
Open switch when $V_G = V_{DD}$
Closed switch when $V_G = 0$ V
Transistor off
Transistor on
drain is pulled up to VDD

AND and OR gate using NMOS

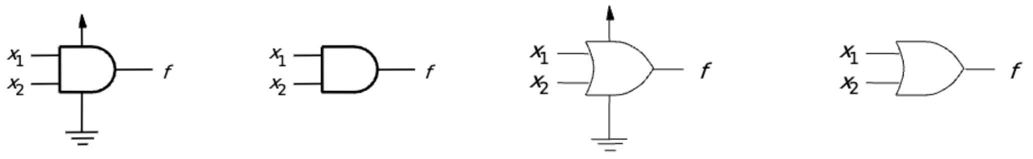
- An AND and an OR gate using NMOS technology:
-



x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1



#ECE

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)

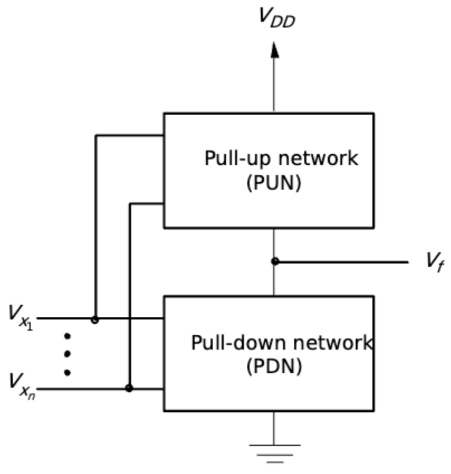
In the image below, the **pull-up network (PUN)** is made of PMOS transistors and connects the output node to V_{DD} when the logic function requires a `1`

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.

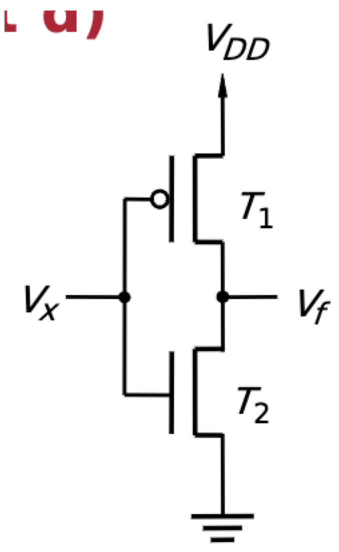


Structure of an CMOS circuit

CMOS Examples

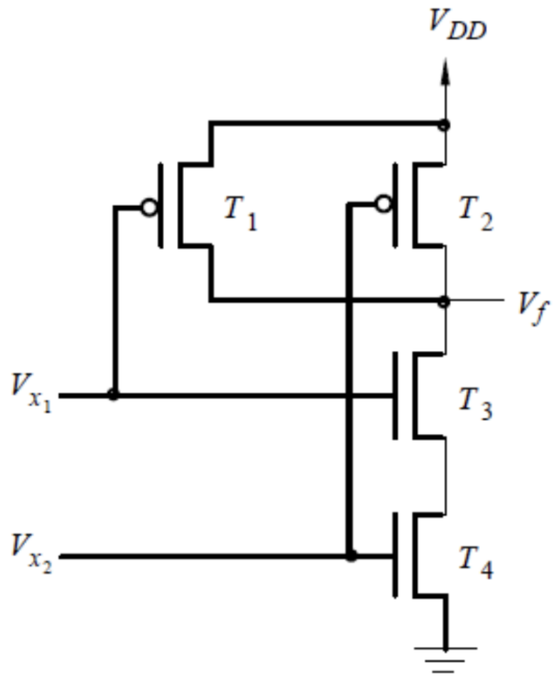
A **NOT gate** reverses the input logic state.

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

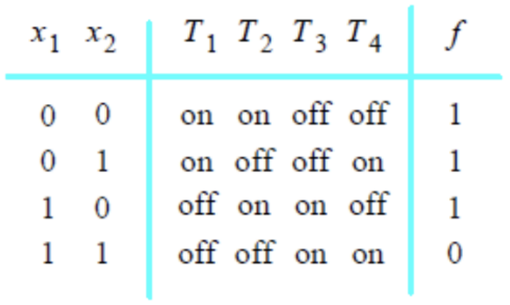


A **NAND** gate outputs the opposite of an AND gate, $f = \overline{x_1 x_2}$

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

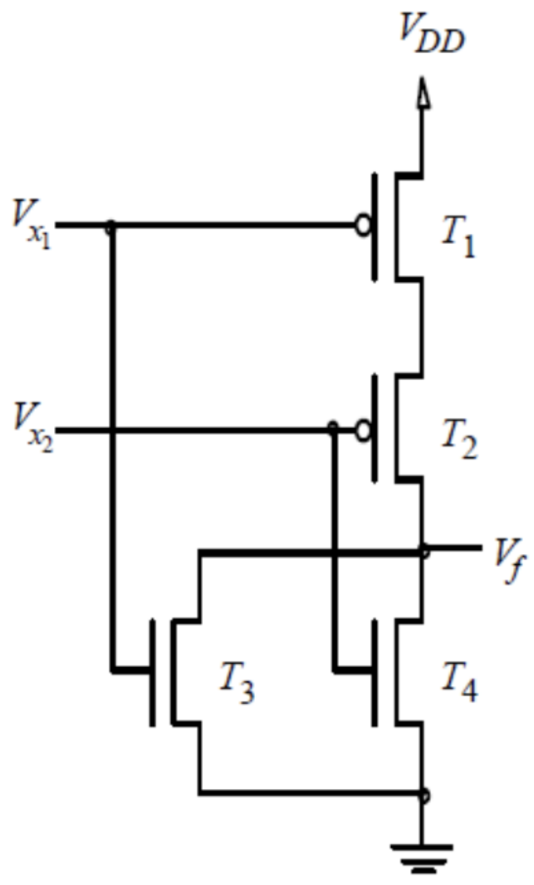


$$f = \overline{x_1 x_2}$$

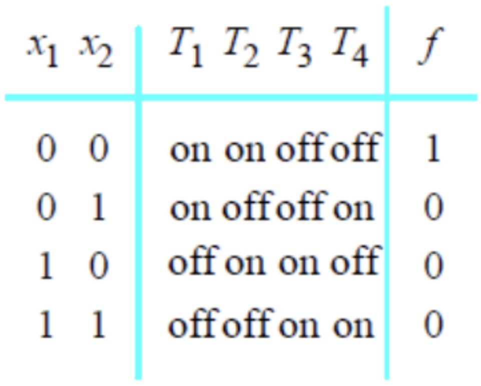


x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

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



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

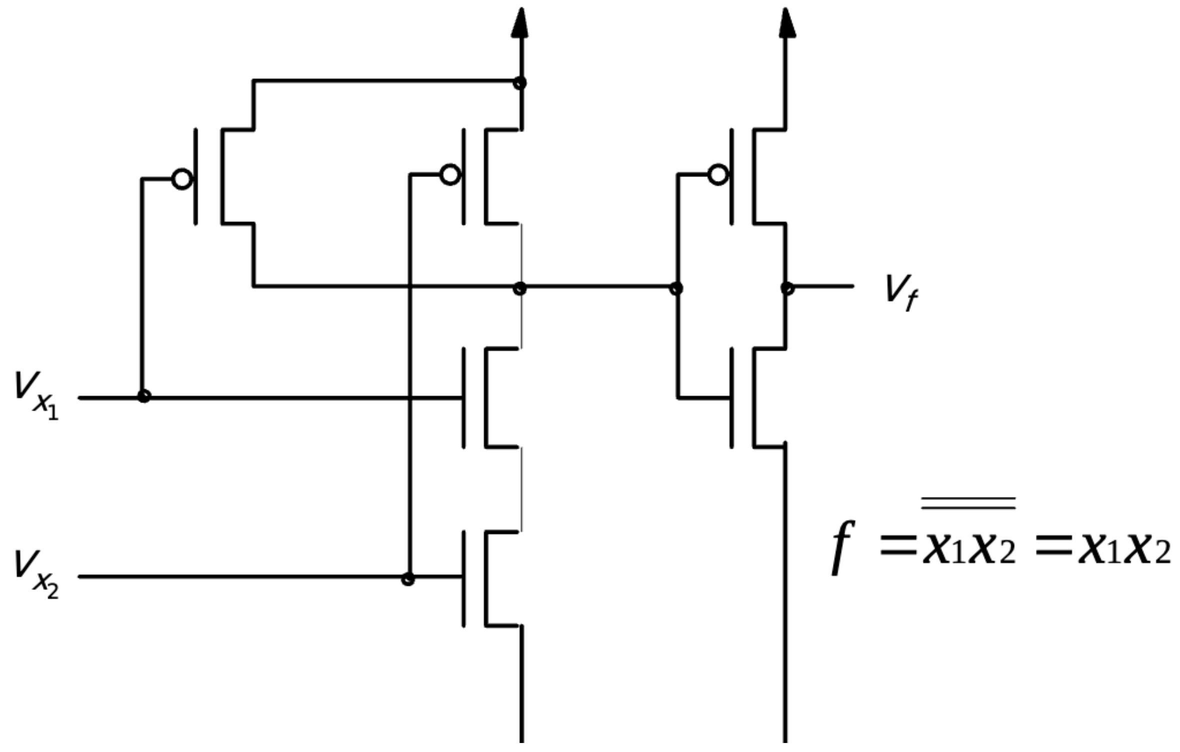


x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

Circuit

Truth table and transistor states

An **AND** gate is mote complex than one may think. If you try to directly wire PMOS/NMOS 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

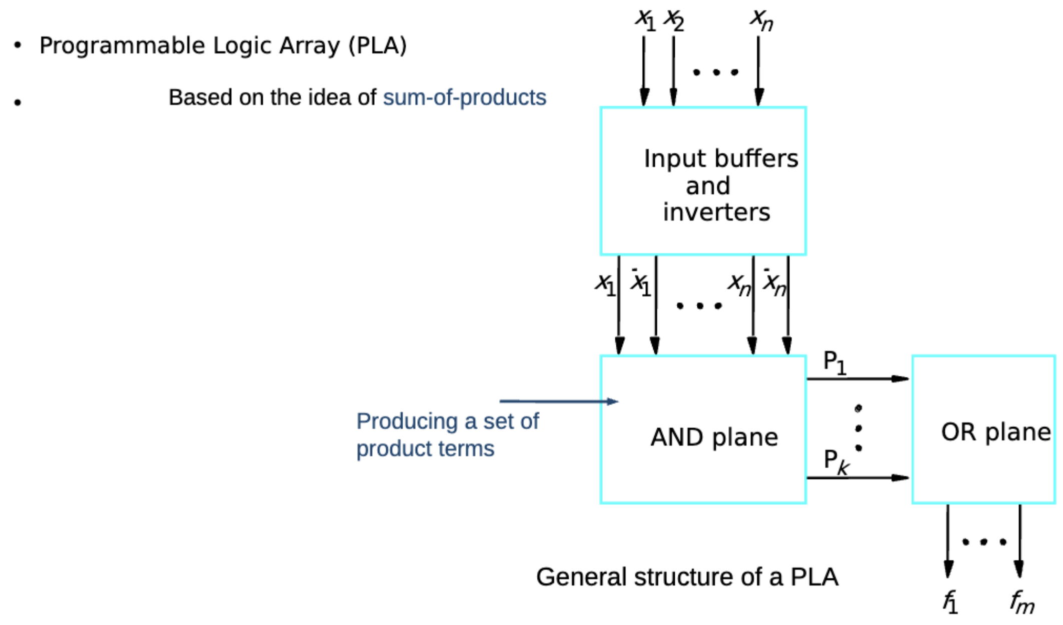
#ECE

8 Programmable Logic Devices

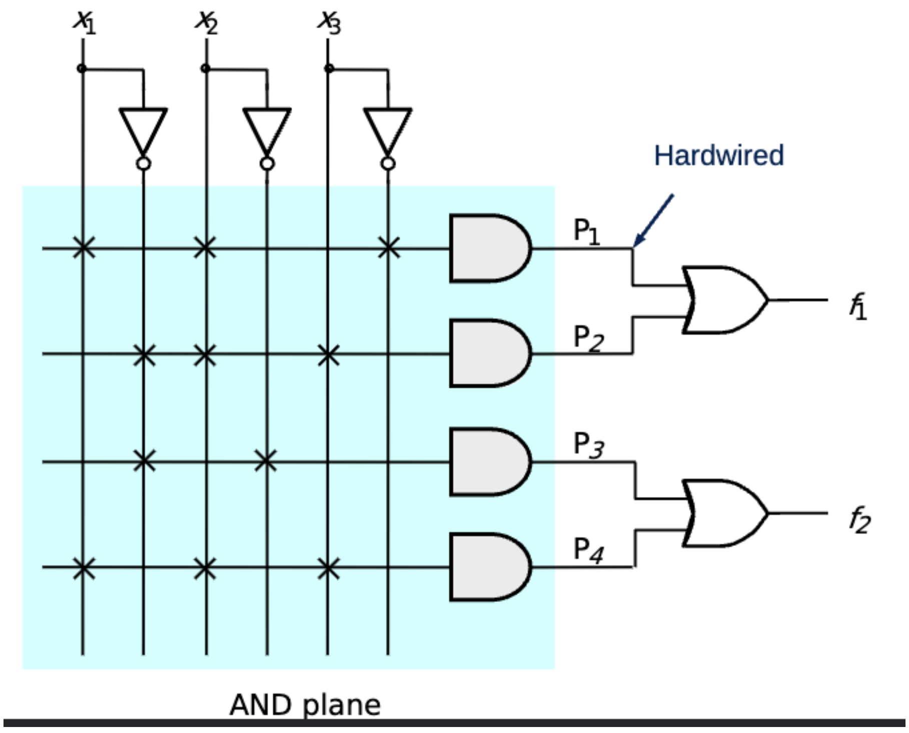
Programmable Logic Devices (PLDs) is a general-purpose chip for implementing logic circuits, where the input is logic variables and the output is logic functions

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
x_1 x_2 x_n
Input buffers and inverters
x_1 $\bar{x}_1$ x_n $\bar{x}_n$
Producing a set of product terms
AND plane
P_1
P_k
OR plane
f_1 f_m
General structure of a PLA

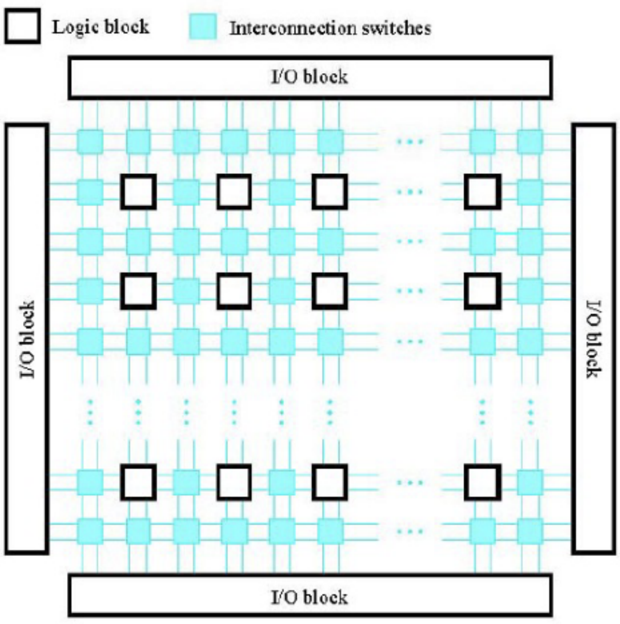
A **Programmable Array Logic** (PAL): only programmable AND array, for cost reason and better performance, but less flexible



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.

A LUT is essentially a small block of memory:

- the input bits serve as the address
- the stored bit at that address is the output (0 or 1)

Because the LUT stores the complete truth table for its inputs, it can represent any Boolean function

Python analogy example:

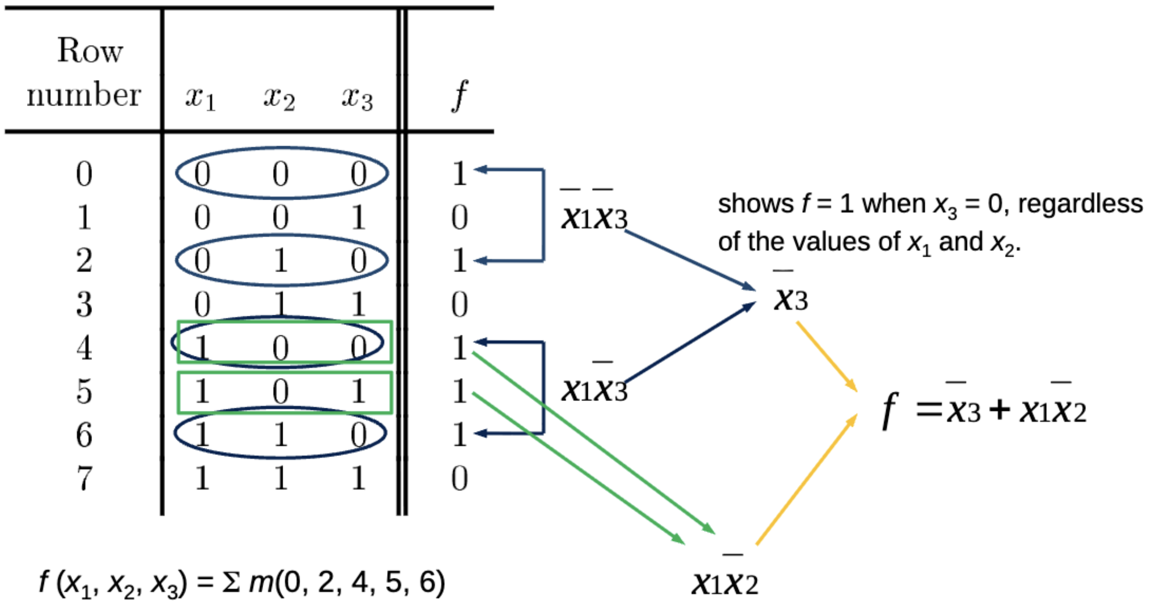
```python
# XOR function implemented as a LUT
lut = {
    (0, 0): 0,
    (0, 1): 1,
    (1, 0): 1,
    (1, 1): 0
}

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

#ECE

9 Karnaugh Maps (K-maps)

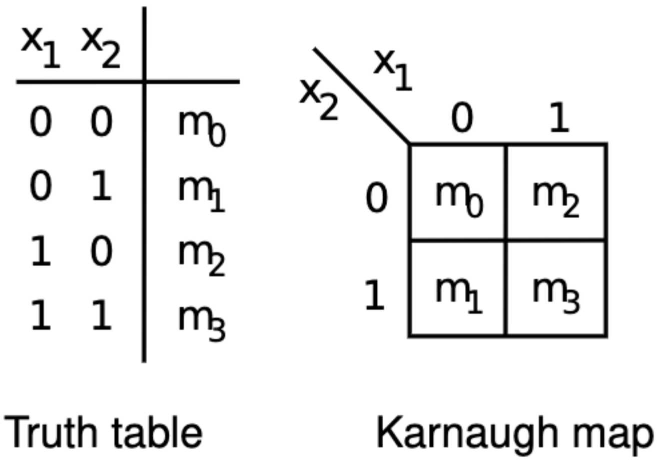
A **Karnaugh Map** provides a systematic way of optimizing logic functions. It is an alternative to the truth table and allows easy discover of groups of *minterms* for which $f = 1$ can be combined into single terms

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



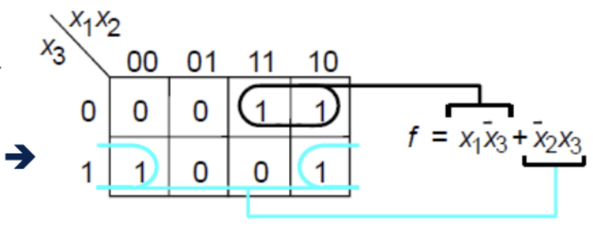
Building a Karnaugh Map:

Two variables:

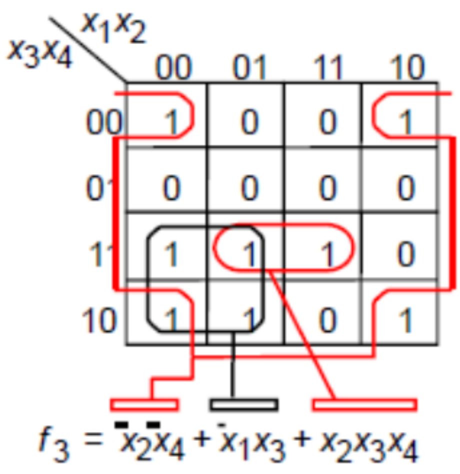


Three variables:

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

$$f = x_1\bar{x}_2 + x_3\bar{x}_4 \qquad \text{Cost} = 9$$

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)

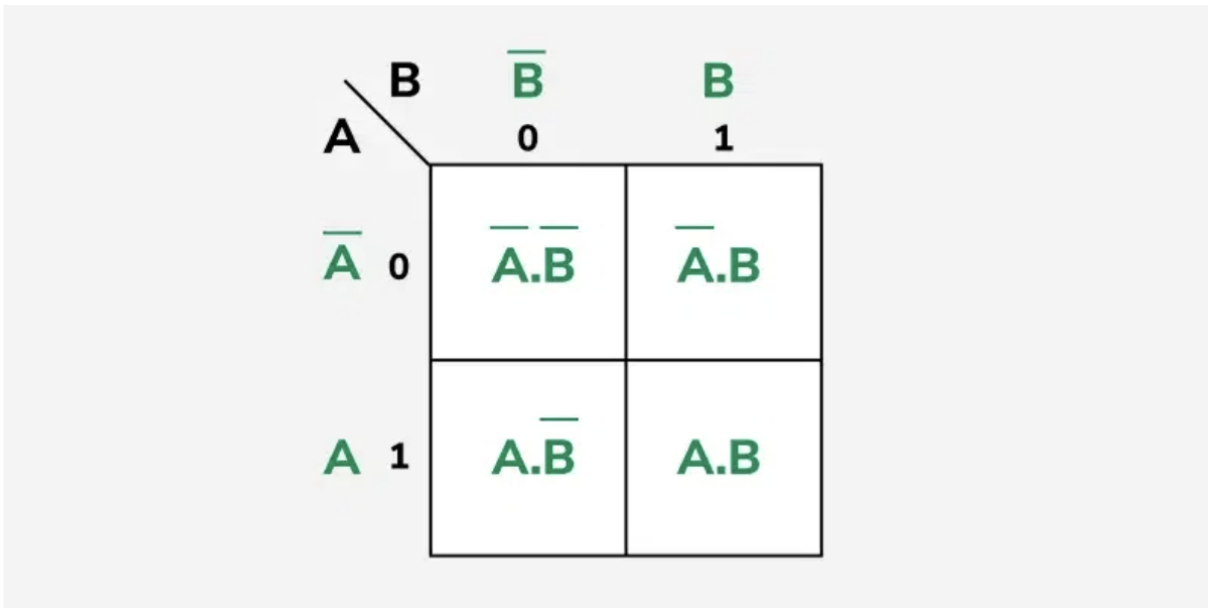
Prime Implicant: A group of 1's that cannot be combined into a larger group of 1's (maximal rectangle of 1's)

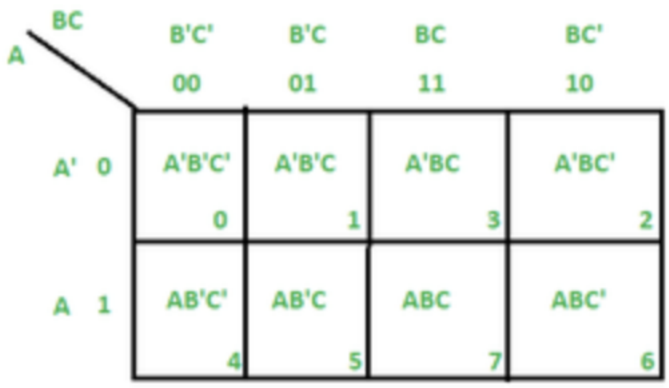
Essential Prime Implicant (EPI): A prime implicant that covers at least one 1 in the K-map that no other prime implicant covers. These are must-have groups in your final simplified function.

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

9.1.1 SOP Solution

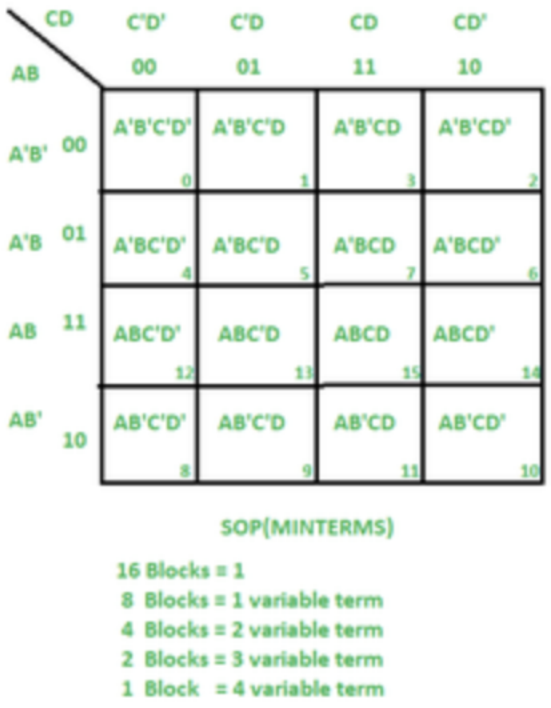
Select K-map according to number of variables





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

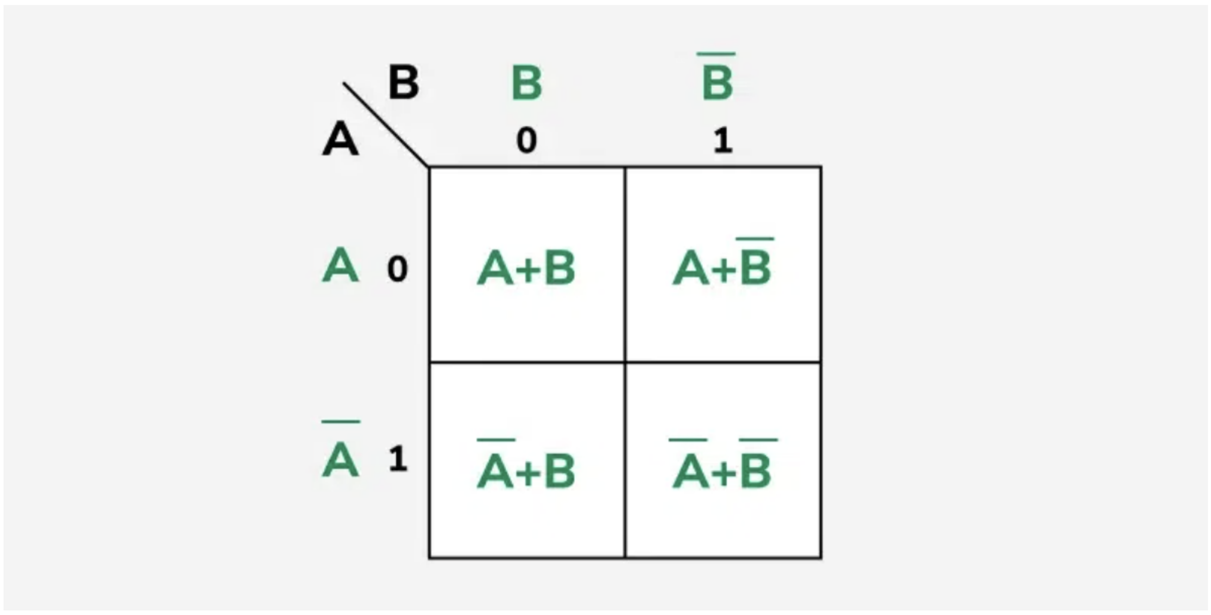
Put 1's in blocks of K-map respective to the minterms (0's elsewhere)

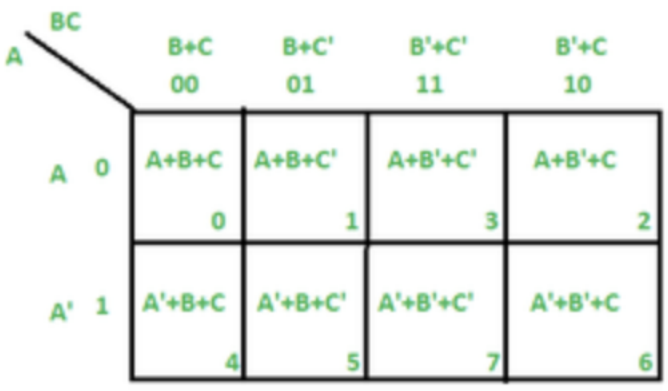
Make rectangular groups of 1 blocks containing total terms in power of two like 2,4,8 and try to cover as many elements as you can in one group

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

9.1.2 POS Solution

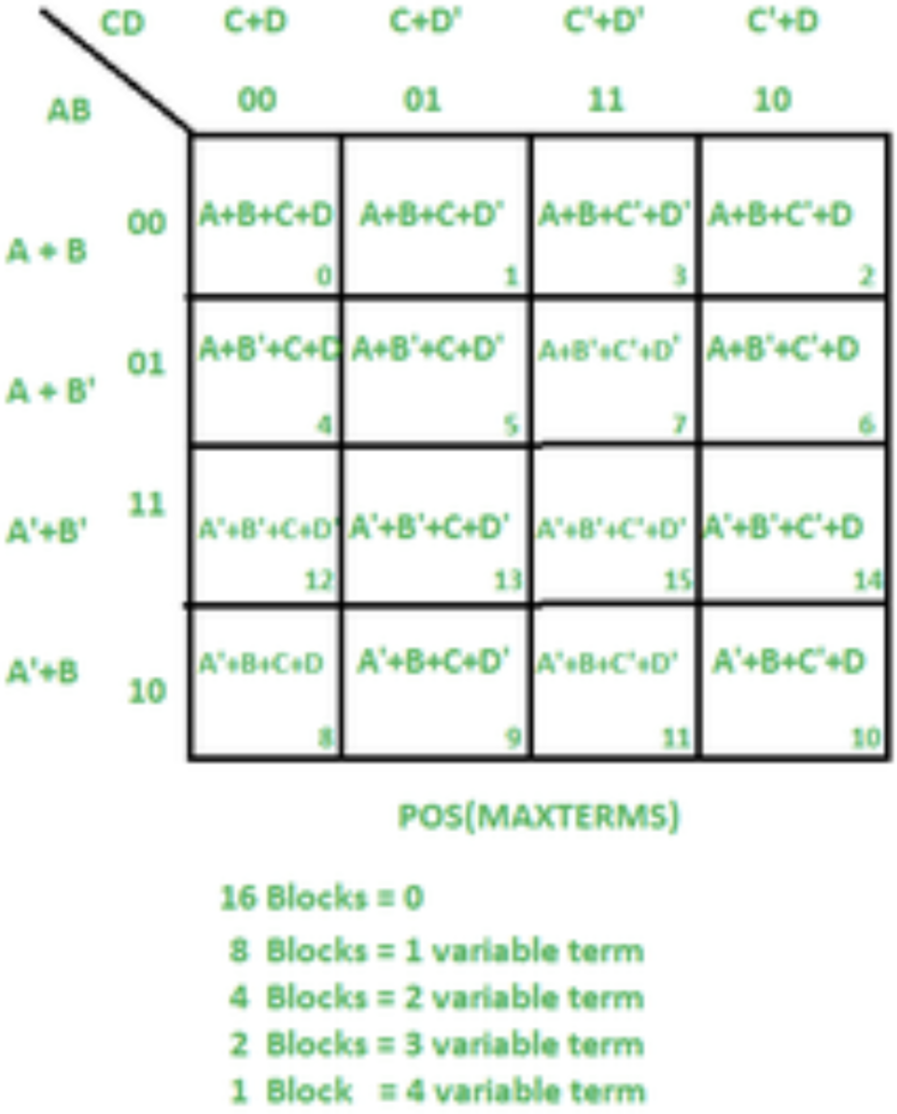
Select K-map according to number of variables





POS (MAXTERMS)

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



AB \ CD	C+D 00	C+D' 01	C'+D' 11	C'+D 10
A + B 00	A+B+C+D 0	A+B+C+D' 1	A+B+C'+D' 3	A+B+C'+D 2
A + B' 01	A+B'+C+D 4	A+B'+C+D' 5	A+B'+C'+D' 7	A+B'+C'+D 6
A'+B' 11	A'+B'+C+D 12	A'+B'+C+D' 13	A'+B'+C'+D' 15	A'+B'+C'+D 14
A'+B 10	A'+B+C+D 8	A'+B+C+D' 9	A'+B+C'+D' 11	A'+B+C'+D 10

POS(MAXTERMS)

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

#ECE

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

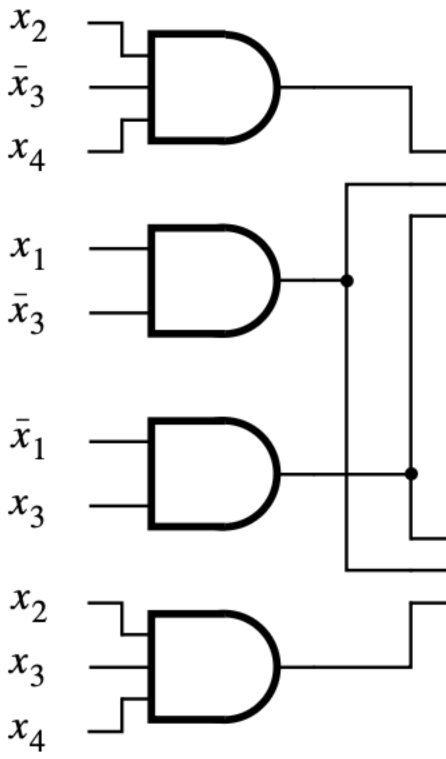
We can treat "d" in a k-map as either 0 or 1, which is useful when trying to find a minimum-cost function

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 $f_1 = x_1\overline{x_3} + \overline{x_1}x_3 + x_2x_3x_4$

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 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).

#ECE

11 VHDL Examples

Use `ieee.std'logic'` library for basic logic data types

ENTITY declares the module's interface (i/o), and `func1` takes three inputs, produces one input

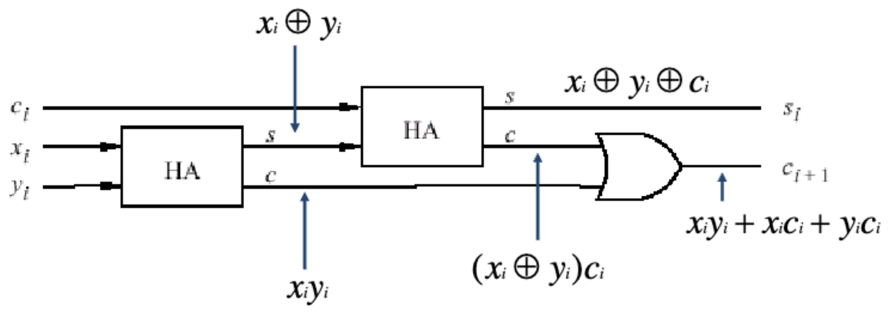
Expression: f is a SOP equation showing minimized logic

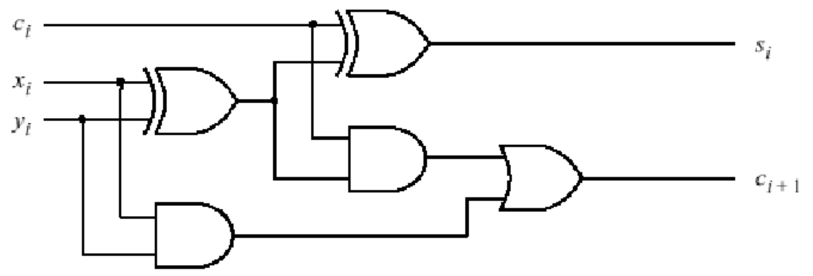
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

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.

11.1 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;
```

Focus on declaring an ENTITY, and the LogicFunc. Expecting proper syntax

#ECE

12 Number Representation in Digital Systems

12.1 Addition of Unsigned Numbers

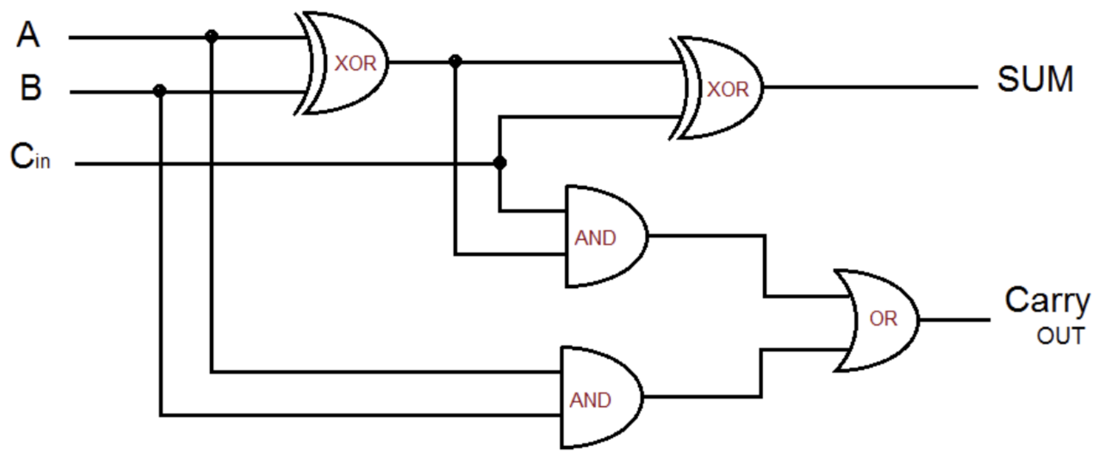
Unsigned numbers: only positive values

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**.

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

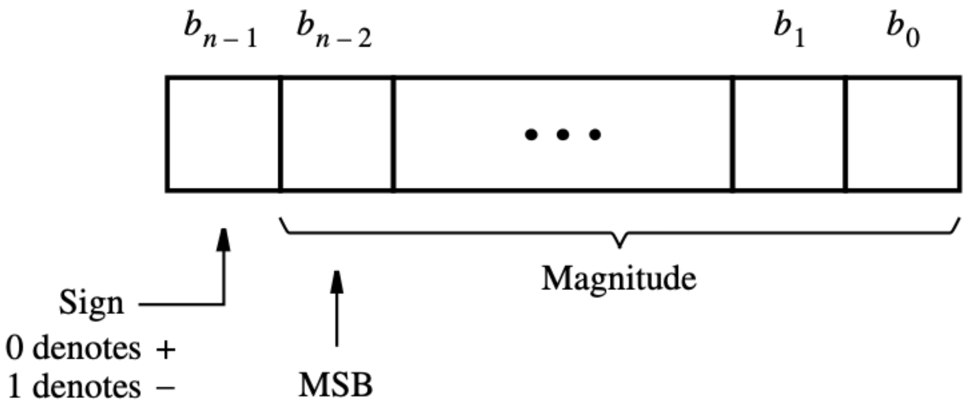


When we add multi-bit binary numbers, we need to combine several full-adders. A **ripple-carry adder** connects a series of full-adders together:

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

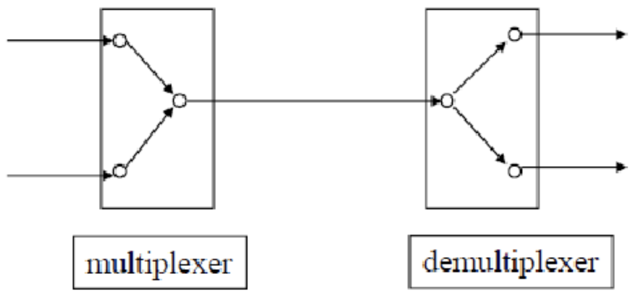
```
Carry in: 0 1 1 1
          | | | |
          0 1 1 1  (7)
        + 0 0 1 1  (3)
        -----------
          1 0 1 0  (sum)
```

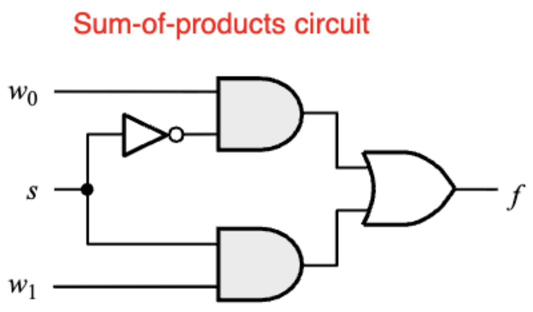
The carry *into sign bit* is 1, but the carry *out from sign bit* is 0. So, since these carries are different, overflow occurred.

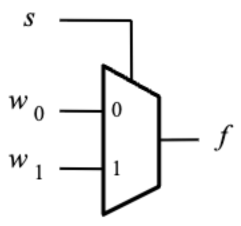
#ECE

13 10-21 Combinational Circuit Blocks

13.1 Multiplexer Review

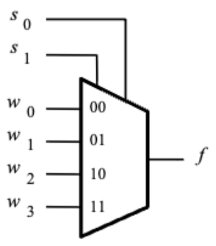






Graphical symbol

2-to-1 multiplexer. The graphical symbol represents the circuit.

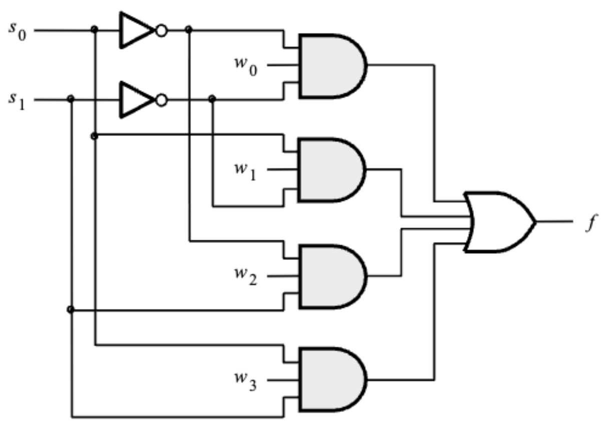


Graphical symbol

$$f = \overline{s_1}\,\overline{s_0}\,w_0 + \overline{s_1}\,s_0\,w_1 + s_1\,\overline{s_0}\,w_2 + s_1\,s_0\,w_3$$

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

Compact Truth table

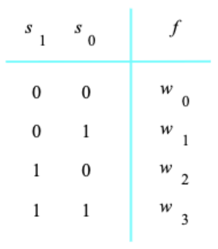


circuit

4-to-1 multiplexer. The amount of bits to represent all inputs is equal to the amount of select signals. Ex: w_8 needs three bits, thus 3 select signals.

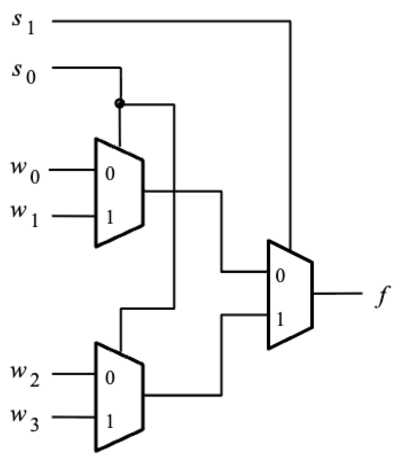
Neatly define the fan-in issue.

- Cascading Muxes: to build larger multiplexers



s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

4-to-1 Mux Truth table

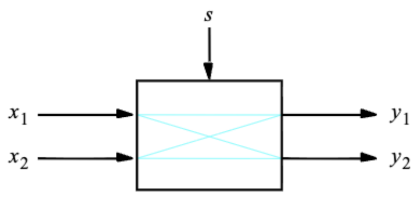


Using 2-to-1 multiplexers to build a 4-to-1 multiplexer

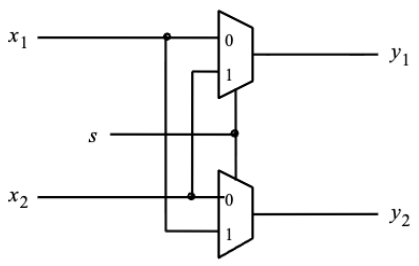
We can use multiplexers as building blocks to make even bigger multiplexers.

13.2 Applications of Multiplexers

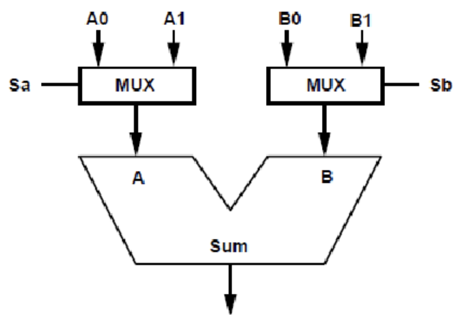
Crossbar switch $n \times k$, where n is not necessarily equal to k



2x2 crossbar switch



Implementation using multiplexers



An adder with selectable input operands

13.3 Synthesis of logic functions using Muxes

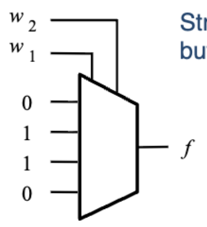
XOR with multiplexer

❑ Synthesis of logic functions using Muxes:

- A 2-input XOR

$$f = w_1 \oplus w_2$$

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0



Straightforward, but not very efficient

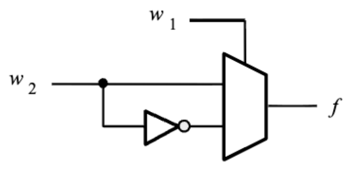
Implementation using a 4-to-1 multiplexer

Consider w_1 as the select signal for the Mux

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\overline{w_2}$

Modified truth table



Circuit

Tie the first example above to a LUT. Storing a result (cheating) and selecting the result by the inputs.

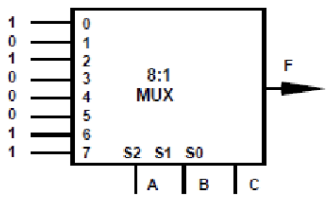
For the second example above: $w_1 = 0 \Rightarrow w_2 = f$ $w_1 = 1 \Rightarrow \overline{w_2} = f$

Explain: A 2^{n-1}-to-1 Mux can implement any function of n variables $n-1$ select signals: the remaining is a data input to the Mux

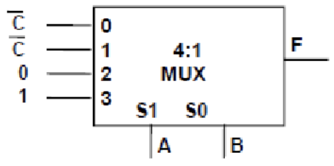
❑ Example:

$$f(A,B,C) = \sum m(0,2,6,7) = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + AB\overline{C} + ABC$$

$$f(A,B,C) = \overline{A}\overline{B}(\overline{C}) + \overline{A}B(\overline{C}) + A\overline{B}(0) + AB(1)$$



A	B	C	F	
0	0	0	1	$\overline{C}$
0	0	1	0	
0	1	0	1	$\overline{C}$
0	1	1	0	
1	0	0	0	0
1	0	1	0	
1	1	0	1	1
1	1	1	1	



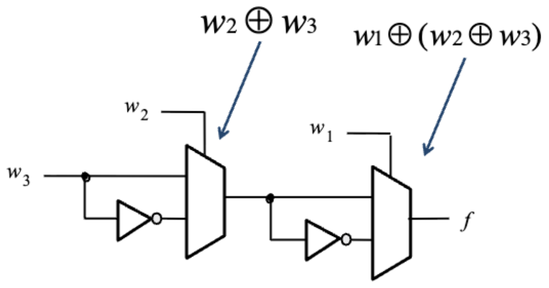
A and B as select signals

Tie back equation to K map

- Synthesis of logic functions using Muxes (cont'd):
  - A 3-input XOR using 2-to-1 Muxes

Truth table

w_1	w_2	w_3	f	
0	0	0	0	$w_2 \oplus w_3$
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	$\overline{w_2 \oplus w_3}$
1	0	1	0	
1	1	0	0	
1	1	1	1	



Circuit

With $w_1 = 0$, f is equal to XOR of w_2 and w_3.

With $w_1 = 1$, f is equal to XNOR of w_2 and w_3.

Find XOR and XNOR pattern in truth table

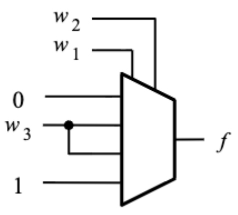
- Synthesis of logic functions using Muxes (cont'd):
  - A 3-input majority function using a 4-to-1 Mux

Any two inputs can be chosen as the Mux select inputs.

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Modified truth table

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1



Circuit

ended at slide 14

#ECE

14 ELEC 271 Digital Systems Cheatsheet

14.1 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

14.2 Logic Circuits

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

14.3 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

14.4 Multiplexer & VHDL Basics

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

14.5 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

14.6 CMOS Logic Gates

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

14.7 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

14.8 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

14.9 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

14.10 VHDL Example

```vhdl
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;
```

14.11 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.