Basics of Digital Logic

Boolean Algebra and Logic Gates

- This is the foundation for design and analysis of digital systems. It deals with the case where variables assume only one of two values: TRUE (usually represented by the symbol '**1**'), and FALSE (usually represented by the symbol '**0**').

Basic Operations

- X and Y: Boolean variables. Boolean variables are used to represent the inputs or outputs of a digital circuit. These three are the basic logical operations. All the other operations are derived from these three.

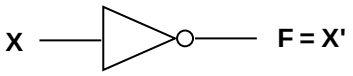
Operation	Boolean Expression	Operation
NOT	$X'(\ or\ \bar{X}\ )$	Logical negation
AND	$X.Y$	Logical conjunction of two statements
OR	$X+Y$	Logical disjunction of two statements

Truth Tables and Logic gates

- **Truth Table**: A tabular listing of function values for all possible combinations of values on its input arguments. If there are n inputs, there are 2^n possible combinations.
- **Logic Gates**: Hardware components that produce a logic 1 or logic 0 depending on the state of inputs. Boolean functions can be implemented with logic gates.

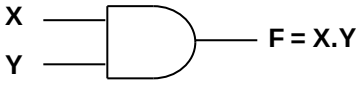
NOT gate:

X	F = X'
0	1
1	0



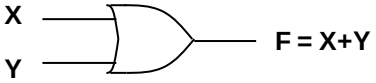
2-input AND gate:

X	Y	F = X.Y
0	0	0
0	1	0
1	0	0
1	1	1

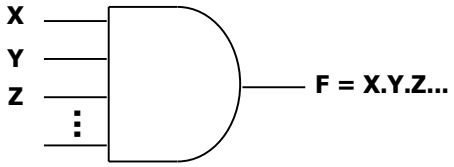


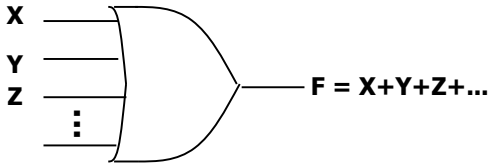
2-input OR gate:

X	Y	F = X+Y
0	0	0
0	1	1
1	0	1
1	1	1



- Logic Gates (AND, OR, etc.) can have multiple inputs:





AXIOMS

0.0 = 0	1.1 = 1	0.1 = 1.0 = 0	$\bar{0} = 1$
1+1=1	0+0 = 0	1+0 = 0+1 = 1	$\bar{1} = 0$

THEOREMS

Variable dominant rule	$X.1 = X$	$X + 0 = X$
Commutative rule	$X.Y = Y.X$	$X + Y = Y + X$
Complement rule	$X.\bar{X} = 0$	$X + \bar{X} = 1$
Idempotency	$X.X = X$	$X + X = X$
Identity Element	$X.0 = 0$	$X + 1 = 1$
Double negation	$\bar{\bar{X}} = X$	
Associative rule	$X.(Y.Z) = (X.Y).Z$	$X + (Y + Z) = (X + Y) + Z$
Distributive rule	$X.(Y + Z) = X.Y + X.Z$	$X + Y.Z = (X + Y).(X + Z)$

Other Theorems

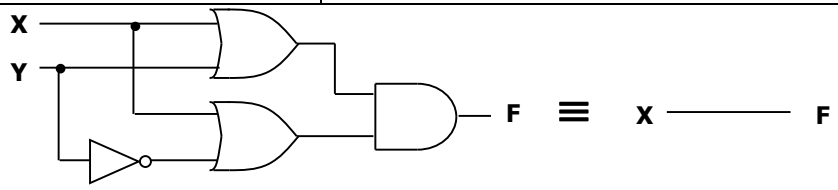
Absorption	$X.(X+Y) = X.X + X.Y = X + X.Y = X.(1+Y) = X$ $X + X.Y = X.(1+Y) = X$
Adjacency	$X.Y + X.\bar{Y} = X$ $(X+Y)(X+\bar{Y}) = X$
Consensus	$X.Y + \bar{X}Z + YZ = XY + \bar{X}Z$ $(X+Y)(\bar{X}+Z)(Y+Z) = (X+Y)(\bar{X}+Z)$ Corollary: $(X+Y)(\bar{X}+Z) = \bar{X}Y + XZ$
DeMorgan	$\overline{X.Y} = \bar{X} + \bar{Y}$, $\quad \overline{X.Y.Z\ldots} = \bar{X} + \bar{Y} + \bar{Z} + \cdots$ $\overline{X+Y} = \bar{X}.\bar{Y}$, $\quad \overline{X+Y+Z+\ldots} = \bar{X}.\bar{Y}.\bar{Z}\ldots$
Simplification	$X.(\bar{X}+Y) = X.Y$ $X + \bar{X}Y = X + Y$

- A useful application of the theorems is on the simplification of Boolean functions which leads to the reduction of the amount of logic gates. For example:

$F = (A + \bar{B}C + D + EF)(A + \bar{B}C + \overline{D + E\bar{F}})$
$F = (X+Y)(X+\bar{Y}), \quad X = A + \bar{B}C, \quad Y = D + EF$
$F = (X+Y)(X+\bar{Y}) = X$
$\rightarrow F = A + \bar{B}C$

$F = \overline{(\overline{X+Y})Z} + X\bar{Y}Z$
$F = \bar{X}\bar{Y}Z + X\bar{Y}Z = \bar{Y}Z(X + \bar{X})$
$\rightarrow F = \bar{Y}Z = Y + \bar{Z}$

$F = (X+Y)(X+\bar{Y})$
$F = XX + X\bar{Y} + YX + Y\bar{Y}$
$F = X + X(\bar{Y}+Y) = X + X$
$\rightarrow F = X$



$F = x_1x_2 + \overline{x_1}\,\overline{x_2} + x_1\overline{x_2}$
$F = x_1x_2 + \overline{x_1}(x_2 + \overline{x_2}) = x_1x_2 + \overline{x_1}$
$F = \overline{x_1} + x_1x_2 = (\overline{x_1} + x_1)(\overline{x_1} + x_2)$
$\rightarrow F = \overline{x_1} + x_2$

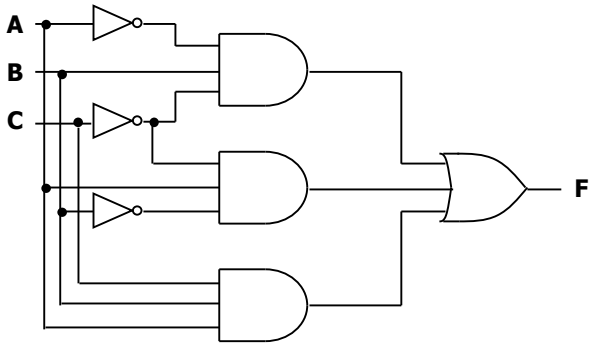
$F = \overline{A(B + \bar{C}) + \bar{A}}$
$F = \overline{A(B+\bar{C})}.A = (\bar{A} + \overline{B + \bar{C}}).A$
$\rightarrow F = (\overline{B + \bar{C}}).A = A\bar{B}C$

DERIVING BOOLEAN FUNCTIONS FROM TRUTH TABLES:

Using 1s:

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

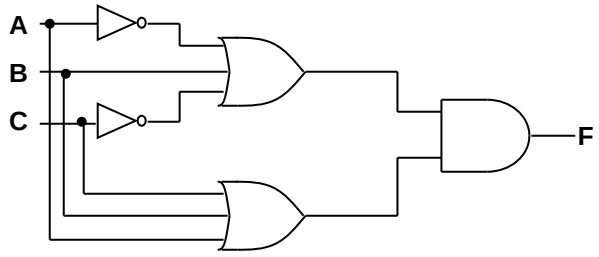
$F = \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$



Using 0s:

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

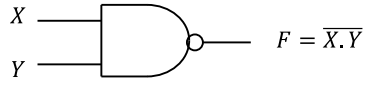
$F = (A + B + C)(\bar{A} + B + \bar{C})$



Other Logic Gates

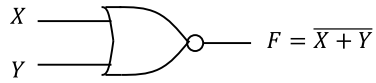
2-input NAND gate

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



2-input NOR gate

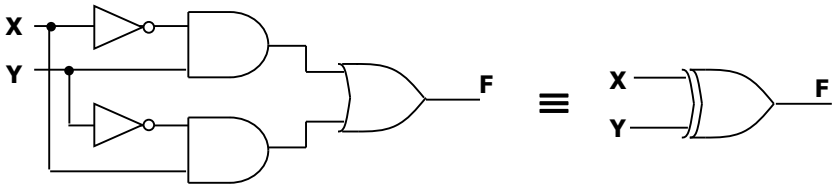
A	B	F
0	0	1
0	1	0
1	0	0
1	1	0



2-input XOR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

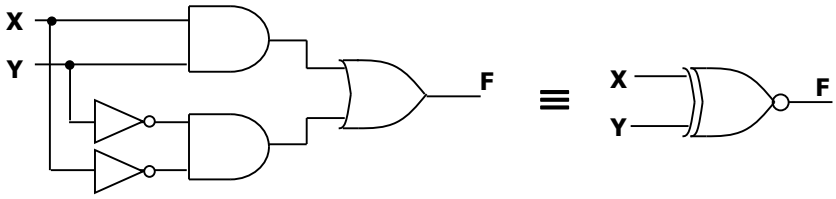
$F = \bar{X}Y + X\bar{Y} = X \oplus Y$



2-input XNOR gate:

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

$F = XY + \bar{X}\bar{Y} = \overline{X \oplus Y}$



SUM OF PRODUCTS (SOP) AND PRODUCT OF SUMS (POS) USING MINTERMS AND MAXTERMS:

MINTERMS and MAXTERMS (3 variable function)

	x_1	x_2	x_3	Minterms	Maxterms
0	0	0	0	$m_0 = \overline{x_1}\,\overline{x_2}\,\overline{x_3}$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x_1}\,\overline{x_2}x_3$	$M_1 = x_1 + x_2 + \overline{x_3}$
2	0	1	0	$m_2 = \overline{x_1}\,x_2\overline{x_3}$	$M_2 = x_1 + \overline{x_2} + x_3$
3	0	1	1	$m_3 = \overline{x_1}\,x_2x_3$	$M_3 = x_1 + \overline{x_2} + \overline{x_3}$
4	1	0	0	$m_4 = x_1\overline{x_2}\,\overline{x_3}$	$M_4 = \overline{x_1} + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x_2}x_3$	$M_5 = \overline{x_1} + x_2 + \overline{x_3}$
6	1	1	0	$m_6 = x_1x_2\overline{x_3}$	$M_6 = \overline{x_1} + \overline{x_2} + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x_1} + \overline{x_2} + \overline{x_3}$

- For a function with n variables, there are 2^n minterms (or 2^n maxterms) from m_0 to m_{2^n-1} (or from M_0 to M_{2^n-1})
- Note that: $\overline{m_i} = M_i$.

- A function can be expressed as a sum of minterms or as a product of maxterms:
  - ✓ A minterm can be 1 or 0. When the minterm is 1, the minterm is a term of the function.
  - ✓ A maxterm can be 1 or 0. When the maxterm is 0, the maxterm is a term of the function.

- Canonical Forms: Sum of products (SOP) that includes only minterms or a Product of sums (POS) containing only maxterms.

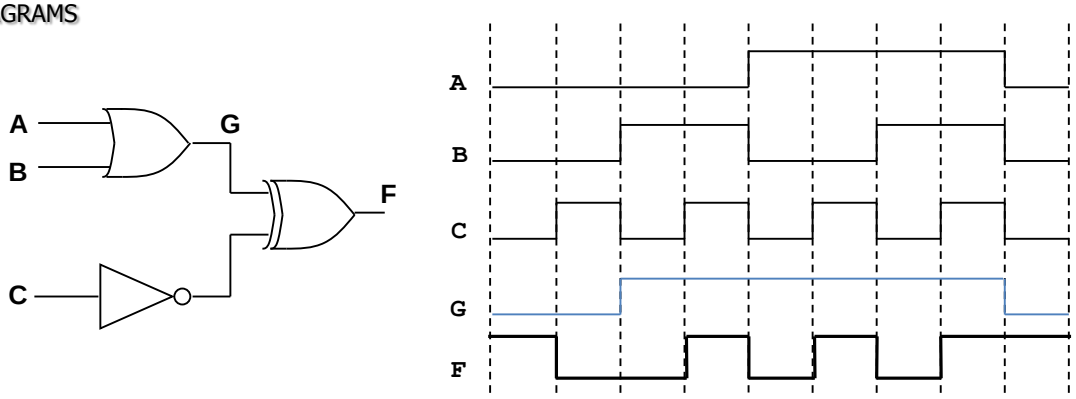
- Non-canonical Forms: SOP that includes terms that are not minterms (or a POS that includes terms that are not maxterms). For example:
  - ✓ $F(x_1, x_2, x_3) = x_1x_2x_3 + \overline{x_1}\,\overline{x_2}$
  - ✓ $F(x_1, x_2, x_3) = (x_1 + x_2 + x_3)(\overline{x_1} + \overline{x_2})$
  - ✓ $F(x_1, x_2, x_3) = x_1x_2x_3 + x_1\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + (\overline{x_1} + x_2 + x_3)$

Example:

X	Y	Z	F	Sum of Products
0	0	0	0	$F = \bar{X}\bar{Y}Z + X\bar{Y}\bar{Z} + X\bar{Y}Z + XY\bar{Z}$
0	0	1	1	$F(X,Y,Z) = \sum(m_1, m_4, m_5, m_6).$
0	1	0	0	$F(X,Y,Z) = \sum m(1,4,5,6)$ Also: $\bar{F}(X,Y,Z) = \sum m(0,2,3,7)$
0	1	1	0	
1	0	0	1	Product of Sums
1	0	1	1	$F = (X+Y+Z)(X+\bar{Y}+Z)(X+\bar{Y}+\bar{Z})(\bar{X}+\bar{Y}+\bar{Z})$
1	1	0	1	$F(X,Y,Z) = \prod(M_0, M_2, M_3, M_7).$
1	1	1	0	$F(X,Y,Z) = \prod M(0,2,3,7)$ Also: $\bar{F}(X,Y,Z) = \prod M(1,4,5,6)$

- Note: $F(X,Y,Z) = \sum m(1,4,5,6) = \prod M(0,2,3,7)$.

TIMING DIAGRAMS



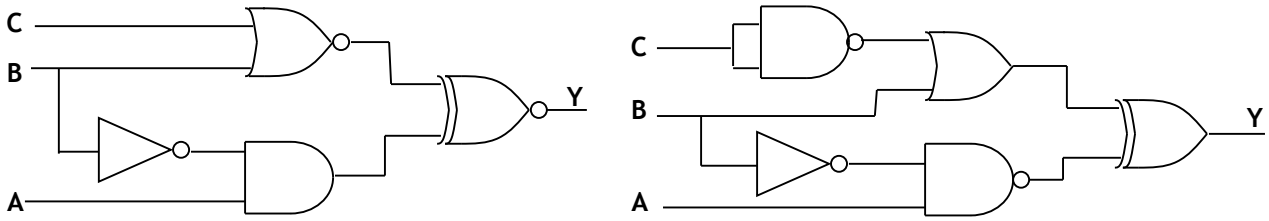
PRACTICE EXERCISES

- Simplify the following functions:
  - ✓ $F = \bar{X}\bar{Y}Z + X\bar{Y}\bar{Z} + X\bar{Y}Z + XY\bar{Z}$
  - ✓ $F(X,Y,Z) = \sum(m_0, m_2, m_6)$
  - ✓ $F(X,Y,Z) = \prod(M_3, M_4, M_7)$
  - ✓ $F = (X+Y+Z)(X+Y+\bar{Z})$
  - ✓ $F = (\bar{A}B + C + D)(\bar{A}B + D)$
  - ✓ $F = A(C + \bar{D}B) + \bar{A}$

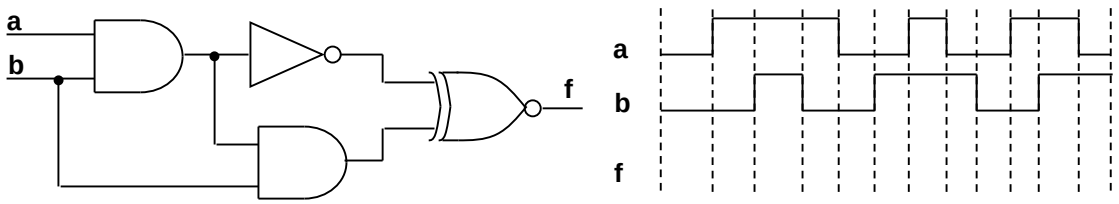
- Provide the Boolean functions and sketch the logic circuit. Use the two representations: i) Sum of Products, ii) Product of Sums. Also, provide the minterms and maxterms representations.

A	B	C	F1	F2	F3	F4	F5	F6	F7
0	0	0	0	1	0	1	0	1	0
0	0	1	1	0	1	1	1	0	0
0	1	0	0	0	1	1	0	1	1
0	1	1	1	0	1	1	1	1	1
1	0	0	1	0	1	0	0	1	0
1	0	1	0	1	0	0	1	0	0
1	1	0	1	1	0	0	0	1	1
1	1	1	1	1	1	0	1	0	1

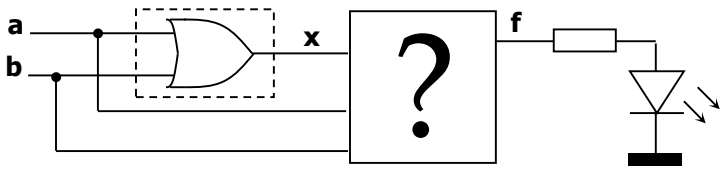
- Obtain the logic function (and minimize if possible) of the following circuits:



- Draw the timing diagram of the following circuit:

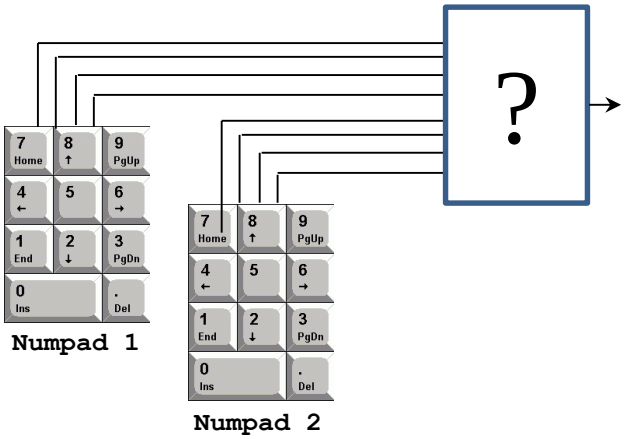


- Design a circuit that verifies the logical operation of the OR gate. f = '1' (LED ON) if the OR gate works properly. Assumption: when the OR gate is not working, it is generating 1's instead of 0's and vice versa. Tip: First, generate the truth table.



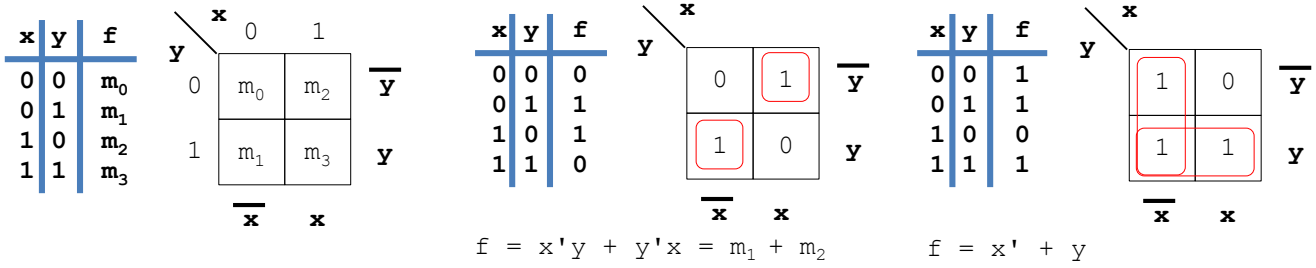
- Security combination: We have a lock that only opens when we set eight (8) switches as in the figure. Each switch represents a Boolean variable. Get the function that opens the lock (a logical '1' is generated) when the switches are configured as in the figure. Here, an open lock is represented by an LED that is ON.

- Design a logic circuit (simplify your circuit) that opens a lock (f='1') whenever one presses the correct number on each numpad. We encode each decimal number on the numpad using BCD encoding. We expect that each group of 4 bits be in the range from 0000 to 1001, the values from 1010 to 1111 are assumed not to occur.
  Tip: create two circuits: one that verifies the first number (**9**), and the other that verifies the second number (**5**). Then perform the AND operation on the two outputs. This avoids creating a truth table with 8 inputs!

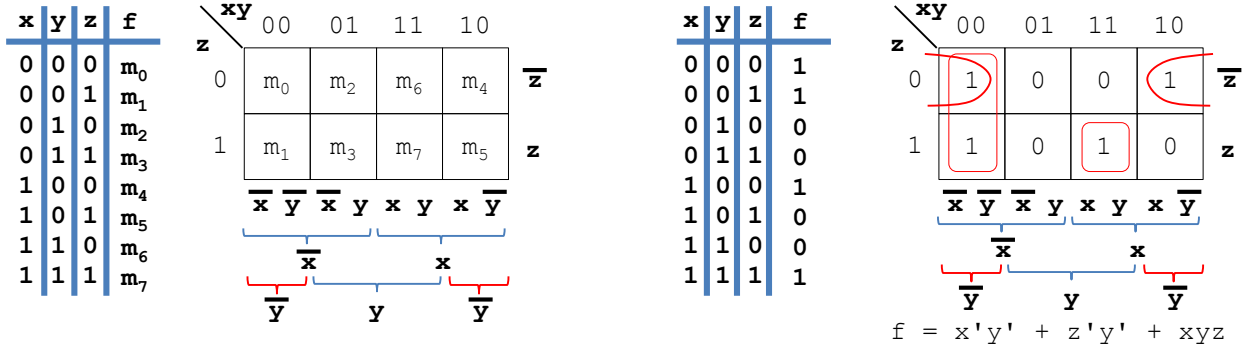


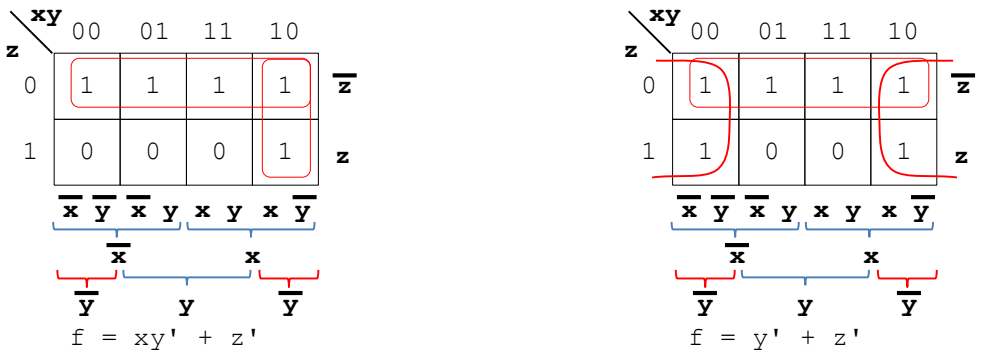
SIMPLIFICATION OF FUNCTIONS USING KARNAUGH MAPS

2 variables:



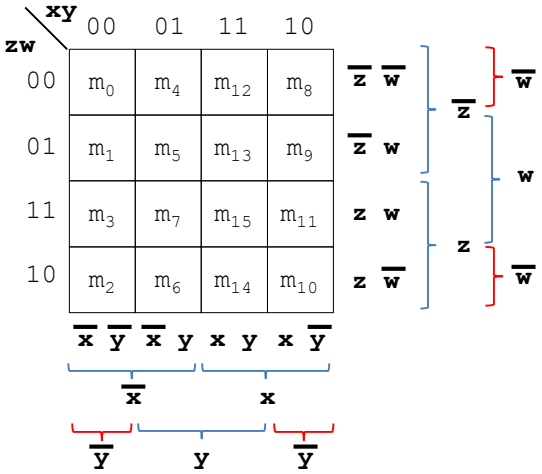
3 variables:

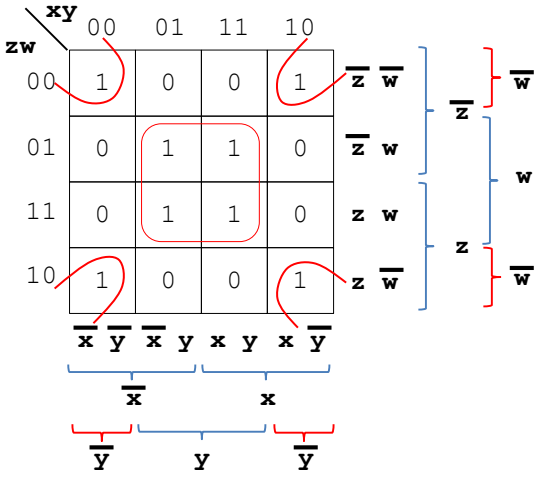




4 variables:

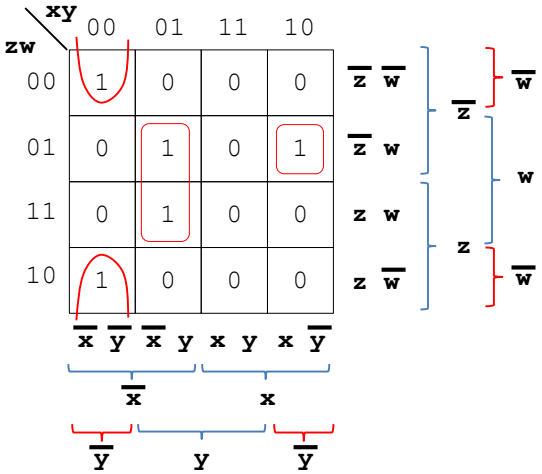
x	y	z	w	f	
0	0	0	0		m_0
0	0	0	1		m_1
0	0	1	0		m_2
0	0	1	1		m_3
0	1	0	0		m_4
0	1	0	1		m_5
0	1	1	0		m_6
0	1	1	1		m_7
1	0	0	0		m_8
1	0	0	1		m_9
1	0	1	0		m_{10}
1	0	1	1		m_{11}
1	1	0	0		m_{12}
1	1	0	1		m_{13}
1	1	1	0		m_{14}
1	1	1	1		m_{15}



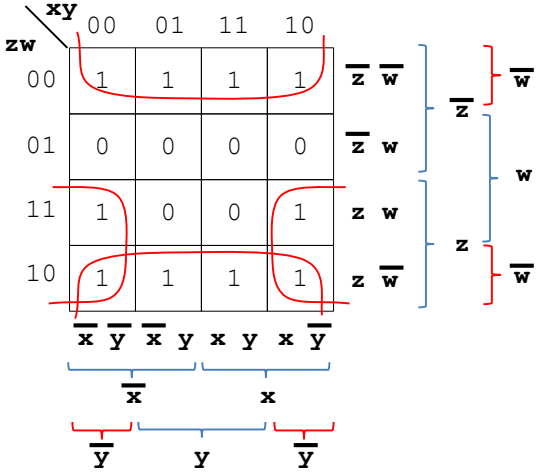


f = y'w' + wy

x	y	z	w	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

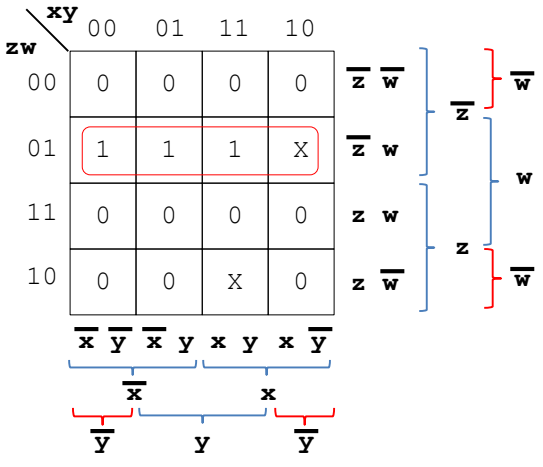


f = x'y'w' + x'yw + xy'z'w

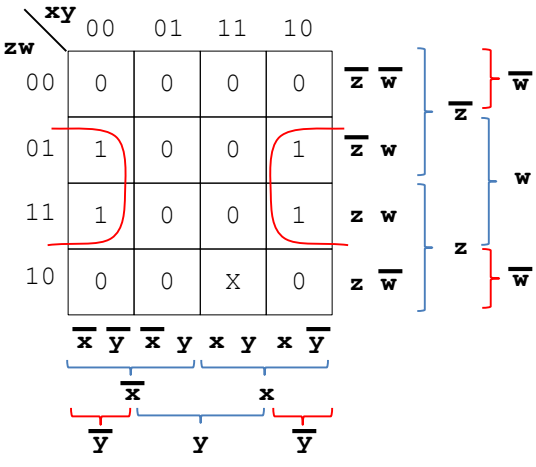


f = y'z + w'

Don't care outputs



f = z'w



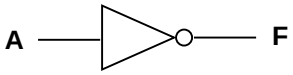
f = wy'

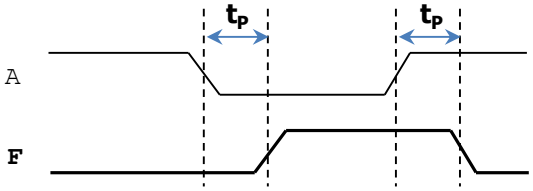
PRACTICAL ASPECTS

- Digital circuits are analog circuits!

PROPAGATION DELAY

- t_P: Propagation delay.

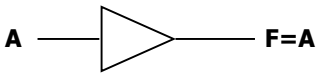


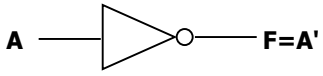


TRI-STATE BUFFERS

Buffers:

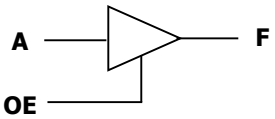
- They can drive more current (e.g.: motors, high-power LEDs) than simple logic gates. A common implementation uses OPAMPs.



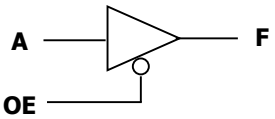


Tri-state Buffers:

- 'Z' State: This is high impedance, which effectively means that F is disconnected from A.



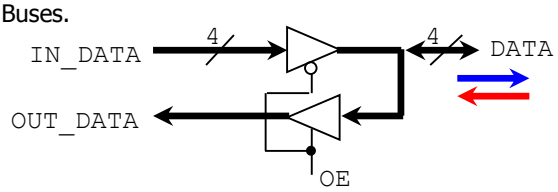
OE = 0 → F = Z
OE = 1 → F = A

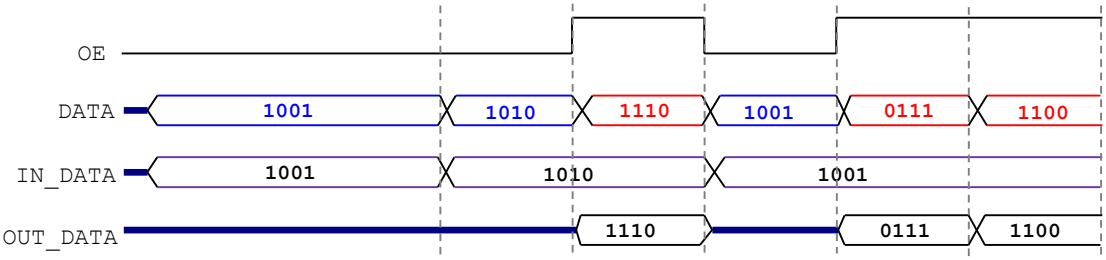


OE = 1 → F = Z
OE = 0 → F = A

Applications:

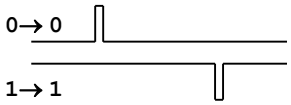
- Multiplexors, Bidirectional pins, Microprocessor Buses.
- Example: Bi-directional port (4 bits):



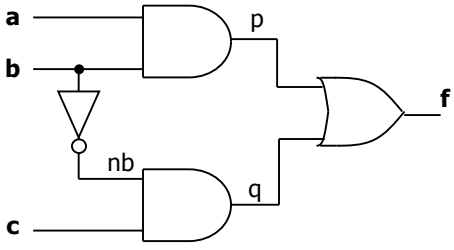


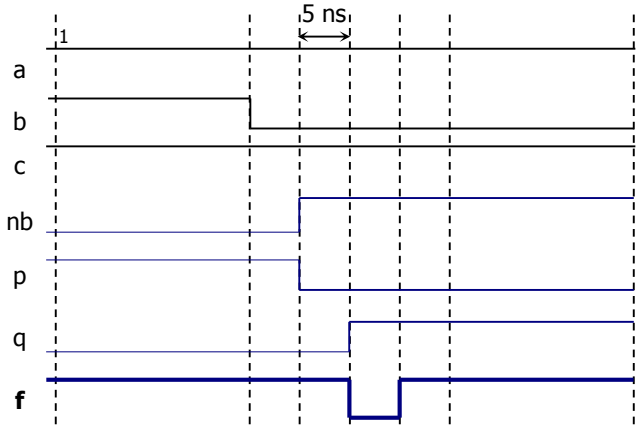
HAZARDS

- A digital circuit can generate glitches, which are fast "spikes", usually unwanted.
- Glitches caused by the propagation delays and/or the structure of the circuit are known as hazards.
- Two types of hazard exist:
  - **Static Hazards**: They occur when the propagation delays are unbalanced. It can be addressed by adding all prime implicants to a function. These hazards happen when inputs change, but the output is not supposed to change. Two types: 0 → 0, or 1 → 1.
    Example: All gates have a propagation delay of 5 ns.

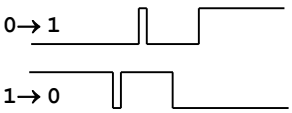


$$f = ab + \bar{b}c$$





- **Dynamic hazards**: They are caused by the structure of the circuit. They are difficult to detect and address. They usually occur in multilevel circuits. To avoid, use only two-level circuits and ensure that there are not static hazards. Two types: $1 \rightarrow 0$, or $0 \rightarrow 1$.

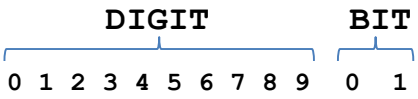


- **Significance of hazards**:
  - Asynchronous circuits: They are very vulnerable to hazards and will usually render the circuits unusable.
  - Synchronous circuits: Hazards do not pose a problem here, as we use registers to safely ignore hazards.
  - Combinational circuits: Hazards are usually not a problem because the outputs solely depend on the current inputs (as long as the duration between input changes is greater than the propagation delay, which is usually the case).

UNSIGNED INTEGER NUMBERS: BINARY REPRESENTATION

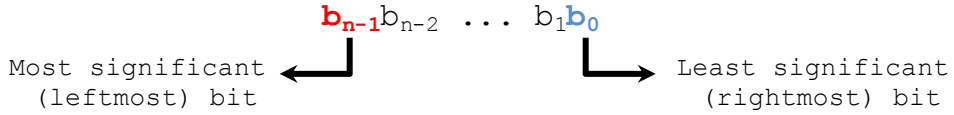
BINARY NUMBER SYSTEM

- Binary numbers are very practical as they are used by digital computers. For binary numbers, the counterpart of the decimal digit (that can take values from 0 to 9) is the binary digit, or bit (that can take the value of 0 or 1).
- **Bit**: Unit of information that a computer uses to process and retrieve data. It can also be used as a Boolean variable.
- **Binary number**: This is represented by a string of bits using the positional number representation: $b_{n-1}b_{n-2}\dots b_1b_0$



CONVERTING A BINARY NUMBER INTO A DECIMAL NUMBER:

- Positional number representation for a binary number with 'n' bits:



The binary number can be converted to a positive decimal number by using the following formula:

$$D = \sum_{i=0}^{i=n-1} b_i \times 2^i = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

- To avoid confusion, we usually write a binary number and attach a suffix '2': $(b_{n-1}b_{n-2}\dots b_1b_0)_2$
  Example: $6\ bits$: $(101011)_2 \equiv D = 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 43$
  $4\ bits$: $(1011)_2 \equiv D = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 11$

- Maximum value and range for a given number of bits:

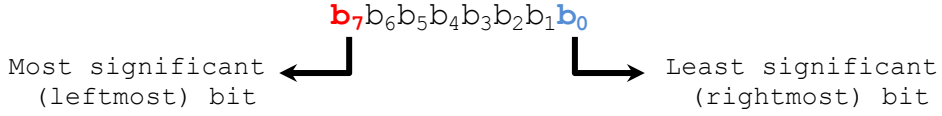
Number of bits	Maximum value	Range
1	$1_2 \equiv 2^1-1$	$0 \rightarrow 1_2 \equiv 0 \rightarrow 2^1-1$
2	$11_2 \equiv 2^2-1$	$0 \rightarrow 11_2 \equiv 0 \rightarrow 2^2-1$
3	$111_2 \equiv 2^3-1$	$0 \rightarrow 111_2 \equiv 0 \rightarrow 2^3-1$
4	$1111_2 \equiv 2^4-1$	$0 \rightarrow 1111_2 \equiv 0 \rightarrow 2^4-1$
…		
n	$111\dots111_2 \equiv 2^n-1$	$0 \rightarrow 111\dots111_2 \equiv 0 \rightarrow 2^n-1$

- **Maximum value for 'n' bits:** The maximum binary number is given by an n-bit string of 1's: 111…111. Then, the maximum decimal number is given by:

$$D = \underbrace{111\dots111}_{n\ bits} = 1\times2^{n-1} + 1\times2^{n-2} + \dots + 1\times2^1 + 1\times2^0 = 2^n-1$$

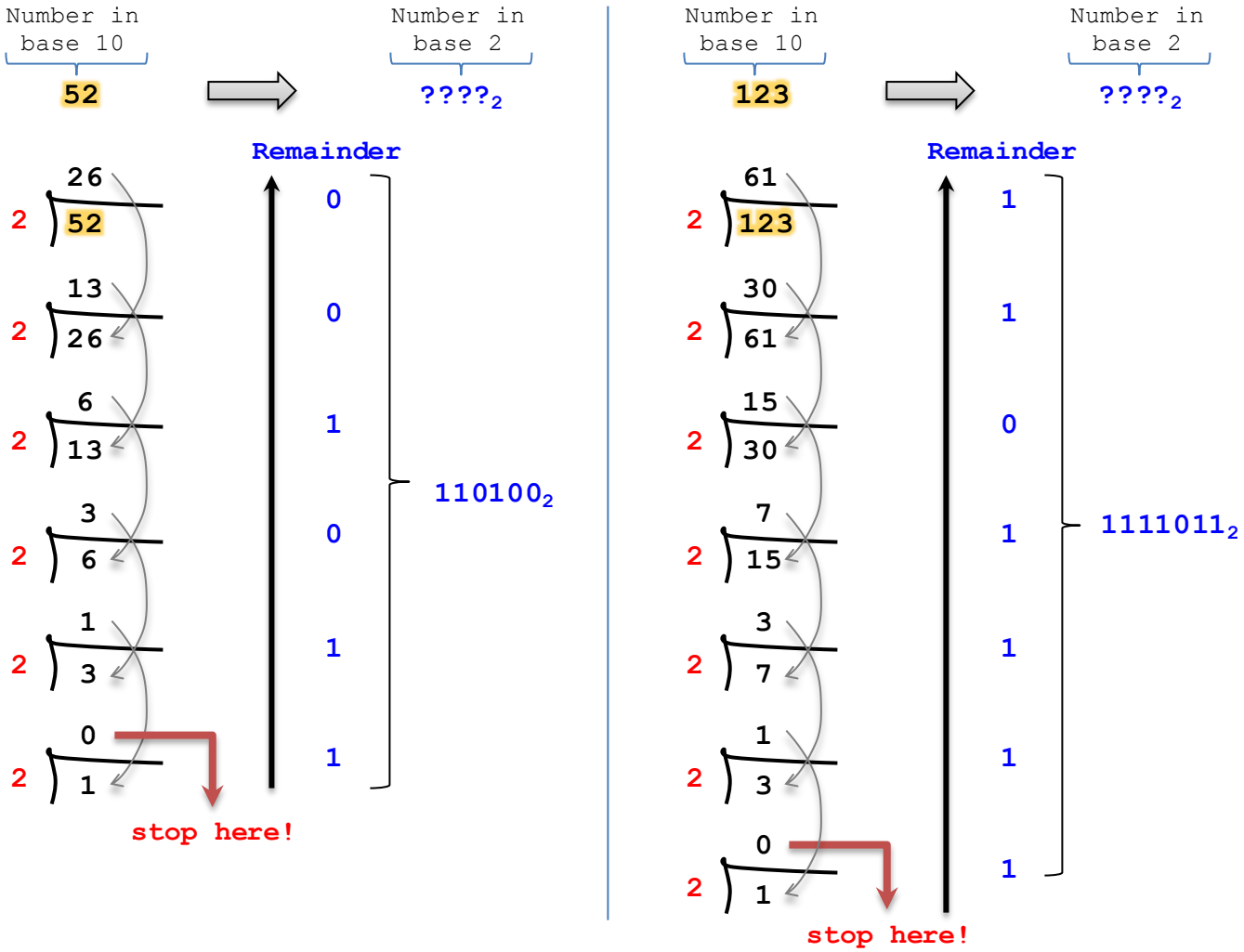
⇒ With 'n' bits, we can represent 2^n positive integer numbers from 0 to 2^n-1.

- The case **n=8** bits is of particular interest, as a string of 8 bits is called a byte. For 8-bit numbers, we have 256 numbers in the range 0 to 2^8-1 ≡ 0 to 255.



CONVERTING A DECIMAL NUMBER (INTEGER POSITIVE) INTO A BINARY NUMBER

- Examples:



- Note that some numbers require fewer bits than others. If we want to use a specific bit representation, e.g., 8-bit, we just need to append zeros to the left until the 8 bits are completed. For example:

  $110100_2 \equiv 00110100_2$ (8-bit number)

  $1111011_2 \equiv 01111011_2$ (8-bit number)

CONVERSION OF A NUMBER IN ANY BASE INTO A DECIMAL NUMBER

- To convert a number of base 'r' (r = 2, 3,4 ,…) to decimal, we use the following formula:
  Number in base 'r': $(r_{n-1}r_{n-2}\ldots r_1r_0)_r$

$$D = \sum_{i=0}^{i=n-1} r_i \times r^i = r_{n-1} \times r^{n-1} + r_{n-2} \times r^{n-2} + \cdots + r_1 \times r^1 + r_0 \times r^0$$

  Also, the maximum decimal value for a number in base 'r' with 'n' digits is:

$$D = rrr\ldots rrr = r \times r^{n-1} + r_{n-2} \times r^{n-2} + \cdots + r \times r^1 + r \times r^0 = r^n - 1$$

- Example: Base-8:

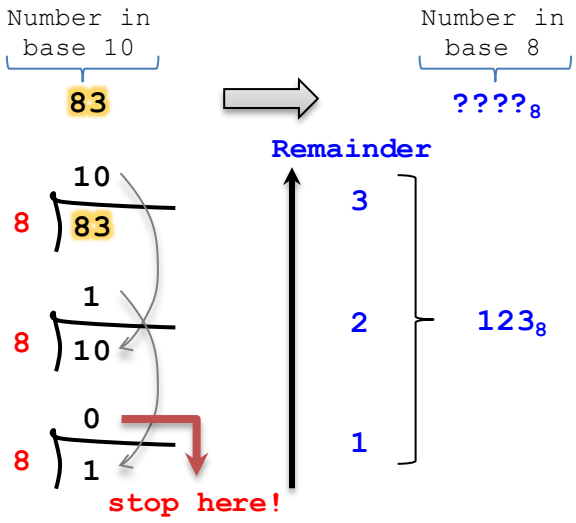
Number of digits	Maximum value	Range
1	$7_8 \equiv 8^1-1$	$0 \rightarrow 7_8 \equiv 0 \rightarrow 8^1-1$
2	$77_8 \equiv 8^2-1$	$0 \rightarrow 77_8 \equiv 0 \rightarrow 8^2-1$
3	$777_8 \equiv 8^3-1$	$0 \rightarrow 777_8 \equiv 0 \rightarrow 8^3-1$
…		
n	$777\ldots777_8 \equiv 8^n-1$	$0 \rightarrow 777\ldots777_8 \equiv 0 \rightarrow 8^n-1$

Examples:

- $(50632)_8$: Number in base 8 (octal system)
  Number of digits: n = 5. Conversion to decimal: $D = 5\times8^4 + 0\times8^3 + 6\times8^2 + 3\times8^1 + 2\times8^0 = 20890$
- $(3102)_4$: Number in base 4 (quaternary system)
  Number of digits: n = 4. Conversion to decimal: $D = 3\times4^3 + 1\times4^2 + 0\times4^1 + 2\times4^0 = 210$

CONVERTING A DECIMAL NUMBER (INTEGER POSITIVE) INTO A NUMBER IN ANY BASE

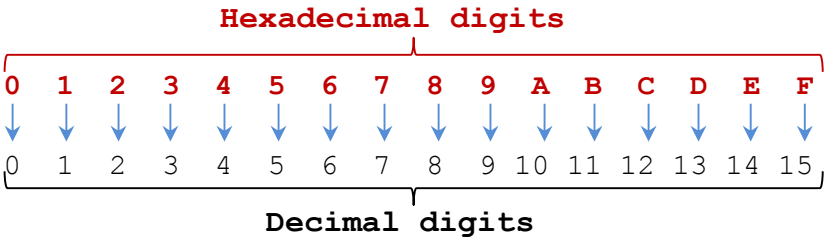
- This is a generalization of the method to convert a decimal number into a binary number. For example, if you want to convert it into a base-8 number, just divide by 8 and group the remainders.



- **Example**: Converting a decimal number to base-8:

HEXADECIMAL NUMBER SYSTEM

- This is a very useful system as it provides a short-hand notation for binary numbers.
- A hexadecimal digit (also called a *nibble*) can take a value from 0 to 15. To avoid confusion, the numbers 10 to 15 are represented by letter (A-F):



CONVERTING A HEXADECIMAL NUMBER INTO A DECIMAL NUMBER:

- Positional number representation for a hexadecimal number with 'n' nibbles (hexadecimal digits):

$h_{n-1}h_{n-2}\ \ldots\ h_1h_0$

Most significant (leftmost) nibble — Least significant (rightmost) nibble

- To convert a hexadecimal number into a decimal, we apply the following formula:

$$D = \sum_{i=0}^{i=n-1} h_i \times 16^i = h_{n-1}\times16^{n-1} + h_{n-2}\times16^{n-2} + \cdots + h_1\times16^1 + h_0\times16^0$$

- To avoid confusion, it is sometimes customary to append the prefix '0x' to a hexadecimal number:

$$0xh_{n-1}h_{n-2}\ldots h_1h_0$$

- Examples:

FD0A90: 0xFD0A90 $\equiv F\times16^5 + D\times16^4 + 0\times16^3 + A\times16^2 + 9\times16^1 + 0\times16^0$
$\equiv 15\times16^5 + 14\times16^4 + 0\times16^3 + 10\times16^2 + 9\times16^1 + 0\times16^0$
0B871C: 0x0B871C $\equiv 0\times16^5 + B\times16^4 + 8\times16^3 + 7\times16^2 + 1\times16^1 + C\times16^0$

- The table presents the maximum attainable value for the given number of nibbles (hexadecimal digits).

Number of nibbles	Maximum value	Range
1	F ≡ 16^1-1	0 → F ≡ 0 → 16^1-1
2	FF ≡ 16^2-1	0 → FF ≡ 0 → 16^2-1
3	FFF ≡ 16^3-1	0 → FFF ≡ 0 → 16^3-1
4	FFFF ≡ 16^4-1	0 → FFFF ≡ 0 → 16^4-1
…		
n	FFF…FFF ≡ 16^n-1	0 → FFF…FFF ≡ 0 → 16^n-1

- **Maximum value for 'n' nibbles:** The maximum decimal value with 'n' nibbles is given by:

$$D = \underbrace{FFF...FFF}_{n \text{ nibbles}} = \begin{matrix} F\times16^{n-1} + F\times16^{n-2} + \ldots + F\times16^{1} + F\times16^{0} \\ 15\times16^{n-1} + 15\times16^{n-2} + \ldots + 15\times16^{1} + 15\times16^{0} \end{matrix} = 16^{n}-1$$

⇒ With 'n' nibbles, we can represent positive integer numbers from 0 to 16^n-1. (16^n numbers)

UNITS OF INFORMATION

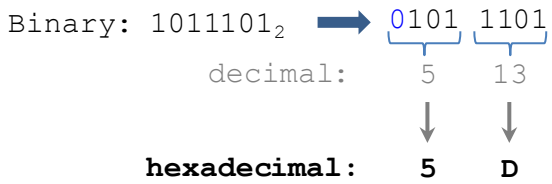
Nibble	Byte	KB	MB	GB	TB
4 bits	8 bits	2^{10} bytes	2^{20} bytes	2^{30} bytes	2^{40} bytes

- Note that the nibble (4 bits) is one hexadecimal digit. Also, one byte (8 bits) is represented by two hexadecimal digits.
- While KB, MB, GB, TB (and so on) should be powers of 10 in the International System, it is customary in digital jargon to use powers of 2 to represent them. In microprocessor systems, memory size is usually a power of 2 since it is determined by the number of addresses the address bus can handle (which is a power of 2). As a result, it is very useful to use the definition provided here for KB, MB, GB, TB (and so on).
- Digital computers usually represent numbers utilizing a number of bits that is a multiple of 8. The fast hexadecimal to binary conversion allows us to quickly convert a string of bits that is a multiple of 8 into a string of hexadecimals digits.
- The size of the data bus in a processor represents the computing capacity of a processor, as the data bus size is the number of bits the processor can operate in one operation (e.g.: 8-bit, 16-bit, 32-bit processor). This is also usually expressed as a number of bits that is a multiple of 8

CONVERTING BETWEEN HEXADECIMAL AND BINARY NUMBERS

- Conversions between hexadecimal and binary systems are commonplace when dealing with digital computers:
  - ✓ **Hexadecimal to binary:** We already know how to convert a hexadecimal number into a decimal number. We can then convert the decimal number into a binary number (using successive divisions).
  - ✓ **Binary to hexadecimal**: We can first convert the binary number to a decimal number. Then, using an algorithm similar to the one that converts decimals into binary, we can convert our decimal number into a hexadecimal number.
- These two conversion processes are too tedious. Fortunately, hexadecimal numbers have an interesting property that allows quick conversion of binary numbers to hexadecimals and viceversa:

- **Binary to hexadecimal**: We group the binary numbers in groups of 4 (starting from the rightmost bit). If the last group has fewer than four bits, we append zeros to the left. Then, we independently convert each 4-bit group to its decimal value. Note that 4 bits can only take decimal values between 0 and $2^4-1 \equiv 0$ to 15, hence 4 bits represent only one hexadecimal digit, i.e., a 4-bit group can represent up to 16 hexadecimal digits. The figure below shows an example.



Then: $01011101_2 = 0x5D$

Verification:

$01011101_2 = 1\times2^6 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^0 = 93$

$0x5D = 5\times16^1 + D\times16^0 = 93$

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

- **Hexadecimal to binary**: It is the reverse process of converting a binary into a hexadecimal numbers. We pick each hexadecimal digit and convert it (always using 4 bits) to its 4-bit binary representation. The binary number is the concatenation of all resulting 4-bit groups.

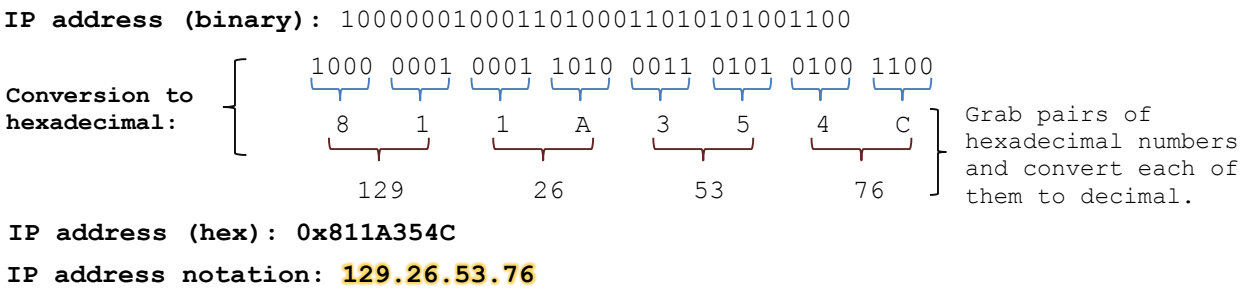
$0xFA = 11111010_2$
$0xC1 = 11000001_2$

DO NOT discard these zeros when concatening!

APPLICATIONS OF BINARY AND HEXADECIMAL REPRESENTATIONS

INTERNET PROTOCOL ADDRESS (IP ADDRESS):

- Hexadecimal numbers represent a compact way of representing binary numbers. The IP address is defined as a 32-bit number, but it is displayed as a concatenation of four decimal values separated by a dot (e.g., 129.26.53.76).
- The following figure shows how a 32-bit IP address expressed as a binary number is transformed into the standard IP address notation.



- The 32-bit IP address expressed as binary number is very difficult to read. So, we first convert the 32-bit binary number to a hexadecimal number.
- The IP address expressed as a hexadecimal (0x811A354C) is a compact representation of a 32-bit IP address. This should suffice. However, it was decided to represent the IP address in a *'human-readable'* notation. In this notation, we grab pairs of hexadecimal numbers and convert each of them individually to decimal numbers. Then we concatenate all the values and separate them by a dot.
- **Important**: Note that the IP address notation (decimal numbers) is NOT the decimal value of the binary number. It is rather a series of four decimal values, where each decimal value is obtained by independently converting each two hexadecimal digits to its decimal value.

  ✓ Given that each decimal number in the IP address can be represented by 2 hexadecimal digits (or 8 bits), what is the range (min. value, max. value) of each decimal number in the IP address?
  With 8 bits, we can represent $2^8 = 256$ numbers from 0 to 255.

  ✓ An IP address represents a unique device connected to the Internet. Given that the IP address has 32 bits (or 8 hexadecimal digits), the how many numbers can be represented (i.e., how many devices can connect to the Internet)?
  $2^{32} = 4294967296$ devices.

  ✓ The number of devices that can be connected to the Internet is huge, but considering the number of Internet-capable devices that exists in the entire world, it is becoming clear that 32 bits is not going to be enough. That is why the Internet Protocol is being currently extended to a new version (IPv6) that uses 128 bits for the addresses. With 128 bits, how many Internet-capable devices can be connected to the Internet?
  $2^{128} \approx 3.4 \times 10^{38}$ devices

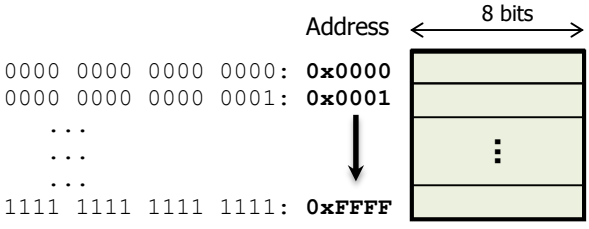
REPRESENTING GRAYSCALE PIXELS

A grayscale pixel is commonly represented with 8 bits. So, a grayscale pixel value varies between 0 and 255, 0 being the darkest (black) and 255 being the brightest (white). Any value in between represents a shade of gray.

MEMORY ADDRESSES

The address bus size in processors is usually determined by the number of memory positions it can address. For example, if we have a microprocessor with an address bus of 16 bits, we can handle up to 2^{16} addresses. If the memory content is one byte wide, then the processor can handle up to $2^{16} bytes = 64KB$.

Here, we use 16 bits per address, or 4 nibbles. The lowest address (in hex) is `0x0000` and highest address (in hex) is `0xFFFF`.



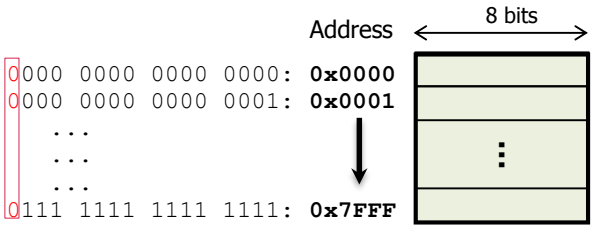
Examples:

- A microprocessor can only handle memory addresses from `0x0000` to `0x7FFF`. What is the address bus size? If each memory position is one byte wide, what is the maximum size (in bytes) of the memory that we can connect?

  We want to cover all the cases from `0x0000` to `0x7FFF`:

  The range from `0x0000` to `0x7FFF` is akin to all possible cases with 15 bits. Thus, the address bus size is **15 bits**.

  We can handle $2^{15} bytes = 32KB$ of memory.

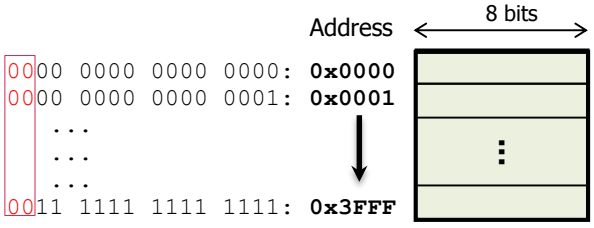


- A microprocessor can only handle memory addresses from `0x0000` to `0x3FFF`. What is the address bus size? If each memory position is one byte wide, what is the maximum size (in bytes) of the memory that we can connect?

  We want to cover all the cases from `0x0000` to `0x3FFF`:

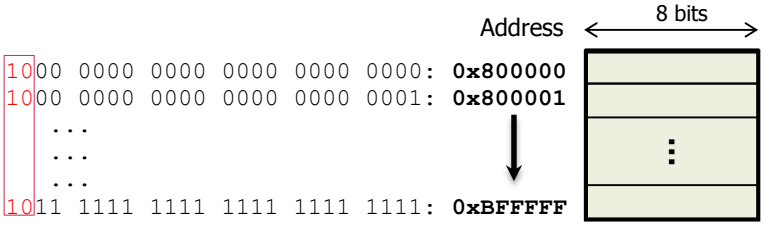
  The range from 0x0000 to 0x3FFF is akin to all possible cases with 14 bits. Thus, the address bus size is **14 bits**.

  We can handle $2^{14} bytes = 16KB$ of memory.



- A microprocessor has a 24-bit address line. We connect a memory chip to the microprocessor. The memory chip addresses are assigned the range `0x800000` to `0xBFFFFF`. What is the minimum number of bits required to represent addresses in that individual memory chip? If each memory position is one byte wide, what is the memory size (in bytes)?

  By looking at the binary numbers from `0x80000` to `0xBFFFFF`, we notice that the addresses in that range require 24 bits. But all those addresses share the same first two MSBs: `10`. Thus, if we were to use only that memory chip, we do not need those 2 bits, and we only need **22 bits**.



  We can handle $2^{22} bytes = 4MB$ of memory.

- A memory has a size of 512KB, where each memory content is 8-bits wide. How many bits do we need to address the contents of this memory?

  Recall that: $512KB = 2^{19} bytes$. So we need 19 bits to address the contents of this memory (address bus size = 19 bits)
  In general, for a memory with N address positions, the number of bits to address those positions is given by: $\lceil \log_2 N \rceil$

- A 20-bit address line in a microprocessor with an 8-bit data bus handles 1 MB (2^{20} $bytes$) of data. We want to connect four 256 KB memory chips to the microprocessor. Provide the address ranges that each memory device will occupy.

  Each memory chip can handle 256KB of memory. $256KB = 2^{18} bytes$, requiring 18 bits for its address.

  For a 20-bit address: we have 5 hexadecimal digits that go from `0x00000` to `0xFFFFF` (2^{20} memory positions).

  We divide the 2^{20} memory positions into 4 contiguous groups, each with 2^{18} memory positions. The figure shows the optimal way of doing so: for each group, the 18 LSBs of the memory addresses correspond to the memory range of a 256 KB memory. And the 2 MSBs of the memory addresses are the same within a group. For a given memory address, we can quickly determine which group it belongs to by looking at its 2 MSBs.

BINARY CODES

- We know that with n bits, we can represent 2^n numbers, from $0\ to\ 2^n - 1$. This is a commonly used range. However, with 'n' bits, we can also represent 2^n numbers in any range.
- Moreover, with n bits we can represent 2^n different symbols. For example, in 24-bit color, each color is represented by 24 bits, providing 2^{24} distinct colors. Each color is said to have a binary code.
- N = 5 symbols. With 2 bits, only 4 symbols can be represented. With 3 bits, 8 symbols can be represented. Thus, the number of bits required is $n = 3 = \lceil log_2 5 \rceil = log_2 8$. Note that 8 is the power of 2 closest to N=5 that is greater than or equal to 5.
- In general, if we have N symbols to represent, the number of bits required is given by $\lceil \log_2 N \rceil$. For example:
  - ✓ Minimum number of bits to represent 70,000 colors: → Number of bits: $\lceil \log_2 70000 \rceil = 17\ bits$.
  - ✓ Minimum number of bits to represent numbers between 15,000 and 19,096: → There are 19,096-15,000+1=4097. Then, number of bits: $\lceil \log_2 4097 \rceil = 13\ bits$.

7-bit US-ASCII character-encoding scheme: Each character is represented by 7 bits. Thus, the number of characters that can be represented is given by $2^7 = 128$. Each character is said to have a binary code.

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

Unicode: This code can represent more than 110,000 characters and attempts to cover all world's scripts. A common character encoding is UTF-16, which uses 2 pair of 16-bit units: For most purposes, a 16 bit unit suffices ($2^{16} = 65536$ characters):

ϑ (Greek theta symbol) = `03D1` Ω (Greek capital letter Omega): `03A9` Ж (Cyrillic capital letter zhe): `0416`

BCD Code:

- In this coding scheme, decimal numbers are represented in binary form by independently encoding each decimal digit in binary form. Each digit requires 4 bits. Note that only values from 0 are 9 are represented here.
- This is a very useful code for input devices (e.g.: keypad). But it is not a coding scheme suitable for arithmetic operations. Also, notice that the binary numbers 1011_2(10) to 1111_2(15) are not used. Only 10 out of 16 values are used to encode each decimal digit.
- **Examples:**
  - ✓ Decimal number **47**: This decimal number can be represented as a binary number: 101111_2. In BCD format, this would be: **0100 0111**$_2$
  - ✓ Decimal number **58**: This decimal number can be represented as a binary number: 111010_2. In BCD format, the binary representation would be: **01011000**$_2$
  - ✓ The BCD code is not the same as the binary number!

BCD	decimal #
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

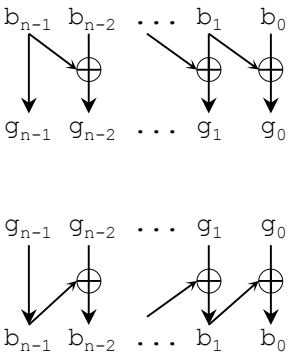
- There exist many other binary codes (e.g., reflective gray code, 6-3-1-1 code, 2-out-of-5 code) to represent decimal numbers. Usually, each of them is tailored to an specific application.

REFLECTIVE GRAY CODE:

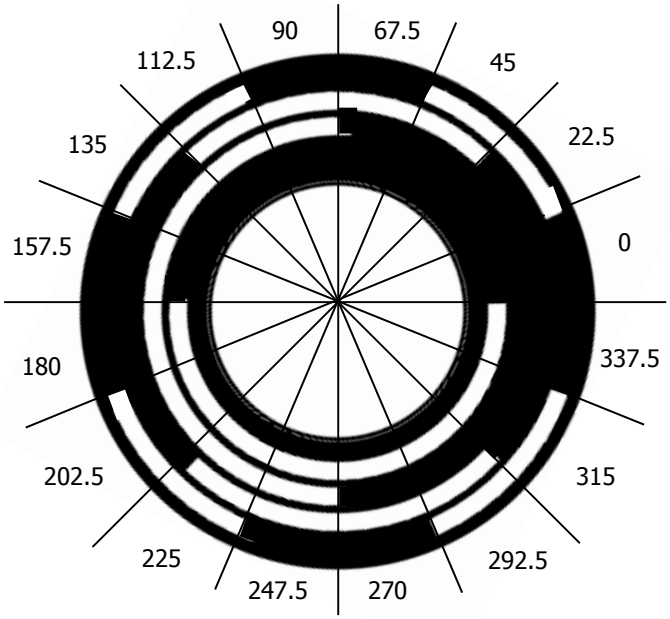
g_1g_0
0 0
0 1
1 1
1 0

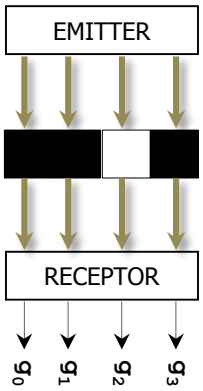
Decimal Number	$b_2b_1b_0$	$g_2g_1g_0$
0	0 0 0	0 0 0
1	0 0 1	0 0 1
2	0 1 0	0 1 1
3	0 1 1	0 1 0
4	1 0 0	1 1 0
5	1 0 1	1 1 1
6	1 1 0	1 0 1
7	1 1 1	1 0 0

$b_3b_2b_1b_0$	$g_3g_2g_1g_0$
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 0	0 0 1 1
0 0 1 1	0 0 1 0
0 1 0 0	0 1 1 0
0 1 0 1	0 1 1 1
0 1 1 0	0 1 0 1
0 1 1 1	0 1 0 0
1 0 0 0	1 1 0 0
1 0 0 1	1 1 0 1
1 0 1 0	1 1 1 1
1 0 1 1	1 1 1 0
1 1 0 0	1 0 1 0
1 1 0 1	1 0 1 1
1 1 1 0	1 0 0 1
1 1 1 1	1 0 0 0



- **Application:** Measuring angular position with 4-bit RGC. 4 beams are emitted along an axis. When a light beam passes (transparent spots, represented as whites), we get a logical 1, 0 otherwise. The RGC encoding makes that between areas only one bit changes, thereby reducing the possibility of an incorrect reading (especially when the beam between adjacent areas). For example: from 0001 to 0011 only one bit flips. If we used 0001 to 0010, two bits would flip: that would be prone to more errors, especially when the beams are close to the line where the two areas meet.



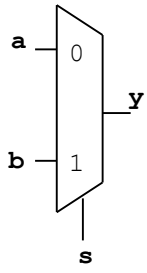


Angle	$g_3g_2g_1g_0$
0 - 22.5	0 0 0 0
22.5 - 45	0 0 0 1
45 - 67.5	0 0 1 1
67.5 -90	0 0 1 0
90 - 112.5	0 1 1 0
112.5 - 135	0 1 1 1
135 - 157.5	0 1 0 1
157.5 - 180	0 1 0 0
180 - 202.5	1 1 0 0
202.5 - 225	1 1 0 1
225 - 247.5	1 1 1 1
247.5 - 270	1 1 1 0
270 - 292.5	1 0 1 0
292.5 - 315	1 0 1 1
315 - 337.5	1 0 0 1
337.5 - 360	1 0 0 0

COMBINATIONAL CIRCUITS:

MULTIPLEXERS (MUXS)

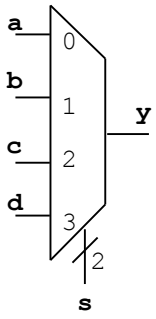
- This logic circuit selects one of many input signals and forwards the selected input to the output line.
- Boolean equations for MUX2-to-1, MUX4-to-1, MUX8-to-1:



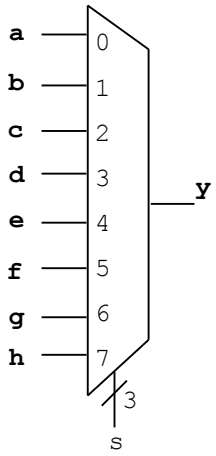
s	a	b	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

s	y
0	a
1	b

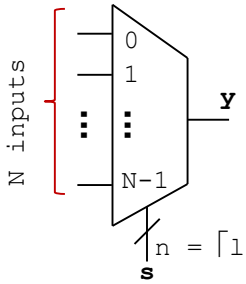
$$y = \bar{s}a + sb$$



$$y = \bar{s}_1\bar{s}_0 a\ \bar{s}_1 s_0 b + s_1\bar{s}_0 c + s_1 s_0 d$$



$$y = \bar{s}_2\bar{s}_1\bar{s}_0 a + \bar{s}_2\bar{s}_1 s_0 b + \bar{s}_2 s_1\bar{s}_0 c + \bar{s}_2 s_1 s_0 d + s_2\bar{s}_1\bar{s}_0 e + s_2\bar{s}_1 s_0 f + s_2 s_1\bar{s}_0 g + s_2 s_1 s_0 h$$



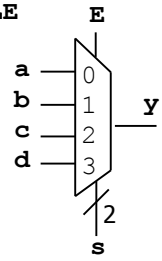
- Normally, a multiplexer has $N = 2^n$ inputs, one output, and a selector with n bits.
- But, if a multiplexer has N inputs, where N is not a power of 2, the number of bits of the selector is given by: $\lceil log_2 N \rceil$.

MULTIPLEXERS WITH ENABLE

- An enable input provides us with an extra level of control. If the multiplexer is enabled, the circuit works. If the multiplexer is not enabled, no input is allowed into the output, and the multiplexer output becomes '0' (if the output is active-high) or '1' (if the output if active-low). The enable input can be either active-high or active-low:

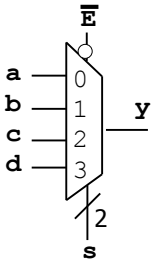
ACTIVE HIGH ENABLE

E	s_1	s_0	y
1	0	0	a
1	0	1	b
1	1	0	c
1	1	1	d
0	X	X	0



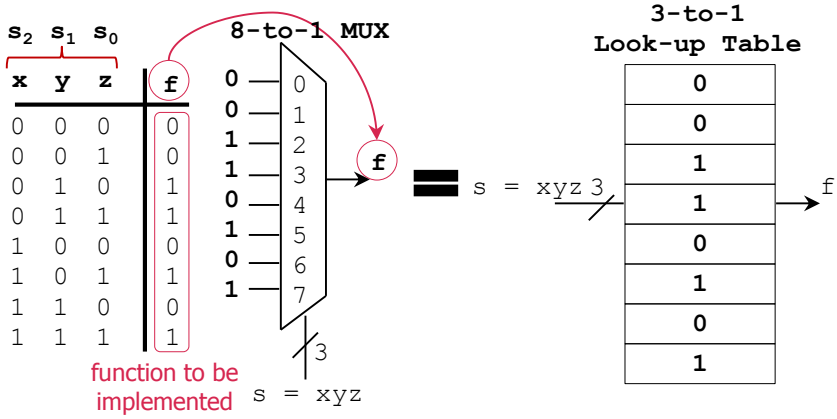
ACTIVE LOW ENABLE

$\bar{E}$	s_1	s_0	y
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	X	X	0



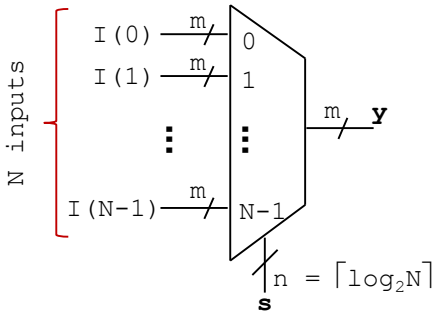
LOGIC CIRCUITS WITH MUXs

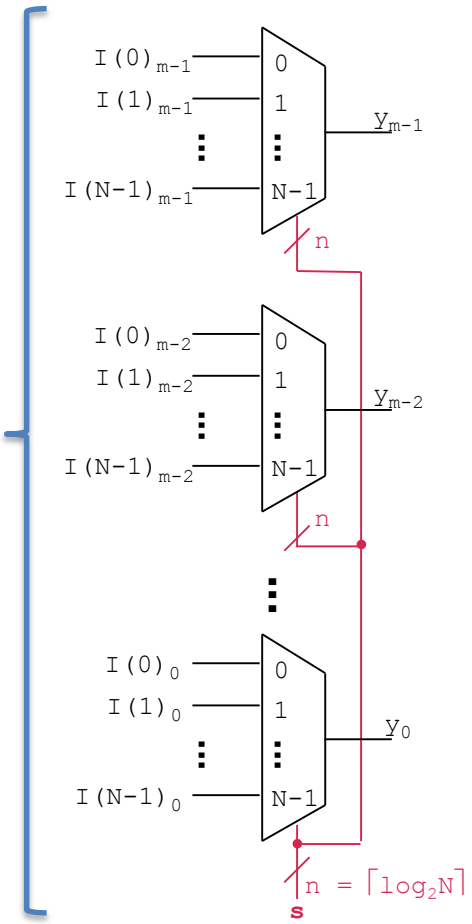
- Multiplexers can be used to implement Boolean Functions. The selector can be thought as the input variables, the input bits are fixed values that are passed onto the output according to the selector.
- This multiplexor with fixed inputs implements a logic function. The functionality of this circuit is similar to that of a Look-Up Table (LUT), which is a ROM-like circuit whose values are obtained by addressing them. FPGAs implement Boolean functions using LUTs. In the example, a 3-to-1 LUT is an LUT with 3 inputs, i.e., it contains $2^3 = 8$ addresses.



BUS MULTIPLEXERS

- Usually we want input signals to contain more than one bit.
- In the figure, each input signal contains 'm' bits.
- This 'bus multiplexer' can be built by 'm' multiplexers, each taking care of only one bit for all the inputs.

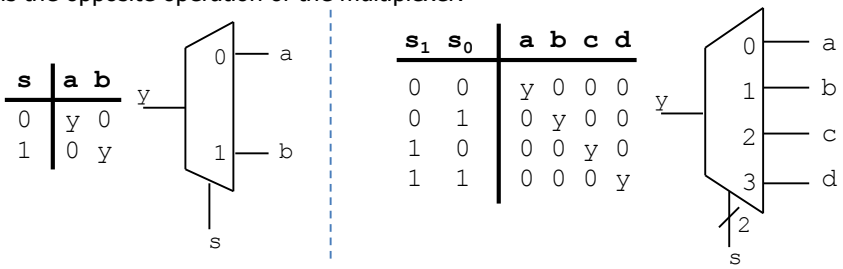




- We have 'N' inputs and therefore the selector has $n = \lceil log_2 N \rceil$ bits.
- Note that the selector is the same for all the multiplexers.

DEMULTIPLEXERS

- A demultiplexer performs the opposite operation of the multiplexer.

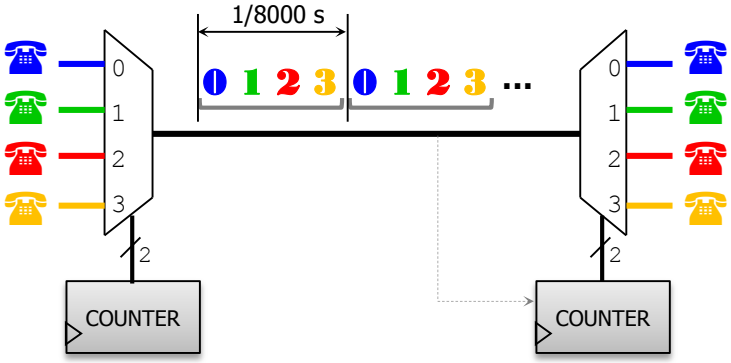


s	a	b
0	y	0
1	0	y

s_1	s_0	a	b	c	d
0	0	y	0	0	0
0	1	0	y	0	0
1	0	0	0	y	0
1	1	0	0	0	y

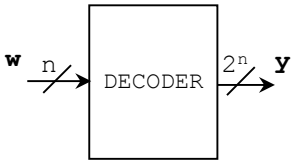
Application: Time Division Multiplexing (TDM)

- Digital Telephony: (4 KHz bandwidth)
- 8000 samples per second, 8 bits per sample. This requires 64000 bits per second.
- In the figure, there are 4 telephone lines (4 signals). To take advantage of the communication channel, only one signal is transmitted at a time. We can do this since we are only required to transmit samples of a particular signal at the rate of 8000 samples per second (or 125 us between samples, this is controlled by counters).

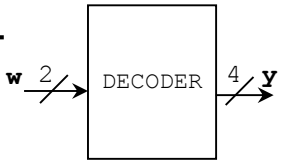


DECODERS

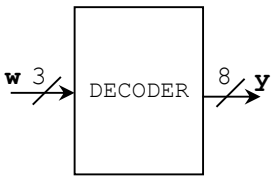
- Generally speaking, decoders are circuits that transform the inputs into outputs following a certain rule, provided that the number of outputs is greater than or equal to the number of inputs.
- Here, we discuss standard decoders for which a specific input/output rule exists. These decoders have n inputs and 2^n outputs. We show examples of: a 2-to-4 decoder, 3-to-8 decoder, and a 2-to-4 decoder with enable. The output y_i is activated when the decimal value of the input w is equal to i.



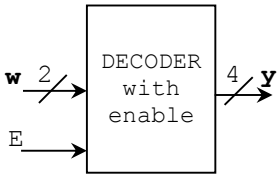
w_1	w_0	y_3	y_2	y_1	y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



w_2	w_1	w_0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



E	w_1	w_0	y_3	y_2	y_1	y_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0

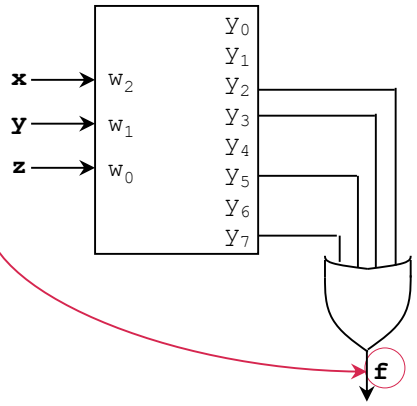


LOGIC CIRCUITS WITH DECODERS

- Decoders can be used to implement Boolean functions. Note that each output is actually a minterm.
- In the example, minterm 2 is activated when xyz=010, here only y_2 is 1. Also: y_5 is activated when xyz=101, y_7 is activated when xyz=111.

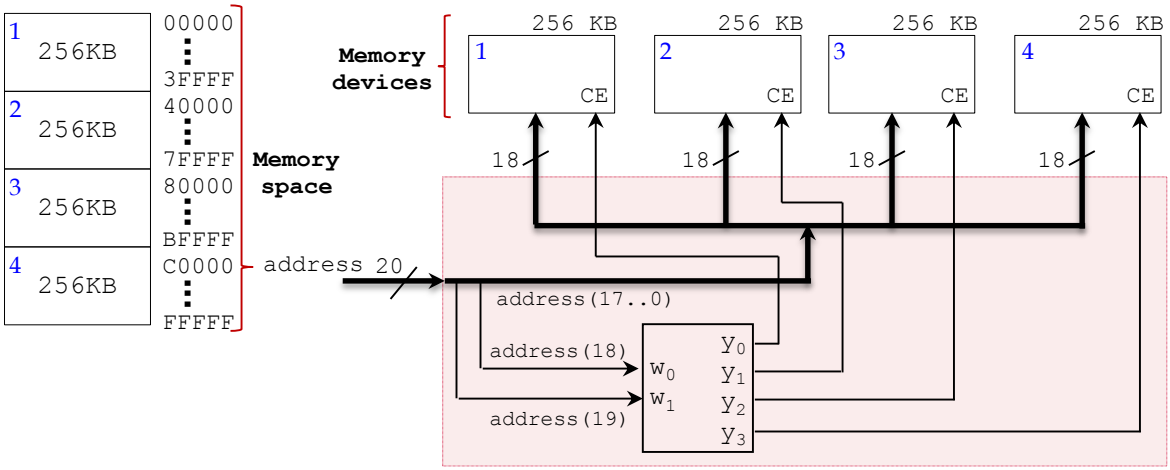
w_2	w_1	w_0	f
x	y	z	
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

function to be implemented



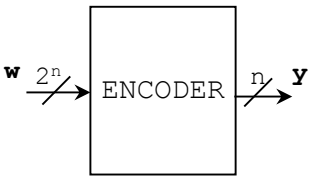
Application: Memory Decoding

- A 20-bit address line in a μprocessor handles up to $2^{20} = 1\,MB$ of addresses, each address containing one-byte of information. We want to connect four 256KB memory chips to the μprocessor.
- The pink-shaded circuit: i) addresses the memory chips, and ii) enables only one memory chip (via CE: chip enable) when the address falls in the corresponding range. Example: if $address = 0x5FFFF$, → only memory chip 2 is enabled (CE=1). If $address = 0xD0123$, → only memory chip 4 is enabled.

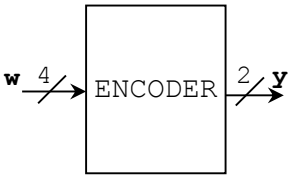


ENCODERS

- Generally speaking, encoders are circuits that transform the inputs into outputs following a certain rule, provided that the number of outputs is lower than the number of inputs.
- Here, we discuss standard encoders for which a specific input/output rule exists. These encoders have 2^n inputs and n outputs. The operation is the opposite of a standard decoder: if an input w_i is activated, then the index i appears at the output y (in binary form).

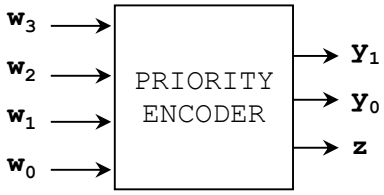


w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



PRIORITY ENCODERS

- Standard encoder: we check whether a specific input is activated for the output to have a value.
- What happens when more than one input is activated? We can include an extra output that is activated to indicate than an unexpected condition has occurred.
- An interesting alternative is to create a ***priority encoder***: if more than one input is activated, then we only pay attention to the input bit of the highest order. For example if $w = 1101$, then we only pay attention to $w(3) = 1$; if $w = 0111$, we only pay attention to $w(2) = 1$. This results in the following truth table for a 4-to-2 priority encoder:



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

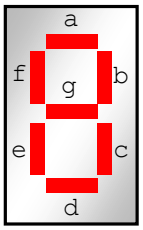
- What if no input is activated? Here we run out of output bits in y to represent this case. Thus, we include an extra output z that it is '0' when no input activated, and '1' otherwise.

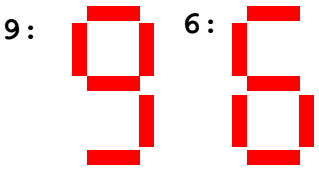
CODE CONVERTERS

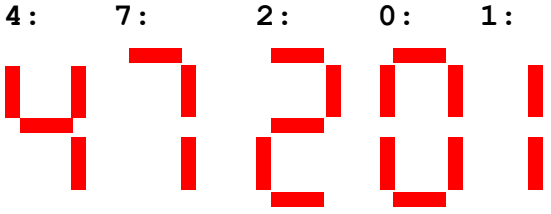
BCD TO 7-SEGMENT DECODER

- It is a decoder because the number of outputs is greater than the number of inputs
- The truth table considers the inputs and outputs to be active-high.

b_3	b_2	b_1	b_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	0	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	x	x	x	x	x	x	x
1	0	1	1	x	x	x	x	x	x	x
1	1	0	0	x	x	x	x	x	x	x
1	1	0	1	x	x	x	x	x	x	x
1	1	1	0	x	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x

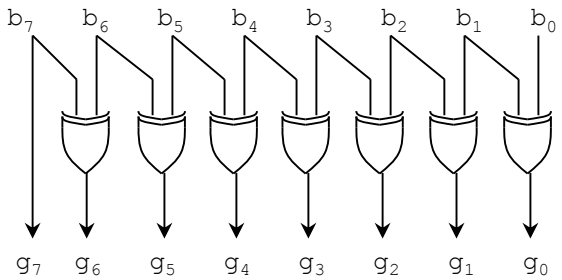


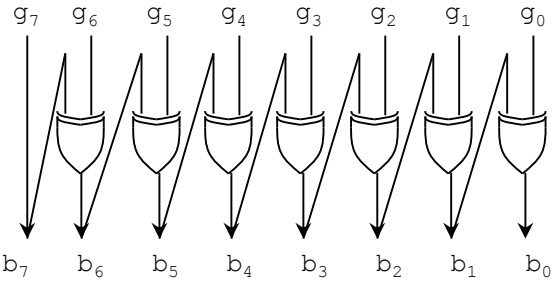




BINARY TO GRAY AND GRAY TO BINARY DECODERS

- It is a decoder because the number of outputs is equal to the number of inputs.
- For small input sizes, we can use the truth table method. But for large input sizes (e.g.: 8 bits), the following circuits are more efficient:





PARITY GENERATORS AND PARITY CHECKERS

- This is defined in the context of an error detection system with transmission and reception units.
- Data to be transmitted: $X = x_{n-1}x_{n-2}\dots x_1x_0$ Transmitted stream: $Y = x_{n-1}x_{n-2}\dots x_1x_0p$, p: parity bit
- **Parity definition**:
  - ✓ Even Parity: Y has an even number of 1s → p_e=1, 0 otherwise
  - ✓ Odd Parity: Y has an odd number of 1s → p_o=1, 0 otherwise.
- This definition is problematic since p is not known. An alternative definition, based on the actual data X is:
  - ✓ Even Parity: X has an odd number of 1s → p_e = 1, 0 otherwise
  - ✓ Odd Parity: X has an even number of 1s → p_o = 1, 0 otherwise.
- **Parity Generator**: Circuit that generates the parity bit based on the actual data X
- **Parity Checker:** Circuit that verifies whether the stream Y has the correct parity.

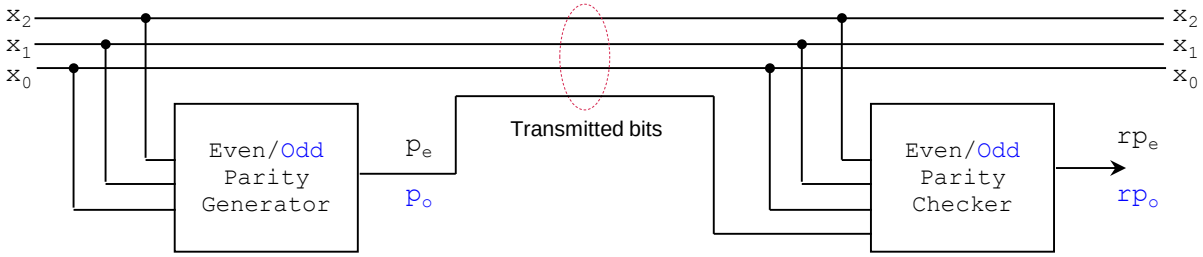
Example:

- For the following error detection system, $X = x_2x_1x_0, n = 3$. The parity generator and checker are always of the same parity:
  - ✓ Even Parity Generator: It generates the parity bit p_e.
  - ✓ Even Parity Checker: It verifies that the received stream Y has even parity. If so, rp_e =0, otherwise rp_e=1 (to signal an error)

$$p_e = x_2 \oplus x_1 \oplus x_0, \quad rp_e = x_2 \oplus x_1 \oplus x_0 \oplus p_e$$

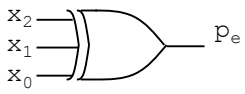
  - ✓ Odd Parity Generator: It generates the parity bit p_o.
  - ✓ Odd Parity Checker: It verifies that the received stream Y has odd parity. If so, rp_o=0, otherwise rp_o=1 (to signal an error)

$$p_o = \overline{x_2 \oplus x_1 \oplus x_0}, \quad rp_o = \overline{x_2 \oplus x_1 \oplus x_0 \oplus p_o}$$



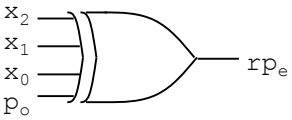
Even Parity Generator

x_2	x_1	x_0	p_e
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



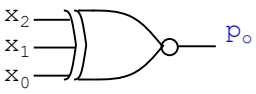
Even Parity Checker

x_2	x_1	x_0	p_e	rp_e
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0



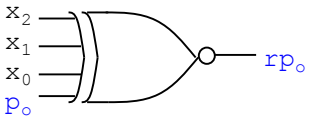
Odd Parity Generator

x_2	x_1	x_0	p_o
0	0	0	1
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	0



Odd Parity Checker

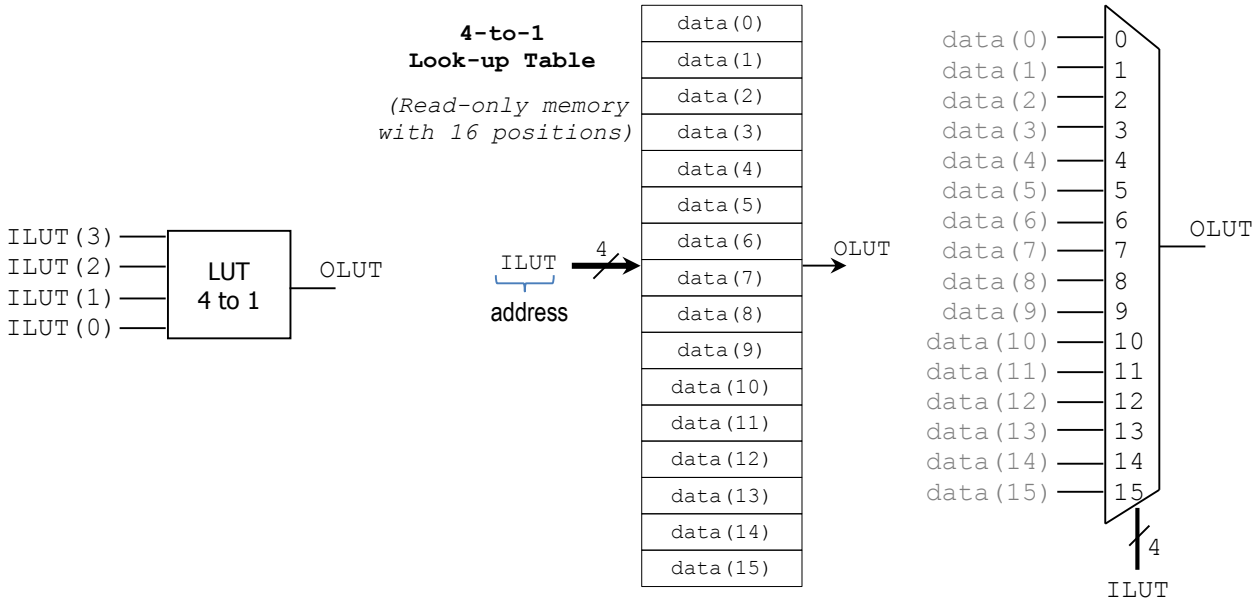
x_2	x_1	x_0	p_o	rp_o
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1



- In general for $X = x_{n-1}x_{n-2}\dots x_1x_0$: $p_e = x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0$. $p_o = \overline{x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0}$
  - ✓ If the # of 1's in an n-bit stream is odd, the n-bit input XOR gate will return 1, 0 otherwise.
  - ✓ If the # of 1's in an n-bit stream is even, the n-bit input XNOR gate will return 1, 0 otherwise.
- $rp_e = x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0 \oplus p_e$. We expect the number of 1s in Y to be even, → an XNOR will detect this. However, we want rp_e to be 1 when this does not happen (to signal an error). Hence, we use an $n+1$-bit input XOR gate.
- $rp_o = \overline{x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0 \oplus p_o}$. We expect the number of 1s in to be odd, → an XOR will detect this. However, we want rp_o to be 1 when this does not happen (to signal an error). Hence, we use an $n+1$-bit input XNOR gate.

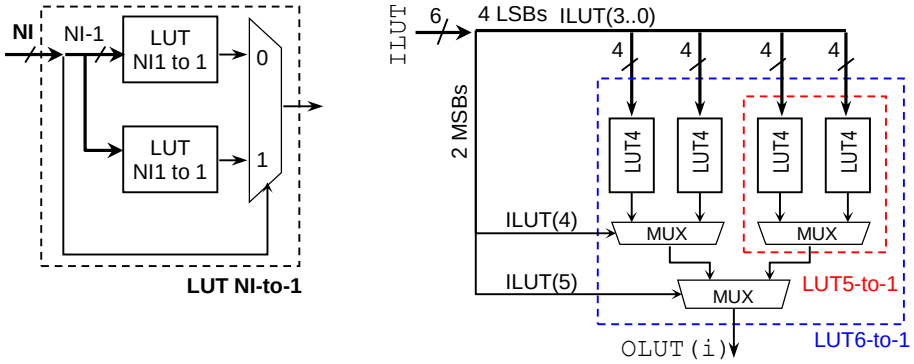
LOOK-UP TABLES (LUTS)

- The LUT contents are hardwired in this circuit. A 4-to-1 LUT can be seen as a ROM with 16 addresses, each address holding one bit. It can also be seen as a multiplexor with fixed inputs.
- This is how FPGAs implement logic functions. A 4-to-1 LUT can implement any 4-input logic function.

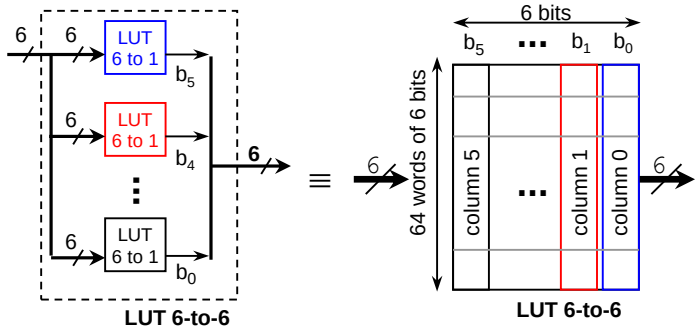


LARGER LUTS

- A larger LUT can be built by building a circuit that allows for more ROM positions.
- Efficient method: A larger LUT can also be built by combining LUTs with multiplexers as shown in the figure. We can build a NI-to-1 LUT with this method. The figure below shows the case for a LUT 6-to-1 built out of two LUT 5-to-1. Each LUT 5-to-1 is build out of two LUT 4-to-1.



- We can build a NI-to-NO LUT using NO NI-to-1 LUTs. This can be seen as a ROM with 2^{NI} addresses, each address holding NO bits. The figure shows how a LUT 6-to-6 is built:



PRACTICE EXERCISES

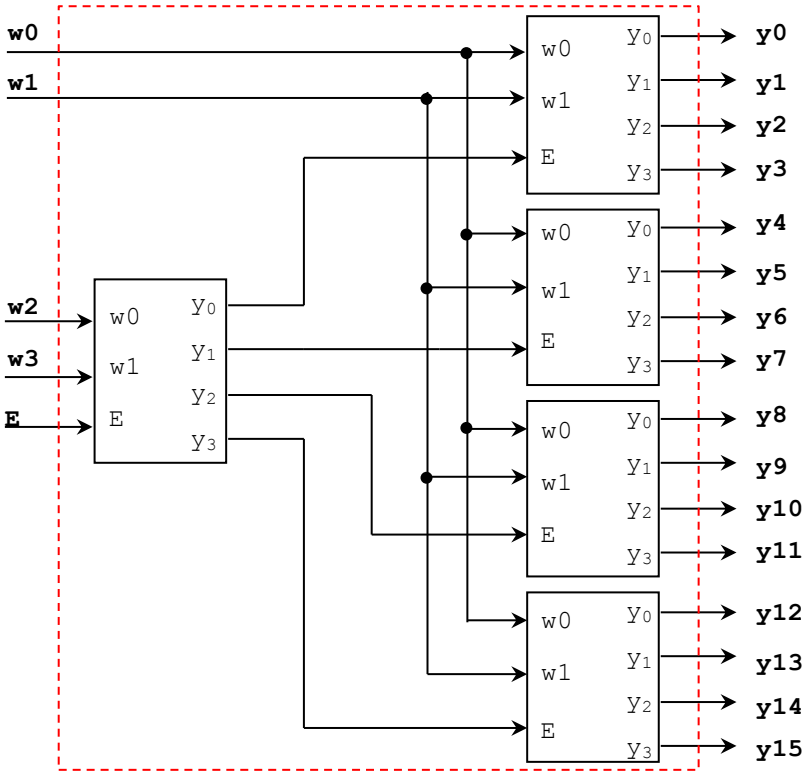
1. Implement the following functions using i) decoders and ii) multiplexers:

✓ $F = \overline{X + Y} + ZY$	✓ $F = (X + Y + Z)(X + Y + \bar{Z})$
✓ $F(X, Y, Z) = \sum(m_0, m_2, m_6)$.	✓ $F = XY + YZ + XZ$
✓ $F(X, Y, Z) = \prod(M_2, M_4, M_7)$	✓ $F = X \oplus Y \oplus Z$

2. Using ONLY 4-to-1 MUXs, implement an 8-to-1 MUX.

3. Implement a 6-to-1 MUX using i) only NAND gates, and ii) only NOR gates.

4. Verify that the following circuit made of out of five 2-to-4 decoders with enable represents a 4-to-16 decoder with enable. Tip: Create the truth table.



5. Using only 2-to-1 MUXs, implement the XOR and XNOR gates.

6. Using only a 4-to-1 MUX, implement the following functions.
   - $F(X, Y, Z) = \sum(m_1, m_3, m_5, m_7)$.
   - $F(X, Y, Z) = \sum(m_1, m_3, m_5)$
   - $F(X, Y, Z) = \sum(m_3, m_5, m_7)$.
   - $F(X, Y, Z) = \sum(m_5, m_7)$.

7. Complete the following timing diagram: