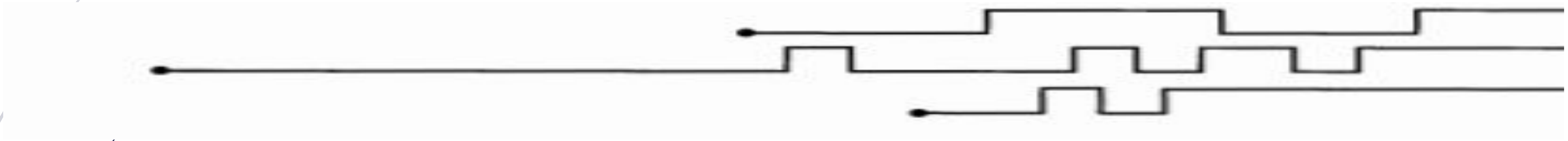
ensia

Digital systems



Chapter 3

Combinational Circuits

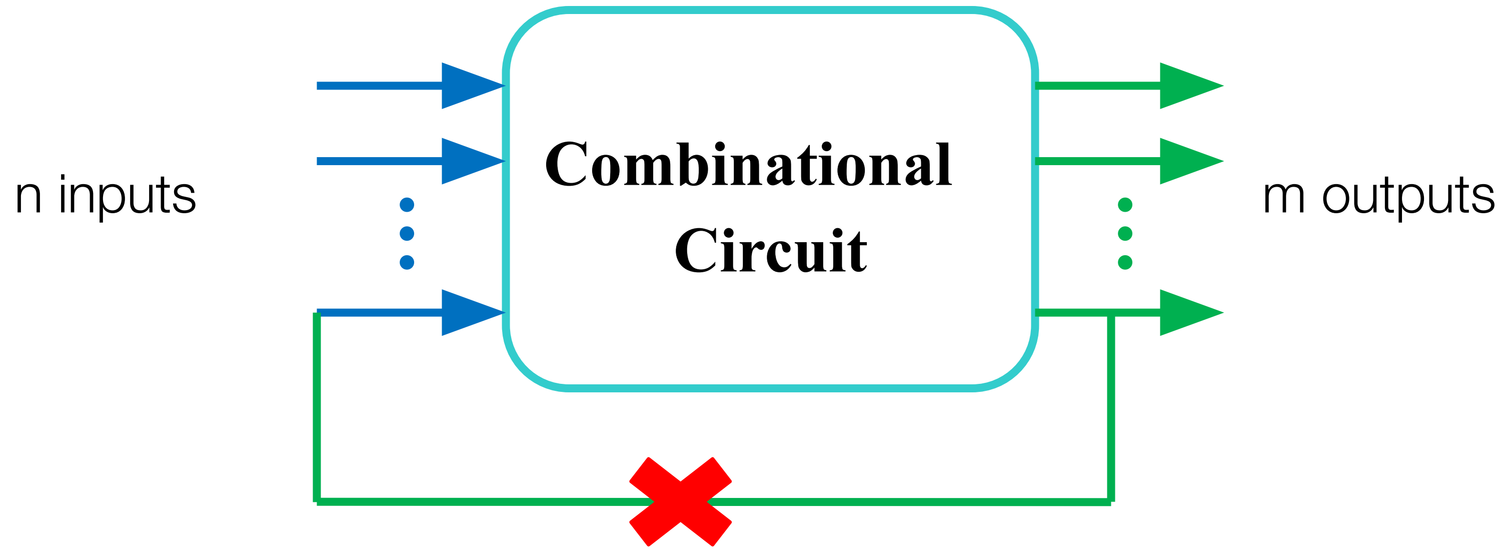
Dr. Iness NEDJI MILAT (Lecturer)
iness.nedji@ensia.edu.dz

Pr. Nasreddine LAGRAA
Nasredine,lagraa@ensia.edu.dz

First year – 2023/2024

Combinational Logic Circuit (CLC)

- Output of CLC depends **only** on the combination of its inputs
- Combinational logic circuits have no feedback



Combinational Logic Circuits

Common combinational circuits made up from logic gates, are basically divided into 3 classes :

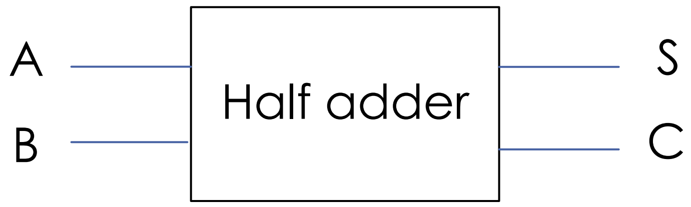
- Arithmetic and Logical circuits
  - Full and Half Adders, Multipliers, Comparator, …
- Code converters circuits
  - Binary, Gray, BCD code converters
- Data transmission circuits
  - Multiplexers, Demultiplexers, Encoders, Decoders,

Half and Full Adders

Functional Block: Half-Adder

- Half adder is a CLC which has **two single bit inputs** and **two single bit outputs**. It performs the following computations:

A	0	0	1	1
+ B	+ 0	+ 1	+ 0	+ 1
C	0	0	0	1
S	0	1	1	0



- A half adder adds two bits to produce a two-bit sum
- The sum is expressed as
  - a sum bit , S and a carry bit, C
- The half adder can be specified as a truth table for S and C ⇒

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Logic Simplification: Half-Adder

- The K-Map for S is:

A \ B	0	1
0	0	1
1	1	0

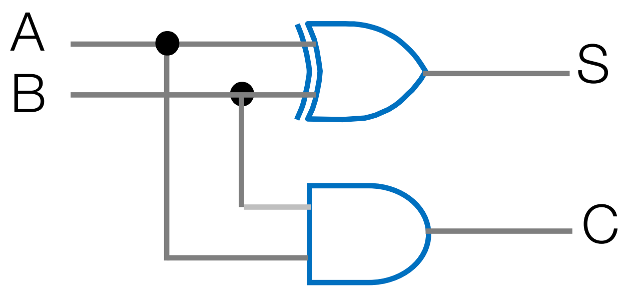
$$S = A \cdot \overline{B} + \overline{A} \cdot B = A \oplus B$$

- The K-Map for C is:

A \ B	0	1
0	0	0
1	0	1

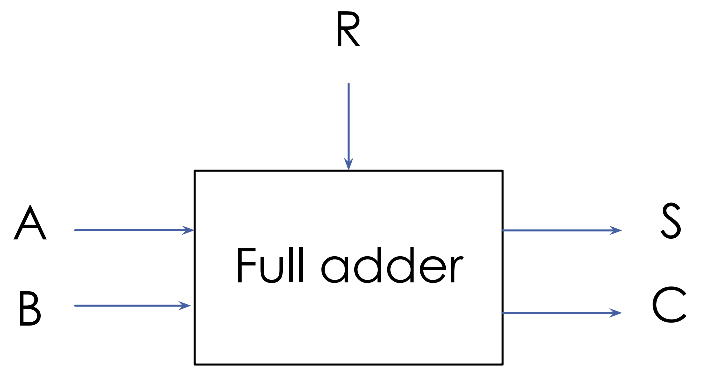
$$C = A \cdot B$$

- These equations lead to the following implementation.



Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. It has **Three single bit inputs** and **two single bit outputs**.
- Like the half-adder, it computes a sum bit, S and a carry bit, C.



- For a carry-in (R) of 0,

it is the same as the half-adder:

R	0	0	0	0
A	0	0	1	1
+ B	+ 0	+ 1	+ 0	+ 1
C	0 0	0	0	1
S		1	1	0

- For a carry- in (R) of 1:

R	1	1	1	1
A	0	0	1	1
+ B	+ 0	+ 1	+ 0	+ 1
C	0	1	1	1 1
S	1	0	0	

Logic Simplification: Full-Adder

Full-Adder Truth Table

A	B	R	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full-Adder K-Map:

K-map for S

BR / A	00	01	11	10
0	0	1	0	1
1	1	0	1	0

K-map for C

BR / A	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Full-Adder Equations :

$$S = A\cdot\overline{B}\cdot\overline{R} + \overline{A}\cdot B\cdot\overline{R} + \overline{A}\cdot\overline{B}\cdot R + A\cdot B\cdot R$$

$$S = A \oplus B \oplus R$$

$$C = A\cdot B + A\cdot R + B\cdot R$$ ✗

$$C = \overline{A}\cdot B\cdot R + A\cdot\overline{B}\cdot R + A\cdot B\cdot\overline{R} + A\cdot B\cdot R$$

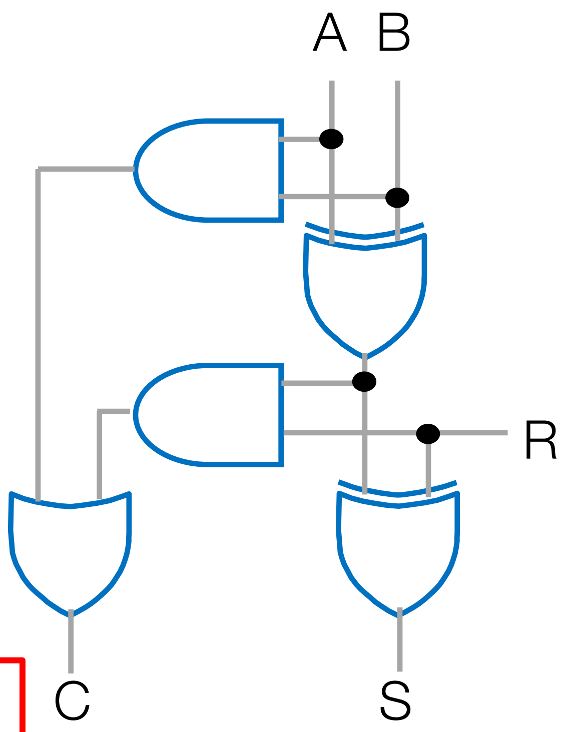
$$C = R\,(\overline{A}.B + A.\overline{B}) + A.B\,(\overline{R} + R)$$

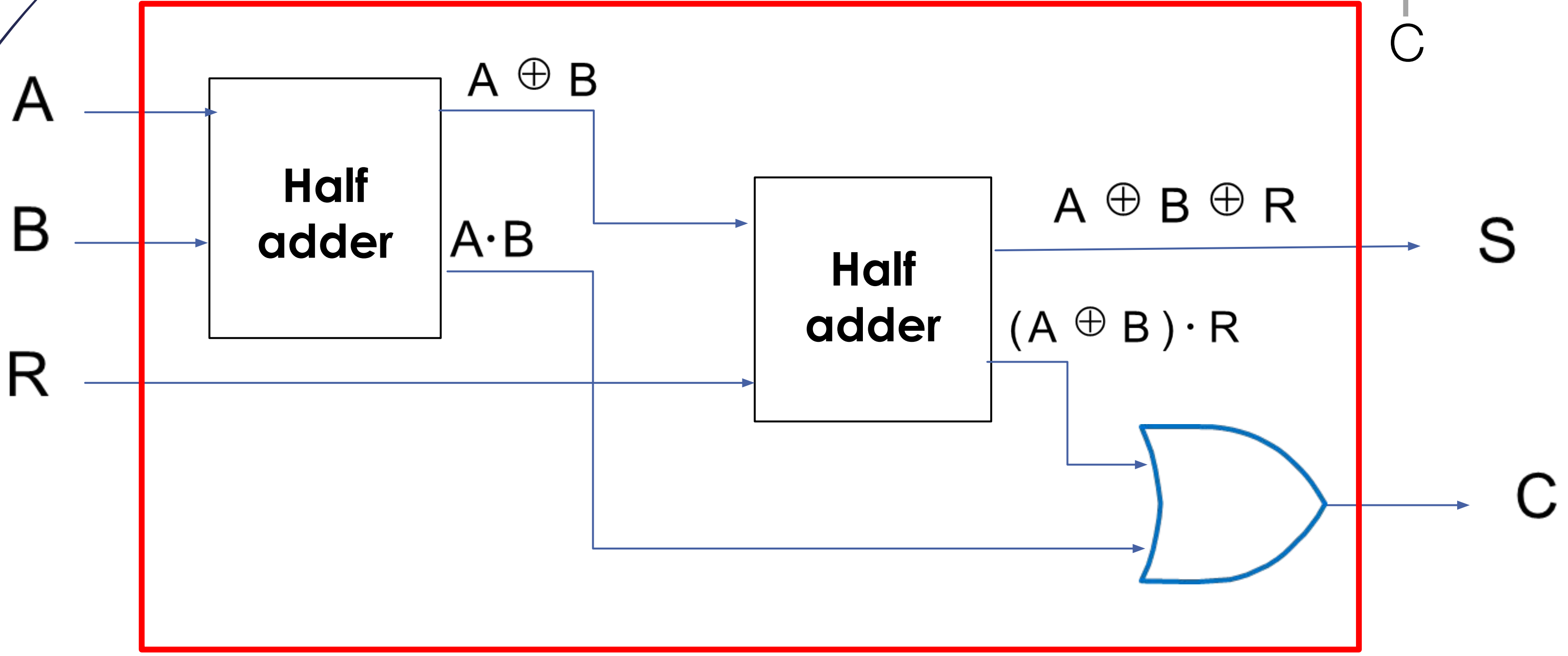
$$C = A\cdot B + (A \oplus B)\cdot R$$

Logic diagram of full adder

$S = A \oplus B \oplus R$

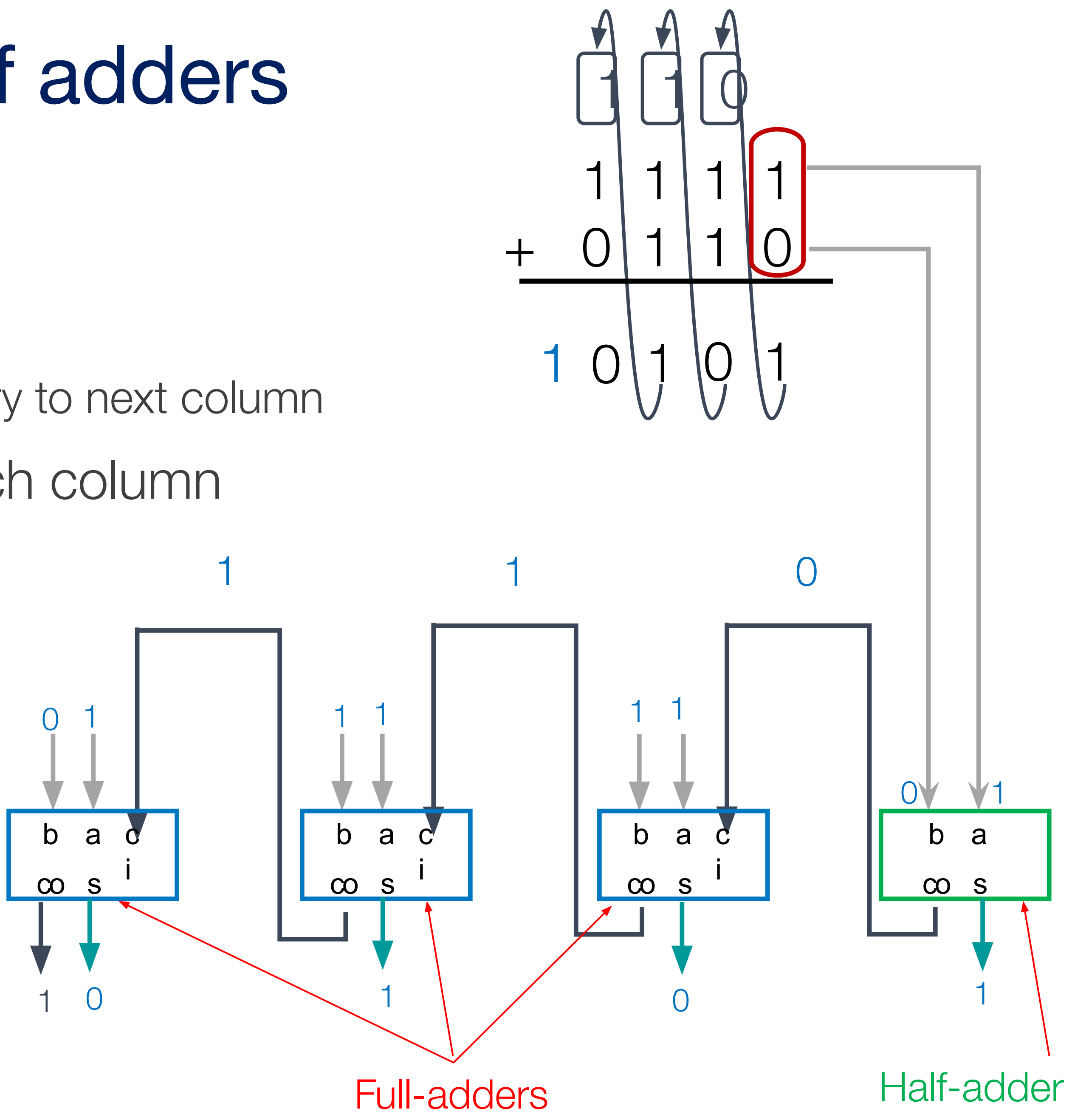
$C = A \cdot B + (A \oplus B) \cdot R$





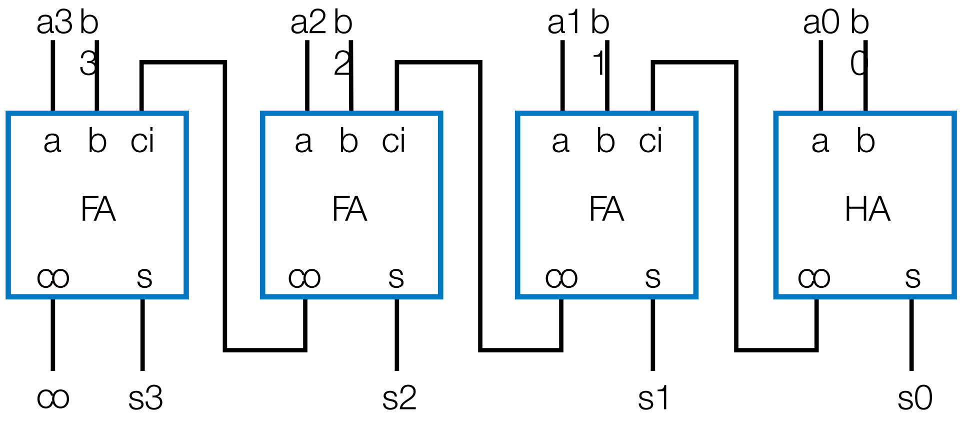
Implementation of adders

- Imitate Adding by Hand
  - One column at a time
    - Compute sum, add carry to next column
- Create component for each column
  - Four bits Fours adders
  - 1 half adder
  - and 3 full adders

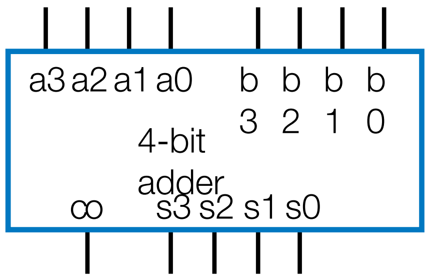


Carry-Ripple Adder

- Using half-adder and full-adders, we can build adder that adds like we would by hand
- Called a carry-ripple adder
  - 4-bit adder shown: Adds two 4-bit numbers, generates 5-bit output
    - 5-bit output can be considered 4-bit "sum" plus 1-bit "carry out"
  - Can easily build any size adder



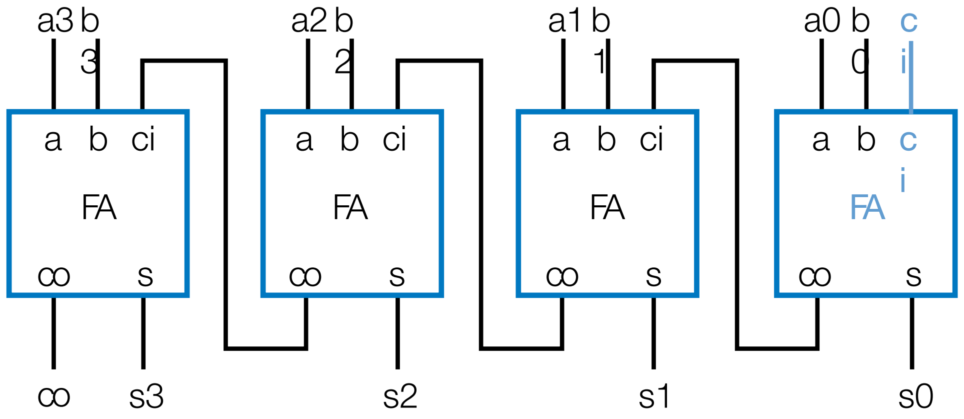
(a)



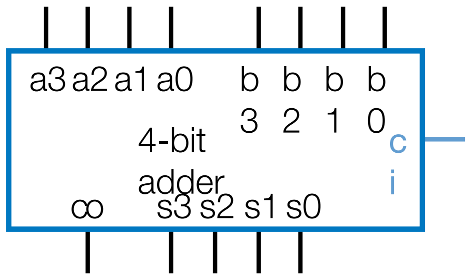
(b)

Carry-Ripple Adder

- Using full-adder instead of half-adder for first bit, we can include a "carry in" bit in the addition
    - Will be useful later when we connect smaller adders to form bigger adders



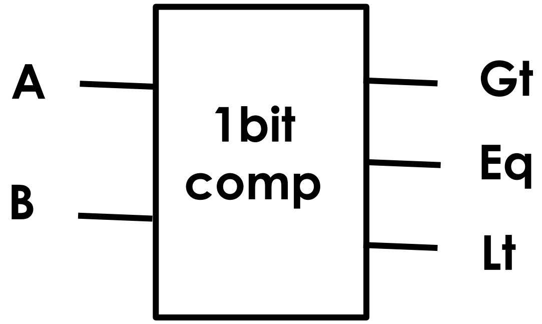
(a)



(b)

Magnitude Comparator

One-bit Comparator

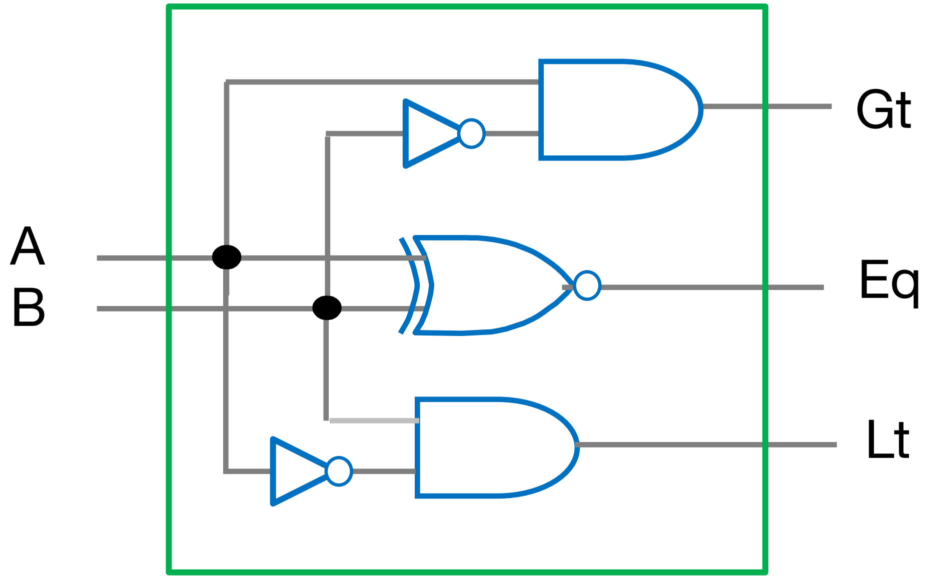


A	B	Gt	Eq	Lt	
0	0	0		1	
0					
0	1	0		0	
1					
1	0	1	0		0
1	1	0	1		0

Gt = AB' (A>B)

Eq = A'B'+ AB (A=B)

Lt = A'B (A<B)



Magnitude comparator

- N-bit magnitude comparator
  - Indicates whether A>B, A=B, or A<B, for its two N-bit inputs A and B
  - How design?
    - Consider how compare by hand.
    - First compare a3 and b3.
    - If equal, compare a2 and b2.
    - And so on…
    - Stop if comparison not equal :
      - whichever's bit is 1 is greater.
      - If never see unequal bit pair, A=B.

- Example: compare A and B

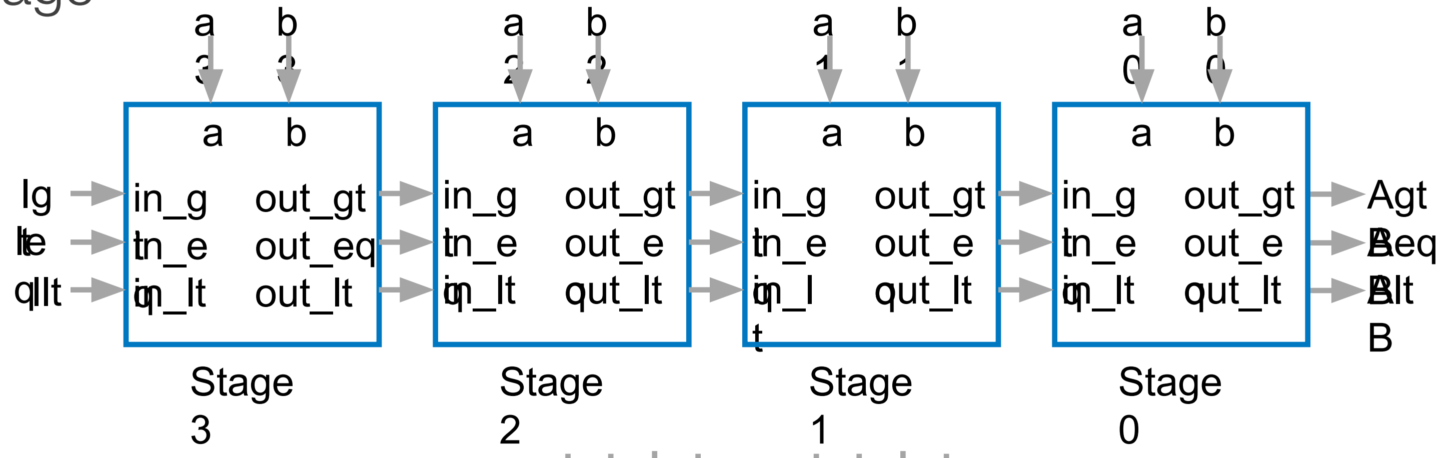
A=1011 B=1001

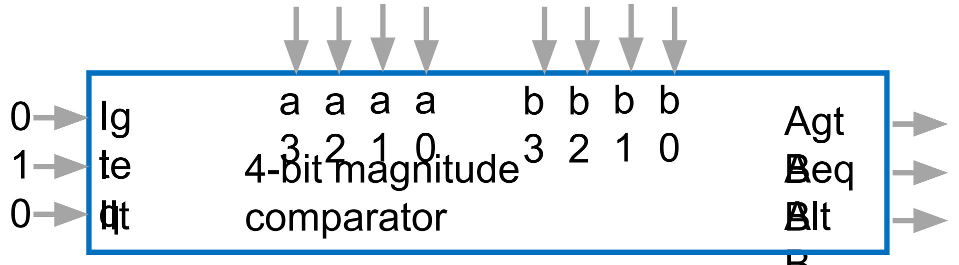
1011	1001	Equal
1011	1001	Equal
1011	1001	Unequal

So A > B

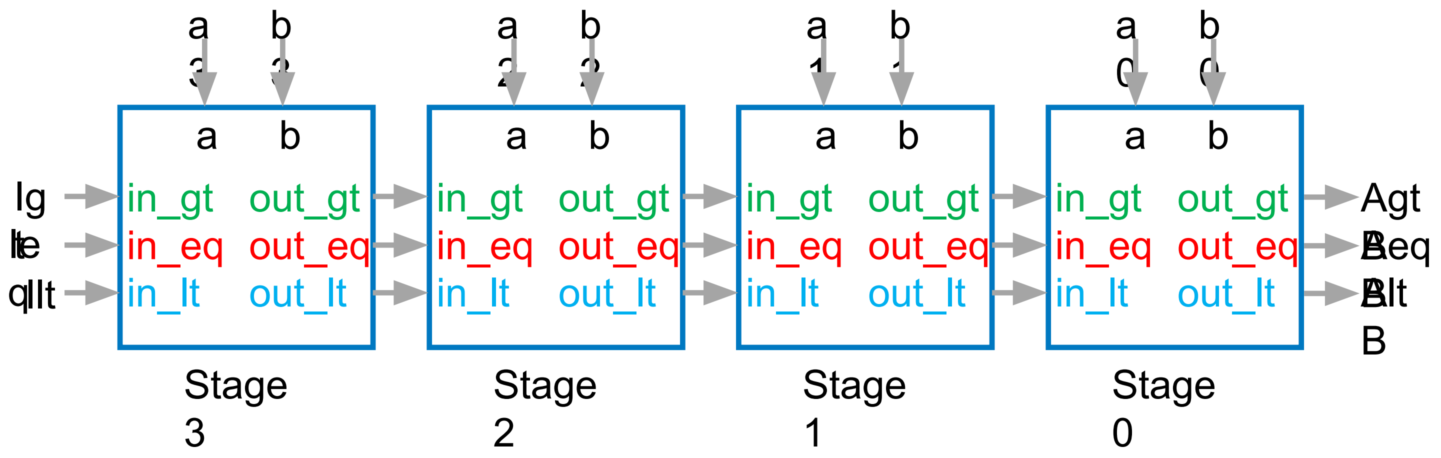
Magnitude comparator

- By-hand example leads to idea for design
  - Start at left, compare each bit pair (MSB), pass results to the right
  - Each bit pair called **a stage**
  - Each stage has 3 inputs indicating results of higher stage, passes results to lower stage





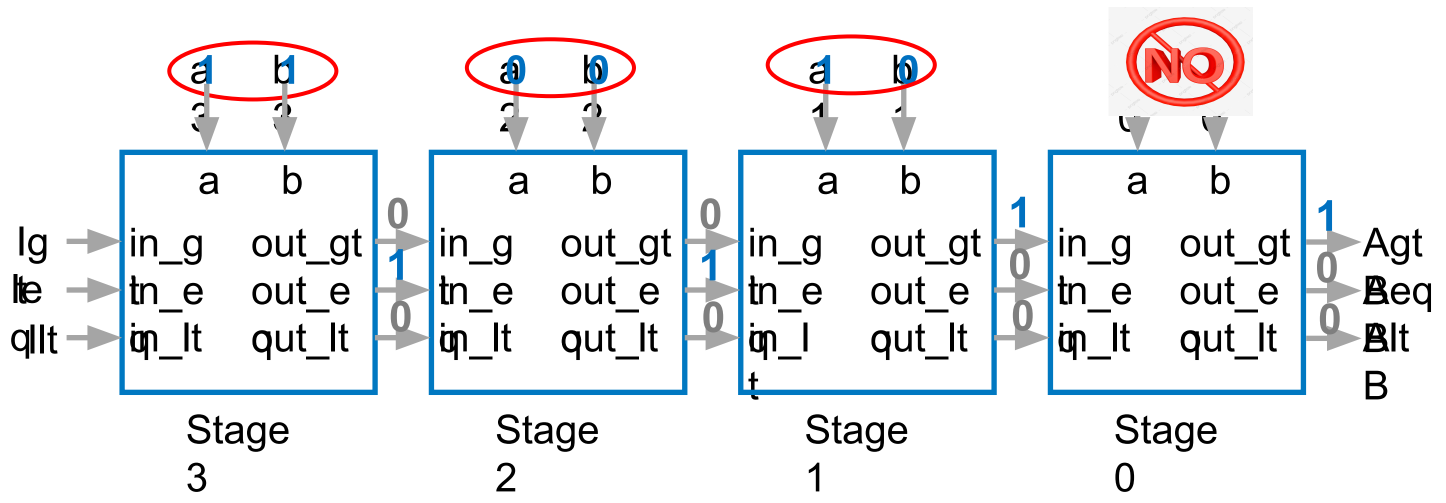
Magnitude comparator



- Each stage:
  - out_gt = in_gt + (in_eq . a . $\overline{b}$)
    - A>B (so far) if already determined in higher stage,
    - or if higher stages equal but in this stage a=1 and b=0
  - out_lt = in_lt + (in_eq . $\overline{a}$. b)
    - A<B (so far) if already determined in higher stage,
    - or if higher stages equal but in this stage a=0 and b=1
  - out_eq = in_eq . (a XNOR b)
    - A=B (so far) if already determined in higher stage and in this stage a=b too

Magnitude comparator : Example

- Compare 1011 and 1001

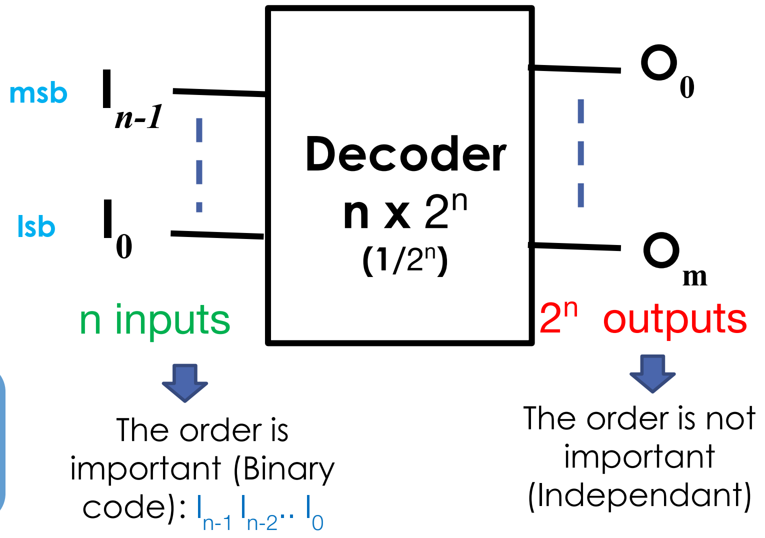


- Final answer appears on the right
- Takes time for answer to “ripple” from left to right

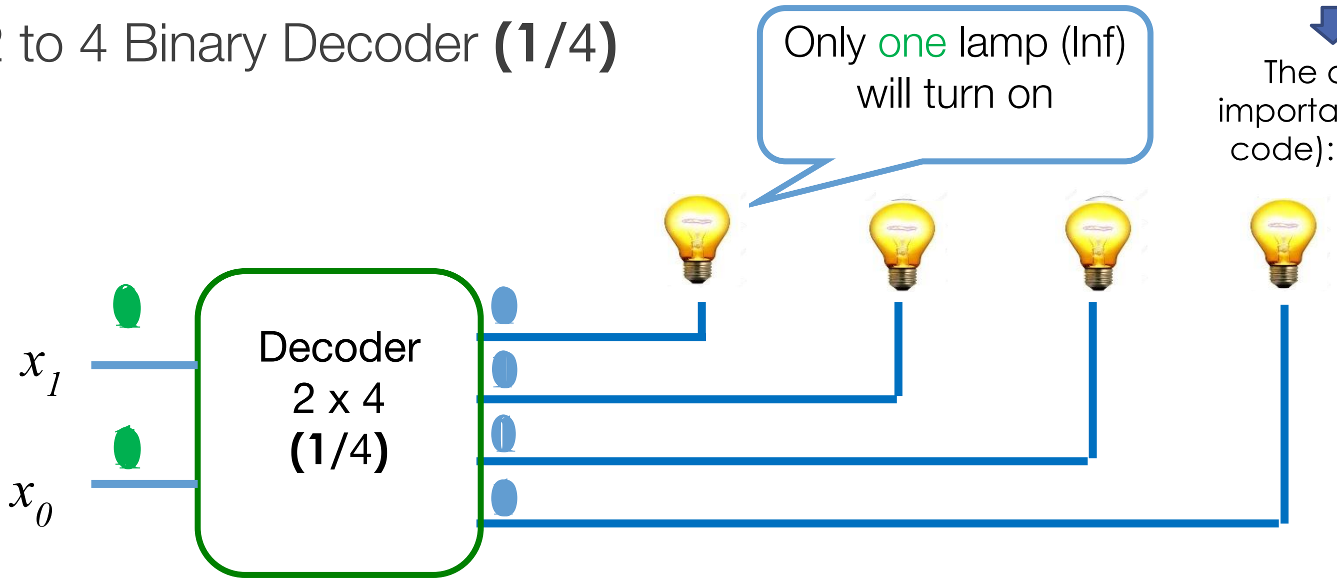
Decoder

Decoders

- A **decoder** is a CLC that has **n inputs** representing a binary number (Code of information) and 2^n **outputs** representing information.
- It activates only **one output** that corresponds to that input number, all other outputs remain inactive.
- It extracts "Information" from the code.

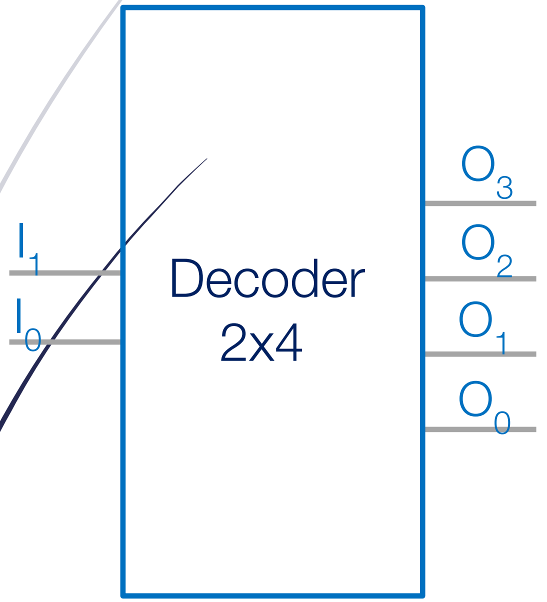

msb
I_{n-1}
lsb
I_0
Decoder
n x 2^n
(1/2^n)
O_0
O_m
n inputs
2^n outputs
The order is important (Binary code): $I_{n-1} I_{n-2} .. I_0$
The order is not important (Independant)

Example: 2 to 4 Binary Decoder **(1/4)**



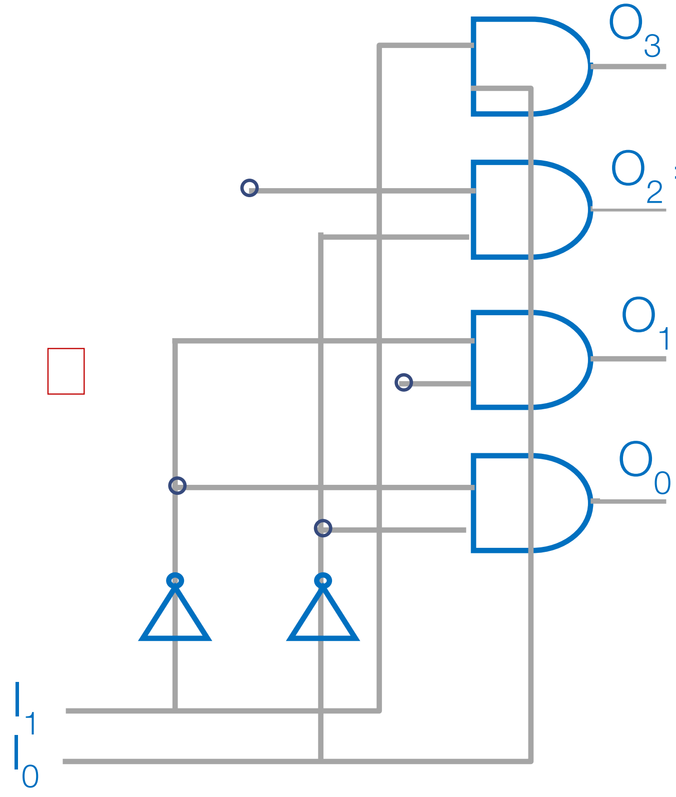
Decoders

 2 to 4 Line Decoder



lsb

I_1	I_0	O_3	O_2	O_1	O_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



$O_3 = I_1 I_0$

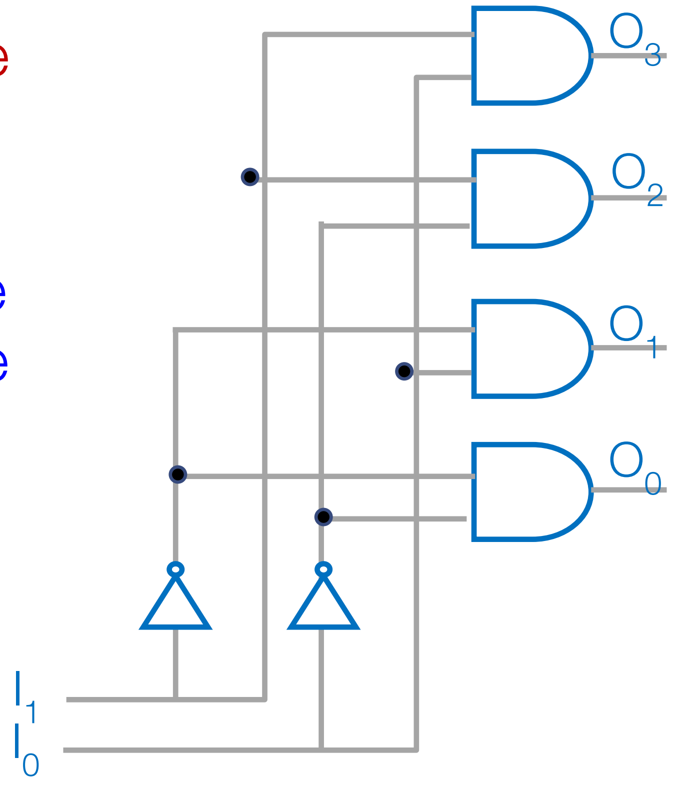
$O_2 = I_1 \bar{I}_0$

$O_1 = \bar{I}_1 I_0$

$O_0 = \bar{I}_1 \bar{I}_0$

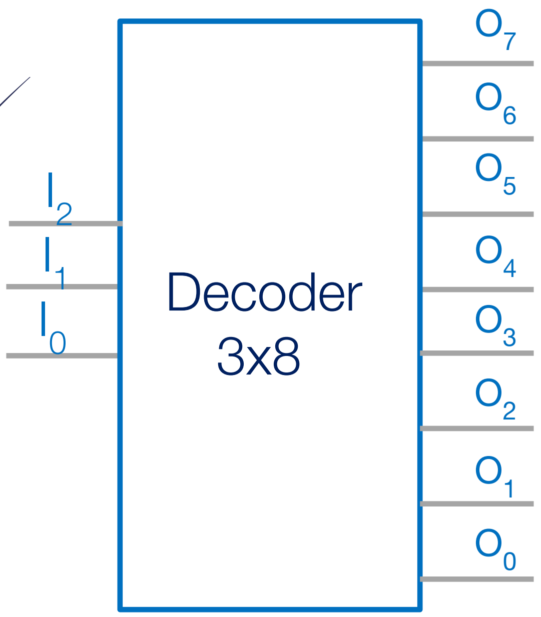
Decoders

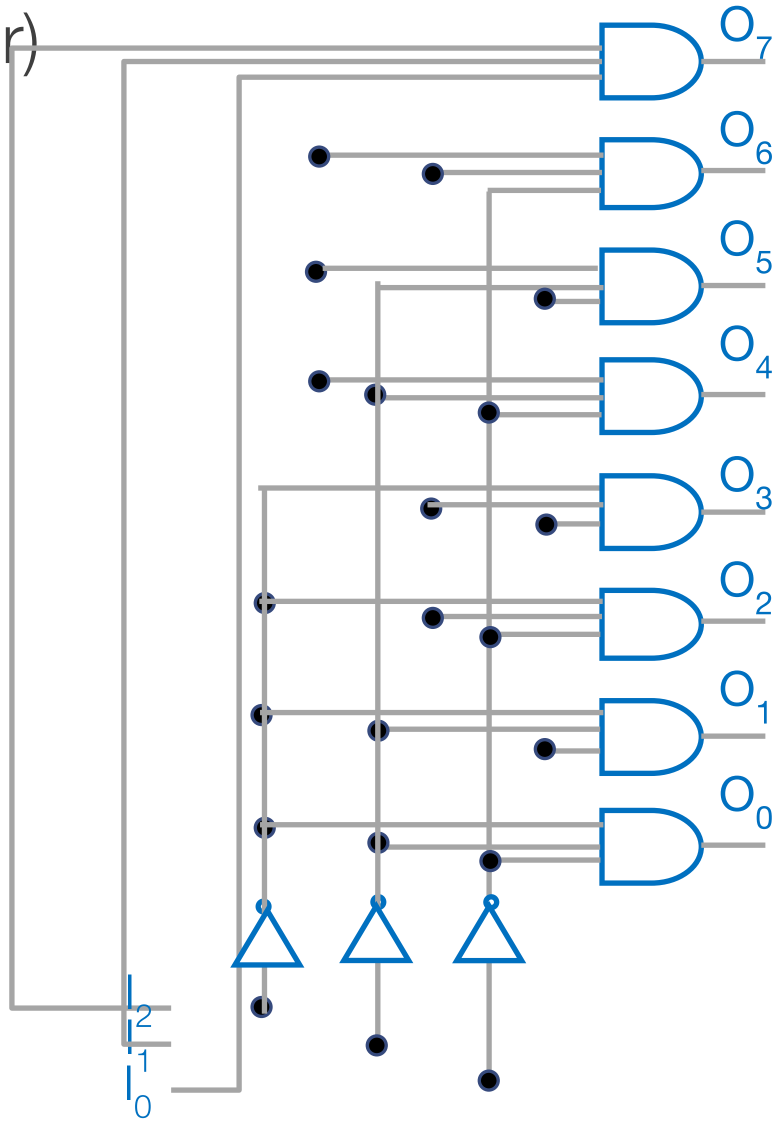
- The decoder is useful in determining how to interpret a **bit pattern**
  - It could be the **address** of a row in DRAM, that the processor intends to read from
  - It could be an **instruction** in the program and the processor needs to decide what action to take (based on *instruction opcode*)



Decoders

- 3-to-8 Line Decoder (1/8 Decoder)
  - Binary-to-Octal Decoder (3-to-8)





$= I_2 I_1 I_0$

$= I_2 I_1 \bar{I}_0$

$= I_2 \bar{I}_1 I_0$

$= I_2 \bar{I}_1 \bar{I}_0$

$= \bar{I}_2 I_1 I_0$

$= \bar{I}_2 I_1 \bar{I}_0$

$= \bar{I}_2 \bar{I}_1 I_0$

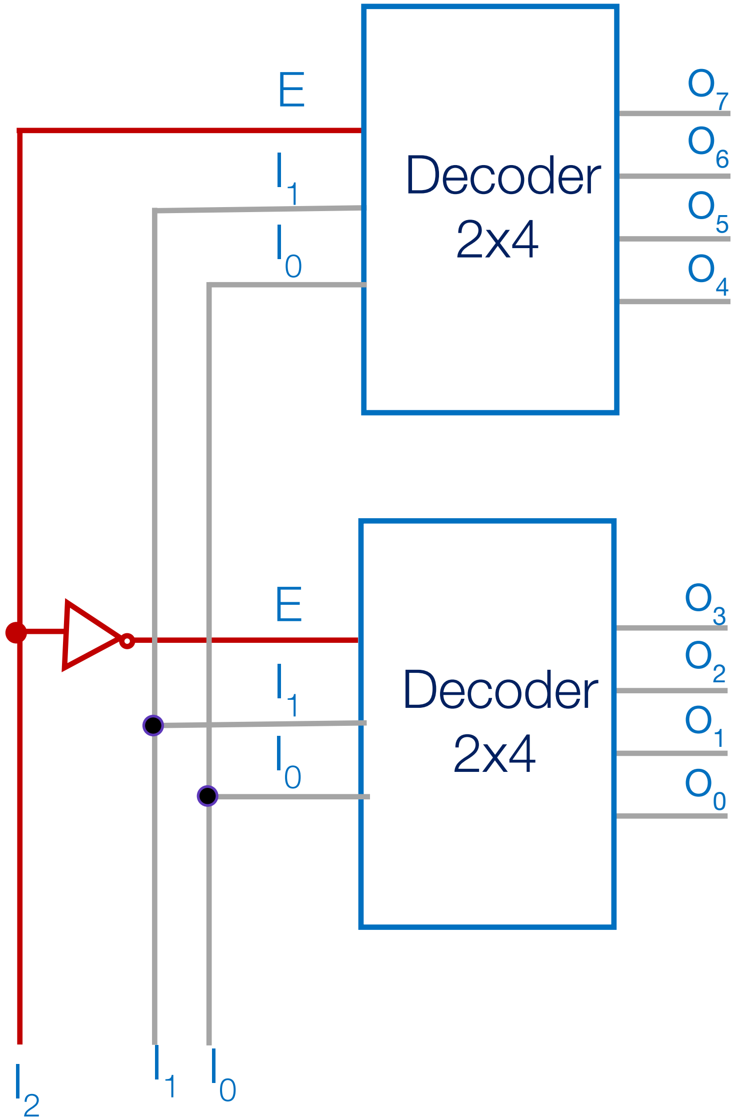
$= \bar{I}_2 \bar{I}_1 \bar{I}_0$

Decoders

- Expansion

lsb

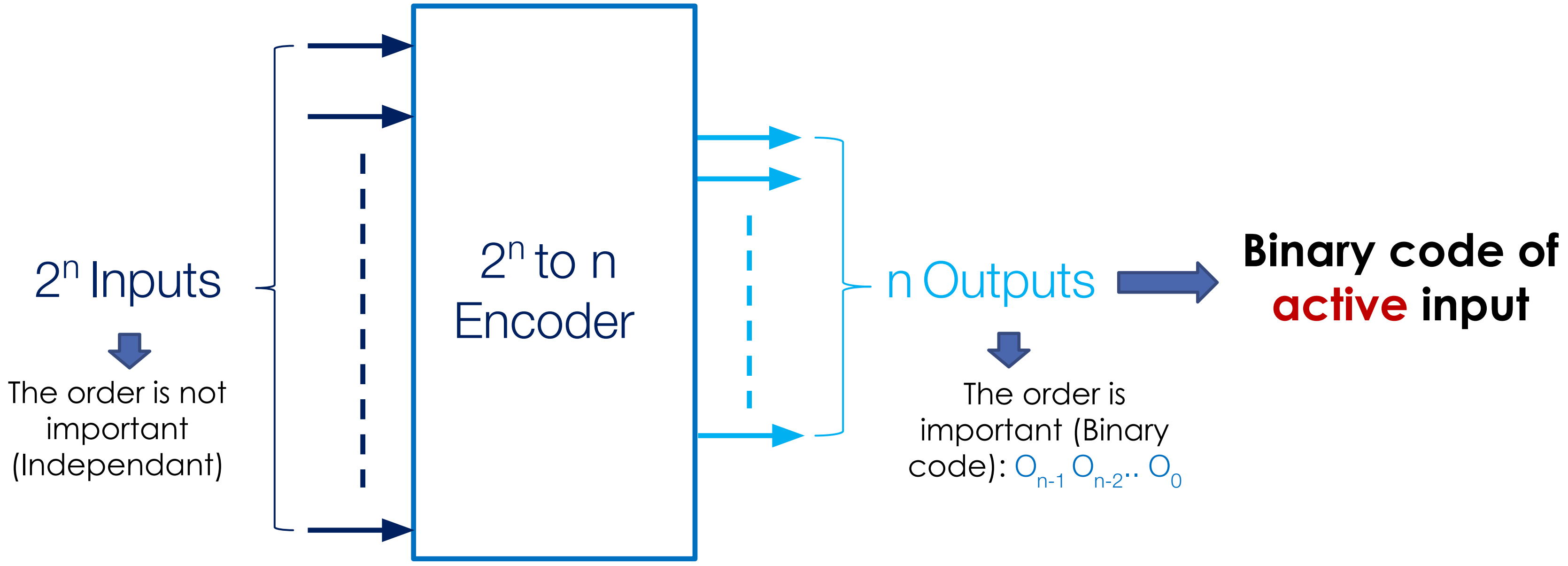
I_2	I_1	I_0	o_0	o_1	o_2	o_3	o_4	o_5	o_6	o_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



Encoders

Encoders

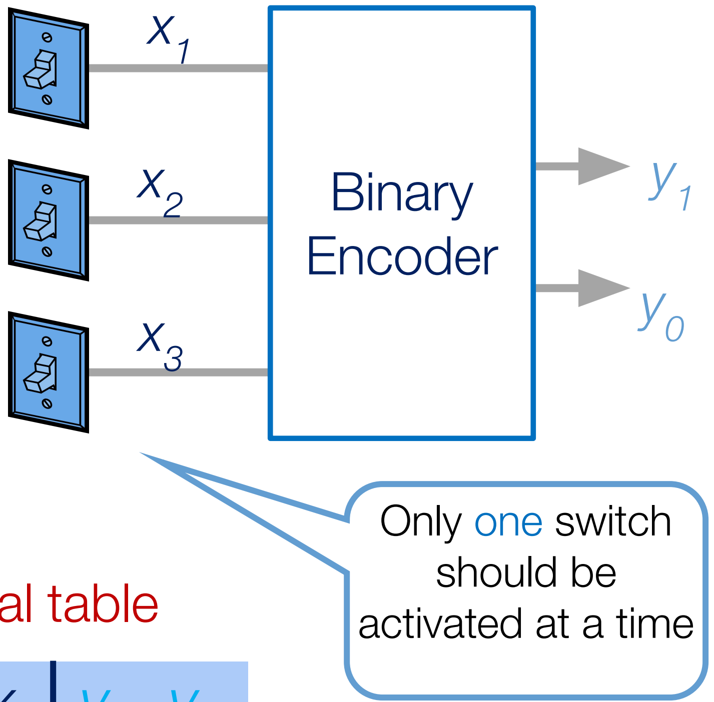
- An encoder performs the inverse operation of a decoder.
- It has 2^n inputs (information), and **n outputs** (Code of information).
- Only **one input** can be logic 1 at any given time (active input). All other inputs must be 0's.
- Output lines generate the binary code corresponding to the active input.



2^n Inputs
The order is not important (Independant)
2^n to n Encoder
n Outputs
The order is important (Binary code): $O_{n-1}\,O_{n-2}..\,O_0$
Binary code of active input

Encoders

- Put “Information” into code
- Binary Encoder
  - Example: 4-to-2 Binary Encoder



Truth table

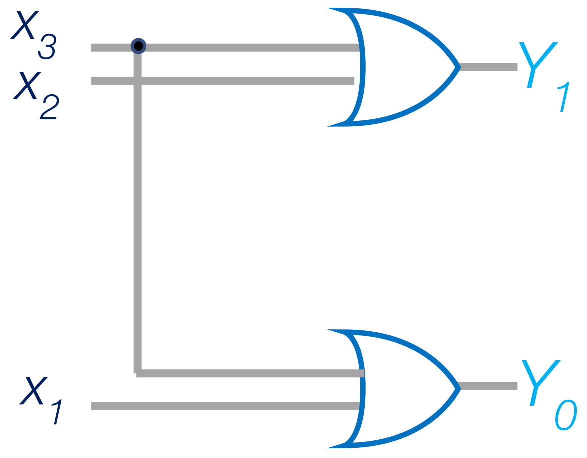
lsb

x_3	x_2	x_1	y_1	y_0	
0	0	0	**0**	**0**	
0	0	1	**0**	**1**	**Code of x_1**
0	1	0	**1**	**0**	**Code of x_2**
0	1	1	x	x	
1	0	0	**1**	**1**	**Code of x_3**
1	0	1	x	x	
1	1	0	x	x	
1	1	1	x	x	

Functional table

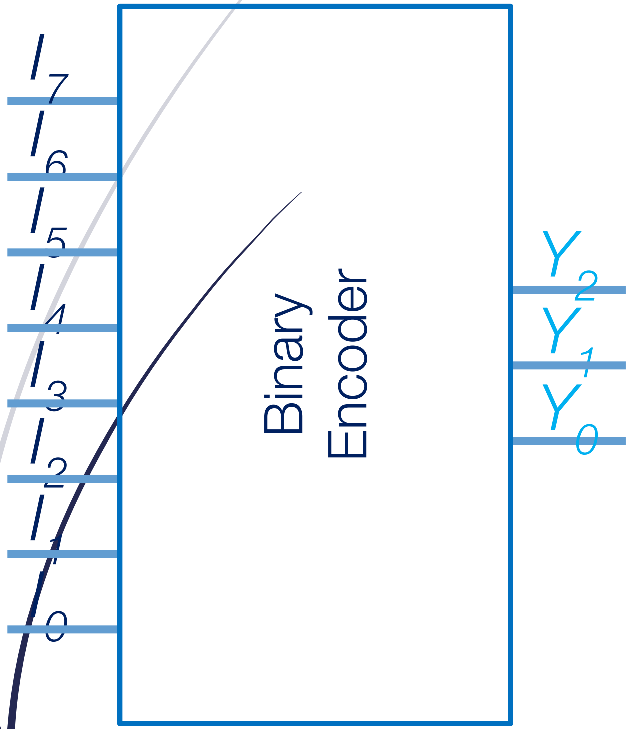
x_3	x_2	x_1	y_1	y_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
1	0	0	1	1

$$y_1 = x_2 + x_3$$
$$y_0 = x_1 + x_3$$



Encoders

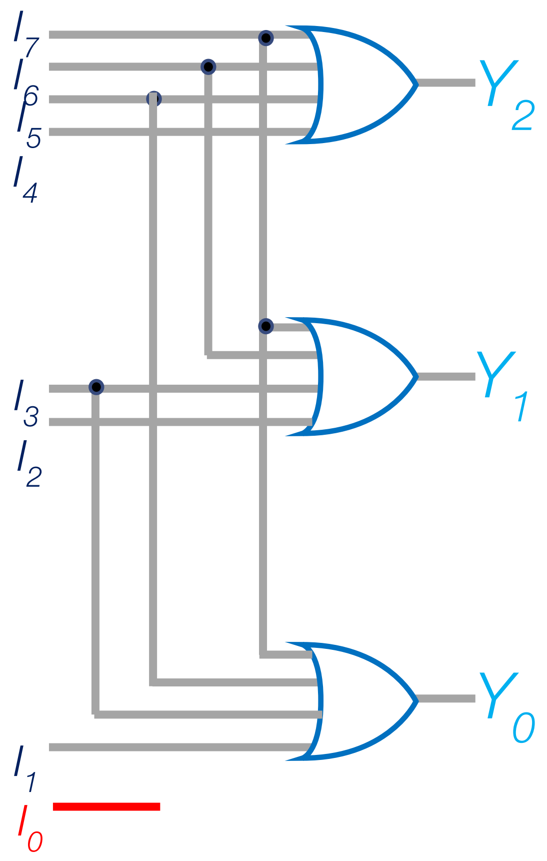
- Octal-to-Binary Encoder (8 to 3)



lsb

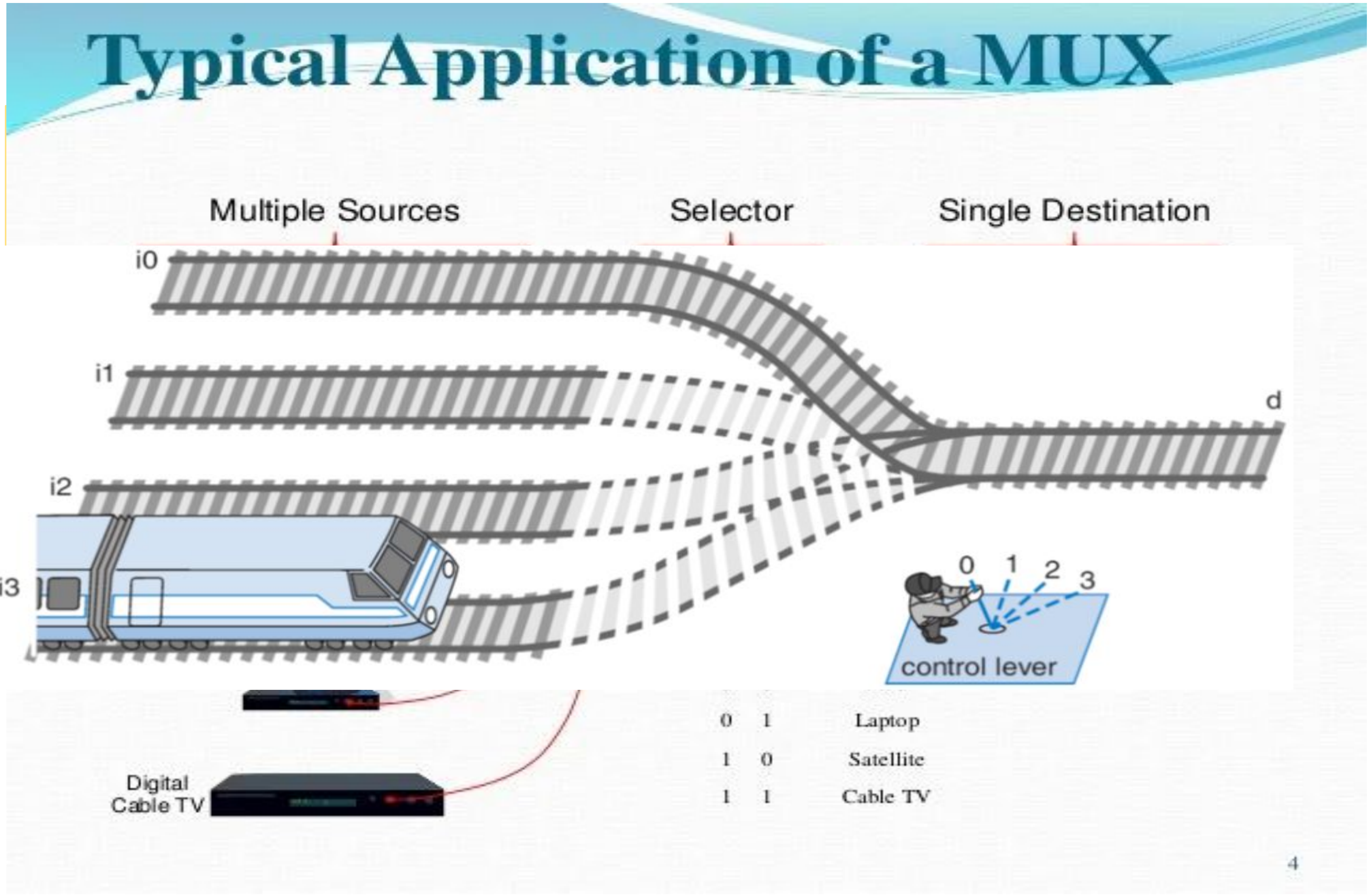
I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$$Y_2 = I_7 + I_6 + I_5 + I_4$$
$$Y_1 = I_7 + I_6 + I_3 + I_2$$
$$Y_0 = I_7 + I_5 + I_3 + I_1$$



Multiplexer

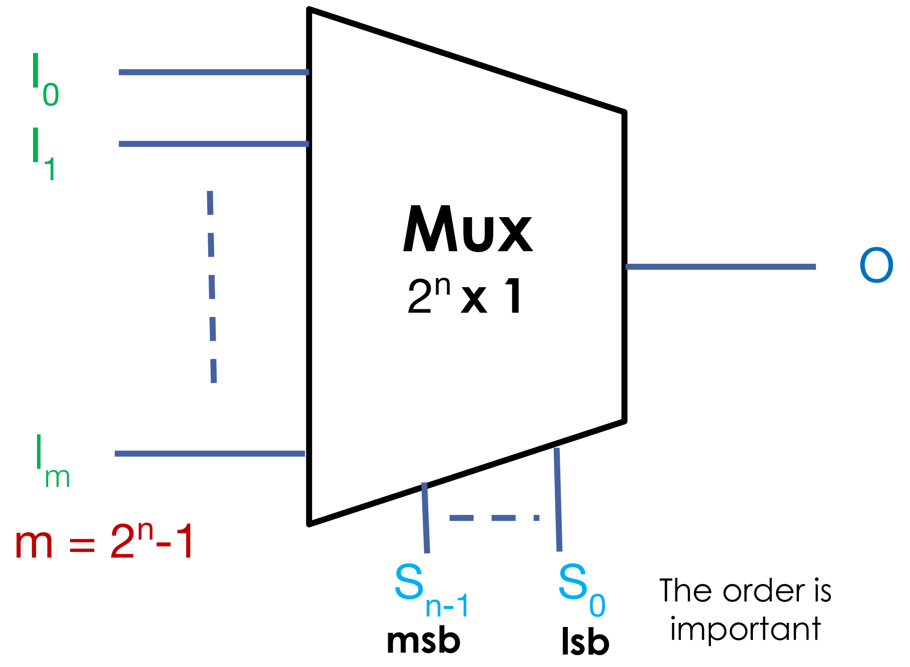
Example



0	1	Laptop
1	0	Satellite
1	1	Cable TV

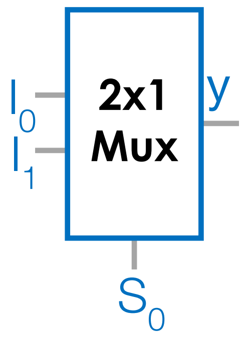
Multiplexer

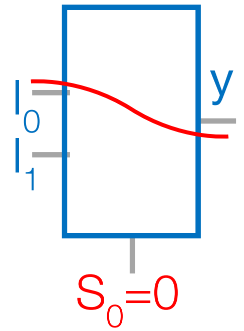
- The multiplexer or *data selector* , shortened to "MUX" is a logic circuit that accepts several ***Data inputs*** and selects one of them at any given time to pass on to the ***output***.
- This selection is controlled by ***Select inputs*** (often referred to as ***Control*** or ***Address inputs***).
    - 4 input mux needs 2 select inputs to indicate which input to route through,
    - 8 input mux 3 select inputs,
    - 2^n inputs n select inputs
- Multiplexers can also be used to implement <u>Boolean functions</u>

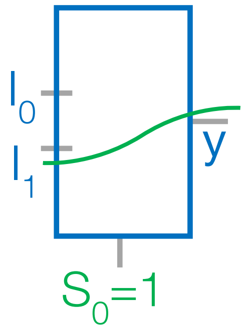


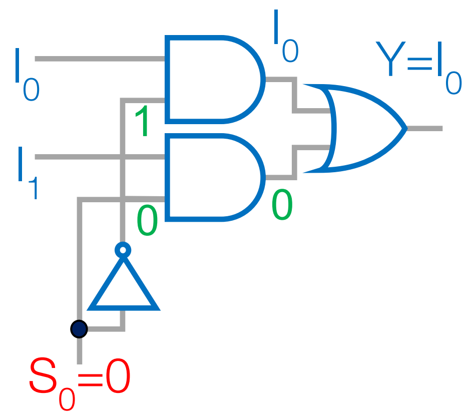
Mux Internal Design

- Mux 2x1

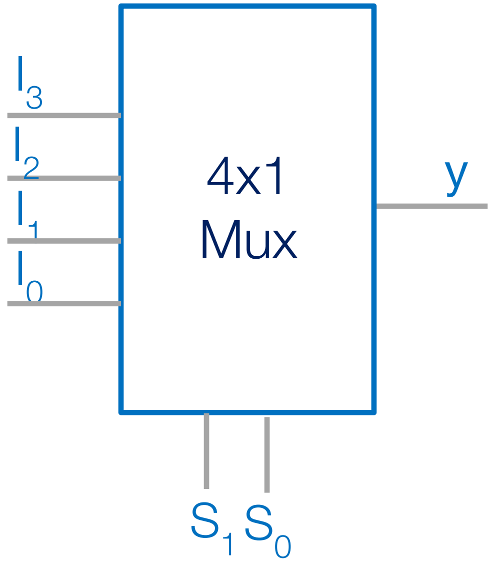




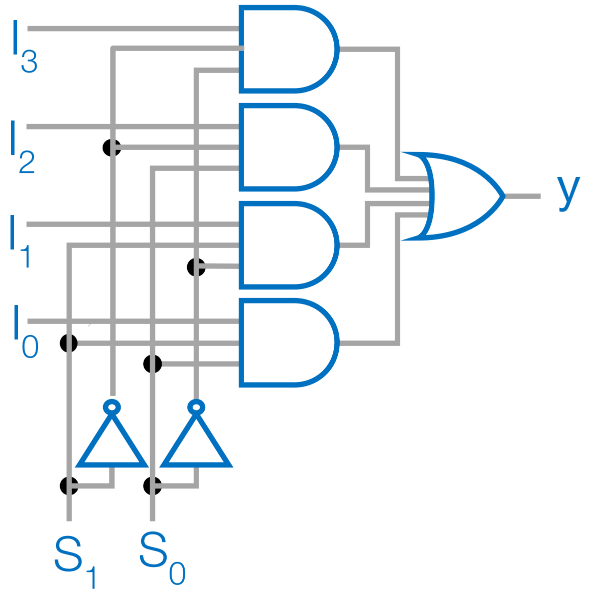




- Mux 4x1

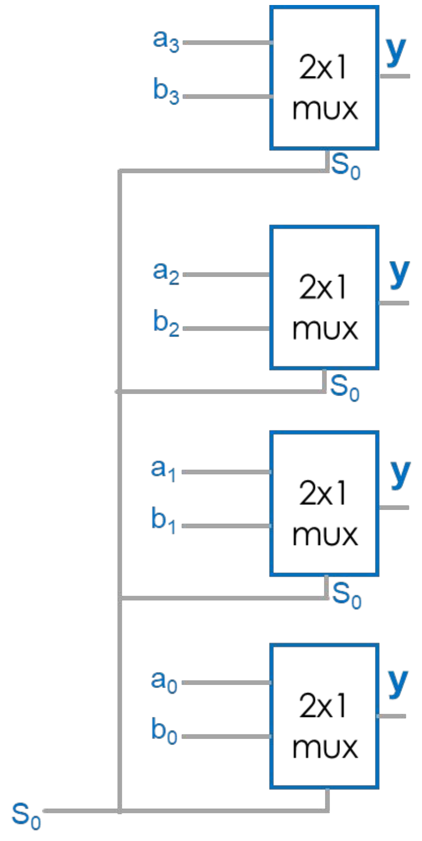


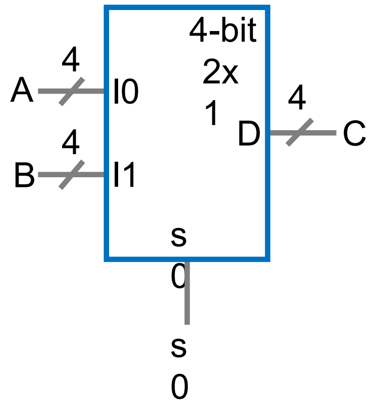
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



Muxes Commonly Together -- N-bit Mux

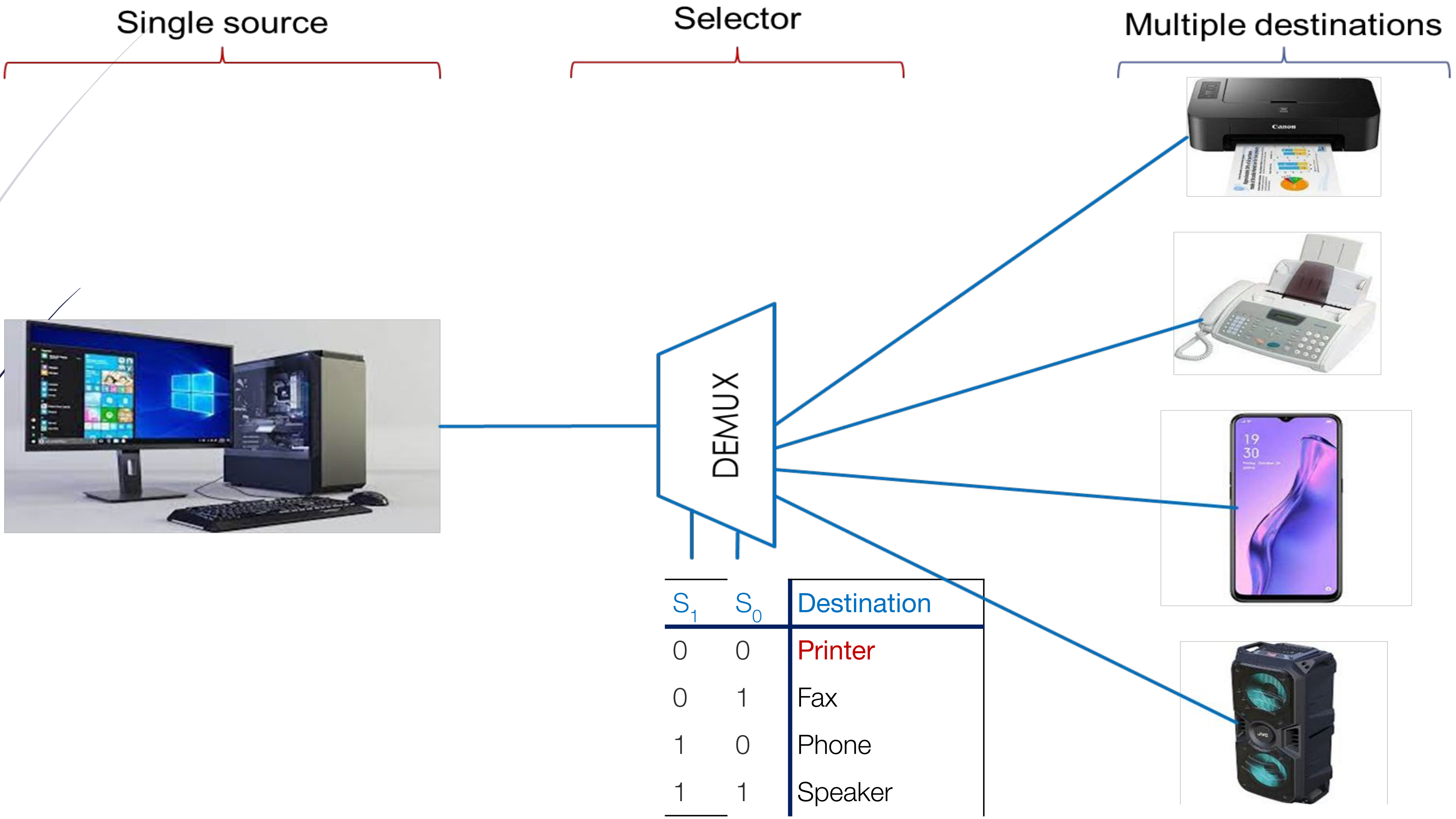
- Ex: Two 4-bit inputs, A (a3 a2 a1 a0), and B (b3 b2 b1 b0)
  - 4-bit 2x1 mux (just four 2x1 muxes sharing a select line) can select between A or B





Demultiplexer

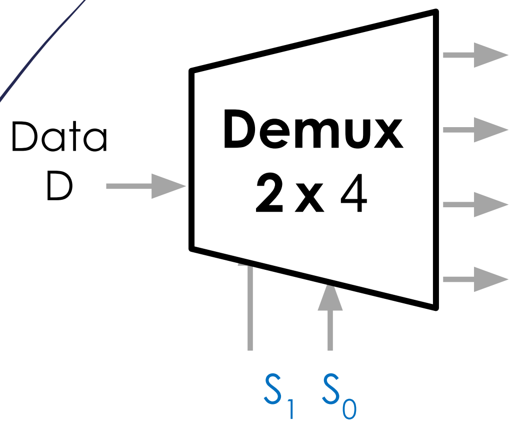
Demultiplexer



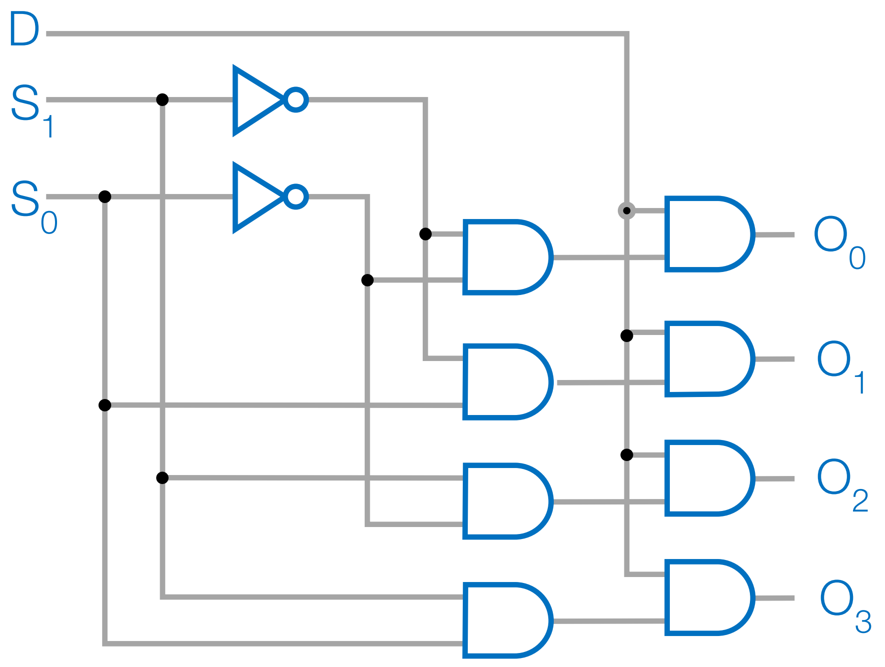
S_1	S_0	Destination
0	0	Printer
0	1	Fax
1	0	Phone
1	1	Speaker

Demultiplexer

- Demultiplexer (Demux) or Data distributors performs the inverse operation of a Mux.
- Given an input line and a set of selection lines, the demultiplexer will direct data from input to a selected output line.



S_1	S_0	O_0	O_1	O_2	O_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D



- Data D is transmitted to only one of the outputs as determined by select input code S_1S_0,