


**SAPIENZA**  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

**Encoder, Decoder, ROM, PLA  
Multiplexer and Demultiplexer**

Prof. Daniele Gorla



**SAPIENZA**  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

### Encoder (4-to-2)

**Input:** 4 input lines, just one of them can hold "1" at any time  
**Output:** 2 output lines that yield the binary coding of the input line that holds "1"

$x_3$	$x_2$	$x_1$	$x_0$	$y_1$	$y_0$
0	0	0	0	-	-
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	-	-
0	1	0	0	1	0
0	1	0	1	-	-
0	1	1	0	-	-
0	1	1	1	-	-
1	0	0	0	1	1
1	0	0	1	-	-
1	0	1	0	-	-
1	0	1	1	-	-
1	1	0	0	-	-
1	1	0	1	-	-
1	1	1	0	-	-
1	1	1	1	-	-

$x_3$	$x_2$	00	01	11	10
$x_1 x_0$					
00	-	1	-	1	-
01	0	-	-	-	-
11	-	-	-	-	-
10	0	-	-	-	-

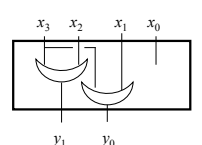
$y_1 = x_3 + x_2$


$x_3$	$x_2$	00	01	11	10
$x_1 x_0$					
00	-	0	-	1	-
01	0	-	-	-	-
11	-	-	-	-	-
10	1	-	-	-	-

$y_0 = x_3 + x_1$

**OR Matrix:**

$x_3$	$x_2$	$x_1$	$x_0$	
○	○	○	○	$y_1$
○	○	○	○	$y_0$





**SAPIENZA**  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

### Encoder $2^n$ -to- $n$ Generalized Encoder

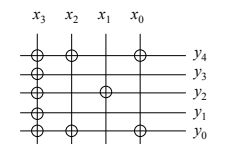
**Encoder 8-to-3:**


$x_7$	$x_6$	$x_5$	$x_4$	$x_3$	$x_2$	$x_1$	$x_0$	
○	○	○	○	○	○	○	○	$y_2$
○	○	○	○	○	○	○	○	$y_1$
○	○	○	○	○	○	○	○	$y_0$

**Encoder  $2^n$ -to- $n$ :**  
 The vertical line associated to  $x_i$  contains the binary code of  $i$ , by considering to circle as a 1 and absence of circle as 0.

**Generalized Encoder**  
**Input:**  $n$  lines, just one of them holds "1" at any time  
**Output:** the ( $m$  bits) binary coding of  $f(i)$ , where  $f$  is fixed and  $i$  is the line that holds 1

Ex.:	$x_3$	$x_2$	$x_1$	$x_0$	$y_4$	$y_3$	$y_2$	$y_1$	$y_0$
$f: 0 \rightarrow 17$	0	0	0	1	1	0	0	0	1
1 $\rightarrow$ 4	0	0	1	0	0	0	1	0	0
2 $\rightarrow$ 17	0	1	0	0	1	0	0	0	1
3 $\rightarrow$ 31	1	0	0	0	1	1	1	1	1





**SAPIENZA**  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

### Decoder 2-to-4

**Input:** 2 input lines  
**Output:** 4 output lines, just one of them can be "1";  
 line  $i$  holds "1" whenever the input is the binary coding of  $i$ .

$x_1$	$x_0$	$y_3$	$y_2$	$y_1$	$y_0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$y_3 = x_1 x_0 = m_3$

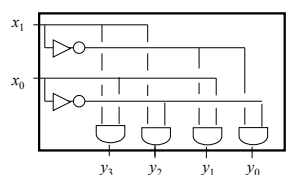
$y_2 = x_1 \bar{x}_0 = m_2$

$y_1 = \bar{x}_1 x_0 = m_1$

$y_0 = \bar{x}_1 \bar{x}_0 = m_0$

**AND Matrix:**

$x_1$	$x_0$	$y_3$	$y_2$	$y_1$	$y_0$
●	●	●	○	○	○
○	●	○	●	○	○
○	○	○	○	●	○
●	○	○	○	○	●



Easy to generalize to  $n$ -to- $2^n$

**ROM**

A ROM (Read Only Memory) is a circuit with  $n$  inputs (also called *address lines*) and  $m$  outputs (also called *data lines*).

Within a ROM, the address lines select one of the  $2^n$  rows of a  $2^n \times m$  matrix. Selecting row  $i$ -th allows us to read, on each of the  $m$  data lines, the binary value stored in the cell of coordinates  $(i, j)$ , for  $j \in \{1, \dots, m\}$ .

It can be seen as the composition of a decoder and of a generalized encoder:

i.e., an AND matrix (whose inputs are the address lines) whose outputs are the inputs an OR matrix (whose outputs are the data lines).

**Realizing FBs through a ROM**

$x_1$	$x_0$	$y_4$	$y_3$	$y_2$	$y_1$	$y_0$
0	0	1	0	0	0	1
0	1	0	0	1	0	0
1	0	1	0	0	0	1
1	1	1	1	1	1	1

A ROM can be used to realize  $m$  BFs  $\{0,1\}^n \rightarrow \{0,1\}$

Through a  $n$ -to- $2^n$  DEC (that is a  $2^n \times m$  matrix), we "copy" the rightmost part of the truth table into the OR matrix of a generalized encoder  $2^n$ -to- $m$ .

On the exit associated to the minterm  $m_i$  of the decoder (AND matrix), we put a  $\square$  corresponding to every 1 in the  $i$ -th row of the TT.

Physically, it is a *diode*, i.e. an elemental circuit that sends a signal from the exits of the DEC to the corresponding data line only if the signal is 1.

**ROM as a read-only memory**

The name ROM (read-only memory) derives from the fact that this combinatorial module can be seen as:

- A memory (i.e., a set of cells with fixed size, each with its own address)
- Non rewritable (hence, read-only)

Ex.:

In the cell with address 2 ( $10_2$ ) it was stored the number 17 ( $10001_2$ )

The memory is read only because, once diodes have been welded, the stored values can be changed only by creating a different circuit (i.e., by welding diodes in a different way)

memory cells (5 bits)

**ROM with a DECODER**

**REMARK:** often, the decoder is not explicitly given (it is written as a black box). In this way, we only need to fill in the OR matrix, that for this reason is called the *ROM matrix*.

$x_2$	$x_1$	$x_0$	$y_1$	$y_0$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	1

**OBS.1:** a ROM realizes the DCF of the BFs

**OBS.2:** some minterms are calculated in the DEC but are never used in the ROM matrix (e.g.:  $m_0$ ,  $m_2$  and  $m_4$  in the example above)

**PLA**

A PLA (Programmable Logic Array) is an integrated combinatorial net with  $n$  inputs,  $m$  outputs and three inner layers: a complementation layer, an AND matrix and an OR matrix.

→ like a ROM

A PLA allows us to implement Bes in DNF, specifically minimal DNFs by allowing also the AND matrix (and not only the OR one) to be programmed

→ more efficient and cheaper than a ROM

**Programming a PLA**

A PLA is sold with all the  $n$  inputs both affirmed and negated; moreover, it has  $K$  AND gates and  $m$  OR gates whose inputs are not linked to anything.

Links are done by the user according to the specifications given, expressed as BEs in DNF.

The user should then give the TT and find the minimal DNFs for each of the  $m$  BF's (more laborious than a ROM!!)

**Example**

$x_1$	$x_0$	$y_3$	$y_2$	$y_1$	$y_0$
0	0	0	0	0	1
0	1	0	1	0	0
1	0	0	0	0	1
1	1	1	1	1	1

$y_3 = y_1 = x_1 x_0$   
 $y_2 = x_0$   
 $y_0 = x_1 + \bar{x}_0$

**Multiplexer (strict sense)**

**Input:**  $n$  data lines and  $n$  control lines, just one of which holds "1" at every time  
**Output:** 1 line that returns the value on the  $i$ -th data line, if the  $i$ -th control line holds 1.

Ex. ( $n = 2$ )

$x_1$	$x_0$	$k_1$	$k_0$	$y$
0	0	0	1	0
0	1	0	1	1
1	0	0	1	0
1	1	0	1	1
0	0	1	0	0
0	1	1	0	0
1	0	1	0	1
1	1	1	0	1

$y = x_1 k_1 + x_0 k_0$

REMARK: lines with  $k_1 = k_0$  are don't care

In general:

**Multiplexer**

A **multiplexer (MUX)** is a combinatorial net with  $n$  inputs, one output and  $\log_2 n$  control lines.

At every moment, output  $y$  is equal to the value of one of the inputs,  $x_i$ . The value of  $i$  is given by the control lines: it is the value (expressed as a natural number) that such lines codify in binary.

A MUX is formed by a multi-entrance OR gate, the receives the outputs of  $n$  AND gates, that work like interruptors; finally, a DEC activates the interruptor selected by the control lines.

REMARK: it is a strict sense multiplexer whose control lines are the outputs of the DEC.

**Demultiplexer (strict sense)**

**Input:** 1 data line and  $n$  control lines, just one holds "1" at every moment  
**Output:**  $n$  lines, where the  $i$ -th one holds the value of the data line if the  $i$ -th control line holds 1.

Ex. ( $n = 2$ )

$x$	$k_1$	$k_0$	$y_1$	$y_0$
0	0	1	0	0
1	0	1	0	1
0	1	0	0	0
1	1	0	1	0

$y_1 = xk_1$   
 $y_0 = xk_0$

In general:

REMARK: rows with  $k_i = k_0$  are *don't care*

**Demultiplexer**

A **demultiplexer (DEMUX)** is a combinatorial net with one input,  $n$  outputs and  $\log_2 n$  control lines.

At every moment, the output  $y_i$  equals the input, where the value of  $i$  is given by the control lines: it is the value (expressed as natural number) that the lines codify in binary.

A DEMUX is made up by  $n$  AND gates, that work like interruptors, and a DEC that activated the interruptor selected by the control lines.

REMARK: it is a demultiplexer in strict sense whose control lines are the outputs of the DEC.

**Use of MUX/DEMUX**

- Parallel/serial transmission (MUX) and serial/parallel one (DEMUX)  
 → at given time intervals, we increase the control lines:

REMARK: to do this timing, we need a circuit (called *counter*) that we shall meet at the end of this course

- Use MUXs to compute BFs

**MUX to compute a BF (1)**

From the construction of a MUX, we have that

$$y = \sum_{i=0}^{2^n-1} x_i \cdot m_i = \sum_{i: x_i=1} m_i$$

Recall that an  $n$  variables BF in FCD is  $f = \sum_{i: f(i_2)=1} m_i$

So, given a BF with  $n$  variables:

- Use a MUX with  $2^n$  inputs;
- The  $n$  control lines of the MUX are the  $n$  variables of the BF;
- The  $2^n$  data lines are put to "0" or "1" according to what is specified in the TT.

**Example**

$x_2$	$x_1$	$x_0$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

**MUX to compute a FB (2)**

To compute an  $n$  variables BF, we can also use a MUX with less than  $n$  control lines; in this case, some variables will be control lines and some others will be used in some BEs in the data lines (that are no more simply 0 or 1, in general).

Ex. ( $n = 3$ ):  $f = \sum_{i=0}^7 f(i_2) \cdot m_i =$

$$= f(000) \cdot \bar{x} \cdot \bar{y} \cdot \bar{z} + f(001) \cdot \bar{x} \cdot \bar{y} \cdot z + f(010) \cdot \bar{x} \cdot y \cdot \bar{z} + f(011) \cdot \bar{x} \cdot y \cdot z +$$

$$f(100) \cdot x \cdot \bar{y} \cdot \bar{z} + f(101) \cdot x \cdot \bar{y} \cdot z + f(110) \cdot x \cdot y \cdot \bar{z} + f(111) \cdot x \cdot y \cdot z =$$

$$= (f(000) \cdot \bar{z} + f(001) \cdot z) \cdot \bar{x} \cdot \bar{y} + (f(010) \cdot \bar{z} + f(011) \cdot z) \cdot \bar{x} \cdot y +$$

$$(f(100) \cdot \bar{z} + f(101) \cdot z) \cdot x \cdot \bar{y} + (f(110) \cdot \bar{z} + f(111) \cdot z) \cdot x \cdot y$$