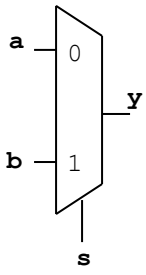
# Notes - Unit 5

## COMBINATIONAL CIRCUITS

### MULTIPLEXERS (MUXS)

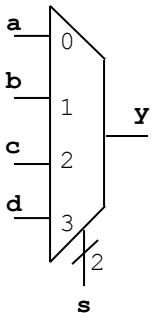
- This logic circuit selects one of many input signals and forwards the selected input to the output line.
- Boolean equations for MUX2-to-1, MUX4-to-1, MUX8-to-1:



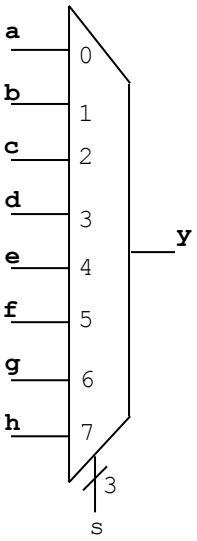

| s | a | b | y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

| s | y |
|---|---|
| 0 | a |
| 1 | b |

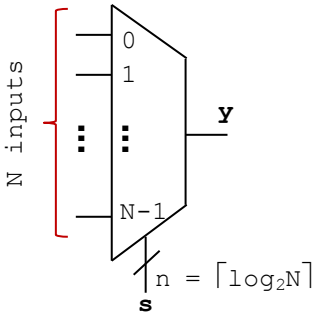
$$y = \bar{s}a + sb$$




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




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



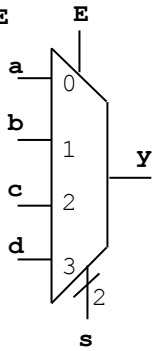

- Normally, a multiplexer has $N = 2^n$ inputs, one output, and a selector with $n$ bits.
- But, if a multiplexer has $N$ inputs, where $N$ is not a power of 2, the number of bits of the selector is given by: $\lceil log_2 N \rceil$.

**MULTIPLEXERS WITH ENABLE**

- An enable input provides us with an extra level of control. If the multiplexer is enabled, the circuit just 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:

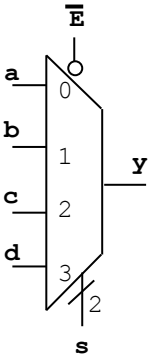
**ACTIVE HIGH ENABLE**

| 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 |



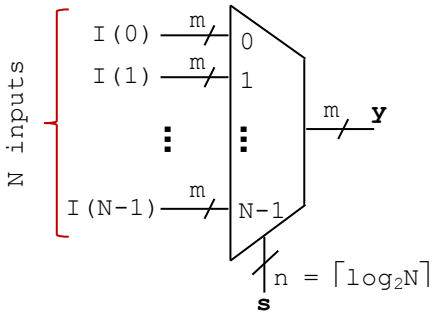

**ACTIVE LOW ENABLE**

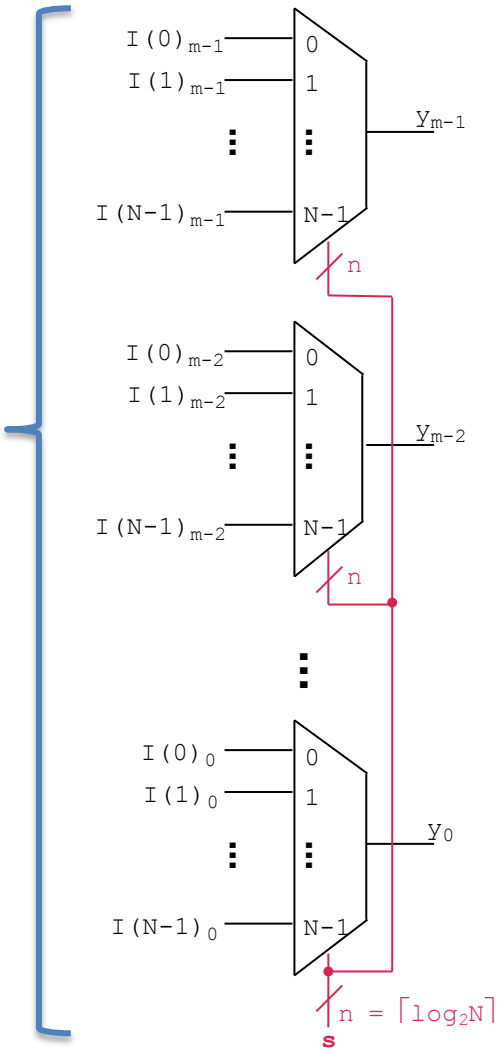
| $\bar{E}$ | $s_1$ | $s_0$ | y |
|---|---|---|---|
| 0 | 0 | 0 | a |
| 0 | 0 | 1 | b |
| 0 | 1 | 0 | c |
| 0 | 1 | 1 | d |
| 1 | X | X | 0 |



## 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.

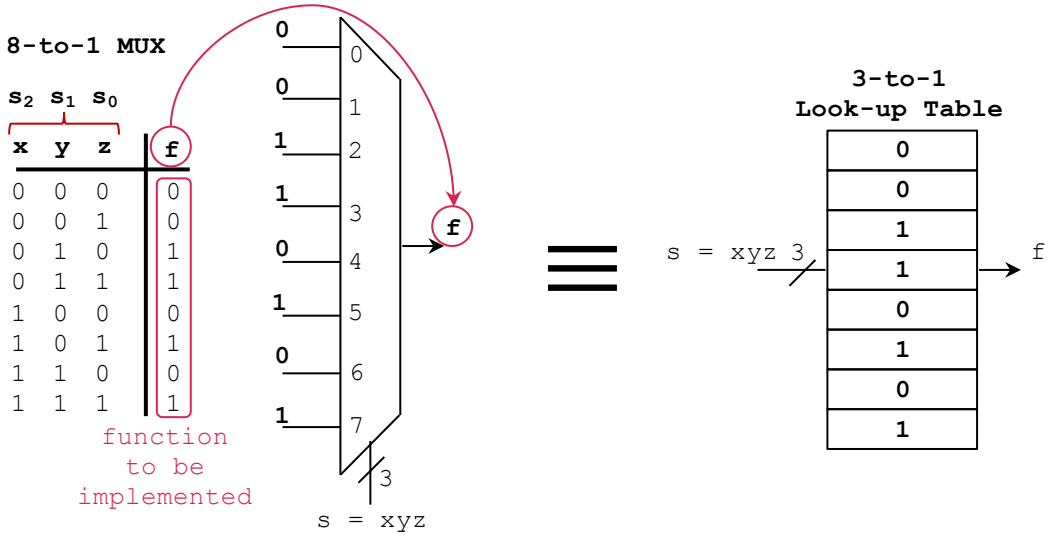



- We have 'N' inputs and therefore the selector has $n = \lceil log_2 N \rceil$ bits.
- Note that the selector is the same for all the multiplexers.

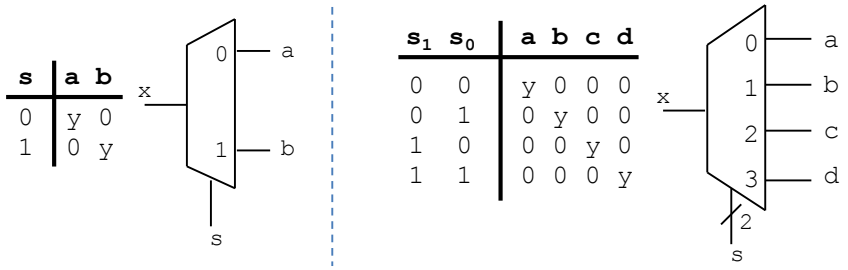
## LOGIC CIRCUITS WITH MUXs

- 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.



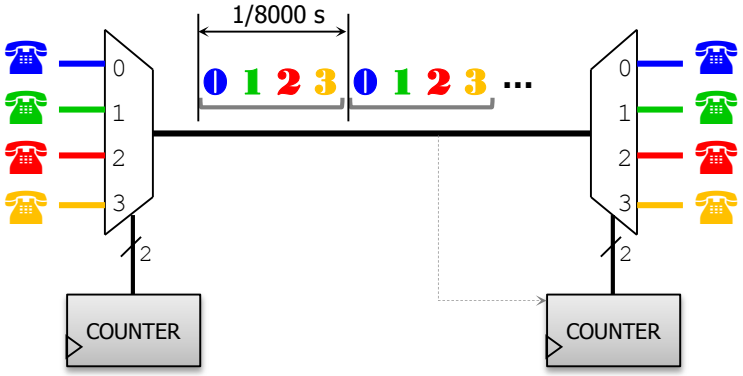
## DEMULTIPLEXERS

- A demultiplexer performs the opposite operation of the multiplexers.



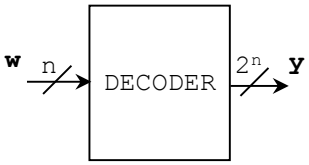

**Application**: Time Division Multiplexing (TDM)

- Digital Telephony: (4 KHz bandwidth)
- 8000 samples per second, 8 samples per second. 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).

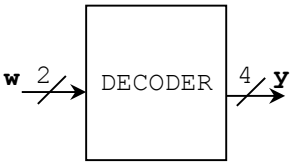



## DECODERS

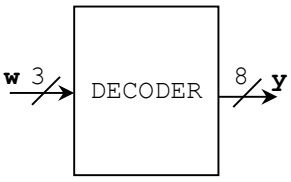
- Generally speaking, decoders are circuits that transform the inputs into outputs following a certain rule, provided that the number of outputs is greater than or equal to the number of inputs.
- Here, we discuss standard decoders for which a specific input/output rule exists. These decoders have $n$ 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 $i$.



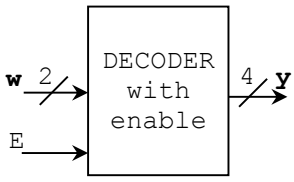

| $w_1$ | $w_0$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |




| $w_2$ | $w_1$ | $w_0$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

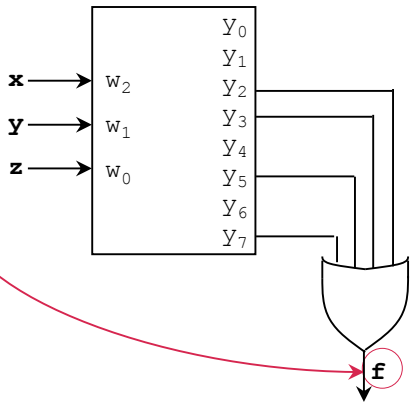



## LOGIC CIRCUITS WITH DECODERS

- 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.

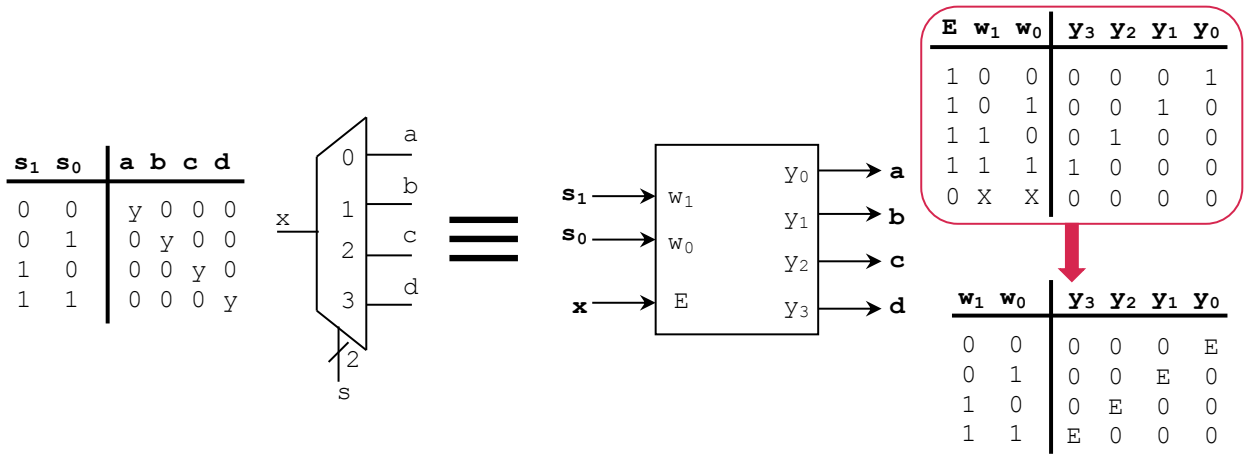
| $w_2$ | $w_1$ | $w_0$ | f |
|---|---|---|---|
| x | y | z | |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

function to be implemented



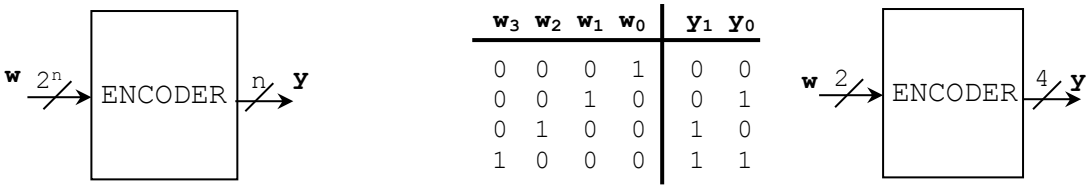
## IMPLEMENTING DEMULTIPLEXORS WITH DECODERS

- By utilizing the enable input of a decoder as our input signal, we can effectively implement a demultiplexor using a decoder:



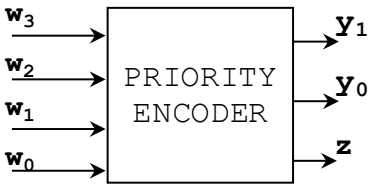

## ENCODERS

- 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 exactly the opposite as in the case of the decoder: whenever an input $w_i$ is activated, then the index $i$ appears at the output $y$ (in binary form).




### PRIORITY ENCODERS

- Standard encoder: we check whether an specific input is activated for the output to have a value.
- What happens when more than one input is activated? A solution is to create an extra output that is activated to indicate than an unexpected condition has occurred.
- Another more interesting solution is to create a ***priority encoder***, that is 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 |

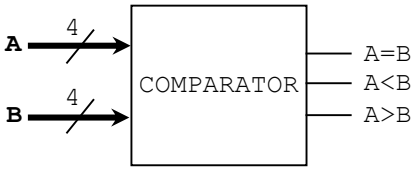
- What happens when no input is activated? Here we run out of output bits in $y$ to represent this case. Thus, we include an extra output $z$ that it is '0' when no input activated, and '1' otherwise.

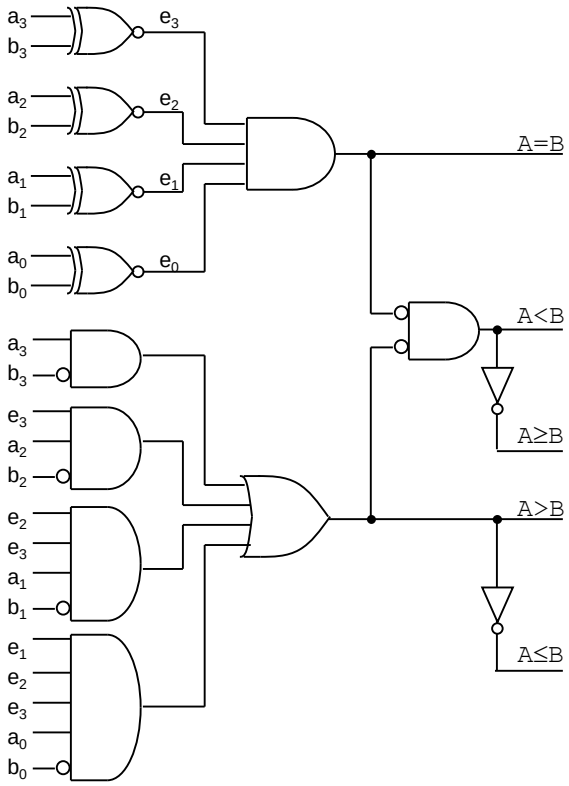
## COMPARATORS

- **Comparator for Unsigned Numbers**:

  For $A = a_3a_2a_1a_0$, $B = b_3b_2b_1b_0$

  ✓ $A > B$ when:
  $a_3 = 1, b_3 = 0$
  Or: $a_3 = b_3$ and $a_2 = 1, b_2 = 0$
  Or: $a_3 = b_3, a_2 = b_2$ and $a_1 = 1, b_1 = 0$
  Or: $a_3 = b_3, a_2 = b_2, a_1 = b_1$ and $a_0 = 1, b_0 = 0$

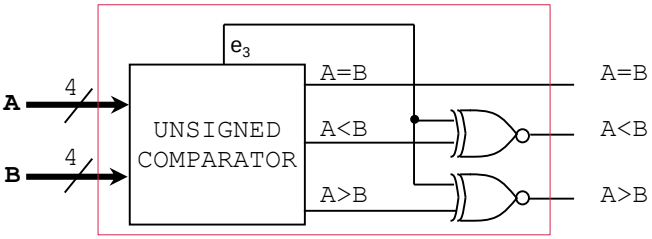



- **Comparator for Signed Numbers**:
  ✓ If both numbers are positive, we can use the unsigned comparator
  ✓ If both numbers are negative, we can also use unsigned comparator.
  Example: 1000 < 1001 (-8 < -7). The closer the number is to zero, the larger the unsigned value is.
  ✓ If one number is positive and the other negative:
  Example: 1000 < 0100 (-8 < 4). However, if we were to use the unsigned comparator, we would get 1000 > 0100. So, we just need to invert the A>B bit (and the A<B bit too) in this case.

  ✓ For a 4-bit number in 2's complement:
  – If $a_3 \neq b_3$, we need to invert the A>B and A<B bits of the unsigned comparator.
  – If $a_3 = b_3$, we do not invert any bit.

  $e_3 = 1$ when $a_3 = b_3$. $e_3 = 0$ when $a_3 \neq b_3$.
  Then it follows that: $(A < B)_{signed} = \overline{e_3} \oplus (A < B)_{unsigned} = \overline{e_3 \oplus (A < B)_{unsigned}}$
  $(A > B)_{signed} = \overline{e_3 \oplus (A > B)_{unsigned}}$

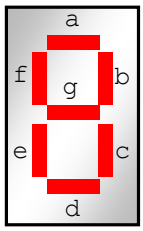


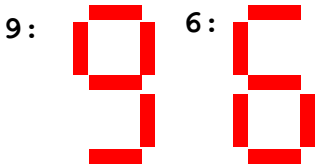
# CODE CONVERTERS

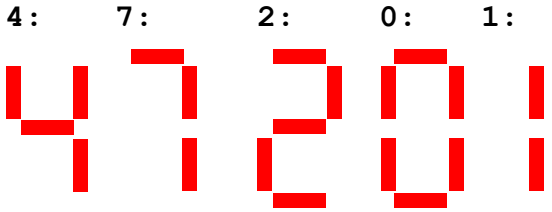
## BCD TO 7-SEGMENT DECODER

- It is a decoder because the number of outputs is greater than the number of inputs
- The truth table below assumes that the input and output are high-level.

| $b_3$ | $b_2$ | $b_1$ | $b_0$ | 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 |



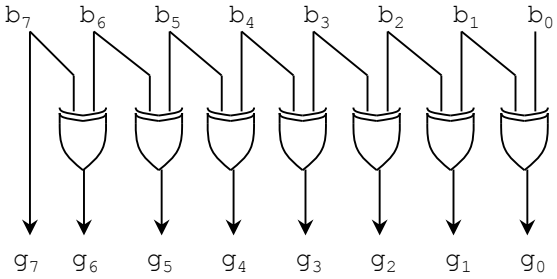

## GRAY TO BCD DECODER

- It is a decoder because the number of outputs is equal to the number of inputs.
- The following is the truth table for a 4-bit case:

| $g_3g_2g_1g_0$ | $b_3b_2b_1b_0$ |
|---|---|
| 0 0 0 0 | 0 0 0 0 |
| 0 0 0 1 | 0 0 0 1 |
| 0 0 1 1 | 0 0 1 0 |
| 0 0 1 0 | 0 0 1 1 |
| 0 1 1 0 | 0 1 0 0 |
| 0 1 1 1 | 0 1 0 1 |
| 0 1 0 1 | 0 1 1 0 |
| 0 1 0 0 | 0 1 1 1 |
| 1 1 0 0 | 1 0 0 0 |
| 1 1 0 1 | 1 0 0 1 |
| 1 1 1 1 | X X X X |
| 1 1 1 0 | X X X X |
| 1 0 1 0 | X X X X |
| 1 0 1 1 | X X X X |
| 1 0 0 1 | X X X X |
| 1 0 0 0 | X X X X |

## BINARY TO GRAY DECODER

- 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, the following circuit is way 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 bits → $p_e$ = 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.

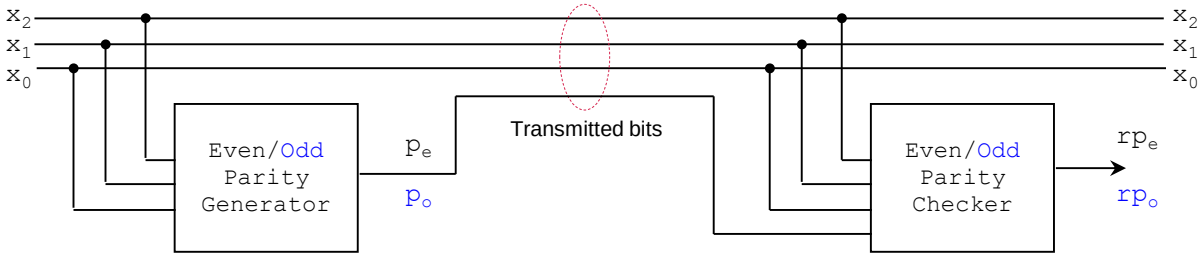
**Example:**

- 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)

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

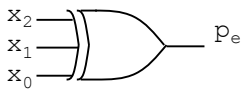
  - ✓ Odd Parity Generator: It generates the parity bit $p_o$.
  - ✓ Odd Parity Checker: It verifies that the received stream Y has odd parity. If so, $rp_o$=0, otherwise $rp_o$=1 (to signal an error)

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



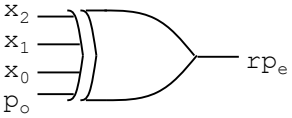

**Even Parity Generator**

| $x_2$ | $x_1$ | $x_0$ | $p_e$ |
|---|---|---|---|
| 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 |



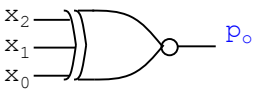

**Even Parity Checker**

| $x_2$ | $x_1$ | $x_0$ | $p_e$ | $rp_e$ |
|---|---|---|---|---|
| 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 |



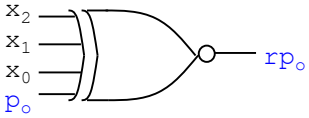

**Odd Parity Generator**

| $x_2$ | $x_1$ | $x_0$ | $p_o$ |
|---|---|---|---|
| 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 |



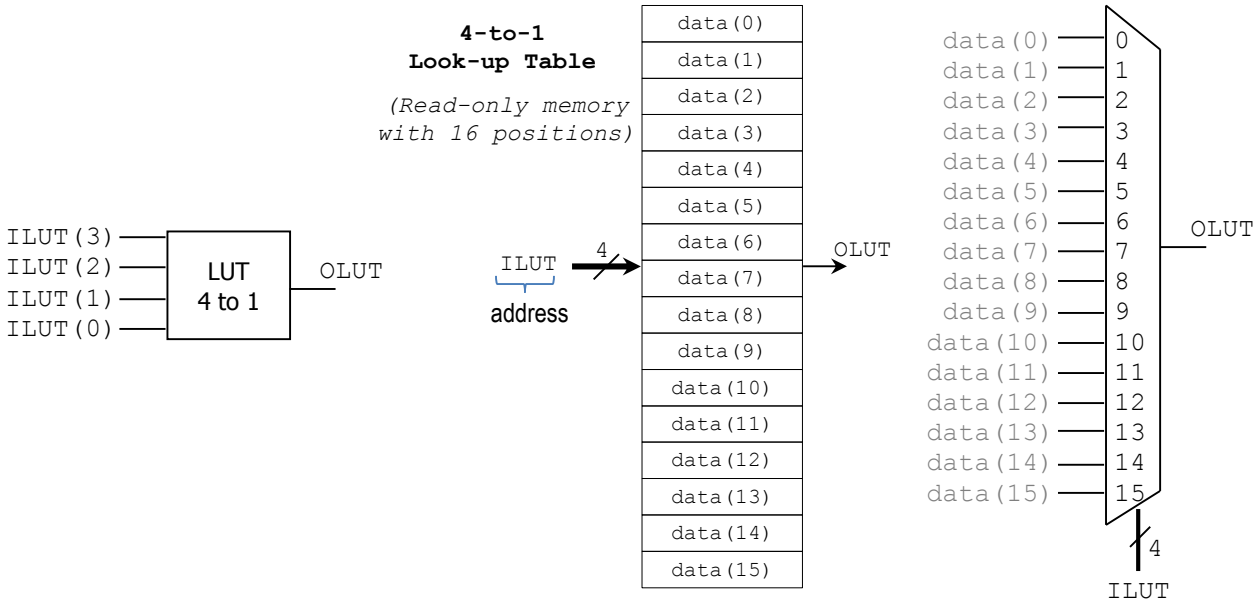

- In general for $X = x_{n-1}x_{n-2}\dots x_1x_0$: $p_e = x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0$. $p_o = \overline{x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0}$
  - ✓ If the # of 1's in an n-bit stream is odd, the n-bit input XOR gate will return 1, 0 otherwise.
  - ✓ If the # of 1's in an n-bit 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.

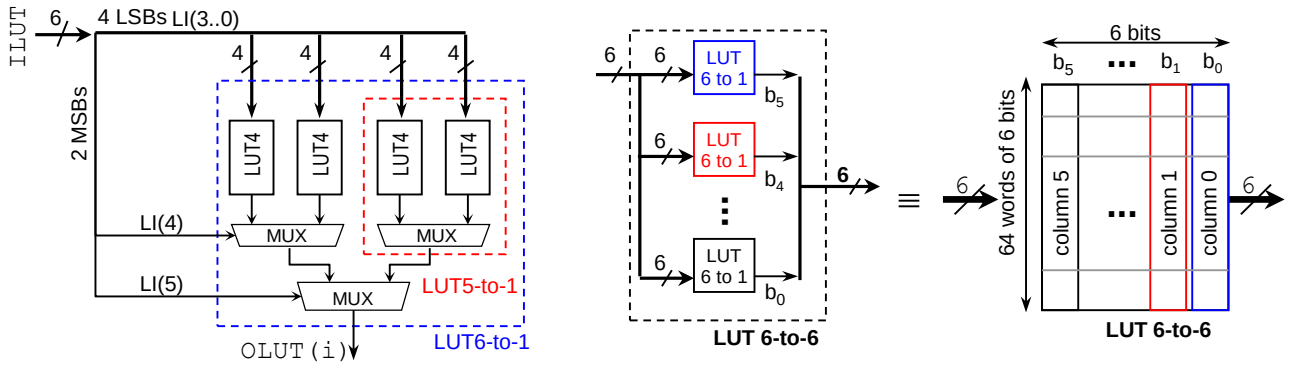
## LOOK-UP TABLES (LUTS)

- 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

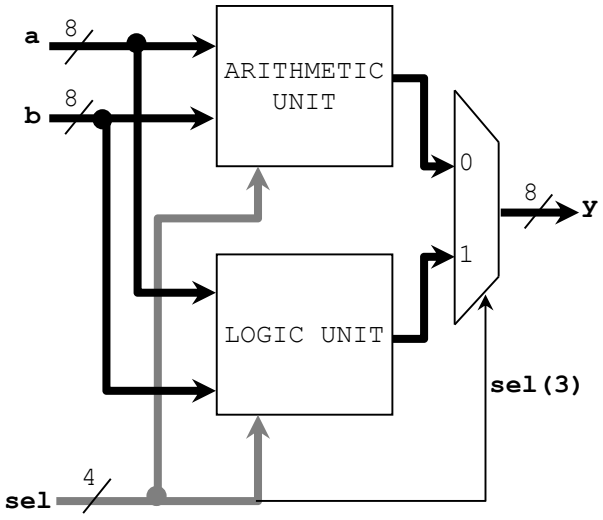
- A larger LUT can be built by building a circuit that allows for more ROM positions.
- Efficient method: A larger LUT can also be built by combining LUTs with multiplexers as shown in the figure. We can build a NI-to-1 LUT with this method.
- We can build a NI-to-NO LUT using NO NI-to-1 LUTs. This can be seen as a ROM with $2^{NI}$ addresses, each address holding $NO$ bits.



## ARITHMETIC LOGIC UNIT (ALU)

- Two types of operation: Arithmetic and Logic (bit-wise). The `sel(3..0)` input selects the operation. `sel(2..0)` selects the operation type within a specific unit. The arithmetic unit consist of adders and subtractors, while the Logic Unit consist of 8-input logic gates.

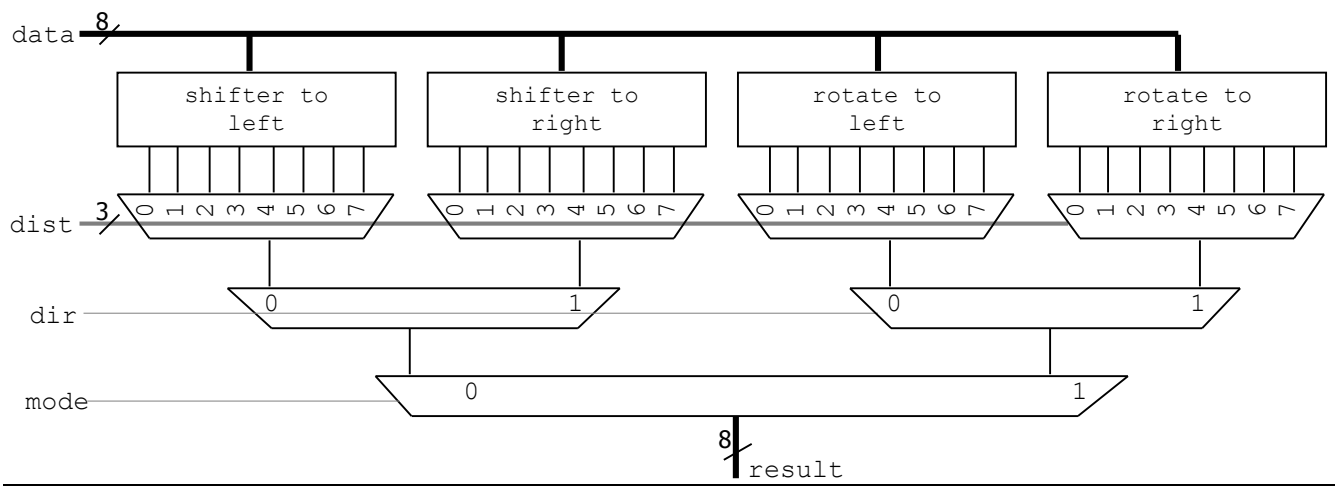



| sel | Operation | Function | Unit |
|---|---|---|---|
| 0 0 0 0 | y <= a | Transfer 'a' | ARITHMETIC |
| 0 0 0 1 | y <= a + 1 | Increment 'a' | |
| 0 0 1 0 | y <= a - 1 | Decrement 'a' | |
| 0 0 1 1 | y <= b | Transfer 'b' | |
| 0 1 0 0 | y <= b + 1 | Increment 'b' | |
| 0 1 0 1 | y <= b - 1 | Decrement 'b' | |
| 0 1 1 0 | y <= a + b | Add 'a' and 'b' | |
| 0 1 1 1 | y <= a - b | Subtract 'b' from 'a' | |
| 1 0 0 0 | y <= NOT a | Complement 'a' | LOGIC |
| 1 0 0 1 | y <= NOT b | Complement 'b' | |
| 1 0 1 0 | y <= a AND b | AND | |
| 1 0 1 1 | y <= a OR b | OR | |
| 1 1 0 0 | y <= a NAND b | NAND | |
| 1 1 0 1 | y <= a NOR b | NOR | |
| 1 1 1 0 | y <= a XOR b | XOR | |
| 1 1 1 1 | y <= a XNOR b | XNOR | |

## BARREL SHIFTER

- Two types of operation: Arithmetic (mode=0) and Rotation (mode=1)
- Truth table for a 8-bit Barrel Shifter:
  `result[7..0]` (output): It is shifted version of the input `data[7..0]`. `sel[2..0]`: number of bits to shift. `dir`: It controls the shifting direction (`dir=1`: to the right, `dir=0`: to the left). When shifting to the right in the Arithmetic Mode, we use sign extension so as properly account for both unsigned and signed input numbers.

| mode = 0. ARITHMETIC MODE | | | | mode = 1. ROTATION MODE | | | |
|---|---|---|---|---|---|---|---|
| dir | dist[2..0] | data[7..0] | result[7..0] | dir | dist[2..0] | data[7..0] | result[7..0] |
| X | 0 0 0 | abcdefgh | abcdefgh | X | 0 0 0 | abcdefgh | abcdefgh |
| 0 | 0 0 1 | abcdefgh | bcdefgh0 | 0 | 0 0 1 | abcdefgh | bcdefgha |
| 0 | 0 1 0 | abcdefgh | cdefgh00 | 0 | 0 1 0 | abcdefgh | cdefghab |
| 0 | 0 1 1 | abcdefgh | defgh000 | 0 | 0 1 1 | abcdefgh | defghabc |
| 0 | 1 0 0 | abcdefgh | efgh0000 | 0 | 1 0 0 | abcdefgh | efghabcd |
| 0 | 1 0 1 | abcdefgh | fgh00000 | 0 | 1 0 1 | abcdefgh | fghabcde |
| 0 | 1 1 0 | abcdefgh | gh000000 | 0 | 1 1 0 | abcdefgh | ghabcdef |
| 0 | 1 1 1 | abcdefgh | h0000000 | 0 | 1 1 1 | abcdefgh | habcdefg |
| 1 | 0 0 1 | abcdefgh | aabcdefg | 1 | 0 0 1 | abcdefgh | habcdefg |
| 1 | 0 1 0 | abcdefgh | aaabcdef | 1 | 0 1 0 | abcdefgh | ghabcdef |
| 1 | 0 1 1 | abcdefgh | aaaabcde | 1 | 0 1 1 | abcdefgh | fghabcde |
| 1 | 1 0 0 | abcdefgh | aaaaabcd | 1 | 1 0 0 | abcdefgh | efghabcd |
| 1 | 1 0 1 | abcdefgh | aaaaaabc | 1 | 1 0 1 | abcdefgh | defghabc |
| 1 | 1 1 0 | abcdefgh | aaaaaaab | 1 | 1 1 0 | abcdefgh | cdefghab |
| 1 | 1 1 1 | abcdefgh | aaaaaaaa | 1 | 1 1 1 | abcdefgh | bcdefgha |



# PRACTICE EXERCISES

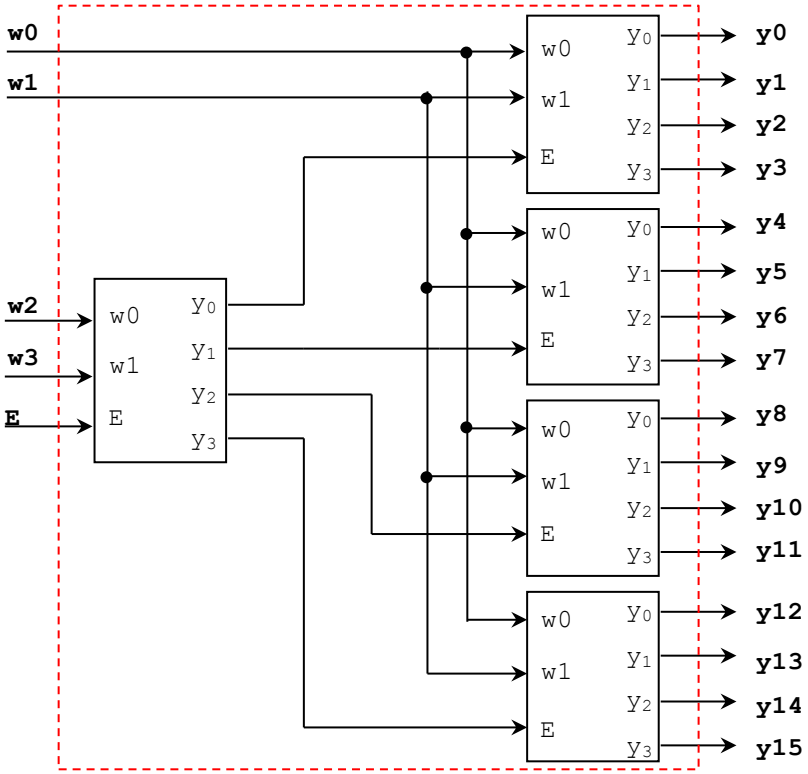
1. Implement the following functions using i) decoders and ii) multiplexers:

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

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: