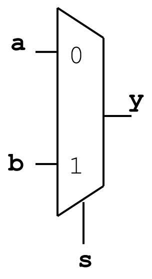
Unit 5 – Combinational Circuits

BASIC CIRCUITS

MULTIPLEXERS (MUXS)

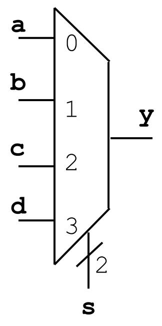
- This logic circuit selects one of many input signals and forwards the selected input to the output line.
- Boolean equations for MUX2-to-1, MUX4-to-1, MUX8-to-1:



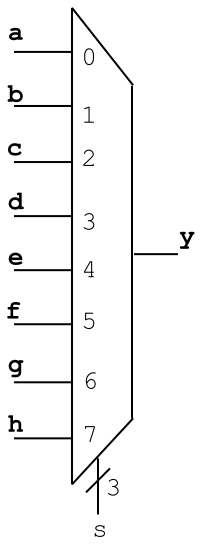
s	a	b	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

s	y
0	a
1	b

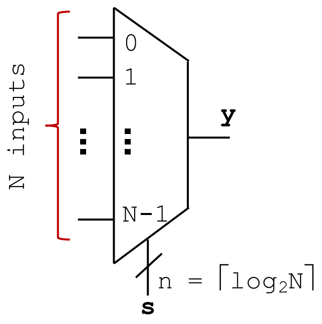
$$y = \overline{s}a + sb$$



$$y = \overline{s_1}\,\overline{s_0}a\ \ \overline{s_1}s_0b\ +$$
$$s_1\overline{s_0}c + s_1s_0d$$



$$y = \overline{s_2}\,\overline{s_1}\,\overline{s_0}a + \overline{s_2}\,\overline{s_1}s_0b + \overline{s_2}s_1\overline{s_0}c + \overline{s_2}s_1s_0d +$$
$$s_2\overline{s_1}\,\overline{s_0}e + s_2\overline{s_1}s_0f + s_2s_1\overline{s_0}g + s_2s_1s_0h$$



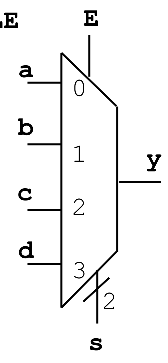
- Normally, a multiplexer has $N = 2^n$ inputs, one output, and a selector with n bits.
- But, if a multiplexer has N inputs, where N is not a power of 2, the number of bits of the selector is given by: $\lceil log_2 N \rceil$.

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:

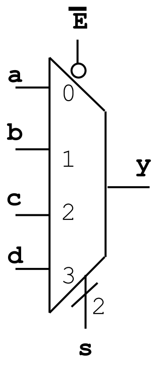
ACTIVE HIGH ENABLE

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



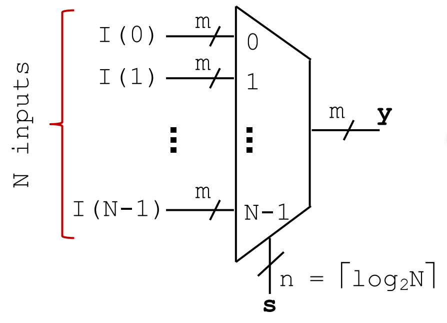
ACTIVE LOW ENABLE

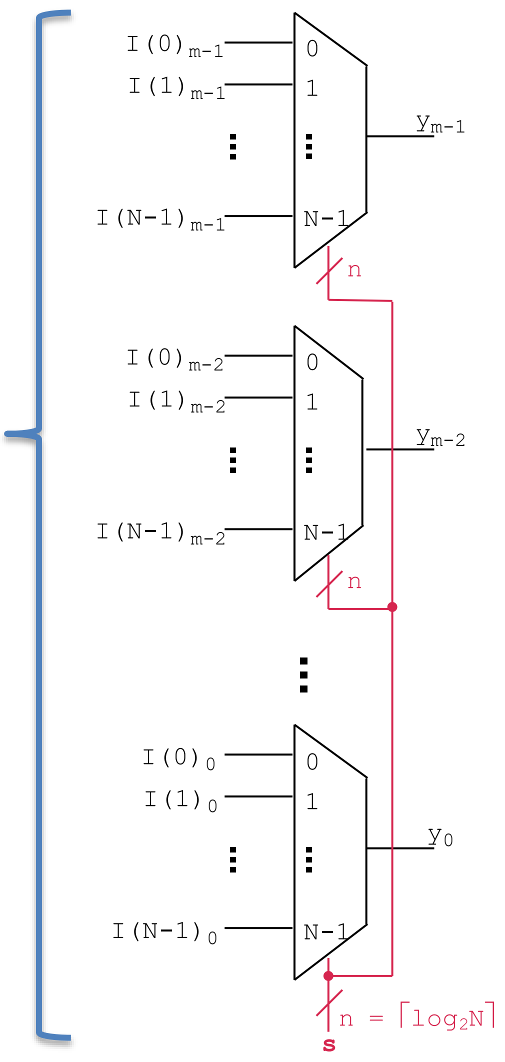
$\overline{E}$	s_1	s_0	y
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	X	X	0



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.

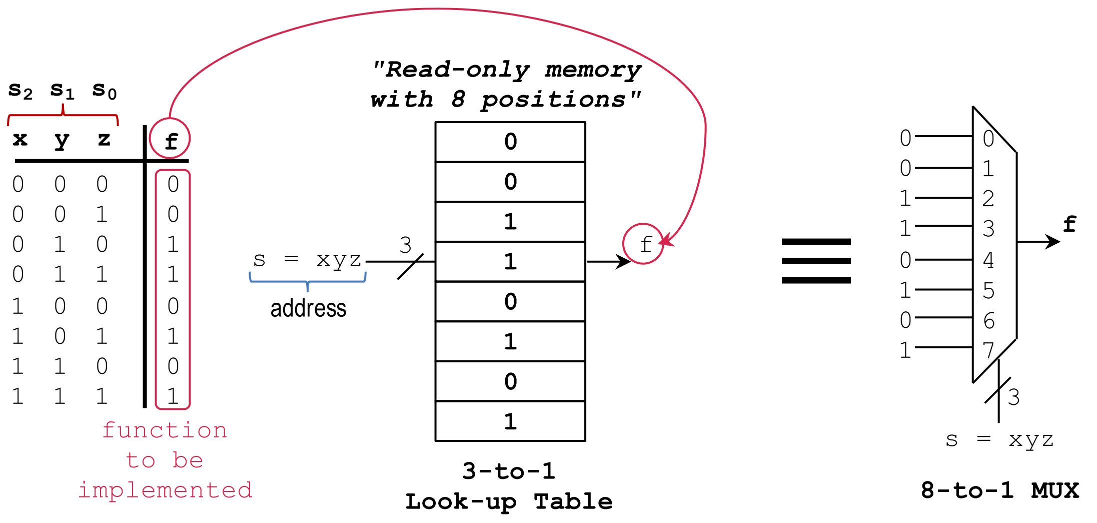




- We have 'N' inputs and therefore the selector has $n = \lceil log_2 N \rceil$ bits.
- Note that the selector is the same for all the multiplexers.

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.

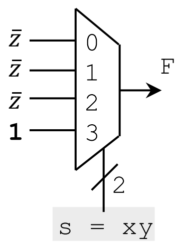


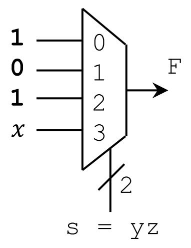
s_2 x	s_1 y	s_0 z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

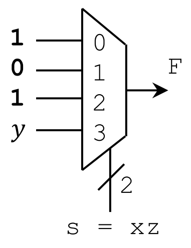
- Note that for a n-variable function, we need a MUX 2^n-to-1 with fixed inputs.

- However, it is possible to use a MUX 2^{n-1}-to-1. This might require extra NOT gates and non-fixed inputs.
  - ✓ $F(x,y,z) = \sum(m_0, m_2, m_4, m_6, m_7)$.

x	y	z	F	
0	0	0	1	$\bar{z}$
0	0	1	0	
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	
1	1	0	1	1
1	1	1	1	

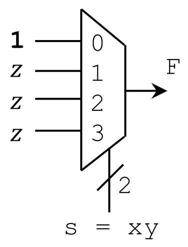


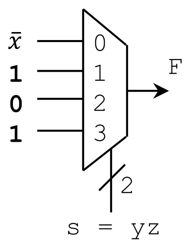


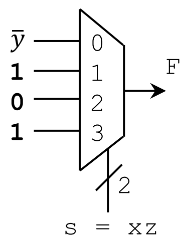


  - ✓ $F(x,y,z) = \sum(m_0, m_1, m_3, m_5, m_7)$.

x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

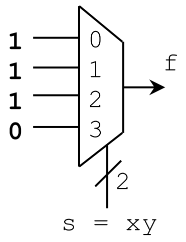


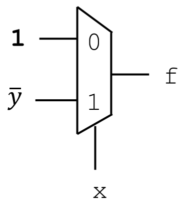


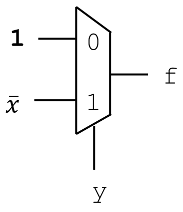


  - ✓ $F(x,y) = \sum(m_0, m_1, m_2)$

x	y	f
0	0	1
0	1	1
1	0	1
1	1	0



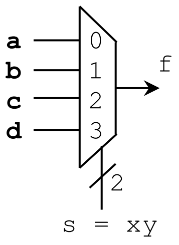


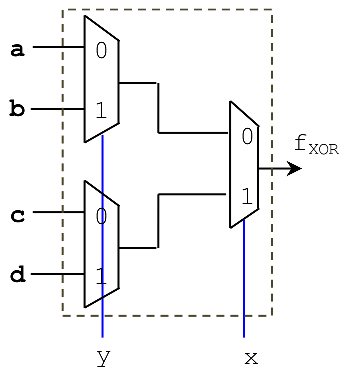


This process of using multiplexors to implement functions can be performed in a systematic fashion using ***Shannon's expansion theorem***. As we will see later for LUTs, this has an important application in the implementation of Boolean functions on FPGAs.

Example:
- Implement a MUX 4-to-1 using MUXes 2to-1.

x	y	f
0	0	a
0	1	b
1	0	c
1	1	d





SHANNON'S EXPANSION

- This is useful to express a Boolean function in terms of multiplexers
- An n-variable Boolean function can be decomposed into two $(n-1)$-variable Boolean functions:

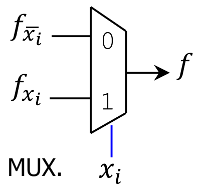
$$f(x_1, x_2, \dots, x_n) = \overline{x_1} f(0, x_2, \dots, x_n) + x_1 f(1, x_2, \dots, x_n)$$

- In the equation, we use the variable x_1 to decompose, but we can use any variable $x_i, i = 1{:}n$. For example, using x_n:

$$f(x_1, x_2, \dots, x_n) = \overline{x_n} f(x_1, x_2, \dots, 0) + x_n f(x_1, x_2, \dots, 1)$$

- A short-hand notation of the Shannon expansion is as follows:

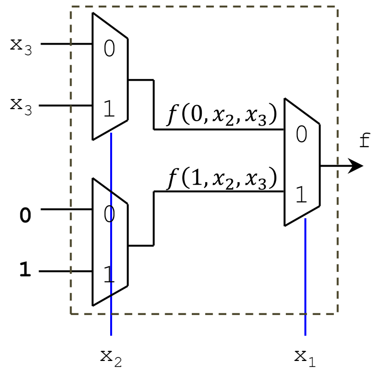
$$f = \overline{x}_i f_{\overline{x}_i} + x_i f_{x_i}$$



Note that we can implement f using a 2-to-1 MUX, as the equation resembles that of the MUX.

- Examples:
  - ✓ $f = x_1x_2 + \overline{x_1}x_3 + x_2x_3$
    $f = \overline{x_1} f(0, x_2, x_3) + x_1 f(1, x_2, x_3) = \overline{x_1}(x_3 + x_2x_3) + x_1(x_2 + x_2x_3)$

    We can further apply Shannon expansion to two variable functions:
    $f = \overline{x_1} g(x_2, x_3) + x_1 h(x_2, x_3)$
    $g(x_2, x_3) = \overline{x_2} g(0, x_3) + x_2 g(1, x_3) = \overline{x_2}(x_3) + x_2(x_3)$
    $h(x_2, x_3) = \overline{x_2} h(0, x_3) + x_2 h(1, x_3) = \overline{x_2}(0) + x_2(1)$



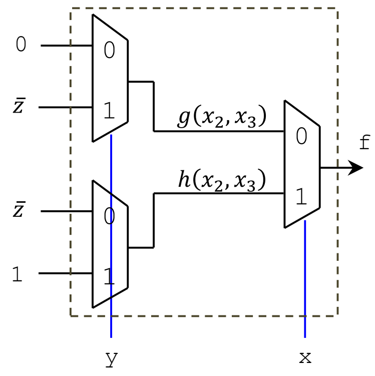
  - ✓ $f = \bar{z}y + \bar{z}x + xyz$
    $f = \bar{x} f(0, y, z) + x f(1, y, z) = \bar{x}(\bar{z}y) + x(\bar{z}y + \bar{z} + yz)$

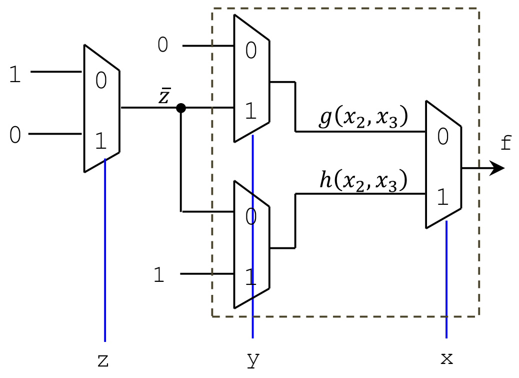
    $f = \bar{x} g(y, z) + x h(y, z)$
    $g(y, z) = \bar{z}y = \bar{y} g(0, z) + y g(1, z) = \bar{y}(0) + y(\bar{z})$
    $h(y, z) = \bar{z}y + \bar{z} + yz = \bar{y} h(0, z) + y h(1, z) = \bar{y}(\bar{z}) + y(1)$

    We can implement $\bar{z}$ using MUXs:

    $p(z) = \bar{z} = \bar{z}p(0) + zp(1) = \bar{z}(1) + z(0)$

    Finally, we implement the function f with only 2-to-1 multiplexors:

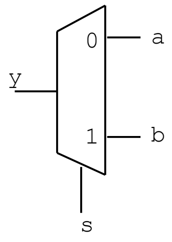




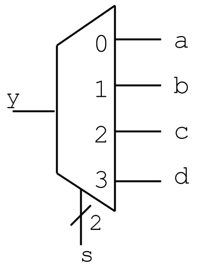
DEMULTIPLEXERS

- A demultiplexer performs the opposite operation of the multiplexers.

s	a	b
0	y	0
1	0	y

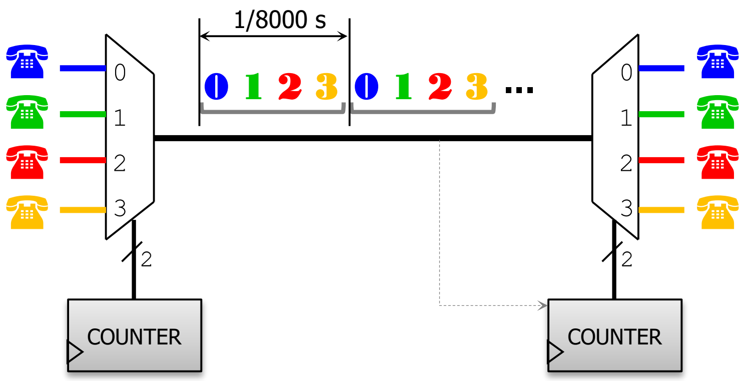


s_1	s_0	a	b	c	d
0	0	y	0	0	0
0	1	0	y	0	0
1	0	0	0	y	0
1	1	0	0	0	y



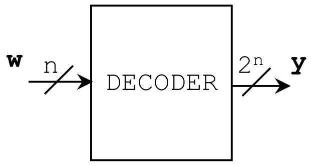
Application: Time Division Multiplexing (TDM)

- 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).

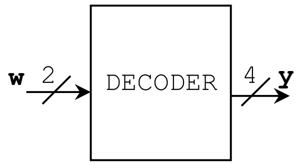


DECODERS

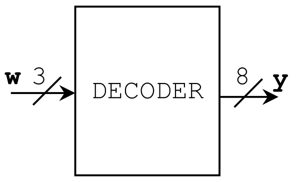
- Generally speaking, decoders are circuits that transform the inputs into outputs following a certain rule, provided that the number of outputs is greater than or equal to the number of inputs.
- Here, we discuss standard decoders for which a specific input/output rule exists. These decoders have n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



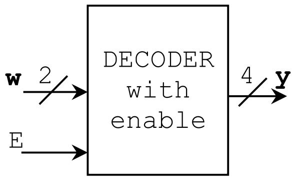
w_1	w_0	y_3	y_2	y_1	y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



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



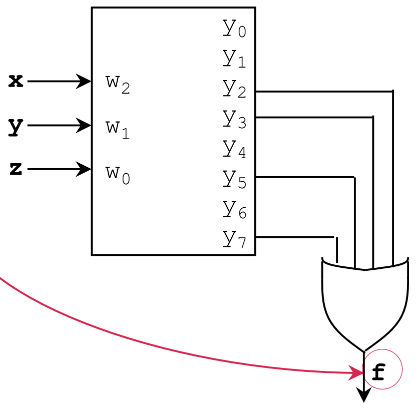
LOGIC CIRCUITS WITH DECODERS

- 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.

$\overbrace{x \quad y \quad z}^{w_2 \quad w_1 \quad w_0}$

x	y	z	f
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

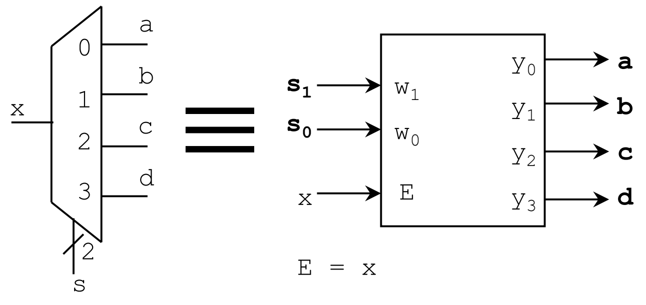
function to be implemented



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:

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

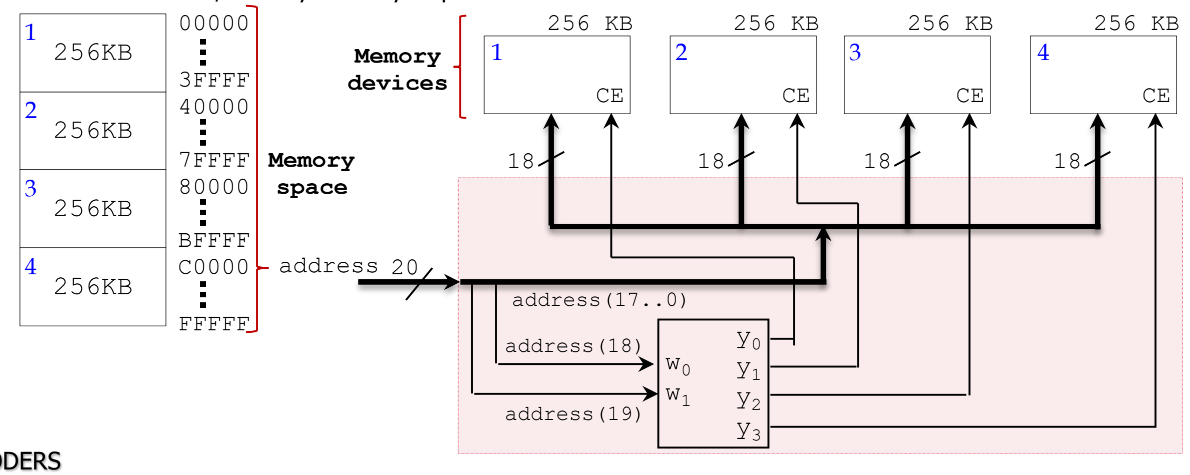


E	w_1	w_0	y_3	y_2	y_1	y_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0

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

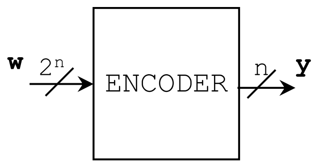
Application: Memory Decoding

- 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.

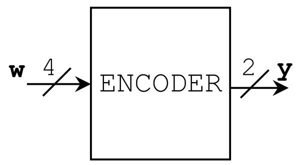


ENCODERS

- 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 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).
- 4 to 2 encoder:



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



$$y_1 = \overline{w_3} w_2 \overline{w_1}.\overline{w_0} + w_3 \overline{w_2}.\overline{w_1}.\overline{w_0}$$
$$y_0 = \overline{w_3}.\overline{w_2} w_1 \overline{w_0} + w_3.\overline{w_2}.\overline{w_1}.\overline{w_0}$$

Assumptions:
If $w_3 = 1 \rightarrow w_2 = w_1 = w_0 = 0$
If $w_2 = 1 \rightarrow w_3 = w_1 = w_0 = 0$
If $w_1 = 1 \rightarrow w_3 = w_2 = w_0 = 0$
If $w_0 = 1 \rightarrow w_3 = w_2 = w_1 = 0$

Thus:

$$y_1 = w_2 + w_3$$
$$y_0 = w_1 + w_3$$

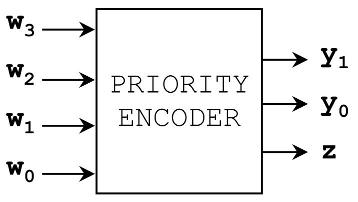
- 8 to 3 encoder:

$$y_2 = w_7 + w_6 + w_5 + w_4$$
$$y_1 = w_7 + w_6 + w_4 + w_3$$
$$y_0 = w_7 + w_5 + w_3 + w_1$$

- Issues:
  - ✓ If two or more inputs are activated, the output $y_{n-1}y_{n-2} \dots y_0$ is undefined.
  - ✓ If no input is activated, the output $y_{n-1}y_{n-2} \dots y_0$ is undefined. In this case, the result is ambiguous, as the result would be the same as if only $w_0 = 1$, i.e., $y_{n-1}y_{n-2} \dots y_0 = 00 \dots 0$.

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? A solution is to create 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.

- For the priority encoder 4 to 2, we can get the Boolean functions directly from the truth table as:

$$y_1 = w_2\overline{w_3} + w_3$$
$$y_0 = \overline{w_3}\,\overline{w_2}w_1 + w_3$$
$$z = \overline{\overline{w_3}\,\overline{w_2}\,\overline{w_1}\,\overline{w_0}} = w_3 + w_2 + w_1 + w_0$$

We could simplify y_1 and y_0 (Boolean Theorems, K-maps, or Quine-McCluskey algorithm)

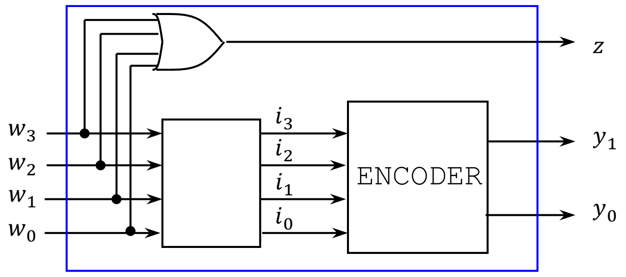
- Alternatively, we can create the following intermediate signals:

$i_3 = w_3$	$i_3 = 1$ if $w_3 = 1$
$i_2 = \overline{w_3}w_2$	$i_2 = 1$ if $w_2 = 1, w_3 = 0$
$i_1 = \overline{w_3}\,\overline{w_2}w_1$	$i_3 = 1$ if $w_1 = 1, w_2 = w_3 = 0$
$i_0 = \overline{w_3}\,\overline{w_2}\,\overline{w_1}w_0$	$i_3 = 1$ if $w_0 = 1, w_1 = w_2 = w_3 = 0$

Then, notice that $i_3 = 1, i_2 = 1, i_1 = 1, i_0 = 1$ are exclusive:

If	i_3	i_2	i_1	i_0	y_1	y_0	z
$w_3 = 1$	1	0	0	0	1	1	1
$w_2 = 1, w_3 = 0$	0	1	0	0	1	0	1
$w_1 = 1, w_2 = w_3 = 0$	0	0	1	0	0	1	1
$w_0 = 1, w_1 = w_2 = w_3 = 0$	0	0	0	1	0	0	1
$w_0 = w_1 = w_2 = w_3 = 0$	0	0	0	0	0	0	0

Note that the relationship of $i_3i_2i_1i_0$ to y_1y_0 is that of a binary encoder. If $z = 0$, then $y_1y_0 = 00$.



PRIORITY ENCODER

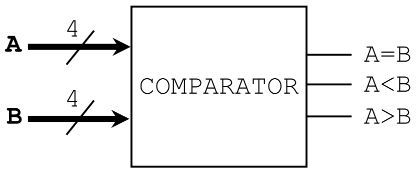
This procedure can be applied to any priority encoder (e.g.: 8 to 3, 16 to 4).

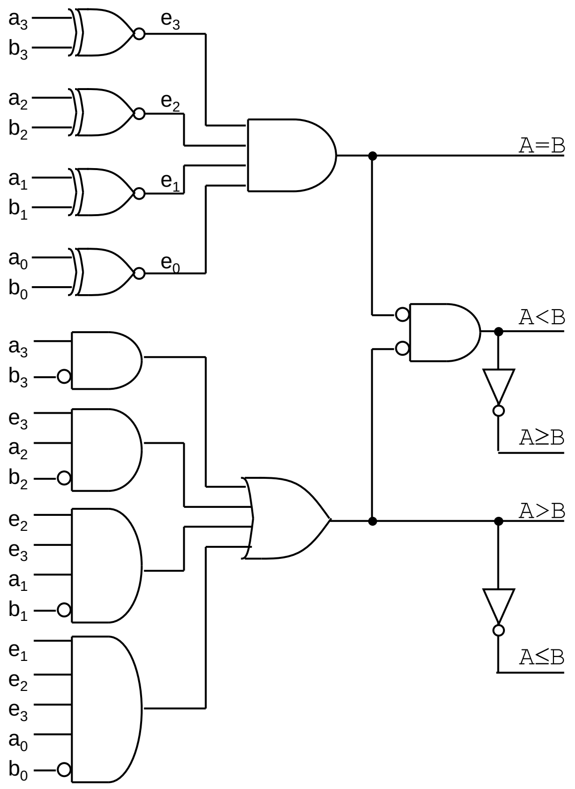
COMPARATORS

UNSIGNED NUMBERS

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

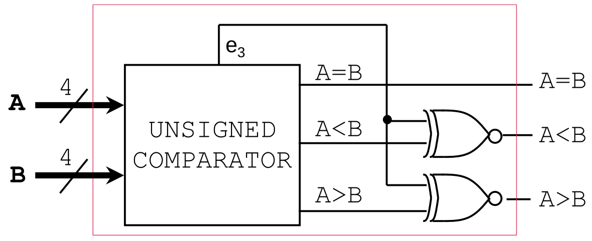
  ✓ $A > B$ when:
  $a_3 = 1, b_3 = 0$
  Or: $a_3 = b_3$ and $a_2 = 1, b_2 = 0$
  Or: $a_3 = b_3, a_2 = b_2$ and $a_1 = 1, b_1 = 0$
  Or: $a_3 = b_3, a_2 = b_2, a_1 = b_1$ and $a_0 = 1, b_0 = 0$





SIGNED NUMBERS

- If $A \geq 0$ and $B \geq 0$, we can use the unsigned comparator.
- If $A < 0$ and $B < 0$, we can also use the unsigned comparator.
  Example: $1000_2 < 1001_2$ (-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: $1000_2 < 0100_2$ (-8 < 4). If we were to use the unsigned comparator, we would get $1000_2 > 0100_2$. So, in this case, we need to invert both the A>B and the A<B bit.

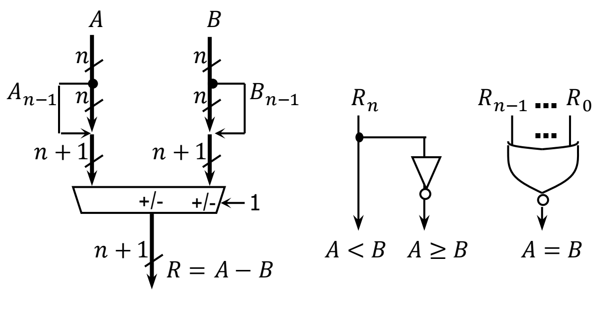


- Rule: For a 4-bit number in 2's complement:
  ✓ If $a_3 = b_3$, A and B have the same sign. Then, we do not need to invert any bit.
  ✓ If $a_3 \neq b_3$, A and B have a different sign. Then, we need to invert the A>B and A<B bits of the unsigned comparator.

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

ALTERNATIVE APPROACH

- Here, we perform A-B in 2C. If the result is positive (MSB=0), then A ≥ B. If the result is negative (MSB=1), then A < B. We use an 2C adder/subtractor unit to implement this operation (R=A-B):
  ✓ Signed numbers: we need to sign-extend the inputs to consider the worst-case scenario.
  ✓ Unsigned numbers: we need to zero-extend the inputs to convert the values to 2C arithmetic.
- To determine whether A is greater than B, we use the MSB (R_n):
$$R_n = \begin{cases} 1 \rightarrow A - B < 0 \\ 0 \rightarrow A - B \geq 0 \end{cases}$$
- To determine whether $A = B$, we compare the $n+1$ bits of R to 0 ($R = A - B$). However, note that $(A - B) \in [-2^n + 1, 2^n - 2]$. So, the case $R = -2^n = 10\ldots0$ will not occur. Thus, we only need to compare the bits R_{n-1} to R_0 to 0.

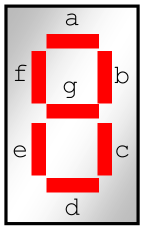


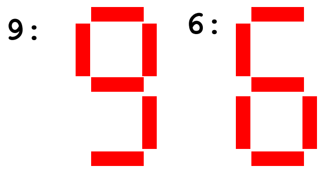
CODE CONVERTERS

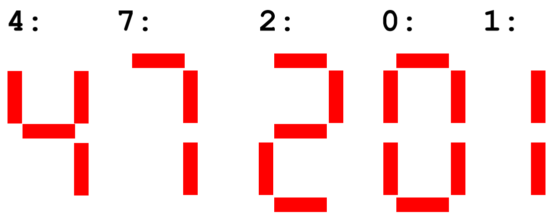
BCD TO 7-SEGMENT DECODER

- The BCD system is useful as it provides a convenient human-readable format. For example, a keypad usually produces 4-bit BCD codes every time a user presses a key. A big challenge is to convert a series of 4-bit BCD codes into its binary representation. For example: 0101 1001 0111 = 597 in BCD, but 597 is 1001010101 in binary (unsigned).
- The BCD to 7-segment converter 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







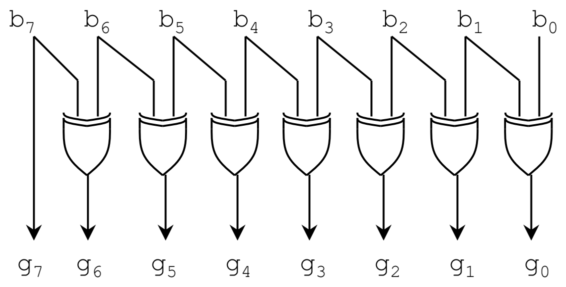
GRAY TO BCD DECODER

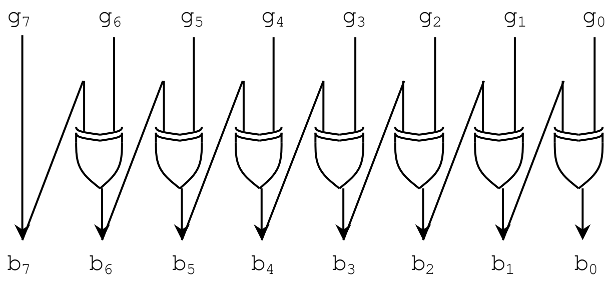
- It is a decoder because the number of outputs is equal to the number of inputs.
- The figure shows the truth table for a 4-bit case.

g_3	g_2	g_1	g_0	b_3	b_2	b_1	b_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 AND GRAY TO BINARY DECODER

- These are decoders because the number of outputs is equal to the number of inputs
- For small input sizes, we can use the truth table method (see *Lecture Notes – Unit 4*). For large input sizes, the following circuits are 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 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.

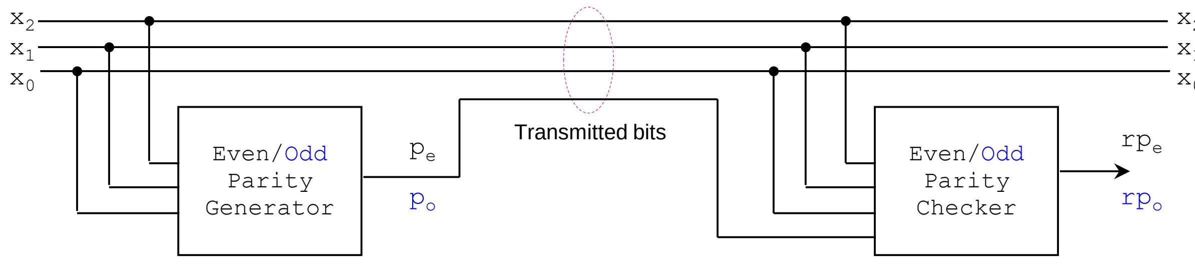
Example:

- 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)

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

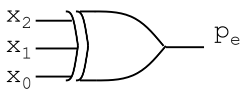
  - ✓ Odd Parity Generator: It generates the parity bit p_o.
  - ✓ Odd Parity Checker: It verifies that the received stream Y has odd parity. If so, rp_o=0, otherwise rp_o=1 (to signal an error)

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



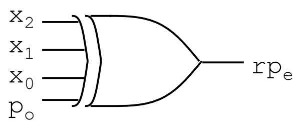
Even Parity Generator

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



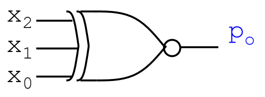
Even Parity Checker

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



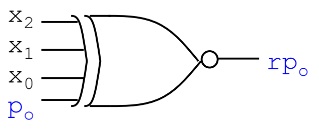
Odd Parity Generator

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

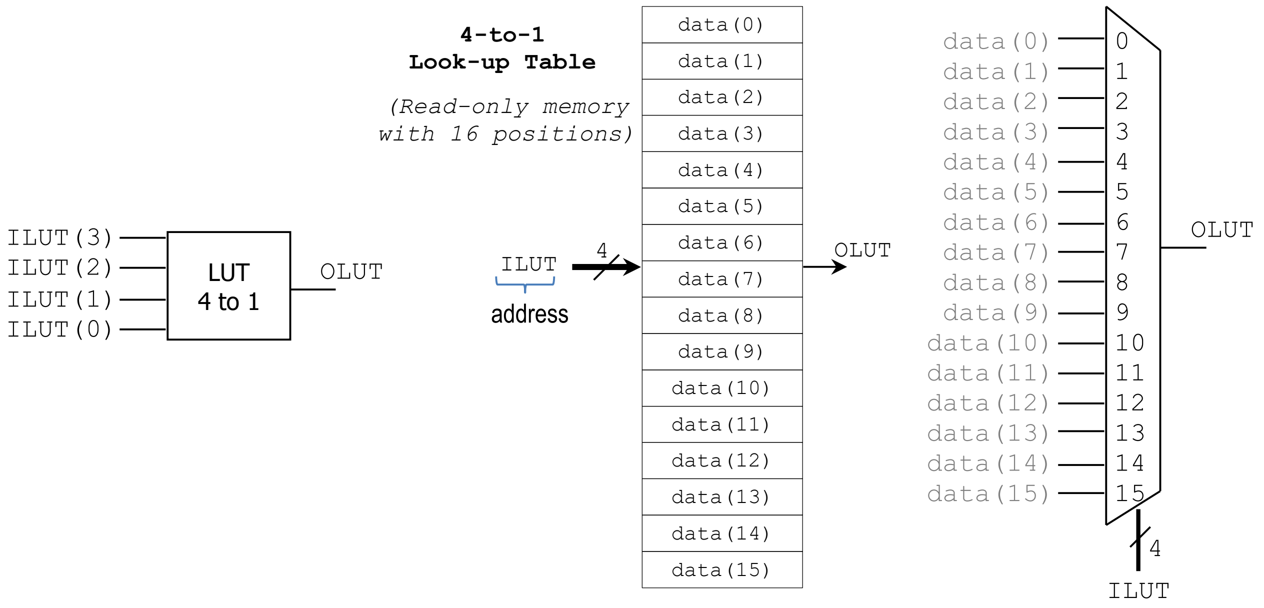


- In general for $X = x_{n-1}x_{n-2} \dots x_1x_0$: $p_e = x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0$. $p_o = \overline{x_{n-1} \oplus x_{n-2} \dots x_1 \oplus x_0}$
  - ✓ If the # of 1's in an n-bit stream is odd, the n-bit input XOR gate will return 1, 0 otherwise.
  - ✓ If the # of 1's in an n-bit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

COMPLEX CIRCUITS

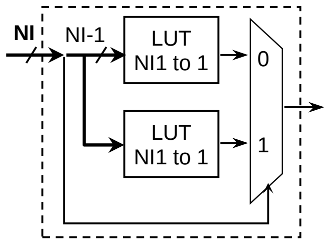
LOOK-UP TABLES (LUTS)

- 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.

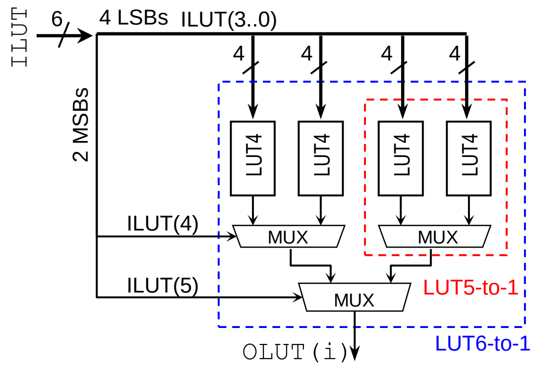

ILUT(3)
ILUT(2)
ILUT(1)
ILUT(0)
LUT 4 to 1
OLUT
4-to-1 Look-up Table
(Read-only memory with 16 positions)
ILUT
address
4
data(0) data(1) data(2) data(3) data(4) data(5) data(6) data(7) data(8) data(9) data(10) data(11) data(12) data(13) data(14) data(15)
OLUT
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
OLUT
4
ILUT

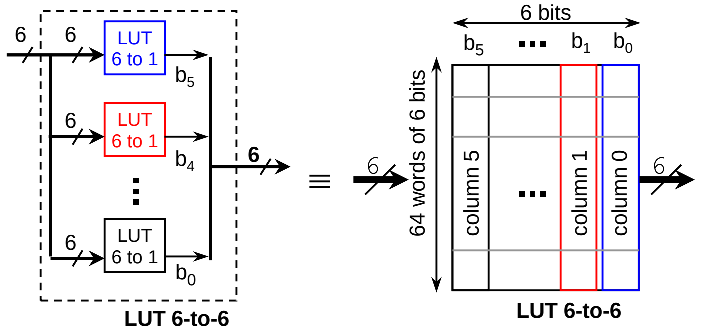
LARGER LUTS

- A larger LUT can be generated 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 on the right. We can build a NI-to-1 LUT with this method.
- The figure below shows a case for a LUT 6-to-1 built out of two LUT 5-to-1. Each LUT 5-to-1 is built 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.



LUT NI-to-1

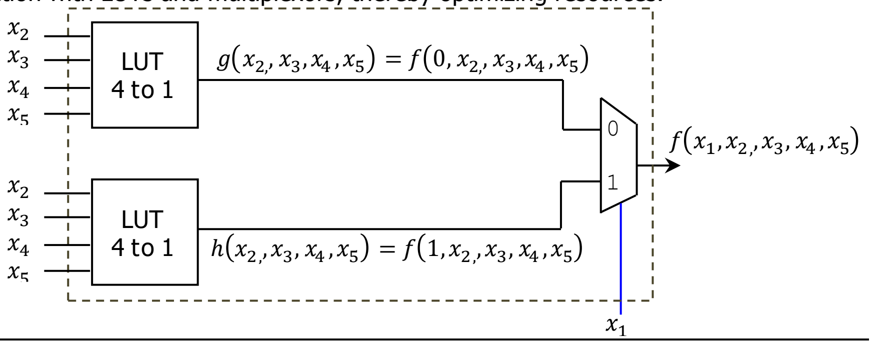




LUT 6-to-6 **LUT 6-to-6**

LUT DECOMPOSITION USING SHANNON EXPANSION

- LUT size grows exponentially with the size of the input. For a n-variable Boolean function, Shannon expansion provides a systematic way of implementing that function with LUTs and multiplexors, thereby optimizing resources.
- An n-variable Boolean function can be decomposed into two $(n-1)$-variable Boolean functions and a MUX using Shannon expansion.
- **Example**: 5-variable function. Instead of using a 5-to-1 LUT, we decompose the function using Shannon expansion. Then we can implement it using a MUX 2-to-1 and two 4-to-1 LUTs.


x_2 x_3 x_4 x_5
LUT 4 to 1
$g(x_2, x_3, x_4, x_5) = f(0, x_2, x_3, x_4, x_5)$
LUT 4 to 1
$h(x_2, x_3, x_4, x_5) = f(1, x_2, x_3, x_4, x_5)$
0
1
$f(x_1, x_2, x_3, x_4, x_5)$
x_1

Example:

- Using 3-to-1 LUTs and 2-to-1 MUXes, implement the following Boolean function (specify the contents of the LUTs):

$$f(x_1,x_2,x_3,x_4,x_5) = x_1x_2\overline{x_4} + x_3\oplus(x_4+x_5) + \overline{x_1}\,\overline{x_2}x_5$$

$$f = \overline{x_1}f(0,x_2,x_3,x_4,x_5) + x_1f(1,x_2,x_3,x_4,x_5) = \overline{x_1}(x_3\oplus(x_4+x_5) + x_2\overline{x_5}) + x_1\big(x_2\overline{x_4} + x_3\oplus(x_4+x_5)\big)$$

✓ $g(x_2,x_3,x_4,x_5) = f(0,x_2,x_3,x_4,x_5) = x_3\oplus(x_4+x_5) + \overline{x_2}x_5$

$g = \overline{x_2}g(0,x_3,x_4,x_5) + x_2g(1,x_3,x_4,x_5) = \overline{x_2}(x_3\oplus(x_4+x_5) + x_5) + x_2\big(x_3\oplus(x_4+x_5)\big)$

Note that: $g(0,x_3,x_4,x_5) = f(0,0,x_3,x_4,x_5)$, $g(1,x_3,x_4,x_5) = f(0,1,x_3,x_4,x_5)$

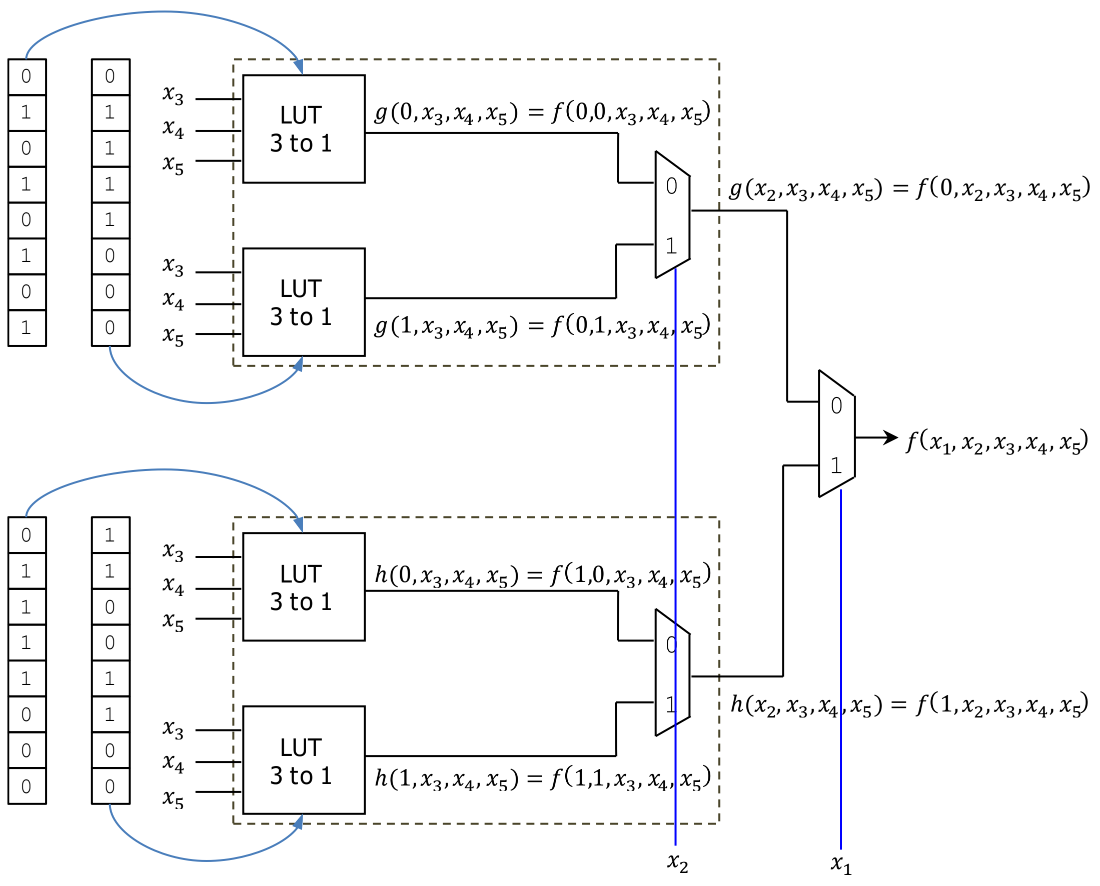
✓ $h(x_2,x_3,x_4,x_5) = f(1,x_2,x_3,x_4,x_5) = x_2\overline{x_4} + x_3\oplus(x_4+x_5)$

$h = \overline{x_2}h(0,x_3,x_4,x_5) + x_2h(1,x_3,x_4,x_5) = \overline{x_2}\big(x_3\oplus(x_4+x_5)\big) + x_2\big(\overline{x_4} + x_3\oplus(x_4+x_5)\big)$

Note that: $h(0,x_3,x_4,x_5) = f(1,0,x_3,x_4,x_5)$, $h(1,x_3,x_4,x_5) = f(1,1,x_3,x_4,x_5)$

These four 3-variable functions will be implemented using 3-to-1 LUTs. We are ready to sketch the circuit using 3-to-1 LUTs and 2-to-1 MUXes. This is how multi-variable functions are implemented on FPGAs.

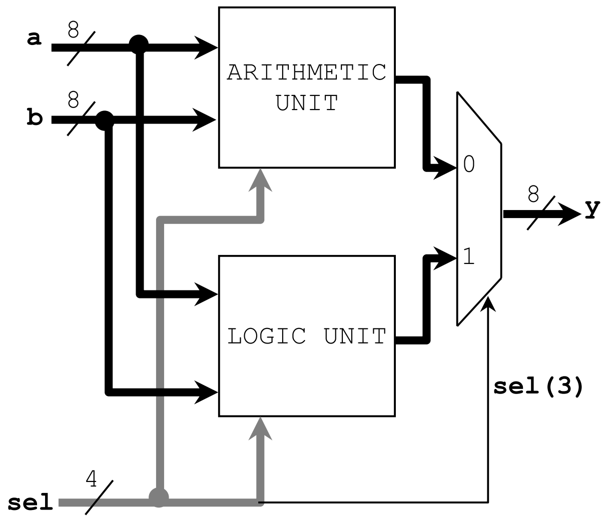
In order to get the LUT contents, we can either evaluate every 3-variable function that was generated, or we can just fill up the truth table for f and identify the LUT contents for each 3-variable function.


0 1 0 1 0 1 0 1
0 1 1 1 1 0 0 0
0 1 1 1 1 0 0 0
1 1 0 0 1 1 0 0
LUT 3 to 1
x_3 x_4 x_5
$g(0,x_3,x_4,x_5) = f(0,0,x_3,x_4,x_5)$
$g(1,x_3,x_4,x_5) = f(0,1,x_3,x_4,x_5)$
$h(0,x_3,x_4,x_5) = f(1,0,x_3,x_4,x_5)$
$h(1,x_3,x_4,x_5) = f(1,1,x_3,x_4,x_5)$
0
1
$g(x_2,x_3,x_4,x_5) = f(0,x_2,x_3,x_4,x_5)$
$h(x_2,x_3,x_4,x_5) = f(1,x_2,x_3,x_4,x_5)$
$f(x_1,x_2,x_3,x_4,x_5)$
x_2
x_1

x_1	x_2	x_3	x_4	x_5	f	
0	0	0	0	0	0	$f(0,0,x_3,x_4,x_5)$
0	0	0	0	1	1	
0	0	0	1	0	0	
0	0	0	1	1	1	
0	0	1	0	0	0	
0	0	1	0	1	1	
0	0	1	1	0	0	
0	0	1	1	1	1	
0	1	0	0	0	0	
0	1	0	0	1	1	
0	1	0	1	0	1	
0	1	0	1	1	1	
0	1	1	0	0	1	
0	1	1	0	1	0	
0	1	1	1	0	0	
0	1	1	1	1	0	
1	0	0	0	0	0	$f(1,0,x_3,x_4,x_5)$
1	0	0	0	1	1	
1	0	0	1	0	1	
1	0	0	1	1	1	
1	0	1	0	0	1	
1	0	1	0	1	0	
1	0	1	1	0	0	
1	0	1	1	1	0	
1	1	0	0	0	1	
1	1	0	0	1	1	
1	1	0	1	0	0	
1	1	0	1	1	0	
1	1	1	0	0	1	
1	1	1	0	1	1	
1	1	1	1	0	0	
1	1	1	1	1	0	

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.

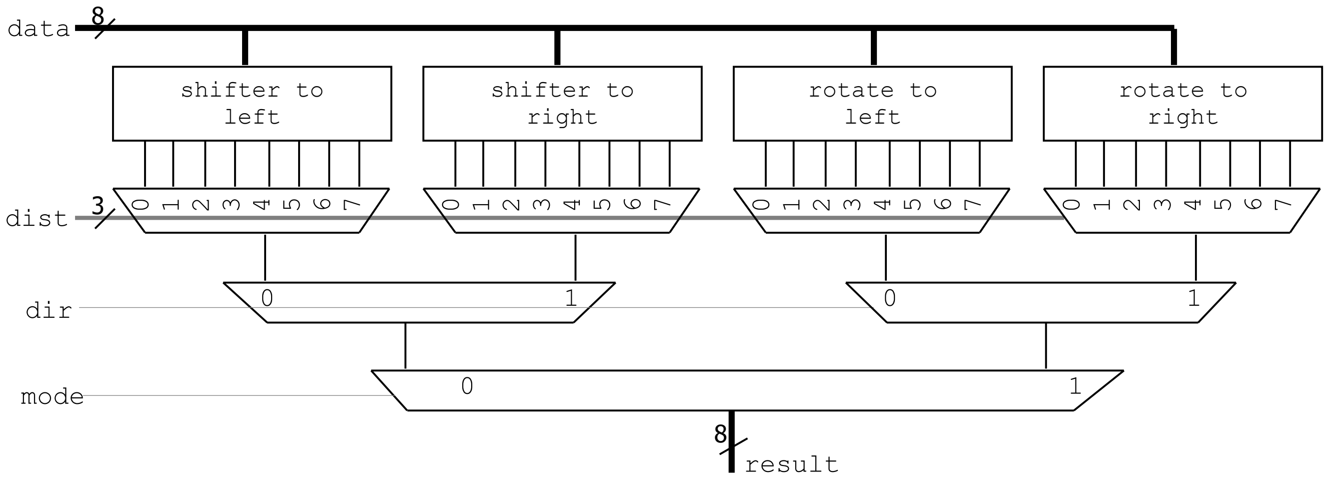


sel	Operation	Function	Unit
0 0 0 0	y <= a	Transfer 'a'	ARITHMETIC
0 0 0 1	y <= a + 1	Increment 'a'	
0 0 1 0	y <= a - 1	Decrement 'a'	
0 0 1 1	y <= b	Transfer 'b'	
0 1 0 0	y <= b + 1	Increment 'b'	
0 1 0 1	y <= b - 1	Decrement 'b'	
0 1 1 0	y <= a + b	Add 'a' and 'b'	
0 1 1 1	y <= a - b	Subtract 'b' from '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, $\times 2^i$) and Rotation (mode=1)
- Truth table for an 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 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

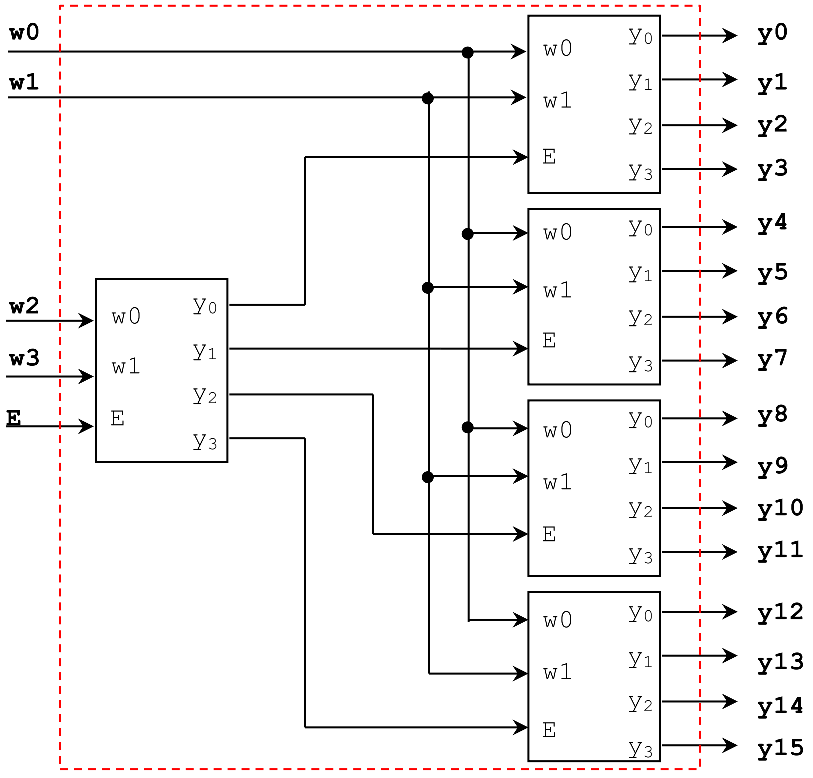
1. Implement the following functions using i) decoders and ii) multiplexers:

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

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.

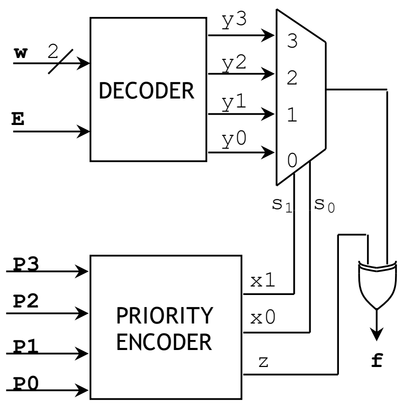
4. Verify that the following circuit made of out of five 2-to-4 decoders with enable represents a 4-to-16 decoder with enable. Tip: Create the truth table.

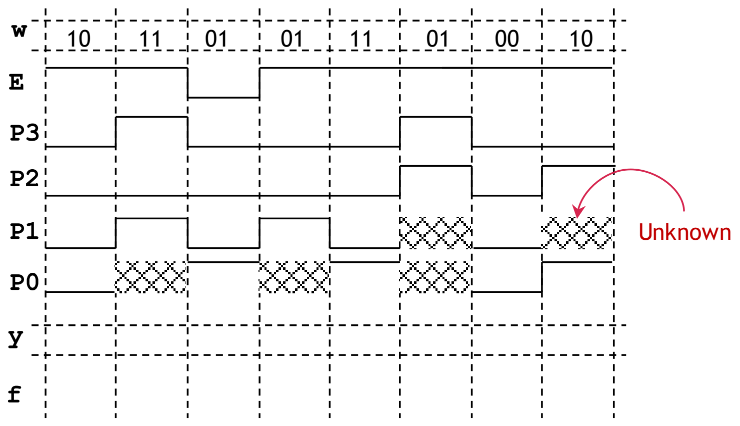


5. Using only 2-to-1 MUXs, implement the XOR and XNOR gates.

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:


w
2
E
DECODER
y3
y2
y1
y0
3
2
1
0
s1
s0
P3
P2
P1
P0
PRIORITY ENCODER
x1
x0
z
f


w
E
P3
P2
P1
P0
y
f
Unknown