COMP3222/9222 Digital Circuits & Systems

4. Combinational Building Blocks

Objectives

- Learn about commonly used combinational sub-circuits

  – Multiplexers, used for signal selection and implementing general logic functions

  – Encoders, decoders and code converters

- Learn about the key VHDL constructs used to specify combinational circuits

  – Non-simple, concurrent assignment statements

  – Sequential statements

Multiplexers

A mutiplexer (MUX) has a number of data inputs, one or more select inputs, and one output that

- Passes the signal value on one of the data inputs to the output
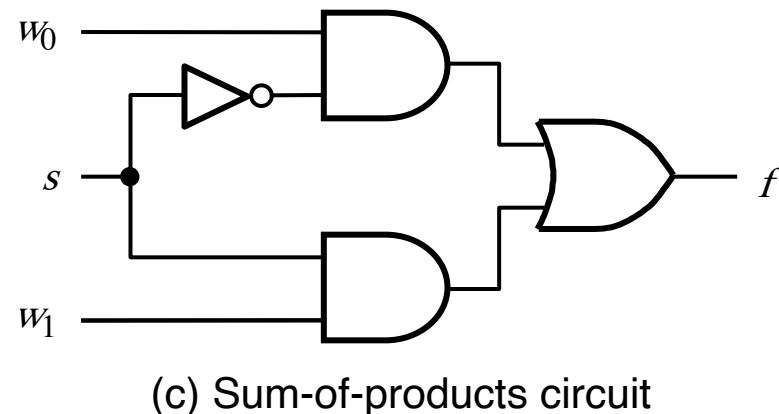- The data input is selected by the values of the select inputs

- A 2-to-1 MUX, which has 2 data inputs and therefore 1 select input, is shown below
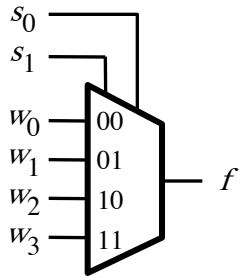 - This 2-to-1 MUX implements the function $f = \overline{s}\cdot w_0 + s\cdot w_1$
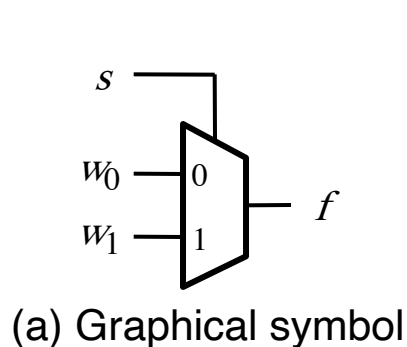
(a) Graphical symbol

s	w_0	w_1	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

s	f
0	w_0
1	w_1

(b) Truth table

(c) Sum-of-products circuit

A 4-to-1 multiplexer

(a) Graphic symbol

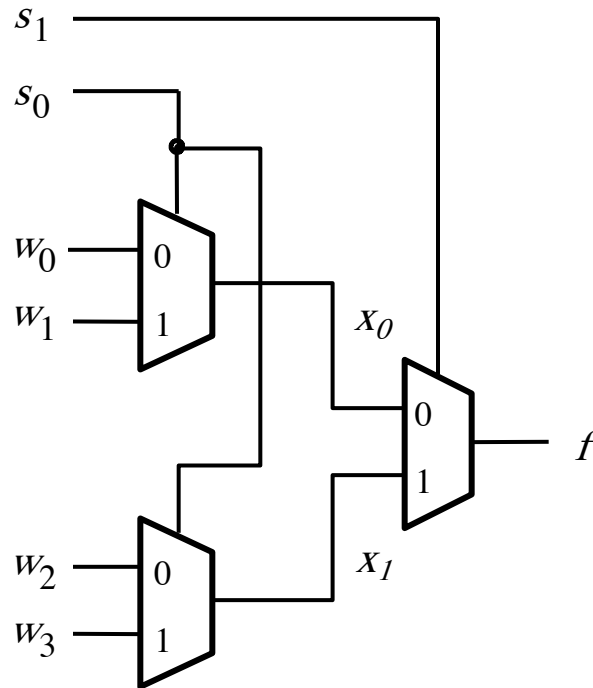| s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table
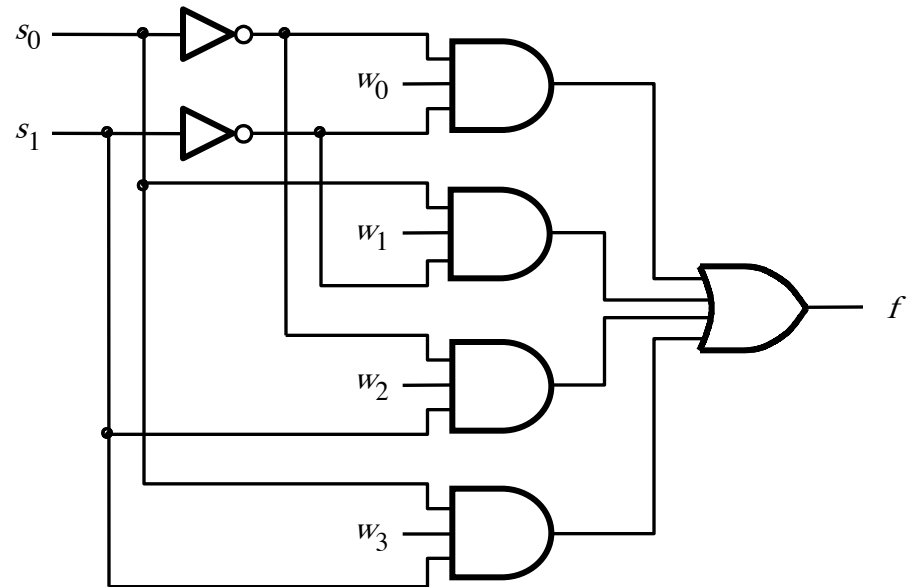
(c) Circuit

- A 4-to-1 MUX, which has 4 data and 2 select inputs, realizes the function

$$f = \overline{s_1}\,\overline{s_0}\,w_0 + \overline{s_1}s_0 w_1 + s_1\overline{s_0}w_2 + s_1 s_0 w_3$$

- Larger MUXes can be built using the same approach, but can also be built from smaller MUXes

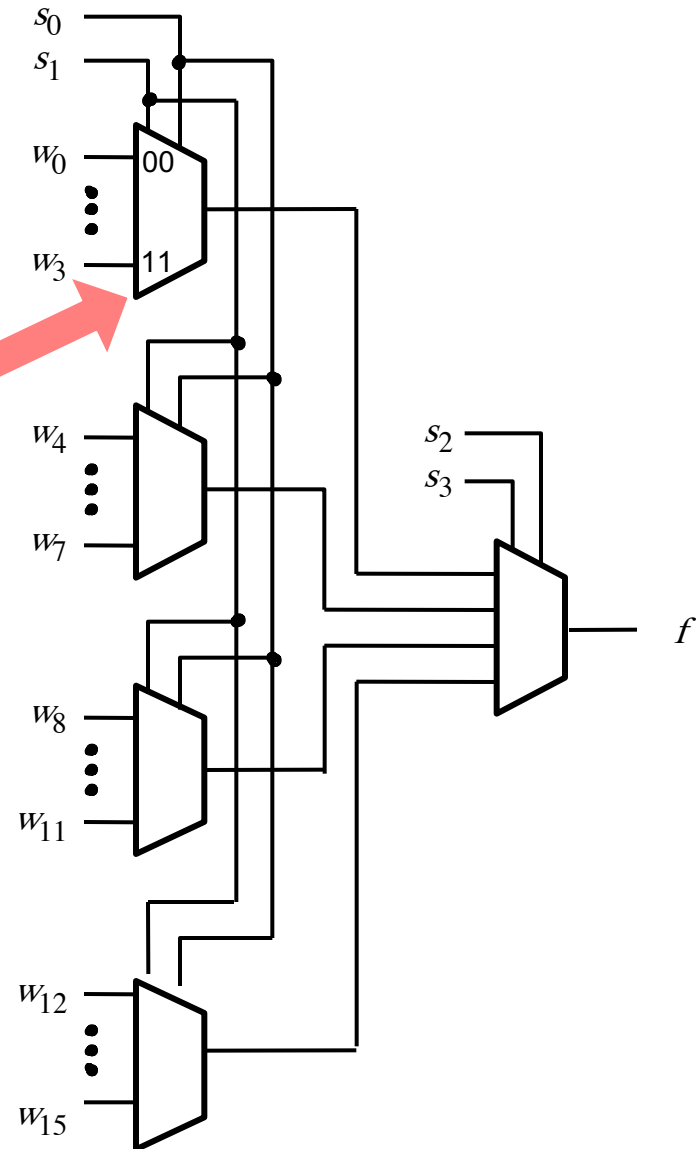# Using 2-to-1 MUXes to build a 4-to-1 MUX

$$f = \overline{s_1}x_0 + s_1 x_1$$
$$= \overline{s_1}(\overline{s_0}w_0 + s_0 w_1) + s_1(\overline{s_0}w_2 + s_0 w_3)$$
$$= \overline{s_1}\,\overline{s_0}w_0 + \overline{s_1}s_0 w_1 + s_1\overline{s_0}w_2 + s_1 s_0 w_3$$

A 16-to-1 multiplexer built from 4-to-1 MUXes

Note: From now on let us assume that the inputs selected by the MUX are numbered from 0..0 at the top to 1..1 at the bottom of the MUX symbol block

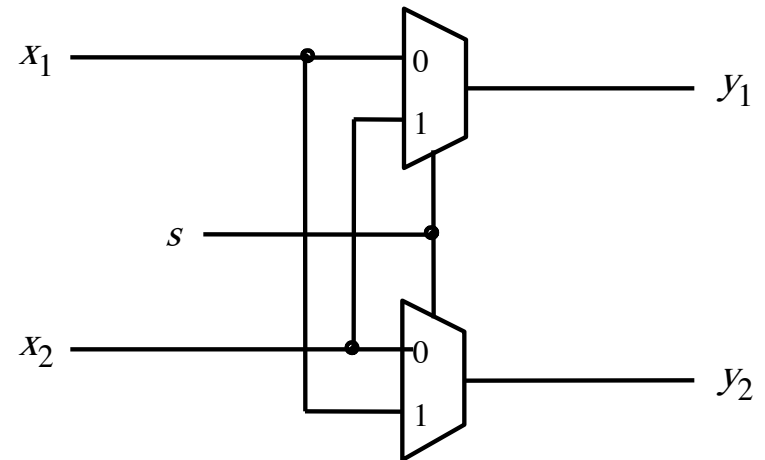Combinational Building Blocks

Practical applications of multiplexers

- A circuit that has n inputs and k outputs, whose function it is to provide a capability to connect any input to any output, is referred to as an $n \times k$ crossbar switch

- A 2×2 crossbar switch can easily be built using two 2-to-1 multiplexers
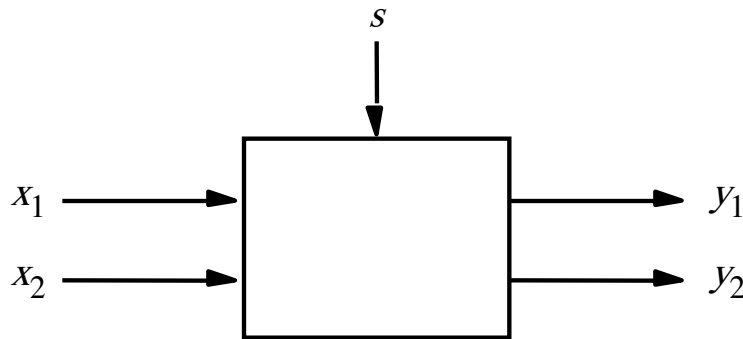
(a) A 2x2 crossbar switch

(b) Implementation using multiplexers

Implementing programmable switches in an FPGA

Programmable input switches

lookup table

i_1
i_2
f
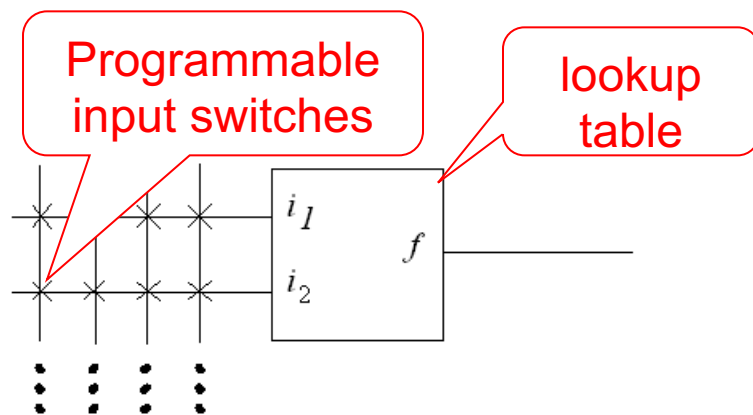
(a) Part of the FPGA in Figure 3.39

0/1 0/1 0/1 0/1

Storage cell

0/1 0/1 0/1 0/1

i_1
i_2
f
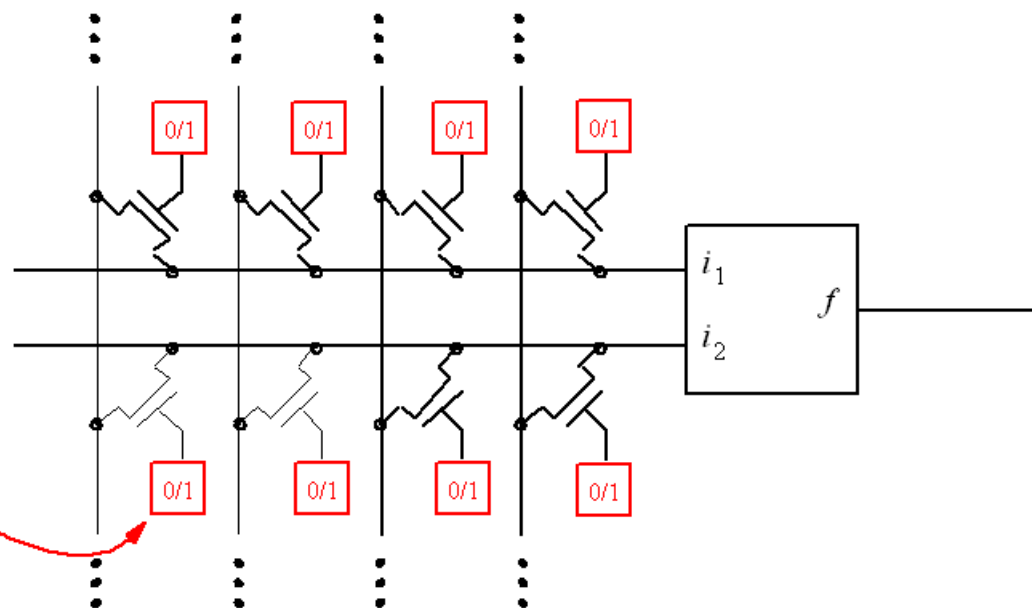
(b) Implementation using pass transistors

0/1 0/1

i_1
i_2
f

0/1 0/1
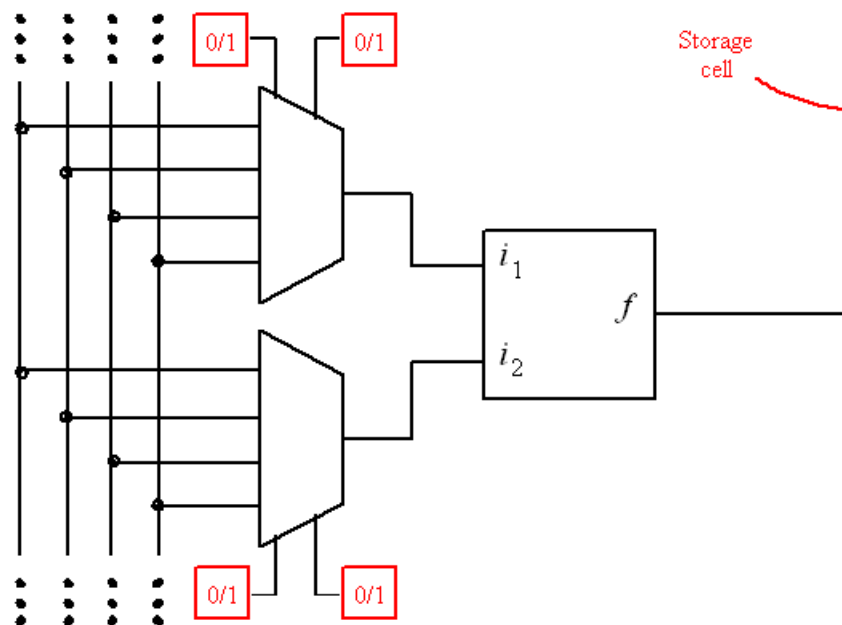
(c) Implementation using multiplexers

Use of MUXes reduces the 6-transistor cost per storage cell and reduces the risk of creating unintended connections

Synthesis of logic functions using multiplexers

- We can also use MUXes to implement functions

- For example, consider the XOR function below
  - Each row of the truth table can be connected to a MUX input as a constant
  - The select inputs are driven by the variables of the function

- But, we can be more efficient if we rewrite the truth table:

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\overline{w}_2$

(b) Modified truth table

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0
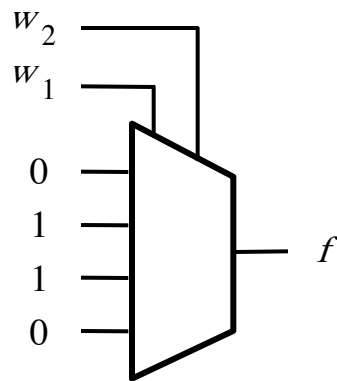
(a) Implementation using a 4-to-1 multiplexer
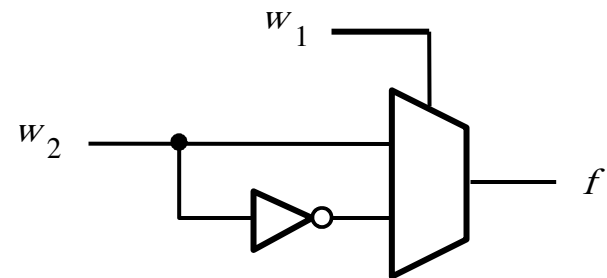
(c) Circuit

3-input majority fn using a 4-to-1 MUX

- In a similar manner, we can efficiently implement other functions

w_1	w_2	w_3	f
0	0	0	0
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	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

(a) Modified truth table

(b) Circuit

w_2

w_1

0

w_3

1

f
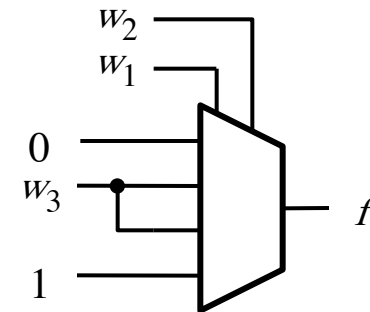
- – Note that any variable pair could be used as the selector inputs without changing the circuit structure

3-input XOR implemented using 2-to-1 MUXes

w_1	w_2	w_3	f
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

$w_2 \oplus w_3$
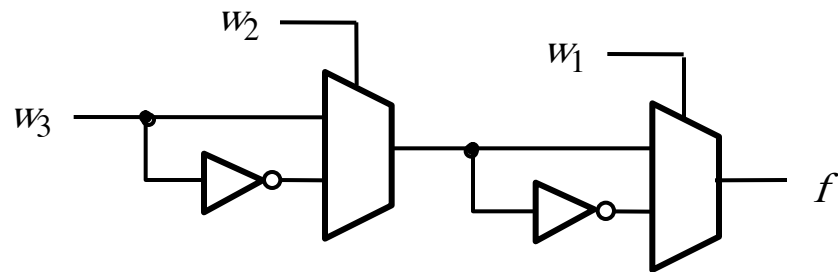
$\overline{w_2 \oplus w_3}$

(a) Truth table

(b) Circuit

3-input XOR implemented with a 4-to-1 MUX
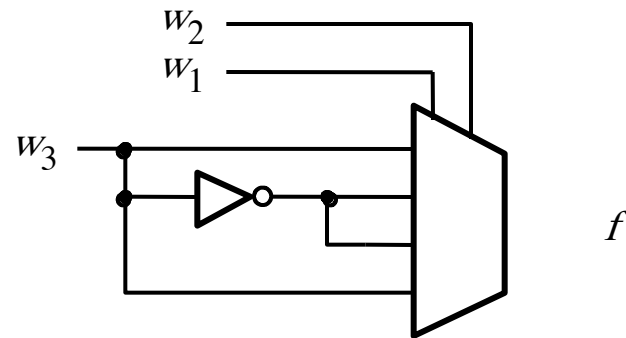
(a) Truth table

(b) Circuit

Multiplexer synthesis using Shannon's expansion

- So far, we have seen how truth tables can be interpreted to implement logic functions using MUXes

  - In each case, the MUX inputs are the constant 0 & 1, or some variable, or its complement

- Besides using simple inputs, it is possible to connect <u>more complex circuits</u> as inputs to a MUX, allowing fns to be synthesized using a combination of MUXes and other logic gates

- The next example illustrates this approach using the 3-input majority function

3-input majority fn implemented using a 2-to-1 MUX

w_1	w_2	w_3	f
0	0	0	0
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	1

w_1	f
0	$w_2 w_3$
1	$w_2 + w_3$

(b) Truth table
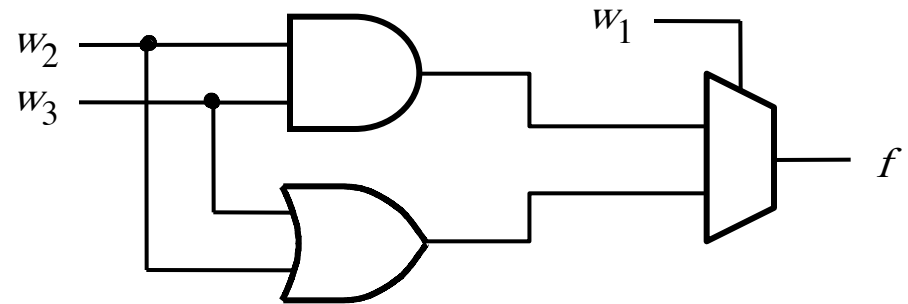
(b) Circuit

This implementation has been derived as follows:

$f = \overline{w}_1 w_2 w_3 + w_1 \overline{w}_2 w_3 + w_1 w_2 \overline{w}_3 + w_1 w_2 w_3$ *-- canonical SOP*

$\quad = \overline{w}_1 (w_2 w_3) + w_1 (\overline{w}_2 w_3 + w_2 \overline{w}_3 + w_2 w_3)$ *-- group and factorize*

$\quad = \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3)$ *-- simplify*

Shannon's expansion theorem

> MUX implementations of logic fns require that a given fn be decomposed with respect to the variables that are used as the select inputs

- This can be achieved by means of Shannon's expansion theorem:
    - Any Boolean function $f(w_1,\ldots,w_n)$ can be written in the form

    $f(w_1,w_2,\ldots,w_n) = \overline{w_1}\cdot f(0,w_2,\ldots,w_n) + w_1\cdot f(1,w_2,\ldots,w_n)$

    (The expansion can be done w.r.t. any of the n variables.)

- Examples:

    Residual factors of terms that include $\overline{w_1}$ Residual factors of terms that include w_1

    - The 3-input majority function:

    $f(w_1, w_2, w_3) = w_1w_2 + w_1w_3 + w_2w_3$

    $\qquad = \overline{w_1}(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) = \overline{w_1}(w_2w_3) + w_1(w_2 + w_3)$

    - The 3-input XOR function:

    $f = w_1 \oplus w_2 \oplus w_3 = \overline{w_1}(\overline{w_2}w_3 + w_2\overline{w_3}) + w_1(\overline{w_2}\overline{w_3} + w_2w_3)$

    $\quad = \overline{w_1}\cdot(w_2 \oplus w_3) + w_1\cdot(\overline{w_2 \oplus w_3})$

Cofactors of *f*

- In Shannon's expansion, the term $f(0,w_2,\ldots,w_n)$ is call-ed the *cofactor* of *f* with respect to $\overline{w}_1$, denoted $f_{\overline{w}_1}$

- Similarly, the term $f(1,w_2,\ldots,w_n)$ is called the *cofactor* of *f* with respect to w_1, written f_{w_1}

- Hence, we can write

  $f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$

- In general, if the expansion is done with respect to variable w_i, then f_{w_i} denotes $f(w_1,\ldots,w_{i-1},1,w_{i+1},\ldots w_n)$ and

  $f(w_1,\ldots,w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$

  whereby the *complexity of the expression may vary*, depending on which variable w_i is used

Complexity of cofactors

- For the function $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, with canonical SOP form:
  $f = \overline{w}_1 \overline{w}_2 w_3 + \overline{w}_1 w_2 w_3 + \overline{w}_1 w_2 \overline{w}_3 + w_1 w_2 \overline{w}_3$,
  decomposition using w_1 gives

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$
$$= \overline{w}_1(\overline{w}_2 w_3 + w_2 w_3 + w_2 \overline{w}_3) + w_1(w_2 \overline{w}_3)$$
$$= \overline{w}_1(w_3 + w_2) + w_1(w_2 \overline{w}_3)$$

- Using w_2 instead of w_1 produces

$$f = \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2}$$
$$= \overline{w}_2(\overline{w}_1 w_3) + w_2(\overline{w}_1 + \overline{w}_3)$$

- Finally, using w_3 gives

$$f = \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3}$$
$$= \overline{w}_3(w_2) + w_3(\overline{w}_1),$$ which is clearly better
b/c it involves less gates

Shannon's expansion with more than one variable

- Shannon's expansion can also be carried out with respect to more than one variable

- For example, expanding a function with respect to variables w_1 and w_2 gives

$$f(w_1,w_2,\ldots,w_n) = \overline{w}_1\overline{w}_2 \cdot f(0,0,w_3,\ldots,w_n) + \overline{w}_1 w_2 \cdot f(0,1,w_3,\ldots,w_n)$$
$$+ w_1\overline{w}_2 \cdot f(1,0,w_3,\ldots,w_n) + w_1 w_2 \cdot f(1,1,w_3,\ldots,w_n)$$

which is in a form that can be implemented with a 4-to-1 MUX using w_1 and w_2 as the select inputs

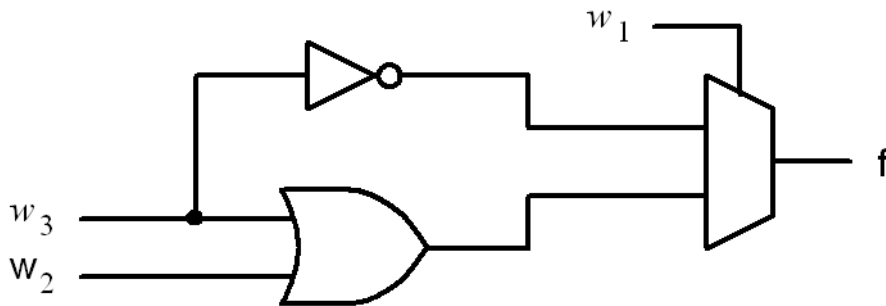- When an expansion is carried out with respect to all n variables, a canonical SOP form results

  – Hence, an n-variable function is implemented by an n-input LUT by programming it with the function's truth table

Example

- Say we wish to implement $f = \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3$ using a 2-to-1 MUX and any other necessary gates.
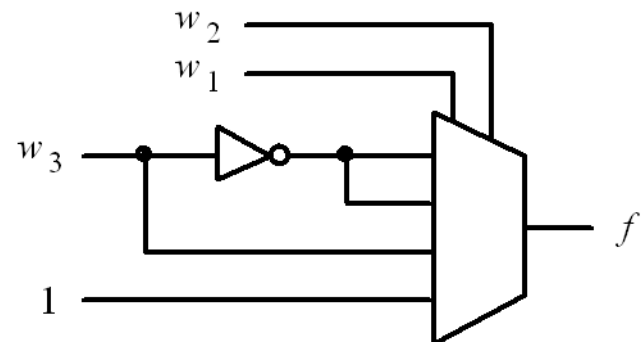
- Shannon's expansion using w_1 gives
$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$
$\quad = \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3)$

- If we are to use a 4-to-1 MUX instead, we can decompose with w_2 as well to give:
$f = \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2}$
$\quad + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1 w_2 f_{w_1 w_2}$
$\quad = \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1 w_2(\overline{w}_3)$
$\quad + w_1\overline{w}_2(w_3) + w_1 w_2(1)$

(a) Using a 2-to-1 multiplexer

(b) Using a 4-to-1 multiplexer

Example (continued)

- To implement
  $f = \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3$ using
  an 8-to-1 MUX, expand f
  using all three variables.

- Shannon's expansion using
  w_1, w_2 and w_3 gives:

$$f = \overline{w}_1\overline{w}_2\overline{w}_3 f_{\overline{w}_1\overline{w}_2\overline{w}_3} +$$
$$\overline{w}_1\overline{w}_2 w_3 f_{\overline{w}_1\overline{w}_2 w_3} +$$
$$\overline{w}_1 w_2\overline{w}_3 f_{\overline{w}_1 w_2\overline{w}_3} +$$
$$\overline{w}_1 w_2 w_3 f_{\overline{w}_1 w_2 w_3} +$$
$$w_1\overline{w}_2\overline{w}_3 f_{w_1\overline{w}_2\overline{w}_3} +$$
$$w_1\overline{w}_2 w_3 f_{w_1\overline{w}_2 w_3} +$$
$$w_1 w_2\overline{w}_3 f_{w_1 w_2\overline{w}_3} +$$
$$w_1 w_2 w_3 f_{w_1 w_2 w_3}$$

- Which equates to:

$$f = \overline{w}_1\overline{w}_2\overline{w}_3 \cdot 1 + \overline{w}_1\overline{w}_2 w_3 \cdot 0 +$$
$$\overline{w}_1 w_2\overline{w}_3 \cdot 1 + \overline{w}_1 w_2 w_3 \cdot 0 +$$
$$w_1\overline{w}_2\overline{w}_3 \cdot 0 + w_1\overline{w}_2 w_3 \cdot 1 +$$
$$w_1 w_2\overline{w}_3 \cdot 1 + w_1 w_2 w_3 \cdot 1 +$$

and is implemented as
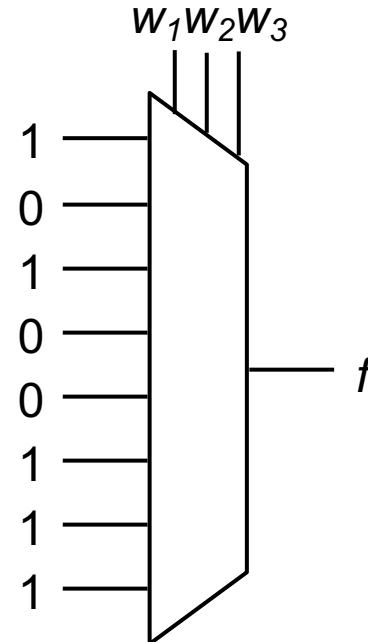
Example: mapping to *3*-LUTs

- Some FPGAs may use 3-input lookup tables (3-LUTs) to implement logic functions

- Any fn of 4 variables can be mapped to at most three 3-LUTs

- Consider

  $f = \overline{w}_2 w_3 + \overline{w}_1 w_2 \overline{w}_3 + w_2 \overline{w}_3 w_4$
  $\quad + w_1 \overline{w}_2 \overline{w}_4$

- Expansion with respect to w_1 produces

  $f = \overline{w}_1(\overline{w}_2 w_3 + \boxed{w_2 \overline{w}_3 + w_2 \overline{w}_3 w_4})$
  $\quad + w_1(\overline{w}_2 w_3 + w_2 \overline{w}_3 w_4 + \overline{w}_2 \overline{w}_4)$

- Using w_2 instead of w_1 gives

  $f = \overline{w}_2(w_3 + w_1 \overline{w}_4)$
  $\quad + w_2(\overline{w}_1 \overline{w}_3 + \overline{w}_3 w_4)$

Implements the truth table for any function of 3 variables

Apply absorption
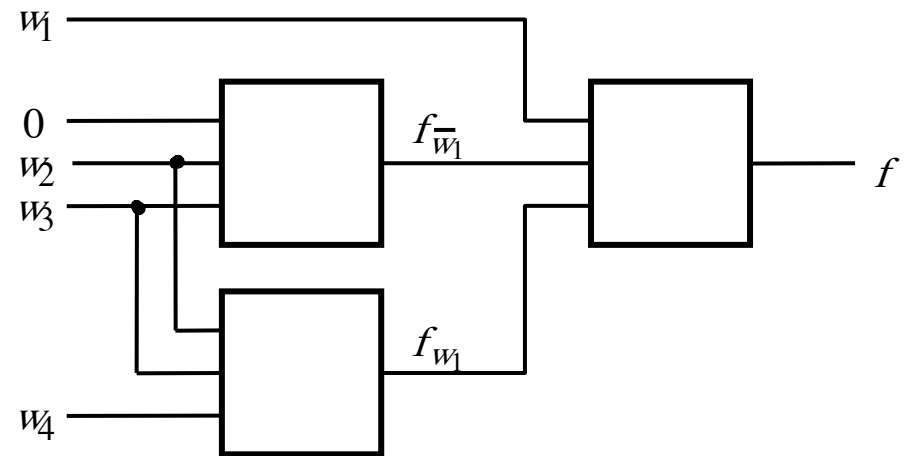
(a) Using three 3-LUTs

$f_{w_2} = \overline{f_{\overline{w_2}}}$ in this case
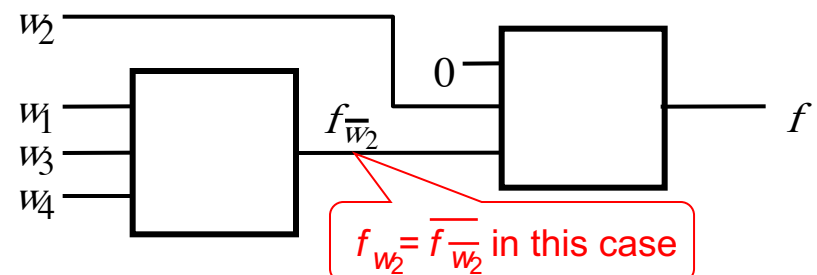
(b) Using two 3-LUTs

Decoders

- Decoder circuits are used to decode encoded information

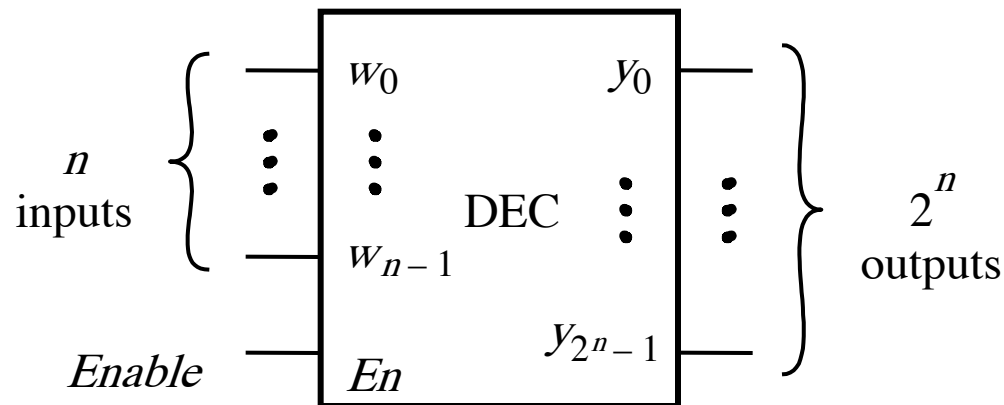 A *binary decoder* (DEC), as shown below, is a logic circuit with *n* inputs and 2^n outputs
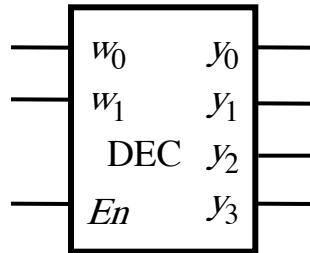
  - Only one output is asserted at a time, and each output corresponds to one valuation of the inputs
  - A decoder has an enable input *En* that is used to disable the outputs so that no output is asserted when *En = 0*

A 2-to-4 binary decoder

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

(a) Truth table

(b) Graphical symbol

- An n-bit binary code in which exactly one of the bits is set to 1 at a time is called *one-hot encoded*
  - The single bit that is set to 1 is said to be "hot"

- <u>The outputs of a binary decoder are one-hot encoded</u>

- Larger decoders can be built using the SOP structure of (c), or they can be built from smaller decoders
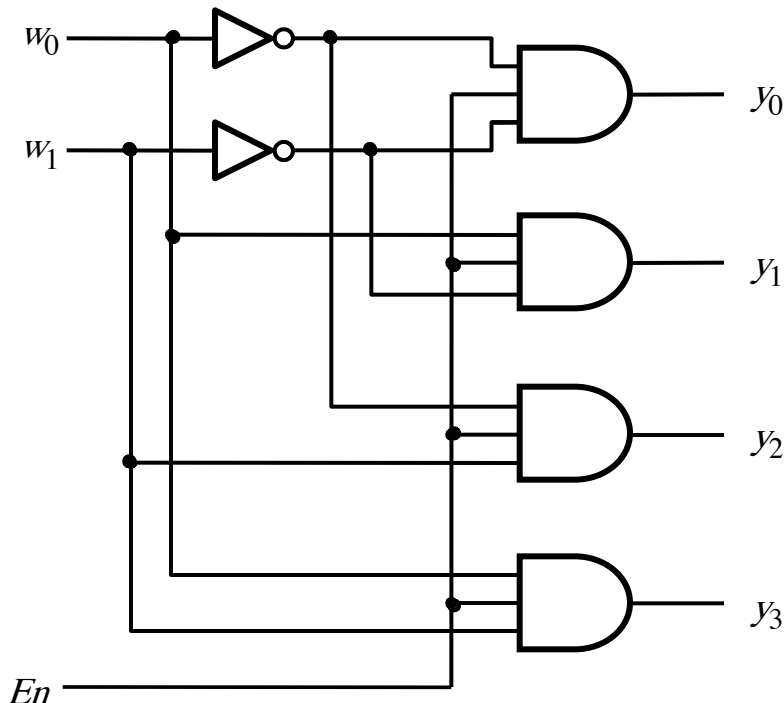
(c) Logic circuit

A 3-to-8 decoder using two 2-to-4 decoders
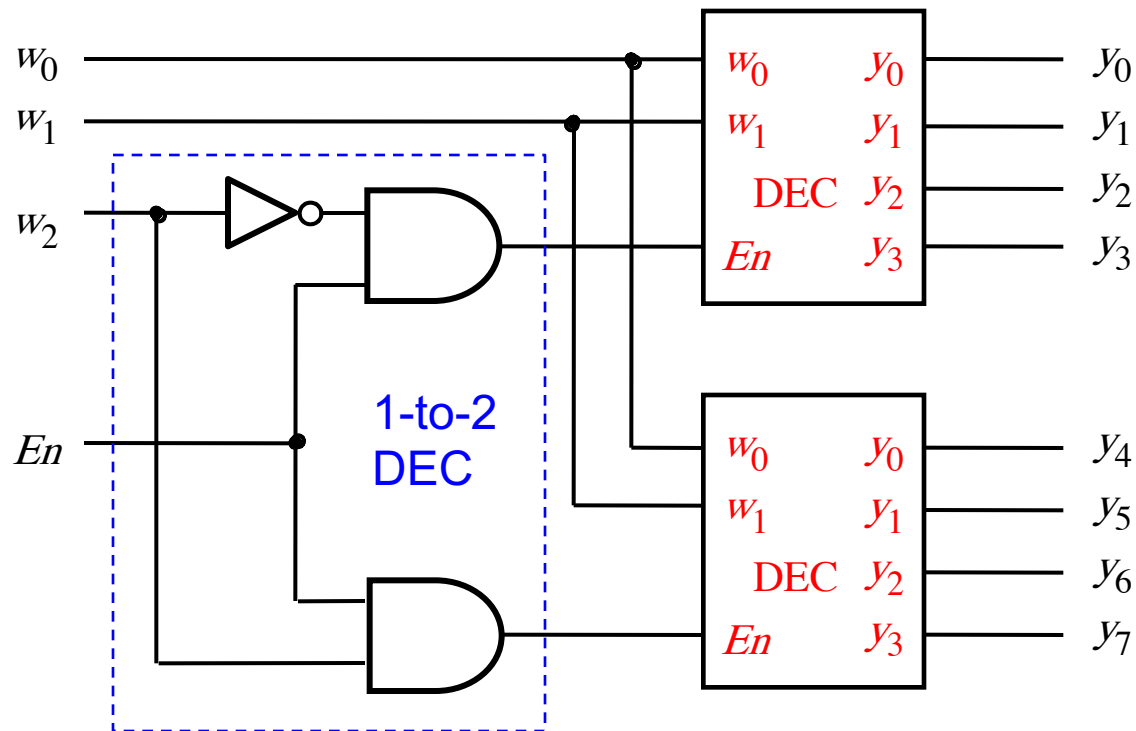
A 4-to-16 decoder built using a decoder tree

Combinational Building Blocks
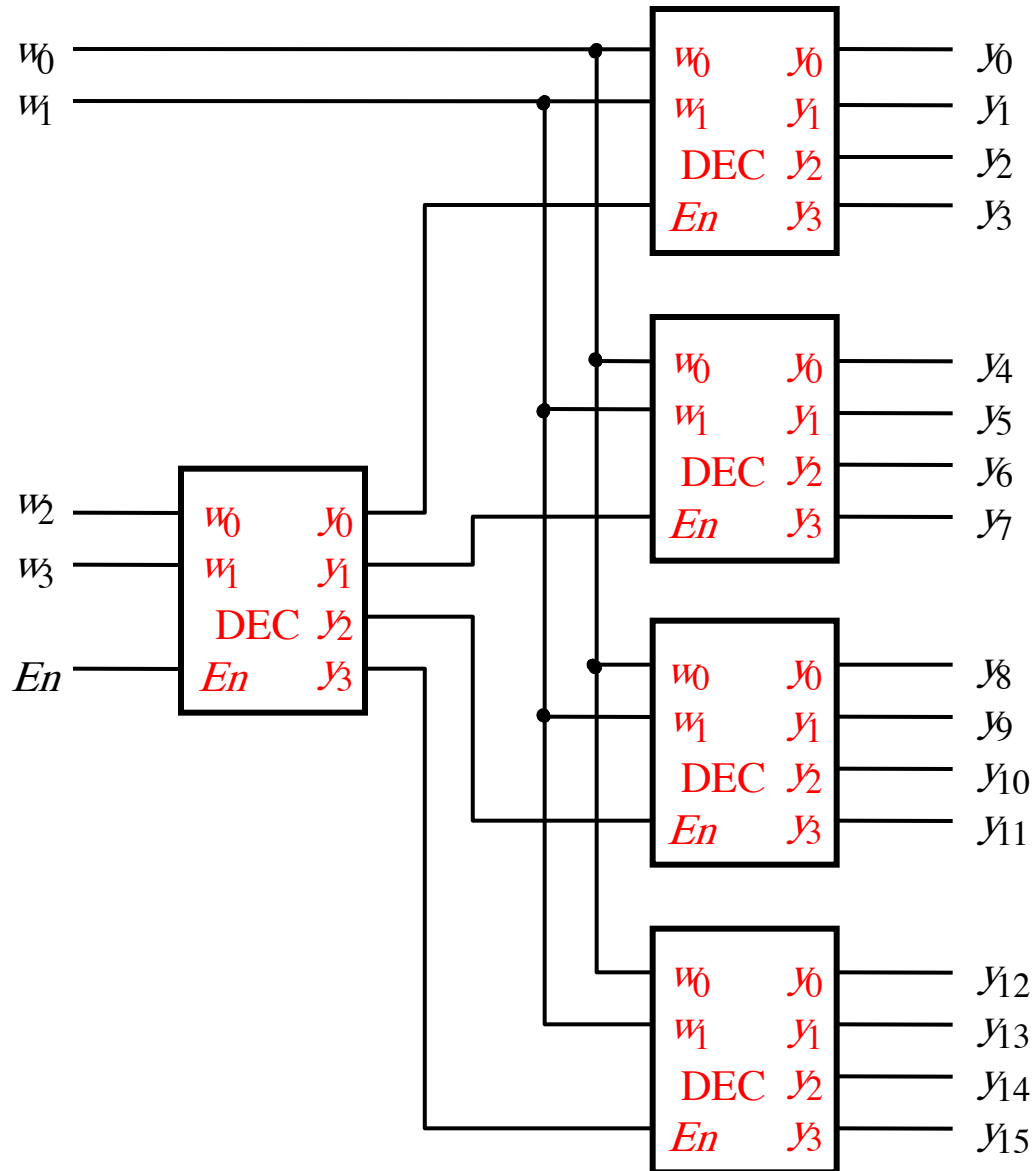
A 4-to-1 multiplexer built using a decoder
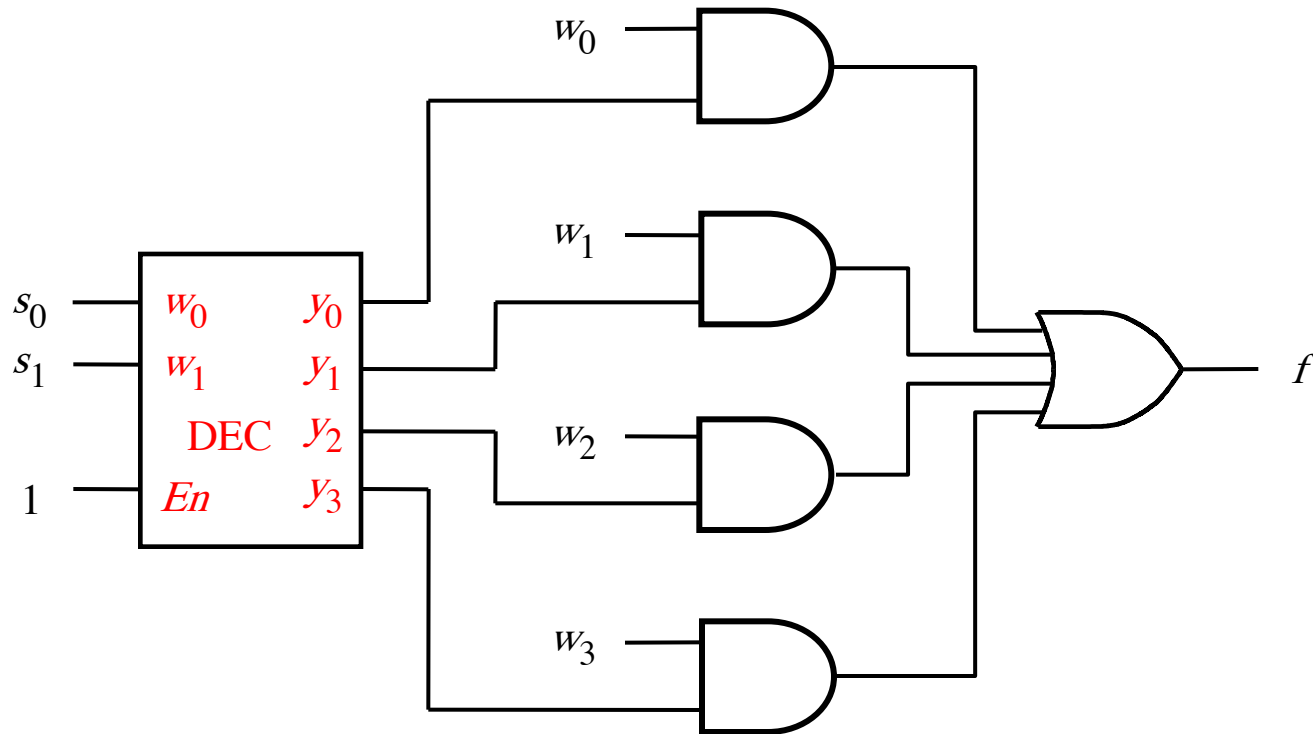
Using a DEC & 3-state buffers to build a 4-to-1 MUX
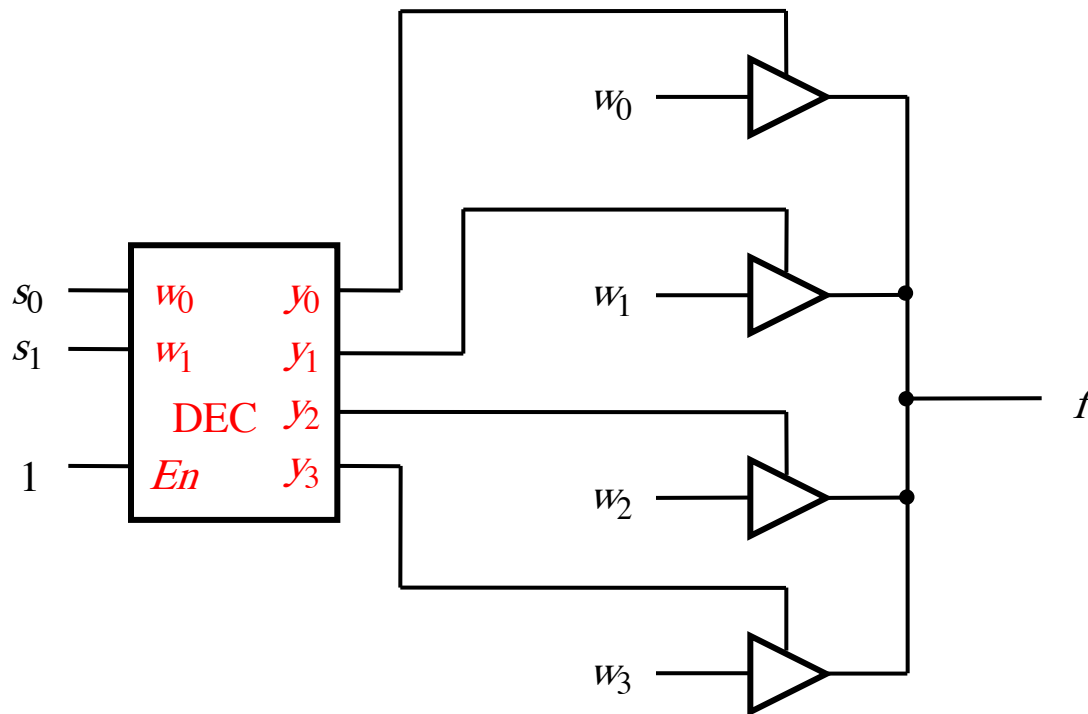
- 3-state buffers are short-circuit when the control input is asserted, and high-impedance (high-Z $\approx$ open circuit) when the control input is deasserted
- This allows their outputs to be wired together as long as only one buffer is asserted at a time (as ensured here by the decoder)

Demultiplexing

- Note that a decoder on its own can also be used as a *1-to-2^n* demultiplexer
  - The *En* input plays the role of data-in and one of 2^n outputs is selected using the *n* select bits
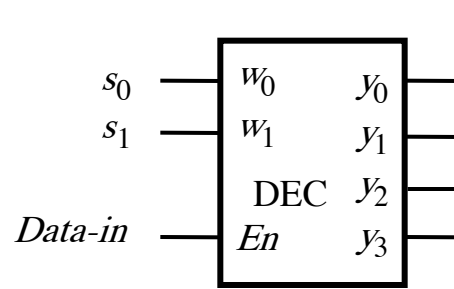
Data-in	s_1	s_0	y_3	y_2	y_1	y_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
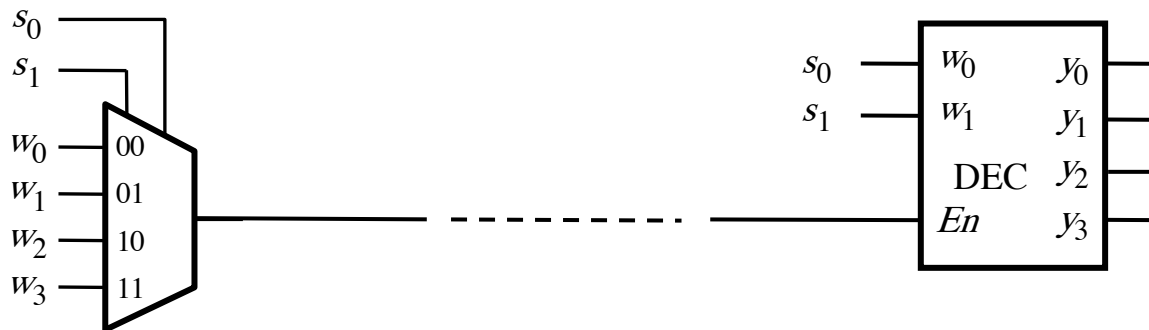
Basis for a simple *time-division multiplexed* communications system, which saves wires/channel

Use of DEC block to decode address for a $2^m \times n$ ROM (read-only memory) block

Address $\left\{ \begin{array}{l} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{array} \right.$

m-to-2^m decoder

Sel_0 | 0/1 | 0/1 | $\cdots$ | 0/1
Sel_1 | 0/1 | 0/1 | $\cdots$ | 0/1
Sel_2 | 0/1 | 0/1 | $\cdots$ | 0/1
Sel_{2^m-1} | 0/1 | 0/1 | $\cdots$ | 0/1

A given address asserts one of the select lines to allow the n-bit word stored at that address to be read.

Read

Data $\left\{ d_{n-1} \quad d_{n-2} \quad\quad d_0 \right\}$
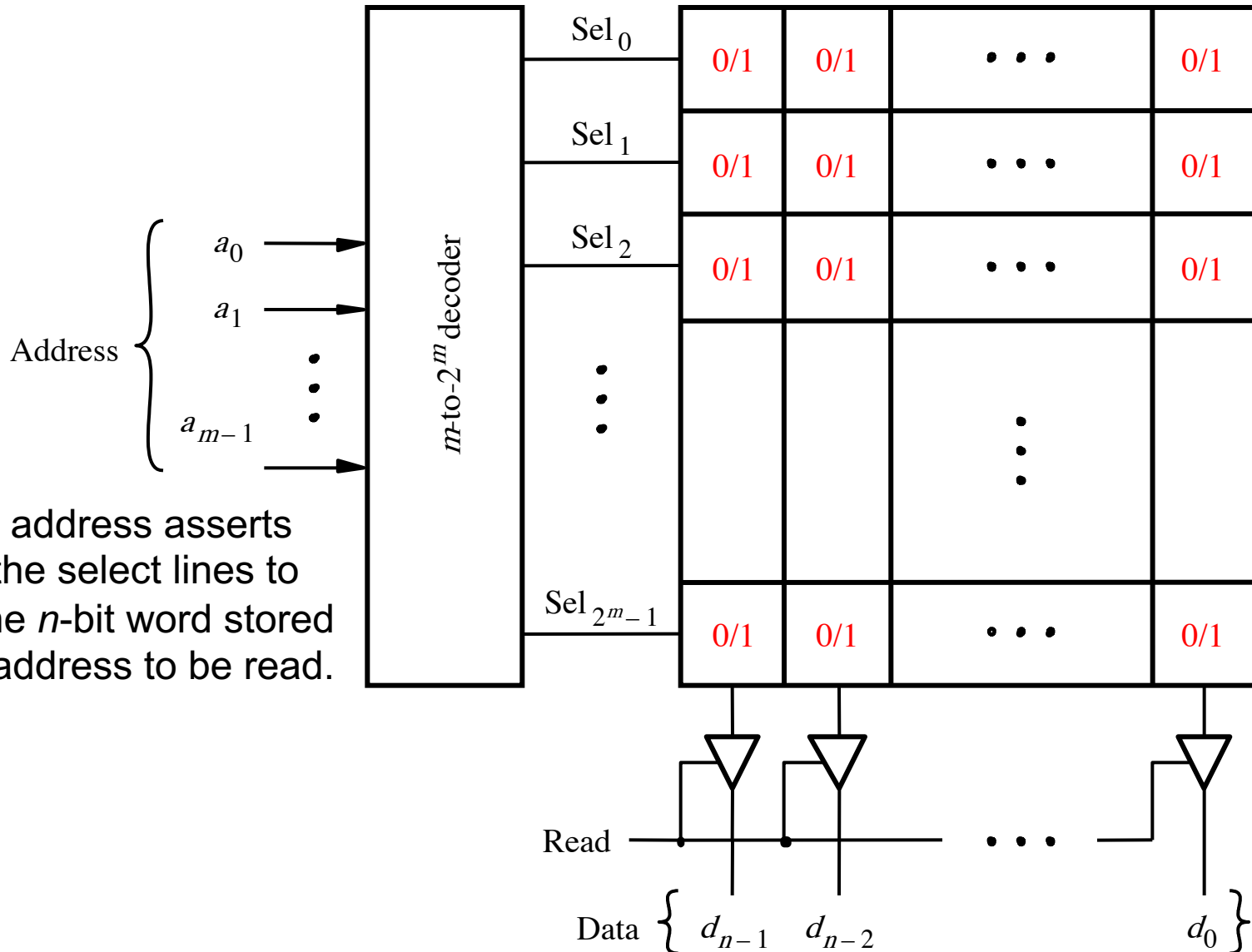
Encoders

- An encoder performs the opposite function of a decoder
  - It encodes given information into a more compact form

A *binary encoder* encodes information from 2^n inputs into an n-bit code as shown below

  - <u>Exactly one of the input signals should have a value of 1</u>, and the outputs present the binary number that identifies which input is equal to 1
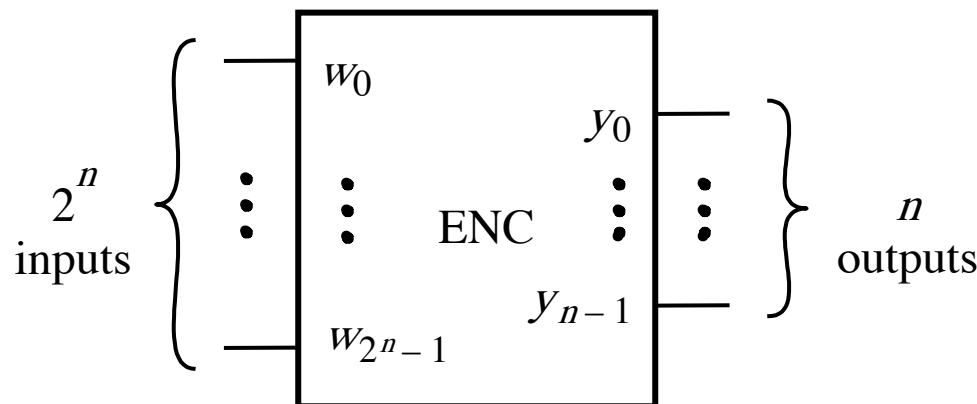
A 4-to-2 binary encoder

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

(a) Truth table

(b) Circuit

Priority encoders

- In a priority encoder each input has a priority level associated with it

- The encoder outputs indicate the active input that has the highest priority

- Truth table for a 4-to-2 priority encoder

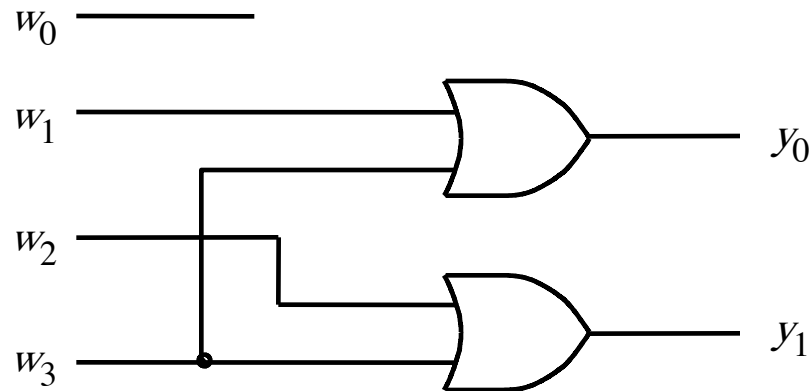| w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

- The previous techniques can be used to synthesize the output functions

- However, a more convenient approach is to define intermediate signals

$i_0 = \overline{w}_3\overline{w}_2\overline{w}_1 w_0$

$i_1 = \overline{w}_3\overline{w}_2 w_1$

$i_2 = \overline{w}_3 w_2$

$i_3 = w_3$

- The outputs can then be written as:

$y_0 = i_1 + i_3$

$y_1 = i_2 + i_3$

$z = i_0 + i_1 + i_2 + i_3$

A BCD-to-7-segment display code converter

- The purpose of decoders and encoders is to convert from one type of input encoding to a different output encoding

- A typical example of a **code converter** is a BCD-to-7-segment decoder, which converts a binary-coded decimal digit into information suitable for driving a digit-oriented display

- A circuit that implements the truth table can be derived using the synthesis techniques previously discussed
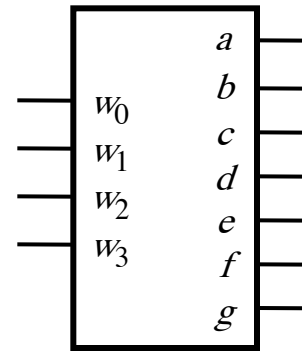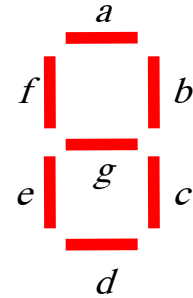
(a) Code converter   (b) 7-segment display

w_3	w_2	w_1	w_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	1	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

(c) Truth table

Combinational VHDL

- Non-simple assignment statements
  - Selected assignment
  - Conditional assignment

- Sequential statements
  - If-then-else
  - Case

- Concurrent statements
  - Process vs assignment statements

VHDL code for a 2-to-1 multiplexer using a <u>selected signal assignment</u>

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (w0, w1, s : IN STD_LOGIC ;
           f : OUT STD_LOGIC) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    WITH s SELECT
       f <= w0 WHEN '0',
          w1 WHEN OTHERS ;

END Behavior ;

(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

(c) Sum-of-products circuit
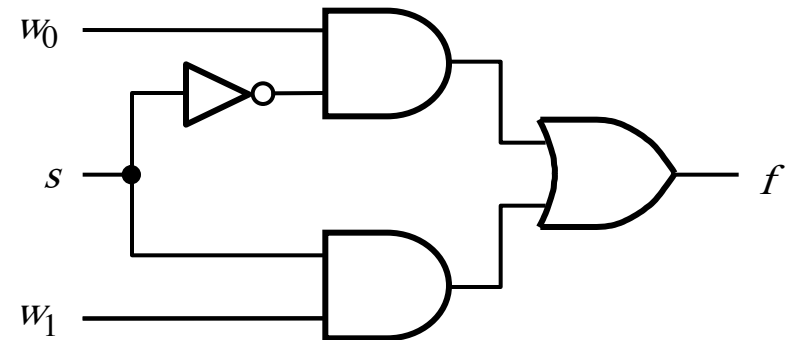
The "WITH x SELECT" idiom ALWAYS infers a multiplexer. So when your design calls for one, use it!

All possible valuations of the condition "s" need to be considered

VHDL code for a 4-to-1 multiplexer
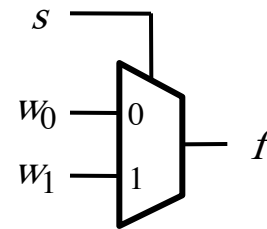
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT (  w0, w1, w2, w3    : IN      STD_LOGIC ;
            s                 : IN      STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            f                 : OUT     STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
    WITH s SELECT
        f <= w0 WHEN "00",
             w1 WHEN "01",
             w2 WHEN "10",
             w3 WHEN OTHERS ;
END Behavior ;
```


(a) Graphic symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

Note "" used for multi-bit constant; '' used for single bit constants

VHDL code for a 4-to-1 MUX (cont.)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE mux4to1_package IS
    COMPONENT mux4to1
        PORT (  w0, w1, w2, w3    : IN STD_LOGIC ;
                s                 : IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                f                 : OUT STD_LOGIC ) ;
    END COMPONENT ;
END mux4to1_package ;
```
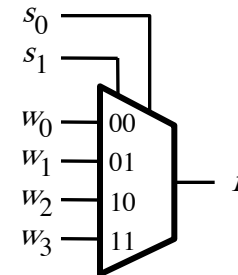
- Allows us to use the component from the WORK library
- Allows the component to be instantiated within the body of another entity's architecture without a declaration within the header of that architecture

Hierarchical code for a 16-to-1 MUX

```
1    LIBRARY ieee ;
2    USE ieee.std_logic_1164.all ;
3    LIBRARY work ;        -- this line not really needed for packages
4    USE work.mux4to1_package.all ;   -- in the "work"ing directory

5    ENTITY mux16to1 IS
6        PORT (   w : IN STD_LOGIC_VECTOR(0 TO 15) ;
7                 s : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
8                 f : OUT STD_LOGIC ) ;
9    END mux16to1 ;

10   ARCHITECTURE Structure OF mux16to1 IS
11       SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
12   BEGIN
13       Mux1: mux4to1 PORT MAP ( w(0), w(1), w(2), w(3), s(1 DOWNTO 0), m(0) ) ;
14       Mux2: mux4to1 PORT MAP ( w(4), w(5), w(6), w(7), s(1 DOWNTO 0), m(1) ) ;
15       Mux3: mux4to1 PORT MAP ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
16       Mux4: mux4to1 PORT MAP ( w(12), w(13), w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
17       Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
18  END Structure ;
```
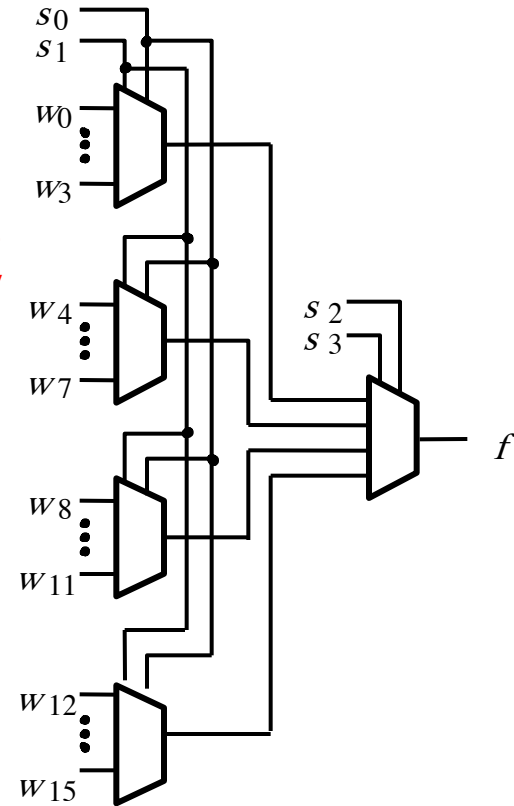
Code for a 16-to-1 MUX using a generate statement

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.mux4to1_package.all ;

ENTITY mux16to1 IS
    PORT ( w  : IN STD_LOGIC_VECTOR(0 TO 15) ;
           s  : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f  : OUT STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Muxes: mux4to1 PORT MAP (
            w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
    END GENERATE ;
    Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```


FOR…GENERATE statement is used like a macro as shorthand to list repeated concurrent statements
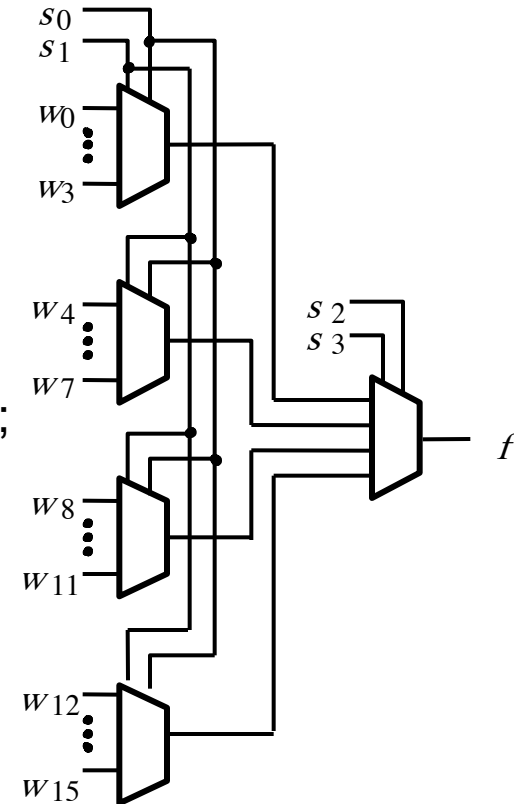
VHDL code for a 2-to-4 binary decoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w    : IN      STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            En  : IN      STD_LOGIC ;
            y   : OUT     STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;        concatenation
    WITH Enw SELECT
        y <= "1000" WHEN "100",
             "0100" WHEN "101",
             "0010" WHEN "110",
             "0001" WHEN "111",
             "0000" WHEN OTHERS ;

END Behavior ;
```

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

(a) Truth table

w_0 y_0
w_1 y_1
DEC y_2
En y_3

(b) Graphical symbol

Hierarchical code for a 4-to-16 binary decoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec4to16 IS
    PORT (   w : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
             En : IN STD_LOGIC ;
             y : OUT STD_LOGIC_VECTOR(0 TO 15) ) ;
END dec4to16 ;

ARCHITECTURE Structure OF dec4to16 IS
    COMPONENT dec2to4
        PORT (   w : IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                 En : IN STD_LOGIC ;
                 y : OUT STD_LOGIC_VECTOR(0 TO 3) ) ;
    END COMPONENT ;
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Dec_right: dec2to4 PORT MAP ( w(1 DOWNTO 0), m(i), y(4*i TO 4*i+3) );
        G2: IF i=3 GENERATE
            Dec_left: dec2to4 PORT MAP ( w(3 DOWNTO 2), En, m ) ;
        END GENERATE ;
    END GENERATE ;
END Structure ;
```
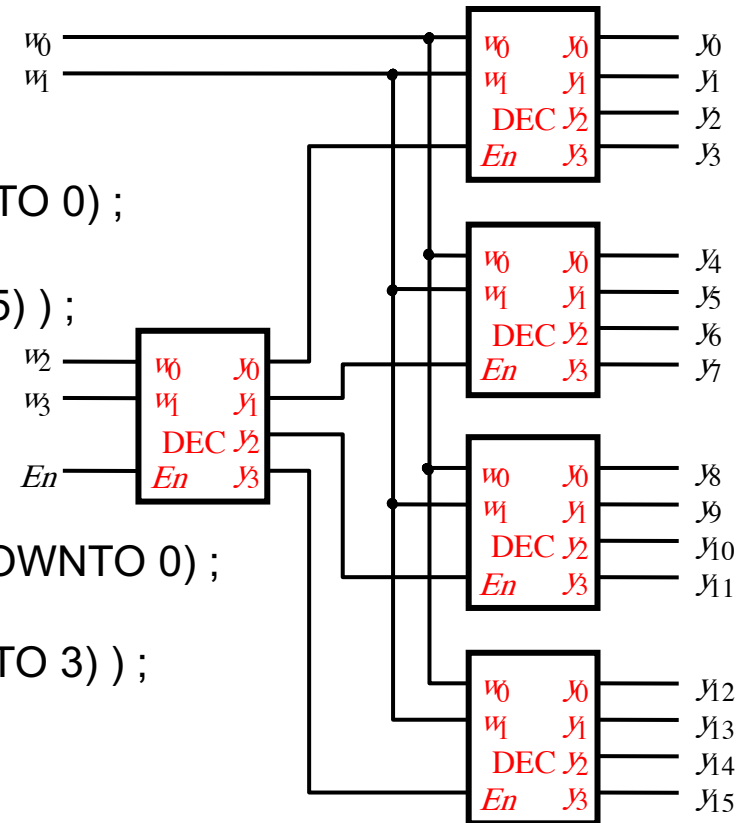
IF…GENERATE statement conditionally instantiates a concurrent statement

Specification of a 2-to-1 multiplexer using a __conditional signal assignment__

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
     PORT (w0, w1, s      : IN      STD_LOGIC ;
                   f                : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
     f <= w0 WHEN s = '0' ELSE w1 ;
END Behavior ;
```

VHDL code for a priority encoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y   : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z   : OUT   STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    y <=  "11" WHEN w(3) = '1' ELSE
          "10" WHEN w(2) = '1' ELSE
          "01" WHEN w(1) = '1' ELSE
          "00" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Conditions evaluated in listed order

Arbitrary, unrelated conditions possible

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

Less efficient code for a priority encoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT (   w : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
             y : OUT     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
             z : OUT     STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    WITH w SELECT
        y <=    "00" WHEN "0001",
                "01" WHEN "0010",
                "01" WHEN "0011",
                "10" WHEN "0100",
                "10" WHEN "0101",
                "10" WHEN "0110",
                "10" WHEN "0111",
                "11" WHEN OTHERS ;
    WITH w SELECT
        z <=    '0' WHEN "0000",
                '1' WHEN OTHERS ;
END Behavior ;
```

Conditions must be mutually exclusive and are evaluated in parallel

A 2-to-1 MUX specified using an if-then-else statement

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (w0, w1, s: IN STD_LOGIC ;
          f: OUT STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Sensititivity list

IMPORTANT! The assignment to *f* is not committed until the current invocation of the process ends!

Processes are typically used to express complex behaviours

- Processes contain sequential statements i.e statements within the *PROCESS* are <u>evaluated one after another</u>
  - *When there are consecutive assignments to the one signal, the last assignment made is the only one that is committed when the process exits*

- A *process is triggered* when a signal in its sensitivity list has a change in value
  - For <u>combinational processes</u>, the sensitivity list **must ONLY include** signals that (i) appear on the RHS of assignment statements, or (ii) that are involved in conditional expressions, within the process. In this case signals *w0*, *w1* and *s* satisfy this requirement.

Alternative code for a 2-to-1 multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (  w0, w1, s      : IN       STD_LOGIC ;
                  f               : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        f <= w0 ;
        IF s = '1' THEN
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Works because of sequential evaluation

Q1: What is the behaviour of the process if the first assignment statement were moved below the IF statement?

Q2: What is the behaviour if the first assignment statement were removed entirely?

A priority encoder specified using if-then-else

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY priority IS
 PORT (
  w : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
  y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  z : OUT STD_LOGIC) ;
END priority ;
ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS (w)
    BEGIN
        IF w(3) = '1' THEN
            y <= "11" ;
        ELSIF w(2) = '1' THEN
            y <= "10" ;
        ELSIF w(1) = '1' THEN
            y <= "01" ;
        ELSE
            y <= "00" ;
        END IF ;
    END PROCESS ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;

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

Note that process statements are concurrently evaluated with all other concurrent statements – *they are, in effect, a compound form of concurrent statement used to express non-trivial behaviour*

Alternative code for the priority encoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT   STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        y <= "00" ;
        IF w(1) = '1' THEN y <= "01" ; END IF ;
        IF w(2) = '1' THEN y <= "10" ; END IF ;
        IF w(3) = '1' THEN y <= "11" ; END IF ;

        z <= '1' ;
        IF w = "0000" THEN z <= '0' ; END IF ;
    END PROCESS ;
END Behavior ;
```

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

Code for a one-bit equality comparator

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
    PORT ( A, B   : IN      STD_LOGIC ;
              AeqB  : OUT  STD_LOGIC ) ;
END compare1 ;

ARCHITECTURE Behavior OF compare1 IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        AeqB <= '0' ;
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Can you visualize the resulting circuit?

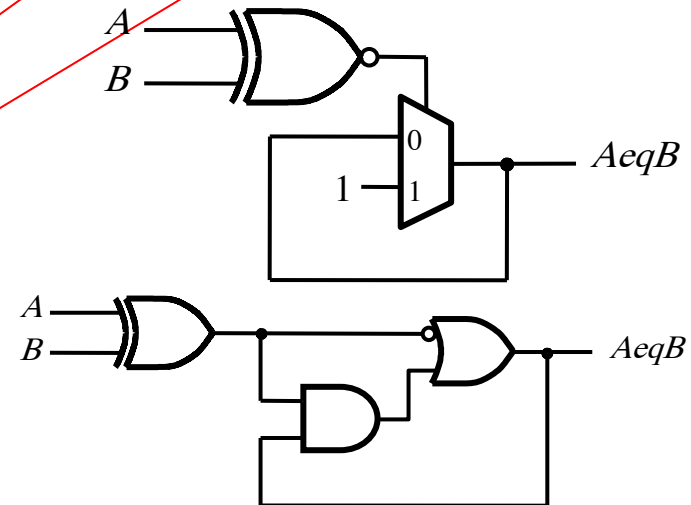*Is there a better way of specifying its behaviour?*

An example of *incorrect code* that results in <u>implied memory</u>

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B   : IN      STD_LOGIC ;
           AeqB  : OUT   STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Resulting circuit has to remember the value of AeqB when either A or B change and A /= B

Processes that describe <u>combinational logic</u> (circuits without memory) **MUST** <u>assign a value</u> to every output signal for every execution path

A case statement that represents a 2-to-1 MUX

```vhdl
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (  w0, w1, s : IN      STD_LOGIC ;
                  f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        CASE s IS
            WHEN '0' =>
                f <= w0 ;
            WHEN OTHERS =>
                f <= w1 ;
        END CASE ;
    END PROCESS ;

END Behavior ;
```

Again, all possible valuations of the conditional expression need to be catered for so as to assign a value to *f* under all input conditions and avoid sequential behaviour (the need for memory)

A 2-to-4 binary decoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY dec2to4 IS
     PORT ( w    : IN       STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            En   : IN       STD_LOGIC ;
            y    : OUT      STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;


ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
     PROCESS ( w, En )
     BEGIN
        IF En = '1' THEN
           CASE w IS
                WHEN "00" =>        y <= "1000" ;
                WHEN "01" =>        y <= "0100" ;
                WHEN "10" =>        y <= "0010" ;
                WHEN OTHERS =>   y <= "0001" ;
           END CASE ;
        ELSE
           y <= "0000" ;
        END IF ;
     END PROCESS ;
END Behavior ;
```

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

Code for a BCD-to-7-segment decoder

```vhdl
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY seg7 IS
    PORT (bcd   : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
          leds  : OUT   STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;
ARCHITECTURE Behavior OF seg7 IS
BEGIN
    PROCESS ( bcd )
    BEGIN
        CASE bcd IS                           -- abcdefg - common anode
            WHEN "0000"     => leds     <=    "0000001" ;
            WHEN "0001"     => leds     <=    "1001111" ;
            WHEN "0010"     => leds     <=    "0010010" ;
            WHEN "0011"     => leds     <=    "0000110" ;
            WHEN "0100"     => leds     <=    "1001100" ;
            WHEN "0101"     => leds     <=    "0100100" ;
            WHEN "0110"     => leds     <=    "0100000" ;
            WHEN "0111"     => leds     <=    "0001111" ;
            WHEN "1000"     => leds     <=    "0000000" ;
            WHEN "1001"     => leds     <=    "0001100" ;
            WHEN OTHERS     => leds     <=    "-------" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```
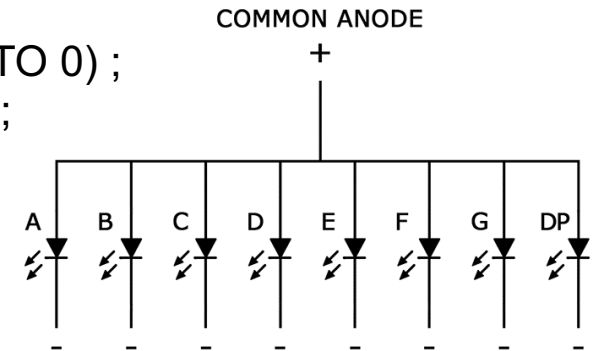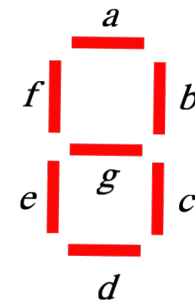
COMMON ANODE
+

A | B | C | D | E | F | G | DP

7-segment display

Specifying the functionality of a 74381 ALU chip

```vhdl
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
ENTITY alu IS
    PORT ( s : IN STD_LOGIC_VECTOR(2 DOWNTO 0) ;
            A, B : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            F : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END alu ;

ARCHITECTURE Behavior OF alu IS
BEGIN
    PROCESS ( s, A, B )
    BEGIN
        CASE s IS
            WHEN "000" =>
                F <= "0000" ;
            WHEN "001" =>
                F <= B - A ;
            WHEN "010" =>
                F <= A - B ;
            WHEN "011" =>
                F <= A + B ;
```

Operation	Inputs $s_2\ s_1\ s_0$	Outputs F
Clear	0 0 0	0 0 0 0
B−A	0 0 1	$B - A$
A−B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

```vhdl
            WHEN "100" =>
                F <= A XOR B ;
            WHEN "101" =>
                F <= A OR B ;
            WHEN "110" =>
                F <= A AND B ;
            WHEN OTHERS =>
                F <= "1111" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

VHDL operators used for synthesis

Table 6.2. VHDL operators (used for synthesis).

Operator category	Operator symbol	Operation performed
Logical	AND OR NAND NOR XOR XNOR NOT	AND OR Not AND Not OR XOR Not XOR NOT
Relational	= /= > < >= <=	Equality Inequality Greater than Less than Greater than or equal to Less than or equal to
Arithmetic	+ − * /	Addition Subtraction Multiplication Division
Concatenation	&	Concatenation
Shift and Rotate	SLL SRL SLA SRA ROL ROR	Shift left logical Shift right logical Shift left arithmetic Shift right arithmetic Rotate left Rotate right

- Precedence in the table to the left is from top to bottom between categories
- Operators within the same category have the same precedence and are therefore evaluated from left to right
- Often good to parenthesize expressions to explicitly confirm evaluation order
- Note also that

  s <= a + b + c + d;

  results in 3 sequential additions, whereas

  s <= (a + b) + (c + d);

  performs two sub-additions in parallel and then one final addition for 2/3 the delay

Examples

Brown & Vranesic: Example 6.25

Problem:

Implement the function $f(w_1, w_2, w_3) = \Sigma\, m(0, 1, 3, 4, 6, 7)$ using a 3-to-8 binary decoder and an OR gate

Example 6.27

Problem:

Implement the function

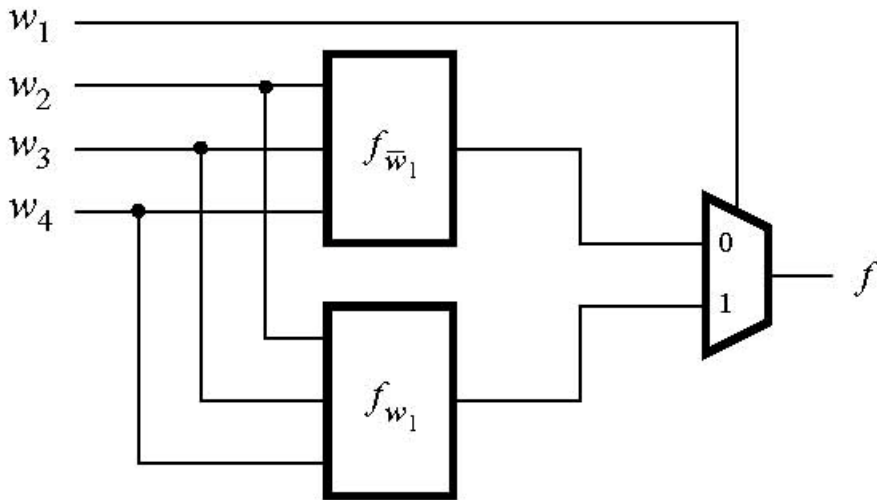$$f = \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

using a 4-to-1 multiplexer and as few gates as possible.

Assume that only the uncomplemented inputs are available as inputs.

Example 6.27

Example 6.29

- For a four-variable function, $f(w_1,\ldots,w_4)$, Shannon's expansion with respect to w_1 is $f = \overline{w_1}f_{\overline{w_1}} + w_1 f_{w_1}$, which can be implemented as in circuit (a):
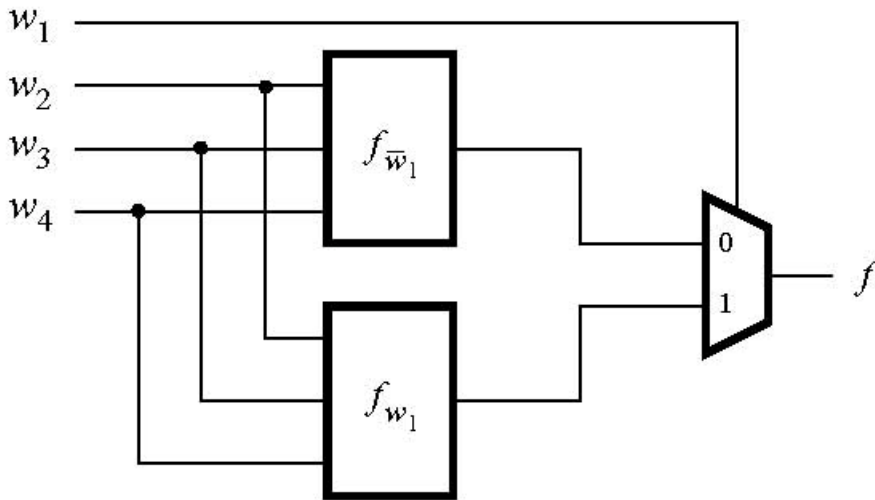
Problem 1:

If the composition yields $f_{\overline{w_1}} = 0$ then the multiplexer in figure (a) can be replaced by a single logic gate. Show this circuit.

(a) Shannon's expansion of the function f.

Example 6.29

- For a four-variable function, $f(w_1, \ldots, w_4)$, Shannon's expansion with respect to w_1 is $f = \overline{w_1} f_{\overline{w_1}} + w_1 f_{w_1}$, which can be implemented as in circuit (a):

Problem 2:

If the composition yields $f_{w_1} = 1$ then the multiplexer in figure (a) can be replaced by a single logic gate. Show this circuit.

(a) Shannon's expansion of the function f.

Example 6.31

Problem:

Design a circuit that can shift a 4-bit vector $W = w_3w_2w_1w_0$ one bit position to the right when a control signal *Shift* is equal to 1.
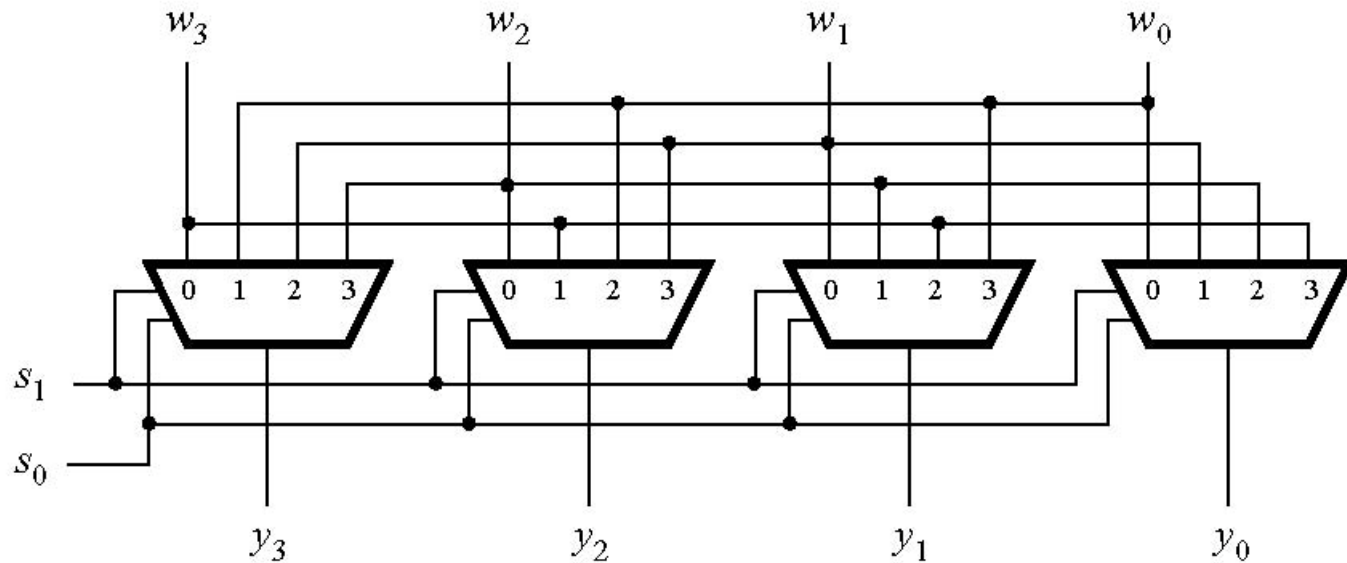
Let the outputs of the circuit be $Y = y_3y_2y_1y_0$ and a signal k, such that if *Shift = 1* then $y_3 = 0$, $y_2 = w_3$, $y_1 = w_2$, $y_0 = w_1$, and $k = w_0$. If *Shift = 0*, then $Y = W$ and $k = 0$.
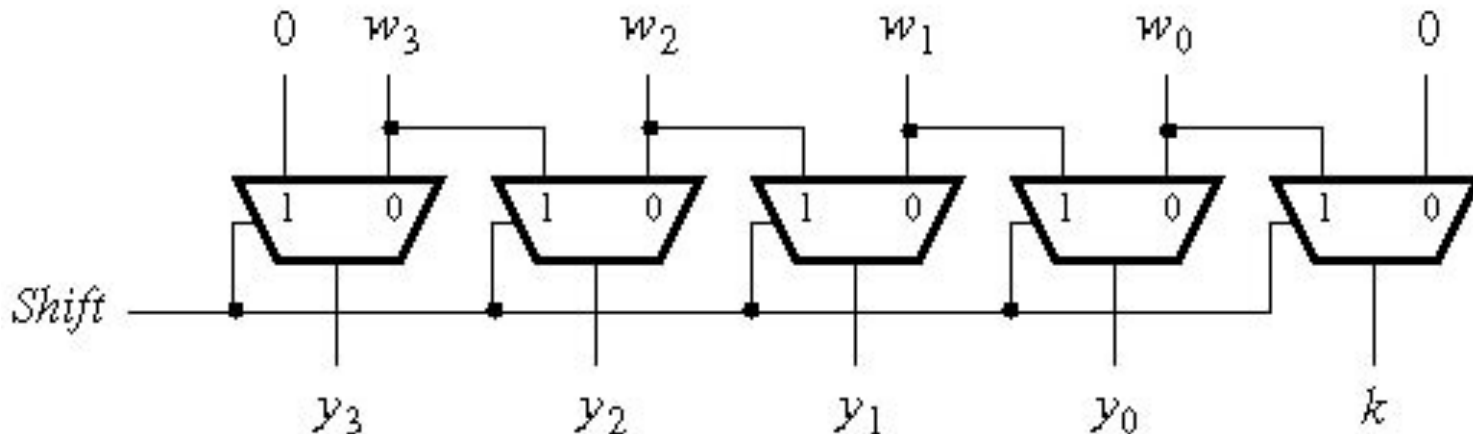
4-bit Barrel Shifter

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

(a) Truth table

(b) Circuit

VHDL code for the shifter of Example 6.31

```vhdl
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shifter IS
    PORT ( w      : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Shift  : IN    STD_LOGIC ;
           y      : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           k      : OUT   STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = '1' THEN
            y(3) <= '0' ;
            y(2 DOWNTO 0) <= w(3 DOWNTO 1) ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= '0' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```
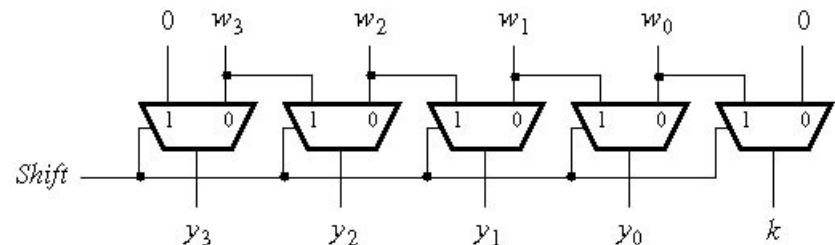
Alternative code using SRL op

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;

ENTITY shifter IS
    PORT ( w      : IN    UNSIGNED(3 DOWNTO 0) ;
            Shift  : IN    STD_LOGIC ;
            y      : OUT   UNSIGNED(3 DOWNTO 0) ;
            k      : OUT   STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = '1' THEN
            y <= w SRL 1 ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= '0' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```