

## Overview

This lab introduces several combinational circuits that are frequently used by digital designers, including a data selector (also called a multiplexor or just "mux"), a binary decoder, a seven-segment decoder, an encoder, and a shifter. Each of these circuits can be used by themselves in the solution of some simpler logic problems, but they are more often used as building blocks in the creation of larger, more complex circuits. In this module, these circuits will be developed from first principles following a general design procedure that will serve as a model for all later designs. In later modules, these circuits will be used as modular (or "macro") building blocks in larger designs.

This general design procedure has five main steps. First, you must gain a clear understanding of the design intent of each circuit before any design activities start. When you are doing original design work, this understanding comes from many sources, including other persons, previous or competing designs, research papers, or your own insightful thinking. For now, the discussion that leads the presentation of each new circuit is intended to impart that clear understanding to you. Second, a block diagram that shows all circuit inputs and outputs will be developed. A block diagram is an indispensable part of any design, especially when dealing with complex circuits. In conceiving and capturing a block diagram, you are committing to a set of input and output signals, and those signals define the context and boundaries of your design. Third, the design requirements will be captured in an engineering formalism like a truth table or logic equations. This formalism removes all ambiguity from the design, and establishes a solid specification for the circuit. Fourth, the formally stated requirements will be used to find minimal circuits that meet the specifications. And finally, those minimal circuits will be created and implemented using the ISE/WebPack tool and a Diligent board, and verified in hardware to ensure they meet their behavioral requirements.

### Before beginning this lab, you should...

- Be able to specify, design, and minimize combinational logic systems
- Be able to create schematic-based or VHDL-based designs in the Xilinx WebPack environment
- Be able to download designs created in WebPack to the Diligent circuit board.

### After completing this lab, you should...

- Understand the application, function, and structure of decoder, multiplexor, encoder, and shifter circuits;
- Know how to use these circuits in the solution of larger problems;
- Be able to quickly implement these circuits in the Xilinx CAD tool environment.

### This lab exercise requires...

- A windows computer running Xilinx WebPack
- A Diligent board

## Combinational Circuit Blocks

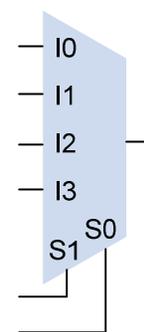
### Data Selectors (Multiplexors)

Data selectors, more commonly called multiplexors (or just muxes), function by connecting one of their input signals to their output signal as directed by their “select” or control input signals. Muxes have N data inputs and  $\log_2 N$  select inputs, and a single output. In operation, the select inputs determine which data input drives the output, and whatever voltage appears on the selected input is driven on the output. All non-selected data inputs are ignored. As an example, if the select inputs of a 4:1 mux are ‘1’ and ‘0’, then the output Y will be driven to the same voltage present on input I2.

S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

**4:1 mux truth table**

Common mux sizes are 2:1 (1 select input), 4:1 (2 select inputs), and 8:1 (3 select inputs). The truth table shown specifies the behavior of a 4:1 mux. Note the use of entered variables in the truth table – if entered variables were not used, the truth table would require six columns and  $2^6$  or 64 rows. In general, when entered-variable truth tables are used to define a circuit, “control” inputs are shown as column-heading variables, and data inputs are used as entered variables.



**Mux circuit symbol**

The truth table can easily be modified for muxes that handle different numbers of inputs, by adding or removing control input columns. A minimal mux circuit can be designed by transferring the information in the truth table to a K-map, or by simply inspecting the truth table and writing an SOP equation directly. A minimal equation for the 4:1 mux follows (you are encouraged to verify this is a minimal equation):

$$Y = \overline{S_1} \cdot \overline{S_0} \cdot I_0 + \overline{S_1} \cdot S_0 \cdot I_1 + S_1 \cdot \overline{S_0} \cdot I_2 + S_1 \cdot S_0 \cdot I_3$$

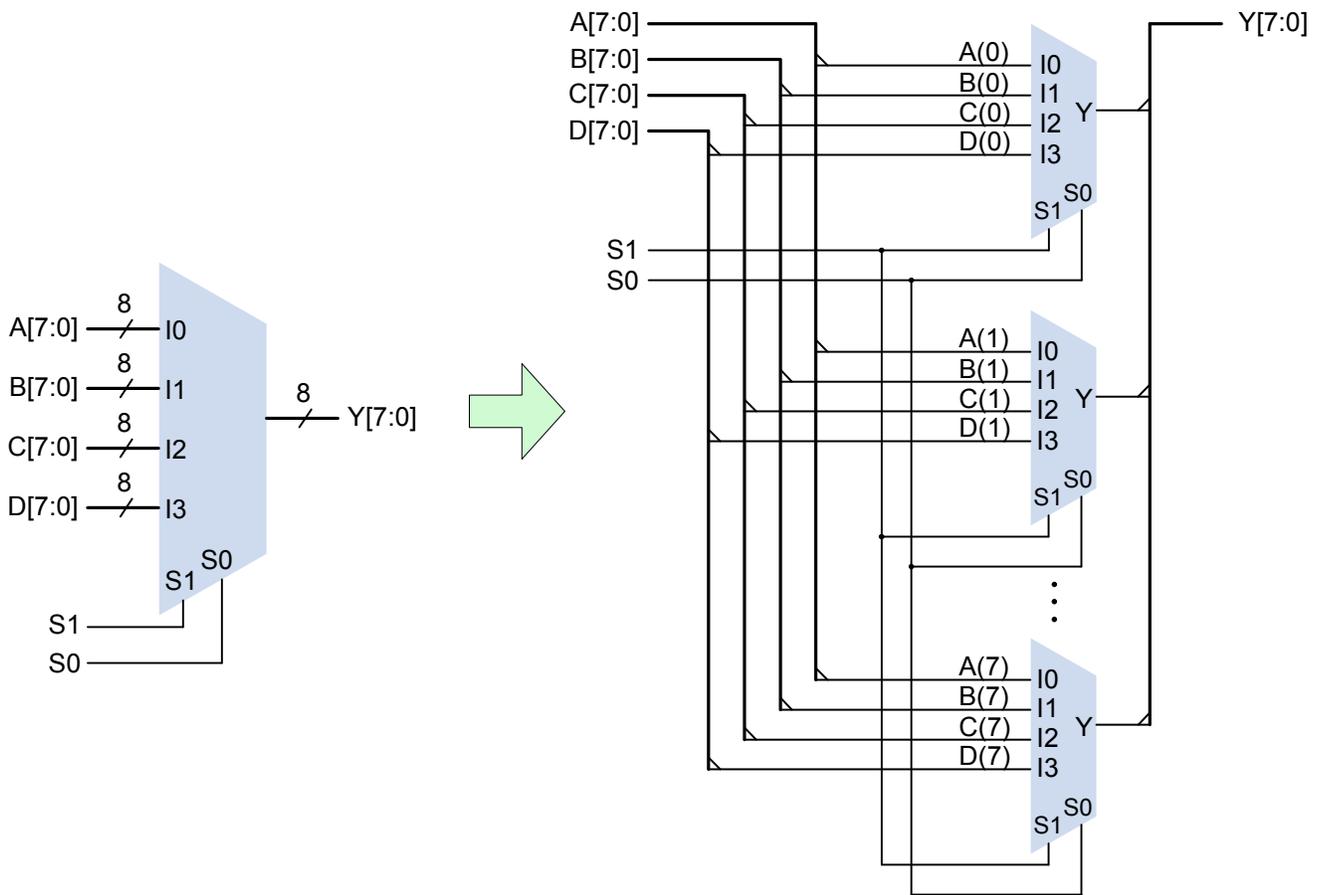
An N-input mux is a simple SOP circuit constructed from N AND gates each with  $\log_2 N + 1$  inputs, and a single output OR gate. The AND gates combine the  $\log_2 N$  select inputs with a data input such that only one AND gate output is asserted at any time, and the OR output stage simply combines the outputs of the AND gates (you will complete the sketch for a mux circuit in the exercises). As an example, to select input I2 in a 4 input mux, the two select lines are set to  $S_1 = 1$  and  $S_0 = 0$ , and the input AND stage would use a three input AND gate combining  $S_1$ , not ( $S_0$ ), and I2.

Often, mux circuits use an enable input in addition to the other inputs. The enable input functions as a sort of global on/off switch, driving the output to logic ‘0’ when it is de-asserted, and allowing normal mux operation when it is asserted.

Larger muxes can easily be constructed from smaller muxes. For example, an 8:1 mux can be created from two 4:1 muxes and one 2:1 mux if the outputs from the 4:1 muxes drive the data inputs of the 2:1 mux, and the most-significant select input drives the select input of the 2:1 mux.

Muxes are most often used in digital circuits to transfer data elements from a memory array to data processing circuit in a computer system. The memory address is presented on the mux select lines, and the contents of the addressed memory location are presented on the mux data inputs (this application of muxes will be presented in later labs that deal with memory systems). Since most data elements in computer systems are bytes or words consisting of 8, 16, or 32 bits, muxes used in computer circuits must switch 8, 16, 32 or more signals all at once. Muxes that can switch many

signals simultaneously are called “bus muxes”. A block diagram and schematic for a bus mux that can select one of four 8-bit data elements is shown below.



Since this most common application of multiplexors is beyond our current presentation, we will consider a less common, somewhat contrived application. Consider the K-map representation of a given logic function, where each K-map cell contains a '0', '1', or an entered variable expression. Each unique combination of K-map index variables selects a particular K-map cell (e.g., cell 6 of an 8 cell K-map is selected when A=1, B=1, C=0). Now consider a mux, where each unique combination of select inputs selects a particular data input to be passed to the output (e.g., I6 of an 8 input mux can be selected by setting the select inputs to A=1, B=1, C=0). It follows that if the input signals in a given logic function are connected to the select inputs of a mux, and those same input signals are used as K-map index variables, then each cell in the K-map corresponds to a particular mux data input. This suggests a mux can be used to implement a logic function by “connecting” the K-map cell contents to the data lines of the mux, and connecting the K-map index variables to the select lines of the mux. Mux data inputs are connected to: '0' (or ground) when the corresponding K-map cell contains a '0'; '1' (or Vdd) when the corresponding K-map cell contains a '1'; and if a K-map cell contains an entered variable expression, then a circuit implementing that expression is connected to the corresponding mux data input. Note that when a mux is used to implement a logic circuit directly from a truth table or K-map, logic minimization is not performed. This saves design time, but usually creates a less efficient circuit (however, a logic synthesizer would remove the inefficiencies before such a circuit was implemented in a programmable device).

A mux can easily be described in behavioral VHDL using a selected signal assignment statement as shown below. The statement functions by comparing the value of the *sel* input to the value shown in the *when* clause: the output variable *Y* gets assigned *I0*, *I1*, *I2*, or *I3* depending on whether *sel* = "00", "01", "10", or "11" (in a selected signal assignment statement, the "when others" clause is used for the final case for reasons that will be explained later). In addition to assigning values to individual signals or busses, the selected signal assignment statement can also be used to assign the result of arithmetic and/or logic operations to an output.

The example code on the left below is for a mux that switches logic signals, and the code on the right is for an 8-bit bus mux. Note the only difference in the code is in the port statement, where the data elements for the bus mux are declared to be vectors instead of signals. Note also that the assignment statement in the bus mux example assigns vector quantities just like signals. When you examine the code examples, particularly the bus mux, look again at the previous figure and consider the amount of effort required to create a bus mux schematic vs. the bus mux VHDL code.

```
entity mux_select is
  port ( I3, I2, I1, I0: in std_logic;
        sel : in std_logic_vector (1 downto 0);
        Y : out std_logic);
end mux_select;
```

```
architecture behavioral of mux_select is
  Begin
  with sel select
    Y <= I0 when "00";
        I1 when "01";
        I2 when "10";
        I3 when "others";
  end behavioral;
```

```
entity busmux_select is
  port ( I3, I2, I1, I0: in std_logic_vector (7 downto 0);
        sel : in std_logic_vector (1 downto 0);
        Y : out std_logic_vector (7 downto 0));
end busmux_select;
```

```
architecture behavioral of busmux_select is
  Begin
  with sel select
    Y <= I0 when "00";
        I1 when "01";
        I2 when "10";
        I3 when "others";
  end behavioral;
```

VHDL source code for implementing a more complex mux'ing circuit, such as one that might select any one of four logic function outputs to pass through to the output, is shown on the right. This example code uses a "conditional assignment" statement. Conditional assignment statements and selected signal assignments both allow more complex logic requirements to be succinctly described, and they can generally be used interchangeably. In most cases, a synthesizer will produce the same circuit regardless of whether a selected or conditional assignment statement is used. There are subtle differences between the statements, and these differences will be discussed later. For now, it is a matter of personal taste as to which one is used.

```
entity mux_cond is
  port ( A, B, C : in std_logic_vector (7 downto 0);
        Sel : in std_logic_vector (1 downto 0);
        Y : out std_logic_vector (7 downto 0));
end mux_cond;
```

```
architecture behavioral of mux_cond is
  Begin
  Y <= (A or not C) when (Sel = "00") else
        (A xor B) when (Sel = "01") else
        not A when (Sel = "10") else
        (B nand C);
  end behavioral;
```

#### VHDL code for a mux using a conditional assignment

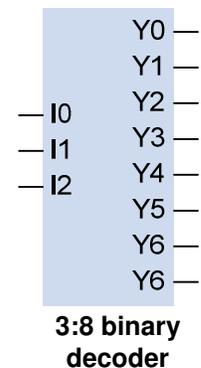
A conditional assignment statement uses the "when-else" language feature to describe compound logic statements. By following the example code shown, conditional assignments can be written to describe a wide variety of assignments.

## Decoders

Decoder circuits receive inputs in the form of an N-bit binary number and generate one or more outputs according to some requirement. Decoder inputs are typically viewed as a binary number representing some encoded quantity, and outputs typically drive some other circuit or device based on decoding that quantity. For example, a PS/2 keyboard decoder decodes the “scan codes” that are generated each time a given key is pressed (scan codes are unique binary numbers that are assigned to individual keys on a PS/2 keyboard). Most scan codes are simply sent to the host computer for parsing, but some perform specific functions. If the “caps lock” key is pressed, a signal is generated to illuminate an LED on the keyboard, and if “Cntrl-Alt-Del” is pressed, a signal is generated to interrupt PC operations.

Here, we will examine two different types of decoders – a simple binary decoder, and a seven-segment decoder that can drive a common numeric data display.

A binary decoder has N inputs and  $2^N$  outputs. It receives N inputs (often grouped as a binary number on a bus) and then asserts one and only one of its  $2^N$  outputs based on that input. If the N inputs are taken as an N-bit binary number, then only the output that corresponds to the input binary number is asserted. For example, if a binary 5 (or “101”) is input to a 3:8 decoder, then only the 5<sup>th</sup> output of the decoder will be asserted and all other outputs will be de-asserted. Practical decoder circuits are usually built as 2:4 decoders with 2 inputs and  $2^2$  (4) outputs, 3:8 decoders with 3 inputs and  $2^3$  (8) outputs, or 4:16 decoders with 4 inputs  $2^4$  (16) outputs. A decoder circuit requires one AND gate to drive each output, and each AND gate decodes a particular binary number. For example, a 3:8 decoder requires 8 AND gates, with the first AND gate having inputs  $A' \cdot B' \cdot C'$ , the second  $A' \cdot B' \cdot C$ , the third  $A' \cdot B \cdot C'$ , etc.



If a binary decoder larger than 4:16 is needed, it can be built from smaller decoders. Only decoders with an enable input can be used to construct larger decoder circuits. As with the mux, the enable input drives all outputs to ‘0’ when de-asserted, and allows normal decoder operation when asserted.

Decoders are most often used in more complex digital systems to access a particular memory location based on an “address” produced by a computing device. In this application, the address represents the coded data inputs, and the outputs are the particular memory element select signals. A typical memory circuit contains a decoder to select which memory element to write, the memory elements themselves, and a mux to select which element to read.

As with multiplexors, this most common application of decoders is beyond our current presentation, so instead we will consider a less common, somewhat contrived application. Consider the function of a decoder and the truth table, K-map, or minterm representation of a given function. Each row in a truth table, each cell in a K-map, or each minterm number in an equation represents a particular combination of inputs. Each output of a decoder is uniquely asserted for a particular combination of inputs. Thus, if the inputs to a given logic function are connected to the inputs of a decoder, and those same inputs are used as K-map input logic variables, then a direct one-to-one mapping is created between the K-map cells and the decoder outputs. It follows that any given function represented in a truth table or K-map can be directly implemented using a decoder, by simply by OR'ing the decoder outputs that correspond to a truth table row or K-map cell containing a “1” (decoder outputs that correspond to K-map cells that contain a zero are simply left unconnected). In such a circuit, any input combination with a ‘1’ in the corresponding truth table row or K-map cell will drive the output OR gate to a ‘1’, and any input combination with a ‘0’ in the corresponding K-map cell will allow the OR gate to

output a '0'. Note that when a decoder is used to implement a circuit directly from a truth table or K-map, no logic minimization is performed. Using a decoder in this fashion saves time, but usually results in a less efficient implementation (here again, a logic synthesizer would remove the inefficiencies before such a circuit was implemented in a programmable device).

A decoder can easily be described in behavioral VHDL using a selected signal assignment statement as shown below. In the example, both the inputs and outputs are grouped as busses so that a selected assignment statement can be used. In this example, the inputs can be individually referred to as I(1) and I(0), and the outputs as Y(0) through Y(3). The code can easily be modified to describe a decoder of any size.

```
entity decoder is
  port ( in:      in std_logic_vector (1 downto 0);
        Y:      out std_logic_vector (7 downto 0));
end decoder;

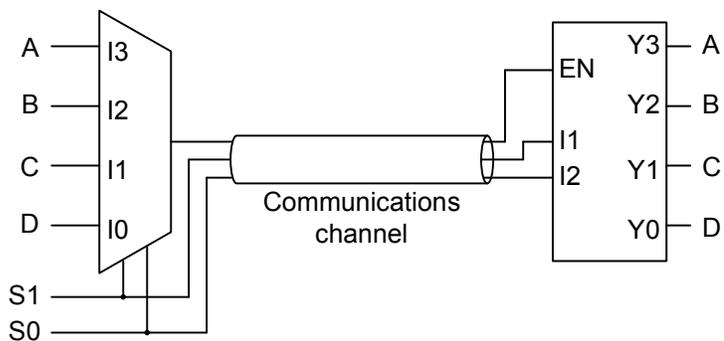
architecture behavioral of decoder is
  Begin
  with in select
  Y <=  "0001" when "00";
        "0010" when "01";
        "0100" when "10";
        "1000" when "others";
end behavioral;
```

**VHDL code for a 2:4 decoder**

De-multiplexor

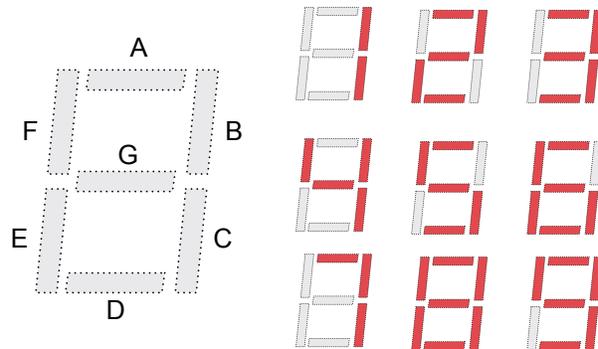
Our use of the word "multiplexor" has its origins in telecommunications, defining a system where one signal is used to transmit many different messages, either simultaneously or at different times. "Time-multiplexing" describes a system where different messages use the same physical signal, with different messages being sent at different times. Time multiplexing works if a given signal can carry more traffic than any one message needs. For example, if ten messages each require that 1Kbit of information be sent every second, and if a communication signal is available that can carry 10Kbits per second, then time-multiplexing can be used to provide ten 1Kbit time windows each second, one for each signal. A multiplexor can be used as a simple time multiplexor, if the select inputs are used to define the time window, and the data inputs are used as the data sources.

A decoder with an enable can be used as a de-multiplexor. Whereas a multiplexor selects one on N inputs to pass through to the output, a de-multiplexor takes a single input and routes it to one of N outputs. A multiplexor/de-multiplexor (or more simply, mux/de-mux) circuit can be used to transmit the state of N signals from one place to another using only  $\log_2 N + 1$  signals.  $\log_2 N$  signals are used to select the data input for the mux and to drive the decoder inputs, and the rate at which these signals change define the time-window length. The data-out of the mux drives the enable-in of the decoder, so that the same logic levels that appear on the mux inputs also appear on the corresponding decoder outputs, but only for the mux input/decoder output currently selected. In this way, the state of N signals can be sent from one place to another using only  $\log_2 N + 1$  signals, but only one signal at a time is valid.



Seven-Segment Displays and Decoders

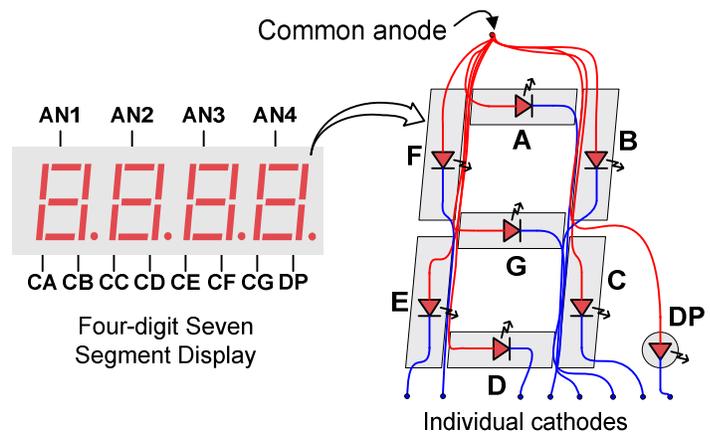
Seven-segment displays (7sd) are some of the most common electronic display devices in use. They can be used to display any decimal digit by illuminating particular segments and leaving other segments dark. 7sd devices are constructed from seven LEDs that have been arranged in a figure "8" pattern as shown in the figure below. These LEDs function identically to the individual LEDs – they emit light when a small current passes through them. The 7sd device can display a particular digit if certain LED segments are illuminated while others remain dark. As examples, if only segments b and c are illuminated, then the display will show a '1', and if segments a, b and c are illuminated then the display will show a '7'. To cause an illuminating current to flow through any given LED segment, a logic signal must be impressed across the segment LED. In a typical 7sd circuit, a current-limiting resistor is placed on the cathode lead, and a transistor is used on the anode lead to provide additional current (most signal pins on digital ICs – like the FPGA on the Digilent board – cannot provide enough current to light all the display segments, so a transistor is used to provide more current).



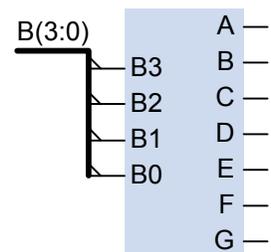
An un-illuminated seven-segment display, and nine illumination patterns corresponding to decimal digits

In order that all 10 decimal digits can be displayed, a 7sd device requires seven logic signals, one for each segment. By asserting particular combinations of these signals, all ten decimal digits can be displayed.

The Digilent board uses a common anode display, which means that all the anode connections for a given digit are tied together into a common circuit node as shown below. To illuminate a given segment in a given digit, a '1' must be applied to the digit's anode, and '0' must be applied to the segment's cathodes (NOTE: With Digilent boards, a '1' is applied to a digit's anode by applying a '0' to the circuit node that drives the transistor; thus, the anode signals AN3 – AN0 are "active low").



A seven-segment decoder (SSD) receives four signals that represent the four bits of a binary number, and produces seven output signals that can drive the seven segments in the seven-segment display. Thus, for example, if "0000" is input to the SSD, all outputs except "g" should be asserted (to cause a '0' to be displayed on the 7sd). And if "1000" is input to the SSD, then all outputs should be asserted (to cause an '8' to be displayed). Typically, the input signals are named B3-B0, and the output signals are given a letter to indicate which segment they must drive (A-F). As discussed above, each of the seven outputs could be thought of as a separate 4-input logic design problem, and optimal circuits for each output could easily be found using the techniques developed in previous labs. In lab project that accompanies this module, various methods will be used to optimize (or minimize) the system as a whole, considering all seven outputs at the same time.



A 7sd can easily be described in VHDL using a selected signal assignment statement. In fact, a selected assignment statement can be used to implement any truth table by listing the function inputs on the right of the "when" clause, and the associated outputs on the left. In the example shown below, the input and output variables are both vectors – the *ins* represents a 2-bit binary number, and the *outs* represent a 4-bit binary number. As discussed in the "muxes" section above, the output variable *outs* gets assigned the binary values shown in quotes when *ins* is equal to the value in the "when" clause. Thus, if *ins* is "01", then *outs* gets assigned "1010".

<pre>With ins select   outs &lt;= "0010" when "00";          "1010" when "01";          "1100" when "10";          "1110" when others;</pre>		<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th colspan="2" style="border: none;">INs</th> <th colspan="4" style="border: none;">OUTs</th> </tr> <tr> <th style="border: none;">A</th> <th style="border: none;">B</th> <th style="border: none;">F1</th> <th style="border: none;">F2</th> <th style="border: none;">F3</th> <th style="border: none;">F4</th> </tr> </thead> <tbody> <tr> <td style="border: none;">0</td> <td style="border: none;">0</td> <td style="border: none;">0</td> <td style="border: none;">0</td> <td style="border: none;">1</td> <td style="border: none;">0</td> </tr> <tr> <td style="border: none;">0</td> <td style="border: none;">1</td> <td style="border: none;">1</td> <td style="border: none;">0</td> <td style="border: none;">1</td> <td style="border: none;">0</td> </tr> <tr> <td style="border: none;">1</td> <td style="border: none;">0</td> <td style="border: none;">1</td> <td style="border: none;">1</td> <td style="border: none;">0</td> <td style="border: none;">0</td> </tr> <tr> <td style="border: none;">1</td> <td style="border: none;">0</td> </tr> </tbody> </table>	INs		OUTs				A	B	F1	F2	F3	F4	0	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1	0	0	1	1	1	1	1	0
INs		OUTs																																				
A	B	F1	F2	F3	F4																																	
0	0	0	0	1	0																																	
0	1	1	0	1	0																																	
1	0	1	1	0	0																																	
1	1	1	1	1	0																																	

VHDL code for a seven-segment decoder is partially supplied below. The four inputs (representing a binary number) have been grouped into a vector called BIN, and the seven segment outputs have been grouped into a vector called SEG\_OUT. Note the "when others" clause in the last line as is typical for any selected assignment statement. This catch-all "when others" clause is used to assign the value "0000001" to the seven segment decoder outputs whenever an unspecified input condition occurs. In this case, this clause can be used to assign an output value when the binary numbers 1010 through 1111 are present on the inputs.

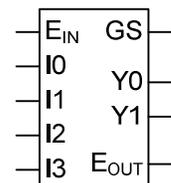
```
entity seven_seg_dec is
  port (bin: in STD_LOGIC_VECTOR (3 downto 0);
        segout : out STD_LOGIC_VECTOR (6 downto 0));
end seven_seg_dec;

architecture behavioral of seven_seg_dec is
begin
  with bin select
    segout <= "1111110" when "0000";
             "0110000" when "0001";
             .
             .
             "0000001" when others;
end behavioral;
```

**Example VHDL code for a Seven-Segment Decoder**

Priority Encoders

A priority encoder is, in a sense, the dual (or opposite) of the decoder circuit – it receives N inputs (where N is typically 4, 8 or 16), and asserts an output binary code of  $M = \log_2 N$  bits (so the M-bit binary code is typically 2, 3, or 4 bits). The M-bit binary code indicates which input was asserted (i.e., in a 4:2 binary encoder, binary code 00 would be output if the 0<sup>th</sup> input line was asserted, binary code 01 would be



**Priority Encoder**

output of the 1<sup>st</sup> input line was asserted, etc.). Since more than one input line to the encoder might be asserted at any given time, the priority encoder asserts an output code corresponding to the highest numbered input that is asserted (i.e., if both input line 0 and input line 2 were asserted in a 4:2 encoder, then binary code 10 would be output indicating that input line 2 is the highest line number – or highest priority input – currently asserted).

At first thought, a four input encoder circuit should require just two outputs. In such a circuit, asserting the 3<sup>rd</sup> input signal would cause a “11” output, asserting the 2<sup>nd</sup> input signal would output a “10”, asserting the 1<sup>st</sup> input signal would output a “01”, and asserting the 0<sup>th</sup> input would output “00”. But what if no inputs are asserted? Again, a “00” would be appropriate. To avoid creating an ambiguous “00” output, encoders typically use an “Enable In” ( $E_{IN}$ ) signal and an “Enable Output” ( $E_{OUT}$ ) signal.  $E_{IN}$  functions like other enable signals – when it is de-asserted, all outputs are driven to logic ‘0’, and when it is asserted, the encoder outputs can be driven by the inputs.  $E_{OUT}$  is asserted only when  $E_{IN}$  is asserted and no input signals are asserted. Thus,  $E_{OUT}$  can be used to distinguish between no inputs asserted and the 0<sup>th</sup> input asserted.

Larger encoders can be built from smaller encoder modules in much the same way that larger decoders can be built from smaller decoder modules. An encoder module that can be used as a building block for larger encoders must have one additional output called group-signal (GS). GS is asserted whenever  $E_{IN}$  is asserted along with any other input signal, and it is used to form the most significant bit of the encoded output data element.

Encoder circuits are typically used in digital systems when a binary number that corresponds to a given input must be generated. For example, individual “call attendant” signals arising from passengers seated on an airplane could be encoded into a seat number. Priority encoders are also used when certain input signals must be dealt with in a special manner. For example, if inputs from several sources can all arrive simultaneously, a priority encoder can indicate which signal should be dealt with first. Behavioral VHDL code for an encoder is shown below.

```
entity encoder is
  port (ein :          in std_logic;
        I :          in std_logic_vector (3 downto 0);
        eout, gs :    out std_logic;
        Y :          out std_logic_vector (1 downto 0));
end encoder;

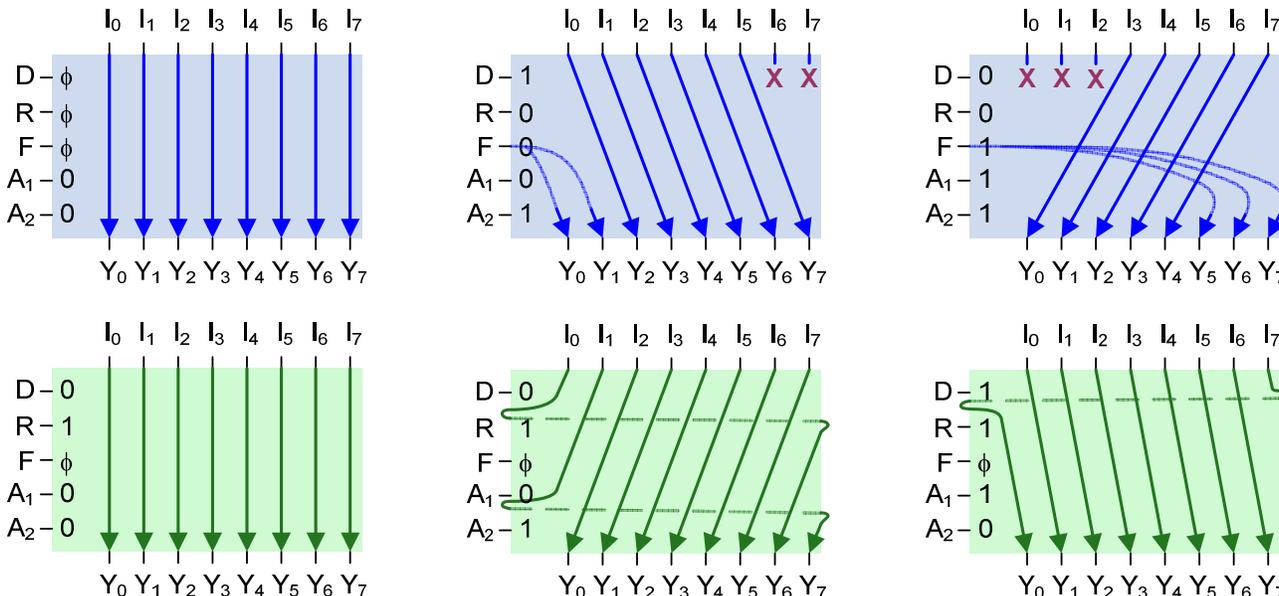
architecture Behavioral of encoder is
  Begin
    eout <= ein and not I(3) and not I(2) and not I(1) and not I(0);
    gs <= ein and (I(3) or I(2) or I(1) or I(0));
    Y(1) <= I(3) or I(2);
    Y(0) <= I(3) or I(1);
  end Behavioral;
```

#### VHDL code for a priority encoder

### Shifters

A shifter is a circuit that produces an N-bit output based on an N-bit data input and an M-bit control input, where the N output bits are place-shifted copies of the input bits, shifted some number of bits to the left or right as determined by the control inputs. As an example, the function of an 8-bit shifter capable of shifting one, two, or three bits to the right or left is illustrated in the top row of the figure

below. The control signals enable several different functions: two bits ( $A_1$  and  $A_0$ ) to determine how many bit positions to shift (0, 1, 2, or 3); a Fill signal ( $F$ ) determines whether bits vacated by shift operations receive a '1' or a '0'; a Rotate signal ( $R = '1'$  for rotate) determines whether shifted-out bits are discarded or recaptured in vacated bits; and a Direction signal ( $D = '1'$  for right) determines which direction the shift will take.



When bits are shifted left or right, some bits “fall off” one end of the shifter, and are simply discarded. New bits must then be shifted in from the opposite side. If no Fill input signal exists, then 0’s are shifted in (otherwise, the Fill input defines whether 1’s or 0’s are shifted in to vacated bits). Shifters that offer a Rotate function recapture shifted-out bits in vacated bits as shown in the lower row of the figure above.

Based on the shifter functions Shift, Rotate, Direction, Fill, and Number of Bits, many different shifter circuits could be designed to operate on any number of inputs. As an example of a simple shifter design, the truth table on the right shows input/output requirements for a four-bit shifter that can shift or rotate an input value left or right by one bit ( $R=0$  for shift,  $R=1$  for rotate,  $D=0$  for left,  $D=1$  for right). Note the truth table uses entered variables to compress the number of rows that would otherwise be required. A minimal circuit can be found from this truth table using pencil-and-paper methods or a computer-based minimization program.

EN	R	D	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	$\phi$	$\phi$	0	0	0	0
1	0	0	$I_2$	$I_1$	$I_0$	0
1	0	1	0	$I_3$	$I_2$	$I_1$
1	1	0	$I_2$	$I_1$	$I_0$	$I_3$
1	1	1	$I_0$	$I_3$	$I_2$	$I_1$

Truth table for 4-bit shifter with shift/rotate left/right functions

Shifters are most often found in circuits that work with groups of signals that together represent binary numbers, where they are used to move data bits to new locations on a data bus (i.e., the data bit in position 2 could be moved to position 7 by right shifting five times), or to perform simple multiplication and division operations (exactly why a bit might want to be moved from one location to another on a data bus is left for a later module). A shifter circuit can multiply a number by 2, 4, or 8 simply by shifting the number right by 1, 2, or 3 bits (and similarly, a shifter can divide a number by 2, 4, or 8 by shifting the number left by 1, 2, or 3 bits).

A behavioral VHDL design of a simple 8-bit shifter that can shift or rotate left or right by one bit is shown below. A conditional assignment statement is used in this example as the only statement in the architecture body. The “when-else” clause evaluates the state of enable (en), rotate (r), and direction (d) to distinguish between the possible output vector signal assignments. The first assignment in the conditional assignment statement assigns all zero’s to the dout bus when en='0'. The remaining four assignments make use of the concatenation operator (&) to assign shifted or rotated versions of the input data bus to the output bus, depending on the states of r and d.

```
entity my_shift is
  port ( din:      in std_logic_vector (7 downto 0);
        r, d, en: in std_logic;
        dout:     out std_logic_vector (7 downto 0));
end my_shift;

architecture my_shift_arch of my_shift is
begin
  dout <= "00000000" when en = '0' else
    din(6 downto 0) & din(7) when (r = '1' and d = '0') else
    din(0) & din(7 downto 1) when (r = '1' and d = '1') else
    din(6 downto 0) & '0' when (r = '0' and d = '0') else
    '0' & din(7 downto 1);
end my_shift_arch;
```