CHAPTER 6

Modeling Combinational Logic Circuits

Chapter 6 Contents

Modeling Combinational Logic

This chapter demonstrates the different ways in which purely combinational logic may be modeled. It does not include tri-state logic which is covered separately in Chapter 10. The types of combinational logic circuit commonly used in digital design and covered in this chapter are listed in Table 6.1.

logical/arithmetic equations
logical structure control
multiplexers
encoders
priority encoders
decoders
comparators
ALUs

Table 6.1 Functional types of combinational logic circuit

These more standard functional types of circuit are used in both control path and datapath structures. Typically each circuit type can be modeled in different ways using **if**, **case**, and **for** statements etc. Additionally for VHDL only, the concurrent selected and conditional signal assignments can also be used. The selected signal assignment is synonymous with the **if** statement and the conditional signal assignment is synonymous with the **case** statement, but reside outside a **process**. This means they are always active and so may increase the time it takes to simulate a model when compared to using a **process** with a sensitivity list. Also, the VHDL **for-loop** may include one or more **next** or **exit** statements. The **next** statement causes a jump to the next loop iteration, while the **exit** statement causes an exit from the **for-loop** altogether. There is no equivalent to the **next** or **exit** statements in Verilog. The VHDL **while-loop** statement, and the Verilog **forever** and **while-loop** statements, are not often used to model combinational logic; their loop range must have a static value at synthesis compile time so that a predetermined amount of logic can be synthesized. They are not supported by the synthesis tools from VeriBest Incorporated.

Note, that when modeling combinational logic, the sensitivity list of a **process** statement (VHDL) or the event list of an **always** statement (Verilog), must contain all inputs used in the particular statement. If it does not, the model will still synthesize correctly, but may not simulate correctly. This is because **process/always** statements are concurrent and will not be triggered into being executed when the omitted signals change, and means the output signals will not be updated.

Because the examples in this chapter are relatively small for demonstration purposes, VHDL models use mostly signal assignments and relatively few variable assignments. VHDL models with more code in a **process**, typically use more variable assignments. Variables and constants are used in the computation of signal values, see Chapter 4. A number of VHDL model versions in this chapter use **for-loop** statements. It is better to use only variable assignments, and not signal assignments in **for-loop** statements. This is not mandatory as identical circuits will be synthesized, but it will simulate faster for reasons given in Chapter 4.

The logic synthesized from the majority of the models in this chapter have little or no inherent logical structure. This means area, timing and power characteristics are often considerably improved when the synthesized circuit is optimized. Logic optimization breaks down the logical structure of a circuit and creates a new one in the process of attempting to improve any area, timing or power requirements that have been specified.

The following sections describe each of the circuit functions listed in Table 6.1. Shifters, multipliers and dividers can also be modeled using synchronous logic and are included in Chapter 9.

Logical/Arithmetic Equations

Both logical and arithmetic equations may be modeled using the logical and arithmetic operators in the expressions of continuous data flow assignments, see Example 6.1.

Example 6.1 Equations modeled using continuous assignments

Logical and arithmetic equations are modeled using continuous data flow assignments, incorporating both logical and arithmetic operators. Both concurrent (outside **process/always**) and sequential (inside **process/always**) assignments are shown.

VHDL signals S1 and S2 and variables V1 and V2, have identical names in the Verilog model for comparison, but are all variables of type **reg** in the Verilog model. The Verilog variables V1 and V2 are not local to the sequential block as the variables are in the VHDL model. Although Verilog supports locally defined data types of type **reg**, this is not generally supported by synthesis tools. The VHDL output Y1 is defined from a concurrent continuous assignment and so the Verilog equivalent must be of type **wire**. The data type of Y1 could have been explicitly defined as a **wire**, for example, "**wire** Y1;", however, this is not necessary as type **wire** is implied by default as defined by the Verilog language.

Mathemematical equations modeled using continous assignments

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity SIMPLE_DATA_FLOW is
   port ( A1: in  unsigned(7 downto 0);
          A2, B2, C2, D2, E2: in  unsigned(1 downto 0);
          Y1: out std_logic;
          Y2: out unsigned(3 downto 0);
end entity SIMPLE_DATA_FLOW;

architecture DATA_FLOW of SIMPLE_DATA_FLOW is
   signal S1, S2: std_logic;
begin
   process (A1(3 downto 0))
      variable V: unsigned(1 downto 0);
   begin
      V := ((A1(0) nor A1(1)), (A1(2) nor A1(3)));
      S1 <= V(0) nand V(1);
   end process;

   process (A1(7 downto 4))
      variable V: unsigned(1 downto 0);
   begin
      V := (A1(4) nor A1(5)) & (A1(6) nor A1(7));
      S2 <= V(0) nand V(1);
   end process;

   Y1 <= S1 nor S2;

   process (A2, B2, C2, D2, E2)
   begin
      Y2 <= A2 + (B2 - C2) + (D2 * E2);
   end process;
end architecture DATA_FLOW;
```

Verilog

```
module SIMPLE_DATA_FLOW
   (A1, A2, B2, C2, D2, E2, Y1, Y2);
      input    (7:0) A1;
      input    (1:0) A2, B2, C2, D2, E2;
      output         Y1;
      output   (3:0) Y2;

      reg       S1, S2;
      reg (1:0) V1,V2;
      reg (3:0) Y2;

   always @(A1(3:0))
      begin
         V1 = {! (A1(0) | A1(1)), ! (A1(2) | A1(3))};
         S1 = ! (V1(0) & V1(1));
      end

   always @(A1(7:4))
      begin
         V2 = {! (A1(4) | A1(5)), ! (A1(6) | A1(7))};
         S2 = ! (V2(0) & V2(1));
      end

   assign Y1 = ! (S1 | S2);

   always @(A2 or B2 or C2 or D2 or E2)
      Y2 = A2 + (B2 - C2) + (D2 * E2);

endmodule
```

Full sensitivity list.

Sequential assignments.

Full sensitivity list.

Sequential procedural assignments.

Full sensitivity list.

Concurrent (continuous) assignments.

Full sensitivity list.

Full sensitivity list.

The multiplier operation is not an efficient way of inferring a combinational multiplier circuit. See Chapter 9.

Logical Structure Control

Parentheses can be used for course grain control of synthesized logic structure. Logic optimization can still be used to break down all, or most, of a circuit's logic structure and restructure it in the process of attempting to meet specific constraints. However, the use of parentheses in the model's expressions can make the optimizer's job far easier and less cpu intensive, but more importantly, the optimizer may not be able to achieve such good results that careful choice of parentheses can bring, see Example 6.2.

Example 6.2 Parentheses used to control logical structure

Parentheses are used to control the structure of inferred adders. The model contains two assignments, each implying the synthesis of three adders. The first assignment to Y1 does not use parentheses and so defaults to a left to right priority; this results in a worst case timing delay which passes through three adders. The second assignment to Y2 does use parentheses for a more course grain structural control and infers a circuit structure whose longest timing delay this time passes through only two adders instead of three.

Parentheses used to control logical structure

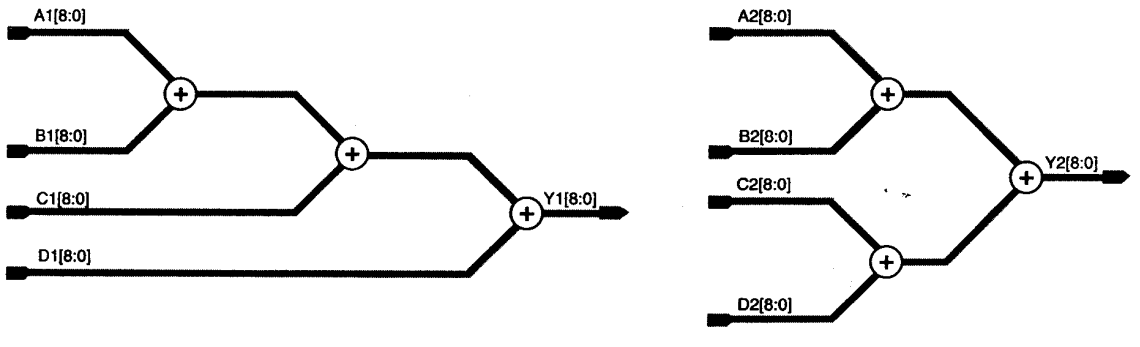
VHDL

```
library IEEE;
use IEEE.STD_STDLOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity COMB_LOGIC_STRUCT is
   port (A1,B1,C1,D1,A2,B2,C2,D2: in unsigned(8 downto 0);
         Y1, Y2: out unsigned(10 downto 0));
end entity COMB_LOGIC_STRUCT;

architecture LOGIC of COMB_LOGIC_STRUCT is
begin

   process (A1, B1, C1, D1, A2, B2, C2, D2)
   begin

      Y1 <= A1 + B1 + C1 + D1;

      Y2 <= (A2 + B2) + (C2 + D2);

   end process;

end architecture LOGIC;
```

Verilog

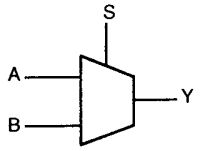
```
module COMB_LOGIC_STRUCT
   (A1, B1, C1, D1, A2, B2, C2, D2, Y1, Y2);
      input (8:0)  A1, B1, C1, D1, A2, B2, C2, D2;
      output (10:0) Y1, Y2;

      reg (10:0) Y1, Y2;


   always @(A1 or B1 or C1 or D1 or A2 or B2 or C2 or D2)
      begin

         Y1 <= A1 + B1 + C1 + D1;

         Y2 <= (A2 + B2) + (C2 + D2);

      end

endmodule
```

Synthesized Circuit Structure



Multiplexers

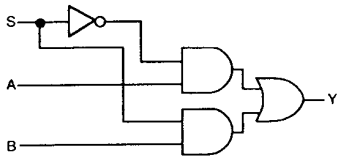
A multiplexer selectively passes the value of one, of two or more input signals, to the output. One or more control signals control which input signal's value is passed to the output, see Figure 6.1. Each input signal, and the output signal, may represent single bit or multiple bit busses. The select inputs are normally binary encoded such that *n* select inputs can select from one of up to 2^n inputs.



a) Circuit symbol

S	A	B	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

b) Truth table



c) Logic implementation

Figure 6.1 The Multiplexer

RTL level synthesis tools are not particularly good at identifying multiplexer type functions and mapping them directly onto multiplexer macro cells in a given technology library. If this is desired a multiplexer macro cell should be explicitly instantiated in the HDL model. However, a multiplexer circuit is often better implemented in cell primitives as they can be optimized with their surrounding logic and often produce a more optimal overall circuit implementation. Example 6.3 shows three ways of modeling a 2-1 multiplexer. Example 6.4 shows a 4-1 multiplexer modeled in several different ways and Example 6.5 shows a 2-bit wide 8-1 multiplexer.

Example 6.3 One-bit wide 2-1 multiplexer

The model of the one-bit wide 2-1 multiplexer described above is shown modeled using the **if** statement in its most simplest form. Multiplexer output Y1 is derived concurrently via a selected signal assignment in VHDL and a conditional continuous assignment in Verilog. The second and third multiplexer outputs, Y2 and Y3, are derived from an **if** statement. The first **if** statement defines a default output value for Y2 in an assignment immediately before the **if** statement, while the second **if** statement uses the more normal method of using an **else** clause.

One bit wide 2-1 multiplexer

VHDL

```
library IEEE;
use IEEE.STD_Logic_1164.all;
entity MUX_2_1 is
   port (Sel1,A1,B1, Sel2,A2,B2, Sel3,A3,B3: in std_logic;
         Y1, Y2, Y3: out std_logic);
end entity MUX_2_1;

architecture COND_DATA_FLOW of MUX_2_1 is
begin
   Y1 <= A1 when Sel1 = '1' else
         B1;
   TWO_2_1_MUXES:
   process (Sel2, A2, B2, Sel3, A3, B3)
   begin
      Y2 <= B2;
      if (Sel2 = '1') then
         Y2 <= A2;
      end if;
      if (Sel3 = '1') then
         Y3 <= A3;
      else
         Y3 <= B3;
      end if;
   end process TWO_2_1_MUXES;
end architecture COND_DATA_FLOW;
```

Selected signal assignment.

Default defined before the **if** so no **else** clause.

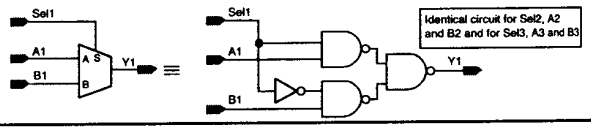
Most common and easiest to read.

Verilog

```
module MUX_2_1 (Sel1,A1,B1,Y1, Sel2,A2,B2,Y2, Sel3,A3,B3,Y3);
      input   Sel1,A1,B1, Sel2,A2,B2, Sel3,A3,B3;
      output  Y1,Y2,Y3;
      reg Y2,Y3;

   wire Y1 = Sel1 ? A1 : B1;

   always @(Sel2 or A2 or B2 or Sel3 or A3 or B3)
      begin
         Y2 = B2;
         if (Sel2)
            Y2 = A2;
         if (Sel3)
            Y3 = A3;
         else
            Y3 = B3;
      end
endmodule
```

Conditional continuous and wire assignment.

Synthesized Circuit



Example 6.4 Modeling styles of a 4-1 multiplexer

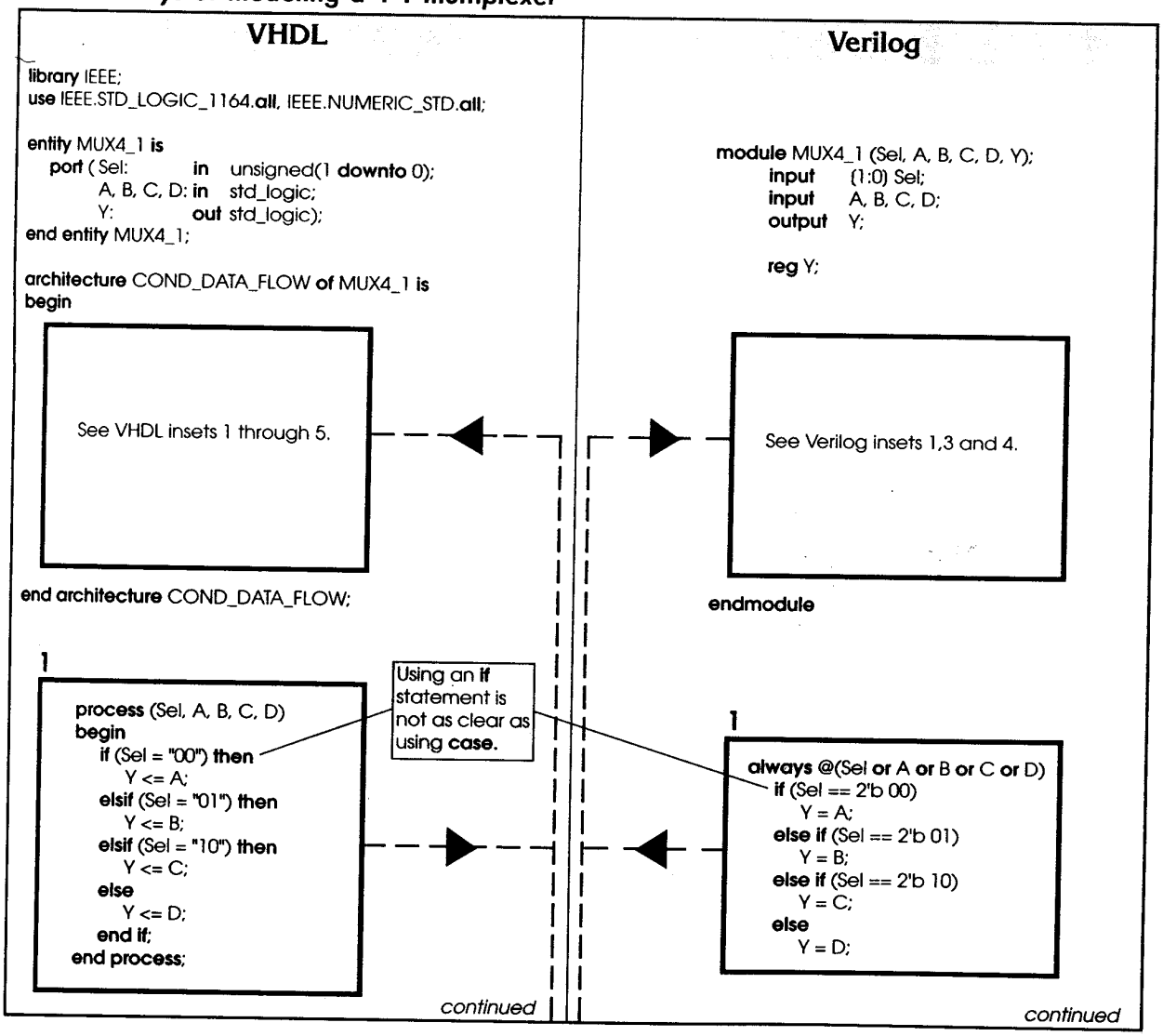
Five ways of modeling a 4-1 multiplexer in VHDL, and three ways of modeling it in Verilog are indicated. They are:

1. one **if** statement with multiple **elsif/else if** clauses,
2. a conditional signal assignment (VHDL),
3. nested **if** statements,
4. **case** statement,
5. uses a selected signal assignment (VHDL).

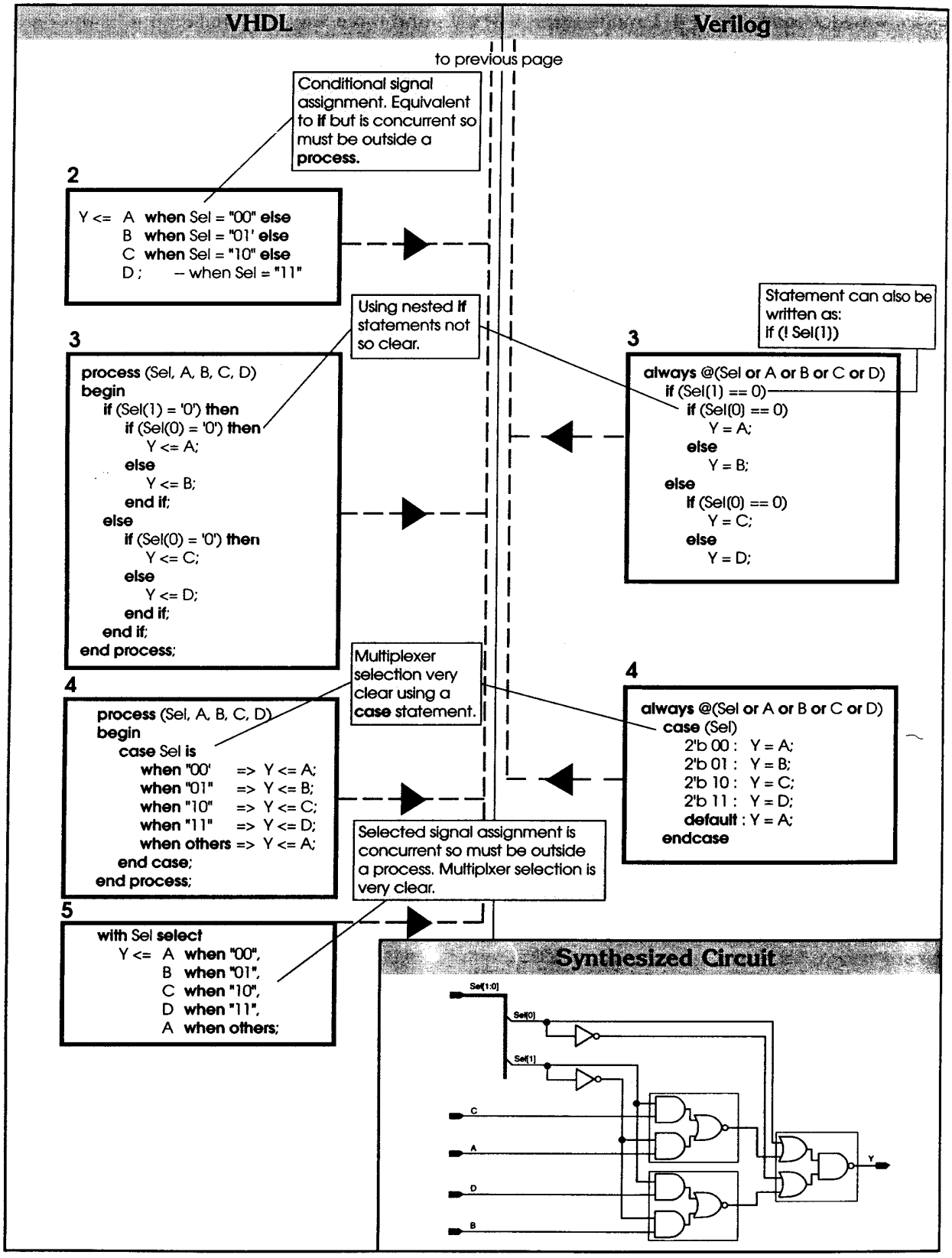
All models synthesize to the same circuit as shown.

There is no incorrect modeling method, however using the **case** statement requires less code and is easier to read when compared with the **if** statement. This becomes more distinct with increasing inputs per output; see also Example 6.5. The two VHDL only models, 2 and 5, use concurrent signal assignments so reside outside a process. This means they are always active and so will usually take longer to simulate.

Different ways of modeling a 4-1 multiplexer



Different ways of modeling a 4-1 multiplexer



Example 6.5 Two-bit wide 8-1 multiplexer using case

A 2-bit wide 8-1 multiplexer is modeled to the truth table in Table 6.1. Models use the **case** statement, and additionally for VHDL only, selected signal assignment. The **if** statement becomes cumbersome for the wider inputs. It is different from the previous example in that a VHDL integer data type is used for the select input Sel, and the Verilog **case** selector values are specified in integer form, that is, 4 instead of 3'b 0100.

Sel	A7	A6	A5	A4	A3	A2	A1	A0	Y
000	XX	XX	XX	XX	XX	XX	XX	DD	DD
001	XX	XX	XX	XX	XX	XX	DD	XX	DD
010	XX	XX	XX	XX	XX	DD	XX	XX	DD
011	XX	XX	XX	XX	DD	XX	XX	XX	DD
100	XX	XX	XX	DD	XX	XX	XX	XX	DD
101	XX	XX	DD	XX	XX	XX	XX	XX	DD
110	XX	DD	XX	XX	XX	XX	XX	XX	DD
111	DD	XX	XX	XX	XX	XX	XX	XX	DD

XX = two bit don't care DD = two bit data

Table 6.1 Truth table for a two bit wide 8-1 multiplexer

Two bit wide 8-1 multiplexer

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity MUX2X8_1_CASE is
   port ( Sel: in  integer range 0 to 7;
          A0,A1,A2,A3,A4,A5,A6,A7: in  unsigned(1 downto 0);
          Y: out unsigned(1 downto 0));
end entity MUX2X8_1_CASE;
architecture COND_DATA_FLOW of MUX2X8_1_CASE is
begin
   process (Sel, A0, A1, A2, A3, A4, A5, A6, A7)
   begin
      case Sel is
         when 0  => Y <= A0;
         when 1  => Y <= A1;
         when 2  => Y <= A2;
         when 3  => Y <= A3;
         when 4  => Y <= A4;
         when 5  => Y <= A5;
         when 6  => Y <= A6;
         when 7  => Y <= A7;
      end case;
   end process;

end architecture COND_DATA_FLOW;
```

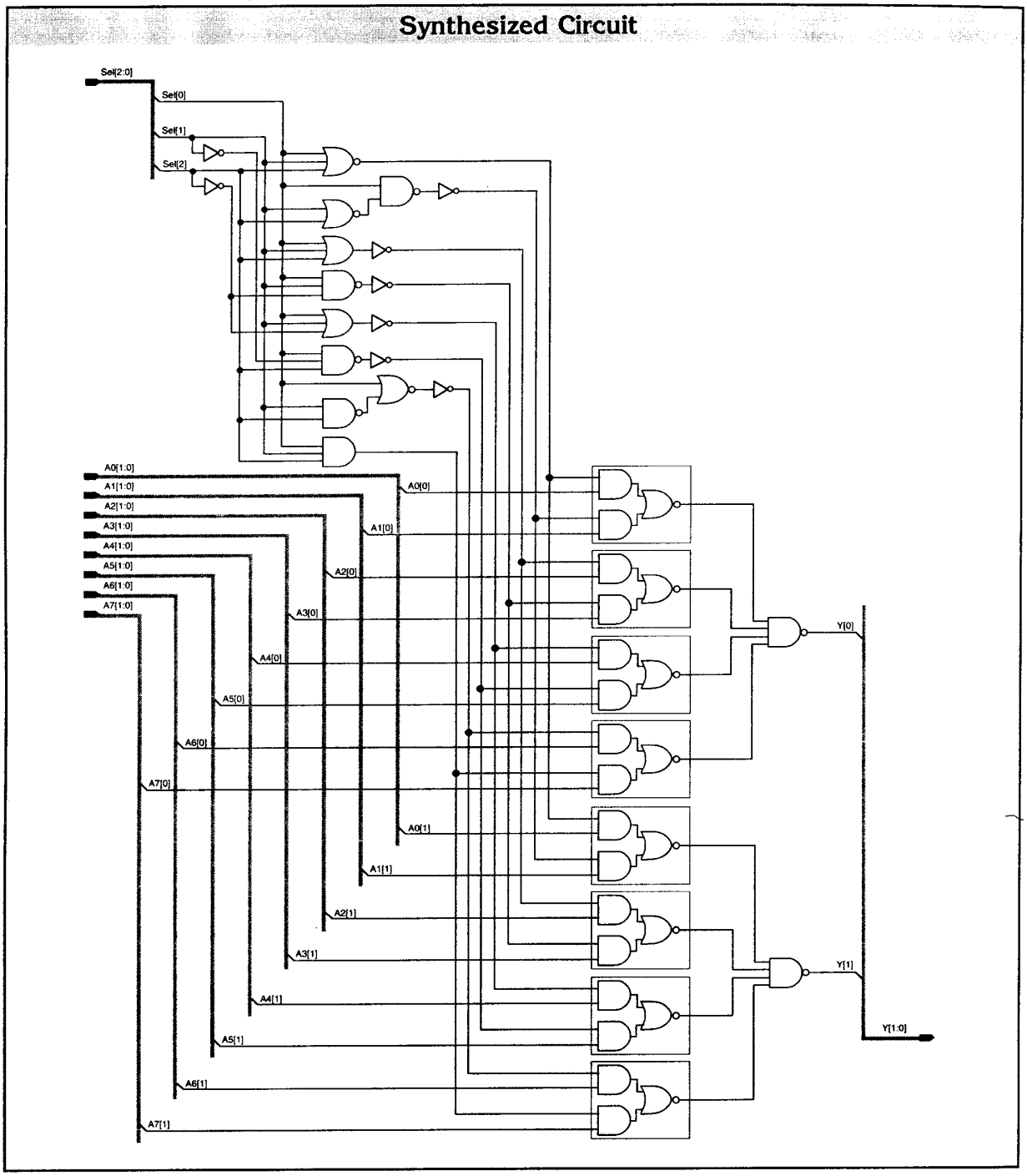
Verilog

```
module MUX2X8_1_CASE
   (Sel, A0, A1, A2, A3, A4, A5, A6, A7, Y);
      input  [2:0] Sel;
      input  [1:0] A0, A1, A2, A3, A4, A5, A6, A7;
      output [1:0] Y;

      reg [1:0] Y;

   always @(Sel or A0 or A1 or A2 or
            A3 or A4 or A5 or A6 or A7)
      case (Sel)
         0 : Y = A0;
         1 : Y = A1;
         2 : Y = A2;
         3 : Y = A3;
         4 : Y = A4;
         5 : Y = A5;
         6 : Y = A6;
         7 : Y = A7;
         default : Y = A0;
      endcase

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity MUX2X8_1_SSA is
   port ( Sel: in  integer range 0 to 7;
          A0,A1,A2,A3,A4,A5,A6,A7: in  unsigned(1 downto 0);
          Y: out unsigned(1 downto 0));
end entity MUX2X8_1_SSA;
architecture COND_DATA_FLOW of MUX2X8_1_SSA is
begin
   with Sel select
      Y <=  A0 when 0,
            A1 when 1,
            A2 when 2,
            A3 when 3,
            A4 when 4,
            A5 when 5,
            A6 when 6,
            A7 when 7;

end architecture COND_DATA_FLOW;
```

Two-bit wide 8-1 multiplexer



Encoders

Discrete quantities of digital information, data, are often represented in a coded form; binary being the most popular. Encoders are used to encode discrete data into a coded form and decoders are used to convert it back into its original undecoded form. An encoder that has 2^n (or less) input lines encodes input data to provide n encoded output lines. The truth table for an 8-3 binary encoder (8 inputs and 3 outputs) is shown in Table 6.2. It is assumed that only one input has a value of 1 at any given time, otherwise the output has some undefined value and the circuit is meaningless.

inputs								outputs		
A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Table 6.2 Truth table for an 8-3 binary encoder

The truth table can be modeled using the **if**, **case** or **for** statements.

Models using a **case** statement are clearer than those using an **if** statement. The **for** loop is better for modeling a larger or more generic *m-n* bit encoder. All models of such a circuit must use a default "don't care" value to minimize the synthesized circuit as only 8 of the 256 (2^8) input conditions need to be specified. The synthesis tool, if capable, replaces "don't care" values with logic 0 or 1 values as necessary in order to minimize the circuit's logic. This means VHDL integer data type cannot be used for the case selector in a **case** statement. However, whatever data type is used, for example, unsigned, it can always be converted from an integer data type before the **case** statement and back again after, although this can be cumbersome.

Example 6.6 shows models of the 8-3 binary encoder described above, using either the **if**, **case** or **for** statement.

Example 6.6 An 8-3 binary encoder

An 8-3 encoder is modeled according to the truth table of Table 6.2 using the **if**, **case** or **for** statement, and additionally for VHDL, conditional and selected signal assignments.

All models use a default assigned output value to avoid having to explicitly define all $2^8 - 8 = 248$ input conditions that should not occur under normal operating conditions. The default assignment is a "don't care" value to minimize synthesized logic. If all 248 input conditions that are not explicitly defined default to binary 000, more logic would be synthesized than is necessary.

8-3 encoder modeled from the truth table

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity ENCODE_8_3_IF_ELSE is
   port ( A:   in  unsigned(7 downto 0);
          Y:   out unsigned(2 downto 0));
end entity ENCODE_8_3_IF_ELSE;

architecture COND_DATA_FLOW of ENCODE_8_3_IF_ELSE is
begin
   process (A)
   begin
      if (A = "00000001") then Y <= "000";
      elsif (A = "00000010") then Y <= "001";
      elsif (A = "00000100") then Y <= "010";
      elsif (A = "00001000") then Y <= "011";
      elsif (A = "00010000") then Y <= "100";
      elsif (A = "00100000") then Y <= "101";
      elsif (A = "01000000") then Y <= "110";
      elsif (A = "10000000") then Y <= "111";
      else Y <= "XXX";
      end if;
   end process;
end architecture COND_DATA_FLOW;
```

Verilog

```
module ENCODER_8_3_IF_ELSE (A, Y);
      input    (7:0) A;
      output   (2:0) Y;

      reg (2:0) Y;


   always @(A)
      begin
         if (A == 8'b 00000001) Y = 0;
         else if (A == 8'b 00000010) Y = 1;
         else if (A == 8'b 00000100) Y = 2;
         else if (A == 8'b 00001000) Y = 3;
         else if (A == 8'b 00010000) Y = 4;
         else if (A == 8'b 00100000) Y = 5;
         else if (A == 8'b 01000000) Y = 6;
         else if (A == 8'b 10000000) Y = 7;
         else Y = 3'b X;
      end

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity ENCODE_8_3_CSA is
   port ( A: in  unsigned(7 downto 0);
          Y: out unsigned(2 downto 0));
end entity ENCODE_8_3_CSA;

architecture LOGIC of ENCODE_8_3_CSA is
begin
   Y <=  "000" when A = "00000001" else
         "001" when A = "00000010" else
         "010" when A = "00000100" else
         "011" when A = "00001000" else
         "100" when A = "00010000" else
         "101" when A = "00100000" else
         "110" when A = "01000000" else
         "111" when A = "10000000" else
         "XXX";
end architecture LOGIC;
```

Conditional signal assignment which is the concurrent equivalent of if statement.

8-3 encoder modeled from the truth table (continued)

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity ENCODE_8_3_CASE is
   port ( A: in   unsigned(7 downto 0);
          Y: out  unsigned(2 downto 0));
end entity ENCODE_8_3_CASE;

architecture LOGIC of ENCODE_8_3_CASE is
begin
   process (A)
   begin
      case A is
         when "00000001" => Y <= "000";
         when "00000010" => Y <= "001";
         when "00000100" => Y <= "010";
         when "00001000" => Y <= "011";
         when "00010000" => Y <= "100";
         when "00100000" => Y <= "101";
         when "01000000" => Y <= "110";
         when "10000000" => Y <= "111";
         when others     => Y <= "XXX";
      end case;
   end process;
end architecture LOGIC;
```

case statement is very clear.

Verilog

```
module ENCODE_8_3_CASE (A, Y);
      input   (7:0) A;
      output  (2:0) Y;
      reg (2:0) Y;

   always @(A)
      begin
         casex (A)
            8'b 00000001 : Y = 0;
            8'b 00000010 : Y = 1;
            8'b 00000100 : Y = 2;
            8'b 00001000 : Y = 3;
            8'b 00010000 : Y = 4;
            8'b 00100000 : Y = 5;
            8'b 01000000 : Y = 6;
            8'b 10000000 : Y = 7;
            default :      Y = 3'b X;
         endcase
      end

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity ENCODE_8_3_SSA is
   port ( A: in   unsigned(7 downto 0);
          Y: out  unsigned(2 downto 0));
end entity ENCODE_8_3_SSA;

architecture LOGIC of ENCODE_8_3_SSA is
begin
   with A select
      Y <=  "000" when "00000001",
            "001" when "00000010",
            "010" when "00000100",
            "011" when "00001000",
            "100" when "00010000",
            "101" when "00100000",
            "110" when "01000000",
            "111" when "10000000",
            "XXX" when others;
end architecture LOGIC;
```

Selected signal assignment is also very clear. It is the concurrent equivalent of **case** statement.

8-3 encoder modeled from the truth table (continued)

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity ENCODE_8_3_FOR is
   port ( A: in  unsigned(7 downto 0);
          Y: out unsigned(2 downto 0));
end entity ENCODE_8_3_FOR;

architecture LOGIC of ENCODE_8_3_FOR is
begin

   process (A)
      variable N: integer range 0 to 7;
      variable Test: unsigned(7 downto 0);
   begin
      Test := "00000001";
      Y <= "XXX";
      for N in 0 to 7 loop
         if (A = Test) then
            Y <= To_unsigned(N, 3);
            exit;
         end if;
         Test := shift_left(Test, 1);
      end loop;
   end process;
end architecture LOGIC;
```

Integer range specified to avoid 32 bit logic being synthesized.

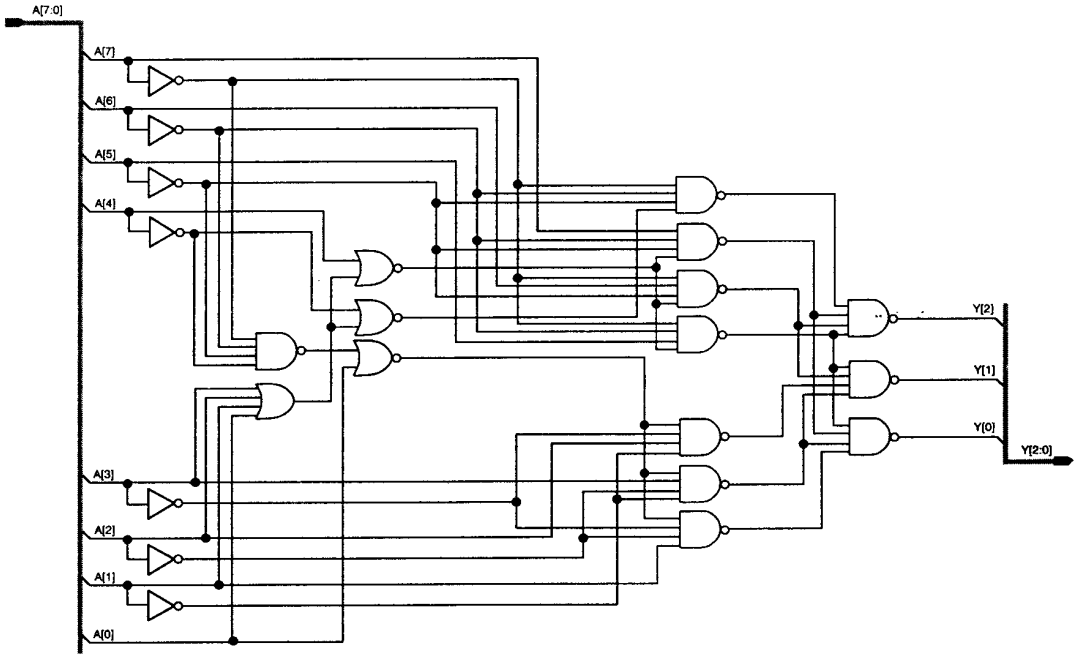
for loop statement. Code stays condensed when bit width increase.

loop integer converted to type unsigned for output.

Verilog

```
module ENCODE_8_3_FOR (A, Y);
      input   (7:0) A;
      output  (2:0) Y;
      reg (2:0) Y;

      reg (7:0) Test;
      integer (2:0) N;

   always @(A)
      begin
         Test = 8'b 00000001;
         Y = 3'b X;
         for (N = 0; N < 8; N = N + 1)
            begin
               if (A == Test)
                  Y = N;
               Test = Test << 1;
            end
      end

endmodule
```

Synthesized Circuit



Priority Encoders

The operation of the priority encoder is such that if two or more single bit inputs are at a logic 1, then the input with the highest priority will take precedence, and its particular coded value will be output. Models of an 8-3 binary priority encoder are included in Example 6.7.

Example 6.7 An 8-3 binary priority encoder

An 8-3 priority encoder is modeled in several different ways to the truth table shown in Table 6.3. The most significant bit, A7, has the highest priority. The output signal Valid indicates that at least one input bit is at logic 1 and signifies the 3-bit output Y is valid.

Different models use **if**, **case** and **for** statements. They all use "don't care" default value for the 3-bit output Y for the condition when all 8 inputs are at logic 0. This gives the synthesis tool the potential to reduce the logic, although it makes little or no difference in this particular model.

inputs								outputs			
A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0	Valid
0	0	0	0	0	0	0	0	X	X	X	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	X	0	0	1	1
0	0	0	0	0	1	X	X	0	1	0	1
0	0	0	0	1	X	X	X	0	1	1	1
0	0	0	1	X	X	X	X	1	0	0	1
0	0	1	X	X	X	X	X	1	0	1	1
0	1	X	X	X	X	X	X	1	1	0	1
1	X	X	X	X	X	X	X	1	1	1	1

X = don't care

Table 6.3 Truth table of an 8-3 binary priority encoder

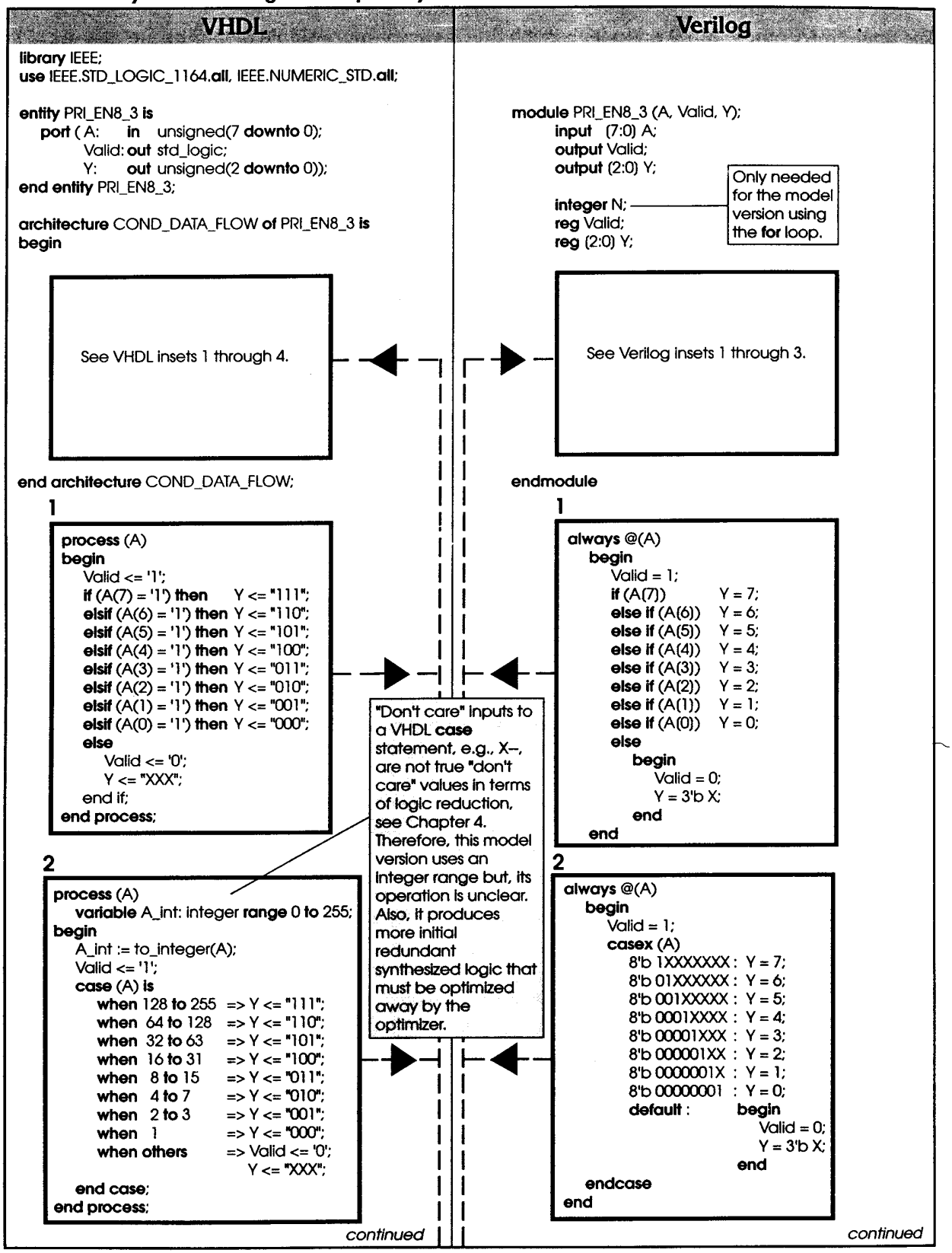
Using **if** *statements.* The first model uses an **if** statement to test each bit in turn starting from the highest priority bit, A7.

Using **case/casex** *statements.* The second model uses a VHDL **case** statement and Verilog **casex** statement. The Verilog **casex** statement is ideally suited for this model as it allows "don't care" input conditions to be used. The VHDL **case** statement is <u>not</u> suitable at all, and the only practical way of using it is to convert signal A from an unsigned to integer data type and specify the appropriate range or each choice value. This type of model will typically cause a synthesis tool to generate large amounts of redundant logic which must then be optimized away by the optimizer. In this particular sized model the optimizer is able to produce identical circuits. However, this may not be the case for larger priority encoders due to the heuristic nature of logic optimizers.

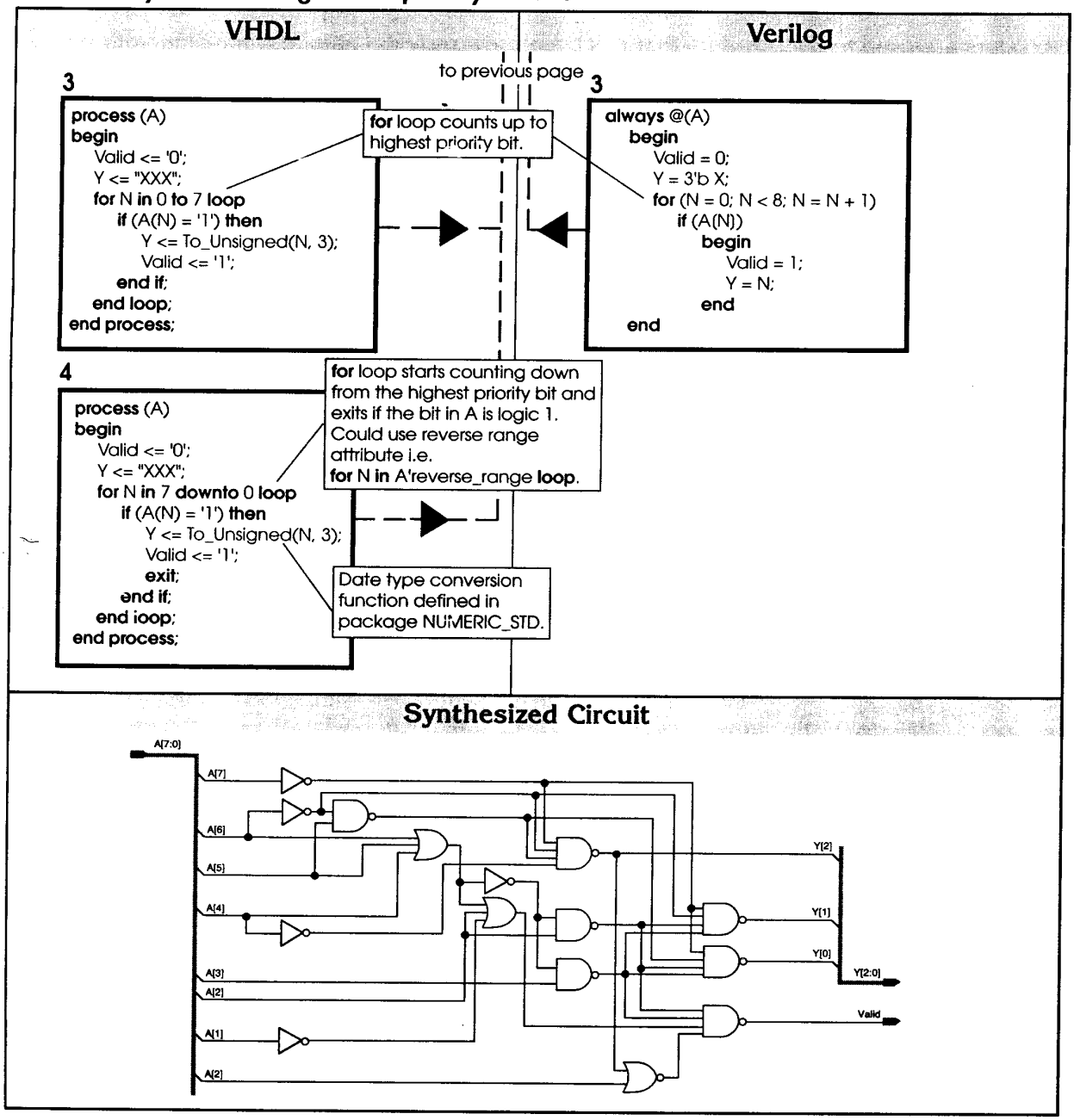
Using conditional signal assignments (VHDL). If the priority encoder was modeled using VHDL conditional signal assignments, two assignments would be needed; one for each output, Valid and Y. Each assignment would be similar in that they would separately select each value of the input A. The synthesized circuit would also be the same, but there would be code duplication for the input selection. This results in more code that is less comprehensible. It is not recommended, and not shown in this example.

Using **for** *loop statements.* The third model uses the **for** loop and tests each bit in turn. The advantage is that the code does not get progressively larger as input and output bit widths increase. Default output values are defined before the **for** statement. There are two VHDL versions; the first checks each bit in turn starting from the least LSB, the second checks each bit in turn starting from the MSB and exits the loop when it has found the first bit having a logic 1 value.

Different ways of modeling an 8-3 priority encoder



Different ways of modeling an 8-3 priority encoder



Decoders

Decoders are used to decode data that has been previously encoded using a binary, or possibly other, type of coded format. An *n*-bit code can represent up to 2^n distinct bits of coded information, so a decoder with *n* inputs can decode up to 2^n outputs. Various models of a 3-8 binary decoder are included in Example 6.8, while various models of a 3-6 binary decoder having a separate enable input are included in Example 6.9.

Example 6.8 A 3-8 binary decoder

The models of a 3-8 binary decoder in this example conform to the truth table in Table 6.4.

Different model versions use **if**, **case** and **for** statements along with VHDL conditional and selected signal assignments. All $2^3 = 8$ possible input values of this 3-8 decoder are decoded to a unique output. This means the automatic priority encoding employed by **if** and Verilog **case** statements do not affect the circuit and "don't care" output values are not needed. Like most other examples in this chapter there is no right or wrong modeling technique. The **case** statement is commonly used because of its clarity, and the fact it is not a continuous assignment and so may simulate faster. As input and output bit widths increase, it is more code efficient to use the **for** loop statement. Again, all models synthesize to the same circuit.

inputs			outputs							
A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
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

Table 6.4 Truth table for a 3-8 line binary decoder

Different ways of modeling a 3-8 decoder

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity DECODER3_8 is
   port ( A: in  integer range 0 to 7;
          Y: out unsigned(7 downto 0));
end entity DECODER3_8;

architecture COND_DATA_FLOW of DECODER3_8 is
begin
```

See VHDL insets 1 through 6 on the following page

```
end architecture COND_DATA_FLOW;
```

continued

Verilog

```
module DECODER3_8 (A, Y);
     input  (2:0) A;
     output (7:0) Y;

     reg (7:0) Y;
     integer N;
```

Only needed for the model version using the **for** loop.

See Verilog insets 1 through 3 on the following page

```
endmodule
```

continued

Different ways of modeling an 3-8 decoder

VHDL

1

```
process (A)
begin
   if (A = 0) then      Y <= "00000001";
   elsif (A = 1) then   Y <= "00000010";
   elsif (A = 2) then   Y <= "00000100";
   elsif (A = 3) then   Y <= "00001000";
   elsif (A = 4) then   Y <= "00010000";
   elsif (A = 5) then   Y <= "00100000";
   elsif (A = 6) then   Y <= "01000000";
   else   Y <= "10000000";
   end if;
end process;
```

2

```
Y <=  "00000001" when A = 0 else
      "00000010" when A = 1 else
      "00000100" when A = 2 else
      "00001000" when A = 3 else
      "00010000" when A = 4 else
      "00100000" when A = 5 else
      "01000000" when A = 6 else
      "10000000";
```

3

```
process (A)
begin
   case A is
      when 0 => Y <= "00000001";
      when 1 => Y <= "00000010";
      when 2 => Y <= "00000100";
      when 3 => Y <= "00001000";
      when 4 => Y <= "00010000";
      when 5 => Y <= "00100000";
      when 6 => Y <= "01000000";
      when 7 => Y <= "10000000";
   end case;
end process;
```

4

```
with A select
   Y <=  "00000001" when 0,
         "00000010" when 1,
         "00000100" when 2,
         "00001000" when 3,
         "00010000" when 4,
         "00100000" when 5,
         "01000000" when 6,
         "10000000" when 7,
         "00000000" when others;
```

5

```
process (A)
begin
   for N in 0 to 7 loop
      if (A = N) then
         Y(N) <= '1';
      else
         Y(N) <= '0';
      end if;
   end loop;
end process;
```

6

```
process (A)
begin
   Y <= (others => '0');
      if (A = N) then
         Y(N) <= '1';
         exit;
      else
         Y(N) <= '0';
      end if;
   end loop;
end process;
```

to previous page

Verilog

1

```
always @(A)
   begin
      if (A == 0)      Y = 8'b 00000001;
      else if (A == 1) Y = 8'b 00000010;
      else if (A == 2) Y = 8'b 00000100;
      else if (A == 3) Y = 8'b 00001000;
      else if (A == 4) Y = 8'b 00010000;
      else if (A == 5) Y = 8'b 00100000;
      else if (A == 6) Y = 8'b 01000000;
      else  Y = 8'b 10000000;
   end
```

All conditional signal assignments must end with an **else** clause with no **when** branch.

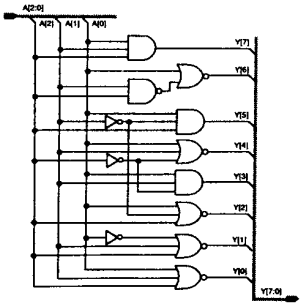
2

```
always @(A)
   begin
      case (A)
         0 : Y = 8'b 00000001;
         1 : Y = 8'b 00000010;
         2 : Y = 8'b 00000100;
         3 : Y = 8'b 00001000;
         4 : Y = 8'b 00010000;
         5 : Y = 8'b 00100000;
         6 : Y = 8'b 01000000;
         7 : Y = 8'b 10000000;
         default : Y = 8'b X;
      endcase
   end
```

3

```
always @(A)
   begin
      for (N = 0; N <= 7; N = N + 1)
         if (A == N)
            Y(N) = 1;
         else
            Y(N) = 0;
   end
```

Synthesized Circuit



Example 6.9 A 3-6 binary decoder with enable

The two model versions of a 3-6 binary decoder are included in this example and conform to the truth table; Table 6.5. Because of the similarities of this example to Example 6.8, only the versions using a **case** statement are covered. This example is different because it has a separate input enable signal and there are two unused binary values for the 3-bit input A. When the enable is inactive (En = 0), or A has an unused value, the 6-bit output must be at logic 0. Like the previous example, "don't care" default assigned values cannot be used.

inputs				outputs					
En	A2	A1	A0	Y5	Y4	Y3	Y2	Y1	Y0
0	X	X	X	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	1	0
1	0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	1	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	0	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

X=don't care

Table 6.5 Truth table for a 3-6 line binary decoder with enable

The first model version below also uses an **if** statement to check the enable input En, separately from the enclosed **case** statement. The second version on the following page has the enable input En, concatenated with the encoded input A and the combined signal used in the **case** statement. Both are correct and synthesize to the same circuit as shown.

3-6 decoder with separate if branch which tests the enable input

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity DECODER3_6_CASE1 is
   port ( En: in  std_logic;
          A:  in  integer range 0 to 7;
          Y:  out unsigned(5 downto 0));
end entity DECODER3_6_CASE1;

architecture COND_DATA_FLOW of DECODER3_6_CASE1 is
begin
   process (A)
   begin
      if (En = '0') then
         Y <= "000000";
      else
         case A is
            when 0 => Y <= "000001";
            when 1 => Y <= "000010";
            when 2 => Y <= "000100";
            when 3 => Y <= "001000";
            when 4 => Y <= "010000";
            when 5 => Y <= "100000";
            when others => Y <= "000000";
         end case;
      end if;
   end process;
end architecture COND_DATA_FLOW;
```

Verilog

```
module DECODER3_6_CASE1 (A, En, Y);
      input  En;
      input  [2:0] A;
      output [5:0] Y;

      reg [5:0] Y;

   always @(En or A)
      begin
         if (! En)
            Y = 6'b 0;
         else
            case (A)
               0 : Y = 6'b 000001;
               1 : Y = 6'b 000010;
               2 : Y = 6'b 000100;
               3 : Y = 6'b 001000;
               4 : Y = 6'b 010000;
               5 : Y = 6'b 100000;
               default : Y = 6'b 0;
            endcase
      end

endmodule
```

3-6 decoder with concantenated enable/encoded input for case selector

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity DECODER3_6_CASE2 is
   port ( En: in  std_logic;
          A:  in  unsigned(2 downto 0);
          Y:  out unsigned(5 downto 0));
end entity DECODER3_6_CASE2;

architecture COND_DATA_FLOW of DECODER3_6_CASE2 is
begin
   process (En, A)
      variable En_concat_A: unsigned(3 downto 0);
   begin
      En_concat_A := En & A;
      case En_concat_A is
         when "1000" => Y <= "000001";
         when "1001" => Y <= "000010";
         when "1010" => Y <= "000100";
         when "1011" => Y <= "001000";
         when "1100" => Y <= "010000";
         when "1101" => Y <= "100000";
         when others => Y <= "000000";
      end case;
   end process;
end architecture COND_DATA_FLOW;
```

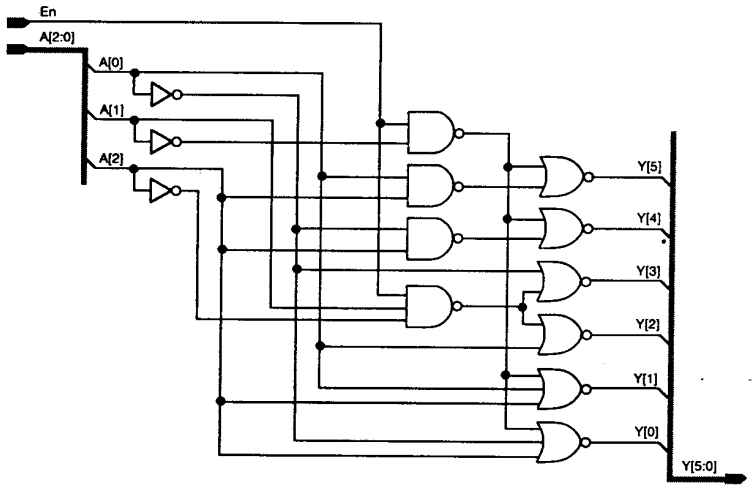
Separate concatenation expression. Not allowed in **case** expression like Verilog.

Verilog

```
module DECODER3_6_CASE2 (A, En, Y);
      input  En;
      input  (2:0) A;
      output (5:0) Y;

      reg (3:0) En_concat_A;
      reg (5:0) Y;

   always @(En or A)
      begin
         case ({En, A})
            4'b 1000 : Y = 6'b 000001;
            4'b 1001 : Y = 6'b 000010;
            4'b 1010 : Y = 6'b 000100;
            4'b 1011 : Y = 6'b 001000;
            4'b 1100 : Y = 6'b 010000;
            4'b 1101 : Y = 6'b 100000;
            default  : Y = 6'b 0;
      endcase
   end

endmodule
```

Concatenation in **case** expression.

Synthesized Circuit



Example 6.10 Four bit address decoder

This example is of a four bit address decoder. It provides enable signals for segments of memory, the address map of which, is shown in Figure 6.3. The decoder's inputs could be the upper four bits of a larger address bus in which case the decoded outputs would enable larger segments of memory. Seven enable outputs are provided; one for each memory segment. The address map is divided into quarters, and the second quarter is further subdivided into four. There are four outputs from the second quarter corresponding to four consecutive binary input values.

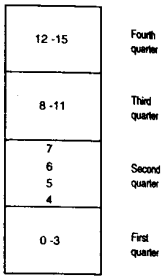


Figure 6.3 Address map

The first model version uses a **for** loop enclosing an **if** statement while the second model uses a **case** statement. As a general rule, it is better to use the **for** loop and **if** statements when a large number of consecutively decoded outputs are required. This is because a **case** statement requires a separate choice branch for each decoded output.

Four bit address decoder using "if" in a "for" loop

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity ADD_DEC_IF is
   port ( Address: in  integer range 0 to 15;
          AddDec_0to3,AddDec_8to11,
          AddDec_12to15: out std_logic;
          AddDec_4to7:  out unsigned(3 downto 0));
end entity ADD_DEC_IF;
architecture COND_DATA_FLOW of ADD_DEC_IF is
begin
   process (Address)
   begin
      -- First quarter
      if (Address >= 0 and Address <= 3) then
         AddDec_0to3 <= '1';
      else
         AddDec_0to3 <= '0';
      end if;

      -- Third quarter
      if (Address >= 8 and Address <= 11) then
         AddDec_8to11 <= '1';
      else
         AddDec_8to11 <= '0';
      end if;

      -- Fourth quarter
      if (Address >= 12 and Address <= 15) then
         AddDec_12to15 <= '1';
      else
         AddDec_12to15 <= '0';
      end if;

      -- Second quarter
      for N in AddDec_4to7'range loop
         if (Address = N + 4) then
            AddDec_4to7(N) <= '1';
         else
            AddDec_4to7(N) <= '0';
         end if;
      end loop;
   end process;

end architecture COND_DATA_FLOW;
```

Verilog

```
module ADD_DEC_IF
   (Address, AddDec_0to3, AddDec_4to7,
    AddDec_8to11, AddDec_12to15);
      input  (3:0)  Address;
      output        AddDec_0to3, AddDec_8to11,
                    AddDec_12to15;
      output (3:0)  AddDec_4to7;
      integer N;
      reg  AddDec_0to3, AddDec_8to11,
           AddDec_12to15;
      reg (3:0) AddDec_4to7;
   always @(Address)
   begin
      // First quarter
      if (Address >= 0 && Address <= 3)
         AddDec_0to3 = 1;
      else
         AddDec_0to3 = 0;

      // Third quarter
      if (Address >= 8 && Address <= 11)
         AddDec_8to11 = 1;
      else
         AddDec_8to11 = 0;

      // Fourth quarter
      if (Address >= 12 && Address <= 15)
         AddDec_12to15 = 1;
      else
         AddDec_12to15 = 0;

      // Second quarter
      for (N = 0; N <= 3; N = N + 1)
         if (Address == N + 4)
            AddDec_4to7(N) = 1;
         else
            AddDec_4to7(N) = 0;
   end

endmodule
```

Four bit address decoder using "case"

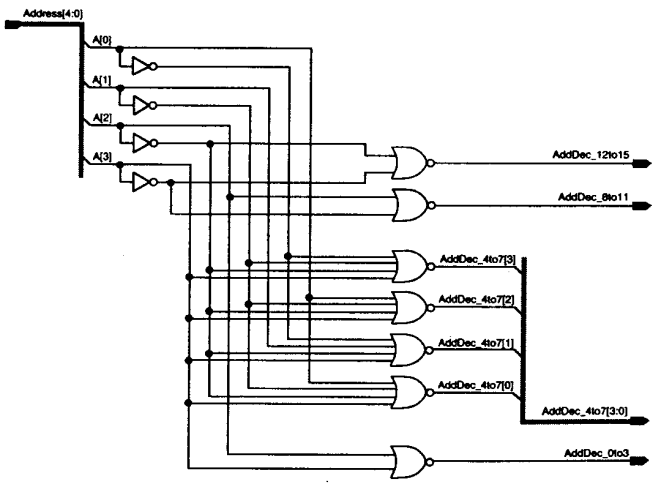
VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity ADD_DEC_CASE is
   port ( Address:          in  integer range 0 to 15;
          AddDec_0to3, AddDec_8to11,
          AddDec_12to15: out std_logic;
          AddDec_4to7:   out unsigned(3 downto 0));
end entity ADD_DEC_CASE;

architecture COND_DATA_FLOW of ADD_DEC_CASE is
begin

   process (Address)
   begin
      AddDec_0to3    <= '0';
      AddDec_4to7    <= (others => '0');
      AddDec_8to11   <= '0';
      AddDec_12to15  <= '0';

      case Address is
         -- First quarter
         when 0 to 3 =>
            AddDec_0to3 <= '1';

         -- Second quarter
         when 4 =>  AddDec_4to7(0) <= '1';
         when 5 =>  AddDec_4to7(1) <= '1';
         when 6 =>  AddDec_4to7(2) <= '1';
         when 7 =>  AddDec_4to7(3) <= '1';

         -- Third quarter
         when 8 to 11 =>
            AddDec_8to11 <= '1';

         -- Fourth quarter
         when 12 to 15 =>
            AddDec_12to15 <= '1';
      end case;
   end process;

end architecture COND_DATA_FLOW;
```

Verilog

```
module ADD_DEC_CASE
   (Address, AddDec_0to3, AddDec_4to7, AddDec_8to11,
    AddDec_12to15);
      input  (3:0)  Address;
      output        AddDec_0to3, AddDec_8to11,
                    AddDec_12to15;
      output (3:0)  AddDec_4to7;

      reg  AddDec_0to3, AddDec_8to11, AddDec_12to15;
      reg (3:0) AddDec_4to7;

   always @(Address)
    begin
      AddDec_0to3    = 0;
      AddDec_4to7    = 0;
      AddDec_8to11   = 0;
      AddDec_12to15  = 0;

      case (Address)
         // First quarter
         0, 1, 2, 3:
            AddDec_0to3 = 1;

         // Second quarter
         4:  AddDec_4to7(0) = 1;
         5:  AddDec_4to7(1) = 1;
         6:  AddDec_4to7(2) = 1;
         7:  AddDec_4to7(3) = 1;

         // Third quarter
         8, 9, 10, 11:
            AddDec_8to11 = 1;

         // Fourth quarter
         12, 13, 14, 15:
            AddDec_12to15 = 1;
      endcase
    end
endmodule
```

Synthesized Circuit



Example 6.11 Generic N to M bit binary decoder

A generic *n*-bit input, *m*-bit output binary decoder is illustrated and incorporates a separate enable input. Like Example 6.9, all outputs will be at logic 0 if the decoder is not enabled, that is, En = 0, or it is enabled, but *n* has a value that is not used in the decoder. This generic decoder is called twice for the inference of a 2-4 and a 3-6 decoder.

The four models in this example are:

VHDL 1 - a generic VHDL decoder using an **entity**,
Verilog 1 - a generic Verilog decoder using a **module**,
VHDL 2 - a generic VHDL decoder using a function in a package,
Verilog 2 - non-generic Verilog using a decoder function Verilog 2.

There are two parts to each of the four models. The first part of VHDL 1 and Verilog 1 show the decoder model while the second part shows two separate instantiations of it. The first part of VHDL 2 and Verilog 2 show the decoder modeled in a function and the second part shows two function calls to it.

VHDL 1. Modeled using an **entity** statement and separately instantiated. The number of decoded input and output lines, needed for any given instance, are specified using a generic clause which specifies a value for SizeIn and SizeOut.

Verilog 1. Modeled using a **module** statement and instantiated from a separate module. Uses overloaded parameters for both input and output bit widths. These parameters have values defined for them in the generic decoder (SizeIn = 3 and SizeOut = 8), and overridden when instantiated from the instantiation in another **module**.

VHDL 2. Uses a generic function defined in a package. This is a more practical and easier method to use when compared with using a VHDL **entity** as described above. The function is called from an expression, either concurrently (outside a process) or sequentially (inside a process), by supplying;

- the enable input of type std_logic,
- the encoded input of type unsigned,
- the desired number of encoded inputs of type integer,
- the desired number of decoded outputs of type integer.

Verilog 2. The Verilog language does not support the overriding of parameters in a function call. Instead of being able to model a generic decoder, a predefined number of *n-m* bit decoders must be specified so that the appropriate decoder may be called when needed. In this version, two functions have been declared for 2-4 and 3-6 decoders. These decoders have been placed in a separate file and included in the calling **module** using the compiler directive `include.

Generic decoder (entity/module)

VHDL 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity GENERIC_DECODER_ENTITY is
   generic (SizeIn, SizeOut: integer);
   port ( En: in  std_logic;
          A: in  unsigned(SizeIn - 1 downto 0);
          Y: out unsigned(SizeOut - 1 downto 0));
end entity GENERIC_DECODER_ENTITY;

architecture DATA_FLOW of GENERIC_DECODER_ENTITY is
begin

   process (En, A)
   begin
      if (En = '0') then
         Y <= (others => '0');
      else
         for N in 0 to SizeOut - 1 loop
            if (to_integer (A) = N) then
               Y(N) <= '1';
            else
               Y(N) <= '0';
            end if;
         end loop;
      end if;
   end process;

end architecture DATA_FLOW;
```

Verilog 1

```
module GENERIC_DECODER_MODULE (En, A, Y);
      parameter SizeIn  = 3,
                SizeOut = 8;
      input     En;
      input     [SizeIn - 1:0] A;
      output    [SizeOut - 1:0] Y;
      reg [SizeOut - 1:0] Y;

      integer N;

   always @(En or A)
      begin
         if (!En)
            Y = 0;
         else
            if (A > SizeOut - 1)
               for (N = 0; N <= SizeOut - 1; N = N + 1)
                  Y[N] = 1'b X;
            else
               for (N = 0; N <= SizeOut - 1; N = N + 1)
                  if (A == N)
                     Y[N] = 1;
                  else
                     Y[N] = 0;
      end

endmodule
```

Two instantiations of the generic decoder

VHDL 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity GENERIC_DECODER_ENTITY_CALL is
   port ( EnA, EnB:   in  std_logic;
          AddA:       in  unsigned(1 downto 0);
          AddB:       in  unsigned(2 downto 0);
          DecAddA: out unsigned(3 downto 0);
          DecAddB: out unsigned(5 downto 0));
end entity GENERIC_DECODER_ENTITY_CALL;

architecture STRUCT of GENERIC_DECODER_ENTITY_CALL is
   component GENERIC_DECODER_ENTITY
      generic (SizeIn, SizeOut: integer);
      port ( En: in  std_logic;
             A: in  unsigned(SizeIn - 1 downto 0);
             Y: out unsigned(SizeOut - 1 downto 0));
   end component;
begin

   Decoder2_4: GENERIC_DECODER_ENTITY
                  generic map (2, 4)
                  port map (EnA, AddA, DecAddA);

   Decoder3_6: GENERIC_DECODER_ENTITY
                  generic map (3, 6)
                  port map (EnB, AddB, DecAddB);

end architecture STRUCT;
```

Verilog 1

```
module GENERIC_DECODER_MODULE_CALL
   (EnA, EnB, AddA, AddB, DecAddA, DecAddB);
      input          EnA, EnB;
      input    [1:0] AddA;
      input    [2:0] AddB;
      output   [3:0] DecAddA;
      output   [5:0] DecAddB;

   GENERIC_DECODER_MODULE
            #(2, 4) Decoder2_4(EnA, AddA, DecAddA);

   GENERIC_DECODER_MODULE
            #(3, 6) Decoder3_6(EnB, AddB, DecAddB);

endmodule
```

Two instantiations, 2-4 and 3-6, of the generic decoder.

VHDL generic decoder (function) - Verilog specific decoders (functions)

VHDL 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

package GENERIC_DECODER_FN_PKG is
   function Decoder (En:  std_logic;
                     A:   unsigned;
                     SizeIn, SizeOut: integer) return unsigned;
end GENERIC_DECODER_FN_PKG;

package body GENERIC_DECODER_FN_PKG is
   function Decoder (En:  std_logic;
                     A:   unsigned;
                     SizeIn, SizeOut: integer) return unsigned is
      variable Y: unsigned(Size - 1 downto 0);
   begin
      if (En = '0' or A > Size - 1) then
         Y := (others => '0');
      else
         for N in 0 to SizeOut - 1 loop
            if (to_integer(A) = N) then
               Y(N) := '1';
            else
               Y(N) := '0';
            end if;
         end loop;
      end if;
      return Y;
   end Decoder;
end GENERIC_DECODER_FN_PKG;
```

Generic decoder function defined in a package.

Verilog 2

```
// ------------------------------------
// This file must be called "decoder_fns.v".
// ------------------------------------
function (3:0) Decode2_4;
      input En;
      input (1:0) A;
      integer N;
   begin
      if (!En)
         Decode2_4 = 4'b 0;
      else
         for (N = 0; N < 4; N = N + 1)
            if (A == N)
               Decode2_4(N) = 1;
            else
               Decode2_4(N) = 0;
   end
endfunction

function (5:0) Decode3_6;
      input En;
      input (2:0) A;
      integer N;
   begin
      if (!En)
         Decode3_6 = 6'b 0;
      else
         for (N = 0; N < 6; N = N + 1)
            if (A == N)
               Decode3_6(N) = 1;
            else
               Decode3_6(N) = 0;
   end
endfunction
```

Two specific decoder functions.

Two decoder function calls

VHDL 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

use work.GENERIC_DECODER_PKG.all;

entity GENERIC_DECODER_CALL_FN is
   port ( EnA, EnB:   in  std_logic;
          AddA:       in  unsigned(1 downto 0);
          AddB:       in  unsigned(2 downto 0);
          DecAddA:    out unsigned(3 downto 0);
          DecAddB:    out unsigned(5 downto 0));
end entity GENERIC_DECODER_CALL_FN;

architecture DATA_FLOW of GENERIC_DECODER_CALL_FN is
begin

   process (EnA, AddA, EnB, AddB)
   begin
      DecAddA <= Decoder(EnA, AddA, 2, 4);
      DecAddB <= Decoder(EnB, AddB, 3, 6);
   end process;

end architecture DATA_FLOW;
```

Two function calls to the generic decoder.

Verilog 2

```
module DECODER_FN_CALLS
   (EnA, EnB, AddA, AddB, DecAddA, DecAddB);
      input    EnA, EnB;
      input    (1:0) AddA;
      input    (2:0) AddB;
      output   (3:0) DecAddA;
      output   (5:0) DecAddB;

      reg (3:0) DecAddA;
      reg (5:0) DecAddB;

      `include "decoder_fns.v"

always @(EnA or EnB or AddA or AddB)
   begin
      DecAddA = Decode2_4(EnA, AddA);
      DecAddB = Decode3_6(EnB, AddB);
   end

endmodule
```

Function calls to 2_4 and 3_6 decoders.

Comparators

A comparator compares two or more inputs using one, or a number of different comparisons. When the given relationship(s) is true, an output signal is given (logic 0 or logic 1). Comparators are only modeled using the **if** statement with an **else** clause and no **else-if** clauses. A VHDL conditional signal assignment or Verilog conditional continuous assignment could also be used, but is less common as a sensitivity list (VHDL) or event list (Verilog) cannot be specified to improve simulation time. Any two data objects are compared using equality and relational operators in the expression part of the **if** statement. Only two data objects can be compared at once, that is, statements like "**if** (A = B = C)" cannot be used. However, logical operators can be used to logically test the result of multiple comparisons, for example, **if** ((A = B) **and** (A = C)). These equality, relational and logical operators are listed in Table 6.6.

Example 6.12 shows a 6-bit two input equality comparator. Example 6.13 shows how multiple comparisons are used.

Operators	VHDL	Verilog
Equality & Relational	=	==
	!=	!=
	<	<
	<=	<=
	>	>
	>=	>=
Logical	not	!
	and	&&
	or	||

Table 6.6 Equality, relational and logical operators

Example 6.12 Simple Comparator

Identical equality comparators are shown coded in three different ways. The single bit output is at logic 1 when the two 6-bit input busses are the same, otherwise it is at logic 0.

Three ways to infer a 6-bit equality comparator

VHDL

```
library IEEE;
use IEEE.STD_Logic_1164.all, IEEE.Numeric_STD.all;
entity COMPARATOR_EQUALITY is
   port (A1,B1,A2,B2,A3,B3:  in   unsigned(5 downto 0);
         Y1, Y2, Y3:         out  std_logic);
end entity COMPARATOR_EQUALITY;
architecture LOGIC of COMPARATOR_EQUALITY is
begin
   COMPARE:
   process (A1, B1, A2, B2, A3, B3)
   begin
      Y1 <= '1';
      for N in 0 to 5 loop
         if (A1(N) /= B1(N)) then
            Y1 <= '0';
            exit;
         else
            null;
         end if;
      end loop;
      Y2 <= '0';
      if (A2 = B2) then
         Y2 <= '1';
      end if;
      if (A3 = B3) then
         Y3 <= '1';
      else
         Y3 <= '0';
      end if;
   end process;
end architecture LOGIC;
```

Each bit compared in turn in a **for** loop.

Default defined before the **if** so no **else** clause.

Most common and easiest to read method.

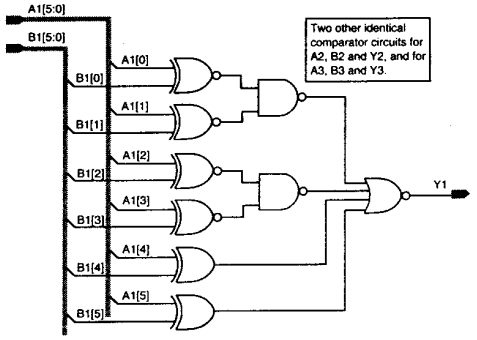
Verilog

```
module COMPARATOR_EQUALITY
   (A1, B1, A2, B2, A3, B3, Y1, Y2, Y3);
      input  (5:0)  A1, B1, A2, B2, A3, B3;
      output        Y1, Y2, Y3;

      integer N;
      reg  Y1, Y2, Y3;

   always @(A1 or B1 or A2 or B2 or A3 or B3)
      begin: COMPARE
         Y1 = 1;
         for (N = 0; N < 6; N = N + 1)
            if (A1(N) != B1(N))
               Y1 = 0;
            else
               ;
         Y2 = 0;
         if (A2 == B2)
            Y2 = 1;
         if (A3 == B3)
            Y3 = 1;
         else
            Y3 = 0;
      end

endmodule
```

Semi colon means null, do nothing.

Three ways to infer a 6-bit equality comparator

Synthesized Circuit



Example 6.13 Multiple Comparison Comparator

Extra parentheses enclosing "C /= D **or** E >= F" means that either one of these conditions and "A = B" must be true for the output to be at logic 1.

Comparator using multiple comparisons

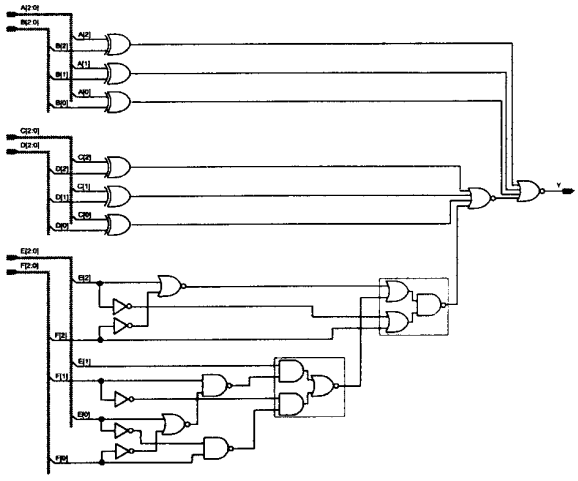
VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity COMPARATOR_MULT_COMP is
   port (A, B, C, D, E, F: in  unsigned(2 downto 0);
         Y:  out std_logic);
end entity COMPARATOR_MULT_COMP;
 architecture LOGIC of COMPARATOR_MULT_COMP is
begin
   process (A, B, C, D, E, F)
   begin
      if (A = B and (C /= D or E >= F )) then
         Y <= '1';
      else
         Y <= '0';
      end if;
   end process;
end architecture LOGIC;
```

Verilog

```
module COMPARATOR_MULT_COMP (A, B, C, D, E, F, Y);
      input  [2:0]  A, B, C, D, E, F;
      output        Y;

      reg  Y;


   always @(A or B or C or D or E or F)
      if (A == B && (C != D | | E >= F))
         Y = 1;
      else
         Y = 0;

endmodule
```

Synthesized Circuit



ALU

An arithmetic logic unit (ALU) is the center core of a central processing unit (CPU). It consists of purely combinational logic circuit and performs a set of arithmetic and logic micro operations on two input busses. It has *n* encoded inputs for selecting which operation to perform. The select lines are decoded within the ALU to provide up to 2^n different operations. The ALU in Example 6.14 is capable of performing 14 different micro operations.

Example 6.14 An arithmetic logic unit

An Arithmetic Logic Unit (ALU) is modeled to the function table of Table 6.7.

S4	S3	S2	S1	S0	Cin	Operation	Function	Implementation block
0	0	0	0	0	0	Y <= A	Transfer A	Arithmetic Unit
0	0	0	0	0	1	Y <= A + 1	Increment A	Arithmetic Unit
0	0	0	0	1	0	Y <= A + B	Addition	Arithmetic Unit
0	0	0	0	1	1	Y <= A + B + 1	Add with carry	Arithmetic Unit
0	0	0	1	0	0	Y <= A + Bbar	A plus 1's complement of B	Arithmetic Unit
0	0	0	1	0	1	Y <= A + Bbar + 1	Subtraction	Arithmetic Unit
0	0	0	1	1	0	Y <= A - 1	Decrement A	Arithmetic Unit
0	0	0	1	1	1	Y <= A	Transfer A	Arithmetic Unit
0	0	1	0	0	0	Y <= A and B	AND	Logic Unit
0	0	1	0	1	0	Y <= A or B	OR	Logic Unit
0	0	1	1	0	0	Y <= A xor B	XOR	Logic Unit
0	0	1	1	1	0	Y <= Abar	Complement A	Logic Unit
0	0	0	0	0	0	Y <= A	Transfer A	Shifter Unit
0	1	0	0	0	0	Y <= shl A	Shift left A	Shifter Unit
1	0	0	0	0	0	Y <= shr A	Shift right A	Shifter Unit
1	1	0	0	0	0	Y <= 0	Transfer 0's	Shifter Unit

Table 6.7 ALU Function table

This whole function table could be modeled using a single **case** statement, however, its synthesized structure would be poor. Instead, the ALU has been modeled with a separate arithmetic unit, logic unit and shifter, as indicated by the modeled circuit structure. By separating the arithmetic and logic units in this way, and multiplexing their outputs to the shifter, better pre-optimized timing will result. It is very likely, that even after optimization, the shortest timing delay through the ALU will be longer if the arithmetic and logic units were combined into one process.

The arithmetic unit modeled using a single **case** statement. The reason it can be modeled in this way is because the synthesis tools from VeriBest Incorporated, synthesizes expressions like A + B + 1 to a single adder with the carry in set to logic 1. If a synthesis tool is being used that does not support this, it is necessary to remodel it in a way that avoids multiple adders being synthesized. Provided the synthesis tools resource sharing option is turned on, the synthesized logic of the arithmetic unit will consist of just one adder for all add and subtract operations.

Arithmetic logic unit

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity ALU is
  port ( Sel:     in  unsigned(4 downto 0);
         CarryIn: in  std_logic;
         A, B:    in  unsigned(7 downto 0);
         Y:       out unsigned(7 downto 0));
end entity ALU;
```

continued

Verilog

```
module ALU (Sel, CarryIn, A, B, Y);
    input   (4:0) Sel;
    input         CarryIn;
    input   (7:0) A, B;
    output  (7:0) Y;
    reg     (7:0) Y;
```

continued

Arithmetic logic unit

VHDL

```
architecture COND_DATA_FLOW of ALU is
begin
   ALU_AND_SHIFT:
   process (Sel, A, B, CarryIn)
      variable Sel0_1_CarryIn: unsigned(2 downto 0);
      variable LogicUnit, ArithUnit,
               ALU_NoShift: unsigned(7 downto 0);
   begin
      ----------------
      -- Logic Unit
      ----------------
      LOGIC_UNIT: case Sel(1 downto 0) is
         when "00"   =>  LogicUnit := A and B;
         when "01"   =>  LogicUnit := A  or B;
         when "10"   =>  LogicUnit := A xor B;
         when "11"   =>  LogicUnit := not A;
         when others =>  LogicUnit := (others => 'X');
      end case LOGIC_UNIT;
      ------------------------
      -- Arithmetic Unit
      ------------------------
      Sel0_1_CarryIn := Sel(1 downto 0) & CarryIn;
      ARITH_UNIT: case Sel0_1_CarryIn is
         when "000"  =>  ArithUnit := A;
         when "001"  =>  ArithUnit := A + 1;
         when "010"  =>  ArithUnit := A + B;
         when "011"  =>  ArithUnit := A + B + 1;
         when "100"  =>  ArithUnit := A + not B;
         when "101"  =>  ArithUnit := A - B;
         when "110"  =>  ArithUnit := A - 1;
         when "111"  =>  ArithUnit := A;
         when others =>  ArithUnit := (others => 'X');
      end case ARITH_UNIT;
      ------------------------------------------------------------
      -- Multiplex between Logic & Arithmetic Units
      ------------------------------------------------------------
      LA_MUX: if (Sel(2) = '1') then
         ALU_NoShift := LogicUnit;
      else
         ALU_NoShift := ArithUnit;
      end if LA_MUX;
      ------------------------
      -- Shift operations
      ------------------------
      SHIFT: case Sel(4 downto 3) is
         when "00"   => Y <= ALU_NoShift;
         when "01"   => Y <= Shift_left(ALU_NoShift, 1);
         when "10"   => Y <= Shift_right(ALU_NoShift, 1);
         when "11"   => Y <= (others => '0');
         when others => Y <= (others => 'X');
      end case SHIFT;
   end process ALU_AND_SHIFT;
end architecture COND_DATA_FLOW;
```

Separate VHDL concatenation. Cannot be incorporated in case expression like Verilog.

Verilog

```
reg (7:0) LogicUnit, ArithUnit,
          ALU_NoShift;

always @(Sel or A or B or CarryIn)
   begin: ALU_PROC

      //----------------
      // Logic Unit
      //----------------
      case (Sel(1:0))
         2'b 00  : LogicUnit = A & B;
         2'b 01  : LogicUnit = A | B;
         2'b 10  : LogicUnit = A ^ B;
         2'b 11  : LogicUnit = ! A;
         default : LogicUnit = 8'b X;
      endcase
      //------------------------
      // Arithmetic Unit
      //------------------------
      case ({Sel(1:0), CarryIn})
         3'b 000 : ArithUnit = A;
         3'b 001 : ArithUnit = A + 1;
         3'b 010 : ArithUnit = A + B;
         3'b 011 : ArithUnit = A + B + 1;
         3'b 100 : ArithUnit = A + ! B;
         3'b 101 : ArithUnit = A - B;
         3'b 110 : ArithUnit = A - 1;
         3'b 111 : ArithUnit = A;
         default : ArithUnit = 8'b X;
      endcase
      //------------------------------------------------------------
      // Multiplex between Logic & Arithmetic Units
      //------------------------------------------------------------
      if (Sel(2))
         ALU_NoShift = LogicUnit;
      else
         ALU_NoShift = ArithUnit;

      //------------------------
      // Shift operations
      //------------------------
      case (Sel(4:3))
      2'b 00  : Y = ALU_NoShift;
      2'b 01  : Y = ALU_NoShift << 1;
      2'b 10  : Y = ALU_NoShift >> 1;
      2'b 11  : Y = 8'b 0;
      default : Y = 8'b X;
   endcase
end

endmodule
```

Modeled Circuit Structure