XAPP105 (v2.0) August 30, 2001

A CPLD VHDL Introduction

Summary

This introduction covers the fundamentals of VHDL as applied to Complex Programmable Logic Devices (CPLDs). Specifically included are those design practices that translate soundly to CPLDs, permitting designers to use the best features of this powerful language to extract optimum performance for CPLD designs.

Introduction

VHDL, an extremely versatile tool developed to aid in many aspects of IC design, allows a user to structure circuits in many levels of detail. This versatility also makes the job of the VHDL synthesis tool a lot more complex, and there is latitude for interpretation depending on the VHDL coding style. One synthesis tool may implement the same code very differently from another. In order to achieve the best results using VHDL, the designer should work at the Register Transfer Level (RTL).

Although working at the RTL for designs may be more time-consuming, all major synthesis tools on the market are capable of generating a clear cut implementation of designs for CPLDs at this level. Using higher levels of abstraction may give adequate results, but tend to be less efficient. Additionally, by expressing designs in this manner, the designer also gains the ability to port VHDL designs from one synthesis tool to another with minimal effort. The following examples will show designers the best design practices when targeting Xilinx XC9500XL, XC9500XV and CoolRunner™ XPLA3 families.

This application note covers the following topics:

- **Multiplexers**
- **Encoders**
- **Decoders**
- **Comparators**
- **Adders**
- **Modeling Synchronous Logic Circuits**
- **Asynchronous Counters**
- **Finite State Machines**
- **Coding Techniques**

Multiplexers

Multiplexers can be modeled in various ways. The four common methods are to:

1. Use an **if** statement followed by multiple **elsif** statements.
2. Usage of a **case** statement.
3. Conditional signal assignment.
4. Selected signal assignment

The example below shows the coding for a 1-bit wide 4:1 multiplexer.

There is no incorrect method of modeling a multiplexer. However, **case** statements require less code than **if** statements. The conditional and selected signal assignments have to reside outside a process. Therefore, they will always be active and will take longer to simulate.

One-bit Wide 4:1 Mux

```
library ieee;
use ieee.std_logic_1164.all;
--Comments are denoted by two - signs.
entity MUX4_1 is
  port
  (
  Sel       : in std_logic_vector(1 downto 0);
  A, B, C, D  : in std_logic;
  Y        : outstd_logic
  );
end MUX4_1;

architecture behavior of MUX4_1 is
begin

--INSERT 1, 2, 3 or 4 here

--1 : If statements
  process (Sel, A, B, C, D)
  begin
    if (Sel = “00”) then
      Y<= A;
    Elsif (Sel = “01”) then
      Y<= B;
    Elsif (Sel = “10”) then
      Y<= C;
    Else
      Y<= D;
   end if;
  end process;
end behavior;

--2: case statements
process (Sel, A, B, C, D)
  begin
     case Sel is
        when “00” => Y<=A;
        when “01” => Y<=B;
        when “10” => Y<=C;
        when “11” => Y<=D;
        when others => Y<=A;
      end case;
    end process;
 end behavior;

--3: Conditional signal assignment
-- equivalent to if but is concurrent and so outside of a process.
Y <= A when Sel = “00” else
     B when Sel = “01” else
     C when Sel = “10” else
     D;          -- when Sel = “11”;
End behavior;

--4 Selected signal assignment
-- Multiplexer selection is very clear. Again it is a concurrent statement.
-- Equivalent to a case statement but outside of a process
```

```
with Sel select
   Y<= A when “00”,
       B when “01”,
       C when “10”,
       D when “11”,
       A when others;
End behavior;
```

When compiled onto an XC9536XL, the resulting usage is as follows:

```
Design name: Multi
Device used: XC9536XL –5-PC44
Fitting Status: Successful
*************************  Resource Summary  *************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
1 /36 (2%)	4 /180 (2%)	0 /36 (0%)	7 /34 (20%)	6 /108 (5%)

As shown above, a 4:1 multiplexer can be implemented in a single XC9500 macrocell.

Seven pins are used in the design – A, B, C, D, Sel<0>, and Sel<1> are inputs and Y is an output. The design is purely combinatorial, as there are no registers used. Four product terms are used. A closer look at the "Implemented Equations" section of the Fitter report will explain what these four product terms are:

; Implemented Equations.

```
Y  =  A * /"Sel<0>" * /"Sel<1>"
         + "Sel<0>" * B * /"Sel<1>"
         + "Sel<0>" * D * "Sel<1>"
         + /"Sel<0>" * C * "Sel<1>"
```

Two-bit Wide 8:1 Mux

```
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 std_logic_vector(1 downto 0);
       Y: out std_logic_vector(1 downto 0));
End entity MUX2X8_1_CASE;
Architecture behavior of MUX2X8_1_CASE is
Begin
```

--INSERT 1 or 2

```
-- 1: case statements
   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 behavior;
```

```
-- 2: selected signal assignment
   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 behavior;
```

In the example above, a 2-bit wide 8:1 multiplexer is implemented using case statements and selected signal assignments. The resulting code gives the following usage summary:

```
Design Name: multi2
Device Used: XC9536XL –5 –PC44
Fitting Status: Successful
**************************  Resource Summary  **************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
2 /36 (5%)	16 /180 (8%)	0 /36 (0%)	21 /34 (61%)	22 /108 (20%)

This 8:1 multiplexer uses a total of 2 macrocells, 16 product terms and 21 pins. The two macrocells used in this design are for Y<0> and Y<1>, which reside in FB1_4 (Function Block 1, Macrocell 4) and FB2_4 respectively. Both Y<0> and Y<1> use eight product terms as shown in the "Implemented Equations" section below. The architecture of the XC9500 family has five local product terms available to it. When more than five P-terms are necessary, they may be borrowed from the neighboring macrocells above and/or below the macrocell. In this case, Y<0> and Y<1> borrow P-terms from macrocells above and below. For example, Y<0>, which is in FB1_4, borrows two product terms from its neighbor above, FB1_3, and also borrows one product term from its neighbor below, FB1_5.

; Implemented Equations.

```
 "Y<1>"  =  "Sel<2>" * "Sel<0>" * "A7<1>" * "Sel<1>"
  + "Sel<2>" * /"Sel<0>" * "A6<1>" * "Sel<1>"
  + /"Sel<2>" * "Sel<0>" * "A3<1>" * "Sel<1>"
  + /"Sel<2>" * /"Sel<0>" * "A2<1>" * "Sel<1>"
  + /"Sel<2>" * /"Sel<0>" * "A0<1>" * /"Sel<1>"
;Imported pterms FB1_3
  + "Sel<2>" * /"Sel<0>" * "A4<1>" * /"Sel<1>"
  + /"Sel<2>" * "Sel<0>" * "A1<1>" * /"Sel<1>"
;Imported pterms FB1_5
  + "Sel<2>" * "Sel<0>" * "A5<1>" * /"Sel<1>"

 "Y<0>"  =  "Sel<2>" * "Sel<0>" * "A7<0>" * "Sel<1>"
  + "Sel<2>" * /"Sel<0>" * "A6<0>" * "Sel<1>"
  + /"Sel<2>" * "Sel<0>" * "A3<0>" * "Sel<1>"
  + /"Sel<2>" * /"Sel<0>" * "A2<0>" * "Sel<1>"
  + /"Sel<2>" * /"Sel<0>" * "A0<0>" * /"Sel<1>"
;Imported pterms FB2_3
  + "Sel<2>" * "Sel<0>" * "A5<0>" * /"Sel<1>"
  + /"Sel<2>" * "Sel<0>" * "A1<0>" * /"Sel<1>"
;Imported pterms FB2_5
  + "Sel<2>" * /"Sel<0>" * "A4<0>" * /"Sel<1>"
```

Encoders

An encoder creates a data output set that is more compact than the input data. A decoder reverses the encoding process. The truth table for an 8:3 encoder is shown below. We must assume that only one input may have a value of "1" at any given time. Otherwise the circuit is undefined. Note that the binary value of the output matches the subscript of the asserted inputs.

Table 1: **Truth Table, 8:3 Encoder**

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

The encoder described by the truth table may be modeled by using the **if, case** statements, or selected signal assignments. Once again, **case** statements are more concise and clear than **if** statements, and this becomes increasingly obvious when the number of inputs to the encoder increase. Selected signal assignment is also very clear. It is the concurrent equivalent of **case** statement. The **for** loop is better for modeling a larger or more generic encoder. The **if** statement and the **for** loop encoder are not depicted in this example.

An 8:3 Binary Encoder

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity ENCODER8 is
   port (A: in std_logic_vector (7 downto 0);
         Y: out std_logic_vector (2 downto 0));
end entity ENCODER8;

architecture ARCH of ENCODER8 is
begin

-- 1: case statement
   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 ARCH;

-- 2: selected signal assignment
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 ARCH;
```

In both cases, three macrocells (one each for Y<1>, Y<2>, and Y<0>), twelve product terms, and eleven pins are used:

```
Design Name: encoder8
Device Used: XC9536XL –5 –PC44
Fitting Status: Successful
*************************  Resource Summary  *************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
3 /36 (8%)	12 /180 (6%)	0 /36 (0%)	11 /34 (32%)	16 /108 (14%)

; Implemented Equations.

```
"y<1>"  =  /"a<0>" * /"a<1>" * "a<2>" * /"a<3>" * /"a<4>" *
 /"a<5>" * /"a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * "a<3>" * /"a<4>" *
 /"a<5>" * /"a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 /"a<5>" * "a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 /"a<5>" * /"a<6>" * "a<7>"

"y<2>"  =  /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * "a<4>" *
 /"a<5>" * /"a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 "a<5>" * /"a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 /"a<5>" * "a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 /"a<5>" * /"a<6>" * "a<7>"

"y<0>"  =  /"a<0>" * "a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 /"a<5>" * /"a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * "a<3>" * /"a<4>" *
 /"a<5>" * /"a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 "a<5>" * /"a<6>" * /"a<7>"
 + /"a<0>" * /"a<1>" * /"a<2>" * /"a<3>" * /"a<4>" *
 /"a<5>" * /"a<6>" * "a<7>"
```

An additional standard encoder is the “priority encoder” which permits multiple asserted inputs. VHDL code for priority encoders is not presented but the operation 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.

Decoders

The truth table for a 3:8 decoder is shown in Table 2. Note the reverse relationship to Table 1.

Table 2: **Truth Table, 3:8 decoder**

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

Like the encoder, this decoder can be modeled by an **if, case** statements along with selected signal assignment. When inputs and outputs become wide, **for** statements should be used for code efficiency. However, all the models synthesize to the same circuit.

3:8 Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Decoder3_8 is
    Port ( A : in integer range 0 to 7;
           Y : out std_logic_vector(7 downto 0));
end Decoder3_8;

architecture behavioral of Decoder3_8 is
begin

--1: Case statement
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;
end behavioral;

--2: Select Signal Assignment
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;
end behavioral;

-- 3: for statement
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;

end behavioral;
```

Again, the corresponding result summary follows:

```
Design Name: encoder8
Device Used: XC9536XL –5 –PC44
Fitting Status: Successful
*************************  Resource Summary  *************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
8 /36 (22%)	8 /180 (4%)	0 /36 (0%)	11 /34 (32%)	6 /108 (5%)

Eight macrocells are used, one for each of the individual bits of Y, and the design is combinational. The implemented equations are:

; Implemented Equations.

```
"y<0>"  =  /"a<0>" * /"a<1>" * /"a<2>"
"y<1>"  =  "a<0>" * /"a<1>" * /"a<2>"
"y<2>"  =  /"a<0>" * "a<1>" * /"a<2>"
"y<3>"  =  "a<0>" * "a<1>" * /"a<2>"
"y<4>"  =  /"a<0>" * /"a<1>" * "a<2>"
"y<5>"  =  "a<0>" * /"a<1>" * "a<2>"
"y<6>"  =  /"a<0>" * "a<1>" * "a<2>"
"y<7>"  =  "a<0>" * "a<1>" * "a<2>"
```

Four Bit Address Decoder

The following is an example of a 4-Bit Address Decoder. It provides enable signals for segments of memory. The address map for this example is shown in the figure below.

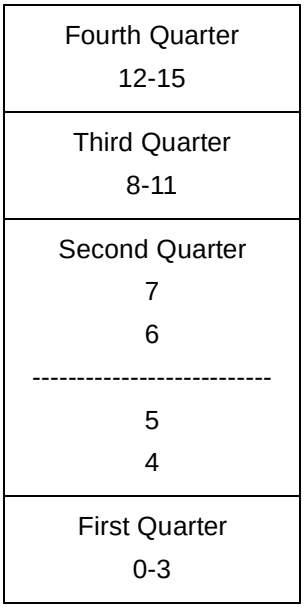

Figure 1: **4-bit Address Decoder Address Map**

The address map is divided into quarters. The second quarter is further divided into four segments. Thus, seven enable outputs (one for each memory segment) are provided. Two examples are provided below – one that uses the **for** loop enclosing an **if** statement, and the other uses a **case** statement. Both model the same circuit, but in general, it is much more efficient to use a **for** loop when a large number of consecutively decoded outputs are required. A **case** statement requires a separate branch for every output, thus increasing the lines of code.

Four Bit Address Decoder

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

entity decoder_4_bit is
    Port ( Address : in integer range 0 to 15;
           AddDec_0to3 : out std_logic;
           AddDec_8to11 : out st66d_logic;
           AddDec_12to15 : out std_logic;
           AddDec_4to7 : out std_logic_vector(3 downto 0));
end decoder_4_bit;

architecture behavioral of decoder_4_bit is
begin

-- 1: for loop
   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 behavioral;

--2: case statements
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 behavioral;
```

Here is the summary of the compiled results:

```
Design Name: Fb_decoder
Device Used: XC9536XL –5 –PC44
Fitting Status: Successful
*************************** Resource Summary ***************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
7 /36 (19%)	7 /180 (3%)	0 /36 (0%)	11 /34 (32%)	8 /108 (7%)

A total of seven equations have been mapped into two function blocks. Each of these seven equations occupies one macrocell. The six product terms used in this design can be seen in the Implemented equations:

```
; Implemented Equations.
 adddec_12to15  =  "address<2>" * "address<3>"
 "adddec_4to7<0>"  =  /"address<0>" * /"address<1>" * "address<2>" *
  /"address<3>"
 "adddec_4to7<1>"  =  "address<0>" * /"address<1>" * "address<2>" *
  /"address<3>"
 "adddec_4to7<2>"  =  /"address<0>" * "address<1>" * "address<2>" *
  /"address<3>"
 "adddec_4to7<3>"  =  "address<0>" * "address<1>" * "address<2>" *
  /"address<3>"
 adddec_8to11  =  /"address<2>" * "address<3>"
 adddec_0to3  =  /"address<2>" * /"address<3>"
```

Comparators

The code for a simple 6-bit equality comparator is shown in the example below. Comparators are modeled using the **if** statement with an **else** clause. A conditional signal assignment can also be used, but is less common as a sensitivity list cannot be specified to improve simulation.

The equality and relational operators in VHDL are:

```
=
!=
<
<=
>
>=
```

The logical operators are:

```
not
and
or
```

It is important to note that only two data objects can be compared at once. Thus, a statement like **if**(A=B=C) may not be used. Logical operators can, however, be used to test the result of multiple comparisons, such as **if**((A=B) **and** (A=C)).

Six-Bit Equality Comparator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.Numeric_STD.all;

entity Comparator6 is
    Port ( A1, B1, A2, B2, A3, B3 : in std_logic_vector(5 downto 0);
           Y1, Y2, Y3 : out std_logic);
end Comparator6;
```

```
architecture behavioral of Comparator6 is
begin
  COMPARE: process (A1, B1, A2, B2, A3, B3)
  begin
     Y1 <= '1';
   -- each bit is compared in a for loop
   for N in 0 to 5 loop
        if (A1(N) /= B1(N)) then
        Y1 <= '0';
    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 behavioral;
```

The corresponding resource summary is as follows when fit into a XC9572XL –TQ100 device:

```
Design Name: Comparator6
Device Used: XC9572XL –5 –TQ100
Fitting Status: Successful
*************************  Resource Summary  *************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
3 /72 (4%)	36 /360 (10%)	0 /72 (0%)	39 /72 (54%)	36 /216 (16%)

Three macrocells are used for this design although product terms are imported from the neighboring macrocells.

; Implemented Equations.

```
/y1  =  "a1<3>" * /"b1<3>"
  + /"a1<3>" * "b1<3>"
  + "a1<4>" * /"b1<4>"
  + "a1<2>" * /"b1<2>"
  + /"a1<2>" * "b1<2>"
;Imported pterms FB1_2
  + /"a1<4>" * "b1<4>"
  + "a1<5>" * /"b1<5>"
  + /"a1<5>" * "b1<5>"
  + /"a1<0>" * "b1<0>"
;Imported pterms FB1_4
  + "a1<0>" * /"b1<0>"
  + "a1<1>" * /"b1<1>"
  + /"a1<1>" * "b1<1>"

/y2  =  "a2<3>" * /"b2<3>"
  + /"a2<3>" * "b2<3>"
  + /"a2<2>" * "b2<2>"
  + "a2<4>" * /"b2<4>"
  + /"a2<4>" * "b2<4>"
```

```
;Imported pterms FB2_2
  + "a2<2>" * /"b2<2>"
  + "a2<0>" * /"b2<0>"
  + /"a2<0>" * "b2<0>"
  + /"a2<5>" * "b2<5>"
;Imported pterms FB2_4
  + "a2<1>" * /"b2<1>"
  + /"a2<1>" * "b2<1>"
  + "a2<5>" * /"b2<5>"
/y3  =  "a3<3>" * /"b3<3>"
  + /"a3<3>" * "b3<3>"
  + /"a3<2>" * "b3<2>"
  + "a3<4>" * /"b3<4>"
  + /"a3<4>" * "b3<4>"
;Imported pterms FB3_2
  + "a3<2>" * /"b3<2>"
  + "a3<0>" * /"b3<0>"
  + /"a3<0>" * "b3<0>"
  + /"a3<5>" * "b3<5>"
;Imported pterms FB3_4
  + "a3<1>" * /"b3<1>"
  + /"a3<1>" * "b3<1>"
  + "a3<5>" * /"b3<5>"
```

Adders

A dataflow model of a full adder is shown below. This is a single bit adder which can be easily extended using a parametric declaration.

Single Bit Half Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity half_adder is
    Port ( A, B : in std_logic;
           Sum, Cout : out std_logic);
end half_adder;

architecture behavioral of half_adder is
begin
   Sum <= A xor B;
   Cout <= A and B;
end behavioral;
```

Full Adder

```
Library IEEE;
Use IEEE.STD_Logic_1164.all, IEEE.Numeric_STD.all;

entity Fulladder is
    Port ( A, B, Cin : in std_logic;
           Cout, Sum : out std_logic);
end Fulladder;

architecture behavioral of Fulladder is
   component HALF_ADDER
     port (A, B: in std_logic;
          Sum, Cout: out std_logic);
   end component;
   signal AplusB, CoutHA1, CoutHA2: std_logic;
   begin
   HA1: HALF_ADDER port map (A=> A, B=> B, Sum => AplusB,
```

```
                              Cout => CoutHA1);
   HA2: HALF_ADDER port map (A => AplusB, B=> Cin,
                              Sum => Sum, Cout => CoutHA2);
   Cout <= CoutHA1 or CoutHA2;
end behavioral;
```

The corresponding resource summary is as follows when fit into an XC9572XL -TQ100 device:

```
Design Name: FULLADDER
Device Used: XC9572XL –5 –TQ100
Fitting Status: Successful
**************************  Resource Summary  **************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
2 /72 (2%)	6 /360 (1%)	0 /72 (0%)	5 /72 (6%)	6 /216 (2%)

Here is the corresponding implemented equations:

; Implemented Equations.

```
 cout  =  b * a
  + b * cin
  + a * cin

/sum  =  cin
  Xor b * a
  + /b * /a
```

Larger Adders can be defined behaviorally. The declaration statements are provided below.

Larger Adder Defined Behaviorally

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Fulladder is
    generic (parameter: integer := 10);
    Port ( A, B: in std_logic_vector (parameter downto 0)
           Cin : in std_logic;
           Cout: out std_logic;
           Sum : out std_logic_vector (parameter downto 0));
end Fulladder;
```

Modeling Synchronous Logic Circuits

The following example shows how to implement a 16-bit counter.

Sixteen Bit Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL; -- Two very useful
use IEEE.STD_LOGIC_UNSIGNED.ALL; -- IEEE libraries

entity sixteen is
    Port ( rst : in std_logic;
           clk : in std_logic;
           count: out std_logic_vector(15 downto 0));
end sixteen;

architecture behavioral of sixteen is
```

```
signal temp: std_logic_vector(15 downto 0);
begin
process (clk, rst)
   begin
   if (rst = '1') then
     temp <= "0000000000000000";
  elsif (clk'event and clk='1') then
     temp <= temp + 1;
  end if;
   end process;
  count <= temp;
end behavioral;
```

In this implementation 16 registers are used, namely Count0 through Count15. The corresponding resource summary is as shown below:

```
Design Name: Sixteen
Device Used: XC9536XL –5 –PC44
Fitting Status: Successful
**************************  Resource Summary  **************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
16 /36 (44%)	16 /180 (8%)	16 /36 (44%)	18 /34 (52%)	22 /108 (20%)

The 'Implemented equations' section of the fitter report, is given below. The equations were implemented with D-Type flops:

; Implemented Equations.

```
"count<0>"  :=  /"count<0>"
   "count<0>".CLKF  =  clk;FCLK/GCK
   "count<0>".RSTF  =  rst;GSR
   "count<0>".PRLD  =  GND

"count<1>".T  =  "count<0>"
   "count<1>".CLKF  =  clk;FCLK/GCK
   "count<1>".RSTF  =  rst;GSR
   "count<1>".PRLD  =  GND

"count<10>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>" *
 "count<7>" * "count<8>" * "count<9>"
   "count<10>".CLKF  =  clk;FCLK/GCK
   "count<10>".RSTF  =  rst;GSR
   "count<10>".PRLD  =  GND

"count<11>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>" *
 "count<7>" * "count<8>" * "count<9>" * "count<10>"
   "count<11>".CLKF  =  clk;FCLK/GCK
   "count<11>".RSTF  =  rst;GSR
   "count<11>".PRLD  =  GND

"count<12>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>" *
 "count<7>" * "count<8>" * "count<9>" * "count<10>" *
 "count<11>"
   "count<12>".CLKF  =  clk;FCLK/GCK
   "count<12>".RSTF  =  rst;GSR
   "count<12>".PRLD  =  GND
```

```
"count<13>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>" *
 "count<7>" * "count<8>" * "count<9>" * "count<10>" *
 "count<11>" * "count<12>"
   "count<13>".CLKF  =  clk;FCLK/GCK
   "count<13>".RSTF  =  rst;GSR
   "count<13>".PRLD  =  GND

"count<14>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>" *
 "count<7>" * "count<8>" * "count<9>" * "count<10>" *
 "count<11>" * "count<12>" * "count<13>"
   "count<14>".CLKF  =  clk;FCLK/GCK
   "count<14>".RSTF  =  rst;GSR
   "count<14>".PRLD  =  GND

"count<15>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>" *
 "count<7>" * "count<8>" * "count<9>" * "count<10>" *
 "count<11>" * "count<12>" * "count<13>" * "count<14>"
   "count<15>".CLKF  =  clk;FCLK/GCK
   "count<15>".RSTF  =  rst;GSR
   "count<15>".PRLD  =  GND

"count<2>".T  =  "count<0>" * "count<1>"
   "count<2>".CLKF  =  clk;FCLK/GCK
   "count<2>".RSTF  =  rst;GSR
   "count<2>".PRLD  =  GND

"count<3>".T  =  "count<0>" * "count<1>" * "count<2>"
   "count<3>".CLKF  =  clk;FCLK/GCK
   "count<3>".RSTF  =  rst;GSR
   "count<3>".PRLD  =  GND

"count<4>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>"
   "count<4>".CLKF  =  clk;FCLK/GCK
   "count<4>".RSTF  =  rst;GSR
   "count<4>".PRLD  =  GND

"count<5>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>"
   "count<5>".CLKF  =  clk;FCLK/GCK
   "count<5>".RSTF  =  rst;GSR
   "count<5>".PRLD  =  GND

"count<6>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>"
   "count<6>".CLKF  =  clk;FCLK/GCK
   "count<6>".RSTF  =  rst;GSR
   "count<6>".PRLD  =  GND

"count<7>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>"
   "count<7>".CLKF  =  clk;FCLK/GCK
   "count<7>".RSTF  =  rst;GSR
   "count<7>".PRLD  =  GND

"count<8>".T  =  "count<0>" * "count<1>" * "count<2>" *
 "count<3>" * "count<4>" * "count<5>" * "count<6>" *
 "count<7>"
```

```
    "count<8>".CLKF  =  clk;FCLK/GCK
    "count<8>".RSTF  =  rst;GSR
    "count<8>".PRLD  =  GND

 "count<9>".T  =  "count<0>" * "count<1>" * "count<2>" *
  "count<3>" * "count<4>" * "count<5>" * "count<6>" *
  "count<7>" * "count<8>"
    "count<9>".CLKF  =  clk;FCLK/GCK
    "count<9>".RSTF  =  rst;GSR
    "count<9>".PRLD  =  GND
```

The next example illustrates how to implement a 5-bit up by 1 down by 2 counter. This circuit counts up by 1 when the signal *Up* is a logic "1" and counts down by 2 when the signal *down* is logic "1". A **case** statement of the concatenation of *Up* and *Down* makes the model easy to read.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CNT_UP1_DOWN2 is
    Port ( Clock, reset, Up, Down : in std_logic;
           Count : out std_logic_vector(4 downto 0));
end CNT_UP1_DOWN2;

architecture behavioral of CNT_UP1_DOWN2 is
begin
   process (Clock)
     variable UpDown: std_logic_vector (1 downto 0);
    variable Count_V: std_logic_vector (4 downto 0);
  begin
     UpDown := Up & Down;
    if (clock'event and clock = '1')then
       if (Reset = '1') then
         Count_V := "00000";
      else
         case UpDown is
          when "00" => Count_V := Count_V;
          when "10" => Count_V := Count_V +1;
          when "01" => Count_V := Count_V -2;
          when others => Count_V := Count_V;
         end case;
      end if;
    end if;
     Count <= Count_V;
    end process;
end behavioral;
```

Note the utilization is three P-terms per bit.

```
Design Name: UPONEDOWNTWO
Device Used: XC9536XL –5  -PC44
Fitting Status: Successful
**************************  Resource Summary  **************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
5 /36 (13%)	15 /180 (8%)	5 /36 (13%)	9 /34 (26%)	14 /108 (12%)

All bits of the “count” signal have been automatically converted to T-type registers.

```
; Implemented Equations.
 "count<0>".T  =  reset * "count<0>"
  + /reset * /down * up
     "count<0>".CLKF  =  clock;FCLK/GCK
     "count<0>".PRLD  =  GND

/"count<1>".T  =  reset * /"count<1>"
  + /reset * /"count<0>" * /down
  + /reset * down * up
  + /reset * /down * /up
     "count<1>".CLKF  =  clock;FCLK/GCK
     "count<1>".PRLD  =  GND

 "count<2>".T  =  reset * "count<2>"
  + /reset * /"count<1>" * down * /up
  + /reset * "count<0>" * "count<1>" * /down * up
     "count<2>".CLKF  =  clock;FCLK/GCK
     "count<2>".PRLD  =  GND

 "count<3>".T  =  reset * "count<3>"
  + /reset * /"count<2>" * /"count<1>" * down * /up
  + /reset * "count<0>" * "count<2>" * "count<1>" *
  /down * up
     "count<3>".CLKF  =  clock;FCLK/GCK
     "count<3>".PRLD  =  GND

 "count<4>".T  =  reset * "count<4>"
  + /reset * /"count<2>" * /"count<3>" * /"count<1>" *
  down * /up
  + /reset * "count<0>" * "count<2>" * "count<3>" *
  "count<1>" * /down * up
     "count<4>".CLKF  =  clock;FCLK/GCK
     "count<4>".PRLD  =  GND
```

Asynchronous Counters

Asynchronous Counters are sometimes referred to as Ripple Counters. Each single flip-flop phase divides the input signal by two. The example below is of a Divide by 16 Clock divider using an asynchronous (ripple) approach. It has four ripple stages each consisting of a D-type flip-flop. Each of the flip-flops' Q-bar outputs is connected back to its D input. A fifth flip-flop is needed to synchronize the divided by 16 clock (Div16) to the source clock (Clock).

Divide by 16 Clock Divider Using an Asynchronous (ripple) Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Async_counter is
    Port ( Clk, Reset : in std_logic;
           Y : out std_logic);
end Async_counter;

architecture behavioral of Async_counter is
signal Div2, Div4, Div8, Div16: std_logic;
begin
   process (Clk, Reset, Div2, Div4, Div8)
  begin
     if (Reset = '0') then
       Div2 <= '0';
    elsif rising_edge(Clk) then
```

```
      Div2 <= not Div2;
    end if;

    if (Reset = '0') then
        Div4 <= '0';
    elsif rising_edge (Div2) then
      Div4 <= not Div4;
    end if;

    if (Reset = '0') then
      Div8 <= '0';
    elsif rising_edge (Div4) then
      Div8 <= not Div8;
    end if;

    if (Reset = '0') then
      Div16 <= '0';
    elsif rising_edge (Div8) then
      Div16 <= not Div16;
    end if;
    -- resynchronize back to clock
    if (Reset = '0') then
      Y <= '0';
    elsif rising_edge(clk) then
      Y <= Div16;
    end if;
  end process;
end behavioral;

Design Name: Asynch_adder
Device used: XC9536XL –5 –PC44
Fitting Status: Successful
*************************** Resource Summary ***************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
5 /36 (13%)	8 /180 (4%)	5 /36 (13%)	3 /34 (8%)	4 /108 (3%)

It uses only two macrocells and six product terms, and the implemented equations are as follows:

; Implemented Equations.

```
y  :=  div16
   y.CLKF  =  clk;FCLK/GCK
   y.RSTF  =  /reset;GSR
   y.PRLD  =  GND

div16  :=  /div16
   div16.CLKF  =  div8
   div16.RSTF  =  /reset;GSR
   div16.PRLD  =  GND

div2  :=  /div2
   div2.CLKF  =  clk;FCLK/GCK
   div2.RSTF  =  /reset;GSR
   div2.PRLD  =  GND

div4  :=  /div4
```

```
   div4.CLKF  =  div2
   div4.RSTF  =  /reset;GSR
   div4.PRLD  =  GND

div8  :=  /div8
   div8.CLKF  =  div4
   div8.RSTF  =  /reset;GSR
   div8.PRLD  =  GND
```

Finite State Machines

A Finite State Machine is a circuit specifically designed to cycle through a chosen sequence of operations (states) in a well defined manner. FSMs are an important aspect of hardware design. A well written model will function correctly and meet requirements in an optimal manner; a poorly written model may not. Therefore, a designer should fully understand and be familiar with different HDL modeling basics.

FSM Design and Modeling Issues

FSM issues to consider are:

- HDL coding style
- Resets and fail safe behavior
- State encoding
- Mealy or Moore type outputs

HDL coding style

There are many ways of modeling the same state machine. HDL code may be partitioned into three different portions to represent the three parts of a state machine (next state logic, current state logic, and output logic). It may also be structured so that the three different portions are combined in the model. For example, current state and next state logic may be combined with separate output logic, as shown in example FSM1; or next state and output logic may be combined with a separate current state logic, as shown in example FSM2. However, in VHDL, it is impossible to synthesize a combined current state, next state, and output logic in a single always statement.

A FSM with n state flip-flops may have 2^n binary numbers that can represent states. Often, all of the 2^n states are not needed. Unused states should be managed by specifying a fall back state during normal operation. For example, a state machine with six states requires a minimum of three flip-flops. Since $2^3 = 8$ possible states, there are two unused states. Therefore, Next-state logic is best modeled using the case statement even though this means the FSM can not be modeled in one process. The default clause used in a case statement avoids having to define these unused states.

Resets and fail safe behavior

Depending on the application, different types of resets may or may not be available. There may be a synchronous and an asynchronous reset, there may only be one, or there may be none. In any case, to ensure fail safe behavior, one of two things must be done, depending on the type of reset:

Use an asynchronous reset. This ensures the FSM is always initialized to a known state before the first clock transition and before normal operation commences. This has the advantage of minimizing the next state logic by not having to decode any unused current state values.

With no reset or a synchronous reset. When an asynchronous reset is unavailable, there is no way of predicting the initial value of the state register flip-flops when the IC is powered up. In the worst case scenario, it could power up and become stuck in an uncoded state. Therefore, all 2^n binary values must be decoded in the next state logic, whether they form part of the state machine or not.

In VHDL an asynchronous reset can only be modeled using the **if** statement, while a synchronous reset can be modeled using either a **wait** or **if** statement; the disadvantage of using the **wait** statement is that the whole process is synchronous so other **if** statements cannot be used to model purely combinational logic.

Table 3: **Asynchronous and Synchronous Reset**

Asynchronous Reset Example	***Synchronous Reset Example***

`Process (Clock, Reset)`
`Begin`
`If (Reset = '1') then`
`State <= ST0;`
`Elsif rising_edge (Clock) then`
`Case (State) is`
`…..`
`end case;`
`end if;`
`end process;`
	`Process (Clock, Reset)`
	`Begin`
	`If (rising_edge (clock) then`
	`If (Reset = '1') then`
	`State <= ST0;`
	`Else`
	`Case (State) is`
	`……`
	`end case;`
	`end if;`
	`end process;`

State Encoding

The way in which states are assigned binary values is referred to as state encoding. Some different state encoding schemes commonly used are:

- Binary
- Gray
- Johnson
- One-hot

Table 4: **State Encoding Format Values**

No.	Binary	Gray	Johnson	One-hot
0	000	000	000	001
1	001	001	001	010
2	010	011	011	100
3	011	010	111	1000
4	100	110		
5	101	111		
6	110	101		
7	111	100		

CPLDs, unlike FPGAs, have fewer flip-flops available to the designer. While one-hot encoding is sometimes preferred because it is easy, a large state machine will require a large number of flip-flops (*n* states will require *n* flops). Therefore, when implementing finite state machines on CPLDs, in order to conserve available resources, it is recommended that binary or gray encoding be used. Doing so enables the largest number of states to be represented by as few flip-flops as possible.

Mealy or Moore type outputs

There are generally two ways to describe a state machine – Mealy and Moore. A Mealy state machine has outputs that are a function of the current state and primary inputs. A Moore state machine has outputs that are a function of the current state only, and so includes outputs direct from the state register. If outputs come direct from the state register only, there is no output logic.

The examples below show the same state machine modeled with a Mealy or Moore type output. A state diagram is also associated with each of the two examples.

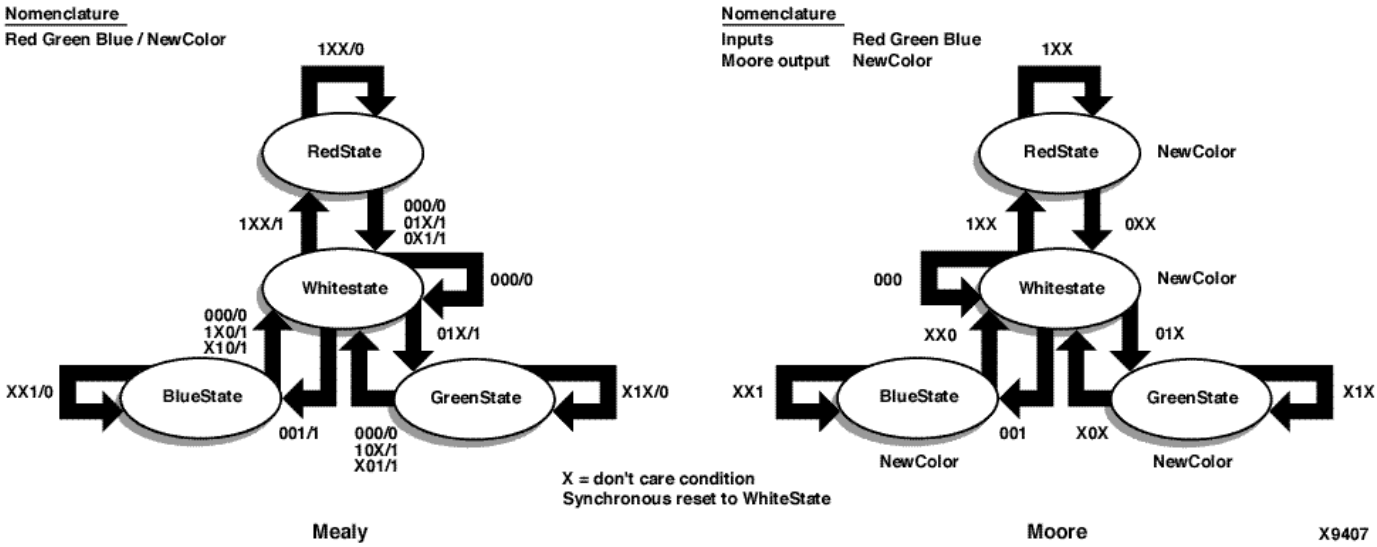

Figure 2: **State Machines**

FSM Example 1 – FSM modeled with "NewColor" as a Mealy type output

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FSMEX_1 is
    Port ( clock : in std_logic;
           reset : in std_logic;
           red, green, blue : in std_logic;
           NewColor : out std_logic);
end FSMEX_1;

architecture behavioral of FSMEX_1 is
type Color is (Redstate, GreenState, BlueState, WhiteState);
signal CurrentState, NextState: Color;
begin
   FSM_COMB: process (Red, Green, Blue, CurrentState)
  begin
     case CurrentState is
       when Redstate =>
         if (Red = '1') then
           NewColor <= '0';
           NextState <= RedState;
         else
           if (Green = '1' or Blue = '1') then
             NewColor <= '1';
           else
             NewColor <= '0';
```

```
            end if;
            NextState <= WhiteState;
           end if;

        when GreenState =>
          if (Green = '1') then
            NewColor <= '0';
            NextState <= GreenState;
          else
            if (Red = '1' or Blue = '1') then
              NewColor <= '1';
            else
              NewColor <= '0';
            end if;
              NextState <= WhiteState;
          end if;

        when BlueState =>
           if (Blue = '1') then
             NewColor <= '0';
          NextState <= BlueState;
         else
             if (Red = '1' or Green = '1') then
               NewColor <= '1';
             else
               NewColor <= '0';
          end if;
             NextState <= WhiteState;
         end if;

        when WhiteState =>
           if (Red = '1') then
             NewColor <= '1';
             NextState <= RedState;
         elsif (Green = '1') then
             NewColor <= '1';
             NextState <= GreenState;
         elsif (Blue = '1') then
             NewColor <= '1';
             NextState <= BlueState;
         else
             NewColor <= '0';
             NextState <= WhiteState;
         end if;
          when others =>
            NewColor <= '0';
         NextState <= WhiteState;
      end case;
    end process FSM_COMB;

    FSM_SEQ: process (clock, reset)
    begin
       if (Reset = '0') then
         CurrentState <= WhiteState;
     elsif rising_edge (Clock) then
         CurrentState <= NextState;
     end if;
    end process FSM_SEQ;

  end behavioral;
```

When fit into a XC9536XL device, here is the resource summary:

```
Design Name: FSM_EX1
Device used: XC9536XL –5 –PC44
Fitting Status: Successful
*************************  Resource Summary  *************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
3 /36 (8%)	8 /180 (4%)	2 /36 (5%)	6 /34 (17%)	5 /108 (4%)

FSM Example 2– FSM modeled with "NewColor" as a Moore type output

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FSM_EX_Moore is
    Port ( Clock, Reset : in std_logic;
           Red, Green, Blue : in std_logic;
           NewColor : out std_logic);
end FSM_EX_Moore;

architecture behavioral of FSM_EX_Moore is
type Color is (Redstate, GreenState, BlueState, WhiteState);
signal CurrentState, NextState: Color;
begin

   FSM_COMB: process (Red, Green, Blue, CurrentState)
  begin
      case CurrentState is
       when RedState =>
         NewColor <= '1';      -- Moore Output independent of Red, Green or
Blue
         if (Red = '1') then
           NextState <= RedState;
       else
           NextState <= WhiteState;
       end if;
       when GreenState =>
         NewColor <= '1';
             if (Green = '1') then
         NextState <= GreenState;
         else
         NextState <= WhiteState;
         end if;
       when BlueState =>
         NewColor <= '1';
         if (Blue = '1') then
         NextState <= BlueState;
         else
         NextState <= WhiteState;
         end if;
       when WhiteState =>
         NewColor <= '0';
         if (Red = '1') then
         NextState <= RedState;
         elsif (Green = '1') then
```

```
            NextState <= GreenState;
          elsif (Blue = '1') then
            NextState <= BlueState;
          else
            NextState <= WhiteState;
          end if;
        when others =>
          NewColor <= '0';
          NextState <= WhiteState;
        end case;
    end process FSM_COMB;

    FSM_SEQ: process (clock, reset)
       begin
         if (Reset = '0') then
           CurrentState <= WhiteState;
       elsif rising_edge (Clock) then
           CurrentState <= NextState;
       end if;
       end process FSM_SEQ;

end behavioral;


When fit into a 9536XL device, here is the resource summary:
Design Name: FSM_EX2
Device used: XC9536XL –5 –PC44
Fitting Status: Successful
*****************************  Resource Summary  *****************************
```

Macrocells Used	Product Terms Used	Registers Used	Pins used	Function Block Inputs Used
3 /36 (8%)	11 /180 (6%)	2 /36 (5%)	6 /34 (17%)	5 /108 (4%)

Coding Techniques

Efficient synthesis of VHDL often depends on how the design was coded. As distinct synthesis engines produce different results, leaving as little as possible to chance will increase the speed and regulate the density of your design. This, however, often trades off some of the advantages of using higher level constructs and libraries.

Compare Functions

Address decodes often require a decode of a range of address locations. It is inevitable to use the greater than or less than test. Wait state generation, however, often waits a known number of clock cycles. Consider this VHDL code.

```
when wait_state =>
  if wait_counter < wait_time then
    wait_counter <= wait_counter + 1;
    my_state <= wait_state;
  else
     my_state <= next_state;
  end if;
```

This generates extensive logic to implement the compare. A more efficient implementation would be to branch on the equality condition.

```
when wait_state =>
  if wait_counter = wait_time then
```

```
    my_state <= next_state;
  else
    wait_counter <= wait_counter + 1;
    my_state <= wait_state;
  end if;
```

Don’t Care Conditions

When writing VHDL, it is quite easy to forget about signals that are of no concern in the specific piece of code handling a certain function. For example, you may be generating a memory address for a memory controller that is important when your address strobes are active, however, these outputs are essentially don’t care conditions otherwise. Do not forget to assign them as such; otherwise, the VHDL synthesizer will assume that it should hold the output of the last known value.

```
when RAS_ADDRESS =>
  memory_address <= bus_address[31 downto 16];
    RAS <= ’0’;
    CAS <= ’1’;
    my_state <= CAS_ADDRESS;

when CAS_ADDRESS =>
  memory_address <= bus_address[15 downto 0];
    RAS <= ’0’;
    CAS <= ’0’;
    wait_count <= zero;
    my_state <= WAIT_1;

when WAIT_1 =>
    RAS <= ’0’;
    CAS <= ’1’;
  if wait_count = wait_length then
    my_state <= NEXT_ADDRESS;
  else
    wait_count <= wait_count + 1;
  end if;
```

This design can be implemented much more efficiently if it is coded as:

```
when RAS_ADDRESS =>
  memory_address <= bus_address[31 downto 16];
    RAS <= ’0’;
    CAS <= ’1’;
    wait_count <= "XXXX";
    my_state <= CAS_ADDRESS;

when CAS_ADDRESS =>
  memory_address <= bus_address[15 downto 0];
    RAS <= ’0’;
    CAS <= ’0’;
    wait_count <= "0000";
    my_state <= WAIT_1;

when WAIT_1 =>
  memory_address <= "XXXXXXXXXXXXXXXX";
    RAS <= ’0’;
    CAS <= ’1’;
  if wait_count = wait_length then
    my_state <= NEXT_ADDRESS;
  else
    wait_count <= wait_count + 1;
  end if;
```

Note that we add "don't care" for the wait_count register in the RAS_ADDRESS state as well as adding a "don't care" assignment for the memory address register in the WAIT_1 state. By specifying these don't cares, the final optimized implementation will be improved.

Using Specific Assignments

There is a temptation with VHDL to use language tricks to compact code. For example, using a counter to increment a state variable. While this allows you to write prompter and visually appealing code, it results in the synthesis tool generating more complex logic to implement the adder and trying to optimize the logic that is unused later. It is generally better to simply assign your desired bit pattern directly. This generates logic that is quicker to collapse during the subsequent fitter process.

Modularity

A designer can "rubber stamp" a design by instantiating multiple instances of an existing design entity. Component instantiation is basically the same as applying schematic macros. First, apply the COMPONENT declaration to define what the input and output ports are. The component declaration must match the actual entity declaration of the component. For example, if we want to reuse the flip-flop from our previous example in another design, we can declare it with:

```
component DFLOP port
     (
     my_clk        :in      std_logic;
     D_input       :in      std_logic;
     Q_output      :out     std_logic
     );
```

The component DFLOP can than be instantiated with the signals necessary to connect the component to the rest of the design. The signals can be mapped positionally or explicitly. Positional mapping is quicker to enter, but forbids omitting unnecessary logic. For example, if you had an 8-bit loadable counter that was never reloaded, explicit mapping allows you to omit signal assignment to the input ports and still use the same 8-bit counter definition.

To instantiate the component requires a unique label. Then we state the component name being instantiated followed by the positional signal assignments we are attaching to the component. An example of position signal mapping would be:

```
my_flop: DFF port map(clk, my_input, my_output);
```

The same flop mapped explicitly is shown below. Note that the order can be altered when mapping explicitly.

```
my_second_flop: DFLOP port map (my_clk => clk,
Q_output => my_other_output,
D_input => my_other_input);
```

Bidirectional Ports

To implement bidirectional ports, we must first define the port to be of type inout. Then, we must define when the port is driving, and when it is in a high-Z mode. This example implements this structure.

```
library ieee;
use ieee.std_logic_1164.all;
entity bidi is
  port
  (
  Data:inout std_logic_vector (7 downto 0);
```

```
  direction: in std_logic;
  clk:in std_logic
  );
end bidi;

architecture behavior of bidi is

signal my_register: std_logic_vector (7 downto 0);

begin
   process (direction, my_register)
   begin
     if (direction = '1') then
     Data <= "ZZZZZZZZ";
     else
     Data <= my_register;
     end if;
   end process;

   process (clk)
   begin
     if (clk'event and clk = '1') then
     my_register <= Data;
     end if;
   end process;
end behavior;
```

Summary

The basic structure of a VHDL design has been illustrated, along with numerous examples of basic building blocks. The examples provided are implemented using Xilinx Webpack ISE software. WebPACK ISE software is a free package that provides everything needed to implement a XC9500/XL/XV or CoolRunner design. Webpack not only supports VHDL, but also Verilog, ABEL, and EDIF netlist designs completely free of charge.

Webpack can be downloaded at **http://www.support.xilinx.com/sxpresso/webpack.htm**

Alternatively, the Xilinx WebFITTER, a web-based design evaluation tool that may also be used. WebFITTER accepts VHDL, Verilog, ABEL, EDIF, and XNF files and returns to the designer a fitter report and a JEDEC file to program the device.

WebFITTER may be accessed at **http://www.support.xilinx.com/sxpresso/webfitter.htm**

Should any problems arise, Xilinx support is available at **http://support.xilinx.com**

Revision History

The following table shows the revision history for this document

Date	Version	Revision
1/12/98	1.0	Initial Xilinx Release
8/30/01	2.0	Update