XILINX®

XAPP143 (v1.0) August 22, 2001

Using Verilog to Create CPLD Designs

Summary

This Application Note covers the basics of how to use Verilog as applied to Complex Programmable Logic Devices. Various combinational logic circuit examples, such as multiplexers, decoders, encoders, comparators and adders are provided. Synchronous logic circuit examples, such as counters and state machines are also provided.

Introduction

Verilog, like VHDL, can be used to describe logic circuits in many different levels of detail. This versatility allows many circuits to be described in more than one way. However, versatility can allow designers to describe certain circuits in a non-optimized manner. The following examples in this Application Note will show designers the best design practices when targeting Xilinx's 9500XL and 9500XV families. However, the approaches described are very appropriate for Xilinx CoolRunner CPLD families also.

This application note covers the following topics:

- Multiplexers
- Encoders
- Decoders
- Comparators
- Adders
- Counters
- Finite State Machines
- Latches
- Read-Back Registers
- Design Optimization

Multiplexers

Multiplexers can be modeled in many different ways. The two most common methods are to:

1. use one **if** statement followed by multiple **else if** statements
2. use a **case** statement.

In the example below, code for a 1-bit wide 4:1 multiplexer is shown.

There is no incorrect modeling method. However, **case** statements require less code than **if** statements and can be easier to read when inputs to the multiplexer increase.

One-bit wide 4:1 Mux

```
module mux4_1(Sel, A, B, C, D, Y);
input [1:0] Sel;
inputA,B,C,D;
output Y;
```

```
reg Y;
```

INSERT A OR B HERE

// A. If Statements

```
always @(Sel or A or B or C or D)
if (Sel == 2'b00)
Y=A;
else if (Sel == 2'b01)
Y=B;
else if (Sel == 2'b10)
Y=C;
else if (Sel == 2'b11)
Y=D;
endmodule
```

// B. Case Statements

```
always @(Sel or A or B or C or D)
case(Sel)
2'b00:    Y=A;
2'b01:    Y=B;
2'b10:    Y=C;
2'b11:    Y=D;
default:  Y=A;
endcase
endmodule
```

When compiled onto a 9536XL, the resulting usage is as follows:

Design Name: Mux41

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 9500 macrocell.

Seven pins are used in this design - A, B, C, D, Sel<0>, and Sel<1> are inputs, and Y is an output. The design is purely combinatorial, as 0 registers are used.

Four product terms are used. A closer look at the 'Implemented Equations' section of the Fitter Report will explain what these 4 product terms are:

```
; Implemented Equations.

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

2 bit wide 8:1 Mux

```
module mux81(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
```

In the example above, a 2 bit wide 8:1 multiplexer is implemented using **case** statements. This time, note that the **case** selector values are integers. In other words, the case selector value is 3 instead of 3’b011.

The resulting code gives the following usage summary:

```
Design Name: mux81                    Date:  8-10-2000, 10:52AM
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%)	6 /180 (8%)	10 /36 (0%)	21 /34 (61%)	22 /108 (20%)

This 8:1 multiplexer utilizes 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 (Function Block 2, Macrocell 4), respectively.

Y<0>, as shown in the ‘Implemented Equations’ section below, uses 8 Product Terms. The same is true for Y<1>. The architecture of the 9500 family is such that each macrocell has 5 local Product terms available to it. When more than 5 P-terms are needed, they may be borrowed from the neighboring macrocells above and/or below the local macrocell.

In this case, Y<0> and Y<1> borrow P-terms from macrocells above and below. For example, Y<0>, which resides in FB1_4, borrows 2 product terms from its neighbor above, FB1_3, and also borrows 1 product term from its neighbor below, FB1_5.

Since each of these two equations uses 8 Product Terms, and

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

+ "Sel<0>" * "A7<1>" * "Sel<1>" * "Sel<2>"

+ "Sel<0>" * /"Sel<1>" * "A5<1>" * "Sel<2>"

+ "Sel<0>" * /"Sel<1>" * /"Sel<2>" * "A1<1>"

+ /"Sel<0>" * /"Sel<1>" * "A4<1>" * "Sel<2>"

;Imported pterms FB2_3

+ "Sel<0>" * "Sel<1>" * /"Sel<2>" * "A3<1>"

+ /"Sel<0>" * /"Sel<1>" * /"Sel<2>" * "A0<1>"

;Imported pterms FB2_5

+ /"Sel<0>" * "Sel<1>" * /"Sel<2>" * "A2<1>"

Three Methods of Multiplexing

Three methods of multiplexing output signals are shown below. The method you choose depends on your application, resources and speed requirements. These methods are more efficient than using internal busses when implemented in a CPLD.

You can implement an output multiplexer by using three-state controls and tying the pins together off-chip. This uses more pins but you do not need a macrocell to implement the multiplexer as shown below.

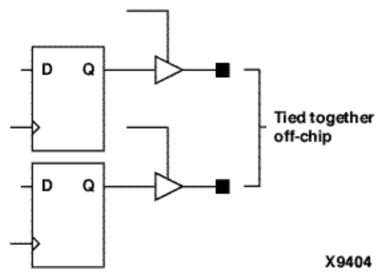

Figure 1: **Output Multiplexer Implemented Using Three-State Controls**

A multiplexer can be implemented in another macrocell and bring the data out though a single pin. This saves pins but costs a macrocell to implement each bit of the multiplexer as shown below.

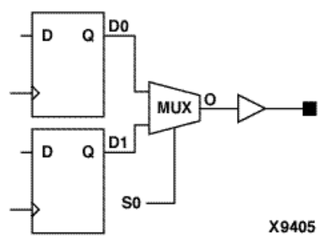

Figure 2: **Mulitplexer Implemented Saving Pin**

A register on the output macrocell can be used to shorten the clock-to-output delay as shown below. This is a form of data pipelining in that the the resulting data takes two clocks to reach the output pin.

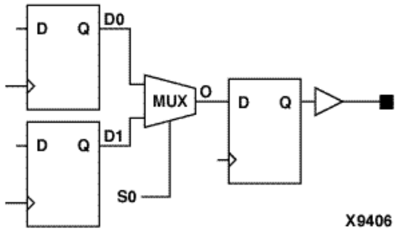

Figure 3: **Multiplexer Implemented with Shortened Clock-to-Output Delay**

The following code shows how to multiplex signals in Verilog:

Using macrocell logic for a conventional multiplexer:

```
always@(A or SEL or B or C or D)
    begin
    case(SEL)
    00:
        muxout=A;
    01:
        muxout=B;
    10:
        muxout=C;
    11:
        muxout=D;
    endcase
    end
```

Using 3-state outputs to multiplex registers fast

```
always@(posedge CLK)
    begin
        REG_A = DATA_A;
        REG_B =DATA_B;
    end
assign DOUT_A = (SEL ==0)?REG_A:1'bz;
assign DOUT_B = (SEL ==1)?REG_B:1'bz;
```

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 using an **if** statement, a **case** statement, or a **for** statement. Again, **case** statements are more concise and easier to read than **if** statements and this becomes increasingly true when the number of inputs to the encoder increase.

The **for** statement also models this encoder but it becomes particularly useful when the inputs to the encoder are very wide.

This 8:3 encoder only specifies 8 of the 256 (2^8) input values. In order to reduce logic created by the synthesis tools, a "don't care" condition must be included ensuring the synthesis tools do not create extra logic.

An 8:3 Binary Encoder

```
module ENCODER8_3(A,Y);
input [7:0] A;
output [2:0] Y;

reg [2:0] Y;
```

INSERT A OR B OR C

// A. If statements

```
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'bX;
end
endmodule
```

// B. Case Statement

```
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'bX;

endcase
end
endmodule
```

// C. For Loop

```
reg [7:0] Test;
integer [2:0] N;

always @(A)
begin
Test=8'b00000001;
Y=3'bX;
for(N=0; N < 8; N = N+1)
begin
if(A == Test)
```

```
Y=N;
Test = Test << 1;
end
end
endmodule
```

In all 3 cases,

*********************** Resource Summary****************************

In all 3 cases, 3 macrocells (one each for Y<1>, Y<2> and Y<0>), 12 product terms (as shown in the ‘Implemented Equations’), and 11 pins are used:

Design Name: encoder83 Date: 8-10-2000, 11:23AM

Device Used: XC9536XL-5-PC44

Fitting Status: Successful

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 addtional standard encoder is the "priority encoder" which permits multiple asserted inputs. Verilog code for priority encoders is not presented but is straightforward.

Decoders

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

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

Inputs			Outputs							
A0	**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

This decoder can be modeled by an **if** statement, a **case** statement, or a **for** statement. **Case** statements are often used for clarity. When inputs and outputs become very wide, **for** statements should be used for code efficiency. However, all three models synthesize to the same circuit.

“Don’t care” conditions do not have to be specified in this example. All 8 input conditions (2^3) are specified.

3:8 Decoder

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

reg [7:0] Y;
```

INSERT A, B, OR C HERE

// A. if statement

```
always @(A)
begin
if (A==0)Y=8'b00000001;
else if (A==1)Y=8'b00000010;
else if (A==2)  Y=8'b00000100;
else if (A==3)  Y=8'b00001000;
else if (A==4)  Y=8'b00010000;
else if (A==5)  Y=8'b00100000;
else if (A==6)  Y=8'b01000000;
else if (A==7)  Y=8'b10000000;
elseY=8'b10000000;
end
endmodule
```

// B. Case Statement

```
always @(A)
begin
casex(A)
0: Y=8'b00000001;
1: Y=8'b00000010;
2: Y=8'b00000100;
3: Y=8'b00001000;
4: Y=8'b00010000;
5: Y=8'b00100000;
6: Y=8'b01000000;
7: Y=8'b10000000;
default: Y=8'bX;
endcase
end
endmodule
```

// C. For Loop

```
integer N;

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

Again, the corresponding result summary follows:

```
Fitter Report
Design Name: decoder38                    Date:  8-10-2000, 11:31AM
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 8 individual bits of Y, and the design is combinatorial.

The Implemented Equations are:

```
"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 Four Bit Address Decoder. It provides enable signals for segments of memory. The address map for this example is shown in figure below.

Fourth Quarter 12-15
Third Quarter 8-11
Second Quarter 7 6 -------------------- 5 4
First Quarter 0-3

Figure 4: **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 a **for** loop enclosing an **if** statement, and another that uses a **case** statement. Both model the same circuit, but as a general rule, it is better to use a **for** loop enclosing an **if** statement when a large number of consecutively decoded outputs is needed. A **Case** statement requires a separate branch for every output, thus increasing the amount of code.

Four Bit address Decoder

```
module adddec
(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;
```

// INSERT A OR B

// A. For Loop and If statement

```
integer N;
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
```

// B. Case Statements

```
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_12to15 = 1;
endcase
end
endmodule
```

As before, the following summarizes the compile results:

```
Design Name: adddec                    Date:  8-10-2000, 11:37AM
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%)	6/180 (3%)	0/36 (0%)	11/34 (32%)	8/108 (7%)

From the information in the Fitter Report, a total of 7 equations have been mapped into 2 function blocks. Each of these 7 equations occupies 1 macrocell.

The 6 Product Terms used in this design can be seen in the Implemented Equations:

```
; Implemented Equations.
 AddDec_0to3  =  /"Address<2>" * /"Address<3>"
 AddDec_12to15  =  /"Address<2>" * "Address<3>"
 "AddDec_4to7<0>"  =  /"Address<0>" * "Address<2>" * /"Address<1>" *
/"Address<3>"
 "AddDec_4to7<1>"  =  "Address<0>" * "Address<2>" * /"Address<1>" *
/"Address<3>"
 "AddDec_4to7<2>"  =  /"Address<0>" * "Address<2>" * "Address<1>" *
```

```
/"Address<3>"
 "AddDec_4to7<3>"  =  "Address<0>" * "Address<2>" * "Address<1>" *
/"Address<3>"
 AddDec_8to11  =  Gnd
```

Comparators

The code for a simple 6-bit equality comparator is shown in the example below. Comparators are only modeled using the **if** statement with an **else** clause. Any two data objects are compared using equality and relational operators in the expression part of the **if** statement.

The equality operators in verilog are:

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

The logical operators are:

```
!
&&
||
```

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) && (A = C)).

// 6 Bit Equality Comparator

```
module comparator (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 (A1 == B3)
```

```
Y3=1;
else
Y3=0;
end
endmodule
```

The corresponding resource summary shows:

Design Name: counter16 Date: 8-10-2000, 12:04PM

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%)	31/180 (17%)	16/36 (44%)	18/34 (52%)	26/108 (24%)

Sixteen bit counters require 16 flip-flops. Sixteen macrocells are used, because each Macrocell in a 9500 has one flip-flop associated with it.

In the ‘Implemented Equations’ section of the Fitter Report, note that count<0> and count<1> are D-Type flip flops. All other flip flops have been automatically converted to T-flops as designated by a “.T” following the name of the equation.

Also note that all 16 registers are clocked by the signal “clk” and are preloaded to ‘0’.

```
; Implemented Equations.

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

 "count<1>"  :=  /rst * "count<0>" * /"count<1>"
+ /rst * /"count<0>" * "count<1>"
    "count<1>".CLKF  =  clk;FCLK/GCK
    "count<1>".PRLD  =  GND

 "count<10>".T  =  rst * "count<10>"
+ /rst * "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>".PRLD  =  GND
```

```
 "count<11>".T  =  rst * "count<11>"
+ /rst * "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>".PRLD  =  GND

 "count<12>".T  =  rst * "count<12>"
+ /rst * "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>".PRLD  =  GND

 "count<13>".T  =  rst * "count<13>"
+ /rst * "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>".PRLD  =  GND

 "count<14>".T  =  rst * "count<14>"
+ /rst * "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>".PRLD  =  GND

 "count<15>".T  =  rst * "count<15>"
+ /rst * "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>".PRLD  =  GND

 "count<2>".T  =  rst * "count<2>"
```

```
+ /rst * "count<0>" * "count<1>"
    "count<2>".CLKF  =  clk;FCLK/GCK
    "count<2>".PRLD  =  GND

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

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

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

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

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

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

```
 "count<9>".T  =  rst * "count<9>"
+ /rst * "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>".PRLD  =  GND
```

Adders

A dataflow model of a full adder is shown below. This is a single bit adder which can be easily extended as we will see.

Full Adder

```
module FullAdder(a,b,cin,sum,cout) ;
input a,b,cin ;
output sum,cout ;
wire s ;
assign s = a ^ b ;
assign sum = s ^ cin ;
assign cout = (a & b) | (s & cin) ;
endmodule
```

The adder resultant summary is as shown:

```
Design Name: comparator                    Date:  8-10-2000, 11:46AM
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%)	36/180 (20%)	0/36 (0%)	33/34 (97%)	30/108 (27%)

The 3 macrocells used in this design come from Y1, Y2, and Y3. Each of the 3 equations, as shown below, use 12 product terms. A 9500 macrocell has 5 local product terms associated with it, and in order to implement an equation requiring 12 P-terms, the 9500 architecture allows for borrowing P-terms from either macrocells above, or from macrocells below the local macrocell. As shown below, all three equations borrow P-terms from their associated neighbors.

```
; Implemented Equations.
```

```
/Y1  =  "A1<4>" * /"B1<4>"
+ /"A1<4>" * "B1<4>"
+ "A1<5>" * /"B1<5>"
+ /"A1<5>" * "B1<5>"
+ /"A1<1>" * "B1<1>"
;Imported pterms FB1_3
+ /"A1<0>" * "B1<0>"
+ "A1<2>" * /"B1<2>"
+ /"A1<2>" * "B1<2>"
+ "A1<1>" * /"B1<1>"
;Imported pterms FB1_5
+ "A1<3>" * /"B1<3>"
+ /"A1<3>" * "B1<3>"
+ "A1<0>" * /"B1<0>"

/Y2  =  "A2<3>" * /"B2<3>"
+ /"A2<3>" * "B2<3>"
+ "A2<2>" * /"B2<2>"
+ /"A2<2>" * "B2<2>"
+ /"A2<0>" * "B2<0>"
;Imported pterms FB2_3
+ "A2<0>" * /"B2<0>"
+ "A2<5>" * /"B2<5>"
+ /"A2<5>" * "B2<5>"
+ /"A2<4>" * "B2<4>"
;Imported pterms FB2_5
+ "A2<1>" * /"B2<1>"
+ /"A2<1>" * "B2<1>"
+ "A2<4>" * /"B2<4>"

/Y3  =  "A1<3>" * /"B3<3>"
+ /"A1<3>" * "B3<3>"
+ "A1<4>" * /"B3<4>"
+ /"A1<4>" * "B3<4>"
+ /"A1<2>" * "B3<2>"
;Imported pterms FB1_10
+ /"A1<0>" * "B3<0>"
+ "A1<2>" * /"B3<2>"
+ "A1<1>" * /"B3<1>"
+ /"A1<1>" * "B3<1>"
```

```
;Imported pterms FB1_12
+ "A1<5>" * /"B3<5>"
+ /"A1<5>" * "B3<5>"
+ "A1<0>" * /"B3<0>"
```

Larger adders may be defined behaviorally as shown below. Note that this example makes use of **parameter**. Parameters are constants allowing a module to be customized at compile time. The larger adder is parameterized at 8 bits.

Larger Adder Defined Behaviorally

```
module adder (sum, a, b) ;
parameter size = 8 ;
input [size - 1:0] a ;
input [size - 1:0] b ;
output [size - 1:0] sum ;
assign sum = a + b ;
endmodule
```

Modeling Synchronous Logic Circuits

Synchronous Counters

-------------------------FROM WEBPACK-----------------------------------

The following example shows how to implement high speed, up, down, and bidirectional counters. The Xilinx CPLD fitter implements the counters as D or T type registers to minimize the product term requirements.

```
always@(posedge CLK or posedge CLEAR)
begin
    if(CLEAR)
        COUNT=1'b0;
    else //if(CLK)
        begin
            if(LOAD)
                COUNT = DIN;
            else
                begin
                    if(CE)
                        begin
                            if(UP) //Up Counter
                                COUNT=COUNT+1;
                            else
```

```
                COUNT=COUNT-1; //Down Counter
              end
            end
          end
        end
```

A synchronous incrementing or decrementing binary counter is modeled by adding or subtracting a constant 1 using the “+” or “-“ operators in the section of code inferring the synchronous logic.

A Sixteen bit counter is shown in the example below.

Sixteen Bit Counter

```
module counter16(rst,clk,count) ;
input rst ;
input clk ;
output [15:0] count ;
reg [15:0] count ;
always @ (posedge clk)

begin
if (rst)
count = 16'b0 ;
else
count = count + 1 ;
end
endmodule
```

In this implementation, 5 registers are used, namely for Div2, Div4, Div8, Div16 and Y:

The corresponding resource summary shows:

```
Design Name: asynccntr                    Date:  8-10-2000,  2:52PM
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%)	5/108 (4%)

Looking at the 'Implemented Equations' section of the fitter report, the equations were implemented with D-Type flops:

```
; Implemented Equations.

 Div16  :=  /Div4
   Div16.CLKF  =  Div8
   Div16.RSTF  =  /Reset;GSR
   Div16.PRLD  =  GND

 Div2  :=  /Div2
   Div2.CLKF  =  Clock;FCLK/GCK
   Div2.RSTF  =  /Reset;GSR
   Div2.PRLD  =  GND

 Div4  :=  /Div4
   Div4.CLKF  =  Div2
   Div4.RSTF  =  /Reset;GSR
   Div4.PRLD  =  GND

 Div8  :=  /Div4
   Div8.CLKF  =  Div4
   Div8.RSTF  =  /Reset;GSR
   Div8.PRLD  =  GND

 Y  :=  Div16
   Y.CLKF  =  Clock;FCLK/GCK
   Y.RSTF  =  /Reset;GSR
   Y.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. For all other conditions of *Up* and *Down*, the counter will hold its value.

A **case** statement of the concatenation of *Up* and *Down* makes the model easy to read.

A 5 bit Up by 1 down by 2 Counter

```
module CNT_UP1_DOWN2(Clock, Reset, Up, Down, Count);
input Clock, Reset, Up, Down;
output [4:0] Count;
reg[4:0] Count;
```

```
reg [1:0] UpDown;

always @(posedge Clock)
begin
if (Reset)
Count=0;
else
case ({Up, Down})
2'b00 :  Count = Count;
2'b10 :  Count = Count + 1;
2'b01 :  Count = Count - 2;
default: Count = Count;
endcase
end
endmodule
```

Note the utilization is three p-terms per bit:

```
Design Name: updowncntr                    Date:  8-10-2000,  2:30PM
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 called 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

```
module asynccntr(Clock, Reset, Y);
input Clock, Reset;
output Y;

reg Div2, Div4, Div8, Div16, Y;

always @(posedge Clock or negedge Reset)
if(!Reset)
```

```
Div2 = 0;
else
Div2 = !Div2;

always @(posedge Div2 or negedge Reset)
if(!Reset)
Div4 = 0;
else
Div4 = !Div4;

always @(posedge Div4 or negedge Reset)
if(!Reset)
Div8 = 0;
else
Div8 = !Div4;

always @(posedge Div8 or negedge Reset)
if(!Reset)
Div16 = 0;
else
Div16 = !Div4;

//Resynchronize back to Clock

always @(posedge Clock or negedge Reset)
if(!Reset)
Y=0;
else
Y=Div16;
endmodule
```

This full adder fits into a 9536XL as shown below:

```
Design Name: fulladder              Date:  8-10-2000, 11:59AM
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%)	6/180 (3%)	0/36 (0%)	5/34 (14%)	6/108 (5%)

It utilizes only 2 macrocells and 6 Product terms, and the implemented equations are as follows:

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

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

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 issues.

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 Verilog, 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, not all of the 2^n states are needed. Unused states should be managed so that they do not occur 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 my 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 Verilog a synchronous or an asynchronous reset can only be modeled using an **if** statement, and if asynchronous, it must be included in the event list of the **always** statement with the **posedge** or **negedge** clause.

Asynchronous Reset Example #1

```
always @(posedge clock or posedge Reset)
begin
if(!Reset)
CurrentState = ST0;
else
CurrentState = NextState;
end
```

Asynchronous Reset Example #2

```
always @(posedge Clock or negedge Reset)
begin
if(!Reset)
State = ST0;
Else
Case(State)
ST0:if(A)
State = ST0;
else
State = ST1;
endcase
end
```

Synchronous Reset Example #1

```
always @(posedge Clock)
begin
if (!Reset)CurrentState = ST0;
elseCurrentState = ST1;
end
```

State Encoding

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

- Binary
- Gray
- Johnson
- One-hot

Table 3: **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	
4	100	110		
5	101	111		
6	110	101		
7	111	100		

CPLD's, 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 CPLD's, in order to conserve available resources, it is recommended that some type of binary 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 3 and 4 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.

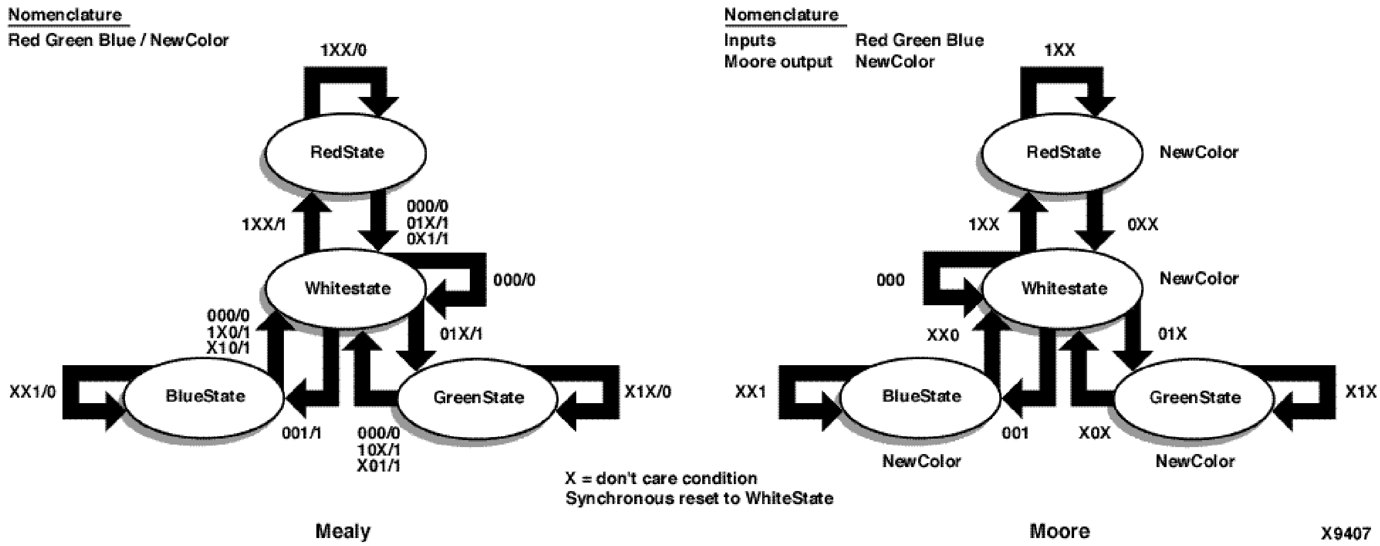

Figure 5: **State Diagram**

FSM Example 1 -- Combined current state, next state logic and output logic

```
module FSM1 (Clock, Reset, Control, Y);
input Clock, Reset, Control;
output [2:0] Y;
reg [2:0] Y;

parameter [1:0] ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
reg [1:0] Current State, NextState;

//Next State Logic:

always @(Control or CurrentState)
begin
NextState = ST0;
case (CurrentState)

ST0:begin
NextState = ST1;
   end
ST1:begin
if(Control)
NextState = ST2;
else
```

```
NextState = ST3;
    end
ST2:begin
NextState = ST3;
       end
ST3:begin
NextState = ST0;
       end
endcase
end

//Current State Logic:

always @(posedge Clock or posedge Reset)
begin
if(Reset)
CurrentState = ST0;
else
CurrentState = NextState;
end

//Output Logic

always @(CurrentState)
begin
case (CurrentState)
ST0: Y = 1;
ST1: Y = 2;
ST2: Y = 3;
ST3: Y = 4;
default: Y = 1; //default clause required to avoid latch inference
endcase
end

endmodule
```

FSM Example 2 Combined current state and next state logic, separate output logic

```
module FSM2 (Clock, Reset, Control, Y);
input Clock, Reset, Control;
```

```
output [2:0] Y;
reg [2:0] Y;

parameter [1:0] ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
reg [1:0] STATE;

//Current State and Next State Logic:

always @(posedge Clock or posedge Reset)
begin
if (Reset)
STATE = ST0;

else
case (STATE)
ST0: STATE = ST1;
ST1: if (Control)
STATE = ST2;
      else
  STATE = ST3;
ST2: STATE = ST3;
ST3: STATE = ST0;
endcase
end

//Output Logic:

always @(STATE)
begin
case (STATE)
ST0:  Y = 1;
ST1:  Y = 2;
ST2:  Y = 3;
ST3:  Y = 4;
default:  Y =1;//default required to avoid inferring a latch
end
endmodule
```

FSM Modeled with “NewColor” as a Mealy type output

```
Module FSM_MEALY
```

```
(Clock, Reset, Red, Green, Blue, NewColor);
input Clock, Reset
input Red, Green, Blue;
output NewColor;
reg NewColor
parameter RedState = 2'b00,
GreenState  = 2'b01,
BlueState = 2'b10,
WhiteState = 2'b11;
Reg (1:0) CurrentState, NextState;

Always @ (Red or Green or Blue or CurrentState)
Begin: FSM_COMB
Case(CurrentState)
RedState:
if(Red)
begin
NewColor=0;
NextState=RedState;
end
else
begin
if(Green || Blue)
NewColor = 1;
else
NewColor =0;
NextState = WhiteState;
end

GreenState:
if(Green)
begin
NewColor = 0;
NextState = GreenState;
end
else
begin
if(Red || Blue)
NewColor = 1;
NextState = WhiteState;
```

```
end

BlueState:
if(Blue)
begin
NewColor = 0;
NextState = BlueState;
end
else
begin
if(Red || Green)
NewColor = 1;
NextState = WhiteState;
end

WhiteState:
if(Red)
begin
NewColor = 1;
NextState = RedState;
end
else if (Green)
begin
NewColor = 1;
NextState = GreenState;
end
else if (Blue)
begin
NewColor = 1;
NextState = BlueState;
end
else
begin
NewColor = 0;
Nextstate = WhiteState;
end
default:
NextState = WhiteState;
endcase
end
```

```
always @ (posedge Clock or negedge Reset)
begin: FSM_SEQ
if (!Reset)
CurrentState = WhiteState;
else
CurrentState = NextState;
end
endmodule
```

FSM modeled with “NewColor” as a Moore type output

```
module FSM_MOORE(Clock, Reset, Red, Green, Blue, NewColor);
input Clock, Reset, Red, Green, Blue;
output NewColor;
reg NewColor;

parameter RedState = 2’b00,
GreenState = 2’b01,
BlueState = 2’b10,
WhiteState = 2’b11;

reg (1:0) CurrentState, NextState;

always @ (Red or Green or Blue or CurrentState)
begin: FSM_COMB
case (CurrentState)
RedState:
NewColor = 1;
if (Red)
NextState = RedState;
else
NextState = WhiteState;
GreenState:
NewColor = 1;
if (Green)
NextState = GreenState;
else
NextState = WhiteState;
```

```
BlueState:
NewColor = 1;
if (Blue)
NextState = BlueState;
else
NextState = WhiteState;

WhiteState:
NewColor = 0;
if (Red)
NextState = RedState;
else if (Green)
NextState = GreenState;
else if (Blue)
NextState = BlueState;
else
NextState = WhiteState;
default:
NextState= WhiteState;
endcase
end

always @ (posedge Clock or negedge Reset)
begin: FSM_SEQ
if(!Reset)
CurrentState = WhiteState;
else
CurrentState = NextState;
end
endmodule
```

Bidirectional Signals

The following example shows how to implement a bidirectional bus.

```
assign inout=(enable)?out:1'bz;
assign in=inout;
```

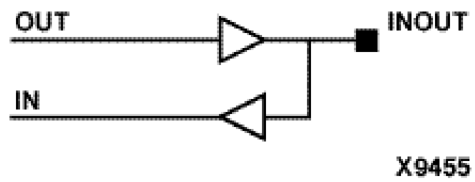

Figure 6: **Bidirectional Bus**

Latches

You can create registered latches that use the macrocell register set and reset product terms or combinatorial latches that use sum-term logic and feedback loops.

Registered latches use fewer resources than combinatorial latches when the data being latched and the latch enable signal generate only single product term set and reset functions as shown below:

The following code shows how to create registered latches using Verilog.

```
always@(GATE,DIN)
    begin
    if(GATE)
        DOUT =DIN;
    end
```

The following picture is associated with the code above:

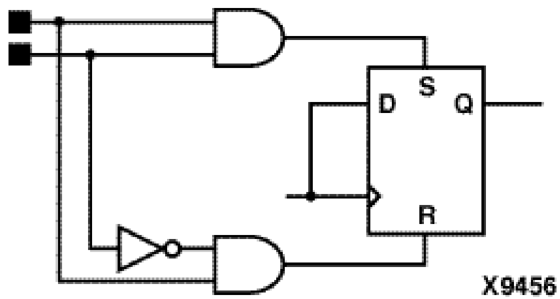

Figure 7: **Latch**

In this case, when a latch is behaviorally expressed, a latch macro (LD) get inferred that consists of a macrocell flop with the latch's D and G inputs gated into the flop's CLR and PRE inputs. This may not be optimal if there is any logic on the D or G inputs since there is only 1 p-term available for CLR and PRE in the XC9500 architecture. Additional logic would require feedback from additional MCs.

An alternative would be to define a combinational feedback latch using Boolean expressions [q=(d&g)|(q&!g)|(d&q)]. The redundant term (d&q) is present in order to prevent a logic hazard. The 'NOREDUCE' attribute must be added to 'q' in order to prevent the fitter from optimizing it out.

Creating Read-Back Registers

This example shows how to implement a simple read-back register. Data is written from the I/O pad to the register on the rising edge of clock if READ_ENABLE is inactive and WRITE_ENABLE is active. Data is read from the I/O pad when READ_ENABLE is active.

The following drawing shows how the code is implemented in the device:

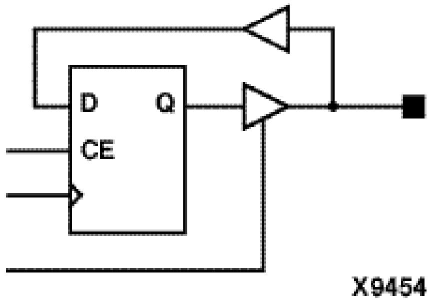

Figure 8: **Readback Register**

The following code shows how to create a readback register using Verilog

```
always@(posedge clk)
    begin
    if(write_enable)
        data_reg_int = data_reg;
        //write data from the I/O pin into the register
    end
assign data_reg = (read_enable==1)?data_reg_int:1'bz;
//Drive I/O pin when read_enable is high
```

DESIGN OPTIMIZATION

Using Global Nets

Clock, output enable, and register set and reset control signals can be implemented either on special globally routed nets or as ordinary signals through the p-terms. Control signals assigned to global nets are faster and do not use up function block resources. By default, the fitter assigns input pads used as control signals to global nets whenever possible. If you want to prevent the fitter from automatically using global nets, disable the appropriate option in the Process Properties dialog of the Project Navigator, or, if you are using the Design Manager, the Implementation Options window. You may also assign global nets or in the UCF for Verilog designs. Global nets can be assigned to input pads or internal nets. When a global net is assigned to an internal net, the CPLD Fitter will create an I/O pad for the net so that it can use the global resources.

Assigning Global Nets Manually in VHDL or Verilog

In Verilog designs, apply the BUFG attribute to an input pad or internal net in the UCF file using the following syntax:

```
NET net_name BUFG=CLK|OE|SR;
```

Summary

The Verilog examples presented in the Application Note should assist designers who are targeting XC9500XL and XC9500XV devices. Note that these examples have all been implemented using Xilinx WebPACK ISE software, a free package that provides everything needed to implement a XC9500/XL/XV or CoolRunner design. WebPACK not only supports Verilog, but also supports VHDL, ABEL, EDIF, and XNF designs completely free of charge.

WebPACK may be downloaded at *http://www.xilinx.com/sxpresso/webpack.htm*

Alternatively, the Xilinx WebFITTER, a web-based design evaluation tool 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.xilinx.com/sxpresso/webfitter.htm*

Should any problems arise, web-based support is located at *http://support.xilinx.com/support/support.htm*

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
8/22/01	1.0	Initial Xilinx release.