# Xilinx HDL Coding Hints

HDLs contain many complex constructs that are difficult to understand at first. Also, the methods and examples included in HDL manuals do not always apply to the design of FPGAs. If you currently use HDLs to design ASICs, your established coding style may unnecessarily increase the number of gates or CLB levels in FPGA designs.

HDL synthesis tools implement logic based on the coding style of your design. To learn how to efficiently code with HDLs, you can attend training classes, read reference and methodology notes, and refer to synthesis guidelines and templates available from Xilinx and the synthesis vendors. When coding your designs, remember that HDLs are mainly hardware description languages. You should try to find a balance between the quality of the end hardware results and the speed of simulation.

The coding hints and examples included in this chapter are not intended to teach you every aspect of VHDL or Verilog, but they should help you develop an efficient coding style.

The following sections are included in this chapter.

- "Comparing Synthesis and Simulation Results"
- "Selecting HDL Formatting Styles"
- "Using Schematic Design Hints with HDL Designs"

## Comparing Synthesis and Simulation Results

VHDL and Verilog are hardware description and simulation languages that were not originally intended as input to synthesis. Therefore, many hardware description and simulation constructs are not supported by synthesis tools. In addition, the various synthesis tools use different subsets of VHDL and Verilog. VHDL and Verilog semantics are well defined for design simulation. The synthesis tools must adhere to these semantics to ensure that designs simulate the same way before and after synthesis. Follow the guidelines presented below to create code that simulates the same way before and after synthesis.

## Omit the Wait for XX ns Statement

Do not use the Wait for XX ns statement in your code. XX specifies the number of nanoseconds that must pass before a condition is executed. This statement does not synthesize to a component. In designs that include this statement, the functionality of the simulated design does not match the functionality of the synthesized design. VHDL and Verilog examples of the Wait for XX ns statement are as follows.

- VHDL

  ```
  wait for XX ns;
  ```

- Verilog

  ```
  #XX;
  ```

## Omit the ...After XX ns or Delay Statement

Do not use the ...After XX ns statement in your VHDL code or the Delay assignment in your Verilog code. Examples of these statements are as follows.

- VHDL

  ```
  (Q <=0 after XX ns)
  ```

- Verilog

  ```
  assign #XX Q=0;
  ```

XX specifies the number of nanoseconds that must pass before a condition is executed. This statement is usually ignored by the synthesis tool. In this case, the functionality of the simulated design does not match the functionality of the synthesized design.

## Use Case and If-Else Statements

You can use If-Else statements, Case statements, or other conditional code to create state machines or other conditional logic. These statements implement the functions differently, however, the simulated designs are identical. The If-Else statement generally specifies priority-encoded logic and the Case statement generally specifies balanced behavior. The If-Else statement can, in some cases, result in a slower circuit overall. These statements vary with the synthesis tool. Refer to the "Comparing If Statement and Case Statement" section of this chapter for more information.

## Order and Group Arithmetic Functions

The ordering and grouping of arithmetic functions can influence design performance. For example, the following two VHDL statements are not necessarily equivalent.

```
ADD <= A1 + A2 + A3 + A4;
ADD <= (A1 + A2) + (A3 + A4);
```

For Verilog, the following two statements are not necessarily equivalent.

```
ADD = A1 + A2 + A3 + A4;
ADD = (A1 + A2) + (A3 + A4);
```

The first statement cascades three adders in series. The second statement creates two adders in parallel: A1 + A2 and A3 + A4. In the second statement, the two additions are evaluated in parallel and the results are combined with a third adder. RTL simulation results are the same for both statements, however, the second statement results in a faster circuit after synthesis (depending on the bit width of the input signals).

Although the second statement generally results in a faster circuit, in some cases, you may want to use the first statement. For example, if the A4 signal reaches the adder later than the other signals, the first statement produces a faster implementation because the cascaded structure creates fewer logic levels for A4. This structure allows A4 to catch up to the other signals. In this case, A1 is the fastest signal followed by A2 and A3; A4 is the slowest signal.

Most synthesis tools can balance or restructure the arithmetic operator tree if timing constraints require it. However, Xilinx recommends that you code your design for your selected structure.

### Omit Initial Values

Do not assign signals and variables initial values because initial values are ignored by most synthesis tools. The functionality of the simulated design may not match the functionality of the synthesized design.

For example, do not use initialization statements like the following VHDL and Verilog statements.

- VHDL

  ```
  variable SUM:INTEGER:=0;
  ```

- Verilog

  ```
  wire SUM=1'b0;
  ```

## Selecting HDL Formatting Styles

Because HDL designs are often created by design teams, Xilinx recommends that you agree on a style for your code at the beginning of your project. An established coding style allows you to read and understand code written by your fellow team members. Also, inefficient coding styles can adversely impact synthesis and simulation, which can result in slow circuits. Additionally, because portions of existing HDL designs are often used in new designs, you should follow coding standards that are understood by the majority of HDL designers. This section of the manual provides a list of suggested coding styles that you should establish before you begin your designs.

### Selecting a Capitalization Style

Select a capitalization style for your code. Xilinx recommends using a consistent style (lower or upper case) for entity or module names in FPGA designs.

### Verilog

For Verilog, the following style is recommended.

- Use lower case letters for the following.
  - Module names
  - Verilog language keywords
- Use upper case letters for the following.
  - Labels
  - Reg, wire, instance, and instantiated cell names

**Note:** Cell names must be upper case to use the UNISIM simulation library and certain synthesis libraries. Check with your synthesis vendor.

### VHDL

**Note:** VHDL is case-insensitive.

For VHDL, use lower case for all language constructs from the IEEE-STD 1076. Any inputs defined by you should be upper case. For example, use upper case for the names of signals, instances, components, architectures, processes, entities, variables, configurations, libraries, functions, packages, data types, and sub-types. For the names of standard or vendor packages, the style used by the vendor or uppercase letters are used, as shown for IEEE in the following example:

```
library IEEE;
use IEEE.std_logic_1164.all;
signal SIG: UNSIGNED (5 downto 0);
```

## Using Xilinx Naming Conventions

Use the Xilinx naming conventions listed in this section for naming signals, variables, and instances that are translated into nets, buses, and symbols.

**Note:** Most synthesis tools convert illegal characters to legal ones.

- User-defined names can contain A–Z, a–z, $, _, –, <, and >. A “/” is also valid, however, it is not recommended because it is used as a hierarchy separator

- Names must contain at least one non-numeric character
- Names cannot be more than 256 characters long

The following FPGA resource names are reserved and should not be used to name nets or components.

- Components (Comps), Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), basic elements (bels), clock buffers (BUFGs), tristate buffers (BUFTs), oscillators (OSC), CCLK, DP, GND, VCC, and RST
- CLB names such as AA, AB, and R1C2
- Primitive names such as TD0, BSCAN, M0, M1, M2, or STARTUP
- Do not use pin names such as P1 and A4 for component names
- Do not use pad names such as PAD1 for component names

Refer to the language reference manual for Verilog or VHDL for language-specific naming restrictions. Xilinx does not recommend using escape sequences for illegal characters. Also, if you plan on importing schematics into your design, use the most restrictive character set.

## Matching File Names to Entity and Module Names

The VHDL or Verilog source code file name should match the designated name of the entity (VHDL) or module (Verilog) specified in your design file. This is less confusing and generally makes it easier to create a script file for the compilation of your design. Xilinx also recommends that if your design contains more than one entity or module, each should be contained in a separate file with the appropriate file name. It is also a good idea to use the same name as your top-level design file for your synthesis script file with either a .do, .scr, .script, or the appropriate default script file extension for your synthesis tool.

## Naming Identifiers, Types, and Packages

You can use long (256 characters maximum) identifier names with underscores and embedded punctuation in your code. Use meaningful names for signals and variables, such as CONTROL_REGISTER. Use meaningful names when defining VHDL types and packages as shown in the following examples.

```
type LOCATION_TYPE is ...;
package STRING_IO_PKG is
```

## Using Labels

Use labels to group logic. Label all processes, functions, and procedures as shown in the following examples. Labeling makes it easier to debug your code.

- VHDL

  ```
  ASYNC_FF: process (CLK,RST)
  ```

- Verilog

  ```
  always @ (posedge CLK or posedge RST)
  begin: ASYNC_FF
  ```

### Labeling Flow Control Constructs

You can use optional labels on flow control constructs to make the code structure more obvious, as shown in the following VHDL and Verilog examples. However, you should note that these labels are not translated to gate or register names in your implemented design. Flow control constructs can slow down simulations in some Verilog simulators.

- VHDL Example

```
-- D_REGISTER.VHD
-- May 1997

-- Changing Latch into a D-Register

library IEEE;
use IEEE.std_logic_1164.all;

entity d_register is
    port (CLK, DATA: in STD_LOGIC;
          Q: out STD_LOGIC);
end d_register;

architecture BEHAV of d_register is
begin
My_D_Reg: process (CLK, DATA)
    begin
        if (CLK'event and CLK='1') then
```

```
                Q <= DATA;
            end if;
        end process; --End My_D_Reg
    end BEHAV;
```

- Verilog Example

```
/* Changing Latch into a D-Register
 * D_REGISTER.V
 * May 1997                                        */

module d_register (CLK, DATA, Q);

input CLK;
input DATA;
output Q;

reg Q;

    always @ (posedge CLK)
    begin: My_D_Reg
              Q <= DATA;
    end

endmodule
```

## Using Variables for Constants (VHDL Only)

Do not use variables for constants in your code. Define constant numeric values in your code as constants and use them by name. This coding convention allows you to easily determine if several occurrences of the same literal value have the same meaning. In some simulators, using constants allows greater optimization. In the following code example, the OPCODE values are declared as constants, and the constant names refer to their function. This method produces readable code that may be easier to modify.

### Using Constants to Specify OPCODE Functions (VHDL)

```
constant ZERO   : STD_LOGIC_VECTOR (1 downto 0):=“00”;
constant A_AND_B: STD_LOGIC_VECTOR (1 downto 0):=“01”;
constant A_OR_B : STD_LOGIC_VECTOR (1 downto 0):=“10”;
constant ONE    : STD_LOGIC_VECTOR (1 downto 0):=“11”;
```

```
process (OPCODE, A, B)
begin
  if    (OPCODE = A_AND_B)then OP_OUT <= A and B;
  elsif (OPCODE = A_OR_B) then OP_OUT <= A or B;
  elsif    (OPCODE = ONE) then OP_OUT <= '1';
  else                         OP_OUT <= '0';
  end if;
end process;
```

## Using Parameters for Constants (Verilog Only)

You can specify a constant value in Verilog using the *parameter* special data type, as shown in the following examples. The first example includes a definition of OPCODE constants as shown in the previous VHDL example. The second example shows how to use a parameter statement to define module bus widths.

### Using Parameters to Specify OPCODE Functions (Verilog)

```
parameter ZERO = 2'b00;
parameter A_AND_B = 2'b01;
parameter A_OR_B = 2'b10;
parameter ONE = 2'b11;

always @ (OPCODE or A or B)
begin
  if     (OPCODE==ZERO)    OP_OUT=1'b0;
  else if(OPCODE==A_AND_B) OP_OUT=A&B;
  else if(OPCODE==A_OR_B)  OP_OUT=A|B;
  else                     OP_OUT=1'b1;
end
```

### Using Parameters to Specify Bus Size (Verilog)

```
parameter BUS_SIZE = 8;

output [BUS_SIZE-1:0] OUT;
input [BUS_SIZE-1:0] X,Y;
```

## Using Named and Positional Association

Use positional association in function and procedure calls, and in port lists only when you assign all items in the list. Use named association when you assign only some of the items in the list. Also, Xilinx

suggests that you use named association to prevent incorrect connections for the ports of instantiated components. Do not combine positional and named association in the same statement as illustrated in the following examples.

- VHDL

| | |
|---|---|
| Incorrect | `CLK_1: BUFGS port map (I=>CLOCK_IN,CLOCK_OUT);` |
| Correct | `CLK_1: BUFGS port map (I=>CLOCK_IN,O=>CLOCK_OUT);` |

- Verilog

| | |
|---|---|
| Incorrect | `BUFGS CLK_1 (.I(CLOCK_IN), CLOCK_OUT);` |
| Correct | `BUFGS CLK_1 (.I(CLOCK_IN), .O(CLOCK_OUT));` |

## Managing Your Design

As part of your coding specifications, you should include rules for naming, organizing, and distributing your files. In VHDL designs, use explicit configurations to control the selection of components and architectures that you want to compile, simulate, or synthesize. In some synthesis tools, configuration information is ignored. In this case, you only need to compile the architecture that you want to synthesize.

## Creating Readable Code

Use the recommendations in this section to create code that is easy to read.

### Indenting Your Code

Indent blocks of code to align related statements. You should define the number of spaces for each indentation level and specify whether the Begin statement is placed on a line by itself. In the examples in this manual, each level of indentation is four spaces and the Begin statement is on a separate line that is not indented from the previous line of code. The examples below illustrate the indentation style used in this manual.

- VHDL Example

```
-- D_LATCH.VHD
-- May 1997

library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
    port (GATE, DATA: in STD_LOGIC;
          Q: out STD_LOGIC);
end d_latch;

architecture BEHAV of d_latch is
begin
LATCH: process (GATE, DATA)
    begin
    if (GATE = '1') then
        Q <= DATA;
    end if;
    end process; -- end LATCH

end BEHAV;
```

- Verilog Example

```
/* Transparent High Latch
 * D_LATCH.V
 * May 1997                                          */

module d_latch (GATE, DATA, Q);

input GATE;
input DATA;
output Q;

reg Q;

    always @ (GATE or DATA)
    begin: LATCH
        if (GATE == 1'b1)
            Q <= DATA;
    end  // End Latch

endmodule
```

## Using Empty Lines

Use empty lines to separate top-level constructs, designs, architectures, configurations, processes, subprograms, and packages.

## Using Spaces

Use spaces to make your code easier to read. You can omit or use spaces between signal names as shown in the following examples.

- VHDL Example

```
process (RST,CLOCK,LOAD,CE)
process (RST, CLOCK, LOAD, CE)
```

- Verilog Example

```
module test (A,B,C)
module test (A, B, C)
```

Use a space after colons as shown in the following examples.

- VHDL Example

```
signal QOUT: STD_LOGIC_VECTOR (3 downto 0);
CLK_1: BUFGS port map (I=>CLOCK_IN,O=>CLOCK_OUT);
```

- Verilog Example

```
begin: CPU_DATA
```

## Breaking Long Lines of Code

Break long lines of code at an appropriate point, such as at a comma, a colon, or a parenthesis to make your code easier to read, as illustrated in the following code fragments.

- VHDL Example

```
U1: load_reg port map
(INX=>A,LOAD=>LD,CLK=>SCLK,OUTX=>B);
```

- Verilog Example

```
load_reg U1
   (.INX(A), .LOAD(LD), .CLK(SCLK), .OUTX(B));
```

### Adding Comments

Add comments to your code to improve readability, reduce debugging time, and make it easier to maintain your code.

- VHDL Example

```
-- Read Counter (16-bit)
-- Updated 1-25-98 to add Clock Enable, John Doe
-- Updated 1-28-98 to add Terminal Count, Joe Cool

process (RST, CLOCK, CE)
begin
.
.
.
```

- Verilog Example

```
// Read Counter (16-bit)
// Updated 1-25-98 to add Clock Enable, John Doe
// Updated 1-28-98 to add Terminal Count, Joe Cool

always @ (posedge RST or posedge CLOCK)
begin
.
.
.
```

## Using Std_logic Data Type (VHDL only)

The Std_logic (IEEE 1164) type is recommended for hardware descriptions for the following reasons.

- It has nine different values that represent most of the states found in digital circuits.
- Automatically initialized to an unknown value. This automatic initialization is important for HDL designs because it forces you to initialize your design to a known state, which is similar to what is required in a schematic design. Do not override this feature by initializing signals and variables to a known value when they are declared because the result may be a gate-level circuit that cannot be initialized to a known value.
- Easy to perform a board-level simulation. For example, if you use an integer type for ports for one circuit and standard logic for ports for another circuit, your design can be synthesized;

however, you will need to perform time-consuming type conversions for a board-level simulation.

The back-annotated netlist from Xilinx implementation is in Std_logic. If you do not use Std_logic type to drive your top-level entity in the testbench, you cannot reuse your functional testbench for timing simulation. Some synthesis tools can create a wrapper for type conversion between the two top-level entities; however, this is not recommended by Xilinx.

## Declaring Ports

Xilinx recommends that you use the Std_logic package for all entity port declarations. This package makes it easier to integrate the synthesized netlist back into the design hierarchy without requiring conversion functions for the ports. A VHDL example using the Std_logic package for port declarations is shown below.

```
Entity alu is
   port( A : in STD_LOGIC_VECTOR(3 downto 0);
         B : in STD_LOGIC_VECTOR(3 downto 0);
         CLK : in STD_LOGIC;
         C : out STD_LOGIC_VECTOR(3 downto 0) );
end alu;
```

Since the downto convention for vectors is supported in a back-annotated netlist, the RTL and synthesized netlists should use the same convention if you are using the same test bench. This is necessary because of the loss of directionality when your design is synthesized to an EDIF or XNF netlist.

## Minimizing the Use of Ports Declared as Buffers

Do not use buffers when a signal is used internally and as an output port. In the following VHDL example, signal C is used internally and as an output port.

```
Entity alu is
   port( A : in STD_LOGIC_VECTOR(3 downto 0);
         B : in STD_LOGIC_VECTOR(3 downto 0);
         CLK : in STD_LOGIC;
         C : buffer STD_LOGIC_VECTOR(3 downto 0) );
end alu;

architecture BEHAVIORAL of alu is
begin
```

```
   process begin
      if (CLK'event and CLK='1') then
         C <= UNSIGNED(A) + UNSIGNED(B) + UNSIGNED(C);
      end if;
   end process;
end BEHAVIORAL;
```

Because signal C is used both internally and as an output port, every level of hierarchy in your design that connects to port C must be declared as a buffer. However, buffer types are not commonly used in VHDL designs because they can cause problems during synthesis. To reduce the amount of buffer coding in hierarchical designs, you can insert a dummy signal and declare port C as an output, as shown in the following VHDL example.

```
Entity alu is
   port( A : in STD_LOGIC_VECTOR(3 downto 0);
         B : in STD_LOGIC_VECTOR(3 downto 0);
         CLK : in STD_LOGIC;
         C : out STD_LOGIC_VECTOR(3 downto 0));
   end alu;

architecture BEHAVIORAL of alu is
-- dummy signal
signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
begin
   C <= C_INT;
   process begin
      if (CLK'event and CLK='1') then
      C_INT < =UNSIGNED(A) + UNSIGNED(B) +
         UNSIGNED(C_INT);

      end if;
   end process;
end BEHAVIORAL;
```

## Comparing Signals and Variables (VHDL only)

You can use signals and variables in your designs. Signals are similar to hardware and are not updated until the end of a process. Variables are immediately updated and, as a result, can effect the functioning of your design. Xilinx recommends using signals for hardware descriptions; however, variables allow quick simulation.

The following VHDL examples show a synthesized design that uses signals and variables, respectively. These examples are shown implemented with gates in the "Gate implementation of XOR_SIG" figure and the "Gate Implementation of XOR_VAR" figure.

**Note:** If you assign several values to a signal in one process, only the final value is used. When you assign a value to a variable, the assignment takes place immediately. A variable maintains its value until you specify a new value.

## Using Signals (VHDL)

```
-- XOR_SIG.VHD
-- May 1997
Library IEEE;
use IEEE.std_logic_1164.all;

entity xor_sig is
    port (A, B, C: in  STD_LOGIC;
          X, Y: out STD_LOGIC);
end xor_sig;

architecture SIG_ARCH of xor_sig is
    signal D: STD_LOGIC;
begin
    SIG:process (A,B,C)
    begin
        D <= A; -- ignored !!
        X <= C xor D;
        D <= B; -- overrides !!
        Y <= C xor D;
    end process;
end SIG_ARCH;
```

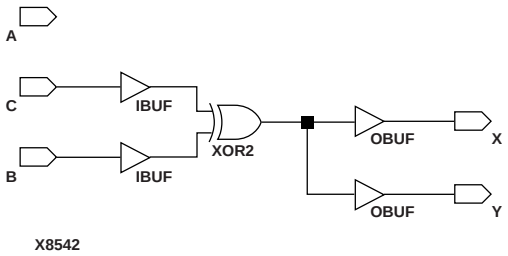


**Figure 1-1 Gate implementation of XOR_SIG**

### Using Variables (VHDL)

```
-- XOR_VAR.VHD
-- May 1997

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity xor_var is
    port (A, B, C: in  STD_LOGIC;
          X, Y:    out STD_LOGIC);
end xor_var;

architecture VAR_ARCH of xor_var is
begin

    VAR:process (A,B,C)
        variable D: STD_LOGIC;
    begin
        D := A;
        X <= C xor D;
        D := B;
        Y <= C xor D;
    end process;
end VAR_ARCH;
```

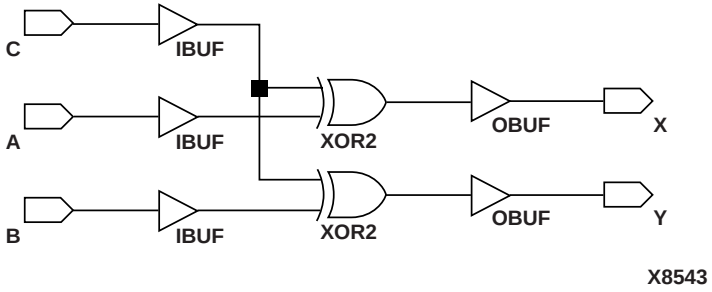


**Figure 1-2 Gate Implementation of XOR_VAR**

# Using Schematic Design Hints with HDL Designs

This section describes how to apply schematic entry design strategies to HDL designs.

## Barrel Shifter Design

The schematic version of the barrel shifter design is included in the "Multiplexers and Barrel Shifters in XC3000/XC3100" application note (XAPP 026.001) available on the Xilinx web site at http://www.xilinx.com. In this example, two levels of multiplexers are used to increase the speed of a 16–bit barrel shifter. This design is for XC3000 and XC3100 device families; however, it can also be used for other Xilinx devices.

The following VHDL and Verilog examples show a 16-bit barrel shifter implemented using sixteen 16–to–1 multiplexers, one for each output. A 16–to–1 multiplexer is a 20-input function with 16 data inputs and four select inputs. When targeting an FPGA device based on 4-input lookup tables (such as XC4000 and XC3000 family of devices), a 20-input function requires at least five logic blocks. Therefore, the minimum design size is 80 (16 x 5) logic blocks.

### 16-bit Barrel Shifter (VHDL)

```
-------------------------------------------------
-- VHDL Model for a 16-bit Barrel Shifter       --
-- barrel_org.vhd                               --
-- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! --
-- THIS EXAMPLE IS FOR COMPARISON ONLY          --
-- May 1997                                     --
-- USE barrel.vhd                               --
-------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity barrel_org is
   port (S:in    STD_LOGIC_VECTOR (3 downto 0);
         A_P:in    STD_LOGIC_VECTOR (15 downto 0);
         B_P:out   STD_LOGIC_VECTOR (15 downto 0));
end barrel_org;

architecture RTL of barrel_org is

begin
    SHIFT: process (S, A_P)
    begin
        case S is
```

```
when “0000” =>
   B_P <= A_P;

when “0001” =>
   B_P(14 downto 0)  <= A_P(15 downto 1);
   B_P(15) <= A_P(0);

when “0010” =>
   B_P(13 downto 0)  <= A_P(15 downto 2);
   B_P(15 downto 14) <= A_P(1 downto 0);

when “0011” =>
   B_P(12 downto 0)  <= A_P(15 downto 3);
   B_P(15 downto 13) <= A_P(2 downto 0);

when “0100” =>
   B_P(11 downto 0)  <= A_P(15 downto 4);
   B_P(15 downto 12) <= A_P(3 downto 0);

when “0101” =>
   B_P(10 downto 0)  <= A_P(15 downto 5);
   B_P(15 downto 11) <= A_P(4 downto 0);

when “0110” =>
   B_P(9 downto 0)   <= A_P(15 downto 6);
   B_P(15 downto 10) <= A_P(5 downto 0);

when “0111” =>
   B_P(8 downto 0)   <= A_P(15 downto 7);
   B_P(15 downto 9)  <= A_P(6 downto 0);

when “1000” =>
   B_P(7 downto 0)   <= A_P(15 downto 8);
   B_P(15 downto 8)  <= A_P(7 downto 0);

when “1001” =>
   B_P(6 downto 0)  <= A_P(15 downto 9);
   B_P(15 downto 7) <= A_P(8 downto 0);

when “1010” =>
   B_P(5 downto 0)  <= A_P(15 downto 10);
   B_P(15 downto 6) <= A_P(9 downto 0);

when “1011” =>
   B_P(4 downto 0)  <= A_P(15 downto 11);
   B_P(15 downto 5) <= A_P(10 downto 0);
```

```
                        when “1100” =>
                           B_P(3 downto 0)  <= A_P(15 downto 12);
                           B_P(15 downto 4) <= A_P(11 downto 0);

                        when “1101” =>
                          B_P(2 downto 0)  <= A_P(15 downto 13);
                          B_P(15 downto 3) <= A_P(12 downto 0);

                       when “1110” =>
                          B_P(1 downto 0)  <= A_P(15 downto 14);
                          B_P(15 downto 2) <= A_P(13 downto 0);

                       when “1111” =>
                          B_P(0) <= A_P(15);
                          B_P(15 downto 1) <= A_P(14 downto 0);

                       when others =>
                          B_P <= A_P;
                       end case;
                end process; -- End SHIFT

           end RTL;
```

## 16-bit Barrel Shifter (Verilog)

```
      ////////////////////////////////////////////////
     // BARREL_ORG.V Version 1.0                     //
    // Xilinx HDL Synthesis Design Guide            //
   // Unoptimized model for a 16-bit Barrel Shifter  //
  // THIS EXAMPLE IS FOR COMPARISON ONLY            //
 // Use BARREL.V                                   //
// January 1998                                   //
////////////////////////////////////////////////

module barrel_org (S, A_P, B_P);

    input  [3:0]  S;
    input  [15:0] A_P;
    output [15:0] B_P;

    reg    [15:0] B_P;

    always @ (A_P or S)
        begin
        case (S)
```

```
4'b0000 : // Shift by 0
  begin
  B_P <= A_P;
  end

4'b0001 : // Shift by 1
  begin
  B_P[15]   <= A_P[0];
  B_P[14:0] <= A_P[15:1];
  end

4'b0010 : // Shift by 2
  begin
  B_P[15:14] <= A_P[1:0];
  B_P[13:0]  <= A_P[15:2];
  end

4'b0011 : // Shift by 3
  begin
  B_P[15:13] <= A_P[2:0];
  B_P[12:0]  <= A_P[15:3];
  end

4'b0100 : // Shift by 4
   begin
   B_P[15:12] <= A_P[3:0];
   B_P[11:0]  <= A_P[15:4];
   end

4'b0101 : // Shift by 5
   begin
   B_P[15:11] <= A_P[4:0];
   B_P[10:0]  <= A_P[15:5];
   end

4'b0110 : // Shift by 6
   begin
   B_P[15:10] <= A_P[5:0];
   B_P[9:0]   <= A_P[15:6];
   end

4'b0111 : // Shift by 7
   begin
   B_P[15:9]  <= A_P[6:0];
   B_P[8:0]   <= A_P[15:7];
   end
```

```
4'b1000 : // Shift by 8
   begin
   B_P[15:8]  <= A_P[7:0];
   B_P[7:0]   <= A_P[15:8];
   end

4'b1001 : // Shift by 9
   begin
   B_P[15:7]  <= A_P[8:0];
   B_P[6:0]   <= A_P[15:9];
   end

4'b1010 : // Shift by 10
   begin
   B_P[15:6]  <= A_P[9:0];
   B_P[5:0]   <= A_P[15:10];
   end

4'b1011 : // Shift by 11
   begin
   B_P[15:5] <= A_P[10:0];
   B_P[4:0]  <= A_P[15:11];
   end

4'b1100 : // Shift by 12
   begin
   B_P[15:4] <= A_P[11:0];
   B_P[3:0]  <= A_P[15:12];
   end

4'b1101 : // Shift by 13
   begin
   B_P[15:3]  <= A_P[12:0];
   B_P[2:0]   <= A_P[15:13];
   end

4'b1110 : // Shift by 14
   begin
   B_P[15:2]  <= A_P[13:0];
   B_P[1:0]   <= A_P[15:14];
   end

4'b1111 : // Shift by 15
   begin
   B_P[15:1]  <= A_P[14:0];
   B_P[0]     <= A_P[15];
```

```
                end

             default :
                B_P        <= A_P;
         endcase
    end

endmodule
```

The following modified VHDL and Verilog designs use two levels of multiplexers and are twice as fast as the previous designs. These designs are implemented using 32 4–to–1 multiplexers arranged in two levels of sixteen. The first level rotates the input data by 0, 1, 2, or 3 bits and the second level rotates the data by 0, 4, 8, or 12 bits. Since you can build a 4–to–1 multiplexer with a single CLB, the minimum size of this version of the design is 32 (32 x 1) CLBs.

## 16-bit Barrel Shifter with Two Levels of Multiplexers (VHDL)

```
-- BARREL.VHD
-- Based on XAPP 26 (see http://www.xilinx.com)
-- 16-bit barrel shifter (shift right)
-- May 1997

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity barrel is
port (S:   in STD_LOGIC_VECTOR(3 downto 0);
A_P: in STD_LOGIC_VECTOR(15 downto 0);
B_P: out STD_LOGIC_VECTOR(15 downto 0));
end barrel;

architecture RTL of barrel is

signal SEL1,SEL2: STD_LOGIC_VECTOR(1 downto 0);
signal C:         STD_LOGIC_VECTOR(15 downto 0);

begin
    FIRST_LVL: process (A_P, SEL1)
    begin
        case SEL1 is
            when “00” => -- Shift by 0
```

```
                    C <= A_P;

                when “01” => -- Shift by 1
                    C(15) <= A_P(0);
                    C(14 downto 0) <= A_P(15 downto 1);

                when “10” => -- Shift by 2
                    C(15 downto 14) <= A_P(1 downto 0);
                    C(13 downto 0) <= A_P(15 downto 2);

                when “11” => -- Shift by 3
                    C(15 downto 13) <= A_P(2 downto 0);
                    C(12 downto 0) <= A_P(15 downto 3);

                when others =>
                    C <= A_P;
            end case;
        end process; --End FIRST_LVL

    SECND_LVL: process (C, SEL2)
        begin
            case SEL2 is
                when “00” => --Shift by 0
                    B_P <=  C;

                when “01” => --Shift by 4
                    B_P(15 downto 12) <= C(3 downto 0);
                    B_P(11 downto 0)  <= C(15 downto 4);

                when “10” => --Shift by 8
                    B_P(7 downto 0)   <= C(15 downto 8);
                    B_P(15 downto 8)  <= C(7 downto 0);

                when “11” => --Shift by 12
                   B_P(3 downto 0)   <= C(15 downto 12);
                   B_P(15 downto 4)  <= C(11 downto 0);

                when others =>
                    B_P <= C;
            end case;
        end process; -- End SECOND_LVL

        SEL1 <= S(1 downto 0);
        SEL2 <= S(3 downto 2);

    end rtl;
```

## 16-bit Barrel Shifter with Two Levels of Multiplexers (Verilog)

```
/**************************************
 * BARREL.V                           *
 * XAPP 26   http://www.xilinx.com    *
 * 16-bit barrel shifter [shift right]*
 * May 1997                           *
 **************************************/

module barrel (S, A_P, B_P);

input [3:0] S;
input [15:0] A_P;
output [15:0] B_P;

reg [15:0] B_P;

wire [1:0] SEL1, SEL2;
reg [15:0] C;

assign SEL1 = S[1:0];
assign SEL2 = S[3:2];

    always @ (A_P or SEL1)
           begin
        case (SEL1)
             2'b00 : // Shift by 0
                       begin
                       C <= A_P;
                       end

             2'b01 : // Shift by 1
                       begin
                       C[15] <= A_P[0];
                       C[14:0] <= A_P[15:1];
                       end

             2'b10 : // Shift by 2
                       begin
                       C[15:14] <= A_P[1:0];
                       C[13:0] <= A_P[15:2];
                       end

             2'b11 : // Shift by 3
```

```
                                begin
                                C[15:13] <= A_P[2:0];
                                C[12:0] <= A_P[15:3];
                                end

                       default :
                                C <= A_P;
                    endcase
                end

                always @ (C or SEL2)
                begin
                    case (SEL2)
                        2'b00 : // Shift by 0
                                begin
                                B_P <=  C;
                                end

                        2'b01 : // Shift by 4
                                begin
                                B_P[15:12] <= C[3:0];
                                B_P[11:0]  <= C[15:4];
                                end

                        2'b10 : // Shift by 8
                                begin
                                B_P[7:0] <= C[15:8];
                                B_P[15:8] <= C[7:0];
                                end

                        2'b11 : // Shift by 12
                                begin
                                B_P[3:0] <= C[15:12];
                                B_P[15:4] <= C[11:0];
                                end

                      default :
                                B_P <= C;
                    endcase
                end

            endmodule
```

When these two designs are implemented in an XC4005E-2 device with a popular synthesis tool, there is a 64% improvement in the gate count (88 occupied CLBs reduced to 32 occupied CLBs) in the

barrel.vhd design as compared to the barrel_org.vhd design. Additionally, there is a 19% improvement in speed from 35.58 ns (5 logic levels) to 28.85 ns (4 logic levels).

## Implementing Latches and Registers

Synthesizers infer latches from incomplete conditional expressions, such as an If statement without an Else clause. This can be problematic for FPGA designs because not all FPGA devices have latches available in the CLBs. In addition, you may think that a register is created, and the synthesis tool actually created a latch. The XC4000EX/XL and XC5200 FPGAs do have registers that can be configured to act as latches. For these devices, synthesizers infer a dedicated latch from incomplete conditional expressions. XC4000E, XC3100A, and XC3000A devices do not have latches in their CLBs. For these devices, latches described in RTL code are implemented with gates in the CLB function generators. For XC4000E devices, if the latch is directly connected to an input port, it is implemented in an IOB as a dedicated input latch. For example, the D latch described in the following VHDL and Verilog designs is implemented with one function generator as shown in the "D Latch Implemented with Gates" figure.

### D Latch Inference

- VHDL Example

```
-- D_LATCH.VHD
-- May 1997

library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
    port (GATE, DATA: in STD_LOGIC;
  Q: out STD_LOGIC);
end d_latch;

architecture BEHAV of d_latch is
begin
LATCH: process (GATE, DATA)
    begin
    if (GATE = '1') then
      Q <= DATA;
```

```
    end if;
end process; -- end LATCH

end BEHAV;
```

- Verilog Example

```
/* Transparent High Latch
 * D_LATCH.V
 * May 1997                                        */

module d_latch (GATE, DATA, Q);

input GATE;
input DATA;
output Q;

reg Q;

    always @ (GATE or DATA)
    begin
           if (GATE == 1'b1)
              Q <= DATA;
    end  // End Latch

endmodule
```

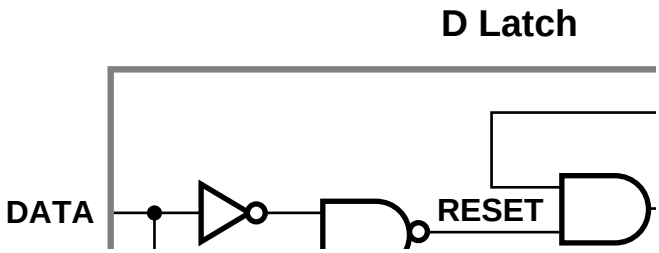


**Figure 1-3 D Latch Implemented with Gates**

In this example, a combinatorial loop results in a hold-time requirement on DATA with respect to GATE. Since most synthesis tools do not process hold-time requirements because of the uncertainty of routing delays, Xilinx does not recommend implementing latches

with combinatorial feedback loops. A recommended method for implementing latches is described in this section.

To eliminate this possible problem, use D registers instead of latches. For example, to convert the D latch to a D register, use an Else statement or modify the code to resemble the following example.

## Converting a D Latch to a D Register

- VHDL Example

```
-- D_REGISTER.VHD
-- May 1997

-- Changing Latch into a D-Register

library IEEE;
use IEEE.std_logic_1164.all;

entity d_register is
    port (CLK, DATA: in STD_LOGIC;
          Q: out STD_LOGIC);
end d_register;

architecture BEHAV of d_register is
begin
MY_D_REG: process (CLK, DATA)
    begin
    if (CLK'event and CLK='1') then
       Q <= DATA;
    end if;
    end process; --End MY_D_REG
end BEHAV;
```

- Verilog Example

```
/* Changing Latch into a D-Register
 * D_REGISTER.V
 * May 1997                          */

module d_register (CLK, DATA, Q);

input CLK;
input DATA;
output Q;
```

```
reg Q;

    always @ (posedge CLK)
    begin: My_D_Reg
     Q <= DATA;
    end

endmodule
```

With some synthesis tools you can determine the number of latches that are implemented in your design. Check the manuals that came with your software for information on determining the number of latches in your design.

You should convert all If statements without corresponding Else statements and without a clock edge to registers. Use the recommended register coding styles in the synthesis tool documentation to complete this conversion.

In XC4000E devices, you can implement a D latch by instantiating a RAM 16x1 primitive, as illustrated in the following figure.

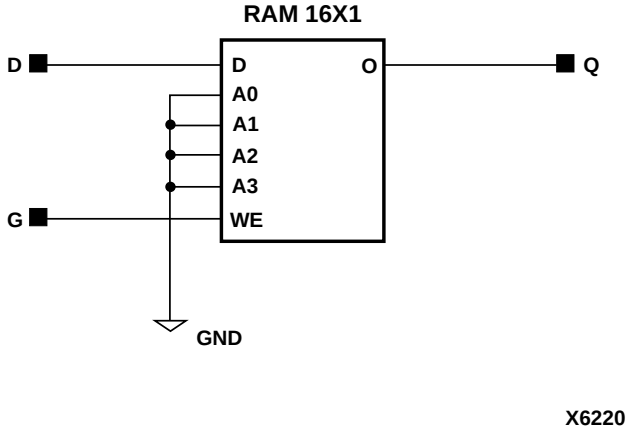


**Figure 1-4 D Latch Implemented by Instantiating a RAM**

In all other cases (such as latches with reset/set or enable), use a D flip-flop instead of a latch. This rule also applies to JK and SR flip-flops.

The following table provides a comparison of area and speed for a D latch implemented with gates, a 16x1 RAM primitive, and a D flip-flop.

**Table 1-5 D Latch Implementation Comparison**

| Comparison | Spartan, XC4000E CLB Latch Implemented with Gates | XC4000EX/ XL/XV, XC5200 CLB Latch | All Spartan and XC4000 Input Latch | XC4000 E/EX/XL/XV Instantiated RAM Latch | All Families D Flip Flop |
|---|---|---|---|---|---|
| Advantages | RTL HDL infers latch | RTL HDL infers latch, no hold times | RTL HDL infers latch, no hold times (if not specifying NODELAY, saves CLB resources) | No hold time or combinatorial loops, best for XC4000E when latch needed in CLB | No hold time or combinatorial loop. FPGAs are register abundant. |
| Disadvantages | Feedback loop results in hold time requirement, not suggested | Not available in XC4000E or Spartan | Not available in XC5200, input to latch must directly connect to port | Must be instantiated, uses logic resources | Requires change in code to convert latch to register |
| Area[a] | 1 Function Generator | 1 CLB Register / Latch | 1 IOB Register / Latch | 1 Function Generator | 1 CLB Register / Latch |

a.Area is the number of function generators and registers required. XC4000 CLBs have two function generators and two registers; XC5200 CLBs have four function generators and four register / latches.

## Resource Sharing

Resource sharing is an optimization technique that uses a single functional block (such as an adder or comparator) to implement several operators in the HDL code. Use resource sharing to improve design performance by reducing the gate count and the routing congestion. If you do not use resource sharing, each HDL operation is built with

separate circuitry. However, you may want to disable resource sharing for speed critical paths in your design.

The following operators can be shared either with instances of the same operator or with an operator on the same line.

*

+ –

> >= < <=

For example, a + operator can be shared with instances of other + operators or with – operators. A * operator can be shared only with other * operators.

You can implement arithmetic functions (+, –, magnitude comparators) with gates or with your synthesis tool's module library. The library functions use modules that take advantage of the carry logic in XC4000 family, XC5200 family, and Spartan CLBs. Carry logic and its dedicated routing increase the speed of arithmetic functions that are larger than 4-bits. To increase speed, use the module library if your design contains arithmetic functions that are larger than 4-bits or if your design contains only one arithmetic function. Resource sharing of the module library automatically occurs in most synthesis tools if the arithmetic functions are in the same process.

Resource sharing adds additional logic levels to multiplex the inputs to implement more than one function. Therefore, you may not want to use it for arithmetic functions that are part of your design's time critical path.

Since resource sharing allows you to reduce the number of design resources, the device area required for your design is also decreased. The area that is used for a shared resource depends on the type and bit width of the shared operation. You should create a shared resource to accommodate the largest bit width and to perform all operations.

If you use resource sharing in your designs, you may want to use multiplexers to transfer values from different sources to a common resource input. In designs that have shared operations with the same output target, the number of multiplexers is reduced as illustrated in the following VHDL and Verilog examples. The HDL example is shown implemented with gates in the "Implementation of Resource Sharing" figure.

- VHDL Example

```
-- RES_SHARING.VHD
-- May 1997

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity res_sharing is
    port (A1,B1,C1,D1: in STD_LOGIC_VECTOR (7 downto 0);
          COND_1: in STD_LOGIC;
          Z1: out STD_LOGIC_VECTOR (7 downto 0));
end res_sharing;

architecture BEHAV of res_sharing is
begin
P1: process (A1,B1,C1,D1,COND_1)
    begin
       if (COND_1='1') then
           Z1 <= A1 + B1;
       else
           Z1 <= C1 + D1;
       end if;
    end process; -- end P1

end BEHAV;
```

- Verilog Example

```
/* Resource Sharing Example
 * RES_SHARING.V
 * May 1997                                    */

module res_sharing (A1, B1, C1, D1, COND_1, Z1);

input       COND_1;
input  [7:0] A1, B1, C1, D1;
output [7:0] Z1;

reg [7:0] Z1;

    always @(A1 or B1 or C1 or D1 or COND_1)
    begin
           if (COND_1)
               Z1 <= A1 + B1;
           else
              Z1 <= C1 + D1;
    end

endmodule
```

If you disable resource sharing or if you code the design with the adders in separate processes, the design is implemented using two separate modules as shown in the "Implementation without Resource Sharing" figure.

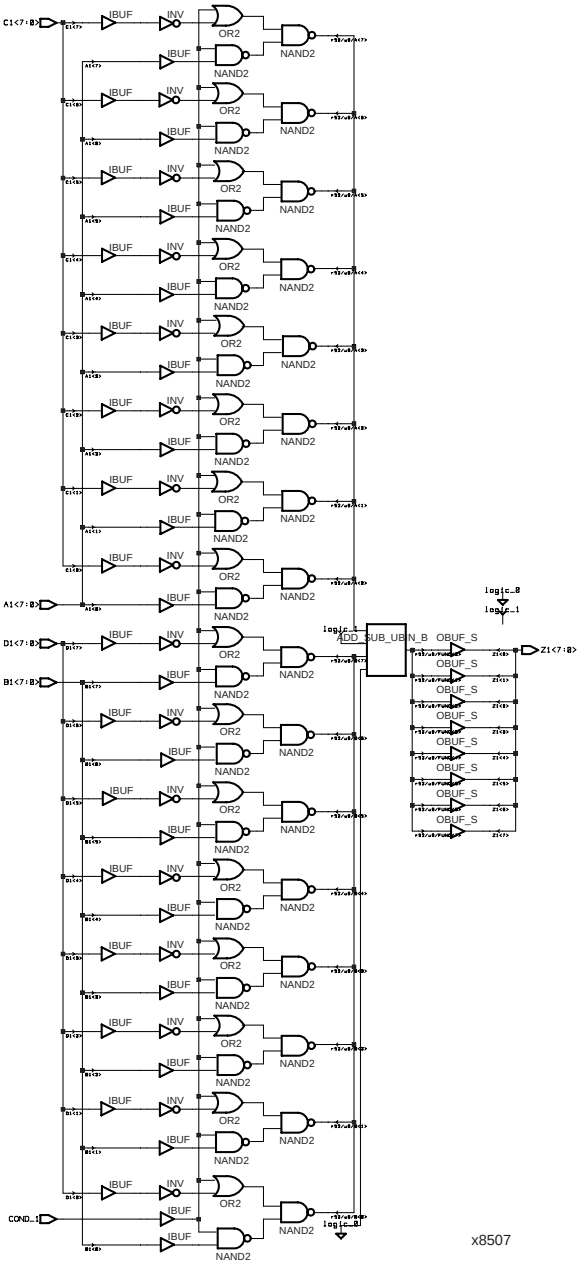


**Figure 1-5 Implementation of Resource Sharing**

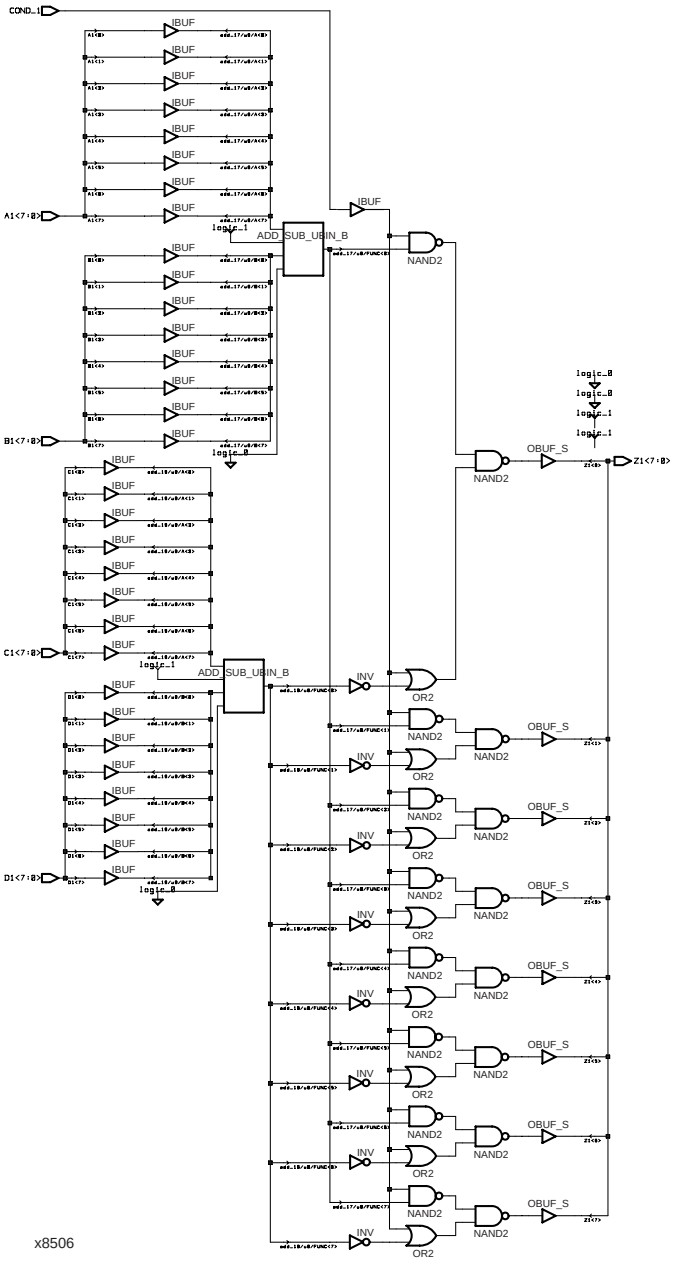


**Figure 1-6 Implementation without Resource Sharing**

Some synthesis tools generate modules from special Xilinx module generation algorithms. Generally, this module generation is used for operators such as adders, subtracters, incrementers, decrementers, and comparators. The following table provides a comparison of the number of CLBs used and the delay for the VHDL and Verilog designs with and without resource sharing.

**Table 1-6 Resource Sharing/No Resource Sharing Comparison for XC4005EPC84-2**

| Comparison | Resource Sharing with Xilinx Module Generation | No Resource Sharing with Xilinx Module Generation | Resource Sharing without Xilinx Module Generation | No Resource Sharing without Xilinx Module Generation |
|---|---|---|---|---|
| F/G Functions | 24 | 24 | 19 | 28 |
| H Function Generators | 0 | 0 | 11 | 8 |
| Fast Carry Logic CLBs | 5 | 10 | 0 | 0 |
| Longest Delay | 27.878 ns | 23.761 ns | 47.010 ns | 33.386 ns |
| Advantages/ Disadvantages | Potential for area reduction | Potential for decreased critical path delay | No carry logic increases path delays | No carry logic increases CLB count |

**Note:** Refer to the appropriate reference manual for more information on resource sharing.

## Gate Reduction

Use the generatedmodule components to reduce the number of gates in your designs. The module generation algorithms use Xilinx carry logic to reduce function generator logic and improve routing and speed performance. Further gate reduction can occur with synthesis tools that recognize the use of constants with the modules.

## Preset Pin or Clear Pin

Xilinx FPGAs consist of CLBs that contain function generators and flip-flops. The XC4000 family flip-flops have a dedicated clock enable pin and either a clear (asynchronous reset) pin or a preset (asynchronous set) pin. All synchronous preset or clear functions can be implemented with combinatorial logic in the function generators.

The XC3000 family and XC5200 family FPGAs have an asynchronous reset pin on the CLB registers. An asynchronous preset can be inferred, but is built by connecting one inverter to the D input and connecting a second inverter to the Q output of a register. In this case, an asynchronous preset is created when the asynchronous reset is activated. This may require additional logic and increase delays. If possible, the inverters are merged with existing logic connected to the register input or output.

You can configure FPGA CLB registers to have either a preset pin or a clear pin. You cannot configure the CLB register for both pins. You must modify any process that requires both pins to use only one pin or you must use two registers to implement the process. If a register is described with an asynchronous set and reset, your synthesis tool may issue an error message similar to the following during the compilation of your design.

```
Warning: Target library contains no replacement for
register ‘Q_reg’ (**FFGEN**) . (TRANS-4)
Warning: Cell ‘Q_reg’ (**FFGEN**) not translated.
(TRANS-1)
```

During the implementation of the synthesized netlist, NGDBuild issues the following error message.

```
ERROR:basnu – logical block “Q_reg” of type “_FFGEN_”
is unexpanded.
```

An XC4000 CLB is shown in the following figure.

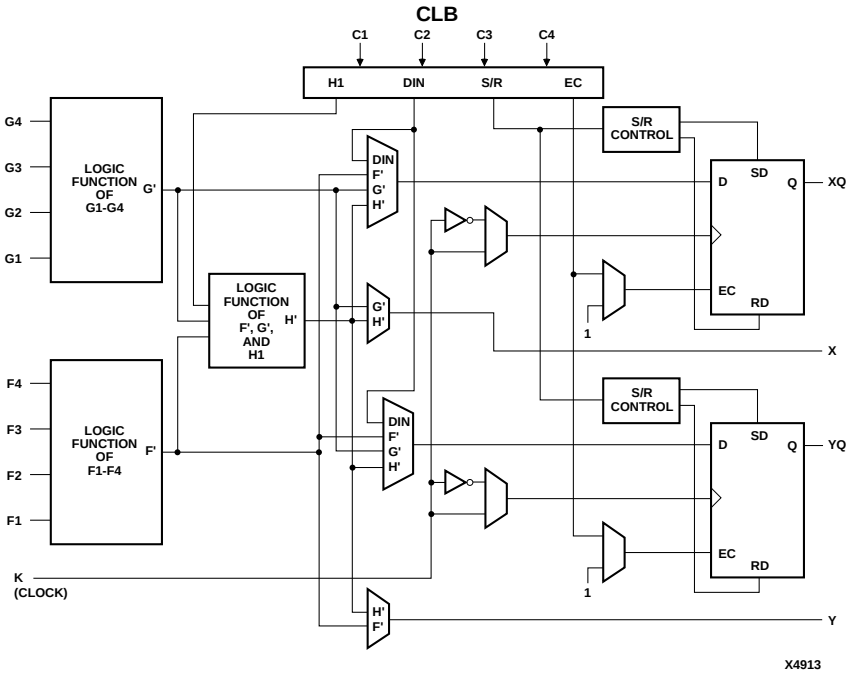


**Figure 1-7 XC4000 Configurable Logic Block**

The following VHDL and Verilog designs show how to describe a register with a clock enable and either an asynchronous preset or a clear.

## Register Inference

- VHDL Example

```
-- FF_EXAMPLE.VHD
-- May 1997
-- Example of Implementing Registers

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ff_example is
   port ( RESET, CLOCK, ENABLE: in STD_LOGIC;
          D_IN: in STD_LOGIC_VECTOR (7 downto 0);
```

```
          A_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
          B_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
          C_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
         D_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0));
end ff_example;

architecture BEHAV of ff_example is
begin

    -- D flip-flop
    FF: process (CLOCK)
    begin
        if (CLOCK'event and CLOCK='1') then
            A_Q_OUT <= D_IN;
        end if;
    end process; -- End FF

    -- Flip-flop with asynchronous reset
    FF_ASYNC_RESET: process (RESET, CLOCK)
    begin
        if (RESET = '1') then
            B_Q_OUT <= "00000000";
        elsif (CLOCK'event and CLOCK='1') then
            B_Q_OUT <= D_IN;
        end if;
    end process; -- End FF_ASYNC_RESET

    -- Flip-flop with asynchronous set
    FF_ASYNC_SET: process (RESET, CLOCK)
    begin
        if (RESET = '1') then
            C_Q_OUT <= "11111111";
        elsif (CLOCK'event and CLOCK = '1') then
            C_Q_OUT <= D_IN;
        end if;
    end process; -- End FF_ASYNC_SET

    -- Flip-flop with asynchronous reset and clock
       enable
    FF_CLOCK_ENABLE: process (ENABLE, RESET, CLOCK)
    begin
        if (RESET = '1') then
           D_Q_OUT <= "00000000";
        elsif (CLOCK'event and CLOCK='1') then
           if (ENABLE='1') then
               D_Q_OUT <= D_IN;
```

```
            end if;
        end if;
    end process; -- End FF_CLOCK_ENABLE

end BEHAV;
```

- Verilog Example

```
/* Example of Implementing Registers
 * FF_EXAMPLE.V
 * May 1997                                    */

module ff_example (RESET, CLOCK, ENABLE, D_IN,
                   A_Q_OUT, B_Q_OUT, C_Q_OUT, D_Q_OUT);

input RESET, CLOCK, ENABLE;
input       [7:0] D_IN;
output      [7:0] A_Q_OUT;
output      [7:0] B_Q_OUT;
output      [7:0] C_Q_OUT;
output      [7:0] D_Q_OUT;

reg         [7:0] A_Q_OUT;
reg         [7:0] B_Q_OUT;
reg         [7:0] C_Q_OUT;
reg         [7:0] D_Q_OUT;

    // D flip-flop
    always @(posedge CLOCK)
    begin
        A_Q_OUT <= D_IN;
    end

    // Flip-flop with asynchronous reset
    always @(posedge RESET or posedge CLOCK)
    begin
        if (RESET)
            B_Q_OUT <= 8'b00000000;
            else
               B_Q_OUT <= D_IN;
    end

    // Flip-flop with asynchronous set
    always @(posedge RESET or posedge CLOCK)
    begin
        if (RESET)
```

```
            C_Q_OUT <= 8'b11111111;
            else
                C_Q_OUT <= D_IN;
    end

    //Flip-flop with asynchronous reset & clock enable
    always @(posedge RESET or posedge CLOCK)
    begin
            if (RESET)
                 D_Q_OUT <= 8'b00000000;
            else if (ENABLE)
                 D_Q_OUT <= D_IN;
    end

endmodule
```

## Using Clock Enable Pin Instead of Gated Clocks

Use the CLB clock enable pin instead of gated clocks in your designs. Gated clocks can introduce glitches, increased clock delay, clock skew, and other undesirable effects. The first two examples in this section (VHDL and Verilog) illustrate a design that uses a gated clock. The "Implementation of Gated Clock" figure shows this design implemented with gates. Following these examples are VHDL and Verilog designs that show how you can modify the gated clock design to use the clock enable pin of the CLB. The "Implementation of Clock Enable" figure shows this design implemented with gates.

- VHDL Example

```
----------------------------------------
-- GATE_CLOCK.VHD Version 1.1          --
-- Illustrates clock buffer control    --
-- Better implementation is to use      --
-- clock enable rather than gated clock --
-- May 1997                            --
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity gate_clock is
    port (IN1,IN2,DATA,CLK,LOAD: in STD_LOGIC;
          OUT1: out STD_LOGIC);
end gate_clock;
```

```
architecture BEHAVIORAL of gate_clock is

signal GATECLK: STD_LOGIC;

begin

GATECLK <= (IN1 and IN2 and CLK);

    GATE_PR: process (GATECLK,DATA,LOAD)
    begin
        if (GATECLK'event and GATECLK='1') then
            if (LOAD='1') then
                OUT1 <= DATA;
            end if;
        end if;
    end process; --End GATE_PR

end BEHAVIORAL;
```

- Verilog Example

```
       ////////////////////////////////////
      // GATE_CLOCK.V Version 1.1            //
     // Gated Clock Example                 //
    // Better implementation to use clock   //
   // enables than gating the clock        //
  // May 1997                             //
 ////////////////////////////////////////

module gate_clock(IN1, IN2, DATA, CLK, LOAD, OUT1) ;
input       IN1 ;
input       IN2 ;
input       DATA ;
input       CLK ;
input       LOAD ;
output      OUT1 ;

reg         OUT1 ;

wire GATECLK ;

assign GATECLK = (IN1 & IN2 & CLK);

always @(posedge GATECLK)
begin
```

```
   if (LOAD == 1'b1)
      OUT1 = DATA;
end

endmodule
```

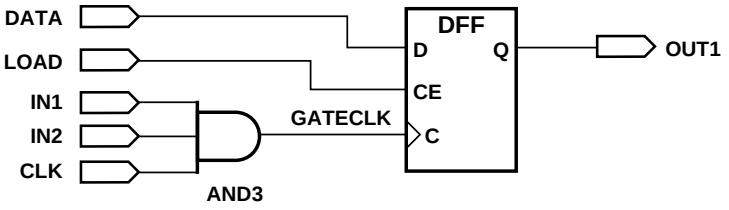


**Figure 1-8 Implementation of Gated Clock**

- VHDL Example

```
-- CLOCK_ENABLE.VHD
-- May 1997

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity clock_enable is
    port (IN1,IN2,DATA,CLOCK,LOAD: in STD_LOGIC;
          DOUT: out STD_LOGIC);
end clock_enable;

architecture BEHAV of clock_enable is
signal ENABLE: STD_LOGIC;
begin

    ENABLE <= IN1 and IN2 and LOAD;

    EN_PR: process (ENABLE,DATA,CLOCK)
    begin
        if (CLOCK'event and CLOCK='1') then
            if (ENABLE='1') then
                DOUT <= DATA;
            end if;
        end if;
```

```
        end process; -- End EN_PR

    end BEHAV;
```

- Verilog Example

```
/* Clock enable example
 * CLOCK_ENABLE.V
 * May 1997                                        */

module clock_enable (IN1, IN2, DATA, CLK, LOAD, DOUT);

input IN1, IN2, DATA;
input CLK, LOAD;
output DOUT;

wire ENABLE;
reg DOUT;

assign ENABLE = IN1 & IN2 & LOAD;

    always @(posedge CLK)
    begin
             if (ENABLE)
                 DOUT <= DATA;
    end

endmodule
```

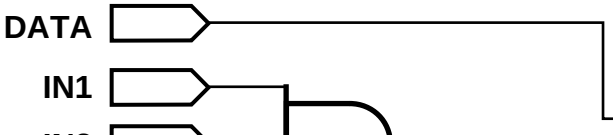


**Figure 1-9 Implementation of Clock Enable**

## Using If Statements

The VHDL syntax for If statements is as follows:

```
if condition then
    sequence_of_statements;
```

```
{elsif condition then
    sequence_of_statements;}
else
    sequence_of_statements;
end if;
```

The Verilog syntax for If statements is as follows:

```
if (condition)
  begin
     sequence of statements;
  end
{else if (condition)
  begin
     sequence of statements;
  end}
else
  begin
     sequence of statements;
  end
```

Use If statements to execute a sequence of statements based on the value of a condition. The If statement checks each condition in order until the first true condition is found and then executes the statements associated with that condition. After a true condition is found and the statements associated with that condition are executed, the rest of the If statement is ignored. If none of the conditions are true, and an Else clause is present, the statements associated with the Else are executed. If none of the conditions are true, and an Else clause is not present, none of the statements are executed.

If the conditions are not completely specified (as shown below), a latch is inferred to hold the value of the target signal.

- VHDL Example

```
if (L = '1') then
    Q <= D;
end if;
```

- Verilog Example

```
if (L==1'b1)
    Q=D;
```

To avoid a latch inference, specify all conditions, as shown here.

- VHDL Example

```
if (L = '1') then
    Q <= D;
else
    Q <= '0';
end if;
```

- Verilog Example

```
if (L==1'b1)
    Q=D;
else
    Q=0;
```

## Using Case Statements

The VHDL syntax for Case statements is as follows.

```
case expression is
    when choices =>
        {sequence_of_statements;}
    {when choices =>
        {sequence_of_statements;}}
    when others =>
        {sequence_of_statements;}
end case;
```

The Verilog syntax for Case statements is as follows.

```
case (expression)
  choices:  statement;
  {choices:  statement;}
  default:  statement;
endcase
```

Use Case statements to execute one of several sequences of statements, depending on the value of the expression. When the Case statement is executed, the given expression is compared to each choice until a match is found. The statements associated with the matching choice are executed. The statements associated with the Others (VHDL) or Default (Verilog) clause are executed when the given expression does not match any of the choices. The Others or Default clause is optional, however, if you do not use it, you must

include all possible values for expression. For clarity and for synthesis, each Choices statement must have a unique value for the expression. If possible, put the most likely Cases first to improve simulation speed.

## Using Nested If Statements

Improper use of the Nested If statement can result in an increase in area and longer delays in your designs. Each If keyword specifies priority-encoded logic. To avoid long path delays, do not use extremely long Nested If constructs as shown in the following VHDL / Verilog examples. These designs are shown implemented in gates in the "Implementation of Nested If" figure. Following these examples are VHDL and Verilog designs that use the Case construct with the Nested If to more effectively describe the same function. The Case construct reduces the delay by approximately 3 ns (using an XC4005E-2 part). The implementation of this design is shown in the "Implementation of If-Case" figure.

### Inefficient Use of Nested If Statement

- VHDL Example

```
-- NESTED_IF.VHD
-- May 1997

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity nested_if is
    port (ADDR_A:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          ADDR_B:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          ADDR_C:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          ADDR_D:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          RESET:    in std_logic;
          CLK :     in std_logic;
          DEC_Q:   out std_logic_vector (5 downto 0)); -- Decode OUTPUT
end nested_if;

architecture xilinx of nested_if is
begin

---------------- NESTED_IF PROCESS --------------
```

```
    NESTED_IF: process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if (RESET = '0') then
                if (ADDR_A = "00") then
                    DEC_Q(5 downto 4) <= ADDR_D;
                    DEC_Q(3 downto 2) <= "01";
                    DEC_Q(1 downto 0) <= "00";
                    if (ADDR_B = "01") then
                        DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                        if (ADDR_C = "10") then
                            DEC_Q(5 downto 4) <= unsigned(ADDR_D) + '1';
                            if (ADDR_D = "11") then
                                DEC_Q(5 downto 4) <= "00";
                            end if;
                        else
                            DEC_Q(5 downto 4) <= ADDR_D;
                        end if;
                    end if;
                else
                    DEC_Q(5 downto 4)  <= ADDR_D;
                    DEC_Q(3 downto 2)  <= ADDR_A;
                    DEC_Q(1 downto 0)  <= unsigned(ADDR_B) + '1';
                end if;
            else
                DEC_Q <= "000000";
            end if;
        end if;
    end process;
end xilinx;
```

- Verilog Example

```
    ////////////////////////////////////
   // NESTED_IF.V                     //
  // Nested If vs. Case Design Example  //
 // August 1997                      //
////////////////////////////////////

module nested_if (ADDR_A, ADDR_B, ADDR_C, ADDR_D, RESET, CLK, DEC_Q);

    input  [1:0]  ADDR_A ;
    input  [1:0]  ADDR_B ;
    input  [1:0]  ADDR_C ;
    input  [1:0]  ADDR_D ;
```

```
    input         RESET, CLK ;
    output [5:0]  DEC_Q ;

    reg    [5:0]  DEC_Q ;

    //  Nested If Process  //
    always @ (posedge CLK)
        begin
        if (RESET == 1'b1)
            begin
            if (ADDR_A == 2'b00)
                begin
                DEC_Q[5:4] <= ADDR_D;
                DEC_Q[3:2] <= 2'b01;
                DEC_Q[1:0] <= 2'b00;
                if (ADDR_B == 2'b01)
                    begin
                    DEC_Q[3:2] <= ADDR_A + 1'b1;
                    DEC_Q[1:0] <= ADDR_B + 1'b1;
                    if (ADDR_C == 2'b10)
                        begin
                        DEC_Q[5:4] <= ADDR_D + 1'b1;
                        if (ADDR_D == 2'b11)
                            DEC_Q[5:4] <= 2'b00;
                        end
                    else
                        DEC_Q[5:4] <= ADDR_D;
                    end
                end
            else
                DEC_Q[5:4] <= ADDR_D;
                DEC_Q[3:2] <= ADDR_A;
                DEC_Q[1:0] <= ADDR_B + 1'b1;
            end
        else
            DEC_Q <= 6'b000000;
        end

endmodule
```

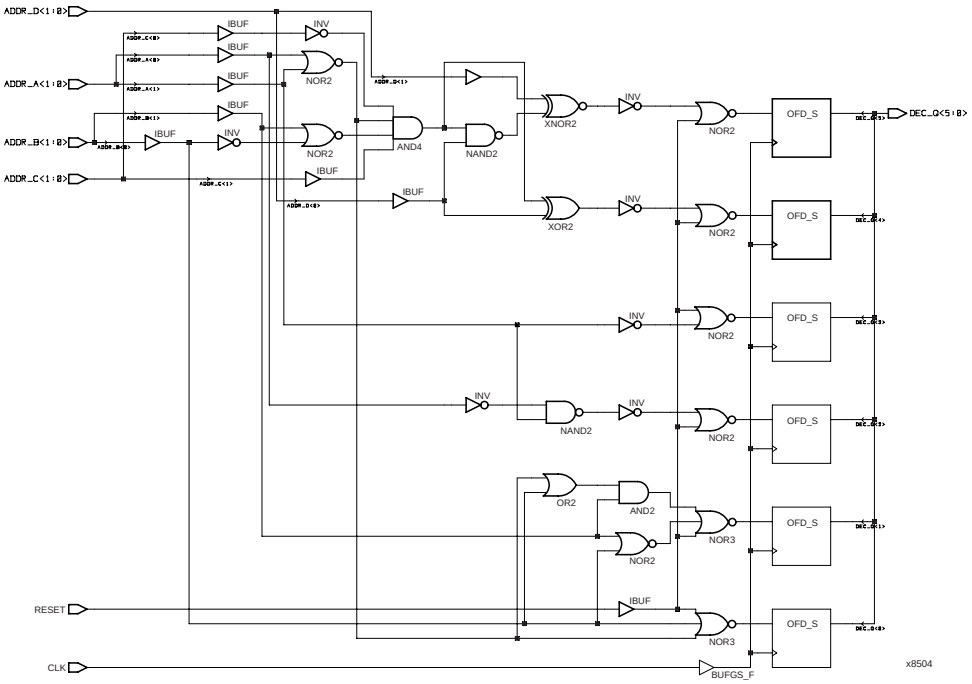


**Figure 1-10 Implementation of Nested If**

## Nested If Example Modified to Use If-Case

**Note:** In the following example, the hyphens ("don't cares") used for bits in the Case statement may evaluate incorrectly to false for some synthesis tools.

- VHDL Example

```
-- IF_CASE.VHD
-- May 1997

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity if_case is
     port (ADDR_A:   in std_logic_vector (1 downto 0); -- ADDRESS Code
           ADDR_B:   in std_logic_vector (1 downto 0); -- ADDRESS Code
```

```
          ADDR_C:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          ADDR_D:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          RESET:    in std_logic;
          CLK :     in std_logic;
          DEC_Q:   out std_logic_vector (5 downto 0)); -- Decode OUTPUT
end if_case;

architecture xilinx of if_case is
signal ADDR_ALL : std_logic_vector (7 downto 0);
begin

----concatenate all address lines ----------------------
ADDR_ALL <= (ADDR_A & ADDR_B & ADDR_C & ADDR_D) ;

--------Use 'case' instead of 'nested_if' for efficient gate netlist------
    IF_CASE: process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if (RESET = '0') then
                case ADDR_ALL is
                    when "00011011" =>
                        DEC_Q(5 downto 4) <= "00";
                        DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                    when "000110--" =>
                        DEC_Q(5 downto 4) <= unsigned(ADDR_D) + '1';
                        DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                    when "0001----" =>
                        DEC_Q(5 downto 4) <= ADDR_D;
                        DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                    when "00------" =>
                        DEC_Q(5 downto 4) <= ADDR_D;
                        DEC_Q(3 downto 2) <= "01";
                        DEC_Q(1 downto 0) <= "00";
                    when others     =>
                        DEC_Q(5 downto 4) <= ADDR_D;
                        DEC_Q(3 downto 2) <= ADDR_A;
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                end case;
            else
                DEC_Q <= "000000";
            end if;
        end if;
```

```
    end process;
end xilinx;
```

- Verilog Example

```
    ////////////////////////////////////
   // IF_CASE.V                        //
  // Nested If vs. Case Design Example  //
 // August 1997                      //
////////////////////////////////////

module if_case (ADDR_A, ADDR_B, ADDR_C, ADDR_D, RESET, CLK, DEC_Q);

    input  [1:0]  ADDR_A ;
    input  [1:0]  ADDR_B ;
    input  [1:0]  ADDR_C ;
    input  [1:0]  ADDR_D ;
    input         RESET, CLK ;
    output [5:0]  DEC_Q ;

    wire   [7:0]  ADDR_ALL ;
    reg    [5:0]  DEC_Q ;

    // Concatenate all address lines //
    assign ADDR_ALL = {ADDR_A, ADDR_B, ADDR_C, ADDR_D} ;

    // Use 'case' instead of 'nested_if' for efficient gate netlist //
    always @ (posedge CLK)
        begin
        if (RESET == 1'b1)
            begin
                casex (ADDR_ALL)
                    8'b00011011: begin
                                 DEC_Q[5:4] <= 2'b00;
                                 DEC_Q[3:2] <= ADDR_A + 1;
                                 DEC_Q[1:0] <= ADDR_B + 1'b1;
                                 end
                    8'b000110xx: begin
                                 DEC_Q[5:4] <= ADDR_D + 1'b1;
                                 DEC_Q[3:2] <= ADDR_A + 1'b1;
                                 DEC_Q[1:0] <= ADDR_B + 1'b1;
                                 end
                    8'b0001xxxx: begin
                                 DEC_Q[5:4] <= ADDR_D;
                                 DEC_Q[3:2] <= ADDR_A + 1'b1;
                                 DEC_Q[1:0] <= ADDR_B + 1'b1;
```

```
                                end
                    8'b00xxxxxx: begin
                                DEC_Q[5:4] <= ADDR_D;
                                DEC_Q[3:2] <= 2'b01;
                                DEC_Q[1:0] <= 2'b00;
                                end
                    default:    begin
                                DEC_Q[5:4] <= ADDR_D;
                                DEC_Q[3:2] <= ADDR_A;
                                DEC_Q[1:0] <= ADDR_B + 1'b1;
                                end
                endcase
            end
        else
            DEC_Q <= 6'b000000;
        end

endmodule
```

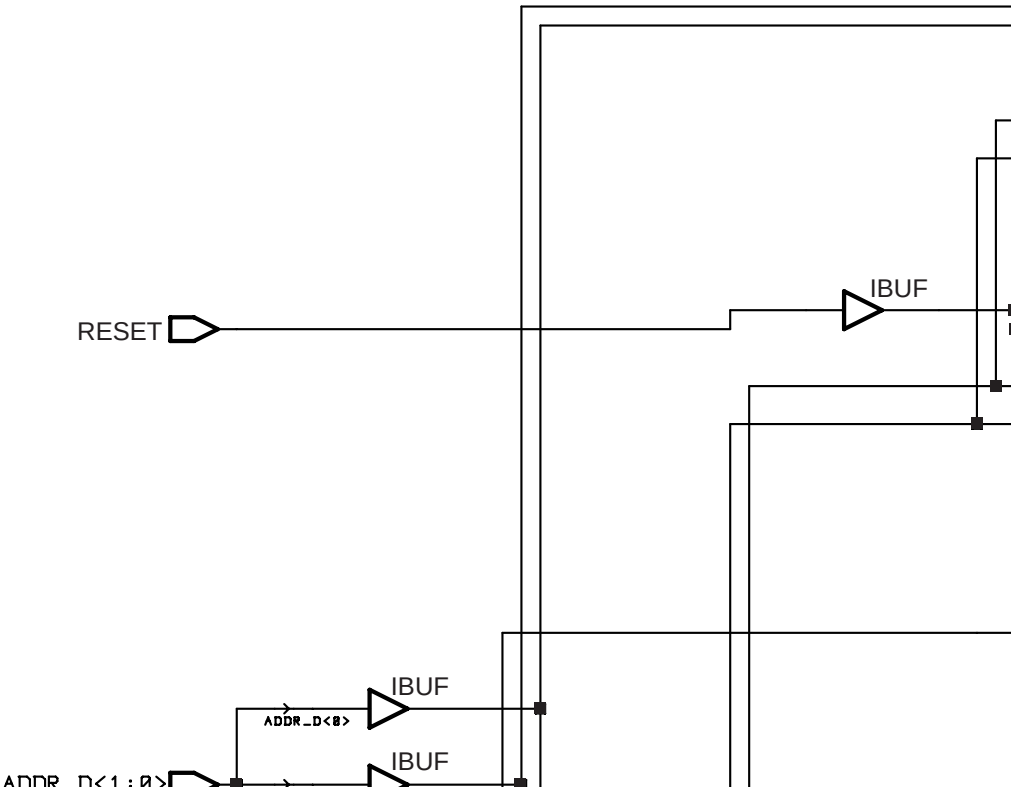


**Figure 1-11 Implementation of If-Case**

## Comparing If Statement and Case Statement

The If statement generally produces priority-encoded logic and the Case statement generally creates balanced logic. An If statement can contain a set of different expressions while a Case statement is evaluated against a common controlling expression. In general, use the Case statement for complex decoding and use the If statement for speed critical paths.

Most current synthesis tools can determine if the if-elsif conditions are mutually exclusive, and will not create extra logic to build the priority tree.

The following examples use an If construct in a 4–to–1 multiplexer design. The "If_Ex Implementation" figure shows the implementation of these designs.

## 4–to–1 Multiplexer Design with If Construct

- VHDL Example

```
-- IF_EX.VHD
-- May 1997

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity if_ex is
    port (SEL: in STD_LOGIC_VECTOR(1 downto 0);
    A,B,C,D: in STD_LOGIC;
    MUX_OUT: out STD_LOGIC);
end if_ex;

architecture BEHAV of if_ex is
begin

    IF_PRO: process (SEL,A,B,C,D)
    begin
        if    (SEL="00") then MUX_OUT <= A;
        elsif (SEL="01") then MUX_OUT <= B;
        elsif (SEL="10") then MUX_OUT <= C;
        elsif (SEL="11") then MUX_OUT <= D;
        else                  MUX_OUT <= '0';
    end if;
end process; --END IF_PRO

end BEHAV;
```

- Verilog Example

```
     // IF_EX.V                                    //
    // Example of a If statement showing a       //
   // mux created using priority encoded logic   //
  // HDL Synthesis Design Guide for FPGAs        //
 // November 1997                               //
////////////////////////////////////////////////

module if_ex (A, B, C, D, SEL, MUX_OUT);

    input        A, B, C, D;
    input  [1:0] SEL;
    output       MUX_OUT;

    reg          MUX_OUT;

    always @ (A or B or C or D or SEL)
    begin
       if (SEL == 2'b00)
          MUX_OUT = A;
       else if (SEL == 2'b01)
          MUX_OUT = B;
       else if (SEL == 2'b10)
          MUX_OUT = C;
       else if (SEL == 2'b11)
          MUX_OUT = D;
       else
          MUX_OUT = 0;
    end

endmodule
```

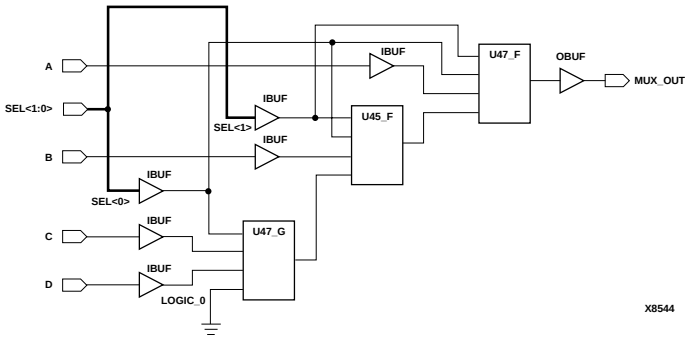


**Figure 1-12 If_Ex Implementation**

The following VHDL and Verilog examples use a Case construct for the same multiplexer. The "Case_Ex Implementation" figure shows the implementation of these designs. In these examples, the Case implementation requires only one XC4000 CLB while the If construct requires two CLBs in some synthesis tools. In this case, design the multiplexer using the Case construct because fewer resources are used and the delay path is shorter.

## 4–to–1 Multiplexer Design with Case Construct

- VHDL Example

```
-- CASE_EX.VHD
-- May 1997

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity case_ex is
    port (SEL: in STD_LOGIC_VECTOR(1 downto 0);
          A,B,C,D: in STD_LOGIC;
          MUX_OUT: out STD_LOGIC);
end case_ex;

architecture BEHAV of case_ex is
begin

    CASE_PRO: process (SEL,A,B,C,D)
```

```
    begin
        case SEL is
            when “00” =>MUX_OUT <= A;
            when “01” =>  MUX_OUT <= B;
            when “10” =>  MUX_OUT <= C;
            when “11” =>  MUX_OUT <= D;
            when others=>  MUX_OUT <= '0';
        end case;
    end process; --End CASE_PRO

end BEHAV;
```

- Verilog Example

```
      ////////////////////////////////////////
     // CASE_EX.V                            //
    // Example of a Case statement showing  //
   // A mux created using parallel logic   //
  // HDL Synthesis Design Guide for FPGAs //
 // November 1997                        //
////////////////////////////////////////

module case_ex (A, B, C, D, SEL, MUX_OUT);

input        A, B, C, D;
input  [1:0] SEL;
output       MUX_OUT;

reg          MUX_OUT;

    always @ (A or B or C or D or SEL)
    begin
    case (SEL)
        2’b00:
            MUX_OUT = A;
        2’b01:
            MUX_OUT = B;
        2’b10:
            MUX_OUT = C;
        2’b11:
            MUX_OUT = D;
            default:
            MUX_OUT = 0;
     endcase
     end
```

```
endmodule
```

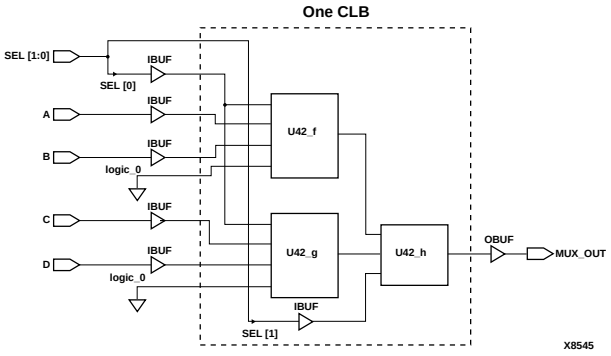


**Figure 1-13 Case_Ex Implementation**