Guide to HDL Coding Styles
for Synthesis

Version 2002.05, June 2002

Comments?
E-mail your comments about Synopsys
documentation to doc@synopsys.com

SYNOPSYS®

Printed in the U.S.A.

Document Order Number: 33345-000 MA
Guide to HDL Coding Styles for Synthesis, v2002.05

Contents

What's New in This Release xiv

About This User Guide xv

Customer Support xviii

1. Coding Styles for if Statements and case Statements

Basic if and case Statements 1-2

Simple case Statement 1-8

2. Coding if and case Statements for Late Arriving Signals

Sequential if Statements: Late Arriving Data Signal 2-2

Single if Statement: Late Arriving Control Signal 2-5

if Statement With Nested case Statement: Late Arriving Data Signal 2-12

case Statement With Nested if Statement: Late Arriving Control Signal 2-17

3. Coding Styles for Logic Building Blocks

Decoder 3-2

Priority Encoder 3-7

Reduction XOR 3-16

Multiplexer 3-22

4. High-Performance Coding Techniques

Data-Path Duplication 4-2

Operator in if Condition 4-9

5. General Coding Style Guidelines

Unintentional Latch Inference 5-2

Incomplete Sensitivity Lists 5-3

Unnecessary Calculations in for Loops 5-4

Resource Sharing 5-5

Figures

Figure 1-1 Structure for Priority Encoded if Statement 1-4

Figure 1-2 Structure Implied by Single if Statement. 1-7

Figure 2-1 Improved Structure: Priority Encoded b if Late Arriving Signal. 2-5

Figure 2-2 Improved Structure for Single if With Late Arriving Control Signal 2-11

Figure 2-3 Structure Implied by Original HDL in Example 2-7, Example 2-8 2-14

Figure 2-4 Structure Implied by Improved HDL in Example 2-9, Example 2-10 2-17

Figure 2-5 Structure Implied by if Statement Nested in case Statement 2-20

Figure 3-1 Decoder Timing Results Versus Address Width 3-5

Figure 3-2 Decoder Area Versus Address Width 3-6

Figure 3-3 Decoder Compile Times Versus Address Width. 3-7

Figure 3-4 Chain Structure for Priority Encoder. 3-11

Figure 3-5 Tree Structure for Priority Encoder 3-15

Figure 3-6 Priority Encoder Compile Time Versus Output Width. . . 3-16

Figure 3-7	Chain Structure for Reduction XOR	3-18
Figure 3-8	Tree Structure for Reduction XOR	3-22
Figure 3-9	Structure Implied by Multiplexer Chain Example	3-25
Figure 3-10	Structure Implied by Multiplexer Tree Example.	3-30
Figure 3-11	Multiplexer Timing Versus Number of Multiplexers	3-31
Figure 3-12	Multiplexer Area Versus Number of Multiplexers	3-32
Figure 4-1	Structure Implied by Original HDL Before Logic Duplication	4-4
Figure 4-2	Structure Implied by Improved HDL With Data Path Duplication	4-8
Figure 4-3	Structure Implied by Original HDL With Late Arriving A Signal	4-11
Figure 4-4	Structure Implied by Improved HDL With A as Input to Comparator	4-13

Tables

Table 2-1 Timing and Area Results for if Statement in case Statement 2-23

Table 3-1 Timing Results for Decoder Coding Styles 3-4

Table 3-2 Area Results for Decoder Coding Styles 3-5

Table 3-3 Compile Time (Seconds) for Decoder Coding Styles . . . 3-6

Table 3-4 Timing Results for Various Encoder Coding Styles 3-14

Table 3-5 Area Results for Various Encoder Coding Styles 3-14

Table 3-6 Compile Time (Seconds) for Various Encoder Coding Styles 3-15

Table 3-7 Timing Results for Various Multiplexer Coding Styles. . . 3-30

Table 3-8 Multiplexer Area Versus Number of Multiplexers 3-31

Table 4-1 Timing and Area Results for Data-Path Duplication 4-8

Table 4-2 Timing and Area Results for Conditional Operator Examples 4-13

Examples

Example 1-1 Verilog Example of Priority Encoded if Statement . . . 1-2

Example 1-2 VHDL Example of Priority Encoded if Statement 1-3

Example 1-3 Verilog Example for Single if Statement (Not Priority Encoded). 1-5

Example 1-4 VHDL Example for Single if Statement (Not Priority Encoded). 1-6

Example 1-5 Verilog for Single case Statement 1-8

Example 1-6 VHDL for Single case Statement 1-9

Example 2-1 Improved Verilog for Priority Encoded if 2-3

Example 2-2 Improved VHDL for Priority Encoded if 2-4

Example 2-3 Verilog Example of Single if With Late Arriving Control Signal 2-6

Example 2-4 VHDL Example of Single if With Late Arriving Control Signal 2-7

Example 2-5 Improved Verilog for Single if With Late Arriving Control Signal 2-9

Example 2-6 Improved VHDL for Single if With Late Arriving Control Signal 2-10

Example 2-7 Original Verilog for case Statement in if Statement . . . 2-12
Example 2-8 Original VHDL for case Statement in if Statement . . . 2-13
Example 2-9 Improved Verilog for case Statement in if Statement . . . 2-15
Example 2-10 Improved VHDL for case Statement in if Statement . . . 2-16
Example 2-11 Original Verilog for if Statement in case Statement. . . 2-18
Example 2-12 Original VHDL for if Statement in case Statement . . . 2-19
Example 2-13 Improved Verilog for if Statement in case Statement . . . 2-21
Example 2-14 Improved VHDL for if Statement in case Statement . . . 2-22
Example 3-1 Verilog for Decoder Using Indexing . . . 3-2
Example 3-2 VHDL for Decoder Using Indexing . . . 3-3
Example 3-3 Verilog for Decoder Using Loop . . . 3-3
Example 3-4 VHDL for Decoder Using Loop . . . 3-4
Example 3-5 Verilog for Priority Encoder Using Loop Starting With Lowest-Priority Bit . . . 3-8
Example 3-6 VHDL for Priority Encoder Using Loop Starting With Lowest-Priority Bit . . . 3-9
Example 3-7 VHDL for Priority Encoder Tree . . . 3-12
Example 3-8 Verilog for Reduction XOR Chain . . . 3-17
Example 3-9 VHDL for Reduction XOR Chain . . . 3-18
Example 3-10 Verilog for XOR Tree . . . 3-19

Example 3-11 VHDL for XOR Tree 3-21
Example 3-12 Verilog for Multiplexer Chain 3-23
Example 3-13 VHDL for Multiplexer Chain 3-24
Example 3-14 Verilog for Multiplexer Tree 3-26
Example 3-15 VHDL for Multiplexer Tree 3-28
Example 4-1 Original Verilog Before Logic Duplication 4-2
Example 4-2 Original VHDL Before Logic Duplication 4-3
Example 4-3 Improved Verilog With Data Path Duplicated 4-5
Example 4-4 Improved VHDL With Data Path Duplicated 4-6
Example 4-5 Original Verilog With Operator in Conditional Expression 4-9
Example 4-6 Original VHDL With Operator in Conditional Expression 4-10
Example 4-7 Improved Verilog With Operator in Conditional Expression 4-11
Example 4-8 Improved VHDL With Operator in Conditional Expression 4-12
Example 5-1 Verilog Showing Unintentional Latch Inference 5-2
Example 5-2 VHDL Showing Unintentional Latch Inference 5-2
Example 5-3 Verilog With Missing Signal in Sensitivity List 5-4
Example 5-4 VHDL With Missing Signal in Sensitivity List 5-4
Example 5-5 Original VHDL With Unnecessary Statement in Loop 5-5
Example 5-6 Improved VHDL With Statement Pulled out of Loop 5-5

Example 5-7	No Resource Sharing for Conditional Operator in Verilog	5-5
Example 5-8	Resource Sharing for equivalent if-then-else in Verilog	5-6

Preface

Although the Synopsys Design Compiler tool does an excellent job of converting HDL to gates, the structure of the HDL may not allow Design Compiler to meet the designer-specified constraints and is very likely to result in an increase in compile time. The startpoint for synthesis affects the quality of results after synthesis. Designers must keep performance requirements in mind and think in terms of hardware, not software, when writing HDL.

This preface includes the following sections:

- What's New in This Release
- About This User Guide
- Customer Support

What’s New in This Release

This user guide is in maintenance. Any new information for the document will be put in the release notes.

Note:

All Verilog content in this document applies to the original HDL Compiler and not the Presto Verilog HDL Complier. For Presto Verilog, the content has been incorporated into the *HDL Compiler (Presto Verilog) Reference Manual*.

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about resolved Synopsys Technical Action Requests (STARs), is available in the *HDL Compiler Release Note* in SolvNet.

To see the *HDL Compiler Release Note,*

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNet.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, click New Synopsys User Registration.)
3. Click Release Notes in the Main Navigation section, find the 2002.05 Release Notes, then open the *HDL Compiler Release Note.*

About This User Guide

This guide describes the structure implied by some HDL constructs and provides coding style examples and techniques HDL designers can apply to their designs. All coding style examples include the timing and area results after synthesis. These results demonstrate that coding style has a direct impact on synthesis quality of results (QOR).

Audience

This guide is for designers who are familiar with

- VHDL or Verilog
- The UNIX operating system
- The X Window System
- The basic concepts of synthesis and simulation

This guide describes procedures for a Sun workstation running UNIX. If you use another platform, you might see minor differences between the manual descriptions and the way you interact with your operating system.

Related Publications

For additional information about VHDL Compiler and HDL Compiler for Verilog, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys Electronic Software Transfer (EST) system

- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

You might also want to refer to the documentation for the following related Synopsys products:

- Design Analyzer
- Design Compiler
- DesignWare
- Library Compiler
- VHDL System Simulator (VSS)

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* `[`*`pin2 ... pinN`*`]`
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call With the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, click New Synopsys User Registration.)

If you need help using SolvNet, click SolvNet Help in the column on the left side of the SolvNet Web page.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), then clicking "Enter a Call With the Support Center."
- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Coding Styles for if Statements and case Statements

This chapter includes a section about the structure implied by simple if statements and case statements and a section about more-complex examples that use a combination of if statements and case statements.

This chapter contains the following sections:

- Basic if and case Statements
- Simple case Statement

Basic if and case Statements

Examine the following basic if and case statements to understand the structure they imply and the differences, if any, between the two.

Example 1-1 and Example 1-2 show Verilog and VHDL versions of four sequential if statements. These designs result in a priority encoded structure as shown in Figure 1-1.

Example 1-1 Verilog Example of Priority Encoded if Statement

```
module mult_if(a, b, c, d, sel, z);
input a, b, c, d;
input [3:0] sel;
output z;
reg z;

always @(a or b or c or d or sel)
begin
    z = 0;
    if (sel[0]) z = a;
    if (sel[1]) z = b;
    if (sel[2]) z = c;
    if (sel[3]) z = d;
end
endmodule
```

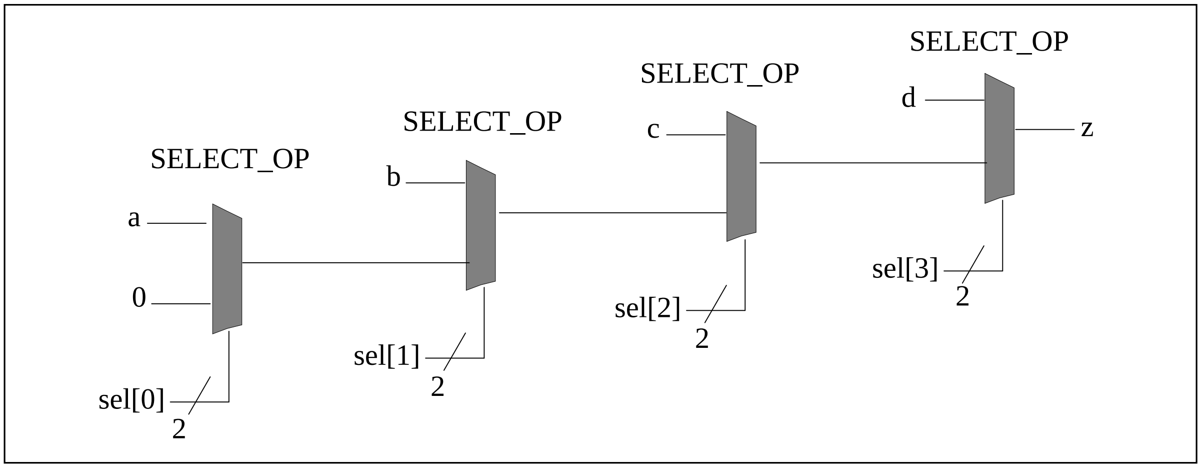
Example 1-2 is the equivalent VHDL example.

Example 1-2   VHDL Example of Priority Encoded if Statement

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mult_if is
port (a, b, c, d: in std_logic;
    sel: in std_logic_vector(3 downto 0);
    z: out std_logic);
end mult_if;

architecture one of mult_if is
begin
    process (a, b, c, d, sel)
    begin
        z <= 0;
        if (sel(0) = ’1’) then
            z <= a;
        end if;
        if (sel(1) = ’1’) then
            z <= b;
        end if;
        if (sel(2) = ’1’) then
            z <= c;
        end if;
        if (sel(3) = ’1’) then
            z <= d;
        end if;
    end process;
end one;
```

Figure 1-1 shows the priority encoded, cascaded structure generated for Example 1-1 and Example 1-2.

Figure 1-1 Structure for Priority Encoded if Statement



Note:

> The generic `SELECT_OP` components shown in Figure 1-1 are used by the VHDL Compiler tool to implement conditional operations in the HDL (for example, if statements or case statements). `SELECT_OP` components behave like one-hot multiplexers; the control lines are mutually exclusive, and each `SELECT_OP` control input allows the data on the corresponding `SELECT_OP` data input to pass to the output of the `SELECT_OP`. Do not confuse these `SELECT_OP` components with one-hot multiplexers in a technology library. The Design Compiler tool does not map to one-hot multiplexers in a technology library.

Sequential if statements allow you to structure HDL for late arriving signals. In Figure 1-1, the inputs to the first `SELECT_OP` in the chain (`sel[0]` and a or `0`) have the longest delay to the output `z`. The inputs to the last `SELECT_OP` in the chain (`sel[3]` and d) have the shortest delay to the output. This is discussed further in Chapter 2, “Coding if and case Statements for Late Arriving Signals."

An `if...else if` construct in Verilog and an `if...elsif` in VHDL are implemented with single `SELECT_OP` components. Therefore, they do not result in the cascaded structure shown in

Figure 1-1. Single if statements result in a parallel structure. Example 1-3 and Example 1-4 show the Verilog and VHDL examples that use the single if statement coding style. Figure 1-2 shows the parallel structure inferred for these examples.

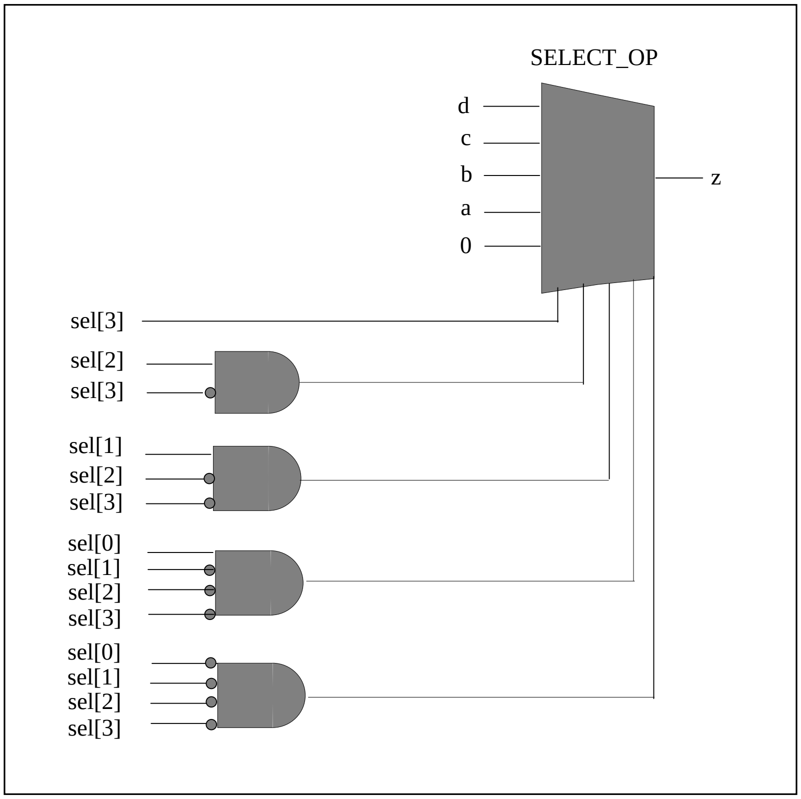
Example 1-3 Verilog Example for Single if Statement (Not Priority Encoded)

```
module single_if(a, b, c, d, sel, z);
input a, b, c, d;
input [3:0] sel;
output z;
reg z;

always @(a or b or c or d or sel)
begin
     z = 0;
     if (sel[3])
          z = d;
     else if (sel[2])
          z = c;
     else if (sel[1])
          z = b;
     else if(sel[0])
          z = a;
end
endmodule
```

Example 1-4 VHDL Example for Single if Statement (Not Priority Encoded)

```
library IEEE;
use IEEE.std_logic_1164.all;
entity single_if is
port (a, b, c, d: in std_logic;
    sel: in std_logic_vector(3 downto 0);
    z: out std_logic);
end single_if;

architecture one of single_if is
begin
    process(a, b, c, d, sel)
    begin
        z <= 0;
        if (sel(3) = ’1’) then
            z <= d;
        elsif (sel(2) = ’1’) then
            z <= c;
        elsif (sel(1) = ’1’) then
            z <= b;
        elsif (sel(0) = ’1’) then
            z <= a;
        end if;
    end process;
end one;
```

Figure 1-2   Structure Implied by Single if Statement



Now that you have seen some basic if statement examples, examine the structure implied by a simple case statement.

Simple case Statement

Example 1-5 and Example 1-6 show, respectively, Verilog and VHDL examples of a single case statement.

Example 1-5 Verilog for Single case Statement

```
module case1(a, b, c, d, sel, z);
input a, b, c, d;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or sel)
begin
    casex (sel)
        4'b1xxx: z = d;
        4'bx1xx: z = c;
        4'bxx1x: z = b;
        4'bxxx1: z = a;
        default: z = 1'b0;
    endcase
end
endmodule
```

Notice the use of the casex statement in Example 1-5. The casex statement enables you to take advantage of don't care conditions, so you don't have to specify all possible 0/1 combinations of `sel`. VHDL, on the other hand, does not let you take advantage of don't care conditions. The VHDL language requires that VHDL Compiler treat comparisons with don't care conditions as false. Example 1-6 shows the equivalent VHDL for Example 1-5. In Example 1-6, all conditions have to be specified in the branches of the case statement.

Example 1-6 VHDL for Single case Statement

```
library IEEE;
use IEEE.std_logic_1164.all;
entity case1 is
port (a, b, c, d: in std_logic;
     sel: in std_logic_vector(3 downto 0);
     z: out std_logic);
end case1;

architecture one of case1 is
begin
     process(a,b,c,d,sel)
     begin
          case sel is
               when "1000"|"1001"|"1010"|"1011"|
                    "1100"|"1101"|"1110"|"1111" => z <= d;
               when "0100"|"0101"|"0110"|"0111" => z <= c;
               when "0010"|"0011" => z <= b;
               when "0001" => z <= a;
               when others => z <= '0';
          end case;
     end process;
end one;
```

The `others` branch is required by the language in the case statement in Example 1-6, because the case statement does not cover all possible values. The “0000" case is missing, as are combinations of other `std_logic` values (the `std_logic` type is defined to have nine possible values). The `others` branch is required to cover combinations in addition to the 0 and 1 combinations.

The structure implied for the single case statement in Example 1-5 and Example 1-6 is identical to that shown in Figure 1-2 for a single if statement. A case statement gives the same results as a single if statement if the selector in the case statement branches is the same as the selector in the if statement branches. "

The preceding VHDL examples cover multiple if statements, single if statements, and single case statements. These are summarized as follows:

Multiple if

```
statement4;
if (cond = cond1) then
     statement1;
end if;
if (cond = cond2) then
     statement2;
end if;
if (cond = cond3) then
     statement3;
end if;
```

A multiple if statement style has the longest path. Both the data and the control signals cascade through multiple `SELECT_OP` constructs.

Single if

```
if (cond = cond1) then
     statement1;
elsif (cond = cond2) then
     statement2;
elsif (cond = cond3) then
     statement3;
else
     statement4;
end if;
```

A single if statement is represented by a single `SELECT_OP`, but the control signals are priority encoded. The delay through the `SELECT_OP` is the same for all the data inputs.

Single case

```
case (cond) is
     when cond1 =>
          statement1;
     when cond2 =>
          statement2;
     when cond3 =>
          statement3;
     when others =>
          statement4;
end case;
```

A single case statement is also represented by a single `SELECT_OP`. There is no priority encoding for the control signals. The delay through the `SELECT_OP` is the same for all the data inputs. A single if and a single case statement give identical results if the selector is the same in every branch of the if statement.

2

Coding if and case Statements for Late Arriving Signals

Designers usually know which signals in a design are late arriving signals. This knowledge can be used to structure HDL so that the late arriving signals appear closer to the output. This chapter gives examples of restructuring if and case statements for late arriving signals.

This chapter contains the following sections:

- Sequential if Statements: Late Arriving Data Signal
- Single if Statement: Late Arriving Control Signal
- if Statement With Nested case Statement: Late Arriving Data Signal
- case Statement With Nested if Statement: Late Arriving Control Signal

Sequential if Statements: Late Arriving Data Signal

Sequential if statements allow you to structure your HDL based on critical signals. You may want to use sequential if statements, as shown in Example 1-1 on page 1-2 and Example 1-2 on page 1-3, if the input d in those examples is a late arriving signal. Figure 1-1 on page 1-4 shows that d is an input to the last `SELECT_OP` in the chain. Therefore, d is as close to the output as possible.

However, if b is a late arriving signal, Example 1-1 on page 1-2 and Example 1-2 on page 1-3 do not show the optimal coding style. When b is the late arriving signal, the optimal coding style involves restructuring the HDL as shown in Example 2-1 and Example 2-2. This restructuring moves b (`b_is_late` in Example 2-1 and Example 2-2) closer to the output `z`.

Example 2-1 Improved Verilog for Priority Encoded if

```
module mult_if_improved(a, b_is_late, c, d, sel, z);
input a, b_is_late, c, d;
input [3:0] sel;
output z;
reg z, z1;

always @(a or b_is_late or c or d or sel)
begin
    z1 = 0;
    if (sel[0])
        z1 = a;
    if (sel[2])
        z1 = c;
    if (sel[3])
        z1 = d;
    if (sel[1] & ~(sel[2]|sel[3]))
        z = b_is_late;
    else
        z = z1;
end
endmodule
```

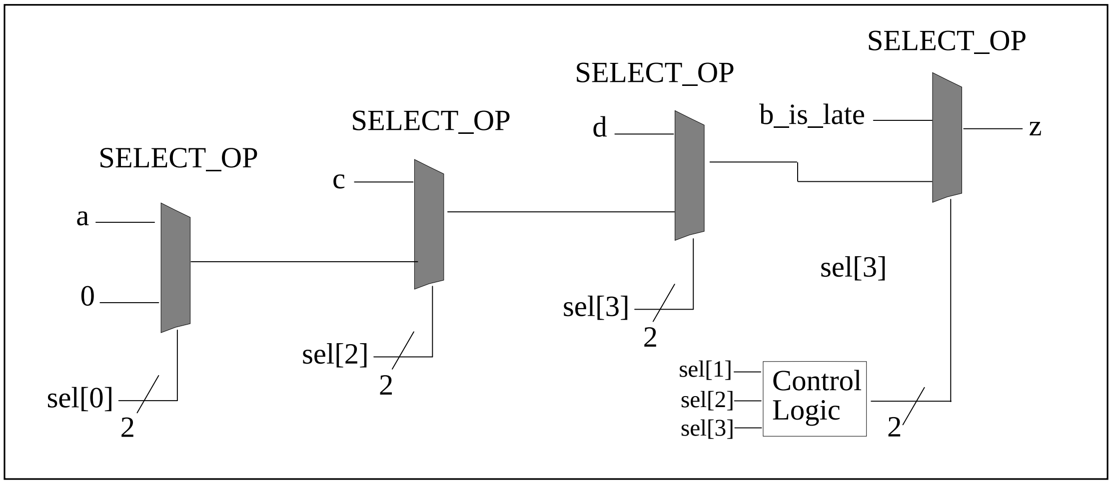
Example 2-2 is the equivalent VHDL example.

Example 2-2 Improved VHDL for Priority Encoded if

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mult_if_improved is
port (a, b_is_late, c, d: in std_logic;
        sel: in std_logic_vector(3 downto 0);
        z: out std_logic);
end mult_if_improved;

architecture one of mult_if_improved is
signal z1: std_logic;
begin
        process (a,b_is_late,c,d,sel,z1)
        begin
             z1 <= ’0’;
             if (sel(0) = ’1’) then
                  z1 <= a;
             end if;
             if (sel(2) = ’1’) then
                  z1 <= c;
             end if;
             if (sel(3) = ’1’) then
                  z1 <= d;
             if ((sel(1) = ’1’) and ((sel(2) or sel(3)) = ’0’)) then
                  z <= b_is_late;
             else
                  z = z1;
             end if;
        end process;
end one;
```

Figure 2-1 shows the structure implied by the HDL in Example 2-1 and Example 2-2.

Figure 2-1   Improved Structure: Priority Encoded b if Late Arriving Signal



Single if Statement: Late Arriving Control Signal

If a single if statement, as shown in Example 1-3 on page 1-5 and Example 1-4 on page 1-6, has a late arriving signal as a condition in one of the branches of the if statement, consider the following Verilog and VHDL examples, where `CTRL_is_late_arriving` is a late arriving input signal.

Example 2-3   Verilog Example of Single if With Late Arriving Control Signal

```
module single_if_late(A, C, CTRL_is_late_arriving, Z);
input [6:1] A;
input [5:1] C;
input CTRL_is_late_arriving;
output Z;
reg Z;

always @(C or A or CTRL_is_late_arriving)
begin
    if (C[1] == 1'b1)
        Z = A[1];
    else if (C[2] == 1'b0)
        Z = A[2];
    else if (C[3] == 1'b1)
        Z = A[3];
    else if (C[4] == 1'b1 && CTRL_is_late_arriving == 1'b0)
                    // late arriving signal in if condition
        Z = A[4];

    else if (C[5] == 1'b0)
        Z = A[5];
    else
        Z = A[6];
end
endmodule
```

Example 2-4 shows the equivalent VHDL.

Example 2-4 VHDL Example of Single if With Late Arriving Control Signal

```
library IEEE;
use IEEE.std_logic_1164.all;

entity single_if_late is
port (A: in std_logic_vector(6 downto 1);
        C: in std_logic_vector(5 downto 1);
        CTRL_is_late_arriving: in std_logic;
        Z: out std_logic);
end single_if_late;

architecture one of single_if_late is
begin
        process(A, C, CTRL_is_late_arriving)
        begin
             if (C(1) = ’1’) then
                  Z <= A(1);
             elsif (C(2) = ’0’) then
                  Z <= A(2);
             elsif (C(3) = ’1’) then
                  Z <= A(3);
             elsif (C(4) = ’1’ and CTRL_is_late_arriving = ’0’) then
                  Z <= A(4);
             elsif (C(5) = ’0’) then
                  Z <= A(5);
             else
                  Z <= A(6);
             end if;
        end process;
end one;
```

Example 2-3 and Example 2-4 result in a structure similar to that shown in Figure 1-2 on page 1-7. The control logic in Example 2-3 and Example 2-4 is slightly different, due to the logical AND (&&) in one of the if conditions in Example 2-3.

Because `CTRL_is_late_arriving` is a late arriving signal, the objective is to push this signal as far out as possible. That is, the signal should be as close to the output port, Z, as possible in the structure implied by the HDL.

Example 2-5 and Example 2-6 show the modified Verilog and VHDL that achieve this objective. The signal `CTRL_is_late_arriving` is pulled out of the `if...else if` statement and is thereby pushed closer to the output.

Example 2-5 Improved Verilog for Single if With Late Arriving Control Signal

```
module single_if_improved(A, C, CTRL_is_late_arriving, Z);
input [6:1] A;
input [5:1] C;
input CTRL_is_late_arriving;
output Z;
reg Z;
reg Z1;
wire Z2, prev_cond;

always @(A or C)
begin
     if (C[1] == 1'b1)
          Z1 = A[1];
     else if (C[2] == 1'b0)
          Z1 = A[2];
     else if (C[3] == 1'b1)
          Z1 = A[3];
     else if (C[5] == 1'b0)// removed the branch with the
          Z1 = A[5];    //late-arriving control signal
     else
          Z1 = A[6];
end

assign Z2 = A[4];
assign prev_cond = (C[1] == 1'b1) || (C[2] == 1'b0) || (C[3]
== 1'b1);

always @(C or prev_cond or CTRL_is_late_arriving or Z1 or Z2)
begin
     if (C[4] == 1'b1 && CTRL_is_late_arriving == 1'b0)
          if (prev_cond)
               Z = Z1;
          else
               Z = Z2;
          else
               Z = Z1;
     end
endmodule
```

Example 2-6 shows the equivalent improved VHDL.

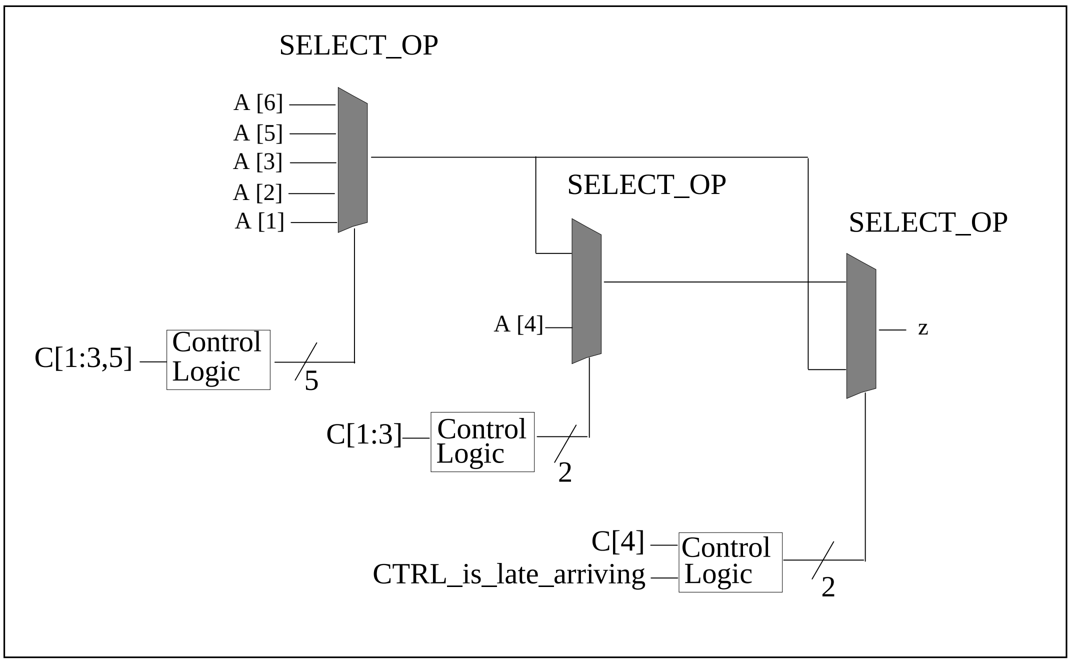
Example 2-6   Improved VHDL for Single if With Late Arriving Control Signal

```
library IEEE;
use IEEE.std_logic_1164.all;

entity single_if_improved is
port (A: in std_logic_vector(6 downto 1);
     C: in std_logic_vector(5 downto 1);
     CTRL_is_late_arriving: in std_logic;
     Z: out std_logic);
end single_if_improved;

architecture one of single_if_improved is
signal Z1, Z2: std_logic;
signal prev_cond: boolean;
begin
     Z2 <= A(4);
     prev_cond <= ((C(1) = ’1’) or (C(2) = ’0’) or (C(3) =
’1’));
     process(A, C)
     begin
          if (C(1) = ’1’) then
               Z1 <= A(1);
          elsif (C(2) = ’0’) then
               Z1 <= A(2);
          elsif (C(3) = ’1’) then
               Z1 <= A(3);
          elsif (C(5) = ’0’) then-- removed the branch with
the late
               Z1 <= A(5);       -- arriving control signal
          else
               Z1 <= A(6);
          end if;
     end process;

     process(C, prev_cond, CTRL_is_late_arriving, Z1, Z2)
     begin
          if ((C(4) = ’1’) and (CTRL_is_late_arriving = ’0’))
then
               if (prev_cond) then
                    Z <= Z1;
               else
```

```
                Z <= Z2;
            end if;
        else
            Z <= Z1;
        end if;
    end process;
end one;
```

Figure 2-2 shows the structure implied by the improved HDL.

Figure 2-2 Improved Structure for Single if With Late Arriving Control Signal



Now that you have seen some basic if and case statements, you can apply the information to some more-complex examples (with late arriving signals) that use if and case statements together.

if Statement With Nested case Statement: Late Arriving Data Signal

Example 2-7 is a Verilog example of a case statement nested in an if statement. The data signal `DATA_is_late_arriving` in one branch of the case statement is a late arriving signal.

Example 2-7 Original Verilog for case Statement in if Statement

```
module case_in_if_01(A, DATA_is_late_arriving, C, sel, Z);
input [8:1] A;
input DATA_is_late_arriving;
input [2:0] sel;
input [5:1] C;
output Z;
reg Z;

always @ (sel or C or A or DATA_is_late_arriving)
begin
    if (C[1])
        Z = A[5];
    else if (C[2] == 1'b0)
        Z = A[4];
    else if (C[3])
        Z = A[1];
    else if (C[4])
        case (sel)
            3'b010: Z = A[8];
            3'b011: Z = DATA_is_late_arriving;
            3'b101: Z = A[7];
            3'b110: Z = A[6];
            default: Z = A[2];
        endcase
    else if (C[5] == 1'b0)
        Z = A[2];
    else
        Z = A[3];
end
endmodule
```

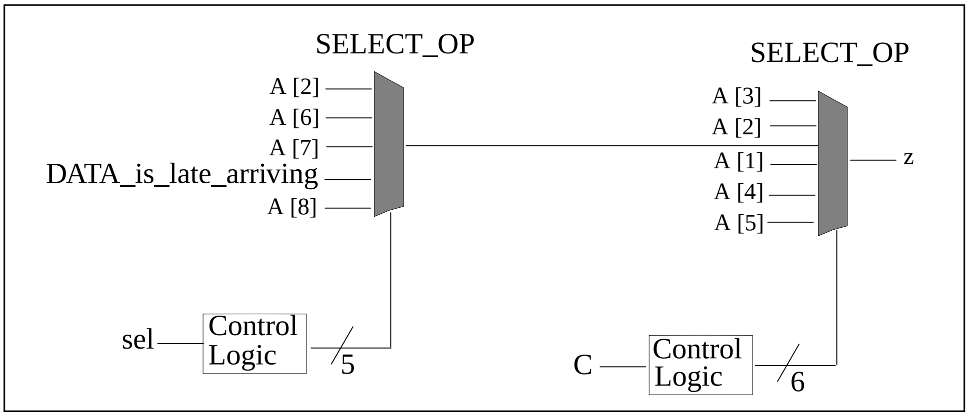
Example 2-8 is the VHDL equivalent of Example 2-7.

Example 2-8 Original VHDL for case Statement in if Statement

```
library IEEE;
use IEEE.std_logic_1164.all;

entity case_in_if_01 is
port(A: in std_logic_vector(8 downto 1);
     DATA_is_late_arriving: in std_logic;
     sel: in std_logic_vector(2 downto 0);
     C: in std_logic_vector(5 downto 1);
     Z: out std_logic);
end case_in_if_01;

architecture orig of case_in_if_01 is
begin
     process(sel, C, A, DATA_is_late_arriving)
     begin
          if (C(1) = '1') then
               Z <= A(5);
          elsif (C(2) = '0') then
               Z <= A(4);
          elsif (C(3) = '1') then
               Z <= A(1);
          elsif (C(4) = '1') then
               case sel is
                    when "010" =>Z <= A(8);
                    when "011" =>Z <= DATA_is_late_arriving;
                    when "101" =>Z <= A(7);
                    when "110" =>Z <= A(6);
                    when others =>Z <= A(2);
               end case;
          elsif (C(5) = '0') then
               Z <= A(2);
          else
               Z <= A(3);
          end if;
     end process;
 end orig;
```

Figure 2-3 shows the structure implied by the HDL in Example 2-7 and Example 2-8.

Figure 2-3 Structure Implied by Original HDL in Example 2-7, Example 2-8



You know that `DATA_is_late_arriving` is a late arriving signal in Example 2-7 and Example 2-8. In Figure 2-3, this signal is an input to the first `SELECT_OP` in the path. To improve the startpoint for synthesis, modify the HDL to move `DATA_is_late_arriving` closer to the output `Z`.

You can do this by moving the assignment of `DATA_is_late_arriving` out of the nested case statement into a separate if statement. This makes `DATA_is_late_arriving` an input to another `SELECT_OP` that is closer to the output port `Z`.

Example 2-9 shows the improved version of the Verilog shown in Example 2-7.

Example 2-9 Improved Verilog for case Statement in if Statement

```
module case_in_if_01_improved(A,DATA_is_late_arriving,C,sel,Z);
input [8:1] A;
input DATA_is_late_arriving;
input [2:0] sel;
input [5:1] C;
output Z;
reg Z;
reg Z1, FIRST_IF;

always @(sel or C or A or DATA_is_late_arriving)
begin
        if (C[1])
             Z1 = A[5];
        else if (C[2] == 1'b0)
             Z1= A[4];
        else if (C[3])
             Z1 = A[1];
        else if (C[4])
             case (sel)
                  3'b010: Z1 = A[8];
             //3'b011: Z1 = DATA_is_late_arriving;
                  3'b101: Z1 = A[7];
                  3'b110: Z1 = A[6];
                  default: Z1 = A[2];
             endcase
        else if (C[5] == 1'b0)
             Z1 = A[2];
        else
             Z1 = A[3];

        FIRST_IF = (C[1] == 1'b1) || (C[2] == 1'b0) || (C[3] == 1'b1);

        if (!FIRST_IF && C[4] && (sel == 3'b011))
             Z = DATA_is_late_arriving;
        else
             Z = Z1;
end
endmodule
```

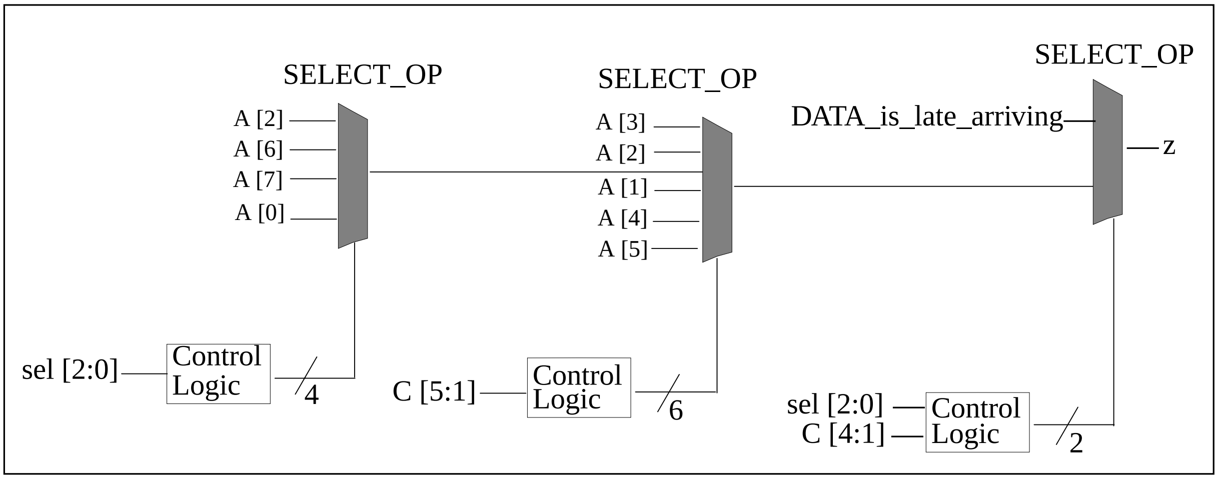
Example 2-10   Improved VHDL for case Statement in if Statement

```
library IEEE;
use IEEE.std_logic_1164.all;

entity case_in_if_01 is
port(A: in std_logic_vector(8 downto 1);
        DATA_is_late_arriving: in std_logic;
        sel: in std_logic_vector(2 downto 0);
        C: in std_logic_vector(5 downto 1);
        Z: out std_logic);
end case_in_if_01;

architecture improved of case_in_if_01 is
begin
        process(sel, C, A, DATA_is_late_arriving)
        variable Z1: std_logic;
        variable FIRST_IF: boolean;
        begin
             if (C(1) = ’1’) then
                  Z1 := A(5);
             elsif (C(2) = ’0’) then
                  Z1 := A(4);
             elsif (C(3) = ’1’) then
                  Z1 := A(1);
             elsif (C(4) = ’1’) then
                  case sel is
                       when ”010” => Z1 := A(8);
                  -- when ”011” => Z1 := DATA_is_late_arriving;
                       when ”101” => Z1 := A(7);
                       when ”110” => Z1 := A(6);
                       when others => Z1 := A(2);
                  end case;
             elsif (C(5) = ’0’) then
                  Z1 := A(2);
             else
                  Z1 := A(3);
             end if;

             FIRST_IF := (C(1) = ’1’) OR (C(2) = ’0’) OR (C(3) = ’1’);

             if (NOT FIRST_IF AND (C(4) = ’1’) AND (sel = ”011”)) then
                  Z <= DATA_is_late_arriving;
             else
                  Z <= Z1;
             end if;
        end process;
end improved;
```

Example 2-10 is an improved version of the VHDL shown in Example 2-8.

The structure implied by the modified HDL in Example 2-9 and Example 2-10 is given in Figure 2-4.

Figure 2-4 Structure Implied by Improved HDL in Example 2-9, Example 2-10



case Statement With Nested if Statement: Late Arriving Control Signal

Example 2-11 and Example 2-12 show Verilog and VHDL for an if statement nested in a case statement. These examples assume `sel[1]` is a late arriving signal.

Example 2-11 Original Verilog for if Statement in case Statement

```
module if_in_case(sel, X, A, B, C, D, Z);
input [2:0] sel;  // sel[1] is late arriving
input X, A, B, C, D;
output Z;
reg Z;

always @(sel or X or A or B or C or D)
begin
    case (sel)
        3’b000: Z = A;
        3’b001: Z = B;
        3’b010: if (X == 1’b1)
                     Z = C;
                else
                     Z = D;
        3’b100: Z = A ^ B;
        3’b101: Z = !(A && B);
        3’b111: Z = !A;
        default: Z = !B;
    endcase
end
endmodule
```

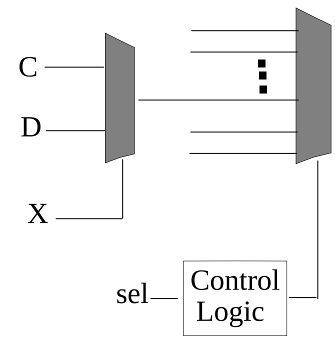
Example 2-12 Original VHDL for if Statement in case Statement

```
library IEEE;
use IEEE.std_logic_1164.all;

entity if_in_case is
port(sel: in std_logic_vector(2 downto 0);
          -- sel(1) is late arriving
     X, A, B, C, D: in std_logic;
     Z: out std_logic);
end;

architecture orig of if_in_case is
begin
     process(sel, X, A, B, C, D)
     begin
          case sel is
               when ”000” => Z <= A;
               when ”001” => Z <= B;
               when ”010” =>if (X = ’1’) then
                                   Z <= C;
                              else
                                   Z <= D;
                              end if;
               when ”100” => Z <= A XOR B;
               when ”101” => Z <= A NAND B;
               when ”111” => Z <= NOT A;
               when others => Z <= NOT B;
          end case;
     end process;
end orig;
```

Figure 2-5 shows pseudo-HDL for an if statement nested in a case statement and the structure implied by this HDL.

Figure 2-5   Structure Implied by if Statement Nested in case Statement

```
case(sel)
  . . .
  s010;
     if (X='1')
        Z = C;
     else Z = D;
  . . .
  s101;
  . . .
endcase
```



In Figure 2-5, due to the control logic, there is a large delay between `sel` and the output Z. If `sel[1]` is a late arriving input, you want to minimize this delay.

In Example 2-11 and Example 2-12, you know that `sel[1]` is a late arriving input. Therefore, you should restructure the HDL in Example 2-11 and Example 2-12 to get the best startpoint for synthesis.

Example 2-13 and Example 2-14 show the modified Verilog and VHDL. The HDL has been modified to shift the dependency on `sel[1]` closer to the output; that is, to move `sel[1]` closer to the output, the nested if statement has been removed and placed outside the case statement.

Example 2-13 Improved Verilog for if Statement in case Statement

```
module if_in_case_improved(sel, X, A, B, C, D, Z);
input [2:0] sel;  // sel[1] is late arriving
input X, A, B, C, D;
output Z;
reg Z;
reg Z1, Z2;
reg [1:0] i_sel;

always @ (sel or X or A or B or C or D)
begin
     i_sel = {sel[2],sel[0]};
     case (i_sel)  // for sel[1]=0
          2'b00: Z1 = A;
          2'b01: Z1 = B;
          2'b10: Z1 = A ^ B;
          2'b11: Z1 = !(A && B);
          default: Z1 = !B;
     endcase

     case (i_sel)  // for sel[1]=1
          2'b00:if (X == 1'b1)
                         Z2 = C;
                    else
                         Z2 = D;
          2'b11: Z2 = !A;
          default: Z2 = !B;
     endcase
     if (sel[1])
          Z = Z2;
     else
          Z = Z1;
end
endmodule
```

Example 2-14 Improved VHDL for if Statement in case Statement

```
library IEEE;
use IEEE.std_logic_1164.all;

entity if_in_case is
port(sel: in std_logic_vector(2 downto 0);
             -- sel(1) is late arriving
         X, A, B, C, D: in std_logic;
         Z: out std_logic);
end if_in_case;

architecture improved of if_in_case is
begin
    process(sel, X, A, B, C, D)
    variable Z1, Z2: std_logic;
    variable i_sel: std_logic_vector(1 downto 0);
    begin
        i_sel := sel(2) & sel(0);
        case i_sel is   -- for sel(1)= ’0’
            when ”00” => Z1 := A;
            when ”01” => Z1 := B;
            when ”10” => Z1 := A XOR B;
            when ”11” => Z1 := A NAND B;
            when others => Z1 := NOT B;
        end case;

        case i_sel is   -- for sel(1)= ’1’
            when ”00” => if (X = ’1’) then
                             Z2 := C;
                         else
                             Z2 := D;
                         end if;
            when ”11” => Z2 := NOT A;
            when others => Z2 := NOT B;
        end case;

        if (sel(1) = ’1’) then
            Z <= Z2;
        else
            Z <= Z1;
        end if;
```

```
     end process;
end improved;
```

Table 2-1 shows the area and timing results for the original and improved versions of the designs in Example 2-11 through Example 2-14. The timing results are for the path from `sel[1]` to `Z`.

Table 2-1   Timing and Area Results for if Statement in case Statement

	Data Arrival Time	**Area**
Original Design	2.81	47.7
Improved Design	2.36	44.4

Notice the improvements in timing and area.

3

Coding Styles for Logic Building Blocks

This chapter shows different coding styles for logic building blocks such as decoders and priority encoders. A typical coding style and a recommended coding style are presented for each building block.

The examples in this chapter are parameterizable: They can be modified for any bit-width. Therefore, they may appear more complex than examples written for a specific bit-width.

This chapter contains the following sections:

- Decoder
- Priority Encoder
- Reduction XOR
- Multiplexer

Decoder

Example 3-1 and Example 3-2 show Verilog and VHDL with a frequently used coding style for decoders. The input is used as an index to the output in these examples.

Example 3-1 Verilog for Decoder Using Indexing

```
module decoder_index (in1, out1);
parameter N = 8;
parameter log2N = 3;
input [log2N-1:0] in1;
output [N-1:0] out1;
reg [N-1:0] out1;
always @(in1)
begin
    out1 = 0;
    out1[in1] = 1’b1;
end
endmodule
```

Example 3-2 VHDL for Decoder Using Indexing

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

entity decoder38_index is
generic (N: natural := 8; log2N: natural := 3);
port (in1: in std_logic_vector(log2N-1 downto 0);
      out1: out std_logic_vector(N-1 downto 0));
end decoder38_index;

architecture one of decoder38_index is
signal in1_int: natural range 0 to N-1;
begin
    in1_int <= CONV_INTEGER(in1);
    process(in1_int)
    begin
        out1 <= (others => ’0’);
        out1(in1_int) <= ’1’;
    end process;
end one;
```

Example 3-3 and Example 3-4 show an alternative coding style for decoders, using a `for` loop.

Example 3-3 Verilog for Decoder Using Loop

```
module decoder38_loop (in1, out1);
parameter N = 8;
parameter log2N = 3;
input [log2N-1:0] in1;
output [N-1:0] out1;
reg [N-1:0] out1;
integer i;

always @(in1)
begin
    for(i=0;i<N;i=i+1)
        out1[i] = (in1 == i);
end
endmodule
```

Example 3-4 VHDL for Decoder Using Loop

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

entity decoder_loop is
generic (N: natural := 8; log2N: natural := 3);
port (in1: in std_logic_vector(log2N-1 downto 0);
      out1: out std_logic_vector(N-1 downto 0));
end decoder_loop;

architecture one of decoder_loop is
signal in1_int: natural range 0 to N-1;
begin
    in1_int <= CONV_INTEGER(in1);
    process(in1_int)
    begin
        out1 <= (others => '0');
        for i in 0 to N-1 loop
            if (in1_int = i) then
                out1(i) <= '1';
            end if;
        end loop;
    end process;
end one;
```

Table 3-1 and Figure 3-1 show timing results for different-size decoders, using the decoder coding styles described in the preceding examples.

Table 3-1 Timing Results for Decoder Coding Styles

Input Address Width	**3**	**4**	**5**	**6**	**7**	**8**
Index	0.64	0.86	1.33	1.52	2.11	2.37
Loop	0.64	0.86	1.33	1.57	1.98	2.10

Figure 3-1   Decoder Timing Results Versus Address Width

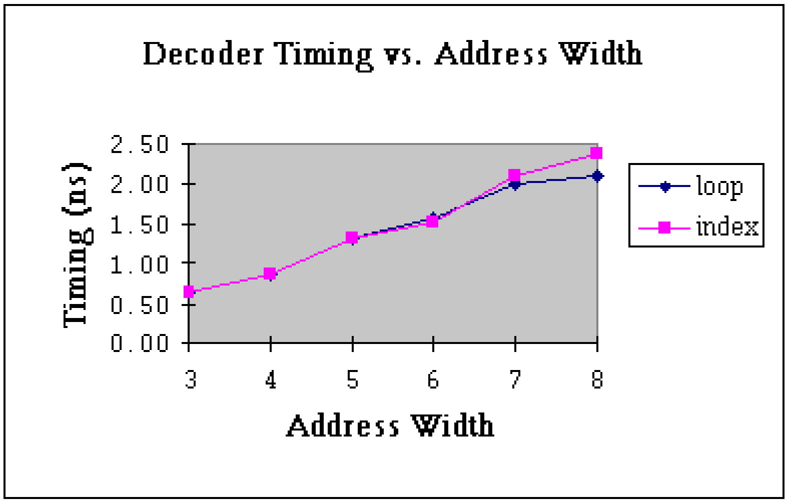


Table 3-2 and Figure 3-2 show area results for the decoder coding styles described in the preceding examples.

Table 3-2   Area Results for Decoder Coding Styles

Input Address Width	**3**	**4**	**5**	**6**	**7**	**8**
Index	18	29	61	115	195	583
Loop	18	30	61	116	195	346

Figure 3-2 Decoder Area Versus Address Width

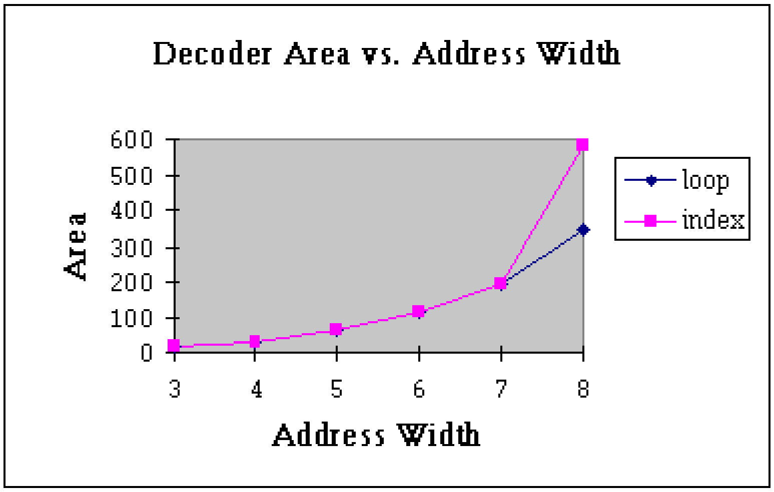
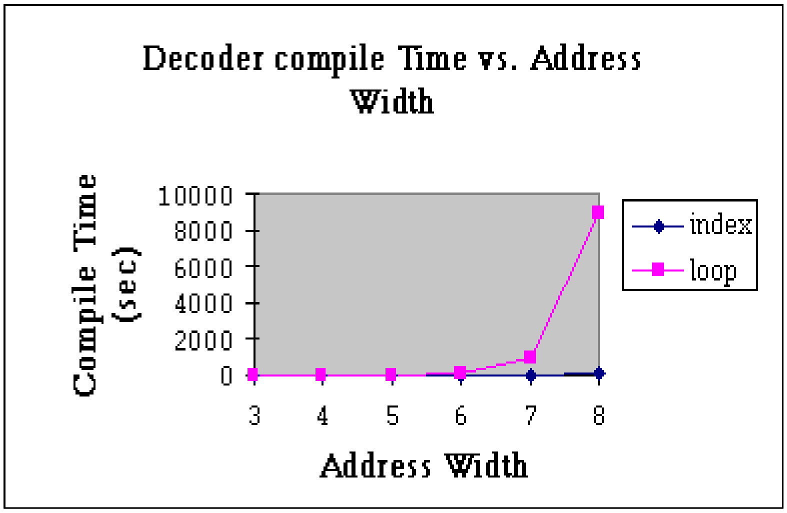


Table 3-3 and Figure 3-3 show compile time for the decoder coding styles described previously.

Table 3-3 Compile Time (Seconds) for Decoder Coding Styles

Input Address Width	3	4	5	6	7	8
Index	2	3	11	18	58	132
Loop	16	13	42	163	946	9000

Figure 3-3 Decoder Compile Times Versus Address Width



In conclusion, Example 3-1 and Example 3-2, using indexing, are more concise and readable than the other examples and have faster compile time overall. On the other hand, Example 3-3 and Example 3-4, using a `for` loop, give slightly better timing results for address widths larger than 6 and better area results for address widths larger than 7. Select a specific coding style based on design requirements (decoder size required and so on).

Priority Encoder

Example 3-5 and Example 3-6 show Verilog and VHDL versions of an 8-to-3 priority encoder using a `for` loop. A function is used in the Verilog example to calculate the highest priority index. A procedure is used in the VHDL example because procedures can have multiple return values.

Example 3-5   Verilog for Priority Encoder Using Loop Starting With Lowest-Priority Bit

```
module priority_low_high (A, P, F);
parameter N = 8;
parameter log2N = 3;
input [N-1:0] A; //Input Vector
output [log2N-1:0] P; // High Priority Index
output F; // Found a one?
reg [log2N-1:0] P;
reg F;

function [log2N:0] priority;
input [N-1:0] A;
reg F;
integer I;
begin
     F = 1'b0;
     priority = {3'b0, F};
     for (I=0; I<N; I=I+1)
          if (A[I])
          begin
               F = 1'b1;
               priority = {I, F};// Override previous index
          end
end
endfunction

always @(A)
begin
     {P, F} <= priority(A);
end
endmodule
```

Example 3-6 is the equivalent VHDL example. This example uses a function, `log2`, to calculate log base 2 and a procedure to find the highest-priority index.

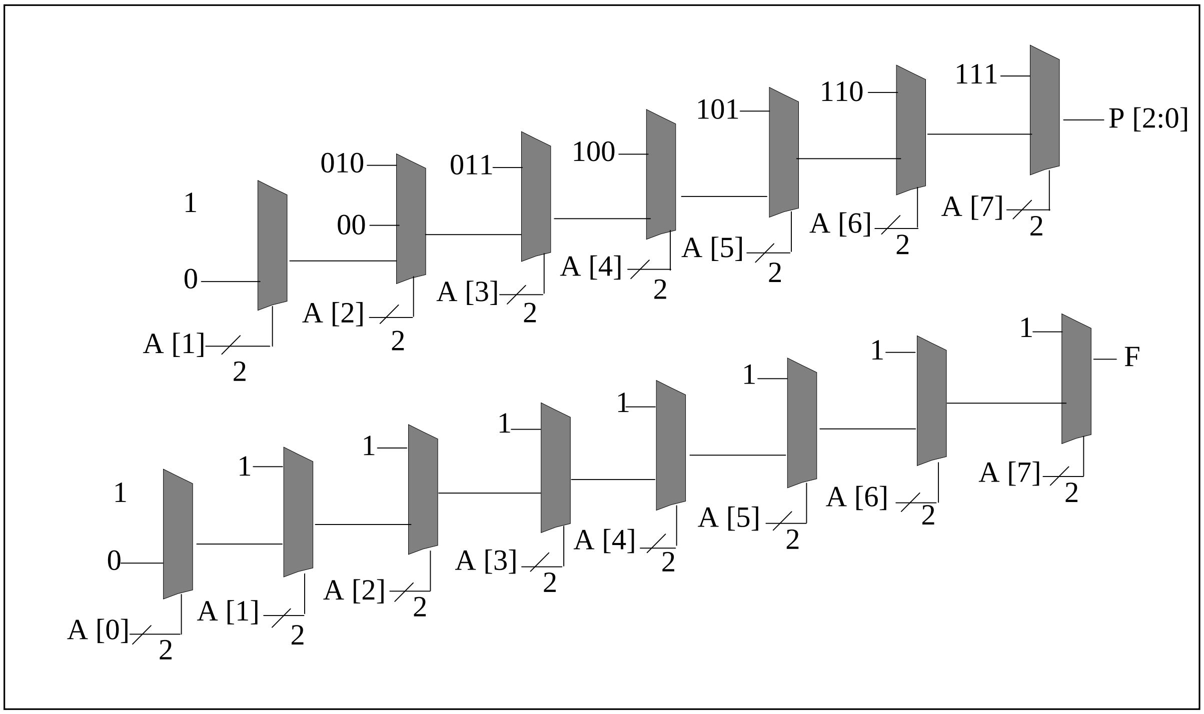
Example 3-6 VHDL for Priority Encoder Using Loop Starting With Lowest-Priority Bit

```
package pri_pack is
     function log2(A: natural) return natural;
end pri_pack;

package body pri_pack is
     function log2(A: natural) return natural is
     begin
          for I in 1 to 30 loop -- Works for up to 32 bit
integers
               if (2**I > A) then
                    return(I-1);
               end if;
          end loop;
          return(30);
     end;
end pri_pack;

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

entity priority_low_high is
generic(N: natural := 3);
port (A: in std_logic_vector(2**N - 1 downto 0);
      P: out std_logic_vector(N-1 downto 0);
      F: out std_logic);
end priority_low_high;

architecture a of priority_low_high is
procedure priority(A: in std_logic_vector;-- Input Vector
               P: out std_logic_vector;-- High Priority
Index
               F: out std_logic) isb  -- Found a one?
constant WIDTH: NATURAL := A'length;
constant LOG_WIDTH: NATURAL := log2(WIDTH);
begin
     P := CONV_STD_LOGIC_VECTOR(CONV_UNSIGNED(0,
LOG_WIDTH),
```

```
            LOG_WIDTH); -- Set output default
        F := '0';
        for I in 0 to WIDTH-1 loop
            if(A(I) = '1') then-- If found a '1'
                P := CONV_STD_LOGIC_VECTOR(CONV_UNSIGNED(I,
LOG_WIDTH),
                    LOG_WIDTH);  -- Override previous index
                F := '1';
            end if;
        end loop;
end priority;

begin
    process(A)
    variable PV: std_logic_vector(N-1 downto 0);
    variable FV: std_logic;

    begin
        priority(A, PV, FV);
        P <= PV;
        F <= FV;
    end process;
end a;
```

Figure 3-4 shows the chain structure implied by the HDL in Example 3-5 and Example 3-6.

Figure 3-4   Chain Structure for Priority Encoder



Example 3-5 and Example 3-6 can be modified to create a tree structure. Tree structures generally result in better performance. Only the VHDL tree structure is shown in Example 3-7.

Using recursion in VHDL gives you the ability to create a priority encoder in a tree structure. Example 3-7 uses recursive procedure calls to generate a tree structure for a priority encoder.

Example 3-7 VHDL for Priority Encoder Tree

```
package pri_pack2 is
        function log2(A: integer) return integer;
        function max(A,B: integer) return integer;
end pri_pack2;

package body pri_pack2 is
        function max(A,B: integer) return integer is
        begin
             if(A<B) then
                  return(B);
             else
                  return(A);
             end if;
        end;

        function log2(A: integer) return integer is
        begin
             for I in 1 to 30 loop-- Works for up to 32 bit integers
                  if(2**I > A) then
                       return(I-1);
                  end if;
             end loop;
             return(30);
        end;
end pri_pack2;

library IEEE;
use IEEE.std_logic_1164.all;
use work.pri_pack2.all;

entity priority_tree is
        port (A: in std_logic_vector(2**N - 1 downto 0);
             P: out std_logic_vector(N-1 downto 0);
             F: out std_logic);
        end priority_tree;

architecture a of priority_tree is

procedure priority(A: in std_logic_vector;    -- Input Vector
                   P: out std_logic_vector;   -- High Priority Index
                   F: out std_logic) is       -- Found a one?
constant WIDTH: INTEGER := A'length;
constant LOG_WIDTH: INTEGER := log2(WIDTH);
variable AT: std_logic_vector(WIDTH-1 downto 0);
variable F1, F0: std_logic;
variable PRET: std_logic_vector(LOG_WIDTH-1 downto 0);
variable P1, P0, PT:std_logic_vector(max(LOG_WIDTH-2,0) downto 0);
```

```
begin
        AT := A;    -- Normalize array indices

        -- Handle Degenerate case of single input

        if(WIDTH = 1) then
              F := AT(0);

        -- This is the bottom of the recursion: a 2-bit priority encoder

        elsif(WIDTH = 2) then
              PRET(0) := AT(0);
              F := AT(1) or AT(0);

        -- Recurse on the two halves, and compute combined result

        else
              priority(AT(WIDTH-1 downto WIDTH/2), P1, F1);
              priority(AT(WIDTH/2-1 downto 0), P0, F0);

              F := F1 or F0;

              if(F1 = '1') then     --If the first half had a '1', use its index.
                    PT := P1;
              else
                    PT := P0;       -- Otherwise, use the second half's index.
              end if;
              PRET := F1 & PT;      -- The result MSB is '1' if the first half
                                    -- had a '1'.
        end if;
        P := PRET;
end priority;

begin
        process(A)
        variable PV: std_logic_vector(N-1 downto 0);
        variable FV: std_logic;
        begin
              priority(A, PV, FV);
              P <= PV;
              F <= FV;
        end process;
end a;
```

Table 3-4 shows timing results for different-size priority encoders using the coding styles described in the preceding example.

Table 3-4 Timing Results for Various Encoder Coding Styles

Output Width	2	3	4	5	6
low_high	0.51	1.29	1.85	3.93	6.00
tree	0.51	1.48	1.71	2.85	4.05

Table 3-5 shows area results for different-size priority encoders using the coding styles described previously.

Table 3-5 Area Results for Various Encoder Coding Styles

Output Width	2	3	4	5	6
low_high	17	43	101	227	536
tree	17	33	90	168	419

Figure 3-5 shows the tree structure implied by the HDL in Example 3-7 on page 3-12.

Figure 3-5 Tree Structure for Priority Encoder

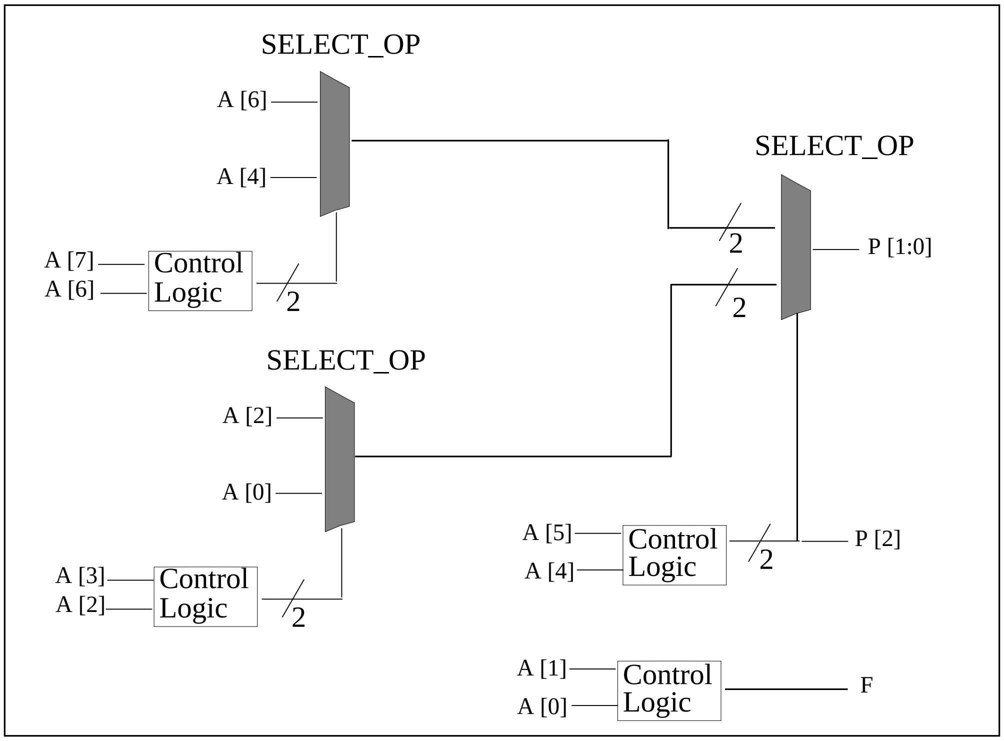
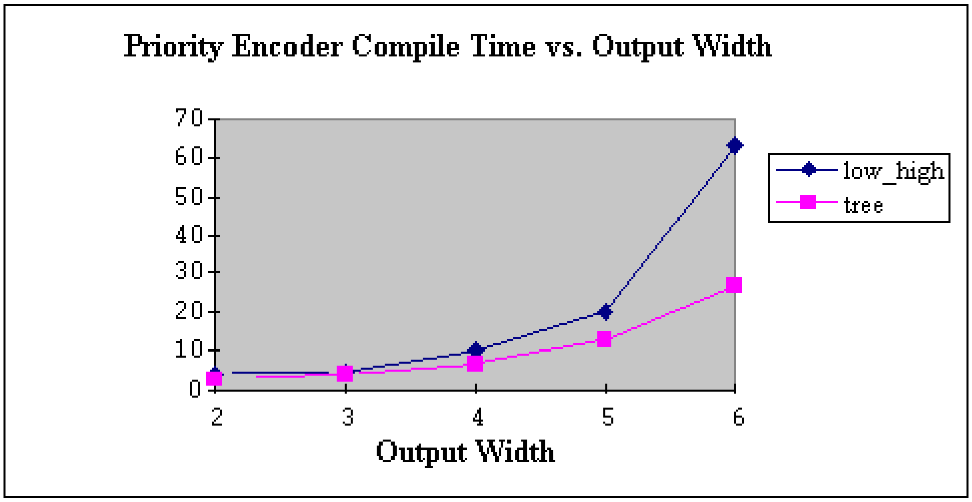


Table 3-6 and Figure 3-6 on page 3-16 show compile time in seconds for the various priority encoder coding styles.

Table 3-6 Compile Time (Seconds) for Various Encoder Coding Styles

Output Width	**2**	**3**	**4**	**5**	**6**
low_high	4	5	10	20	63
tree	3	4	7	13	27

Figure 3-6 Priority Encoder Compile Time Versus Output Width



In conclusion, Example 3-5 on page 3-8 and Example 3-6 on page 3-9, using loops to override the previous index, are more concise and readable. But the tree version in Example 3-7 on page 3-12 is better with respect to timing, area, and compile time. In addition, the QOR difference between the two versions increases as the size of the priority encoder gets larger. For designs that are pushing performance, the tree version is the recommended coding style.

Reduction XOR

Reduction functions, especially reduction `XOR`s, are frequently used in designs. Example 3-8 and Example 3-9 show Verilog and VHDL for a reduction `XOR` implemented in a chain structure. Chain structures are common for reduction functions.

Example 3-8 Verilog for Reduction XOR Chain

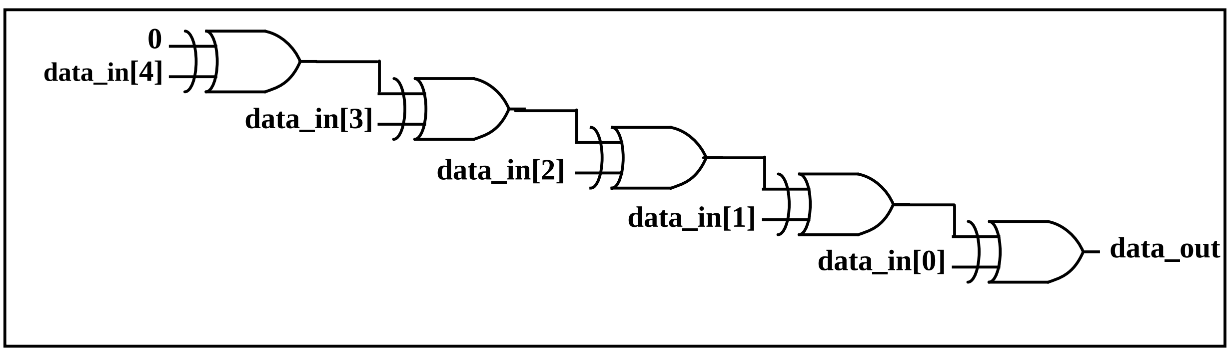
```
module XOR_reduce (data_in, data_out);
parameter N = 5;
input [N-1:0] data_in;
output data_out;
reg data_out;

function XOR_reduce_func;
input [N-1:0] data;
integer I;
begin
    XOR_reduce_func = 0;
    for (I = N-1; I >= 0; I=I-1)
        XOR_reduce_func = XOR_reduce_func ^ data[I];
end
endfunction

always @(data_in)
begin
    data_out <= XOR_reduce_func(data_in);
end
endmodule
```

Example 3-9 VHDL for Reduction XOR Chain

```
library IEEE;
use IEEE.std_logic_1164.all;

entity XOR_reduce is
generic (N: natural := 5);
port (data_in: in std_logic_vector(N-1 downto 0);
        data_out: out std_logic);
end XOR_reduce;

architecture one of XOR_reduce is
function XOR_reduce_func(data:std_logic_vector)return std_logic is
variable result : std_logic;
begin
        result := '0';
        for I in data'RANGE loop
             result := result XOR data(I);
        end loop;
        return result;
end;

begin
        data_out <= XOR_reduce_func(data_in);
end one;
```

Figure 3-7 shows the chain structure implied by the HDL in Example 3-8 and Example 3-9.

Figure 3-7 Chain Structure for Reduction XOR



Example 3-10 on page 3-19 and Example 3-11 on page 3-21 show Verilog and VHDL, respectively, for the reduction XOR implemented in a tree structure.

Example 3-10 Verilog for XOR Tree

```
module XOR_tree(data_in, data_out);
parameter N = 5;
parameter logN = 3;
input[N-1:0] data_in;
output data_out;
reg data_out;

function even;
input [31:0] num;
begin
    even = ~num[0];
end
endfunction

function XOR_tree_func;
input [N-1:0] data;
integer I, J, K, NUM;
reg [N-1:0] temp, result;
begin
    temp[N-1:0] = data_in[N-1:0];
    NUM = N;
    for (K=logN-1; K>=0; K=K-1)
    begin
        J = (NUM+1)/2;
        J = J-1;
        if (even(NUM))
            for (I=NUM-1; I>=0; I=I-2)
            begin
                result[J] = temp[I] ^ temp[I-1];
                J = J-1;
            end
        else
        begin

            for (I=NUM-1; I>=1; I=I-2)
            begin
                result[J] = temp[I] ^ temp[I-1];
                J = J-1;
            end
            result[0] = temp[0];
```

```
            end
            temp[N-1:0] = result[N-1:0];
            NUM = (NUM+1)/2;
      end
      XOR_tree_func = result[0];
end
endfunction

always @(data_in)
begin
      data_out <= XOR_tree_func(data_in);
end
endmodule
```

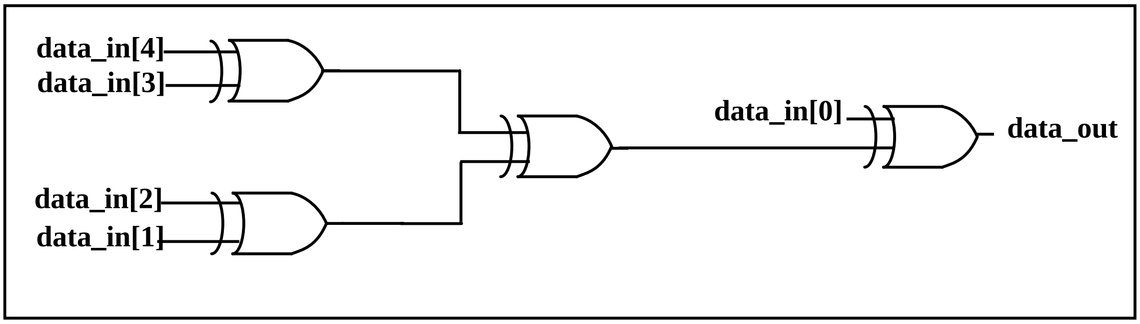
The VHDL version shown in Example 3-11 uses recursion to build a tree.

Example 3-11 VHDL for XOR Tree

```
library IEEE;
use IEEE.std_logic_1164.all;

entity XOR_tree is
generic (N: natural := 4);
port (data_in: in std_logic_vector(N downto 0);
        data_out: out std_logic);
end XOR_tree;

architecture one of XOR_tree is
function XOR_tree_func(data: std_logic_vector) return std_logic is
variable UPPER_TREE, LOWER_TREE: std_logic;
variable MID, LEN: natural;
variable result: std_logic;
variable i_data: std_logic_vector(data'LENGTH-1 downto 0);
begin
        i_data := data;
        LEN := i_data'LENGTH;
        if LEN = 1 then
             result := i_data(i_data'LEFT);
        elsif LEN = 2 then
             result := i_data(i_data'LEFT) XOR i_data(i_data'RIGHT);
        else
             MID := (LEN + 1)/2 + i_data'RIGHT;
             UPPER_TREE := XOR_tree_func(i_data(i_data'LEFT downto MID));
             LOWER_TREE := XOR_tree_func(i_data(MID-1 downto i_data'RIGHT));
             result := UPPER_TREE XOR LOWER_TREE;
        end if;
             return result;

end;
begin
        data_out <= XOR_tree_func(data_in);
end one;
```

Figure 3-8 shows the tree structure implied by the HDL in Example 3-10 and Example 3-11.

Figure 3-8   Tree Structure for Reduction XOR



In conclusion, Design Compiler can convert the `XOR` chain structure to a tree structure during compile. However, it does not do so if the chain is used in a design that accesses intermediate points in the chain (outputs of gates along the chain). Therefore, it is best to start with the tree structure. `OR` chains with intermediate points, on the other hand, are converted to trees.

Multiplexer

Example 3-12 on page 3-23 and Example 3-13 on page 3-24 show, respectively, Verilog and VHDL for multiplexer chains. The structure implied by the HDL is a chain of multiplexing logic. This does not mean Design Compiler necessarily infers multiplexer cells for this logic. For information on how to get Design Compiler to map to multiplexer cells (especially large multiplexer cells) in the technology library, see the *HDL Compiler for Verilog Reference Manual*.

Example 3-12 Verilog for Multiplexer Chain

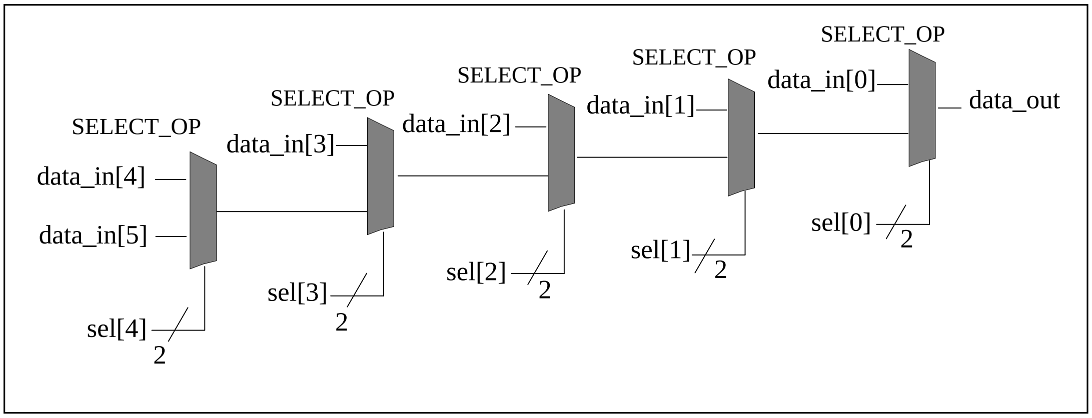
```
module mux_chain (sel, data_in, data_out);
parameter N = 5;
input [N-1:0] sel;
input [N:0] data_in;
output data_out;
reg data_out;

function mux_chain_func;
input [N-1:0] sel;
input [N:0] data_in;
reg [N-1:0] i_sel;
reg [N:0] i_data;
reg result;
integer I;
begin
     i_sel = sel;
     i_data = data_in;
     mux_chain_func = i_data[N];
     for (I = N-1; I >= 0; I=I-1)
          if (i_sel[I])
               mux_chain_func = i_data[I];
end
endfunction

always @(sel or data_in)
begin
     data_out <= mux_chain_func(sel, data_in);
end
endmodule
```

Example 3-13 VHDL for Multiplexer Chain

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux_chain is
generic (N: natural := 5);
port (sel: in std_logic_vector(N-1 downto 0);
     data_in: in std_logic_vector(N downto 0);
     data_out: out std_logic);
end mux_chain;

architecture one of mux_chain is
function mux_chain_func(sel, data: std_logic_vector)
         return std_logic is
variable i_sel: std_logic_vector(sel'LENGTH-1 downto 0);
variable i_data: std_logic_vector(data'LENGTH-1 downto 0);
variable result: std_logic;
begin
     i_sel:= sel;
     i_data:= data;
     result:= i_data(i_data'LEFT);
     for I in i_sel'LENGTH - 1 downto 0 loop
          if i_sel(I) = '1' then
               result := i_data(I);
          end if;
     end loop;
     return result;
end;

begin
     data_out <= mux_chain_func(sel, data_in);
end one;
```

Figure 3-9 shows the structure implied by the HDL in Example 3-12 and Example 3-13.

Figure 3-9 Structure Implied by Multiplexer Chain Example



Example 3-14 and Example 3-15 show Verilog and VHDL that imply the same multiplexing functionality shown in Example 3-12 and Example 3-13 but in a tree structure rather than a chain structure.

Example 3-14 Verilog for Multiplexer Tree

```
module mux_tree(sel, data_in, data_out);
parameter N = 8;
parameter log2N = 3;
input [N-2:0] sel;
input [N-1:0] data_in;
output data_out;
reg data_out;

function even;
input [31:0] num;
begin
        even = ~num[0];
end
endfunction

function mux_2_1;
input sel;
input [1:0]data;

begin
        if (sel)
              mux_2_1 = data[0];
        else
              mux_2_1 = data[1];
end
endfunction

function mux_tree_func;
input [N-2:0] sel;
input [N-1:0] data_in;
reg [N-1:0] i_sel, temp_sel;
reg [N-1:0] i_data, result;
integer I, J, K, S;
integer TREE_DEPTH;
integer SEL_LEN, DATA_LEN;

begin
        i_data[N-1:0] = data_in[N-1:0];
        i_sel[N-2:0] = sel[N-2:0];
        i_sel[N-1] = 1'b0;
        DATA_LEN = N;
        SEL_LEN  = N-1;
        for (TREE_DEPTH=log2N-1; TREE_DEPTH>=0;
              TREE_DEPTH=TREE_DEPTH-1)
        begin
              SEL_LEN  = (DATA_LEN+1)/2;
              S = SEL_LEN-1;
```

```
            J = (DATA_LEN+1)/2;
            J = J-1;
            if (even(DATA_LEN))
            for (I=DATA_LEN-1; I>=1; I=I-2)
            begin
                 result[J] = mux_2_1(i_sel[I-1], {i_data[I],
                           i_data[I-1]});
                 temp_sel[S] = |{i_sel[I-1], i_sel[I]};
                 J = J-1;
                 S = S-1;
            end
            else
            begin
            for (I=DATA_LEN-1; I>=2; I=I-2)
            begin
                 result[J] = mux_2_1(i_sel[I-1], {i_data[I],
                           i_data[I-1]});
                 temp_sel[S] = |{i_sel[I-1], i_sel[I]};
                 J = J-1;
                 S = S-1;
            end
                 result[0] = i_data[0];
                 temp_sel[0] = i_sel[0];
            end
            i_data[N-1:0] = result[N-1:0];
            i_sel[N-1:0]  = temp_sel[N-1:0];
            DATA_LEN = (DATA_LEN+1)/2;
        end
        mux_tree_func = result[0];
end
endfunction

always @(sel or data_in)
begin
        data_out <= mux_tree_func(sel, data_in);
end
endmodule
```

Example 3-15 VHDL for Multiplexer Tree

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux_tree is
generic (N: natural := 4);
port (sel: in std_logic_vector(N downto 0);
        data_in: in std_logic_vector(N+1 downto 0);
        data_out: out std_logic);
end mux_tree;
architecture one of mux_tree is
function XOR_tree_func...
  -- See Example 3-11 on page 3-20 for XOR_tree_func source
end;
function mux_2_1(sel: std_logic; input: std_logic_vector)
                   return std_logic is
        variable result: std_logic;
        variable i_input: std_logic_vector(1 downto 0);
        begin
             i_input := input;
             if sel = '1' then
                  result := i_input(0);
             else
                  result := i_input(1);
             end if;
                  return result;
        end;

function mux_tree_func(sel, data: std_logic_vector)
                   return std_logic is
        variable result : std_logic;
        variable upper_tree, lower_tree : std_logic;
        variable i_sel : std_logic_vector(sel'LENGTH-1 downto 0);
        variable i_data : std_logic_vector(data'LENGTH-1downto 0);
        variable final_sel : std_logic;
        variable val : std_logic_vector(1 downto 0);
        variable  SEL_LEN, DATA_LEN, MID : natural;
begin
        i_sel   := sel;
        i_data  := data;
        DATA_LEN := i_data'LENGTH;
        SEL_LEN  := i_sel'LENGTH;
        if SEL_LEN = 0 or DATA_LEN = 0 then
        elsif SEL_LEN = 1 then
             result := mux_2_1(i_sel(0), i_data);
        elsif SEL_LEN = 2 then
             upper_tree := mux_2_1(i_sel(1), i_data(2 downto 1));
             val := (upper_tree, i_data(0));
             result := mux_2_1(i_sel(0), val);
```

```
        elsif SEL_LEN = 3 and DATA_LEN = 3 then
             upper_tree := mux_2_1(i_sel(2), i_data(2 downto 1));
             val := (upper_tree, i_data(0));
             final_sel := i_sel(1) OR i_sel(0);
             result := mux_2_1(final_sel, val);
        elsif SEL_LEN = 3 and DATA_LEN = 4 then
             upper_tree := mux_2_1(i_sel(2), i_data(3 downto 2));
             lower_tree := mux_2_1(i_sel(0), i_data(1 downto 0));
             val := (upper_tree, lower_tree);
             final_sel := i_sel(1) OR i_sel(0);
             result := mux_2_1(final_sel, val);
        else
        MID := (DATA_LEN + 1)/2 + i_data'RIGHT;
             upper_tree := mux_tree_func(i_sel(i_sel'LEFT downto MID),
                  i_data(i_data'LEFT downto MID));
             lower_tree := mux_tree_func(i_sel(MID-2 downto i_sel'RIGHT),
                  i_data(MID - 1 downto i_data'RIGHT));
             val := (upper_tree, lower_tree);
             final_sel := XOR_tree_func(i_sel(MID - 1 downto 0));
             result := mux_2_1(final_sel, val);
        end if;
        return result;
end;

begin
        data_out <= mux_tree_func(sel, data_in);
end one;
```

Figure 3-10 shows the structure implied by Example 3-14 and Example 3-15.

Figure 3-10 Structure Implied by Multiplexer Tree Example

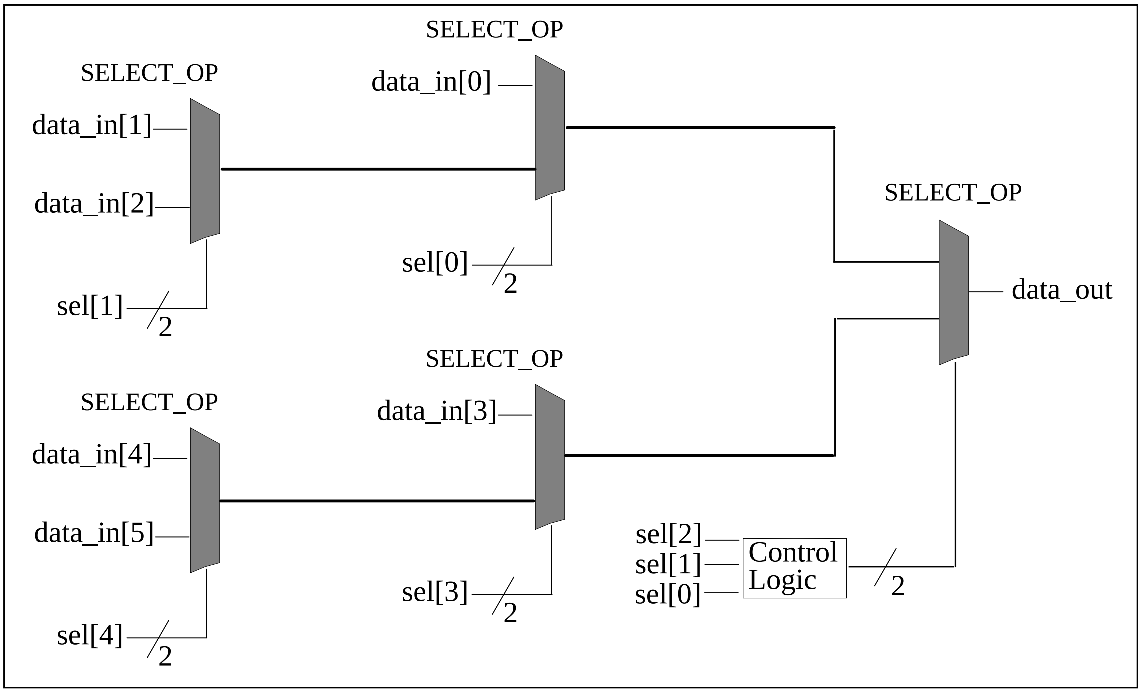


Table 3-7 and Figure 3-11 show timing results for different-size multiplexer chains and trees, using the coding styles described previously.

Table 3-7 Timing Results for Various Multiplexer Coding Styles

# of MUXs	**3**	**4**	**5**	**6**	**7**	**8**
Chain	0.72	1.18	1.3	1.76	1.88	2.34
Tree	0.72	1.01	1.01	1.09	1.29	1.38

Figure 3-11   Multiplexer Timing Versus Number of Multiplexers

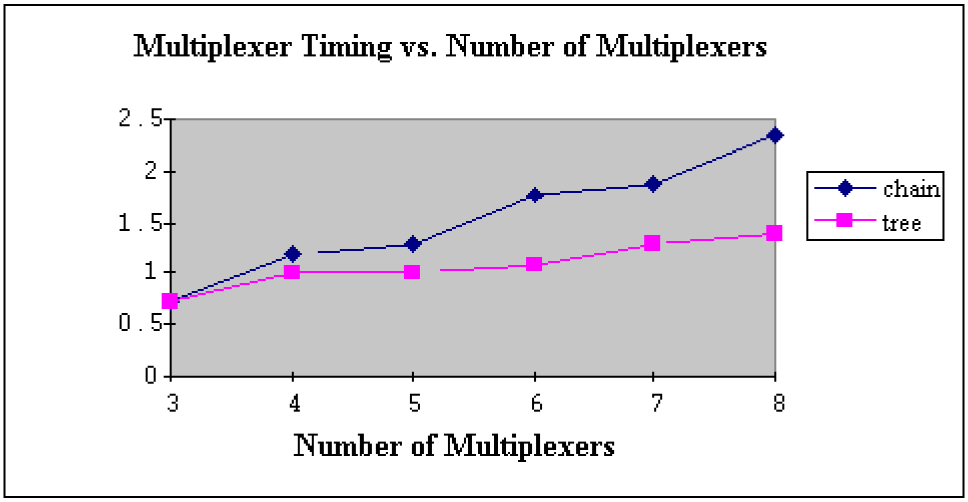
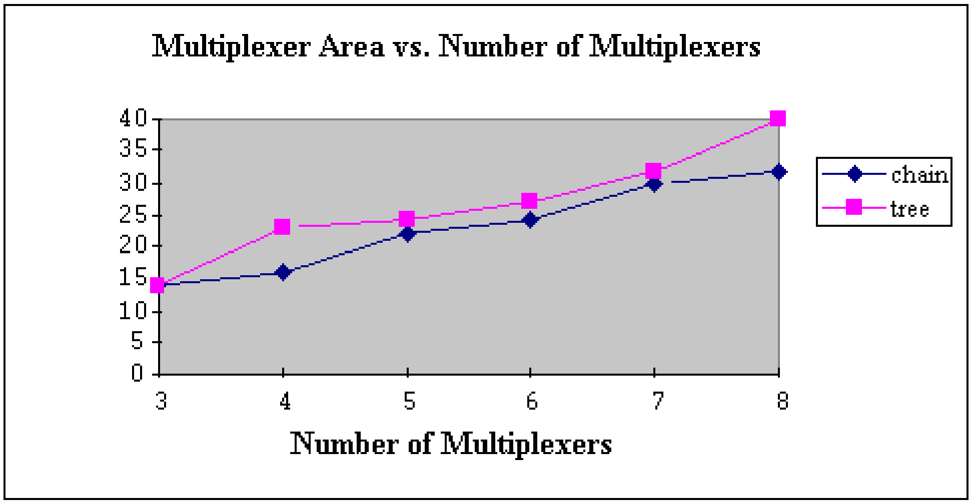


Table 3-8 and Figure 3-12 show area results for different-size multiplexer chains and trees, using the coding styles described previously.

Table 3-8   Multiplexer Area Versus Number of Multiplexers

# of MUXs	**3**	**4**	**5**	**6**	**7**	**8**
Chain	14	16	22	24	30	32
Tree	14	23	24	27	32	40

Figure 3-12   Multiplexer Area Versus Number of Multiplexers



From this data, it is apparent that the tree version is better with respect to timing (as expected) but a little worse with respect to area. To optimize your HDL for timing, use the tree version. If area is of greater concern, use the chain version.

A late arriving signal can also indicate the need for a chain structure. For example, if `data_in[0]` is a late arriving input, the chain structure shown in Figure 3-9 is the better startpoint.

4

High-Performance Coding Techniques

This chapter contains examples utilizing various high-performance coding techniques.

This chapter contains the following sections:

- Data-Path Duplication
- Operator in if Condition

Data-Path Duplication

The following examples illustrate how to duplicate logic in HDL to improve timing.

In Example 4-1 and Example 4-2, `CONTROL` is a late arriving input signal. The goal is to reduce the logic from `CONTROL` to the output port `COUNT`.

Example 4-1 Original Verilog Before Logic Duplication

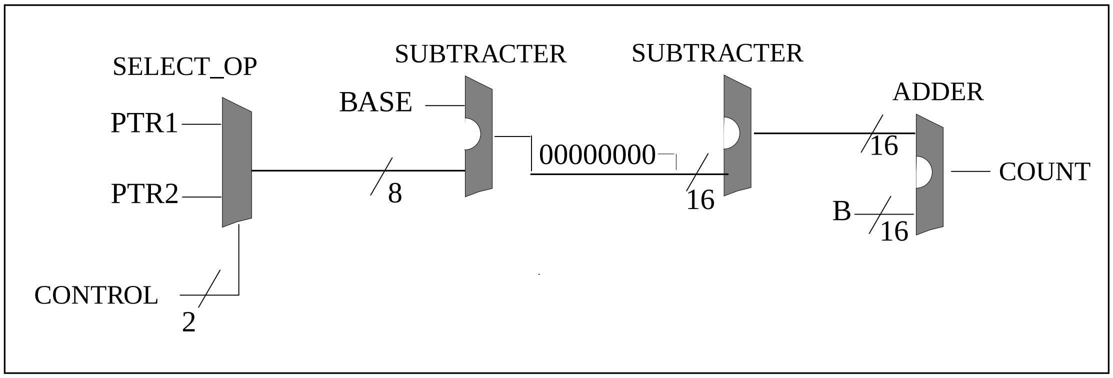
```
module BEFORE (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1,PTR2;
input [15:0] ADDRESS, B;
input CONTROL;       // CONTROL is late arriving
output [15:0] COUNT;

parameter [7:0] BASE = 8'b10000000;
wire [7:0] PTR, OFFSET;
wire [15:0] ADDR;

assign PTR = (CONTROL == 1'b1) ? PTR1 : PTR2;
assign OFFSET = BASE - PTR; //Could be any function
f(BASE,PTR)
assign ADDR = ADDRESS - {8'h00, OFFSET};
assign COUNT = ADDR + B;

endmodule
```

Example 4-2   Original VHDL Before Logic Duplication

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

entity BEFORE is
port(ADDRESS, B : in std_logic_vector (15 downto 0);
        PTR1, PTR2 : in std_logic_vector (7 downto 0);
  CONTROL    : in  std_logic; -- CONTROL is late arriving
        COUNT      : out std_logic_vector (15 downto 0));
end BEFORE;

architecture RTL of BEFORE is
begin
        process (B, CONTROL, ADDRESS, PTR1, PTR2)
        constant BASE : std_logic_vector (7 downto 0) := ”10000000”;
        variable PTR, OFFSET : std_logic_vector (7 downto 0);
        variable ADDR        : std_logic_vector (15 downto 0);
begin
             if CONTROL = ’1’ then
                  PTR := PTR1;
             else
                  PTR := PTR2;
             end if;

             OFFSET := BASE - PTR;  -- Could be any function f(BASE,PTR)
             ADDR := ADDRESS - (”00000000” & OFFSET);

             COUNT <= ADDR + B;
        end process;
end RTL;
```

Figure 4-1 shows the structure implied by the original HDL.

Figure 4-1 Structure Implied by Original HDL Before Logic Duplication



In Figure 4-1, notice that there is a `SELECT_OP` next to a subtracter. When you see a `SELECT_OP` next to an operator, there is a good chance that you can move the `SELECT_OP` to after the operator. You might want to do this if the control signal for the `SELECT_OP` is late arriving. You can move the `SELECT_OP` by duplicating the logic in the branches of the conditional statement that implied the `SELECT_OP`.

In Figure 4-1, you can also see the signal that `CONTROL` selects between two inputs. The selected input drives a chain of arithmetic operations (the data path) and ends at the output port `COUNT`. If `CONTROL` arrives late, you will want to move the selection closer to the output port `COUNT`.

Example 4-3 and Example 4-4 show the improved HDL for Example 4-1 and Example 4-2. The improved HDL shows the data-path duplication described previously.

Example 4-3 Improved Verilog With Data Path Duplicated

```
module PRECOMPUTED (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1, PTR2;
input [15:0] ADDRESS, B;
input CONTROL;
output [15:0] COUNT;

parameter [7:0] BASE = 8’b10000000;
wire [7:0] OFFSET1,OFFSET2;
wire [15:0] ADDR1,ADDR2,COUNT1,COUNT2;

assign OFFSET1 = BASE - PTR1;  // Could be f(BASE,PTR)
assign OFFSET2 = BASE - PTR2;  // Could be f(BASE,PTR)
assign ADDR1 = ADDRESS - {8’h00 , OFFSET1};
assign ADDR2 = ADDRESS - {8’h00 , OFFSET2};
assign COUNT1 = ADDR1 + B;
assign COUNT2 = ADDR2 + B;
assign COUNT = (CONTROL == 1’b1) ? COUNT1 : COUNT2;

endmodule
```

Example 4-4 Improved VHDL With Data Path Duplicated

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

entity PRECOMPUTED is
port (ADDRESS, B : in std_logic_vector (15 downto 0);
    PTR1, PTR2 : in std_logic_vector (7 downto 0);
    CONTROL    : in  std_logic;
    COUNT      : out std_logic_vector (15 downto 0));
end PRECOMPUTED;

architecture RTL of PRECOMPUTED is
begin
    process (CONTROL, ADDRESS, B, PTR1, PTR2)
    constant BASE : std_logic_vector (7 downto 0) :=
”10000000”;
    variable OFFSET2, OFFSET2 : std_logic_vector (7 downto
0);
    variable ADDR1, ADDR2    : std_logic_vector (15 downto
0);
    variable COUNT1, COUNT2  : std_logic_vector (15 downto
0);
    begin

        OFFSET1 := BASE - PTR1;  -- Could be f(BASE,PTR)
        OFFSET2 := BASE - PTR2;  -- Could be f(BASE,PTR)

        ADDR1 := ADDRESS - (”00000000” & OFFSET1);
        ADDR2 := ADDRESS - (”00000000” & OFFSET2);

        COUNT1 := ADDR1 + B;
        COUNT2 := ADDR2 + B;

        if CONTROL = ’1’ then
            COUNT <= COUNT1;
        else
            COUNT <= COUNT2;
        end if;

    end process;
```

```
end RTL;
```

When you duplicate the operations that depend on the inputs PTR1 and PTR2, the assignment to COUNT becomes a selection between the two parallel data paths. The signal CONTROL selects the data path. The path from CONTROL to the output port COUNT is no longer the critical path, but this change comes at the expense of duplicated logic.

In Example 4-3 and Example 4-4, the entire data path is duplicated, because CONTROL arrives late. Had CONTROL arrived earlier, you could have duplicated only a portion of the logic, thereby decreasing the area expense. The designer controls how much logic is duplicated.

In addition, the amount of duplication is proportional to the number of branches in the conditional statement. For example, if there were four PTR signals in Example 4-1 and Example 4-2 instead of two (PTR1 and PTR2), the area penalty would be larger, because you would have two more duplicated data paths.

Figure 4-2 shows the structure implied by the improved HDL.

Figure 4-2 Structure Implied by Improved HDL With Data Path Duplication

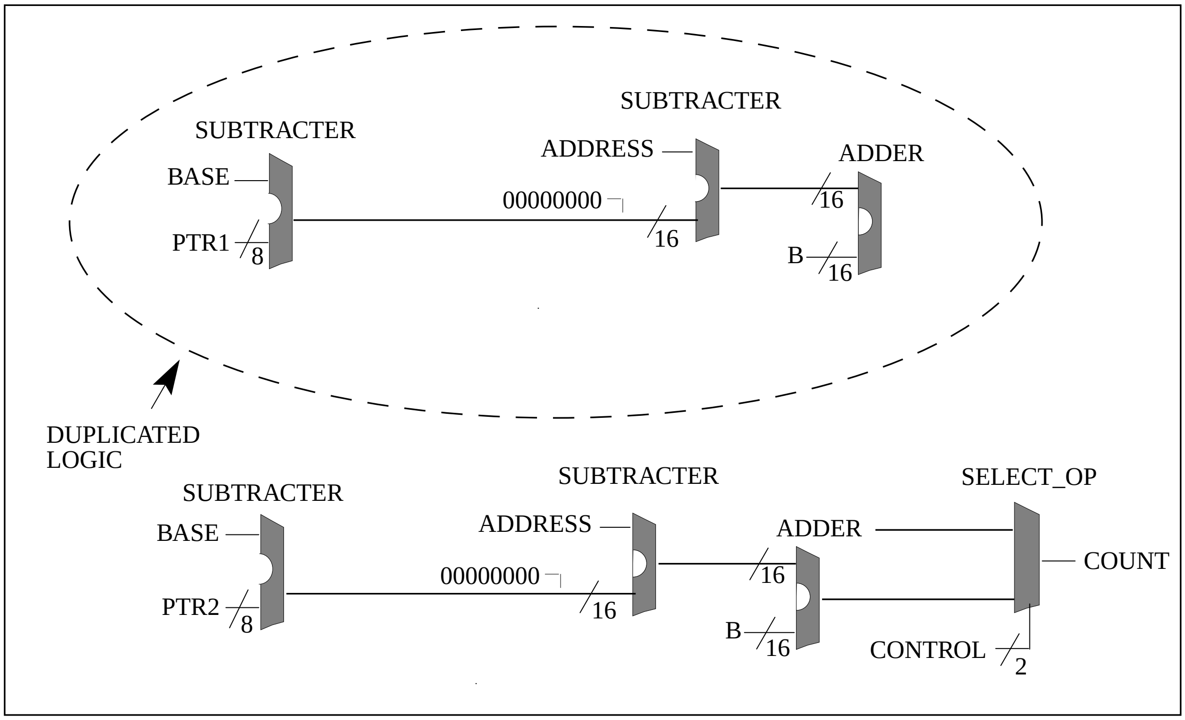


Table 4-1 shows the timing and area results for the original and the improved HDL shown in Example 4-1, Example 4-2, Example 4-3, and Example 4-4. The timing numbers are for the path from CONTROL to COUNT[9], which was the worst path in the original design.

Table 4-1 Timing and Area Results for Data-Path Duplication

	Data Arrival Time	**Area**
Original Design	5.23	1057
Improved Design	2.33	1622

In conclusion, the improved design with the data path duplicated is much better with respect to timing. As expected, the area is worse for the improved design. If you want to optimize your design for timing and are less concerned about area, data-path duplication is the recommended methodology. Note that logic duplication also increases the load on the input pins.

Operator in if Condition

Example 4-5 and Example 4-6 show Verilog and VHDL designs that contain operators in the conditional expression of an if statement. The signal A in the conditional expression is a late arriving signal, so you should move the signal closer to the output.

Example 4-5 Original Verilog With Operator in Conditional Expression

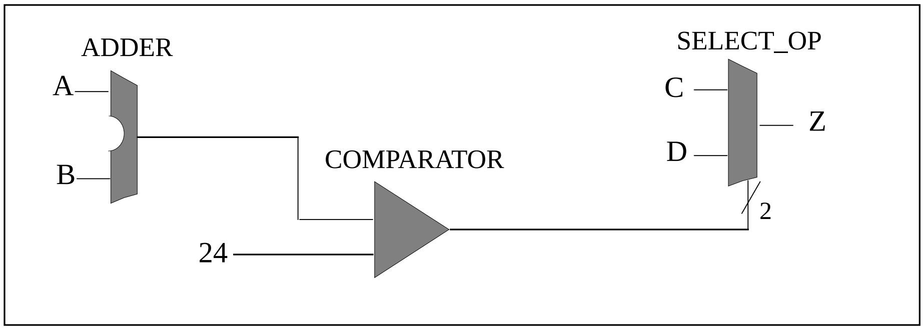
```
module cond_oper(A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D; //A is late arriving
output [N-1:0] Z;

reg [N-1:0] Z;

always @(A or B or C or D)
begin
    if (A + B < 24)
        Z <= C;
    else
        Z <= D;
end

endmodule
```

Example 4-6 Original VHDL With Operator in Conditional Expression

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

entity cond_oper is
generic(N: natural := 8);
port(A, B: in std_logic_vector(N-1 downto 0);
  -- A is late arriving
    C, D: in std_logic_vector(N-1 downto 0);
    Z: out std_logic_vector(N-1 downto 0));
end cond_oper;

architecture one of cond_oper is
begin
    process(A, B, C, D)
    begin
        if (A + B < 24) then
            Z <= C;
        else
            Z <= D;
        end if;
    end process;
end one;
```

Figure 4-3 shows the structure implied by the original HDL in Example 4-5 and Example 4-6. The signal A is an input to the adder in Figure 4-3.

Figure 4-3 Structure Implied by Original HDL With Late Arriving A Signal



You want to reduce the number of operations that have the signal A in their `fanin` cone. Example 4-7 and Example 4-8 show the improved HDL for Example 4-5 and Example 4-6.

Example 4-7 Improved Verilog With Operator in Conditional Expression

```
module cond_oper_improved (A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D; // A is late arriving
output [N-1:0] Z;

reg [N-1:0] Z;

always @(A or B or C or D)
begin
    if (A < 24 - B)
        Z <= C;
    else
        Z <= D;
    end
    endmodule
```

Example 4-8 Improved VHDL With Operator in Conditional Expression

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

entity cond_oper_improved is
generic (N : natural := 8);
port (A, B : in std_logic_vector(N-1 downto 0);
  -- A is late arriving
    C, D : in std_logic_vector(N-1 downto 0);
    Z    : out std_logic_vector(N-1 downto 0));
end cond_oper_improved;

architecture one of cond_oper_improved is
begin
    process(A, B, C, D)
    begin
        if (A < 24 - B) then
            Z <= C;
        else
            Z <= D;
        end if;
    end process;
end one;
```

Figure 4-4 shows the structure implied by the improved HDL. The signal A is an input to the comparator in Figure 4-4.

Figure 4-4 Structure Implied by Improved HDL With A as Input to Comparator

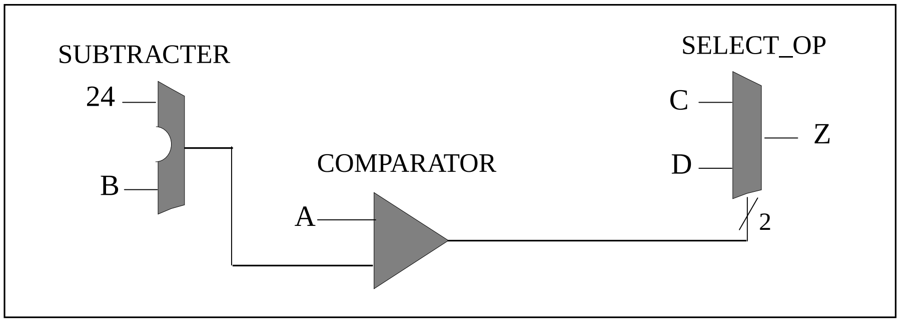


Table 4-2 shows the timing and area results (given that A is a late arriving input) for the original and improved HDL shown in Example 4-5, Example 4-6, Example 4-7, and Example 4-8. The timing results are for the worst path in the design.

Table 4-2 Timing and Area Results for Conditional Operator Examples

	Data Arrival Time	**Area**
Original Design	4.33	411.1
Improved Design	3.89	271.0

5

General Coding Style Guidelines

This chapter lists some general guidelines for writing HDL.

This chapter contains the following sections:

- Unintentional Latch Inference
- Incomplete Sensitivity Lists
- Unnecessary Calculations in for Loops
- Resource Sharing

Unintentional Latch Inference

Incompletely specified if statements and case statements cause the HDL Compiler tool to infer latches. The code segments shown in Example 5-1 and Example 5-2 infer a latch, because the output, `data_out`, is not assigned under all possible conditions.

Example 5-1 Verilog Showing Unintentional Latch Inference

```
always @(cond_1)
begin
    if (cond_1)
        data_out <= data_in;
end
```

Example 5-2 VHDL Showing Unintentional Latch Inference

```
process(cond1)
begin
    if (cond_1 = '1') then
        data_out <= data_in;
    end if;
end process;
```

Example 5-1 and Example 5-2 result in latches because `data_out` is not given a value when `cond_1` is not equal to `'1'`. To prevent HDL Compiler from inferring unintentional latches for these examples, you should make a default assignment to `data_out` outside the if statement or add an `else` branch to the if statement.

VHDL requires case statements to be completely specified (incompletely specified case statements result in a syntax error). In VHDL, latches are inferred if the output signal is not assigned in each branch of the case statement.

Latches are inferred for incompletely specified case statements in Verilog. To prevent this unintentional latch inference in Verilog, specify all possible conditions in the case statement and assign the output signal in each branch of the case statement.

For Verilog, you can also use the HDL Compiler `full_case` directive with caution to tell HDL Compiler that the case statement is fully specified. For additional information on the `full_case` directive, see the *HDL Compiler for Verilog Reference Manual*.

To get HDL Compiler to issue a warning when latches are inferred, set the variable `hdlin_check_no_latch` to true before HDL input. You can also check the inference report after HDL input to see if any latches were inferred.

For additional information on inference reports and HDL examples to infer flip-flops and latches, see the *HDL Compiler for Verilog Reference Manual*.

Incomplete Sensitivity Lists

Incomplete sensitivity lists can cause a simulation/synthesis mismatch. HDL Compiler issues warnings for signals that are read in a process or in an `always` block but are not listed in the sensitivity list. Sensitivity lists do not affect the logic generated by HDL Compiler, but an incomplete sensitivity list can cause unexpected simulation results, because the process does not trigger when necessary.

Consider the Verilog and VHDL code segments in Example 5-3 and Example 5-4.

Example 5-3 Verilog With Missing Signal in Sensitivity List

```
always @(d or clr)
    if (clr)
        q = 1'b0
    else if (e)
        q = d;
```

Example 5-4 VHDL With Missing Signal in Sensitivity List

```
process(d, clr)
begin
    if (clr = '1') then
        q <= '0';
    elsif (e = '1') then
        q <= d;
    end if;
end process;
```

In Example 5-3 and Example 5-4, the signal e is read, but it is not in the sensitivity list. Assuming that `clr` is stable at `0`, a change in e from `0` to `1` does not trigger the `always` block or process, so the value of d does not get latched onto q. This behavior does not match the behavior of the synthesized hardware.

Unnecessary Calculations in for Loops

Avoid placing expressions that do not change inside `for` loops. For VHDL, HDL Compiler unrolls `for` loops, so the structure inferred is repetitive. Moving unchanging expressions outside the loop prevents Design Compiler from spending time optimizing redundant logic.

Example 5-5 is a statement that does not change value in a loop.

Example 5-5 Original VHDL With Unnecessary Statement in Loop

```
for I in 0 to 4 loop
    sig1 <= sig2; -- unchanging statement
    data_out(I) <= data_in(I);
end loop;
```

The unchanging statement should be pulled out of the loop, as shown in Example 5-6.

Example 5-6 Improved VHDL With Statement Pulled out of Loop

```
sig1 <= sig2;
for I in 0 to 4 loop
    data_out(I) <= data_in(I);
end loop;
```

Resource Sharing

Arithmetic operators are shared only if they occur in mutually exclusive branches of if-then-else or case statements. Operators in loops and conditional assignments in Verilog (using the conditional operator ?) are not shared.

Example 5-7 shows a Verilog statement that uses the conditional operator. Resource sharing does not take place for this example.

Example 5-7 No Resource Sharing for Conditional Operator in Verilog

```
z = (cond)?(a+b):(c+d);
```

However, resource sharing does take place for the equivalent if-then-else statement in Example 5-8.

Example 5-8   Resource Sharing for equivalent if-then-else in Verilog

```
if (cond)
     z = a+b;
else
     z = c+d;
```

For additional information about resource sharing, see the *HDL Compiler for Verilog Reference Manual* or the *VHDL Compiler Reference Manual*.