You can generally use several different, but logically equivalent, VHDL descriptions to describe a circuit.

To understand the interaction between VHDL Compiler and Design Compiler, and to write VHDL descriptions that produce efficient synthesized circuits, study the following information:

How Statements Are Mapped to Logic

Design Structure

Adding Structure
Optimizing Arithmetic Expressions
Operator Bit Width
Using State Information
Propagating Constants
Sharing Complex Operators
Design Compiler Optimization

Asynchronous Designs

Don’t-Care Inference

Synthesis Issues

Some general guidelines for writing efficient circuit descriptions are

- Restructure a design that makes repeated use of several large components to minimize the number of instantiations.
- In a design that needs some, but not all, of its variables or signals stored during operation, minimize the number of latches or flip-flops required.
- Consider collapsing hierarchy for more efficient synthesis.

In many circumstances you can improve the quality of synthesized circuits by describing your high-level knowledge of a circuit better. VHDL Compiler cannot always derive details of a circuit architecture. Any additional architectural information that you can provide to VHDL Compiler can result in a more efficient circuit.
How Statements are Mapped to Logic

VHDL descriptions are mapped to combinational logic by creating blocks of logic. A statement or an operator in a VHDL function can represent a block of combinational logic or, in some cases, a latch or register.

When mapping complex operations, such as adders and subtracters, Design Compiler inserts the operations (blocks) in the design as levels of hierarchy.

The statements shown in Example 10–1 represent four logic blocks:

- A comparator that compares the value of $B$ with 10.
- An adder that has $A$ and $B$ as inputs.
- An adder that has $A$ and 10 as inputs.
- A multiplexer (implied by the `if` statement) that controls the final value of $Y$.

**Example 10–1   Four Logic Blocks**

```vhdl
if (B < 10) then
  Y = A + B;
else
  Y = A + 10;
end if;
```

The logic blocks created by VHDL Compiler are custom built for their environment. If $A$ and $B$ are four-bit quantities, a four-bit adder is built. If $A$ and $B$ are nine-bit quantities, a nine-bit adder is built. Because VHDL Compiler incorporates a large set of these customized logic blocks, it can translate most VHDL statements and operators.

**Note:**

If the inputs to an adder or other operator resource are 4 bits or less, the hierarchy is automatically collapsed during the `compile` command.
Design Structure

A design’s structure directly influences the size and complexity of the resulting synthesized circuit. These topics help you understand the concepts:

- Adding structure
- Using design knowledge
- Constant propagation
- Design efficiency
- Design compiler optimization

Adding Structure

VHDL Compiler provides you with significant control over the preoptimization structure, or organization of components, in your design. Whether or not your design structure is preserved after optimization depends upon the Design Compiler options you select (see “Design Compiler Optimization,” later in this chapter). The structure of a VHDL description corresponds directly to VHDL Compiler output.

Using Variables and Signals

You control design structure with your ordering of assignment statements, and your use of variables. Each VHDL signal assignment, process, or component instantiation implies a piece of logic. Each variable or signal implies a wire. By using these constructs, you can connect entities in any configuration.
Example 10–2 and Example 10–3 illustrate two possible descriptions of an adder's carry chain.

**Example 10–2  Ripple-Carry Chain**

-- A is the addend  
-- B is the augend  
-- C is the carry  
-- Cin is the carry in  

\[
\begin{align*}
C_0 & \leq (A_0 \text{ and } B_0) \text{ or } ((A_0 \text{ or } B_0) \text{ and } \text{Cin}); \\
C_1 & \leq (A_1 \text{ and } B_1) \text{ or } ((A_1 \text{ or } B_1) \text{ and } C_0);
\end{align*}
\]

**Example 10–3  Carry-Lookahead Chain**

-- Ps are propagate  
-- Gs are generate  

\[
\begin{align*}
p_0 & \leq a_0 \text{ or } b_0; \\
g_0 & \leq a_0 \text{ and } b_0; \\
p_1 & \leq a_1 \text{ or } b_1; \\
g_1 & \leq a_1 \text{ and } b_1; \\
c_0 & \leq g_0 \text{ or } (p_0 \text{ and } \text{cin}); \\
c_1 & \leq g_1 \text{ or } (p_1 \text{ and } g_0) \text{ or } (p_1 \text{ and } p_0 \text{ and } \text{cin});
\end{align*}
\]
Using Parentheses

Another way to control the structure of a design is by using parentheses to define logic groupings. Example 10-4 and Example 10-5 illustrate two adder groupings.

Example 10-4  Four-Input Adder
Z <= (A + B) + C + D;

Example 10-5  Four-Input Adder Structured with Parentheses
Z <= (A + B) + (C + D);
Using Design Knowledge

In many circumstances you can improve the quality of synthesized circuits by describing your high-level knowledge of a circuit better. VHDL Compiler cannot always derive details of a circuit architecture. Any additional architectural information that you can provide to VHDL Compiler can result in a more efficient circuit.

Optimizing Arithmetic Expressions

Design Compiler uses the properties of arithmetic operators (such as the associative and commutative properties of addition) to rearrange an expression so that it results in an optimized implementation. You can also use arithmetic properties to control the choice of implementation for an expression. Three forms of arithmetic optimization are discussed in this section: merging cascaded adders with a carry, arranging expression trees for minimum delay, and sharing common subexpressions.
Merging Cascaded Adders with a Carry

If your design has two cascaded adders, and one has a bit input, VHDL Compiler replaces the two adders with a simple adder that has a carry input. Example 10-6 shows two expressions where \( \text{cin} \) is a bit variable connected to a carry input. Each expression results in the same implementation.

Example 10-6  Cascaded Adders with Carry Input

\[
\begin{align*}
    z & \leq a + b + \text{cin}; \\
    t & \leq a + b; \\
    z & \leq t + \text{cin}; \\
    t & \leq a + \text{cin}; \\
    z & \leq t + b;
\end{align*}
\]
Arranging Expression Trees for Minimum Delay

If your goal is to speed up your design, arithmetic optimization can minimize the delay through an expression tree by rearranging the sequence of the operations. Consider the following statement

Example 10–7  Simple Arithmetic Expression

\[ Z \leq A + B + C + D; \]

The parser performs each addition in order, as though parentheses were placed around the expression as follows.

\[ Z \leq ((A + B) + C) + D; \]

and constructs the expression tree shown in Figure 10–1.

Figure 10–1  Default Expression Tree
**Considering Signal Arrival Times**

To determine the minimum delay through an expression tree, Design Compiler considers the arrival times of each signal in the expression. If the arrival times of each signal are the same, the length of the critical path of the expression in Example 10-6 equals three adder delays. The critical path delay can be reduced to two adder delays if you add parentheses to the first statement as shown.

\[ Z <= (A + B) + (C + D); \]

The parser evaluates the expressions in parentheses first and constructs a balanced adder tree, as shown in Figure 10-2.

*Figure 10–2 Balaced Adder Tree (Same Arrival Times for All Signals)*

Suppose signals B, C, and D arrive at the same time, and signal A arrives last. The expression tree that produces the minimum delay is shown in Figure 10-3.

*Figure 10–3 Expression Tree with Minimum Delay (Signal A Arrives Last)*
Using Parentheses

You can use parentheses in expressions to exercise more control over the way expression trees are constructed. Parentheses are regarded as user directives that force an expression tree to use the groupings inside the parentheses. The expression tree cannot be rearranged in such a way that it violates these groupings. If you are not sure about the best expression tree for an arithmetic expression, leave the expression ungrouped. Design Compiler can reconstruct the expression for minimum delay.

To illustrate the effect of parentheses on the construction of an expression tree, consider Example 10-8.

Example 10-8  Parentheses in an Arithmetic Expression

Q <= ((A + (B + C)) + D + E) + F;

The parentheses in the expression in Example 10-7 define the following subexpressions:

1. (B + C)
2. (A + (B + C))
3. ((A + (B + C)) + D + E)

These subexpressions must be preserved in the expression tree. The default expression tree for Example 10-8 is shown in Figure 10-4.
Design Compiler restructures the expression tree in Figure 10–4 to minimize the delay and still preserve the subexpressions dictated by the parentheses. If all signals arrive at the same time, the result is the expression tree shown in Figure 10–5.
Design Compiler automatically optimizes expression trees to produce minimum delay. If you do not want VHDL Compiler to optimize the expressions in your design, enter the following command

```
dc_shell> set_minimize_tree_delay false
```

The `set_minimize_tree_delay` command applies to the current design. The default for the `set_minimize_tree_delay` command is `true`.

**Considering Overflow Characteristics**

When Design Compiler performs arithmetic optimization, it considers how to handle the overflow from carry bits during addition. The optimized structure of an expression tree is affected by the bit-widths you declare for storing intermediate results. For example, suppose you write an expression that adds two 4-bit numbers and stores the result in a 4-bit register. If the result of the addition overflows the 4-bit output, the most significant bits are truncated. Example 10–9 shows how VHDL Compiler handles overflow characteristics.

**Example 10–9 Adding Numbers of Different Bit Widths**

```
t <= a + b;  // a and b are 4-bit numbers
z <= t + c;  // c is a 6-bit number
```

In Example 10–9, three variables are added \((a + b + c)\). A temporary variable, \(t\), holds the intermediate result of \(a + b\). Suppose \(t\) is declared as a 4-bit variable, so the overflow bits from the addition of \(a + b\) are truncated. The parser determines the default structure of the expression tree, which is shown in Figure 10–6.
Now suppose the addition is performed without a temporary variable \( z = a + b + c \). VHDL Compiler determines that five bits are needed to store the intermediate result of the addition, so no overflow condition exists. The results of the final addition may be different from the first case, where a 4-bit temporary variable is declared that truncates the result of the intermediate addition. Therefore, these two expression trees do not always yield the same result. The expression tree for the second case is shown in Figure 10–7.
Now suppose the expression tree is optimized for delay, and that signal \( a \) arrives late. The tree is restructured so that \( b \) and \( c \) are added first. Since \( c \) is declared as a 6-bit number, Design Compiler determines that the intermediate result must be stored in a 6-bit variable. The expression tree for this case, where signal \( a \) arrives late, is shown in Figure 10–8. This tree also gives different results than the expression tree in Figure 10–6.

Figure 10–8  Expression Tree for Late-Arriving Signal \( a \)

\[
\begin{align*}
\text{b}[4] & \quad \text{c}[6] \\
+ & \quad \text{a}[4] \\
\text{[6]} & \quad \text{+} \\
\text{z}[6]
\end{align*}
\]

Sharing Common Subexpressions

Subexpressions consist of two or more variables in an expression. If the same subexpression appears in more than one equation, you may want to share these operations to reduce the area of your circuit. You can force common subexpressions to be shared by declaring a temporary variable to store the subexpression, then use the temporary variable wherever you want to repeat the subexpression. Example 10–10 shows a group of simple additions that use the common subexpression \((a + b)\).
**Example 10–10 Simple Additions with a Common Subexpression**

temp <= a + b;
x <= temp;
y <= temp + c;

Instead of manually forcing common subexpressions to be shared, you can let Design Compiler automatically determine whether sharing common subexpressions improves your circuit. You do not need to declare a temporary variable to hold the common subexpression in this case.

In some cases, sharing common subexpressions results in more adders being built. Consider Example 10–11, where $A + B$ is a common subexpression.

**Example 10–11 Sharing Common Subexpressions—Increases Area**

```vhdl
if cond1
  Y <= A + B;
else
  Y <= C + D;
end;
if cond2
  Z <= E + F;
else
  Z <= A + B;
end;
```

If the common subexpression $A + B$ is shared, three adders are needed to implement this section of code.

- $(A + B)$
- $(C + D)$
- $(E + F)$

If the common subexpression is not shared, only two adders are needed: one to implement the additions $A + B$ and $C + D$, and one to implement the additions $E + F$ and $A + B$. 
Design Compiler analyzes common subexpressions during the resource sharing phase of the `compile` command, and considers area costs and timing characteristics. To turn off the sharing of common subexpressions for the current design, enter the following command

```
dc_shell> set_share_cse false
```

The default is `true`.

VHDL Compiler's parser does not identify common subexpressions unless you use parentheses or write them in the same order. For example, the following two equations use the common subexpression `A + B`.

**Example 10–12 Unidentified Common Subexpressions**

```
Y <= A + B + C;
Z <= D + A + B;
```

The parser does not recognize `A + B` as a common subexpression since the second equation is parsed as `(D + A) + B`. You can force the parser to recognize the common subexpression by rewriting the second assignment statement as

```
Z <= A + B + D;
```

or

```
Z <= D + (A + B);
```

**Note:**

You do not have to rewrite the assignment statement since Design Compiler recognizes common subexpressions automatically.
Operator Bit Width

For example, the adder in Example 10–13 sums the eight-bit value of A (a BYTE) with the eight-bit value of TEMP. TEMP’s value is either B, which is used only when it is less than 16, or C, which is a four-bit value (a NIBBLE). Therefore, the upper four bits of TEMP are always 0. VHDL Compiler cannot derive this fact, since TEMP is declared with type BYTE. You can simplify the synthesized circuit by changing the declared type of TEMP to NIBBLE (a four-bit value). With this modification, half adders are required to implement the top four bits of the adder circuit, rather than full adders.
Example 10–13 Function with One Adder

function ADD_IT_16 (A, B: BYTE; C: NIBBLE) return BYTE is
    variable TEMP: BYTE;
begin
    if B < 16 then
        TEMP := B;
    else
        TEMP := C;
    end if;
    return A + TEMP;
end;

Example 10–14 shows how this change in TEMP’s declaration can yield a significant savings in circuit area.
Example 10–14 Using Design Knowledge to Simplify an Adder

```vhdl
function ADD_IT_16 (A, B: BYTE; C: NIBBLE)
  return BYTE is
  variable TEMP: NIBBLE;   -- Now only 4 bits
begin
  if B < 16 then
    TEMP := NIBBLE(B);     -- Cast BYTE to NIBBLE
  else
    TEMP := C;
  end if;

  return A + TEMP;         -- Single adder
end;
```

For further assistance, email support_center@synopsys.com or call your local support center.
Using State Information

You can also apply design knowledge in sequential designs. Often you can make strong assertions about the value of a signal in a particular state of a finite state machine. You can describe this information to Design Compiler.

Example 10–15 shows the VHDL description of a simple state machine that uses two processes.

Example 10–15  A Simple State Machine

package STATES is
    type STATE_TYPE is (SET0, HOLD0, SET1);
end STATES;

use work.STATES.all;

entity MACHINE is
    port(X, CLOCK: in BIT;
        CURRENT_STATE: buffer STATE_TYPE;
        Z: buffer BIT);
end MACHINE;

architecture BEHAVIOR of MACHINE is
    signal NEXT_STATE: STATE_TYPE;
    signal PREVIOUS_Z: BIT;

begin
    -- Process to hold combinational logic.
    COMBIN: process(CURRENT_STATE, X, PREVIOUS_Z)
    begin
        case CURRENT_STATE is
            when SET0 =>
                Z <= '0';                 -- Set Z to '0'
                NEXT_STATE <= HOLD0;
            when HOLD0 =>
                Z <= PREVIOUS_Z;          -- Hold value of Z
                if X = '0' then
                    NEXT_STATE <= HOLD0;
                else
                    NEXT_STATE <= SET1;
                end if;
            when others =>
                Z <= CURRENT_STATE;
        end case;
    end process;

    -- Process to hold combinational logic.
    NEXT_STATE: process(CURRENT_STATE, X, PREVIOUS_Z)
    begin
        case CURRENT_STATE is
            when SET0 =>
                Z <= '0';                 -- Set Z to '0'
                NEXT_STATE <= HOLD0;
            when HOLD0 =>
                Z <= PREVIOUS_Z;          -- Hold value of Z
                if X = '0' then
                    NEXT_STATE <= HOLD0;
                else
                    NEXT_STATE <= SET1;
                end if;
            when others =>
                Z <= CURRENT_STATE;
        end case;
    end process;

    end BEHAVIOR;
when SET1 => -- Set Z to '1'
    Z <= '1';
    NEXT_STATE <= SET0;
end case;
end process COMBIN;

-- Process to hold synchronous elements (flip-flops).
SYNCH: process
begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
    PREVIOUS_Z <= Z;
end process SYNCH;
end BEHAVIOR;
In the state \texttt{HOLD1}, output \texttt{Z} retains its value from the previous state. To accomplish this, a flip-flop is inserted to hold \texttt{PREVIOUS_Z}. However, you can make some assertions about the value of \texttt{Z}. In state \texttt{HOLD0}, the value of \texttt{Z} is always 0. You can deduce this from the fact that state \texttt{HOLD0} is entered only from state \texttt{SET0}, where \texttt{Z} is always assigned '0'.

Example 10–16 shows how you can change the VHDL description to use this assertion, with a resulting simpler circuit.

\textit{Example 10–16 A Better Implementation of a State Machine}

```
package STATES is  
    type STATE_TYPE is (SET0, HOLD0, SET1);  
end STATES;

use work.STATES.all;

entity MACHINE is  
    port(X, CLOCK: in BIT;  
         CURRENT_STATE: buffer STATE_TYPE;  
         Z: buffer BIT);  
end MACHINE;

architecture BEHAVIOR of MACHINE is  
    signal NEXT_STATE: STATE_TYPE;

    begin
        -- Combinational logic.
        COMBIN: process(CURRENT_STATE, X)
        begin
            case CURRENT_STATE is
                when SET0 =>  
                    Z <= '0';  
                    NEXT_STATE <= HOLD0;
                when HOLD0 =>  
                    Z <= '0';  
                    if X = '0' then
                        NEXT_STATE <= HOLD0;
                    else
                        NEXT_STATE <= SET1;
                    end if;
```
when SET1 =>                -- Set Z to '1'
  Z <= '1';
  NEXT_STATE <= SET0;
end case;
end process COMBIN;

-- Process to hold synchronous elements (flip-flops)
SYNCH: process
begin
  wait until CLOCK'event and CLOCK = '1';
  CURRENT_STATE <= NEXT_STATE;
end process SYNCH;
end BEHAVIOR;
Propagating Constants

Constant propagation is the compile-time evaluation of expressions containing constants. VHDL Compiler uses constant propagation to reduce the amount of hardware required to implement operators. For example, a "+" operator with a constant 1 as one of its arguments causes an incrementer to be built, rather than a general adder. If both arguments of "+" or any other operator are constants, no hardware is constructed, since the expression's value is calculated by VHDL Compiler and inserted directly in the circuit.

Other operators that benefit from constant propagation include comparators and shifters. Shifting a vector by a constant amount requires no logic to implement; it requires only a reshuffling (rewiring) of bits.

Sharing Complex Operators

The efficiency of a synthesized design depends primarily on how you describe its component structure. The optimization of individual components, especially those made from random logic, produces similar results from two very different descriptions. Therefore, concentrate the majority of your design effort on the implied component hierarchy (as discussed in the preceding sections), rather than on the logical descriptions. Chapter 3 discusses how to define a VHDL design hierarchy. The next section describes how to use Design Compiler effectively with a hierarchical design.

VHDL Compiler supports many shorthand VHDL expressions. There is no benefit with a verbose syntax when a shorter description is adequate. Example 10-17 shows four equivalent groups of statements.
Example 10–17 Equivalent Statements

signal A, B, C: BIT_VECTOR(3 downto 0);
...
C <= A and B;

---------------------------------------------------------------------
C(3 downto 0) <= A(3 downto 0) and B(3 downto 0);
---------------------------------------------------------------------
C(3) <= A(3) and B(3);
C(2) <= A(2) and B(2);
C(1) <= A(1) and B(1);
C(0) <= A(0) and B(0);
---------------------------------------------------------------------
for I in 3 downto 0 loop
  C(I) <= A(I) and B(I);
end loop;

Design Compiler Optimization

After a VHDL description is translated by VHDL Compiler, you optimize and synthesize the design by using Design Compiler.

Chapter 12 describes how to use Design Compiler to read VHDL descriptions through VHDL Compiler. For a complete description of the Design Compiler compile command, see the Design Compiler Family Reference Manual. For the syntax of Design Compiler commands, see the Synopsys man pages.

The Design Compiler commands set_flatten and set_structure set flatten and structure attributes for the compiler. Flattening reduces a design’s logical structure to a set of two-level (AND/OR) logic equations. Structuring attempts to find common factors in the translated design’s set of logic equations.
Flattening Designs

When a design is flattened, the original structure of its VHDL description is lost. This condition is useful when a description is written at a high level without regard to use of constructs or resource allocation. Random control logic often falls in this category. In general, flattening consolidates logic and often speeds up the final implementation. However, not all logic can be flattened. For example, large adders, exclusive-OR networks, and comparators of two variables cannot be flattened.

Do not flatten the design if you build structure into the VHDL description by resource allocation or by user-defined operators (such as carry-lookahead adders). Flattening a highly structured design can increase its circuit area considerably. You can still use structuring to improve the design’s existing logical structure without removing that structure.

Grouping Logic

You may find it necessary to regroup logic in a VHDL description to achieve an expected optimization. The basic unit used by Design Compiler for optimization is a design (VHDL entity). Entities in a VHDL description have a one-to-one correspondence with designs in Design Compiler. All constraints (see Chapter 11) and compile directives are applied at the design level. To optimize two pieces of logic differently, you must define them as separate designs or entities. Subprograms and operators that occur within a VHDL entity are grouped with that entity for optimization.

You can manually create designs in Design Compiler from VHDL subprograms, processes, and blocks by using the `group -hdl_block` command, as described in Chapter 12.
If you have a function that includes carefully structured logic, separate it from logic that can be flattened. For example, separate a specialized function, such as a carry-lookahead adder, from its associated control logic. This separation is useful because control logic probably benefits from flattening, although an adder does not.

**Asynchronous Designs**

In a synchronous design, all flip-flops use a single clock that is a primary input to the design, and there are no combinational feedback paths. Synchronous designs perform the same function regardless of the clock rate, so long as all signals can propagate through the design’s combinational logic during the clock’s cycle time.

Design Compiler currently assumes that all designs are synchronous. Design Compiler can therefore change the timing behavior of the combinational logic, so long as the maximum and minimum delay requirements are met.

Design Compiler always preserves the Boolean function computed by logic, assuming that the clock arrives after all signals have propagated. Its built-in timing verifier helps determine the slowest path (critical path) through the logic, which determines how fast the clock can run.

Design Compiler provides some support for asynchronous designs. However, you must assume a greater responsibility for the accuracy of your circuits. Although fully synchronous circuits usually agree with their simulation models, asynchronous circuits may not.

Design Compiler may not warn you when a design is not fully synchronous. You must be aware of possible asynchronous timing problems.
The most common way to produce asynchronous logic in VHDL is to use gated clocks on latches or flip-flops.

Example 10–18 shows a fully synchronous design, a counter with synchronous \texttt{ENABLE} and \texttt{RESET} inputs. Since it is synchronous, this counter works if the clock speed is slower than the critical path.

\textit{Example 10–18 Fully Synchronous Counter with Reset and Enable}

\begin{verbatim}
entity COUNT is 
  port(RESET, ENABLE, CLK: in BIT;
       Z: buffer INTEGER range 0 to 7);
end;

architecture ARCH of COUNT is 
begin 
  process(RESET, ENABLE, CLK, Z) 
  begin 
    if (CLK’event and CLK = ’1’) then 
      if (RESET = ’1’) then -- occurs on clock edge 
        Z <= 0;
      elsif (ENABLE = ’1’) then -- occurs on clock edge 
        if (Z = 7) then 
          Z <= 0;
        else 
          Z <= Z + 1;
        end if;
      end if;
    end if;
  end process;
end;
\end{verbatim}
Example 10-19 shows an asynchronous version of Example 10-18’s design that uses two common asynchronous design techniques. The first technique is enabling the counter by ANDing the clock and enable signals. The second technique uses an asynchronous reset. These techniques work only when the proper timing relationships exist between the reset signal (RESET) and the clock signal (CLK), and there are no glitches in these signals.
**Example 10–19 Design with Gated Clock and Asynchronous Reset**

entity COUNT is
  port(RESET, ENABLE, CLK: in BIT;
       Z: buffer INTEGER range 0 to 7);
end;

architecture ARCH of COUNT is
  signal GATED_CLK: BIT;
begin
  GATED_CLK <= CLK and ENABLE; -- clock gated by ENABLE

  process(RESET, ENABLE, GATED_CLK, Z)
  begin
    if (RESET = '1') then  -- asynchronous reset
      Z <= 0;
    elsif (GATED_CLK'event and GATED_CLK = '1') then
      if (Z = 7) then
        Z <= 0;
      else
        Z <= Z + 1;
      end if;
    end if;
  end process;
end;
Example 10–20 shows an asynchronous design that may not work, since Design Compiler does not guarantee that the combinational logic it builds has no hazards (glitches).

Example 10–20 Dangerous Design (Counter with Asynchronous Load)

entity COUNT is
  port(LOAD_ENABLE, CLK: in BIT;
       LOAD_DATA: in INTEGER range 0 to 7;
       Z: buffer INTEGER range 0 to 7);
end;

architecture ARCH of COUNT is
begin
  process(LOAD_ENABLE, LOAD_DATA, CLK, Z)
  begin
    if (LOAD_ENABLE = '1') then
      Z <= LOAD_DATA;
    elsif (CLK'event and CLK = '1') then
      if (Z = 7) then
        Z <= 0;
      else
        Z <= Z + 1;
      end if;
    end if;
  end process;
end;
Example 10-20’s design works only when the logic driving the preset and clear pins of the flip-flops that hold $Z$ is faster than the clock speed. If you must use this design style, you must simulate the synthesized circuit thoroughly. You also need to inspect the synthesized logic, since potential glitches may not appear in simulation. For a safer design, use a synchronous LOAD_ENABLE.
A design synthesized with complex logic driving the gate of a latch rarely works. Example 10-21 shows an asynchronous design that never works.

**Example 10-21 Incorrect Asynchronous Design with Gated Clock**

```vhdl
entity COMP is
  port(A, B: in INTEGER range 0 to 7;
       Z:    buffer INTEGER range 0 to 7);
end;

architecture ARCH of COMP is
begin
  process(A, B)
  begin
    if (A = B) then
      Z <= A;
    end if;
  end process;
end;
```

![Circuit Diagram](image)
In Example 10-21, the comparator’s output latches the value A onto the value Z. This design may work under behavioral simulation, where the comparison happens instantly. However, the hardware comparator generates glitches that cause the latches to store new data when they should not.

**Don’t-Care Inference**

You can greatly reduce circuit area by using don’t-cares. To use a don’t-care in your design, create an enumerated type for the don’t-care (the standard VHDL BIT type does not include don’t-care value).

Don’t-cares are best used as default assignments to variables. You can assign a don’t-care value to a variable at the beginning of a process, in the default section of a case statement, or in the else section of an if statement.

Example 10-22 shows a use of don’t-care encoding for a seven-segment LED decoder. Enumeration encoding ‘D’ represents the don’t-care state.
Example 10-22 Using Don’t-Care Type for Seven-Segment LED Decoder

package P is
  type MULTI is ('0', '1', 'D', 'Z');
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of MULTI : type is "0 1 D Z";
  type MULTI_VECTOR is array (INTEGER range <> ) of MULTI;
end P;

use work.P.all;

entity CONVERTER is
  port(BCD: in MULTI_VECTOR(3 downto 0);
       LED: out MULTI_VECTOR(6 downto 0));
  -- pragma dc_script_begin
  -- set_flatten true
  -- pragma dc_script_end
end CONVERTER;

architecture BEHAVIORAL of CONVERTER is
begin
  CONV: process(BCD)
  begin
    case BCD is
    when "0000" => LED <= "1111110";
    when "0001" => LED <= "1100000";
    when "0010" => LED <= "1011011";
    when "0011" => LED <= "1110011";
    when "0100" => LED <= "1100101";
    when "0101" => LED <= "0110111";
    when "0110" => LED <= "0111111";
    when "0111" => LED <= "1100010";
    when "1000" => LED <= "1111111";
    when "1001" => LED <= "1110111";
    when others => LED <= "DDDDDDD";
    end case;
  end process CONV;
end BEHAVIORAL;
Example 10–23 shows the seven-segment decoder used in Example 10–22, where the default assignment to LED is 0 instead of don’t-care. Note the larger gate count in the circuit without don’t-cares.
Example 10–23 Seven-Segment Decoder without Don’t-Care Type

entity CONVERTER is
    port (BCD: in BIT_VECTOR(3 downto 0);
         LED: out BIT_VECTOR(6 downto 0));
    -- pragma dc_script_begin
    -- set_flatten true
    -- pragma dc_script_end
end CONVERTER;

architecture BEHAVIORAL of CONVERTER is
begin
    CONV: process(BCD)
    begin
        case BCD is
        when "0000" => LED <= "1111110";
        when "0001" => LED <= "1100000";
        when "0010" => LED <= "1011011";
        when "0011" => LED <= "1110011";
        when "0100" => LED <= "1100101";
        when "0101" => LED <= "0110111";
        when "0110" => LED <= "0111111";
        when "0111" => LED <= "1100010";
        when "1000" => LED <= "1111111";
        when "1001" => LED <= "1110111";
        when others => LED <= "0000000";
        end case;
    end process CONV;
end BEHAVIORAL;
Example 10–23 (continued) Seven-Segment Decoder without Don’t-Care Type
Using Don’t-Cares

You do not always want to assign a default value of don’t-care, although some cases exist where it can be beneficial, as shown in the seven-segment decoder in Example 10–23.

The reasons for not always defaulting to don’t-care are these:

- Their potential for mismatches between simulation and synthesis is greater.
- Defaults for variables can hide mistakes in the VHDL code.

For example, if you assign a default don’t-care value to \( \text{VAR} \), and later assign a value to \( \text{VAR} \), expecting \( \text{VAR} \) to be a don’t-care, you may have overlooked an intervening condition under which \( \text{VAR} \) is assigned.

Therefore, when you assign a value to a variable (or signal) containing a don’t-care, make sure that the variable (or signal) is really a don’t-care under those conditions.

Differences Between Simulation and Synthesis

Don’t-cares are treated differently in simulation than they are in synthesis, and there may be a mismatch between the two. To a simulator, a don’t-care is a distinct value, different from a 1 or a 0. In synthesis, however, a don’t-care becomes a 0 or a 1 (and the hardware built treats it as either a 0 or a 1).

Whenever a comparison is made to a variable whose value is don’t-care, simulation and synthesis can differ. The safest way to use don’t-cares is to

- Assign don’t-care values only to output ports.
- Make sure the design never reads output ports.
These guidelines guarantee that when you are simulating in the scope of the design itself, the only difference between simulation and synthesis occurs when the simulator says that an output is a don’t-care.

**WARNING**

If you use don’t-cares internally to a design, expressions comparing to don’t-cares (‘D’) are synthesized as though values are not equal to ‘D’.

For example

```vhdl
if X = ‘D’ then
...
```

is synthesized as

```vhdl
if FALSE then
```

If you use expressions comparing values to ‘D’, there may be a difference between pre- and post-synthesis simulation results. For this reason, VHDL Compiler issues a warning when it synthesizes such comparisons.

Warning: A partial don’t-care value was read in routine test line 24 in file ‘test.vhdl’ This may cause simulation to disagree with synthesis. (HDL-171)

To take advantage of don’t-cares during synthesis, use the Design Compiler command `set_flatten`. To embed this command in your description, see “Embedded Constraints and Attributes” in Chapter 11.

In most cases don’t-cares are used in control-oriented logic, such as state machines. For these circuits, flattening generally does not cause a problem.
Synthesis Issues

Feedback paths and latches result from ambiguities in signal or variable assignments and language supersets, or the differences between a VHDL simulator view and Synopsys’ use of VHDL.

Feedback Paths and Latches

Under certain circumstances, normal-looking VHDL can imply combinational feedback paths or latches in synthesized logic. This happens when a signal or variable in a combinational process (one without a \texttt{wait} or \texttt{if signal’event} statement) is not fully specified. A variable or signal is fully specified when it is assigned under all possible conditions. A variable or signal is not fully specified when a condition exists under which the variable is not assigned.

Fully Specified Variables

Example 10–24 shows several variables. A, B, and C are fully specified, although X is not.

\textit{Example 10–24  Fully Specified Variables}

\begin{verbatim}
process (COND1)
  variable A, B, C, X : BIT;
begin
  A := ’0’     -- A is hereby fully specified
  C := ’0’     -- C is hereby fully specified

  if (COND1) then
    B := ’1’;    -- B is assigned when COND1 is TRUE
    C := ’1’;    -- C is already fully specified
    X := ’1’;    -- X is assigned when COND1 is TRUE
  else
    B := ’0’;    -- B is assigned when COND1 is FALSE
  end if;
\end{verbatim}
-- A is assigned regardless of COND1, so A is fully
-- specified.

-- B is assigned under all branches of if (COND1),
-- that is, both when COND1 is TRUE and when
-- COND1 is FALSE, so B is fully specified.

-- C is assigned regardless of COND1, so C is fully
-- specified. (The second assignment to C does
-- not change this.)

-- X is not assigned under all branches of
-- if (COND1), namely, when COND1 is FALSE,
-- so X is not fully specified.
end process;

The conditions of each if and elsif are considered independent in Example 10–24. A is considered not fully specified in the following fragment:

if (COND1) then
  A <= '1';
end if;
if (not COND1) then
  A <= '0';
end if;

A variable or signal that is not fully specified in a combina-
tional process is considered conditionally specified. In this
case a flow-through latch is implied. You can conditionally
assign a variable, but you cannot read a conditionally speci-

A variable or signal that is not fully specified in a combina-
tional process is considered conditionally specified. In this
case a flow-through latch is implied. You can conditionally
assign a variable, but you cannot read a conditionally speci-

A variable or signal that is not fully specified in a combina-
tional process is considered conditionally specified. In this
case a flow-through latch is implied. You can conditionally
assign a variable, but you cannot read a conditionally speci-
fied variable. You can, however, both conditionally assign
and read a signal.
If a fully specified variable is read before its assignment statements, combinational feedback may exist. For example, the following fragment synthesizes combinational feedback for VAL.

```vhdl
process(NEW, LOAD)
  variable VAL: BIT;
begin
  if (LOAD) then
    VAL := NEW;
  else
    VAL := VAL;
  end if;

  VAL_OUT <= VAL;
end process;
```

In a combinational process, you can ensure that a variable or signal is fully specified by providing an initial (default) assignment to the variable at the beginning of the process. This default assignment assures that the variable is always assigned a value, regardless of conditions. Subsequent assignment statements can override the default. A default assignment is made to variables A and C in Example 10–24.

Another way to make sure that you are not implying combinational feedback is to use a sequential process (one with a wait or if signal’event statement). In this case, variables and signals are registered. The registers break the combinational feedback loop.

See “Register Inference” in Chapter 8 for more information about sequential processes and the conditions under which registers and latches are inferred by VHDL Compiler.
Asynchronous Behavior

Some forms of asynchronous behavior are not supported. For example, a circuit description of a one-shot signal generator of the form

\[ X <= A \text{ nand } \neg \neg \neg A; \]

that you might expect to generate three inverters (an inverting delay line) and a NAND gate, is optimized to

\[ X <= A \text{ nand } \neg A; \]

then

\[ X <= 1; \]

Superset Issues and Error Checking

The Synopsys VHDL Analyzer is a full IEEE 1076 VHDL analyzer, described in the and the .

When Design Compiler Version 3.0 reads in a VHDL design, it first calls the Synopsys VHDL Analyzer to check the VHDL source for errors, then calls VHDL Compiler to translate the VHDL source to an intermediate form for synthesis. If an error is in VHDL source, you get a VHDL Analyzer message and possibly a VHDL Compiler message.

VHDL Compiler allows globally static objects where only locally static objects should be allowed, without issuing an error message. However, the Synopsys VSS Expert and VSS Professional tools detect and flag this error.